Bergische Universität
Wuppertal

C-XSC-Langzahlarithmetiken für reelle und komplexe Intervalle basierend auf den Bibliotheken MPFR und MPFI

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Püttlingen und Wuppertal, den 11. Mai 2011

Preprint
BUW-WRSWT 2011/1

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

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

Internet-Zugriff

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

`http://www2.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

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`

Inhaltsverzeichnis

1 Einleitung

C-XSC [24, 26, 27, 49] ist eine auf **C++**basierende Programmierumgebung, mit deren selbstverifizierenden numerischen Algorithmen die exakten mathematischen Lösungen von z.B. linearen oder nichtlinearen Gleichungssystemen, Integralen, Ableitungen von praktisch beliebig komplizierten Ausdrücken, Differential- und Integralgleichungen, oder Nullstellen von differenzierbaren Funktionen, ... durch Intervalle, Intervallvektoren Funktionsschläuche, etc. nahezu optimal eingeschlossen werden können. In C-XSC sind optimale/semimorphe (siehe [36]) reelle und komplexe Punkt- und Intervall-Arithmetiken auch für Vektoren und Matrizen über Gleitkommazahlen implementiert.

Alle Rechnungen basieren auf dem IEEE-double-Standard, so dass eine Mantisse von etwa 16 Dezimalstellen und ein Zehnerexponentenbereich von −324 bis +308 zur Vefügung stehen. Rechnungen in höherer Präzision können in C-XSC 2.4.0 mit Hilfe der beiden Datentypen `l_interval` und `lx_interval` durchgeführt werden, die beide auf einem Staggered-Correction-Format (Staggered-Format) basieren, so dass jeweils eine maximale Präzision von etwa 324 + 308 = 632 Dezimalstellen benutzt werden kann.

Im Gegensatz zum `l_interval`-Typ, der den obigen Exponentenbereich von −324 bis +308 besitzt, steht mit dem `lx_interval`-Typ der sehr große Exponentenbereich von −2711437152599603 bis +2711437152599603 zur Verfügung, so dass hier Überlauf- oder Unterlaufprobleme praktisch keine Rolle mehr spielen. Auch die Auswertung der komplexwertigen Elementarfunktionen ist in beiden Staggered-Formaten möglich.

Der Vorteil des Staggered-Formats besteht nun darin, dass unabhängig von der gewählten Präzision die vier Grundoperationen $\{+, -, *, /\}$ hochgenau und beliebig lange Skalarprodukte mit Hilfe des langen Akkumulators sogar maximalgenau ausgewertet werden können [3, 36]. Da dieser Akkumulator derzeit hardwaremäßig nicht unterstützt wird, muss er softwaremäßig simuliert werden, was zu langen Laufzeiten führt. Ein weiterer Nachteil der Staggered-Arithmetik besteht darin, dass bei zu breiten Eingangsintervallen die Ergebnisse nur noch in der Präzision des *double*-Formats, d.h. also mit nur etwa 16 Dezimalstellen, berechnet werden können.

Die geschilderten Nachteile des Staggered-Formats lassen sich jedoch vermeiden, wenn man die in Frankreich ab entwickelten MPFR- [18] und MPFI-Bibliotheken [45] benutzt, die von Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann bzw. Nathalie Revol, Fabrice Rouillier, Sylvain Chevillard, Hong Diep NGUYEN und Christoph Lauter entwickelt wurden. Die MPFR-Bibliothek bietet eine Langzahl-Punktarithmetik, wobei die Grundoperationen und die zahlreichen Elementarfunktionen mit den folgenden Rundungsoptionen maximalgenau ausgewertet werden können:

- `MPFR_RNDN` Rundung zur nächsten Rasterzahl des Langzahlformats
- `MPFR_RNDD` Abrundung zur nächstkleineren Rasterzahl des Langzahlformats
- `MPFR_RNDU` Aufrundung zur nächstgrößeren Rasterzahl des Langzahlformats
- `MPFR_RNDZ` Rundung zur Null im gewählten Langzahlformat
- `MPFR_RNDA` Rundung weg von der Null im gewählten Langzahlformat

Mit Hilfe der beiden Rundungsoptionen `MPFR_RNDD` und `MPFR_RNDU` können damit z.B. optimale Einschließungen exakter Funktionswerte der implementierten MPFR-Elementarfunktionen sehr effektiv berechnet werden.

In der MPFI-Bibliothek sind neben den Intervall-Grundoperationen $\{\oplus, \ominus, \odot, \oslash\}$ fast alle Elementarfunktionen implementiert, mit denen zu beliebig breiten Eingangsintervallen maximalgenaue Einschließungen ihrer Funktionswerte berechnet werden können. Dabei werden Ober- und Unterschranken dieser Einschließungen in der vorgegebenen Präzision maximalgenau berechnet. Die aktuelle MPFI-Version ist 1.5. Um sie nutzen zu können, müssen die GMP-Bibliothek (Version 4.1 oder höher) und die MPFR-Bibliothek (Version 3.0.0 oder höher) vorhanden sein. Da die Bibliothek auf der MPFR-Bibliothek basiert, profitiert MPFI von den oben beschriebenen korrekten Rundungen der MPFR-Bibliothek. Die Bibliothek befolgt den IEEE-754-Standard der Gleitkomma-Arithmetik.

Da die MPFR- und MPFI-Bibliotheken auf einer Integer-Arithmetik basieren, können die entsprechenden Hardware-Ressourcen direkt angesprochen werden, was im Vergleich zur Staggered-Arithmetik [32, 39, 12, 13] die Laufzeit erheblich reduziert.

Unter C-XSC lassen sich die beiden Bibliotheken mit Hilfe zweier Interfaces (**C++**-Wrapper-Klassen) benutzen, die beide von Hans-Stephan Brand im Jahre 2010 im Rahmen einer Bachelor-Arbeit [14] entworfen und prototypmäßig realisiert wurden. Die MPFR- und MPFI-Bibliotheken kommen damit unter C-XSC mit den dort definierten Sprachelementen sehr einfach zur Anwendung. Insbesondere die MPFR-Bibliothek liefert zusätzliche Funktionen, wie z.B. $\Gamma(x)$, digamma(x), erf(x), erfc(x), $\zeta(x)$, $\zeta(n)$, $\mathrm{J}_n(x)$, $\mathrm{Y}_n(x)$, wobei $\mathrm{J}_n(x)$ und $\mathrm{Y}_n(x)$ die Besselfunktionen der ersten und zweiten Art sind, mit $n = 0, 1, 2, \ldots$. Die streng monotonen Funktionen erf(x), erfc(x), digamma(x) und $\Gamma'(x)$ lassen sich mit den beschriebebenen Rundungsoptionen in C-XSC zusätzlich auch als Intervallfunktionen sehr einfach implementieren. Für komplexe Punkt- und Intervall-Rechnungen werden für C-XSC entsprechende **C++**-Wrapper-Klassen entwickelt.

Die **C++**-Wrapper-Klassen werden so implementiert, dass sie nicht zusammen mit den in C-XSC schon integrierten Staggered-Formaten genutzt werden können. Für Vergleichsrechnungen mag dies ein gewisser Nachteil sein, dafür bleibt der Programmieraufwand überschaubar.

2 Installation

Die Installation der MPFR- und MPFI-Bibliotheken ist nur unter einem Linux/Unix-System möglich. Dazu müssen unter OpenSuse 11.2 zunächst mit Yast2 u.a. installiert sein:

- `gcc44-c` 4.4.1_20090817-2.3.4 Der GNU **C++**-Compiler
- `gmp-devel` 4.3.1-2.2 Include Files and Libraries for the GNU MP Library
- `libgmp3` 4.3.1-2.2 Shared library for the GNU MP Library
- `libgmpxx4` 4.3.1-2.2 **C++**bindings for the GNU MP Library
- `make` 3.81-130.2 GNU make
- `texlive 2008-13.18.1` und `texinfo 4.13a-3.2`

Weitere Informationen zur GNU MP Library findet man unter

`http://gmplib.org/`

2.1 Installation der MPFR- und MPFI-Bibliotheken

Die Installation der aktuellen MPFR- und MPFI-Bibliotheken erfolgt mit Hilfe der Dateien

`mpfr-3.0.0.tar.gz` `mpfi-1.5.tar.gz`

die aus dem Netz jeweils unter

`http://www.mpfr.org/mpfr-current/`

`https://gforge.inria.fr/frs/?group_id=157`

bezogen werden können. Zuerst ist die MPFR-Bibliothek zu installieren. Die MPFI-Bibliothek wird danach ganz analog wie folgt installiert:

- `mpfi-1.5.tar.gz` ins Hauptverzeichnis kopieren und dort entpacken, wodurch das Unterverzeichnis `mpfi-1.5` entsteht.
- Wechseln ins Unterverzeichnis `mpfi-1.5`
- `./configure` Konfigurieren der Installation (z.B. Installations-Pfad)
- `make` Compilieren der Pakete
- `make check` Überprüfen der Dateien (nicht zwingend notwendig, aber empfohlen!)
- `make install` Installation der Bibliothek (nur hier als root!)

Die Installation konnte problemlos durchgeführt werden, weitere Informationen findet man in der Text-Datei `~/mpfi-1.5/INSTALL`. Mit `make pdf` oder `make dvi` können zusätzlich die Dokumentationsdateien `mpfi.pdf` bzw. `mpfi.dvi` erzeugt werden. Sollten bei `make check` Fehlermeldungen bez. der dynamischen Bibliothek `libmpfr.so.4` auftreten, so können diese als root mit `ldconfig <enter>` abgestellt werden. Hinweise zu `ldconfig` findet man z.B unter

`http://www.fibel.org/linux/lfo-0.6.0/node386.html`

3 MpfrClass-Interface zur Anbindung der MPFR-Bibliothek an C-XSC

3.1 Grundlegendes

Das MpfrClass-Interface ist eine in `mpfrclass.hpp` und `mpfrclass.cpp` implementierte **C++**-Wrapper-Klasse MpfrClass für die C-Bibliothek MPFR, deren C-Funktionen über die implementierten Operatoren und Funktionen aufgerufen werden. Alle Funktionen mit der möglichen Übergabe eines RoundingModes oder eines PrecisionTypes besitzen als Standard die Werte von CurrRndMode bzw. CurrPrecision, die beide beliebig gesetzt werden können. Dies gilt auch für fast alle Konstruktoren.

3.1.1 Allgemein

Um in C-XSC das Interface verwenden zu können, muss der Header mpfrclass.hpp eingebunden werden. In ihm sind die benötigten Header-Dateien der MPFR-Bibliothek enthalten. Die MpfrClass-Klasse liegt im Namensraum "MPFR".

3.1.2 Aufbau

Die Klasse besteht intern aus einer "mpfr_t"-Variablen. Diese dient zum Speichern des Wertes. Zusätzlich gibt es static Elemente, um den Standard-Rundungsmodus, die Standard-Precision und die aktuelle Basis zu speichern.

3.1.3 Präzision

Die Präzision gibt die Anzahl der binären Mantissenstellen einer MpfrClass-Variablen `x` an. Die Präzision von `x` kann mit `x.SetPrecision(prec)` auf den Wert `prec` ≥ 2 gesetzt werden. Die Current-Precision ist die Präzision, mit der z.B. alle arithmetischen Grundoperationen durchgeführt werden. Sie kann global mit `SetCurrPrecision(prec)` gesetzt werden. Wenn dies nicht geschieht, so wird mit der Default-Precision von 53 Bits gerechnet. Unabhängig davon kann die Präzision für jede MpfrClass-Variable `x` auch einzeln festgelegt werden.

3.1.4 Variablentyp PrecisionType

Mithilfe des Variablentyps `PrecisionType` (Name der Variablen meist prec) kann der Präzisionswert einer MpfrClass-Variablen eingestellt werden. Der Variablentyp ist ein typedef für eine "mp_prec_t"-Variable.

3.1.5 Variablentyp RoundingMode

Mit dem Variablentyp "RoundingMode" (Name der Variablen meist rnd) wird der gewünschte Rundungsmodus eingestellt. Der Variablentyp ist ein typdef für eine `mpfr_rnd_t`-Variable.

- RoundNearest – Der Wert wird zur nächsten Rasterzahl gerundet (0)
- RoundUp – Der Wert wird aufgerundet (2)
- RoundDown – Der Wert wird abgerundet (3)

- RoundToZero – Der Wert wird in Richtung Null gerundet (1)
- RoundFromZero – Rundung weg von der Null (4)

3.1.6 Zahlenformat

Im Zusammenhang mit der Current-Präzision, die mit `SetCurrPrecision(prec)` auf $\texttt{prec} \geq 2$ gesetzt werden kann, soll das dazugehörige `MPFR`-Format im Vergleich zum bekannten `IEEE`-Format genauer untersucht werden.

Wir betrachten zunächst das **IEEE-Format**:

- Kleinste positive (denormalisierte) Zahl: $\texttt{minreal} := 2^{-1074}$.
- $\texttt{succ(minreal)} = 2 \cdot \texttt{minreal} = 2^{-1073}$.
- Kleinste positive normalisierte Zahl: $\texttt{MinReal} := 2^{-1022} = \texttt{minreal} \cdot 2^{+52}$.
- Größte positive normalisierte Zahl: $\texttt{MaxReal} < 2^{+1024}$.
- $\texttt{pred(minreal)} = 0$, $\texttt{succ(minreal)} = 2 \cdot \texttt{minreal} = 2^{-1073}$.
- $\texttt{pred(MinReal)} = \texttt{MinReal} - \texttt{minreal} = (2^{52} - 1) \cdot \texttt{minreal}$.
- $\texttt{succ(MinReal)} = \texttt{MinReal} + \texttt{minreal} = (2^{52} + 1) \cdot \texttt{minreal}$.
- $\texttt{expo(MinReal)} = \texttt{expo(succ(MinReal))} = -1021$.
- $\texttt{expo(minreal)} = -1073 < \texttt{expo(succ(minreal))} = -1072$.

Wir betrachten jetzt das **MPFR-Format** mit der Current-Präzision $\texttt{prec} \geq 2$:

- Kleinste positive Zahl: $\texttt{minfloat(prec)} := 2^{-1073741824}$ ist `prec`-unabhängig!
- $\texttt{mant(minfloat(prec))} = 0.5$, $\texttt{expo(minfloat(prec))} = -1073741823$.
- $\texttt{pred(minfloat())} = 0$, $\texttt{succ(minfloat())} < 2 \cdot \texttt{minfloat()}$.
- $\texttt{expo(succ(minfloat()))} = \texttt{expo(minfloat())} = -1073741823$.
- Ausgehend von $\texttt{minfloat()} = 0.5 \cdot 2^{-1073741823}$ ist die kleinste positive Maschinenzahl `r` mit dem nächst-größeren Exponenten -1073741822 gegeben durch: $\texttt{r} = 2 \cdot \texttt{minfloat()}$.
- Größte positive normalisierte Zahl: $2^{+1073741823} < \texttt{MaxFloat(prec)} < 2^{+1073741824}$.

Vergleicht man mit den Werten des `IEEE`-Formats, so erkennt man, dass das `MPFR`-Format keinen denormalisierten Zahlenbereich besitzt. Für **jeden** Exponenten $\texttt{ex} = \texttt{expo(x)}$ einer Maschinenzahl `x`, die nicht unendlich und kein NaN ist, stehen für die Mantisse von `x` genau $\texttt{prec} \geq 2$ Bits zur Verfügung, so dass es im Gegensatz zum `IEEE`-Format genau 2^{prec} Maschinenzahlen für jeden Exponenten gibt.

Bei möglichen Fehlerbetrachtungen muss also nicht mehr zwischen einem normalisierten und denormalisierten Bereich unterschieden werden. Man muss also nur noch darauf achten, dass Zwischenergebnisse möglichst nicht in den Unterlaufbereich fallen.

Für den Vorgänger einer Maschinenzahl $x = \texttt{m} \cdot 2^{\texttt{ex}}$ gilt mit $\texttt{ex} = \texttt{expo}(x)$ und $\texttt{m} = \texttt{mant}(x)$:

$$\texttt{pred}(x) = \begin{cases} -\texttt{minfloat()}, & \text{falls } x = 0, \\ 0, & \text{falls } x = \texttt{minfloat()}, \\ 2^{\texttt{ex}-1} \cdot (1 - 2^{-\texttt{prec}}), & \text{falls } \texttt{m} = 0.5,\ x > \texttt{minfloat()}, \\ 2^{\texttt{ex}} \cdot (\texttt{m} - 2^{-\texttt{prec}}), & \text{falls } 0.5 < \texttt{m} < 1,\ x > \texttt{minfloat()}. \end{cases} \tag{3.1}$$

Für den Nachfolger einer Maschinenzahl $x = \texttt{m} \cdot 2^{\texttt{ex}}$ gilt die einfachere Darstellung:

$$\texttt{succ}(x) = \begin{cases} \texttt{minfloat()}, & \text{falls } x = 0, \\ 2^{\texttt{ex}} \cdot (\texttt{m} + 2^{-\texttt{prec}}), & \text{falls } 0.5 \leq \texttt{m} < 1. \end{cases} \tag{3.2}$$

3.2 Konstruktoren / Destruktor

3.2.1 Konstruktoren

```
MpfrClass ();
```

Der Default-Konstruktor legt ein neues Element mit der Current-Precision an.
Der Aufruf `MpfrClass y;` initialisiert den Wert: `y` = NaN;

```
MpfrClass (const MpfrClass& op,   RoundingMode rnd, PrecisionType prec);
MpfrClass (const mpfr_t& op,      RoundingMode rnd, PrecisionType prec);
MpfrClass (int op,                RoundingMode rnd, PrecisionType prec);
MpfrClass (const double& op,      RoundingMode rnd, PrecisionType prec);
MpfrClass (const cxsc::real& op,  RoundingMode rnd, PrecisionType prec);
```

Mögliche Konstruktor-Aufrufe sind:

1. `MpfrClass y(op);`
2. `MpfrClass y(op, RoundNearest);`
3. `MpfrClass y(op, RoundDown, 3);`

Zu 1. op wird mit dem Current-Rundungsmodus auf die Current-Precision gerundet.
Zu 2. op wird mit `RoundNearest` auf die Current-Precision gerundet.
Zu 3. op wird auf die (sehr kleine) Precision *prec* = 3 abgerundet.
Der Aufruf `MpfrClass y(op, 3);` führt zu einer Fehlermeldung, da in der Parameterliste der Rundungsmodus fehlt. Die oberen fünf Konstruktoren erlauben also eine sehr flexible Initialisierung von `MpfrClass`-Objekten.

Mit den Deklarationen

```
int k;    double dbl;    real r;
```

liefern die folgenden Konstruktor-Aufrufe

- `MpfrClass y(k  , RoundNearest, 32);` `MpfrClass y(dbl, RoundNearest, 53);`
- `MpfrClass y(r  , RoundNearest, 53);`

`MpfrClass`-Objekte `y` mit den **ungerundeten** Werten von `k, dbl, r`.

Der folgende Aufruf: `MpfrClass y(x, RoundNearest, x.GetPrecision());` mit `x` vom Typ `MpfrClass` liefert ein `MpfrClass`-Objekt `y`, mit `y` = `x`, (Copy-Konstruktor).

```
MpfrClass (const std::string& s, RoundingMode rnd, PrecisionType prec);
```

Der Aufruf `MpfrClass y(s);` rundet `s` mittels der voreingestellten Current-Precision und mit dem Current-Rundungsmodus in das Klassenobjekt `y`. Der Aufruf `MpfrClass y(s, RoundNearest, 140);` rundet `s` mit der Präzision von 140 Bits zur nächsten Rasterzahl dieses Formats.
Achtung: Keine Leerzeichen am String-Ende!

3.2.2 Destruktor

```
~MpfrClass ();
```

Der Speicher für das Objekt wird freigegeben. Dieser Destruktor muss nicht explizit aufgerufen werden!

3.3 Eingabe / Ausgabe

```
friend std::ostream& operator << (std::ostream& os, const MpfrClass& x);
```

Ermöglicht die Ausgabe einer MpfrClass-Variablen x über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(x.GetPrecision()/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Der Wert der Variablen x wird dabei in der mit `SetBase(k)` voreingestellten Basis ausgegeben. Für die meist dezimale Ausgabe ist `k` = 10 zu wählen.

```
std::ostream& operator << (std::ostream& os, mpfr_t& x);
```

Ermöglicht die Ausgabe einer Variablen x vom Typ `mpfr_t` über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(mpfr_get_prec(x)/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Der Wert x wird dabei in der mit `SetBase(k)` voreingestellten Basis ausgegeben. Für die meist dezimale Ausgabe ist `k` = 10 zu wählen.

```
friend std::istream& operator >> (std::istream& is, MpfrClass& x);
```

Ermöglicht das Einlesen einer MpfrClass-Variablen über den Standard-Eingabestrom "cin". Die eingegebene Zahl ist auf keine Stellenanzahl begrenzt. Die Zahl muss in der eingestellten Basis eingegeben werden, sonst entsteht eine Fehlermeldung. Die Rundung erfolgt mit dem voreingestelleten Current-Rundungsmodus, siehe das Programm auf Seite 19.

3.4 Rundungsmodi und Precision Handling

```
PrecisionType GetPrecision () const;
```

Diese Memberfunktion gibt die Präzision des aktuellen Objekts in Bits zurück. Als Beispiel entsprechen dabei 302 Bits $302/\log_2(10) \approx 91$ Dezimalstellen.

```
void SetPrecision (PrecisionType prec);
```

Diese Memberfunktion setzt die Präzision des aktuellen Objektes auf prec. Sein Wert bleibt nicht erhalten.

```
void RoundPrecision (PrecisionType prec, RoundingMode rnd);
```

Diese Memberfunktion rundet das aktuelle Objekt auf die neue Precision prec, sein Wert bleibt dabei i.a. nicht erhalten. Sollte die Präzision des Objektes größer sein als prec, wird das Objekt mit Hilfe des eingestellten Rundungsmodus so gerundet, dass es in das Format der Präzision prec passt. Ist die Präzision kleiner als prec, werden die restlichen binären Stellen mit Nullen aufgefüllt.

```
static const PrecisionType GetCurrPrecision ();
```

Gibt die aktuelle Current-Precision in Bits zurück.

```
static void SetCurrPrecision (PrecisionType prec);
```

Setzt die Current-Precision auf prec. Wird die Current-Precision nicht gesetzt, so wird mit der Default-Precision von 53 Bits gerechnet.

```
static const int GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen.

```
static void SetBase (int b);
```

Setzt die aktuelle Basis auf b. Dies muss ein Wert zwischen 2 und 36 sein. Die Basis hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die internen Rechnungen erfolgen stets im Binärsystem!

```
static const RoundingMode GetCurrRndMode ();
```

Gibt den aktuellen Rundungsmodus zurück mit den Werten: (0, 1, 2, 3, 4).

```
static void SetCurrRndMode (RoundingMode rnd);
```

Setzt den Current-Rundungsmodus auf rnd. Für rnd sind fünf verschiedene Modi möglich:

- RoundNearest: Rundung zur nächsten Rasterzahl (0)
- RoundToZero: Rundung in Richtung Null (1)
- RoundFromZero: Rundung weg von der Null (4)
- RoundUp: Aufrunden (2)
- RoundDown: Abrunden (3)

Wird der Rundungsmodus mit `SetCurrRndMode` nicht gesetzt, so wird als Default-Rundungsmodus `RoundNearest` benutzt.

3.5 Anwendungsprogramm

Das folgende C-XSC Programm `MPFR-01.cpp` zeigt einen Konstruktor-Aufruf, den Eingabe- und Ausgabe-Mechanismus und das Rundungs- bzw. Precision-Handling:

```
1  // MPFR-01.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "mpfrclass.hpp"
5  using namespace MPFR;
6  using namespace std;
7  int main(void)
8  {
9    MpfrClass::SetCurrRndMode (RoundUp);
10   cout << "\nRoundingMode = " << MpfrClass::GetCurrRndMode() << endl;
11   MpfrClass::SetCurrPrecision (40);
12   cout << "Current-Precision = " << MpfrClass::GetCurrPrecision() << endl;
13
14   MpfrClass x(1.2345678, RoundDown, 20); // Konstruktor mit double-Argument
15   cout.precision(x.GetPrecision()/3.32192809); // Dezimale Stellenzahl
16   cout << "x = " << x << endl;
17   cout << "Precision of x = " << x.GetPrecision() << endl << endl;
18
19   x.SetPrecision(70); // Inhalt von x wird dabei geloescht
20   cout << "x = ? " ;  cin >> x;
21   cout.precision(x.GetPrecision()/3.32192809);
22   cout << "x = " << x << endl;
23   cout << "Precision of x = " << x.GetPrecision() << endl << endl;
24   return 0;
25 }
```

Bei der interaktiven Eingabe (Zeile 22) von 1.2345678 erhält man die folgende Ausgabe:

```
RoundingMode = 2
Current-Precision = 40
x = 1.23457
Precision of x = 20

x = ? 1.2345678
x = 1.23456780000000000001
Precision of x = 70
```

- Beim Konstruktoraufruf (14) wird 1.2345678 zunächst vom Compiler, vermutlich mit `RondToNearest`, in einem internen *double*-Binärformat gespeichert, das dann vom Konstruktor durch Abrunden in das Objekt `x` mit der Precision von 20 Bit geschrieben wird, was etwa $20/3.3219 \approx 6$ Dezimalziffern entspricht. Also Vorsicht:

  Da man nicht genau weiß, wie der Compiler die dezimale Eingabe 1.2345678 intern speichert, weiß man auch nicht genau, welchen Wert das Objekt `x` durch den Konstruktor-Aufruf erhält. Diese Unsicherheit lässt sich vermeiden, wenn man in C-XSC z.B. eine *real*-Variable anlegt und diese mit `cin` und entsprechenden Rundungsmanipulatoren (z.B `RndNext`) mit 1.2345678 initialisiert und dann diese *real*-Variable an den Konstruktor übergibt.

- Das *double*-Format mit 53 Mantissen-Bits ergibt eine maximale Precision von $53/3.3219 \approx 16$ Dezimalstellen. Will man den Dezimalwert 1.2345678, z.B. aufgerundet, mit einer Precision von z.B. 70 Bit, d.h. etwa 21 Dezimalstellen, in ein Objekt `x` speichern, so kann dies nach Zeile 20 mit `cin` realisiert werden, wobei jedoch zum Aufrunden in Zeile 9 der Current-Rundungsmodus `RoundUp` zu setzen ist.

- In (15) und (21) wird mit `cout.precision(...)` das Ausgabeformat festgelegt.

3.6 Typ-Umwandlungen

Um ein flexibles Arbeiten zwischen C-XSC und der MPFR-Bibliothek zu ermöglichen, wurden möglichst viele Typ-Umwandlungs-Funktionen bereitgestellt:

3.6.1 `real, double, ...` → MPFR

```
MpfrClass real2Mpfr    (const cxsc::real& op);
MpfrClass double2Mpfr  (const double& op);
MpfrClass int2Mpfr     (const int& op);
MpfrClass mpfr_t2Mpfr  (const mpfr_t& op);
```

Obige Funktionen liefern mit dem jeweiligen Eingabewert op einen Rückgabewert vom Typ `MpfrClass` in einer Präzision, die gewährleistet, dass der Rückgabewert genau dem Wert von op entspricht. Die Präzision des Rückgabewertes wird also i.a. nicht mit der Current-Precision übereinstimmen! Die obigen vier Funktionen kommen u.a. bei den Vergleichsoperatoren zur Anwendung.

3.6.2 MPFR → `real, double, ...`

Die folgenden drei Funktionen liefern mit einem Objekt op vom Typ `MpfrClass` jeweils einen i.a. **gerundeten** Rückgabewert vom Typ `real, double, long int;`

```
cxsc::real     to_real    (const MpfrClass& op, RoundingMode rnd);
double         to_double  (const MpfrClass& op, RoundingMode rnd);
long int       to_int     (const MpfrClass& op, RoundingMode rnd);
```

Mit z.B. `rnd = RoundUp` wird aufgerundet. Ohne Angabe eines Rundungsmodus wird jedoch nach dem voreingestellten Current-Rundungsmodus gerundet.

3.6.3 MPFR → `mpfr_t`

```
const mpfr_t& getvalue(const MpfrClass& r)
```

Obige Funktion liefert von einem als `const` deklarierten Objekt r eine Referenz auf den Wert seines Attributs `mpfr_rep` vom Typ `mpfr_t`.
Anwendung: `const`-Parameter in einer Funktionen-Parameterliste, vgl. z.B. die Funktion `MpfiClass MpfrClass2Mpfi(const MPFR::MpfrClass& v)` in der Datei `mpficlass.cpp`.

3.6.4 MPFR ↔ `mpfr_t`

```
mpfr_t& MpfrClass::GetValue();
```

Mithilfe der Memberfunktion `GetValue()`, die eine Referenz auf den Wert `mpfr_rep` vom Typ `mpfr_t` des aktuellen Objekts liefert, können sowohl `MpfrClass`-Objekte an eine Funktion übergeben als auch referenzierte Rückgabewerte vom Typ `mpfr_t` von einer Funktion übernommen werden. Mithilfe von `GetValue()` kann man daher Funktionen mit referenzierten `mpfr_t`-Parametern einfach aufrufen, vergleiche dazu das Programm `MPFR-02` auf Seite 28.

3.6.5 `mpfr_t` → MPFR

```
void SetValue(const mpfr_t& t);
```

Mithilfe der Memberfunktion `SetValue()` wird der `mpfr_rep`-Wert des aktuellen Objekts auf `t` gesetzt, wobei `mpfr_rep` die Präzision von `t` übernimmt, d.h. der Wert von `t` wird ohne Rundung übernommen.

3.6.6 `string` → MPFR

```
MpfrClass string2Mpfr(const std::string& op, Roundingmode rnd,
                                             PrecisionType prec);
```

Der Aufruf `string2Mpfr(op);` rundet den String op mittels der voreingestellten Current-Precision und mit dem Current-Rundungsmodus in ein Klassenobjekt vom Typ `MpfrClass`. Der Aufruf `string2Mpfr(op, RoundNearest, 140);` rundet op in ein Klassenobjekt vom Typ `MpfrClass` der Präzision 140 Bits. Gerundet wird dabei zur nächsten Rasterzahl dieses Formats. Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in eine binäre Zahl konvertiert werden kann, so dass daher Rundungen i.a. nicht zu vermeiden sind. Eine weitere Möglichkeit, einen String in ein `MpfrClass`-Objekt zu verwandeln, besteht in einem entsprechenden Konstruktor-Aufruf, vgl. Seite 16. **Achtung**: Keine Leerzeichen am String-Ende!

3.6.7 MPFR → `string`

```
std::string to_string (const MpfrClass& x, RoundingMode rnd,
                                           PrecisionType prec);
std::string to_string (const mpfr_t& x,    RoundingMode rnd,
                                           PrecisionType prec);
```

x wird mittels rnd in einen String `s` mit prec Dezimalstellen gerundet, wenn Base gleich 10 ist. Wählt man prec hinreichend groß, so stellt der String den Wert von x **exakt** dar, weil eine binäre Zahl stets exakt in eine Dezimalzahl umgewandelt werden kann.

Wird prec nicht angegeben, so wird x mittels rnd in einen String gerundet, der bei Base=10 soviele Dezimalstellen besitzt, wie es der Präzision von `x` entspricht. Ist die Präzision von `x` z.B. 302, so wird ein String von $302/\log_2(10) \approx 91$ Dezimalstellen generiert. Wird neben prec auch rnd nicht angegeben, so wird mittels des Current-Rundungsmodus in den String mit gleicher Dezimalstellenzahl gerundet.

3.6.8 MPFR → MPFR

Beachten Sie bitte, dass bei einer Wertzuweisung an eine `MpfrClass`-Variable mit Hilfe des Operators `=` der linke Operand stets auf die Current-Precision gesetzt wird und dass der rechte `MpfrClass`-Operand dabei **stets** bez. des Current-Rundungsmodus in den linken Operanden gerundet wird, vgl. dazu auch Seite 23. Will man jedoch abweichend von dieser Rundung einen anderen Rundungsmodus benutzen, so kann dies mit folgender Funktion ohne Rückgabewert realisiert werden:

```
void set_Mpfr (MpfrClass& op, const MpfrClass& op1, RoundingMode rnd,
                                                    PrecisionType prec);
```

Folgende Funktionsaufrufe sind möglich:

1. `set_Mpfr (op, op1, RoundUp, prec);`
   op erhält die Präzision prec und den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls aufgerundet wird. Setzt man prec gleich der Präzision von op1, so erhält op den **exakten** Wert von op1, und zwar unabhängig vom gewählten Rundungsmodus rnd.

2. `set_Mpfr (op, op1, RoundDown);`
   op wird auf die Current-Precision gesetzt und erhält den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls abgerundet wird.

3. `set_Mpfr (op, op1);`
   op wird auf die Current-Precision gesetzt und erhält den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls bez. des Current-Rundungsmodus gerundete wird.

3.6.9 Verschiedenes

```
int sign(const MpfrClass& x)
```

Zurückgegeben wird das Vorzeichen von x.

```
sign(NaN) = 0;   sign(+inf) = +1;   sign(-inf) = -1;
```

```
long int expo (const MpfrClass& x);
```

Zurückgegeben wird der Exponent `e` von x. Ist `m` die Mantisse einer normalen Gleitpunktzahl $x \neq 0$, so gilt: $x = \mathtt{m} \cdot 2^{\mathtt{e}}$, $|\mathtt{m}| \in [0.5, 1)$.

```
expo(0)     = -9223372036854775807; expo(NaN)  = -9223372036854775806;
expo(+inf) = -9223372036854775805; expo(-inf) = -9223372036854775805;
expo(MaxFloat()) = 1073741823;     expo(minfloat()) = -1073741823;
```

Beachten Sie: Die Mantisse `m` wird mithilfe von `mant(x)` bestimmt, vgl. Seite 31. Im Gegensatz zu C-XSC ist bei festem `prec` die Präzision der Mantisse `m` jedoch unabhängig vom Wert des Zweier-Exponenten `expo(x)`, wobei für $x \neq 0$ gilt

$$-1073741823 \leq \mathtt{expo}(x) \leq +1073741823.$$

Danach erhält man z.B. `mant(succ(0.5)) = mant(succ(minfloat(0)))`, wobei `minfloat()` die kleinste positive Zahl vom Typ `MpfrClass` ist, vgl. Seite 32.

Beachten Sie auch, dass bei einer Multiplikation von x mit 2^k, $k \in \mathbb{Z}$, z.B. mit Hilfe von `times2pown(x,k,rnd)`, der Rückgabewert x das **exakte** Produkt $x \cdot 2^k$ liefert, solange kein Über- oder Unterlauf eintritt.

3.7 Zuweisungs-Operatoren

Bei den folgenden fünf Zuweisungs-Operationen erhält der linke Operand als Präzision stets die aktuelle Current-Precision, und der rechte Operand op wird nach dem aktuellen Current-Rundungsmodus in den linken Operanden gerundet.

```
MpfrClass& operator = (const MpfrClass& op);
MpfrClass& operator = (const mpfr_t& op);
MpfrClass& operator = (const cxsc::real& op);
MpfrClass& operator = (const double& op);
MpfrClass& operator = (const int& op);
```

Ist z.B. op vom Typ `real` oder `double` und ist die Current-Precision kleiner als 53, so wird op i.a. in den linken Operanden gerundet. Nur wenn die Current-Precision größer oder gleich 53 ist, wird der Wert des linken Operanden gleich dem Wert des rechten Operanden. Für andere Typen des rechten Operanden gelten ganz entsprechende Aussagen.

Ist z.B. op vom Typ `MpfrClass` und ist seine Präzision größer als die Current-Precision, so wird op in den linken Operanden bez. des Current-Rundungsmodus gerundet, d.h. die Werte des linken und rechten Operanden werden dann i.a. **verschieden** sein!

3.8 Abfragen

Bei allen folgenden Abfragefunktionen braucht die Präzision von x nicht mit der Current-Precision übereinzustimmen.

```
bool isNan    (const MpfrClass& x);
bool isInf    (const MpfrClass& x);
bool isNumber(const MpfrClass& x);
```

`isNan` und `isInf` überprüfen, ob x gleich NaN bzw. ±Inf ist. `isNumber` überprüft, ob x eine normale `MpfrClass` Zahl ungleich NaN und ungleich ±Inf ist.

```
bool isZero    (const MpfrClass& x);
bool isNeg     (const MpfrClass& x);
bool isPos     (const MpfrClass& x);
bool isInteger(const MpfrClass& x);
bool isEven    (const MpfrClass& x);
bool isOdd     (const MpfrClass& x);
```

Die oberen sechs Funktionen sind selbsterklärend, wobei die drei letzten wirklich praktische Bedeutung haben!
Mit `isInteger(x)` wird lediglich geprüft, ob $x \in \mathbb{Z}$ erfüllt ist, wobei `x` jedoch nicht von Typ `int` oder `long int` sein muss.

3.9 Vergleiche

Mit den folgenden Vergleichsfunktionen werden die üblichen Vergleichsoperatoren implementiert, wobei wenigstens ein Operand vom Typ `MpfrClass` sein muss.

3.9.1 Vergleichsfunktionen

```
int compare_equal      (const MpfrClass& x, const MpfrClass& y);
int compare_less       (const MpfrClass& x, const MpfrClass& y);
int compare_lessequal  (const MpfrClass& x, const MpfrClass& y);
int compare_greater    (const MpfrClass& x, const MpfrClass& y);
int compare_greaterequal (const MpfrClass& x, const MpfrClass& y);
```

Die obigen Funktionen liefern einen Wert ungleich Null, wenn jeweils gilt: $x = y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$ und den Wert Null sonst. Ist x oder y NaN, so wird Null zurückgegeben. Die Präzisionen von x und y können verschieden sein und müssen mit der Current-Precision nicht übereinstimmen.

3.9.2 Vergleichsoperatoren $=, \neq, >, \geq, <, \leq$

```
bool operator == (const MpfrClass& op1, const MpfrClass&      op2);
bool operator == (const MpfrClass& op1, const double&         op2);
bool operator == (const MpfrClass& op1, const int             op2);
bool operator == (const MpfrClass& op1, const cxsc::real&     op2);
bool operator == (const double&       op2, const MpfrClass& op1);
bool operator == (const int           op2, const MpfrClass& op1);
bool operator == (const cxsc::real&   op2, const MpfrClass& op1);

bool operator != (const MpfrClass& op1, const MpfrClass&      op2);
bool operator != (const MpfrClass& op1, const double&         op2);
bool operator != (const MpfrClass& op1, const int             op2);
bool operator != (const MpfrClass& op1, const cxsc::real&     op2);
bool operator != (const double&       op2, const MpfrClass& op1);
bool operator != (const int           op2, const MpfrClass& op1);
bool operator != (const cxsc::real&   op2, const MpfrClass& op1);

bool operator < (const MpfrClass& op1, const MpfrClass&      op2);
bool operator < (const MpfrClass& op1, const double&         op2);
bool operator < (const MpfrClass& op1, const int             op2);
bool operator < (const MpfrClass& op1, const cxsc::real&     op2);
bool operator < (const double&       op2, const MpfrClass& op1);
bool operator < (const int           op2, const MpfrClass& op1);
bool operator < (const cxsc::real&   op2, const MpfrClass& op1);

bool operator <= (const MpfrClass& op1, const MpfrClass&      op2);
bool operator <= (const MpfrClass& op1, const double&         op2);
bool operator <= (const MpfrClass& op1, const int             op2);
bool operator <= (const MpfrClass& op1, const cxsc::real&     op2);
bool operator <= (const double&       op2, const MpfrClass& op1);
bool operator <= (const int           op2, const MpfrClass& op1);
bool operator <= (const cxsc::real&   op2, const MpfrClass& op1);

bool operator > (const MpfrClass& op1, const MpfrClass&      op2);
bool operator > (const MpfrClass& op1, const double&         op2);
bool operator > (const MpfrClass& op1, const int             op2);
bool operator > (const MpfrClass& op1, const cxsc::real&     op2);
bool operator > (const double&       op2, const MpfrClass& op1);
bool operator > (const int           op2, const MpfrClass& op1);
bool operator > (const cxsc::real&   op2, const MpfrClass& op1);
```

```
bool operator >= (const MpfrClass& op1, const MpfrClass&          op2);
bool operator >= (const MpfrClass& op1, const double&             op2);
bool operator >= (const MpfrClass& op1, const int                 op2);
bool operator >= (const MpfrClass& op1, const cxsc::real&         op2);
bool operator >= (const double&           op2, const MpfrClass& op1);
bool operator >= (const int               op2, const MpfrClass& op1);
bool operator >= (const cxsc::real&       op2, const MpfrClass& op1);
```

Sind die Operanden op1 und op2 beide vom Typ `MpfrClass`, so müssen ihre Präzisionen nicht übereinstimmen und insbesondere auch nicht mit der Current-Precision.

3.10 Arithmetische Operatoren

Für alle arithmetischen Operatoren gilt:

> Das exakte Ergebnis einer arithmetischen Operation wird unabhängig von der Präzision der Operanden mit dem voreingestellten Current-Rundungsmodus optimal gerundet. Die Ergebnis-Präzision ist dabei stets gleich der voreingestellten Current-Precision.
> Die Operatoren $\circledcirc=$, mit $\circledcirc \in \{+, -, \cdot, /\}$, bedeuten $u \circledcirc = v \iff u = u \circledcirc v$. Dabei wird $u \circledcirc v$ mit dem Current-Rundungsmodus in die Current-Precision gerundet und in u gespeichert, wobei u als Präzision die Current-Precision erhält.

3.10.1 Addition

```
MpfrClass operator + (const MpfrClass& op1,  const MpfrClass& op2);
MpfrClass operator + (const MpfrClass& op1,  const mpfr_t& op2);
MpfrClass operator + (const MpfrClass& op1,  const double& op2);
MpfrClass operator + (const MpfrClass& op1,  const cxsc::real& op2);
MpfrClass operator + (const MpfrClass& op1,  const int op2);

MpfrClass operator + (const mpfr_t& op1,     const MpfrClass& op2);
MpfrClass operator + (const double& op1,     const MpfrClass& op2);
MpfrClass operator + (const cxsc::real& op1, const MpfrClass& op2);
MpfrClass operator + (const int op1,         const MpfrClass& op2);
```

```
MpfrClass& operator += (MpfrClass& op1, const MpfrClass& op2);
MpfrClass& operator += (MpfrClass& op1, const mpfr_t& op2);
MpfrClass& operator += (MpfrClass& op1, const double& op2);
MpfrClass& operator += (MpfrClass& op1, const cxsc::real& op2);
MpfrClass& operator += (MpfrClass& op1, const int op2);
```

3.10.2 Subtraktion

Beachten Sie den Hinweis auf Seite 26

```
MpfrClass operator - (const MpfrClass& op1, const MpfrClass& op2);
MpfrClass operator - (const MpfrClass& op1, const mpfr_t& op2);
MpfrClass operator - (const MpfrClass& op1, const double& op2);
MpfrClass operator - (const MpfrClass& op1, const cxsc::real& op2);
MpfrClass operator - (const MpfrClass& op1, const int op2);

MpfrClass operator - (const mpfr_t& op1,         const MpfrClass& op2);
MpfrClass operator - (const double& op1,         const MpfrClass& op2);
MpfrClass operator - (const cxsc::real& op1,     const MpfrClass& op2);
MpfrClass operator - (const int op1,             const MpfrClass& op2);
```

```
MpfrClass& operator -= (MpfrClass& op1, const MpfrClass& op2);
MpfrClass& operator -= (MpfrClass& op1, const mpfr_t& op2);
MpfrClass& operator -= (MpfrClass& op1, const double& op2);
MpfrClass& operator -= (MpfrClass& op1, const cxsc::real& op2);
MpfrClass& operator -= (MpfrClass& op1, const int op2);
```

3.10.3 Multiplikation

Beachten Sie den Hinweis auf Seite 26

```
MpfrClass operator * (const MpfrClass& op1, const MpfrClass& op2);
MpfrClass operator * (const MpfrClass& op1, const mpfr_t& op2);
MpfrClass operator * (const MpfrClass& op1, const double& op2);
MpfrClass operator * (const MpfrClass& op1, const cxsc::real& op2);
MpfrClass operator * (const MpfrClass& op1, const int op2);

MpfrClass operator * (const mpfr_t& op1,          const MpfrClass& op2);
MpfrClass operator * (const double& op1,          const MpfrClass& op2);
MpfrClass operator * (const cxsc::real& op1,      const MpfrClass& op2);
MpfrClass operator * (const int op1,              const MpfrClass& op2);
```

```
MpfrClass& operator *= (MpfrClass& op1, const MpfrClass& op2);
MpfrClass& operator *= (MpfrClass& op1, const mpfr_t& op2);
MpfrClass& operator *= (MpfrClass& op1, const double& op2);
MpfrClass& operator *= (MpfrClass& op1, const cxsc::real& op2);
MpfrClass& operator *= (MpfrClass& op1, const int op2);
```

3.10.4 Division

Beachten Sie den Hinweis auf Seite 26

```
MpfrClass operator / (const MpfrClass& op1, const MpfrClass& op2);
MpfrClass operator / (const MpfrClass& op1, const mpfr_t& op2);
MpfrClass operator / (const MpfrClass& op1, const double& op2);
MpfrClass operator / (const MpfrClass& op1, const cxsc::real& op2);
MpfrClass operator / (const MpfrClass& op1, const int op2);

MpfrClass operator / (const mpfr_t& op1,          const MpfrClass& op2);
MpfrClass operator / (const double& op1,          const MpfrClass& op2);
MpfrClass operator / (const cxsc::real& op1,      const MpfrClass& op2);
MpfrClass operator / (const int op1,              const MpfrClass& op2);
```

```
MpfrClass& operator /= (MpfrClass& op1, const MpfrClass& op2);
MpfrClass& operator /= (MpfrClass& op1, const mpfr_t& op2);
MpfrClass& operator /= (MpfrClass& op1, const double& op2);
MpfrClass& operator /= (MpfrClass& op1, const cxsc::real& op2);
MpfrClass& operator /= (MpfrClass& op1, const int op2);
```

3.10.5 Vom Current-Rundungsmodus abweichende Rundungen

Bei allen arithmetischen Operatoren $\{+, -, *, /\}$ und $\{+=, -=, *=, /=\}$ werden die exakten Ergebnisse nach dem mit `SetCurrRndMode(...)` voreingestellten Current-Rundungsmodus in das jeweilige Ergebnisobjekt gerundet, siehe Seite 18.

Wenn jedoch, z.B. bei einer Multiplikation zweier `MpfrClass`-Objekte `a,b`, abweichend vom Current-Rundungsmodus z.B. ein Aufrunden verlangt wird, so kann dies mit Hilfe der MPFR-Funktion

```
int mpfr_mul (mpfr_t ROP, mpfr_t OP1, mpfr_t OP2, mpfr_rnd_t RND)
```

realisiert werden, wenn `RND` durch `RoundUp` ersetzt wird. Weitere Informationen findet man in den Dateien `mpfr.info`, `mpfr.pdf` oder `mpfr.dvi`, wobei die beiden letzten Dateien durch

```
make pdf   bzw.   make dvi
```

in dem Verzeichnis erzeugt werden können, in dem sich `mpfr.texi` befindet. Das nachfolgende Programm zeigt für das Aufrunden die entsprechenden Anweisung:

```
1  // MPFR-02.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "mpfrclass.hpp"
5
6  using namespace MPFR;
7  using namespace std;
8
9  int main(void)
10 {
11   MpfrClass::SetCurrRndMode (RoundDown);
12   cout << "\nCurrent-RoundingMode = " << MpfrClass::GetCurrRndMode() << endl;
13   MpfrClass::SetCurrPrecision (20);
14   cout << "Current-Precision = " << MpfrClass::GetCurrPrecision() << endl;
15
16   MpfrClass a(1.2345678901234 56, RoundNearest, 20),
17             b(1.2345678901234 56, RoundNearest, 20), y;
18
19   mpfr_mul(y.GetValue(), a.GetValue(), b.GetValue(), RoundUp);
20   cout.precision(y.GetPrecision()/3.32193 + 2); // ca. 8 Dezimalstellenzahl
21   cout << "a*b (RoundUp)     = " << y << endl;
22   cout << "y.GetPrecision() = " << y.GetPrecision() << endl << endl;
23
24   y = a*b; // Rundung mit RoundDown
25   cout.precision(y.GetPrecision()/3.32193 + 2); // ca. 8 Dezimalstellenzahl
26   cout << "a*b (RoundDown)   = " << y << endl;
27   cout << "y.GetPrecision() = " << y.GetPrecision() << endl;
28
29   return 0;
30 }
```

Obiges Programm erzeugt die folgende Ausgabe:

```
Current-RoundingMode = 3
Current-Precision = 20
a*b (RoundUp)     = 1.5241584
y.GetPrecision() = 20

a*b (RoundDown)   = 1.5241565
y.GetPrecision() = 20
```

Im Vergleich zum exakten Produkt 1.5241578... erkennt man den aufgerundeten Wert und den bez. `Current-RoundingMode = RoundDown` (3) abgerundeten Produktwert.

Hier noch einige Anmerkungen zur MPFR-Funktion `mpfr_mul(...)`

1. Neben den C-XSC Funktionen können die in `mpfr.info` beschriebenen MPFR-Funktionen, wie z.B. `mpfr_mul(...)`, ebenfalls aufgerufen werden. Für die Implementierung weiterer Funktionen ist dies eine große Hilfe.

2. Ergebnisse werden in der Parameterliste der MPFR-Funktionen stets an **erster** Stelle abgelegt.

3. Wird die Ergebnis-Variable `y` vor dem MPFR-Funktionsaufruf durch z.B. `MpfrClass y;` lediglich deklariert, so erhält `y` die voreingestelle Current-Precision und zwar **unabhängig** von den Präzisionen beider Operanden in der Parameterliste.

4. Wird die Ergebnis-Variable `y` jedoch vor dem MPFR-Funktionsaufruf, z.B. durch

   ```
   MpfrClass y(0, RoundNearest, 50);
   ```

   mit der Präzision von 50 Bit initialisiert, so liefert `mpfr_mul(...)` das Erbenis `y` mit der Präzision von 50 Bit und zwar wieder **unabhängig** von den Präzisionen beider Operanden.

Beachten Sie, dass bei den arithmetischen Operatoren mit dem Ergebnistyp `MpfrClass` die Ergebnis-Präzision stets gleich der Current-Precision ist, wobei gegebenenfalls das exakte Ergebnis bez. des Default-Rundungsmodus in den Ergebnisoperanden gerundet wird.

3.11 Mathematische Funktionen

3.11.1 Standard-Implementierung

Die MPFR-Bibliothek stellt eine Vielzahl von Elementarfunktionen und einige Funktionen der Mathematischen Physik zur Verfügung, wobei zu einem Argument `x` mit beliebiger Präzision zunächst der exakte Funktionswert `y0` berechnet wird. Danach wird dann `y0` in eine Ergebnisvariable `y` mit einer möglichen anderen Präzision gerundet, wobei diese Rundung durch einen entsprechenden Parameter `rnd` gesteuert werden kann. Die Deklaration z.B. der Exponentialfunktion ist mit diesen Bezeichnungen gegeben durch:

```
int mpfr_exp (mpfr_t y, mpfr_t x, mpfr_rnd_t rnd)
```

Mit obiger MPFR-Funktion wird die Exponentialfunktion für das C-XSC Interface wie folgt implementiert:

```
MpfrClass exp (const MpfrClass& x, RoundingMode rnd)
{
   MpfrClass y(0); // Die Praezision von y ist jetzt die Current-Precision
   mpfr_exp(y.mpfr_rep, x.mpfr_rep, rnd);
   return y;
}
```

Mit Hilfe des Konstruktoraufrufs wird also zunächst die Präzision der Ergebnisvariablen `y` auf die Current-Precision gesetzt, vgl. dazu auch Seite 16. Danach wird mit dem Argument `x` und seiner Präzision der exakte Wert `y0` = $e^{\texttt{x}}$ berechnet und dann mittels des Rundungsmodus `rnd` nach `y` gerundet. Wird `rnd` nicht gesetzt, so wird nach dem Current-Rundungsmodus gerundet und ist dieser nicht gesetzt, so wird zur jeweils nächsten Rasterzahl gerundet, (`RoundNearest`).

Soll nun ganz analog zur Exponentialfunktion im C-XSC Interface eine neue Funktion implementiert werden, die in der MPFR-Bibliothek noch nicht definiert ist, so muss wie folgt vorgegangen werden:

1. Sichere den Wert der ursprünglichen Current-Precision in `prec_old`
2. Setze die neue Current-Precision `prec` auf die Präzision vom Argument `x`. Wenn dann gilt `prec` < `prec_old`, setze `prec` = `prec_old`.
3. Setze mit `MpfrClass y(0);` die Präzision von `y` auf den Wert von `prec`.
4. Berechne in der neuen Current-Precision den mittels `rnd` gerundeten Funktionwert, der in `y` abzulegen ist.
5. Runde mittels `rnd` durch `y.RoundPrecision(prec_old, rnd);` auf die ursprüngliche Current-Precision.
6. Durch `SetCurrPrecision(prec_old);` die alte Current-Precision wiederherstellen.
7. Durch `return y;` den gerundeten Funktionswert zurückgeben, fertig!

Anmerkung:

- Grundsätzlich wird also der mittels `rnd` gerundete Funktionswert in der ursprünglichen Current-Precision `prec_old` zurückgegeben. Die interne Berechnung erfolgt jedoch in der durch `prec` vorgegebenen Präzision, die mindestens so groß ist wie `prec_old` selbst. Ein Beispiel dazu findet man in `mpfrclass.cpp` bei der Definition der `acoth`-Funktion.

3.11.2 Davon abweichende Funktionen und Konstanten

Bei nur wenige mathematische Funktionen ist es sinnvoll, bei ihrer Implementierung von dem auf Seite 30 angegebenen Schema abzuweichen. Die Ausnahmen sind:

```
MpfrClass abs (const MpfrClass& op, RoundingMode rnd, PrecisionType prec);
```

Zurückgegeben wird der i.a. gerundete Absolutbetrag von op. Für die Funktion gibt es drei verschiedene Aufrufmöglichkeiten:

1. `abs (op);`
   $|op|$ wird bez. des Current-Rundungsmodus in die Current-Precision gerundet und zurückgegeben.
2. `abs (op, RoundUp);`
   $|op|$ wird in die Current-Precision aufgerundet und zurückgegeben.
3. `abs (op, RoundDown, prec);`
   $|op|$ wird in ein Format mit der Präzision prec abgerundet und zurückgegeben. Die Präzision des zurückgegebenen Wertes wird also i.a. von der voreingestellten Current-Precision verschieden sein!
   Will man jedoch $|op|$ **rundungsfehlerfrei** mit der gleichen Präzision von op zurückgeben, so gelingt dies in allen Fällen, unabhängig von der voreingestellten Current-Precision, mit dem Funktionsaufruf:

   ```
   abs (op, RoundNearest, op.GetPrecision());
   ```

   wobei der Rundungsmodus (hier `RoundNearest`) natürlich beliebig gesetzt werden kann. Stimmt die Präzision von op mit der Current-Precision überein, so liefert der Aufruf `abs (op);` ebenfalls den **rundungsfehlerfreien** Wert von $|op|$. In der Praxis wird die rundungsfehlerfreie Rückgabe von $|op|$ vermutlich immer im Vordergrund stehen.

Die Funktion `abs` kann also sehr flexibel eingesetzt werden und funktioniert nach 1. und 2. wie bei der Standard-Implementierung von Seite 30. Lediglich der Punkt 3. weicht von dieser Standard-Implementierung ab, um den exakten Wert von $|op|$ in jedem Fall garantieren zu können.
Beachten Sie außerdem: Soll $|k|$, k vom Typ `int`, berechnet werden, so muss dies mit `std::abs(`k`)` erfolgen.

```
MpfrClass mant (const MpfrClass& x);
```

Die Mantisse `m` von `x` wird rundungsfehlerfrei in der Präzision von `x` zurückgegeben. Im Fall einer normalen Gleitpunktzahl `x` $\neq 0$ gilt: $|$`m`$| \in [0.5, 1)$, vgl. auch Seite 22.
`x` $= 0 \rightarrow$ `m` $= 0$; `x` $=$ `NaN` $\rightarrow$ `m` $=$ `NaN`; `x` $= \pm$`Inf` $\rightarrow$ `m` $= \pm$`Inf`;

```
MpfrClass comp (const MpfrClass& x, const long int k);
```

Mit der Mantisse `x` und dem Zweierexponenten `k` wird `x` $\cdot 2^{k}$ in der Präzision von `x` **rundungsfehlerfrei** zurückgegeben. Im Fall `x` $= 0$ wird 0 zurückgegeben und in den Fällen `x` $=$ `NaN` oder `x` $= \pm$`Inf` erhält man `NaN`. Ist `x` $\neq 0$ eine normale Maschinenzahl und gilt: $|$`x`$| \notin [0.5, 1)$ oder $|$`k`$| > 1073741823$, so wird ebenfalls `NaN` zurückgegeben.

```
MpfrClass min (const MpfrClass& op1, MpfrClass& op2);
MpfrClass max (const MpfrClass& op1, MpfrClass& op2);
```

Zurückgegeben wird **rundungsfehlerfrei** das Minimum bzw. das Maximum beider Operanden und zwar genau in der Präzision des jeweils zurückgegebenen Operanden, die von der Current-Precision durchaus verschieden sein kann!

```
MpfrClass Round (const MpfrClass& op);
MpfrClass Floor (const MpfrClass& op);
MpfrClass Ceil  (const MpfrClass& op);
MpfrClass Trunc (const MpfrClass& op);
MpfrClass Frac  (const MpfrClass& op);
```

Die obigen fünf Rückgabewerte vom Typ `MpfrClass` haben alle die Präzision von op, die mit der Current-Precision nicht übereinstimmen muss. Die ersten vier Funktionen liefern den jeweiligen rundungsfehlerfreien ganzzahligen Anteil von op, wobei `Round()` weg von der Null rundet, falls op genau zwischen benachbarten ganzzahligen Werten liegt. Die letzte Funktion liefert den rundungsfehlerfreien nicht-ganzzahligen Teil von op.

```
MpfrClass minfloat (PrecisionType prec);
MpfrClass MaxFloat (PrecisionType prec);
```

Zurückgegeben werden der positive kleinste bzw. größte Zahlenwert im Datenformat der Präzision prec. Im Gegensatz zu `minfloat(prec)` fallen die entsprechenden Rückgabewerte von `MaxFloat(prec)` in Abhängigkeit von prec unterschiedlich aus! Werden die Funktionen ohne prec aufgerufen, so wird das Format mit der Current-Precision benutzt.

```
MpfrClass pred (const MpfrClass& op);
MpfrClass succ (const MpfrClass& op);
```

Beide Funktionen geben den Vorgänger bzw. Nachfolger von op zurück und zwar in der **gleichen** Präzision des Operanden op. Es macht nämlich keinen mathematischen Sinn, diese Werte anschließend in ein Format mit anderer Präzision zu runden, da sich die Funktionen `pred` und `succ` genau auf das Format von op beziehen.

```
void times2pown (MpfrClass& op, long int op1, RoundingMode rnd);
```

Obige Funktion liefert mit dem Eingabewert op den Wert $op \cdot 2^{op1}$ mit gleicher Präzision zurück. Solange kein Über- oder Unterlauf entsteht, wird $op \cdot 2^{op1}$ **exakt**, d.h. rundungsfehlerfrei berechnet. Tritt jedoch z.B. ein Überlauf ein, so wird gemäß rnd gerundet. Wenn rnd nicht gesetzt wird, so erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Ist dieser nicht gesetzt, so erfolgt die Rundung weg von der Null.

```
void set_nan (MpfrClass& x);
```

Setzt x auf NaN, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_inf  (MpfrClass& x, int k);
```

Setzt x auf ±Inf, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss. Das Vorzeichen wird durch k festgelegt.

```
void set_zero (MpfrClass& x, int k);
```

Setzt x auf ±0, wobei die Präzision von x erhalten bleibt und daher mit der Current-Precision nicht übereinstimmen muss. Das Vorzeichen wird durch k festgelegt.

```
void swap(MpfrClass& x, MpfrClass& y);
```

Tauscht den Wert und die Präzision von x und y.

```
void swap(MpfrClass& x, mpfr_t& y);
```

Tauscht den Wert und die Präzision von x und y.

```
static MpfrClass Pi      (RoundingMode rnd    = CurrRndMode,
                          PrecisionType prec = CurrPrecision);
static MpfrClass Ln2     (RoundingMode rnd    = CurrRndMode,
                          PrecisionType prec = CurrPrecision);
static MpfrClass Euler   (RoundingMode rnd    = CurrRndMode,
                          PrecisionType prec = CurrPrecision);
static MpfrClass Catalan(RoundingMode rnd    = CurrRndMode,
                          PrecisionType prec = CurrPrecision);
```

`Pi(rnd, prec)` rundet π mittels rnd in ein Format der Präzision prec. Wird prec nicht angegeben, so wird mittels rnd in ein Format mit der Current-Precision gerundet. Wird auch rnd weggelassen, so wird mittels des Current-Rundungsmodus in ein Format mit der Current-Precision gerundet. Entsprechendes gilt für die drei anderen Konstanten. `Ln2()` rundet also $\ln(2) = 0.693147\ldots$ mittels des Current-Rundungsmodus in ein Format mit der voreingestellten Current-Precision.

```
void random(MpfrClass& x, gmp_randstate_t state);
```

Geliefert wird ein Zufallszahl $x \subseteq [0, +1)$, wobei x die Präzision des vorher deklarierten Objekts x erhält. Das folgende Programm zeigt eine mögliche Anwendung der Funktion `void random(MpfrClass& x, gmp_randstate_t state)`.

```
1  // MPFR-05.cpp
2  #include "mpfrclass.hpp"
3
4  using namespace MPFR;
5  using namespace std;
6
7  int main(void)
8  {
9    MpfrClass::SetCurrRndMode (RoundNearest);
10   cout << "\nCurrent-RoundingMode = " << MpfrClass::GetCurrRndMode() << endl;
11   MpfrClass::SetCurrPrecision (60);
12   cout << "Current-Precision = " << MpfrClass::GetCurrPrecision() << endl;
13
14   gmp_randstate_t state;          // Declaration of state;
15   gmp_randinit_default (state); // Initialization of state;
16
17   MpfrClass x(0,RoundNearest, 50); // Declaration of class object x
18                                    // with a precision of 50 bits
19   cout.precision(x.GetPrecision()/3.321928095);
20   cout << "x = " << x << endl;
21   cout << "x.GetPrecision() = " << x.GetPrecision() << endl;
22   random (x, state);     // Delivers the first random number x
23   cout << "x = " << x << endl;
24   random (x, state);     // Delivers the second random number x
25   cout << "x = " << x << endl;
26
27   return 0;
28 }
```

Das Programm liefert die Ausgabe:

```
Current-RoundingMode = 0
Current-Precision = 60
x = 0
x.GetPrecision() = 50
x = 3.46961336961373e-1
x = 4.62035403472839e-1
```

Weitere Informationen bez. der Initialisierungsfunktion in Zeile 15 findet man unter

`http://gmplib.org/manual/Random-State-Initialization.html`

3.11.3 Elementarfunktionen

Tabelle 3.1: Elementarfunktionen mit `x,y` vom Typ `MpfrClass`

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(\sin(x))$	`ln_sin(x)`
x^2	`sqr(x)`	$\ln(\cos(x))$	`ln_cos(x)`
x^2+y^2	`x2py2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
x^2-y^2	`x2my2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$1/x$	`reci(x)`	x^y	`pow(x, y)`
$\sqrt{x}$	`sqrt(x)`	$\sin(x)$	`sin(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_n(x)`	$\cos(x)$	`cos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$\tan(x)$	`tan(x)`
$\sqrt[3]{x}$	`cbrt(x)`	$\cot(x)$	`cot(x)`
$\sqrt[n]{x},\ n=2,3,\ldots$	`sqrt(x,n)`	$\arcsin(x)$	`asin(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arctan(y/x)$	`atan2(y,x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$\sinh(x)$	`sinh(x)`
e^x	`exp(x)`	$\cosh(x)$	`cosh(x)`
2^x	`exp2(x)`	$\tanh(x)$	`tanh(x)`
10^x	`exp10(x)`	$\coth(x)$	`coth(x)`
e^x-1	`expm1(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
e^{x^2}	`expx2(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
$e^{x^2}-1$	`expx2m1(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
e^{-x^2}	`expmx2(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
$e^{-x^2}-1$	`expmx2m1(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$\ln(x)$	`ln(x)`	$1/\sin(x)$	`csc(x)`
$\log_2(x)$	`log2(x)`	$1/\cos(x)$	`sec(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\log_{10}(x)$	`log10(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\cosh(x)$	`sech(x)`
$\mathrm{AGM}(x,y)$	`agm(x,y)`	$\ln(\sqrt{(1+x)^2+y^2})$	`ln_sqrtxp1_2y2(x,y)`

Anmerkungen:

1. Mit `sqr(x);` wird der exakte Funktionswert `x`^{2} mit dem Current-Rundungsmodus in die Current-Precision gerundet. Weitere Informationen zur Implementierung findet man auf Seite 30.

2. Mit `sqr(x, rnd);` wird wie unter 1. verfahren, jedoch wird der exakte Funktionswert `x`^{2} jetzt mittels `rnd` in die Current-Precision gerundet, vgl. ebenfalls Seite 30.

3. Alle Funktionen der Tabelle 3.1 sind nach 1. bzw. 2. implementiert. Für `rnd` stehen dabei die folgenden Rundungsmodi zur Verfügung[1]: `RoundNearest, RoundUp, RoundDown`.

4. `abs(x)` kann zusätzlich noch mit einem Präzisionsparameter `prec` aufgerufen werden. Mit `abs(x, rnd, x.GetPrecision());` wird dabei der exakte Absolutbetrag |`x`|, und zwar unabhängig von `rnd`, zurückgegeben, vgl. auch Seite 31.

5. `power(x, k)` kann bez. `k` mit den Datentypen `int, long int` aufgerufen werden.

6. `pow(x, y)` kann bez. `y` mit folgenden Datentypen aufgerufen werden: `MpfrClass, real, double`.

7. Die Funktion `atan2(y,x)` ist wie folgt definiert:

$$\mathrm{atan2}(y,x) = \begin{cases} \arctan(\frac{y}{x}) & x > 0 \\ \pi + \arctan(\frac{y}{x}) & y \geq 0,\ x < 0 \\ -\pi + \arctan(\frac{y}{x}) & y < 0,\ x < 0 \\ \frac{\pi}{2} & y > 0,\ x = 0 \\ \frac{\pi}{2} & y = +\infty,\ x = 0 \\ -\frac{\pi}{2} & y < 0,\ x = 0 \\ -\frac{\pi}{2} & y = -\infty,\ x = 0 \\ 0 & y = 0,\ x = 0 \quad \text{Vorsicht!} \\ 0 & |y| < +\infty,\ x = +\infty \\ \pi & y \geq 0,\ x = -\infty \\ -\pi & y < 0,\ x = -\infty \\ \frac{\pi}{4} & y = +\infty,\ x = +\infty \\ -\frac{\pi}{4} & y = -\infty,\ x = +\infty \\ \frac{3\pi}{4} & y = +\infty,\ x = -\infty \\ -\frac{3\pi}{4} & y = -\infty,\ x = -\infty \\ \mathrm{NaN} & y = \mathrm{NaN} \text{ oder } x = \mathrm{NaN}. \end{cases}$$

[1] Für alle Funktionen, die in MPFR direkt implementiert sind, gibt es bez. `rnd` die zusätzlichen Rundungsmodi: `RoundToZero, RoundFromZero`.

8. Die Funktion `agm`(x, y) rundet das exakte Aritmetisch-Geometrische Mittel AGM(x, y) der beiden `MpfrClass`-Objekte $x, y \geq 0$ mit dem Current-Rundungsmodus optimal in ein `MpfrClass`-Objekt mit der Current-Precision. Der absolute Fehler ist dabei höchstens 0.5 `ulp`. Mit `agm`$(x, y,$`RoundDown`$)$ wird das AGM optimal in die Current-Precision abgerundet, d.h. der absolute Fehler ist kleiner als 1 `ulp`. Das AGM ist bez. x, y streng monoton wachsend und es gilt:

$$0 \leq x \leq \mathrm{AGM}(x, y) = \mathrm{AGM}(y, x),\ \texttt{agm}(x, y) = \texttt{agm}(y, x) \leq \texttt{y}.$$

Das AGM spielt bei der Auswertung elliptischer Integrale eine zentrale Rolle.

3.11.4 Skalarprodukt aus zwei Teilprodukten

Mit den folgenden Funktionen lassen sich Skalarprodukte der Form $a \cdot b + c \cdot d$ ohne vorzeitigen Overflow berechnen, wobei nach Bedarf gerundet werden kann.

```
void prod_H1(MpfrClass& r, long int& k, const MpfrClass& a,
                                        const MpfrClass& b, RoundingMode rnd);
```

r wird zunächst auf die Current-Precision gesetzt, dann wird $a \cdot b$ bez. rnd nach r gerundet, so dass gilt: $r \cdot 2^k \approx a \cdot b$.
Wird `rnd` nicht gesetzt, so wird mit dem voreingestellten Current-Rundungsmodus nach r gerundet. Es gilt: $|r| \in [0.5, 2)$.
Es gilt: `r = 0, r = NaN, r = +Inf, r = -Inf ---> k = 0;`

```
void Prod_H1(MpfrClass& r, long int& k, const MpfrClass& a,
                                        const MpfrClass& b, RoundingMode rnd);
```

r wird zunächst auf die Current-Precision gesetzt, dann wird $a \cdot b$ bez. rnd nach r gerundet, so dass gilt: $r \cdot 2^k \approx a \cdot b$.
Wird `rnd` nicht gesetzt, so wird mit dem voreingestellten Current-Rundungsmodus nach r gerundet. Es gilt: `MaxFloat()`$/8 \le |r| <$ `MaxFloat()`$/2.098...$.
Es gilt außerdem: `r = 0, r = NaN, r = +Inf, r = -Inf ---> k = 0;`

```
void sum_k_H1(MpfrClass& r, long int& k, const MpfrClass& x, long int& kx,
                            const MpfrClass& y, long int& ky, RoundingMode rnd);
```

Voraussetzung: $kx \ge ky$, falls ($x, y \neq 0$ `&& isNumber(x,y)` = True).
`r` wird zunächst auf das Maximum der Präzisionen von `x` und `y` gesetzt. Ist diese kleiner als die Current-Precision, so erhält `r` als Präzision diese Current-Precision. Danach wird durch geeignete Skalierung $x \cdot 2^{kx} + y \cdot 2^{ky} = 2^{kx} \cdot (x + y \cdot 2^{ky-kx})$ die Klammer $(x + y \cdot 2^{ky-kx})$ berechnet und mittels `rnd` nach `r` gerundet. Man erhält dann im Fall $x, y \neq 0$ das Ergebnis: $r \cdot 2^k = r \cdot 2^{kx} \approx x \cdot 2^{kx} + y \cdot 2^{ky}$.
Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus.
Es gilt außerdem: `r = 0, r = NaN, r = +Inf, r = -Inf ---> k = 0;`

```
void scal_prod_k(MpfrClass& r, long int& k,
                               const MpfrClass& a, const MpfrClass& b,
                               const MpfrClass& c, const MpfrClass& d,
                                                   RoundingMode rnd);
```

`r` wird zunächst auf das Maximum der Präzisionen von `a,b,c,d` gesetzt. Ist diese kleiner als die Current-Precision, so erhält `r` als Präzision diese Current-Precision. Danach wird mit Hilfe der obigen Funktionen `prod_H1()` und `sum_k_H1()` das Skalarprodukt $a \cdot b + c \cdot d = r \cdot 2^k$ mittels `rnd` nach r gerundet und entsprechend k berechnet. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus.

```
void Scal_prod_k(MpfrClass& r, long int& k,
                               const MpfrClass& a, const MpfrClass& b,
                               const MpfrClass& c, const MpfrClass& d,
                                                   RoundingMode rnd);
```

`r` wird zunächst auf das Maximum der Präzisionen von `a,b,c,d` gesetzt. Ist diese kleiner als die Current-Precision, so erhält `r` als Präzision diese Current-Precision. Um Auslöschung möglichst zu vermeiden, wird dann diese Präzision von `r` noch einmal mehr als verdoppelt. Zusätzlich wird dann die Präzision um den Betrag der Zweier-Exponenten-Differenz vergrößert, um in dieser Präzision auch noch die Summe $a \cdot b + c \cdot d$ rundungsfehlerfrei berechnen zu können. Danach wird mit Hilfe der obigen Funktionen `Prod_H1()` und `sum_k_H1()` das Skalarprodukt $a \cdot b + c \cdot d = r \cdot 2^k$ mittels `rnd` nach r gerundet und entsprechend k berechnet. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus.

```
void scal_prod(MpfrClass& r, const MpfrClass& a, const MpfrClass& b,
               const MpfrClass& c, const MpfrClass& d,
                                   RoundingMode rnd);
```

$a \cdot b + c \cdot d$ wird intern mit `Skal_prod_k()` in so hoher Präzision berechnet, dass es danach mit `rnd` nach `r` höchstens einmal in die Current-Präzision gerundet werden muss. Mit `rnd = RoundDown` und `rnd = RoundUp` erhält man damit für $a \cdot b + c \cdot d$ eine maximalgenau Einschließung. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus. Ein vorzeitiger interner Über- oder Unterlauf ist ausgeschlossen.

```
MpfrClass Compl_Re (const MpfrClass& a, const MpfrClass& b,
                    const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

Rundet den Realteil von $(a + i \cdot b)/(x + i \cdot y)$ in die Current-Precision und gibt diesen gerundeten Wert zurück. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus. Ein vorzeitiger interner Über- oder Unterlauf wird vermieden.

```
MpfrClass Compl_Im (const MpfrClass& a, const MpfrClass& b,
                    const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

Rundet den Imaginärteil von $(a + i \cdot b)/(x + i \cdot y)$ in die Current-Precision und gibt diesen gerundeten Wert zurück. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus. Ein vorzeitiger interner Über- oder Unterlauf wird vermieden.

```
MpfrClass plus_ab (const MpfrClass& x, const MpfrClass& a,
                                       const MpfrClass& b, RoundingMode rnd);
```

$(x + a \cdot b)$ wird bez. `rnd` in den Rückgabewert mit der Current-Precision gerundet. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus. Durch zweimalige Anwendung dieser Funktion:

```
x = plus_ab(0, a, b);     y = plus_ab(x, c, d);
```

wird das Skalarprodukt $a \cdot b + c \cdot d$ jetzt mit bis zu zwei Rundungen berechnet, und im Gegensatz zur Funktion `scal_prod()` kann ein vorzeitiger Überlauf nicht vermieden werden. Die Funktion `scal_prod()` benötigt darüber hinaus höchstens eine Rundung und liefert damit für $a \cdot b + c \cdot d$ i.a. die **bessere** Näherung.

3.11.4.1 Beispiel

Als Beispiel für die Berechnung eines Skalarprodukts der Form $x = a \cdot b + c \cdot d$ wählen wir:

$$a := \texttt{MaxFloat()}, \quad b := e^1 \cdot 2^k, \; k \in \mathbb{Z}, \quad c := -b, \quad d := \texttt{pred}(a), \quad \text{und damit gilt:}$$

$$x := a \cdot b + c \cdot d = e^1 \cdot 2^k (a - \texttt{pred}(a)) =: y, \tag{3.3}$$

wobei man zur Kontrolle $y = e^1 \cdot 2^k (a - \texttt{pred}(a))$ auch ohne Skalarprodukt berechnen kann.

```
1  // MPFR-03.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "mpfrclass.hpp"
5
6  using namespace MPFR;
7  using namespace std;
8
9  int main(void)
10 {
11   MpfrClass::SetCurrRndMode (RoundNearest);
12   cout << "\nCurrent-RoundingMode = " << MpfrClass::GetCurrRndMode() << endl;
13   MpfrClass::SetCurrPrecision (5000000);
14   cout << "Current-Precision = " << MpfrClass::GetCurrPrecision() << endl;
15
16   MpfrClass a,b,c,d,x,y;
17
18   long int k = -2; // 1234567;
19   a = MaxFloat(); b = exp(MpfrClass(1)); times2pown(b, k);
20   c = -b;  d = pred(a);
21
22   y = b*(a-pred(a));
23   scal_prod (x, a, b, c, d, RoundDown);
24
25   cout.precision();
26   cout << "x = " << x << endl;
27
28   if (x==y) cout << "Kontrollrechnung:  x == y" << endl;
29   else cout << "x != y" << endl;
30
31   y = 0;
32   y = plus_ab(y, a, b, RoundDown);
33   x = plus_ab(y, c, d, RoundDown);
34
35   cout << "Zweimalige Anwendg. von  plus_ab(), x = " << x << endl;
36
37   return 0;
38 }
```

Mit `RoundDown` und `RoundUp` in Zeile 23 liefert das Programm `MPFR-03.cpp` die folgende Ausgabe

```
Current-RoundingMode = 0
Current-Precision = 5000000
x = 1.49913e321723346
Kontrollrechnung:  x == y
Zweimalige Anwendg. von  plus_ab(), x = 7.92265e321723345
```

Man erkennt, dass durch die zweimalige Anwendung von `plus_ab(..., RoundDown)`, verbunden mit mindestens einer Rundung (`RoundDown`), im Vergleich zum exakten Wert `x` eine deutlich kleinerer Näherung `y` für das Skalarprodukt berechnet wird. Die Ausgabe der Kontrollrechnung zeigt, dass das Skalarprodukt mit `scal_prod (x, a, b, c, d, RoundDown);` sogar exakt berechnet wurde. Dies wird auch durch die Rechnung mit `RoundUp` in Zeile 23 bestätigt.

Wählt man im Programm in Zeile 18 `k = 1234567;` so werden beide Teilprodukte $a \cdot b$ und $c \cdot d$, jeweils einzeln berechnet, einen Overflow erzeugen. Das Programm liefert jedoch die Ausgabe

```
Current-RoundingMode = 0
Current-Precision = 5000000
x = 2.99610e322094988
Kontrollrechnung:  x == y
Zweimalige Anwendg. von  plus_ab(), x = -@Inf@
```

Mit `scal_prod (x, a, b, c, d, RoundDown);` erhält man wieder das exakte Skalarprodukt `x` $= 2.996\ldots \cdot 10^{322094988}$, jetzt jedoch ohne vorzeitigen internen Overflow, der aber schon beim ersten Aufruf der Funktion `plus_ab(...,RoundDown)` natürlich nicht vermieden werden kann!

Anmerkungen:

- Wählt man im obigen Programm die `CurrentPrecision` mit `prec` = 500000 um den Faktor 10 kleiner, so wird das exakte Skalarprodukt zu groß und kann wegen notwendiger Rundungen nicht mehr exakt berechnet werden, d.h. `x` erhält in Abhängigkeit von `RoundUp` und `RoundDown` **verschiede** Werte. Beachten Sie in diesem Zusammenhang, dass der Faktor $(a - \texttt{pred}(a))$ im Ausdruck für y nur dann hinreichend klein wird, wenn man die Current-Precision hinreichend groß gewählt.

- Der Ausdruck $b \cdot (a - \texttt{pred}(a))$ kann rundungsfehlerfrei berechnet werden, da $a - \texttt{pred}(a)$ die Binärdarstellung $1.0000e....$ besitzt und damit eine reine Zweierpotenz ist, mit der b rundungsfehlerfrei multipliziert werden kann, solange dabei kein Über- oder Unterlauf entsteht.

3.11.5 Funktionen der Mathematischen Physik

Tabelle 3.2: Funktionen der Mathematischen Physik mit `x` vom Typ `MpfrClass`

Funktionsterm	Aufruf	Anmerkung
$\mathrm{erf}(x) = \frac{2}{\pi}\int_0^x e^{-t^2}dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\pi}\int_x^\infty e^{-t^2}dt$	`erfc(x)`	monoton fallend
$\Gamma(x) = \int_0^\infty t^{x-1}\mathrm{e}^{-t}dt,\ x>0$	`gamma(x)`	Pole: x=0, -1, -2, ...
$\Gamma'(x)$	`gamma_D(x)`	Pole: x=0, -1, -2, ...
$\frac{1}{\Gamma(x)}$	`gamma_reci(x)`	überall differenzierbar
$\left(\frac{1}{\Gamma(x)}\right)'$	`gamma_reci_D(x)`	überall differenzierbar
$\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$	`digamma(x)`	Pole: x=0, -1, -2, ...
$\ln(\,\Gamma(x)\,)$	`lngamma(x)`	$2k-1\le$ x $\le -2k \rightsquigarrow$ NaN, k=0,1,2,...
$\ln(\,\lvert\Gamma(x)\rvert\,)$	`int k; lgamma(x,k);`	$k = \begin{cases} +1, & \Gamma(x)>0 \\ -1, & \Gamma(x)<0 \end{cases}$
$k!$	`factorial(k)`	unsigned long int k
$\zeta(x) = \sum_{k=1}^\infty k^{-x},\ x>1;$	`zeta(x)`	x $\neq +1$
$\zeta(k),\ k = 0,2,3,4,...$	`zeta(k)`	unsigned long int k
$\mathrm{Ei}(x) = \gamma + \ln(x) + \sum_{k=1}^\infty \frac{x^k}{k\cdot k!},\ x>0;$	`Ei(x)`	x=0 $\rightsquigarrow$ `-Inf`; x<0 $\rightsquigarrow$ `NaN`;
$\mathrm{Li2}(x) = -\int_0^x \frac{\ln(1-t)}{t}\,dt,\ x<1;$	`Li2(x)`	$x>1$ $\rightsquigarrow$ Nur Realteil!
$\mathrm{J}_n(x) = \sum_{k=0}^\infty \frac{(-1)^k}{k!\Gamma(k+n+1)}\left(\frac{x}{2}\right)^{2k+n};$	`Jn(n,x)`	Bessel-Fkt. 1. Art; $n\in\mathbb{Z}$;
$\mathrm{J}_0(x);$	`J0(x)`	Bessel-Fkt. 1. Art;
$\mathrm{J}_1(x);$	`J1(x)`	Bessel-Fkt. 1. Art;
$\mathrm{Y}_n(x);$	`Yn(n,x)`	Bessel-Fkt. 2. Art; $n\in\mathbb{Z}$;
$\mathrm{Y}_0(x);$	`Y0(x)`	Bessel-Fkt. 2. Art;
$\mathrm{Y}_1(x);$	`Y1(x)`	Bessel-Fkt. 2. Art;

Anmerkungen:

1. Alle Funktionen aus Tabelle 3.2 können mit einem zusätzlichen Rundungsparameter `rnd` aufgerufen werden, womit eine vom Current-Rundungsmodus abweichende Rundung realisiert werden kann. Bezüglich der Implementierung gelten die gleichen Anmerkungen wie auf Seite 30.

2. Die Funktion $\mathrm{Li2}(x)$ ist für $x < 1$ zunächst definiert durch

$$\mathrm{Li2}(x) := -\int_0^x \frac{\ln(1-t)}{t}\,dt = \sum_{k=1}^{\infty} \frac{x^k}{k^2}, \quad x < 1;$$

   Durch analytische Fortsetzung lässt sich diese Definition auf weitere $z \in \mathbb{C}$ in der ab $x = 1$ längs der positiven reellen Achse aufgeschnittenen konplexen Ebene ausdehnen. Für $x > 1$ liefert `Li2(x)` dabei nur den Realteil der dann komplexwertigen Funktion. Der Imaginärteil kann mit Hilfe der Beziehung

$$\mathrm{Li2}(x) = -\mathrm{Li2}(1/x) + \frac{\pi^2}{3} - \frac{1}{2}\ln^2(x) - i \cdot \pi \ln(x), \quad x > 1;$$

   einfach berechnet werden, da $\pi \ln(x)$ für $x > 1$ positiv und streng monoton wachsend ist.

   Den Zusammenhang mit der Polylogarithmus-Funktion $\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$ findet man unter

   `http://de.wikipedia.org/wiki/Polylogarithmus`

   `http://en.wikipedia.org/wiki/Polylogarithm`

3. Die Besselfunktionen 2. Art $\mathrm{Y}_n(x)$ (Neumann-Funktionen) definiert man zunächst für nichtganzzahlige ν als Linearkombination von J_ν und $\mathrm{J}_{-\nu}$

$$\mathrm{Y}_\nu(x) := \frac{\mathrm{J}_\nu(x) \cdot \cos \pi\nu - \mathrm{J}_{-\nu}(x)}{\sin \pi\nu}, \quad \nu \notin \mathbb{Z},$$

   und bildet für $n \in \mathbb{Z}$ die Funktionen $\mathrm{Y}_n(x)$ durch Grenzübergang

$$\mathrm{Y}_n(x) := \lim_{\nu \to n} \mathrm{Y}_\nu(x), \quad n \in \mathbb{Z};$$

   Dies ist lediglich eine Anmerkung zur Definition, nicht aber zur numerischen Auswertung von $\mathrm{Y}_n(x)$. Die Funktionen $\mathrm{Y}_n(x)$ und $\mathrm{J}_n(x)$ sind für festes $n \in \mathbb{Z}$ linear unabhängig und bilden damit eine Basis für die Lösungen der Besselschen Differentialgleichung.

4 MpfiClass-Interface zur Anbindung der MPFI-Bibliothek an C-XSC

4.1 MPFI-Bibliothek

4.1.1 Entwickler

Die MPFI-Bibliothek wurde 2002 von der INRIA-Gruppe an der Universität von Lyon entwickelt. Die Gruppe besteht aus Nathalie Revol, Fabrice Rouillier, Sylvain Chevillard, Hong Diep NGUYEN und Christoph Lauter. Unterstützt wurden sie von den Entwicklern der MPFR-Bibliothek, zu der auch Nathalie Revol gehört.

4.1.2 Allgemein

Die MPFI-Bibliothek wurde für langzahlige Intervallberechnungen entwickelt und ist nur in C implementiert. Sie basiert auf der GMP- und MPFR-Bibliothek. Die Bibliothek entstand im Jahre 2002 und wird seitdem immer weiter entwickelt. Die aktuelle Version ist "1.5". Um die Bibliothek nutzen zu können, müssen die GMP-Bibliothek (Version 4.1 oder höher) und die MPFR-Bibliothek (Version 3.0.0 oder höher) vorhanden sein. Da die Bibliothek auf der MPFR-Bibliothek basiert, profitiert MPFI von den korrekten Rundungen der MPFR-Bibliothek. Die Bibliothek befolgt den IEEE-754-Standard für Gleitkommaarithmetik.

Intern wird die Bibliothek mit Hilfe von zwei MPFR-Variablen realisiert. Die beiden Variablen repräsentieren die beiden Endpunkte des Intervalls. Ein Intervall kann endliche oder unendliche Endpunkte haben. Ebenso kann ein Endpunkt - oder beide - @NaN@ sein. Dies zeigt an, dass eine unzulässige Berechnung durchgeführt wurde. Ein leeres Intervall ist dadurch definiert, dass der rechte Endpunkt kleiner als der linke Endpunkt ist.

4.1.3 Installation

Die Installation der MPFI-Bibliothek ist nur unter einem Linux/Unix-System möglich und erfolgt völlig analog zur Beschreibung auf Seite 11. Die aktuelle Version der MPFI-Bibliothek kann unter:

`http://gforge.inria.fr/projects/mpfi/`

bezogen werden.

4.2 Grundlegendes

Das MpfiClass-Interface ist eine in `mpficlass.hpp` und `mpficlass.cpp` implementierte **C++**-Wrapper-Klasse MpfiClass für die C-Bibliothek MPFI, deren C-Funktionen über die implementierten Operatoren und Funktionen aufgerufen werden. Alle Funktionen mit der möglichen Übergabe eines RoundingModes oder eines PrecisionTypes haben als Default-Wert den Wert von CurrRndMode bzw. CurrPrecision, die beide beliebig gesetzt werden können. Dies gilt in vielen Fällen auch für die Konstruktoren.

4.2.1 Allgemein

Um in C-XSC das Interface verwenden zu können, muss der Header mpficlass.hpp eingebunden werden. In ihm sind die benötigten Header-Dateien der MPFI-Bibliothek enthalten. Die MpfiClass-Klasse liegt im Namensraum "MPFI".

4.2.2 Aufbau

Die Klasse besteht intern aus einer `mpfi_t`-Variablen. Diese dient zum Speichern eines Intervalls. Zusätzlich gibt es static Elemente, um den Standard-Rundungmodus, die Standard-Präzision und die aktuelle Basis zu speichern.

4.2.3 Präzision

Die Präzision gibt die Anzahl der binären Mantissenstellen einer MpfiClass-Variablen an. Der Wert muss mindestens 2 betragen. Die Current-Precision kann global gesetzt werden; wenn dies nicht geschieht, so wird die Default-Precision mit 53 Bits benutzt. Unabhängig davon kann die Präzision für jede MpfiClass-Variable auch einzeln festgelegt werden.

4.2.4 Variablentyp PrecisionType

Mit Hilfe des Variablentyps `PrecisionType`, dessen Variablen i.a. mit prec bezeichnet werden, kann der Präzisionswert einer MpfiClass-Variablen eingestellt werden. Der Variablentyp ist ein typedef für eine `mpfr_prec_t`-Variable.

4.3 Konstruktoren / Destruktoren

4.3.1 Konstruktoren

```
MpfiClass ();
```

Der Default-Konstruktor legt ein neues Intervall-Objekt mit Current-Precision an.
Aufruf: `MpfiClass y;`
Es wird kein Wert initialisiert, in beiden Fällen gilt: `y` = [NaN, NaN];

Alle folgenden Konstruktoren erzeugen ein `MpfiClass`-Intervall `y` mit der Präzision prec, das die jeweiligen Werte op bzw. das Intervall `[op1,op2]` optimal einschließen. Wird beim Konstruktoraufruf prec nicht angegeben, so erhält `y` die voreingestellte Current-Precision. Falls diese mit `SetCurrPrecision` nicht gesetzt wurde, erhält `y` die Präzision *prec* = 53.

```
MpfiClass (const MpfiClass& op,          PrecisionType prec);
MpfiClass (const mpfi_t& op,             PrecisionType prec);
MpfiClass (const MPFR::MpfrClass& op,    PrecisionType prec);
MpfiClass (const mpfr_t& op,             PrecisionType prec);
MpfiClass (const cxsc::interval& op,     PrecisionType prec);
MpfiClass (const cxsc::real& op,         PrecisionType prec);
MpfiClass (const double& op,             PrecisionType prec);
MpfiClass (int op,                       PrecisionType prec);
MpfiClass (const MPFR::MpfrClass& op1,
           const MPFR::MpfrClass& op2,   PrecisionType prec);
```

```
MpfiClass (const std::string& op, PrecisionType prec);
```

Auch hier gelten sinngemäß die gleichen Anmerkungen wie für die oberen neun Konstruktoren, wobei unter dem Wert von op der Dezimalwert des Strings zu verstehen ist. Mit den beiden Strings op = `"[1.0e-1, 1.0e-1]"` bzw. op = `"1.0e-1"` liefert der Konstruktoraufruf `MpfiClass y(op);` mit der voreingestelleten Current-Precision Einschließungen des Dezimalwertes 0.1, wobei zu beachten ist, dass 0.1 im binären System nicht darstellbar ist, so dass alle Einschließungen von 0.1 auch mit noch so großen prec-Werten keine Punktintervalle sein können. Der Konstruktoraufruf `MpfiClass y(op, 140);` liefert eine Einschließung von 0.1 mit einer Präzision von 140 Bits, wobei auch jetzt 0.1 natürlich nicht durch eine Punktintervall eingeschlossen werden kann.

4.3.2 Destruktor

```
~MpfiClass ();
```

Der Speicher für das Objekt wird freigegeben. Dieser Destruktor muss nicht explizit aufgerufen werden!

4.4 Zuweisungs-Operatoren

Unabhängig von der Präzision des rechten Operanden erhält bei allen folgenden Zuweisungsoperatoren der linke Operand `y` stets die Current-Precision und schließt die jeweiligen op-Werte optimal ein.

```
MpfiClass& operator = (const MpfiClass& op);
MpfiClass& operator = (const MPFR::MpfrClass& op);
MpfiClass& operator = (const mpfi_t& op);
MpfiClass& operator = (const mpfr_t& op);
MpfiClass& operator = (const int& op);
MpfiClass& operator = (const cxsc::real& op);
MpfiClass& operator = (const double& op);
MpfiClass& operator = (const cxsc::interval& op);
MpfiClass& operator = (const std::string& op);
```

Anmerkunken:

- Ist z.B. op ein `real`-Wert und wurde die Current-Precision mit `SetCurrPrecision` zu klein gewählt, so ist der linke Operand `y` i.a. kein Punktinterval. Ist die Current-Precision jedoch größer oder gleich 53, so ist das einschließende Intervall `y` stets ein Punktintervall.
- Ist op ein String, z.B. op = "[0.1,0.1]" oder op = "0.1", so kann `y` bei noch so großer Current-Precision kein Punktintervall sein, da 0.1 im vorliegenden Binärsystem nicht exakt darstellbar ist.

4.5 Arithmetische Operatoren

Für alle arithmetischen Intervalloperationen gilt:

> Das exakte Ergebnis einer arithmetischen Intervalloperation wird unabhängig von der Präzision der Operanden mit der voreingestellten Current-Präzision optimal eingeschlossen.
> Die Operatoren $\circledcirc =$, mit $\circledcirc \in \{+, -, \cdot, /\}$, bedeuten $u \circledcirc = v \iff u = u \circledcirc v$. Dabei wird $u \circledcirc v$ optimal durch u eingeschlossen, wobei u als Präzision die Current-Präzision erhält.

4.5.1 Addition

```
MpfiClass operator + (const MpfiClass& op1, const MpfiClass& op2);
MpfiClass operator + (const MpfiClass& op1, const double& op2);
MpfiClass operator + (const MpfiClass& op1, const cxsc::real& op2);
MpfiClass operator + (const MpfiClass& op1, const cxsc::interval& op2);
MpfiClass operator + (const MpfiClass& op1, const int op2);
MpfiClass operator + (const MpfiClass& op1, const mpfr_t& op2);
MpfiClass operator + (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator + (const MpfiClass& op1, const mpfi_t& op2);
MpfiClass operator + (const double& op1, const MpfiClass& op2);
MpfiClass operator + (const cxsc::real& op1,     const MpfiClass& op2);
MpfiClass operator + (const cxsc::interval& op1, const MpfiClass& op2);
MpfiClass operator + (const int op1,             const MpfiClass& op2);
MpfiClass operator + (const mpfr_t& op1,         const MpfiClass& op2);
MpfiClass operator + (const MPFR::MpfrClass& op1, const MpfiClass& op2);
MpfiClass operator + (const mpfi_t& op1,         const MpfiClass& op2);
```

```
MpfiClass& operator += (MpfiClass& op1, const MpfiClass& op2);
MpfiClass& operator += (MpfiClass& op1, const double& op2);
MpfiClass& operator += (MpfiClass& op1, const cxsc::real& op2);
MpfiClass& operator += (MpfiClass& op1, const cxsc::interval& op2);
MpfiClass& operator += (MpfiClass& op1, const int op2);
MpfiClass& operator += (MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass& operator += (MpfiClass& op1, const mpfr_t& op2);
MpfiClass& operator += (MpfiClass& op1, const mpfi_t& op2);
```

4.5.2 Subtraktion

Beachten Sie den Hinweis auf Seite 49.

```
MpfiClass operator - (const MpfiClass& op1, const MpfiClass& op2);
MpfiClass operator - (const MpfiClass& op1, const double& op2);
MpfiClass operator - (const MpfiClass& op1, const cxsc::real& op2);
MpfiClass operator - (const MpfiClass& op1, const cxsc::interval& op2);
MpfiClass operator - (const MpfiClass& op1, const int op2);
MpfiClass operator - (const MpfiClass& op1, const mpfr_t& op2);
MpfiClass operator - (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator - (const MpfiClass& op1, const mpfi_t& op2);
MpfiClass operator - (const double& op1, const MpfiClass& op2);
MpfiClass operator - (const cxsc::real& op1,     const MpfiClass& op2);
MpfiClass operator - (const cxsc::interval& op1, const MpfiClass& op2);
MpfiClass operator - (const int op1,             const MpfiClass& op2);
MpfiClass operator - (const mpfr_t& op1,         const MpfiClass& op2);
MpfiClass operator - (const MPFR::MpfrClass& op1, const MpfiClass& op2);
MpfiClass operator - (const mpfi_t& op1,         const MpfiClass& op2);
```

```
MpfiClass& operator -= (MpfiClass& op1, const MpfiClass& op2);
MpfiClass& operator -= (MpfiClass& op1, const double& op2);
MpfiClass& operator -= (MpfiClass& op1, const cxsc::real& op2);
MpfiClass& operator -= (MpfiClass& op1, const cxsc::interval& op2);
MpfiClass& operator -= (MpfiClass& op1, const int op2);
MpfiClass& operator -= (MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass& operator -= (MpfiClass& op1, const mpfr_t& op2);
MpfiClass& operator -= (MpfiClass& op1, const mpfi_t& op2);
```

4.5.3 Multiplikation

Beachten Sie den Hinweis auf Seite 49.

```
MpfiClass operator * (const MpfiClass& op1, const MpfiClass& op2);
MpfiClass operator * (const MpfiClass& op1, const double& op2);
MpfiClass operator * (const MpfiClass& op1, const cxsc::real& op2);
MpfiClass operator * (const MpfiClass& op1, const cxsc::interval& op2);
MpfiClass operator * (const MpfiClass& op1, const int op2);
MpfiClass operator * (const MpfiClass& op1, const mpfr_t& op2);
MpfiClass operator * (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator * (const MpfiClass& op1, const mpfi_t& op2);
MpfiClass operator * (const double& op1, const MpfiClass& op2);
MpfiClass operator * (const cxsc::real& op1,     const MpfiClass& op2);
MpfiClass operator * (const cxsc::interval& op1, const MpfiClass& op2);
MpfiClass operator * (const int op1,             const MpfiClass& op2);
MpfiClass operator * (const mpfr_t& op1,         const MpfiClass& op2);
MpfiClass operator * (const MPFR::MpfrClass& op1, const MpfiClass& op2);
MpfiClass operator * (const mpfi_t& op1,         const MpfiClass& op2);
```

```
MpfiClass& operator *= (MpfiClass& op1, const MpfiClass& op2);
MpfiClass& operator *= (MpfiClass& op1, const double& op2);
MpfiClass& operator *= (MpfiClass& op1, const cxsc::real& op2);
MpfiClass& operator *= (MpfiClass& op1, const cxsc::interval& op2);
MpfiClass& operator *= (MpfiClass& op1, const int op2);
MpfiClass& operator *= (MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass& operator *= (MpfiClass& op1, const mpfr_t& op2);
MpfiClass& operator *= (MpfiClass& op1, const mpfi_t& op2);
```

4.5.4 Division

Beachten Sie den Hinweis auf Seite 49.

```
MpfiClass operator / (const MpfiClass& op1, const MpfiClass& op2);
MpfiClass operator / (const MpfiClass& op1, const double& op2);
MpfiClass operator / (const MpfiClass& op1, const cxsc::real& op2);
MpfiClass operator / (const MpfiClass& op1, const cxsc::interval& op2);
MpfiClass operator / (const MpfiClass& op1, const int op2);
MpfiClass operator / (const MpfiClass& op1, const mpfr_t& op2);
MpfiClass operator / (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator / (const MpfiClass& op1, const mpfi_t& op2);
MpfiClass operator / (const double& op1, const MpfiClass& op2);
MpfiClass operator / (const cxsc::real& op1,     const MpfiClass& op2);
MpfiClass operator / (const cxsc::interval& op1, const MpfiClass& op2);
MpfiClass operator / (const int op1,             const MpfiClass& op2);
MpfiClass operator / (const mpfr_t& op1,         const MpfiClass& op2);
MpfiClass operator / (const MPFR::MpfrClass& op1, const MpfiClass& op2);
MpfiClass operator / (const mpfi_t& op1,         const MpfiClass& op2);
```

```
MpfiClass& operator /= (MpfiClass& op1, const MpfiClass& op2);
MpfiClass& operator /= (MpfiClass& op1, const double& op2);
MpfiClass& operator /= (MpfiClass& op1, const cxsc::real& op2);
MpfiClass& operator /= (MpfiClass& op1, const cxsc::interval& op2);
MpfiClass& operator /= (MpfiClass& op1, const int op2);
MpfiClass& operator /= (MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass& operator /= (MpfiClass& op1, const mpfr_t& op2);
MpfiClass& operator /= (MpfiClass& op1, const mpfi_t& op2);
```

4.6 Eingabe / Ausgabe

```
std::ostream& operator << (std::ostream& os, const MpfiClass& x);
```

Ermöglicht die Ausgabe einer MpfiClass-Variablen x über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(x.GetPrecision()/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Der Wert der Variablen x wird dabei in der mit `SetBase(k)` voreingestellten Basis ausgegeben. Für die meist dezimale Ausgabe ist natürlich `k` = 10 zu wählen.

```
std::ostream& operator << (std::ostream& os, mpfi_t& x);
```

Ermöglicht die Ausgabe einer Variablen x vom Typ `mpfi_t` über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(mpfi_get_prec(x)/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Der Intervallwert x wird dabei in der mit `SetBase(k)` voreingestellten Basis ausgegeben. Für die meist dezimale Ausgabe ist `k` = 10 zu wählen.

```
std::istream& operator >> (std::istream& is, MpfiClass& x);
```

Ermöglicht das Einlesen einer MpfiClass-Variablen über den Standard-Eingabestrom "cin". Das eingegebene Intervall ist auf keine Stellenanzahl begrenzt. Folgende Eingabeformate sind zulässig:

- `[-1.23e-4401,2.3E+2000]` Vorsicht: Leerzeichen sind nicht erlaubt!
- `1.1`

Mit der letzten Eingabe entsteht ein nicht-punktförmiges Intervall, das die nicht-darstellbare Dezimalzahl 1.1 bezüglich der Current-Precision optimal einschließt. Die Zeichenkette muss mit der voreingestellten Basis übereinstimmen, sonst entsteht eine Fehlermeldung.

4.7 Base und Precision Handling

```
PrecisionType GetPrecision () const;
```

Diese Memberfunktion gibt die Präzision des aktuellen Objekts in Bits zurück. Als Beispiel entsprechen dabei 302 Bits $302/\log_2(10) \approx 91$ Dezimalstellen.

```
void SetPrecision (PrecisionType prec);
```

Diese Memberfunktion setzt die Präzision des aktuellen Objektes auf prec. Sein Wert bleibt nicht erhalten.

```
void RoundPrecision (PrecisionType prec, RoundingMode rnd);
```

Diese Memberfunktion schließt das aktuelle Objekt mit der neue Precision prec ein. Sollte die Präzision des ursprünglichen Objektes größer sein als prec, so erhält man eine gröbere Einschließung. Ist die Präzision jedoch kleiner als prec, so werden die restlichen binären Stellen mit Nullen aufgefüllt, so dass die Werte der Intervallrandpunkte erhalten bleiben.

```
static const PrecisionType GetCurrPrecision ();
```

Gibt die aktuelle Current-Precision in Bits zurück.

```
static void SetCurrPrecision (PrecisionType prec);
```

Setzt die Current-Precision in `MpfrClass` auf prec. Diese Current-Precision wird dann automatisch auch in der Klasse `MpfiClass` benutzt. Wird die Current-Precision nicht gesetzt, so wird in beiden Klassen mit der Default-Precision von 53 Bits gerechnet. Das Setzen der Current-Precision hat auf die Präzision der bis dahin benutzten Variablen **keinerlei** Einfluss.

```
static const int GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen.

```
static void SetBase (int b);
```

Setzt die aktuelle Basis auf b. Dies muss ein Wert zwischen 2 und 36 sein. Die Basis hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die internen Rechnungen erfolgen stets im Binärsystem!

4.8 Typ-Umwandlungen

Um ein flexibles Arbeiten zwischen C-XSC und der MPFI-Bibliothek zu ermöglichen, wurden möglichst viele Typ-Umwandlungs-Funktionen bereitgestellt:

4.8.1 `real, double, ...` → MPFI

```
MpfiClass real2Mpfi      (const cxsc::real& op);
MpfiClass double2Mpfi    (const double& op);
MpfiClass int2Mpfi       (const int& op);
MpfiClass MpfrClass2Mpfi (const MPFR::MpfrClass& op);
MpfiClass mpfr_t2Mpfi    (const mpfr_t& op);
MpfiClass mpfi_t2Mpfi    (const mpfi_t& op);
MpfiClass interval2Mpfi  (const cxsc::interval& op);
```

Obige Funktionen liefern mit dem jeweiligen Eingabewert op einen Rückgabewert vom Typ `MpfiClass` in einer Präzision, die gewährleistet, dass der Rückgabewert genau dem Wert von op entspricht. Die Präzision des Rückgabewertes wird also i.a. nicht mit der Current-Precision übereinstimmen! Die obigen sieben Funktionen kommen u.a. bei den Vergleichsoperatoren zur Anwendung.

4.8.2 MPFI → `interval`

Die folgende Funktion liefert mit einem Objekt op vom Typ `MpfiClass` eine i.a. **gerundete** Einschließung von op vom C-XSC Typ `interval`;

```
cxsc::interval to_interval(const MpfiClass& op);
```

Eine optimale Einschließung von op wird nur erreicht, wenn op im IEEE-System darstellbar ist.

4.8.3 MPFI → `mpfi_t`

```
const mpfi_t& getvalue(const MpfiClass& v)
```

Obige Funktion liefert von einem als `const` deklarierten Objekt v eine Referenz auf den Wert seines Attributs `mpfi_rep` vom Typ `mpfi_t`.
Anwendung: `const`-Parameter in einer Konstruktor-Parameterliste, vgl. z.B. den Konstruktor `MpfciClass::MpfciClass (const MpfiClass& x, PrecisionType prec)` in der Datei `mpfciclass.cpp`.

4.8.4 MPFI ↔ `mpfi_t`

```
mpfi_t& MpfiClass::GetValue();
```

Mithilfe der Memberfunktion `GetValue()`, die eine Referenz auf den Wert `mpfi_rep` vom Typ `mpfi_t` des aktuellen Objekts liefert, können sowohl `MpfiClass`-Objekte an eine Funktion übergeben als auch referenzierte Rückgabewerte vom Typ `mpfi_t` von einer Funktion übernommen werden. Mithilfe von `GetValue()` kann man daher Funktionen mit referenzierten `mpfi_t`-Parametern einfach aufrufen, vergleiche dazu das Programm `MPFR-02` auf Seite 28, in dem die analoge Übergabe an referenzierte Parameter vom Typ `mpfr_t` gezeigt wird.

4.8.5 `mpfi_t` → MPFI

```
void SetValue(const mpfi_t& t);
```

Mithilfe der Memberfunktion `SetValue()` wird der `mpfi_rep`-Wert des aktuellen Objekts auf `t` gesetzt, wobei `mpfi_rep` die Präzision von `t` übernimmt, d.h. der Wert von `t` wird ohne Rundung exakt übernommen.

4.8.6 `string` → MPFI

```
MpfiClass string2Mpfi(const std::string& op, PrecisionType prec);
```

Der Aufruf `string2Mpfi(op);` rundet den String op mittels der voreingestellten Current-Precision in ein Klassenobjekt vom Typ `MpfiClass`. Der zweite mögliche Aufruf `string2Mpfr(op,140);` rundet op ebenfalls in ein Klassenobjekt vom Typ `MpfiClass` der Präzision 140 Bits. Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in ein binäres Format konvertiert werden kann, so dass i.a. Rundungen nicht zu vermeiden sind, es werden jedoch stets Einschließungen von op zurückgegeben. Eine weitere Möglichkeit, einen String in ein `MpfiClass`-Objekt zu verwandeln, besteht in einem entsprechenden Konstruktor-Aufruf, vgl. Seite 47. Dort findet man auch Hinweise auf mögliche Intervall-String-Formate.

4.8.7 MPFI → `string`

```
std::string to_string (const MpfiClass& x, PrecisionType prec);
```

x wird in einen String `s` mit prec Dezimalstellen gerundet, wenn Base gleich 10 ist, dabei ist der String stets eine optimale Einschließung von x. Wählt man prec hinreichend groß, so stellt der String den Wert von x sogar **exakt** dar, weil eine binäre Zahl stets exakt in eine Dezimalzahl umgewandelt werden kann.

Wird prec nicht angegeben, so wird x mit einem String eingeschlossen, der im Fall Base=10 soviele Dezimalstellen besitzt, wie es der Präzision von `x` entspricht. Ist die Präzision von `x` z.B. 302, so wird ein String von $302/\log_2(10) \approx 302/3.321928095 \approx 91$ Dezimalstellen generiert.

4.8.8 MPFI → MPFI

Beachten Sie, dass bei einer Wertzuweisung an eine `MpfiClass`-Variable mit dem Operator `=` der linke Operand stets auf die Current-Precision gesetzt wird und dass der rechte `MpfiClass`-Operand dabei vom linken Operanden **stets** optimal eingeschlossen wird, vgl. dazu auch Seite 48. Will man jedoch, dass der linke einschließende Operand auch eine andere Präzision `prec` erhält, so kann dies mit folgender Funktion realisiert werden:

```
void set_Mpfi (MpfiClass& op, const MpfiClass& op1, PrecisionType prec);
```

Folgende Funktionsaufrufe sind möglich:

1. `set_Mpfi (op, op1, prec);`
   op erhält die Präzision prec und den i.a. gerundeten Wert von op1, wobei op1 von op stets optimal eingeschlossen wird. Setzt man prec gleich der Präzision von op1, so erhält op den **exakten** Wert von op1.

2. `set_Mpfr (op, op1);`
   op wird auf die Current-Precision gesetzt und erhält den i.a. gerundeten Wert von op1, wobei auch hier op1 von op stets optimal eingeschlossen wird.

4.9 Abfragen

Bei allen folgenden Abfragefunktionen braucht die Präzision von x nicht mit der Current-Precision übereinzustimmen.

```
bool isNan      (const MpfiClass& x);
bool isInf      (const MpfiClass& x);
bool isPoint    (const MpfiClass& x);
```

`isNan` und `isInf` überprüfen, ob ein Randpunkt von x gleich NaN bzw. ±Inf ist. `isPoint` überprüft, ob x ein Punktintervall ist.

```
bool isBounded (const MpfiClass& x);
```

`isBounded` überprüft, ob x ein normales Intervall ist, d.h. kein Randpunkt ist NaN oder ±Inf.

```
bool isZero (const MpfrClass& x);
```

Überprüft, ob x das Null-Intervall ist.

```
bool hasZero (const MpfrClass& x);
```

Überprüft, ob x die Null enthält, nicht notwendig im Innern von x.

```
bool isPos (const MpfiClass& x)
```

Überprüft, ob die Elemente von x größer oder gleich Null sind, d.h. der linke Randpunkt kann Null sein.

```
bool isStrictlyPos(const MpfiClass& x)
```

Überprüft, ob alle Elemente von x positiv sind.

```
bool isNonNeg(const MpfiClass& x);
```

Überprüft, ob alle Elemente von x nicht-negativ sind.

```
bool isNeg(const MpfiClass& x)
```

Überprüft, ob alle Elemente von x negativ sind, wobei der rechte Randpunkt Null sein kann.

```
bool isStrictlyNeg(const MpfiClass& x)
```

Überprüft, ob alle Elemente von x negativ sind.

```
bool isNonPos(const MpfiClass& x)
```

Überprüft, ob alle Elemente von x kleiner oder gleich Null sind, d.h. der rechte Randpunkt kann Null sein.

```
bool isEmpty();
```

Die obige Member-Funktion liefert True, falls die Randpunkte des aktuellen Objekts in falscher Ordnung vorliegen und False sonst.

```
bool Disjoint(const MpfiClass& x, const MpfiClass& y);
```

Die Funktion liefert True, falls `x` und `y` keine gemeinsamen Elemente besitzen und False sonst.

4.10 Vergleiche

Mit den folgenden Vergleichsfunktionen werden die üblichen Vergleichsoperatoren implementiert.

4.10.1 Vergleichsfunktionen

```
bool compare_equal     (const MpfiClass& x, const MpfiClass& y);
bool compare_less      (const MpfiClass& x, const MpfiClass& y);
bool compare_lessequal (const MpfiClass& x, const MpfiClass& y);
```

Die obigen Funktionen überprüfen, ob jeweils gilt:
$x = y$, $x \subset y$, $x \subseteq y$, Ist x oder y NaN oder Inf, so wird `False` zurückgegeben.

4.10.2 Vergleichsoperatoren $=, \neq, >, \geq, <, \leq$

```
bool operator == (const MpfiClass& op1, const MpfiClass&          op2);
bool operator == (const MpfiClass& op1, const double&             op2);
bool operator == (const MpfiClass& op1, const int                 op2);
bool operator == (const MpfiClass& op1, const cxsc::real&         op2);
bool operator == (const MpfiClass& op1, const cxsc::interval&     op2);
bool operator == (const MpfiClass& op1, const MPFR::MpfrClass&    op2);
bool operator == (const MpfiClass& op1, const mpfr_t&             op2);
bool operator == (const MpfiClass& op1, const mpfi_t&             op2);
bool operator == (const double&           op1, const MpfiClass& op2);
bool operator == (const int               op1, const MpfiClass& op2);
bool operator == (const cxsc::real&       op1, const MpfiClass& op2);
bool operator == (const mpfr_t&           op1, const MpfiClass& op2);
bool operator == (const mpfi_t&           op1, const MpfiClass& op2);
bool operator == (const MPFR::MpfrClass&  op1, const MpfiClass& op2);
bool operator == (const cxsc::interval&   op1, const MpfiClass& op2);
```

```
bool operator != (const MpfiClass& op1, const MpfiClass&          op2);
bool operator != (const MpfiClass& op1, const double&             op2);
bool operator != (const MpfiClass& op1, const int                 op2);
bool operator != (const MpfiClass& op1, const cxsc::real&         op2);
bool operator != (const MpfiClass& op1, const cxsc::interval&     op2);
bool operator != (const MpfiClass& op1, const MPFR::MpfrClass&    op2);
bool operator != (const MpfiClass& op1, const mpfr_t&             op2);
bool operator != (const MpfiClass& op1, const mpfi_t&             op2);
bool operator != (const double&           op1, const MpfiClass& op2);
bool operator != (const int               op1, const MpfiClass& op2);
bool operator != (const cxsc::real&       op1, const MpfiClass& op2);
bool operator != (const mpfr_t&           op1, const MpfiClass& op2);
bool operator != (const mpfi_t&           op1, const MpfiClass& op2);
bool operator != (const MPFR::MpfrClass&  op1, const MpfiClass& op2);
bool operator != (const cxsc::interval&   op1, const MpfiClass& op2);
```

```
bool operator < (const MpfiClass& op1, const MpfiClass&       op2);
bool operator < (const MpfiClass& op1, const mpfi_t&          op2);
bool operator < (const MpfiClass& op1, const cxsc::interval& op2);
bool operator < (const mpfi_t&          op1, const MpfiClass& op2);
bool operator < (const cxsc::interval& op1, const MpfiClass& op2);
```

'interval' < 'MpfiClass': 'interval' liegt ganz im Innern von 'MpfiClass'

```
bool operator < (const MPFR::MpfrClass& op1, const MpfiClass& op2);
bool operator < (const mpfr_t&          op1, const MpfiClass& op2);
bool operator < (const double&          op1, const MpfiClass& op2);
bool operator < (const cxsc::real&      op1, const MpfiClass& op2);
bool operator < (const int              op1, const MpfiClass& op2);
```

'real' < 'MpfiClass': 'real' liegt ganz im Innern von 'MpfiClass'

```
bool operator <= (const MpfiClass&       op1, const MpfiClass&       op2);
bool operator <= (const MpfiClass&       op1, const mpfi_t&          op2);
bool operator <= (const MpfiClass&       op1, const cxsc::interval& op2);
bool operator <= (const mpfi_t&          op1, const MpfiClass&       op2);
bool operator <= (const cxsc::interval& op1, const MpfiClass&       op2);
```

'interval' <= 'MpfiClass': 'interval' $\subseteq$ 'MpfiClass'

```
bool operator <= (const MpfiClass&       op1, const MPFR::MpfrClass& op2);
bool operator <= (const MpfiClass&       op1, const mpfr_t&          op2);
bool operator <= (const MpfiClass&       op1, const double&          op2);
bool operator <= (const MpfiClass&       op1, const cxsc::real&      op2);
bool operator <= (const MpfiClass&       op1, const int              op2);
```

'MpfiClass' <= 'real': Nur wahr, wenn Punktintervall 'MpfiClass' = 'real'

```
bool operator <= (const MPFR::MpfrClass& op1, const MpfiClass& op2);
bool operator <= (const mpfr_t&          op1, const MpfiClass& op2);
bool operator <= (const double&          op1, const MpfiClass& op2);
bool operator <= (const cxsc::real&      op1, const MpfiClass& op2);
bool operator <= (const int              op1, const MpfiClass& op2);
```

'real' <= 'MpfiClass': 'real' $\in$ 'MpfiClass'

```
bool operator > (const MpfiClass&       op1, const MpfiClass&       op2);
bool operator > (const MpfiClass&       op1, const mpfi_t&          op2);
bool operator > (const MpfiClass&       op1, const cxsc::interval& op2);
bool operator > (const mpfi_t&          op1, const MpfiClass&       op2);
bool operator > (const cxsc::interval& op1, const MpfiClass&       op2);
```

'interval' > 'MpfiClass': 'MpfiClass' liegt ganz im Innern von 'interval'

```
bool operator > (const MpfiClass&, const double&);
bool operator > (const MpfiClass&, const int);
bool operator > (const MpfiClass&, const cxsc::real&);
bool operator > (const MpfiClass&, const mpfr_t&);
bool operator > (const MpfiClass&, const MPFR::MpfrClass&);
```

'MpfiClass' > 'real': 'real' liegt ganz im Innern von 'MpfiClass'

```
bool operator >= (const MpfiClass&       op1, const MpfiClass&       op2);
bool operator >= (const MpfiClass&       op1, const mpfi_t&          op2);
bool operator >= (const MpfiClass&       op1, const cxsc::interval& op2);
bool operator >= (const mpfi_t&          op1, const MpfiClass&       op2);
bool operator >= (const cxsc::interval& op1, const MpfiClass&       op2);
```

'interval' >= 'MpfiClass': 'interval' ⊇ 'MpfiClass'

```
bool operator >= (const MpfiClass& op1, const MPFR::MpfrClass& op2);
bool operator >= (const MpfiClass& op1, const mpfr_t& op2);
bool operator >= (const MpfiClass& op1, const double& op2);
bool operator >= (const MpfiClass& op1, const cxsc::real& op2);
bool operator >= (const MpfiClass& op1, const int op2);
```

'MpfiClass' >= 'real': 'MpfiClass' ∋ 'real'

```
bool operator >= (const MPFR::MpfrClass& op1, const MpfiClass& op2);
bool operator >= (const mpfr_t&          op1, const MpfiClass& op2);
bool operator >= (const double&          op1, const MpfiClass& op2);
bool operator >= (const cxsc::real&      op1, const MpfiClass& op2);
bool operator >= (const int              op1, const MpfiClass& op2);
```

'real' >= 'MpfiClass': Nur wahr, wenn Punktintervall 'MpfiClass' = 'real'

```
int in (const MPFR::MpfrClass::MpfrClass& x, const MpfiClass& y);
int in (const mpfr_t& x,                     const MpfiClass& y);
int in (const double& x,                     const MpfiClass& y);
int in (const cxsc::real& x,                 const MpfiClass& y);
int in (const int& x,                        const MpfiClass& y);
int in (const MpfiClass& x,                  const MpfiClass& y);
int in (const mpfi_t& x,                     const MpfiClass& y);
int in (const cxsc::interval& x,             const MpfiClass& y);
```

Zurückgegeben wird die Eins, wenn `x` ganz im Innern von `y` enthalten ist, sonst wird die Null zurückgegeben. Ist einer der Operanden ein `NaN` oder sind beide Operanden unbegrenzt, so wird ebenfalls die Null zurückgegeben. Die Präzisionen beider Operanden können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen.

Hinweise:

- Alle Vergleichsoperatoren sind wie in C-XSC definiert.
- Die Präzisionen der Operanden können verschieden sein und müssen mit der Current-Präzision nicht übereinstimmen.

4.11 Durchschnitt

Berechnet wird der Durchschnitt zweier Intervalle. Ist einer der nachfolgenden Operanden eine Zahl, so ist diese als Punktintervall zu verstehen. Die Präzisionen der Intervalle oder Zahlen können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen. Die Präzision des zurückgegebenen Durchschnitt-Intervalls ist das Maximum der Präzisionen beider Operanden und muss daher mit der Current-Präzision nicht übereinstimmen. Dadurch wird der Durchschnitt **rundungsfehlerfrei** berechnet. Ist der Durchschnitt leer, so wird NaN zurückgegeben.

```
MpfiClass operator & (const MpfiClass&     op1, const MpfiClass&       op2);
MpfiClass operator & (const MpfiClass&     op1, const mpfi_t&          op2);
MpfiClass operator & (const MpfiClass&     op1, const cxsc::interval& op2);
MpfiClass operator & (const mpfi_t&        op1, const MpfiClass&       op2);
MpfiClass operator & (const cxsc::interval& op1, const MpfiClass&     op2);

MpfiClass operator & (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator & (const MpfiClass& op1, const mpfr_t&          op2);
MpfiClass operator & (const MpfiClass& op1, const double&          op2);
MpfiClass operator & (const MpfiClass& op1, const cxsc::real&      op2);
MpfiClass operator & (const MpfiClass& op1, int                    op2);

MpfiClass operator & (const MPFR::MpfrClass& op1, const MpfiClass& op2);
MpfiClass operator & (const mpfr_t&          op1, const MpfiClass& op2);
MpfiClass operator & (const double&          op1, const MpfiClass& op2);
MpfiClass operator & (const cxsc::real&      op1, const MpfiClass& op2);
MpfiClass operator & (int                    op1, const MpfiClass& op2);
```

```
MpfiClass& operator &= (MpfiClass& op1, const MpfiClass&       op2);
MpfiClass& operator &= (MpfiClass& op1, const mpfi_t&          op2);
MpfiClass& operator &= (MpfiClass& op1, const cxsc::interval& op2);
MpfiClass& operator &= (MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass& operator &= (MpfiClass& op1, const mpfr_t&          op2);
MpfiClass& operator &= (MpfiClass& op1, const double&          op2);
MpfiClass& operator &= (MpfiClass& op1, const cxsc::real&      op2);
MpfiClass& operator &= (MpfiClass& op1, int                    op2);
```

Die Anweisung `op1 &= op2;` liefert an op1 den Durchschnitt (op1 `&` op2), wobei die neue Präzision von op1 gleich dem Maximum der Präzisionen der ursprünglichen Operanden op1 und op2 ist. Auch hier wird wieder erreicht, dass der Durchschnitt **rundungsfehlerfrei** an op1 zurückgegeben wird.

Hinweis:

Wenn beide Operanden als Präzision die Current-Präzision besitzen, so wird der Durchschnitt stets **rundungsfehlerfrei** in der gleichen Präzision zurückgegeben.

4.12 Konvexe Hülle

Berechnet wird die konvexe Hülle zweier Intervalle. Ist einer der nachfolgenden Operanden eine Zahl, so ist diese als Punktintervall zu verstehen. Die Präzisionen der Intervalle oder Zahlen können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen. Die Präzision des zurückgegebenen Hüllen-Intervalls ist das Maximum der Präzisionen beider Operanden und muss mit der Current-Präzision nicht übereinstimmen. Dadurch wird die konvexe Hülle **rundungsfehlerfrei** berechnet.

```
MpfiClass operator | (const MpfiClass& op1, const MpfiClass& op2);
MpfiClass operator | (const double& op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, const double& op2);
MpfiClass operator | (const cxsc::real& op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, const cxsc::real& op2);
MpfiClass operator | (const cxsc::interval& op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, const cxsc::interval& op2);
MpfiClass operator | (int op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, int op2);
MpfiClass operator | (const MpfiClass& op1, const mpfr_t& op2);
MpfiClass operator | (const mpfr_t& op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, const mpfi_t& op2);
MpfiClass operator | (const mpfi_t& op1, const MpfiClass& op2);
MpfiClass operator | (const MpfiClass& op1, const MPFR::MpfrClass& op2);
MpfiClass operator | (const MPFR::MpfrClass& op1, const MpfiClass& op2);
```

```
MpfiClass& operator |= (MpfiClass& op1, const MpfiClass&       op2);
MpfiClass& operator |= (MpfiClass& op1, const double&          op2);
MpfiClass& operator |= (MpfiClass& op1, const cxsc::real&      op2);
MpfiClass& operator |= (MpfiClass& op1, const cxsc::interval&  op2);
MpfiClass& operator |= (MpfiClass& op1, int                    op2);
MpfiClass& operator |= (MpfiClass& op1, const mpfr_t&          op2);
MpfiClass& operator |= (MpfiClass& op1, const mpfi_t&          op2);
MpfiClass& operator |= (MpfiClass& op1, const MPFR::MpfrClass& op2);
```

> Die Anweisung `op1 |= op2;` liefert an op1 die konvexe Hülle (op1 | op2), wobei die neue Präzision von op1 gleich dem Maximum der Präzisionen der ursprünglichen Operanden op1 und op2 ist. Auch hier wird wieder erreicht, dass die konvexe Hülle **rundungsfehlerfrei** zurückgegeben wird.

Hinweis:

Wenn beide Operanden als Präzision die Current-Präzision besitzen, so wird die konvexe Hülle stets **rundungsfehlerfrei** in der gleichen Präzision zurückgegeben.

4.13 Mathematische Intervall-Funktionen

Zu einer gegebenen Funktion $f : \mathbb{R} \to \mathbb{R}$ liefert eine Intervall-Funktion $f[x]$ zu einem vorgegebenen Interval $[x] = [a, b]$ die Menge aller Funktionswerte $f(x)$, mit $x \in [x]$:

$$f([x]) = \{y \in \mathbb{R} \,|\, y = f(x) \wedge x \in [x] = [a, b]\}.$$

Gilt dann für ein Maschinenintervall $\mathtt{x} \supseteq [x]$, so liefert eine implementierte Interfallfunktion, z.B. `exp(x)`, eine garantierte Einschließung aller Funktionswerte $y = f(x) = e^x$, mit $x \in [x] = [a, b] \subseteq \mathtt{x}$.

$$\{y \in \mathbb{R} \,|\, y = e^x \wedge x \in \mathtt{x}\} \subseteq \mathtt{exp(x)}.$$

4.13.1 Standard-Implementierung

Die MPFI-Bibliothek stellt eine Vielzahl von Elementarfunktionen als Intervallfunktionen zur Verfügung, wobei zu einem Argument `x` mit beliebiger Präzision zunächst das exakte Funktionswert-Intervall `y0` berechnet wird. Danach wird dann `y0` durch ein Ergebnisintervall `y` $\supseteq$ `y0` mit einer möglichen anderen Präzision eingeschlossen. Die Deklaration z.B. der Exponentialfunktion ist mit diesen Bezeichnungen gegeben durch:

```
int mpfi_exp (mpfi_t y, mpfr_t x);
```

> Berechnet eine Funktion der MPFI-Bibliothek die Randpunkte einer Einschließung ohne Über- oder Unterlauf, so ist die berechnete Einschließung maximal-genau, d.h. die exakten Randpunkte unterscheiden sich von den berechneten auf- bzw. abgerundeten Randpunkten um weniger als ein `ulp`.

Mit obiger MPFI-Funktion `mpfi_exp(...)` wird die Exponentialfunktion für das C-XSC Interface wie folgt implementiert:

```
MpfiClass exp(const MpfiClass& x)
{
    MpfiClass y(0);
    mpfi_exp(res.mpfi_rep, v.mpfi_rep);
    return res;
}
```

Mit Hilfe des Konstruktoraufrufs wird also zunächst die Präzision der Ergebnisvariablen `y` auf die Current-Precision gesetzt, vgl. dazu auch Seite 47. Danach wird mit dem Argument `x` und seiner Präzision das exakte Intervall `y0` = $e^{\mathtt{x}}$ berechnet und dann durch `y` optimal eingeschlossen.

> Bei allen im MpfiClass-Interface implementierten mathematischen Intervallfunktionen wird das berechnete Einschließungsintervall unabhängig von der Präzision des Arguments in der Current-Präzision zurückgegeben.

Es gibt jedoch einige Intervallfunktionen, bei denen es sinnvoll ist, das Ergebnis mit der gleichen Präzision des Arguments zurückzugeben. So wird man beispielsweise verlangen, dass die Randpunkte eines Intervalls `x` mit der gleichen Präzision von `x` berechnet werden. Dies erreicht man z.B. mit den Funktionen `GetLeft(...)` bzw. `GetRight(...)`. Mit den Funktionen `Inf(...)` bzw. `Sup(...)` werden die Randpunkte jedoch in die Current-Präzision gerundet.

Im folgenden Abschnitt werden diejenigen Intervallfunktionen beschrieben, deren Rückgabewerte Präzisionen besitzen, die nicht mit der Current-Präzision übereinstimmen müssen.

4.13.2 Davon abweichende Funktionen und Konstanten

Bei den folgenden Funktionen kann die Präzision des Rückgabewertes von der Current-Präzision abweichen.

```
MPFR::MpfrClass diam(const MpfiClass& op);
```

Liefert den aufgerundeten absoluten Durchmesser von op, wobei der Rückgabewert die Präzision von op erhält.

```
MPFR::MpfrClass RelDiam(const MpfiClass& op);
```

Liefert den aufgerundeten absoluten Durchmesser von op, falls Null in op enthalten ist. Sonst wird der aufgerundete relative Durchmesser von op berechnet. In beiden Fällen erhält der Rückgabewert die Präzision von op.

```
MPFR::MpfrClass AbsMax(const MpfiClass& op);
MPFR::MpfrClass AbsMin(const MpfiClass& op);
```

Zurückgegeben wird das Maximum bzw. das Minimum aller Absolutbeträge von op, wobei der Rückgabewert die Präzision von op erhält.

```
MPFR::MpfrClass mid(const MpfiClass& op);
```

Zurückgegeben wird der Mittelpunkt von op, wobei der Rückgabewert die Präzision von op erhält.

Die folgenden sechs Funktionen haben die jeweils gleiche Bedeutung wie oben, wobei aber jetzt op2 der Rückgabewert mit der Präzision von op1 ist.

```
void diam    (const MpfiClass& op1, mpfr_t& op2);
void RelDiam (const MpfiClass& op1, mpfr_t& op2);
void AbsMax  (const MpfiClass& op1, mpfr_t& op2);
void AbsMin  (const MpfiClass& op1, mpfr_t& op2);
void mid     (const MpfiClass& op1, mpfr_t& op2);
```

```
MPFR::MpfrClass Inf(const MpfiClass& op, PrecisionType prec);
MPFR::MpfrClass Sup(const MpfiClass& op, PrecisionType prec);
```

Zurückgegeben wird der linke bzw. rechte Randpunkt von op, jeweils gerundet in ein Format mit der Präzision `prec`. One Angabe von `prec` wird dabei nach $-\infty$ bzw. nach $+\infty$ in die Current-Präzision gerundet. Mit `Inf(op,op.GetPrecision())` wird der linke Randpunkt **rundungsfehlerfrei** mit der Präzision von op zurückgegeben. Mit `Sup(op)` erhält man den rechten Randpunkt in der Current-Präzision, wobei im Bedarfsfall nach $+\infty$ gerundet wird.

```
void GetLeft  (mpfr_t& op);
void GetRight(mpfr_t& op);
```

Die obigen Member-Funktionen liefern vom aktuellen Objekt den linken bzw. rechten Randpunkt mit der gleichen Präzision des aktuellen Objekts.

```
MpfiClass Blow(const MpfiClass& op1, const MPFR::MpfrClass& op2);
```

Der Rückgabewert ist ein mit op2 aufgeblähtes Intervall op1 mit gleicher Präzision. `Blow(...)` ist genauso definiert wie die gleichnamige Funktion in C-XSC.

```
int swap_endpoints();
```

Die obigen Member-Funktion tauscht am aktuellen Objekt beide Randpunkt, falls der linke Randpunkt größer ist als der rechte. Der Rückgabewert vom Typ `int` ist positiv, wenn ein Tausch notwendig war, sonst ist der Wert gleich Null.

```
void swap(MpfiClass& x, MpfiClass& y);
```

Tauscht den Wert und die Präzision von x und y.

```
void swap(MpfiClass& x, mpfi_t& y);
```

Tauscht den Wert und die Präzision von x und y.

```
int common_decimals(const MpfiClass& op);
```

Liefert die Anzahl der übereinstimmenden Dezimalziffern, in denen die Randpunkte von op übereinstimmen.

```
MPFR::MpfrClass random(const MpfiClass& op);
```

Zurückgegeben wird eine Zufallszahl aus op, wobei der Rückgabewert die Präzision von op erhält.

```
void random(MpfiClass& x, gmp_randstate_t state);
```

Zurückgegeben wird ein Zufallsintervall $x \subseteq [0,1]$, wobei x die Präzision des vorher deklarierten Objekts x erhält. Das nachfolgende Programm zeigt eine mögliche Anwendung der Funktion `void random(MpfiClass& x, gmp_randstate_t state)`.

```
1  // MPFR-04.cpp
2  #include "mpficlass.hpp"
3
4  using namespace MPFI;
5  using namespace std;
6
7  int main(void)
8  {
9    MpfiClass::SetCurrRndMode (RoundNearest);
10   cout << "\nCurrent-RoundingMode = " << MpfiClass::GetCurrRndMode() << endl;
11   MpfiClass::SetCurrPrecision (60);
12   cout << "Current-Precision = " << MpfiClass::GetCurrPrecision() << endl;
13
14   gmp_randstate_t state;          // Declaration of state;
15   gmp_randinit_default (state); // Initialization of state;
16
17   MpfiClass x(0,50);    // Declaration of interval class object x
18                         // with a precision of 50 bits
19   cout.precision(x.GetPrecision()/3.321928095);
20   cout << "x = " << x << endl;
21   cout << "x.GetPrecision() = " << x.GetPrecision() << endl;
22   random (x, state);     // Delivers the first random interval  x
23   cout << "x = " << x << endl;
24   random (x, state);     // Delivers the second random interval x
25   cout << "x = " << x << endl;
26
27   return 0;
28 }
```

Das Programm liefert die Ausgabe:

```
Current-RoundingMode = 0
Current-Precision = 60
x = [0,-0]
x.GetPrecision() = 50
x = [3.46961336961372e-1,4.62035403472840e-1]
x = [8.09593725498825e-2,3.96496861671305e-1]
```

Weitere Informationen bez. der Initialisierungsfunktion in Zeile 15 findet man unter

`http://gmplib.org/manual/Random-State-Initialization.html`

```
void times2pown (MpfiClass& op, long int op1);
```

Obige Funktion liefert mit dem Eingabewert op eine optimale Einschließung von $op \cdot 2^{op1}$ mit gleicher Präzision zurück. Solange kein Über- oder Unterlauf entsteht, wird $op \cdot 2^{op1}$ **exakt**, d.h. rundungsfehlerfrei eingeschlossen.

```
void set_nan (MpfiClass& x);
```

Setzt x auf NaN, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_inf (MpfiClass& x);
```

Setzt x auf [-Inf, +Inf], wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_zero (MpfiClass& x);
```

Setzt x auf $[0, 0]$, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void MpfiClass::SetInterval(const MPFR::MpfrClass& a,
                            const MPFR::MpfrClass& b);
void MpfiClass::SetInterval(const mpfr_t&      a, const mpfr_t&      b);
void MpfiClass::SetInterval(const double&      a, const double&      b);
void MpfiClass::SetInterval(const cxsc::real&  a, const cxsc::real&  b);
void MpfiClass::SetInterval(const int&         a, const int&         b);
```

Die obigen Memberfunktionen liefern im Fall $a \leq b$ für das jeweils aktuelle Objekt stets eine Einschließung von [a, b], wobei die Präzision des aktuellen Objekts erhalten bleibt. Die Reihenfolge der Randpunkte a und b kann aber beliebig gewählt werden.

```
static MpfiClass Pi     (PrecisionType prec = CurrPrecision);
static MpfiClass Ln2    (PrecisionType prec = CurrPrecision);
static MpfiClass Euler  (PrecisionType prec = CurrPrecision);
static MpfiClass Catalan(PrecisionType prec = CurrPrecision);
```

`Pi(prec)` schließt π mit der Präzision prec ein. Wird prec nicht angegeben, so wird π mittels der Current-Precision eingeschlossen. Entsprechendes gilt für die drei anderen Konstanten. `Ln2()` schließt also $\ln(2) = 0.693147\ldots$ mittels der voreingestellten Current-Precision ein. Vergessen Sie nicht, beim Aufruf der Konstanten `Ln2()` die Klammern zu setzen.

4.13.3 Elementarfunktionen

Tabelle 4.1: Elementarfunktionen mit `x,y` vom Typ `MpfiClass`

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(\sin(x))$	`ln_sin(x)`
x^2	`sqr(x)`	$\ln(\cos(x))$	`ln_cos(x)`
x^2+y^2	`x2py2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
x^2-y^2	`x2my2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$1/x$	`reci(x)`	x^y	`pow(x, y)`
$\sqrt{x}$	`sqrt(x)`	$\sin(x)$	`sin(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_n(x)`	$\cos(x)$	`cos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$\tan(x)$	`tan(x)`
$\sqrt[3]{x}$	`cbrt(x)`	$\cot(x)$	`cot(x)`
$\sqrt[n]{x},\ n=2,3,\ldots$	`sqrt(x,n)`	$\arcsin(x)$	`asin(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arctan(y/x)$	`atan2(y,x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$\sinh(x)$	`sinh(x)`
e^x	`exp(x)`	$\cosh(x)$	`cosh(x)`
2^x	`exp2(x)`	$\tanh(x)$	`tanh(x)`
10^x	`exp10(x)`	$\coth(x)$	`coth(x)`
e^x-1	`expm1(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
e^{x^2}	`expx2(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
$e^{x^2}-1$	`expx2m1(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
e^{-x^2}	`expmx2(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
$e^{-x^2}-1$	`expmx2m1(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$\ln(x)$	`ln(x)`	$1/\sin(x)$	`csc(x)`
$\log_2(x)$	`log2(x)`	$1/\cos(x)$	`sec(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\log_{10}(x)$	`log10(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\cosh(x)$	`sech(x)`
AGM(x,y)	`agm(x,y)`		

Anmerkungen:

1. Mit `sqr(x);` erhält man eine Einschließung aller x^2-Werte für $x \in$ `x`, wobei in die Current-Precision gerundet wird. Weitere Informationen zur Implementierung findet man auf Seite 62.

2. Alle Funktionen der Tabelle 4.1 sind nach 1. implementiert.

3. `abs(x)` kann zusätzlich noch mit einem Präzisionsparameter `prec` aufgerufen werden. Mit `abs(x,x.GetPrecision())` erhält man dabei eine **rundungsfehlerfreie** Einschließung aller $|x|$, mit $x \in$ `x`.

4. `power(x, k)` kann bez. `k` mit folgenden Datentypen aufgerufen werden: `int, long int`.

5. `pow(x, y)` kann bez. `y` mit folgenden Datentypen aufgerufen werden:
   `MpfiClass, interval, MpfrClass, real, double`.

6. Mit dem Aufruf `agm(x,y)` erhält man in Current-Präzision eine optimale Einschließung aller AGM(x,y)-Werte für $x \in$ `x` und $y \in$ `y`. Das AGM spielt bei der Auswertung elliptischer Integrale eine entscheidende Rolle.

4.13.4 Funktionen der Mathematischen Physik

Tabelle 4.2: Funktionen der Mathematischen Physik mit `x` vom Typ `MpfiClass`

Funktionsterm	Aufruf	Anmerkung
$\mathrm{erf}(x) = \frac{2}{\pi}\int_0^x e^{-t^2}dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\pi}\int_x^\infty e^{-t^2}dt$	`erfc(x)`	monoton fallend
$\Gamma'(x)$	* `gamma_D(x)`	Pole: x=0, -1, -2, ...
$\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$	`digamma(x)`	Pole: x=0, -1, -2, ...
$k!$	`ifactorial(k)`	unsigned long int k
$\zeta(k),\ k = 0, 2, 3, 4, ...$	`izeta(k)`	unsigned long int k
$\mathrm{Ei}(x) = \gamma + \ln(x) + \sum_{k=1}^{\infty}\frac{x^k}{k\cdot k!},\ x > 0;$	`Ei(x)`	`Inf(x)=0` $\rightsquigarrow$ `-Inf;` `Inf(x)<0` $\rightsquigarrow$ `NaN;`

Anmerkungen:

1. Mit den Funktionen der Tabelle 4.2 werden zu einem Maschinenintervall `x` maximalgenaue Einschließungen aller Funktionswerte $f(x)$, mit $x \in$ `x` berechnet. Lediglich die mit * gekennzeichneten Funktionen liefern eine etwas gröbere Einschließung, da z.B. eine Einschließung der $\Gamma'(x)$-Funktionswerte mittels $\Gamma'(x) = \psi(x)\cdot\Gamma(x)$ berechnet wird, wobei zwei Funktionen und eine Multiplikation auszuwerten sind, was mit maximal drei Rundungen verbunden ist.

4.13.5 Skalarprodukt aus zwei Intervallprodukten

Um die komplexe Intervallmultiplikation realisieren zu können, benötigt man für die reellen Intervalle `a,b,c,d` vom Typ `MpfiClass` eine möglichst optimale Einschließung des Skalarproduktes

$$\mathtt{r} := \mathtt{a}\cdot\mathtt{b} + \mathtt{c}\cdot\mathtt{d}, \tag{4.1}$$

wobei die reellen Variablen aus `a,b,c,d` als voneinander **unabhängig** anzusehen sind, so dass eine optimale Einschließung von `r` schon durch eine einfache Intervallauswertung der rechten Seite von (4.1) erreicht werden kann. Dabei sind aber noch zwei Probleme zu beachten:

1. Insbesondere bei schmalen Intervallen kann es bei der Intervallauswertung der rechten Seite von (4.1) wegen Auslöschungseffekten zu starken Überschätzungen kommen, die oft vermieden werden können, wenn man die Intervallauswertung z.B. mit doppelter Präzision durchführt.

2. Wenn die Randpunkte von z.B. `a,b` zu groß werden, kann es bei der Auswertung von z.B. $\mathtt{a}\cdot\mathtt{b}$ zu einem vorzeitigen Überlauf kommen, obwohl `r` im Zahlenraster noch darstellbar ist. Das Problem kann wie folgt gelöst werden:

   - Jedes Intervall, also auch `a`, wird so skaliert, dass gilt $\mathtt{a} \subseteq \mathtt{A}\cdot 2^{k_A}$. Dabei wird $k_A \in \mathbb{Z}$ so gewählt, dass wenigstens einer der Randpunkte von $2\cdot(\mathtt{A}\cdot\mathtt{A})$ möglichst groß ist aber gerade noch keinen Überlauf erzeugt. Im Fall $\mathtt{a} \neq [0,0]$ wird k_A wie folgt berechnet: $k_A = 536870910 - \mathtt{max\{expo(Inf(a)),expo(Sup(a))\}}$, und im Fall $\mathtt{a} = [0,0]$ wird k_A auf Null gesetzt. Mit diesen Überlegungen kann das Skalarprodukt rechts in (4.1) wie folgt berechnet werden: $\mathtt{a}\cdot\mathtt{b} + \mathtt{c}\cdot\mathtt{d} \subseteq (\mathtt{A}\cdot\mathtt{B})\cdot 2^{k_A+k_B} + (\mathtt{C}\cdot\mathtt{D})\cdot 2^{k_C+k_D} = \mathtt{P}_1\cdot 2^{k_1} + \mathtt{P}_2\cdot 2^{k_2}$, wobei die Teilprodukte $\mathtt{P}_1 := (\mathtt{A}\cdot\mathtt{B})$ und $\mathtt{P}_2 := (\mathtt{C}\cdot\mathtt{D})$ beide ohne Überlauf berechnet werden.
   - Wir setzen jetzt voraus, dass k_1 das Maximum der beiden Zweier-Exponenten ist. Dann kann die Summe $\mathtt{S} := \mathtt{P}_1\cdot 2^{k_1} + \mathtt{P}_2\cdot 2^{k_2}$ erst nach der folgender Skalierung $\mathtt{S} := \mathtt{P}_1\cdot 2^{k_1} + \mathtt{P}_2\cdot 2^{k_2} = \mathtt{P}_1\cdot 2^{k_1} + (\mathtt{P}_1\cdot 2^{k_2-k_1})\cdot 2^{k_1} = (\mathtt{P}_1+\mathtt{P}_3)\cdot 2^{k_1}$ ausgewertet werden, wobei die Intervallsumme $\mathtt{P}_4 := (\mathtt{P}_1+\mathtt{P}_3)$ **ohne** Überlauf berechnet werden kann. Ein Überlauf kann somit erst bei der Multiplikation $\mathtt{P}_4\cdot 2^{k_1}$ eintreten, der dann aber unvermeidbar ist, sonst gilt: $\mathtt{a}\cdot\mathtt{b} + \mathtt{c}\cdot\mathtt{d} \subseteq \mathtt{P}_4\cdot 2^{k_1}$.

   Abschließend noch ein Hinweis auf einen möglichen Nachteil dieses Verfahrens. Wenn einer der Randpunkte von $\mathtt{a} \subseteq (\mathtt{a}\cdot 2^{-k_A})\cdot 2^{k_A} = \mathtt{A}\cdot 2^{k_A}$ schon selbst dicht vor dem Überlauf liegt, so wird k_A positiv sein, so dass es bei der Skalierung $\mathtt{a}\cdot 2^{-k_A}$ zu einer Überschätzung kommen kann, wenn der andere Randpunkt von `a` zu dicht am Unterlaufbereich liegt. Da aber diese Überschätzung nur eintreten wird, wenn das Intervall `a` praktisch ganz in einer der beiden komplexen Halbebenen liegt, kann aus praktischen Gründen eine solche, vergleichsweise geringe Überschätzung vermutlich immer in Kauf genommen werden.

Die oben beschriebene Skalierung $\mathtt{a} \subseteq \mathtt{A}\cdot 2^{k_A}$ wird mit der folgenden Funktion realisiert.

```
void Interval_Scaling(MpfiClass& z, long int& k)
```

Obige Funktion liefert mit dem Eingabewert `z` = `a` den Rückgabewert `z` = `A` vom Typ `MpfiClass` in der Präzision von `a`, zusätzlich wird k_A zurückgegeben. Ist einer der beiden Randpunkte von `a` unendlich oder ein NaN, so wird $[NaN, NaN]$ und $k_A = 0$ zurückgegeben, sonst gilt $\mathtt{a} \subseteq \mathtt{A}\cdot 2^{k_A}$, wobei $2\cdot(\mathtt{A}\cdot\mathtt{A})$ garantiert ohne Überlauf berechnet werden kann. $\mathtt{a} \subset \mathtt{A}\cdot 2^{k_A}$ gilt nur in den ganz seltenen Fällen, wenn bei der Berechnung von $\mathtt{a}\cdot 2^{-k_A}$ ein Unterlauf eintritt, d.h. wenn gilt $\mathtt{a}\cdot 2^{-k_A} \subset \mathtt{A}$. Weitere Einzelheite siehe oben unter 2.

```
MpfiClass scal_prod(const MpfiClass& a, const MpfiClass& b,
                    const MpfiClass& c, const MpfiClass& d);
```

Der Rückgabewert vom Typ `MpfiClass` ist eine meist optimale Einschließung des Skalarproduktes $\mathtt{a}\cdot\mathtt{b}+\mathtt{c}\cdot\mathtt{d}$ in der Current-Präzision, wobei ein vorzeitiger Über- oder Unterlauf vermieden wird. Die Präzisionen der 4 Eingabeintervalle können verschieden sein. Weitere Einzelheite siehe Seite 69 unter Punkt 2. Zur Anwendung kommt diese Funktion bei der Multiplikation komplexer Intervalle.

Das folgende Programm zeigt die Einschließung spezieller Skalarprodukte mithilfe der Funktion `scal_prod()`.

```
1  // MPFR-06.cpp
2  #include "mpficlass.hpp"
3
4  using namespace MPFR;
5  using namespace MPFI;
6  using namespace std;
7
8  int main(void)
9  {
10   MpfiClass::SetCurrPrecision (300);
11   cout << "Current-Precision = " << MPFR::MpfrClass::GetCurrPrecision() << endl;
12
13   MpfrClass x(372130600.0);
14   x = exp(x);
15   MpfiClass a(x,x), b(succ(x),succ(x)), c(-x,-x), d(x,x), r;
16   r = scal_prod(a,b,c,d);
17   cout.precision( r.GetPrecision()/3.321928095);
18   cout << "r = " << r << endl;
19   cout << "r = " << a*b + c*d << endl;
20
21   return 0;
22 }
```

In Zeile 14 wird mit `x` = 372130600.0 und `exp(x)` eine hinreichend große Maschinenzahl erzeugt. In Zeile 15 werden $\mathtt{a}\approx\mathtt{b}, \mathtt{c}=-\mathtt{a}, \mathtt{d}=\mathtt{a}$ als Punktintervalle definiert, so dass bei der Auswertung ihres Skalarprodukts $r := \mathtt{a}\cdot\mathtt{b}+\mathtt{c}\cdot\mathtt{d}$ mit starker Auslöschung und mit einem vorzeitigen Überlauf zu rechnen ist. Dies wird durch die folgende Programmausgabe bestätigt

```
r = [1.554622378...836989271e323228442,1.554622378...836989271e323228442]
r = [-@Inf@,@Inf@]
```

wobei die Einschließung $r \subseteq \mathtt{r}$ realisiert wird.

Mit dem kleineren Argument `x` = 372130500.0 erhält man die Ausgabe

```
r = [1.296663701...349242307e323228355,1.296663701...349242308e323228355]
r = [0,2.753514027...992982088e323228355]
```

Die erste Ausgabe liefert eine optimale Einschließung von r, die intern mit doppelter Präzision berechnet wird. Die zweite Ausgabe wird mit einfacher Präzision `prec` = 300 berechnet und zeigt bez. des linken Randpunktes Null eine erhebliche Überschätzung der exakten Einschließung.

5 MpfcClass-Interface für komplexe Langzahlrechnungen in C-XSC

5.1 Grundlegendes

Das MpfcClass-Interface ist eine in `mpfcclass.hpp` und `mpfcclass.cpp` implementierte **C++**-Wrapper-Klasse MpfcClass für die C-Bibliothek MPFR, deren C-Funktionen über die implementierten Operatoren und Funktionen aufgerufen werden. Alle Funktionen mit der möglichen Übergabe eines RoundingModes oder eines PrecisionTypes besitzen als Standard die Werte von CurrRndMode bzw. CurrPrecision, die beide beliebig gesetzt werden können. Dies gilt auch für alle Konstruktoren.

5.1.1 Allgemein

Um in C-XSC das Interface verwenden zu können, muss der Header mpfcclass.hpp eingebunden werden. In ihm sind die benötigten Header-Dateien der MPFR-Bibliothek enthalten. Die MpfcClass-Klasse liegt im Namensraum "MPFR".

5.1.2 Aufbau

Die Klasse besteht intern aus zwei "mpfr_t"-Variablen. Diese dienen zum Speichern von Real- und Imaginärteil. Zusätzlich gibt es static Elemente, um den Standard-Rundungsmodus, die Standard-Precision und die aktuelle Basis zu speichern.

5.1.3 Präzision

Die Präzision gibt die Anzahl der binären Mantissenstellen einer MpfcClass-Variablen an, Real- und Imaginärteil erhalten stets die gleiche Präzision. Der Wert `prec` muss mindestens 2 betragen. Die Current-Precision kann global gesetzt werden; wenn dies nicht geschieht, so wird mit der Default-Precision von 53 Bits gerechnet. Unabhängig davon kann die Präzision für jede MpfcClass-Variable auch einzeln festgelegt werden.

5.1.4 Variablentyp PrecisionType

Mithilfe des Variablentyps `PrecisionType` (Name der Variablen meist prec) kann der Präzisionswert einer MpfcClass-Variablen eingestellt werden. Der Variablentyp ist ein typedef für eine "mp_prec_t"-Variable.

5.1.5 Variablentyp RoundingMode

Mit dem Variablentyp "RoundingMode" (Name der Variablen meist rnd) wird der gewünschte Rundungsmodus eingestellt. Der Variablentyp ist ein typdef für eine `mpfr_rnd_t`-Variable.

- RoundNearest – Der Wert wird zur nächsten Rasterzahl gerundet (0)
- RoundUp – Der Wert wird aufgerundet (2)
- RoundDown – Der Wert wird abgerundet (3)

- RoundToZero – Der Wert wird in Richtung Null gerundet (1)
- RoundFromZero – Rundung weg von der Null (4)

5.2 Rundungsmodi und Precision Handling

```
PrecisionType GetPrecision () const;
```

Diese Memberfunktion gibt die maximale Präzision von Real- und Imaginärteil des aktuellen Objekts in Bits zurück. Dabei entsprechen z.B. 302 Bits $302/\log_2(10) \approx 91$ Dezimalstellen.

```
void SetPrecision (PrecisionType prec);
```

Diese Memberfunktion setzt die Präzision des aktuellen Objektes auf prec. Sein Wert bleibt nicht erhalten.

```
void RoundPrecision (PrecisionType prec, RoundingMode rnd);
```

Diese Memberfunktion rundet das aktuelle Objekt auf die neue Precision prec, sein Wert bleibt dabei i.a. nicht erhalten. Sollte die Präzision des Objektes größer sein als prec, wird das Objekt mit Hilfe des eingestellten Rundungsmodus so gerundet, dass es in das Format der Präzision prec passt. Ist die Präzision kleiner als prec, werden die restlichen binären Stellen mit Nullen aufgefüllt.

```
static const PrecisionType GetCurrPrecision ();
```

Gibt die aktuelle Current-Precision in Bits zurück.

```
static void SetCurrPrecision (PrecisionType prec);
```

Setzt die Current-Precision in `MpfrClass`, `MpfiClass` und `MpfcClass` auf prec. Wird die Current-Precision nicht gesetzt, so wird mit der Default-Precision von 53 Bits gerechnet. Durch das Setzen der Current-Precision werden die Präzisionen der bis dahin benutzten Variablen **nicht** geändert.

```
static const int GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen.

```
static void SetBase (int b);
```

Setzt die aktuelle Basis auf b. Dies muss ein Wert zwischen 2 und 36 sein. Die Basis hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die internen Rechnungen erfolgen stets im Binärsystem!

```
static const RoundingMode GetCurrRndMode ();
```

Gibt den aktuellen Rundungsmodus zurück mit den Werten: (0, 1, 2, 3, 4).

```
static void SetCurrRndMode (RoundingMode rnd);
```

Setzt den Current-Rundungsmodus in der Klasse `MpfrClass` und damit automatisch auch in der Klasse `MpfcClass` auf rnd. Für rnd sind fünf verschiedene Modi möglich:

- RoundNearest: Rundung zur nächsten Rasterzahl (0)
- RoundToZero: Rundung in Richtung Null (1)
- RoundFromZero: Rundung weg von der Null (4)
- RoundUp: Aufrunden (2)
- RoundDown: Abrunden (3)

Wird der Rundungsmodus mit `SetCurrRndMode` nicht gesetzt, so wird als Default-Rundungsmodus `RoundNearest` benutzt.

5.3 Konstruktoren / Destruktor

5.3.1 Konstruktoren

```
MpfcClass ();
```

Der Default-Konstruktor legt ein neues Element mit der Current-Precision an.
Der Aufruf `MpfcClass y;` initialisiert den Wert: `y` = (NaN, NaN);

```
MpfcClass::MpfcClass(const MpfcClass&      z,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const cxsc::complex& z,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const MpfrClass& x, const MpfrClass& y,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const mpfr_t& x, const mpfr_t& y,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const cxsc::real& x, const cxsc::real& y,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const double& x, const double& y,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(int x, int y,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const MpfrClass& x,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const mpfr_t& x,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const cxsc::real& x,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(const double& x,
                     RoundingMode  rnd, PrecisionType prec);
MpfcClass::MpfcClass(int x,
                     RoundingMode  rnd, PrecisionType prec);
```

Mit den Konstruktoraufrufen `MpfcClass w(z);` oder `MpfcClass w(x, y);` werden die Objekte `z` bzw. `x,y` nach `w` in die Current-Präzision gerundet. Mit `rnd` und `prec` sind Rundungen auch in eine andere Präzision möglich. Real- und Imaginärteil von `w` erhalten stets die **gleiche** Präzision.

Mögliche Konstruktor-Aufrufe sind:

1. `MpfcClass w(z);` `MpfcClass w(x, y);`
2. `MpfcClass w(z, RoundNearest);` `MpfcClass w(x, y, RoundNearest);`
3. `MpfcClass w(z, RoundDown, 3);` `MpfcClass w(x, y, RoundDown, 3);`

Zu 1. Die Objekte `z,x,y` werden mit dem Current-Rundungsmodus in das Objekt `w` mit der Current-Precision gerundet.

Zu 2. Die Objekte `z,x,y` werden mit `RoundNearest` in das Objekt `w` mit der Current-Precision gerundet.

Zu 3. Die Objekte `z,x,y` werden in das Objekt `w` auf die (sehr kleine) Präzision *prec* = 3 abgerundet.

Der Aufruf `MpfcClass w(x, y, 3);` führt zu einer Fehlermeldung, da in der Parameterliste der Rundungsmodus fehlt. Die oberen 18 Konstruktoren erlauben also eine sehr flexible Initialisierung von `MpfcClass`-Objekten.

```
MpfcClass::MpfcClass (const std::string& s, RoundingMode rnd,
                      PrecisionType prec);
```

Der Aufruf `MpfcClass z(s);` rundet den String s mittels der voreingestellten Current-Precision und mit dem Current-Rundungsmodus in ein Klassenobjekt `z` vom Typ `MpfcClass`. Mit dem Aufruf `MpfcClass z(s, RoundUp);` wird der String s mit der Current-Precision in ein Klassenobjekt `z` vom Typ `MpfcClass` aufgerundet. Der Aufruf `MpfcClass z(s, RoundNearest, 140);` rundet s in ein Klassenobjekt `z` vom Typ `MpfcClass` mit einer Präzision von 140 Bits. Gerundet wird dabei zur nächsten Rasterzahl dieses Formats. Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in eine binäre Zahl konvertiert werden kann, so dass daher Rundungen i.a. nicht zu vermeiden sind.
Der String s muss das Format `(Number,Number)` ohne Leerzeichen besitzen.

5.3.2 Destruktor

```
~MpfcClass ();
```

Der Speicher für das Objekt wird freigegeben. Dieser Destruktor muss nicht explizit aufgerufen werden!

5.4 Zuweisungs-Operatoren

Bei den folgenden sechs Zuweisungs-Operationen erhält der linke Operand als Präzision stets die aktuelle Current-Precision, und der rechte Operand op wird nach dem aktuellen Current-Rundungsmodus in den linken Operanden gerundet.

```
MpfcClass& operator = (const MpfcClass& op);
MpfcClass& operator = (const MpfrClass& op);
MpfcClass& operator = (const double& op);
MpfcClass& operator = (const cxsc::real& op);
MpfcClass& operator = (const cxsc::complex& op);
MpfcClass& operator = (const int opt);
```

Ist z.B. op vom Typ `real`, `double` oder `complex` und ist die Current-Precision kleiner als 53, so wird op i.a. in den linken Operanden gerundet. Nur wenn die Current-Precision größer oder gleich 53 ist, erhält der linke Operand genau den Wert des rechten Operanden. Für andere Typen des rechten Operanden gelten ganz entsprechende Aussagen.

Ist z.B. op vom Typ `MpfrClass` und ist seine Präzision größer als die Current-Precision, so wird op in den Realteil des linken Operanden bez. des Current-Rundungsmodus gerundet, d.h. der Realteil des linken Operanden wird dann i.a. vom Wert des rechten Operanden **verschieden** sein!

Zusammenfassung:

> Bei einer Wertzuweisung erhält der linke Operand stets die Current-Präzision. Ist diese kleiner als die Präzision des rechten Operanden, so wird dieser mittels des Current-Rundungsmodus in den linken Operanden gerundet. Ist die Current-Präzision größer oder **gleich** der Präzision des rechten Operanden, so wird dieser **rundungsfehlerfrei** nach links übertragen. Real- und Imaginärteil erhalten stets die gleiche Präzision.

```
MpfcClass& operator = (const std::string& s);
```

Der String s muss die Form `(Number,Number)` haben und darf keine Leerzeichen enthalten. Real- und Imaginärteil von s werden mit dem Current-Rundungsmodus in die Current-Präzision gerundet. Beachten Sie, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in das interne Binärformat gerundet wird, so dass dann Rundungen unvermeidbar sind.

5.5 Eingabe / Ausgabe

```
std::ostream& operator << (std::ostream& os, const MpfcClass& z);
```

Ermöglicht die Ausgabe einer MpfcClass-Variablen z über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(z.GetPrecision()/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Real- und Imaginärteil von z werden in der mit `SetBase(k)` voreingestellten Basis im Format

```
(Realteil, Imaginärteil)
```

ausgegeben, wobei Real- und Imaginärteil beide mit dem voreingestellten Current-Rundungsmodus gerundet werden. Für die meist dezimale Ausgabe ist natürlich k = 10 zu wählen.

```
std::istream& operator >> (std::istream& is, MpfiClass& z);
```

Ermöglicht das Einlesen einer MpfcClass-Variablen über den Standard-Eingabestrom "cin". Die eingegebene komplexe Zahl ist auf keine Stellenanzahl begrenzt. Nur das folgende Eingabeformat ist zulässig, wobei mit dem Current-Rundungsmodus in das interne Binärsystem gerundet wird:

- `(-1.23e-4401, +2.3E+2000)`

Vorsicht: Leerzeichen sind nur nach dem Komma erlaubt, und die runden Klammern müssen beide gesetzt werden, sonst erfolgt eine Fehlermeldung.

5.6 Typ-Umwandlungen

Um ein flexibles Arbeiten zwischen C-XSC und der Klasse `MpfcClass` zu ermöglichen, wurden möglichst viele Typ-Umwandlungs-Funktionen bereitgestellt:

5.6.1 `real, double, complex, ...` → MPFC

```
MpfcClass MpfrClass2Mpfc (const MpfrClass&      x);
MpfcClass mpfr_t2Mpfc    (const mpfr_t&         x);
MpfcClass real2Mpfc      (const cxsc::real&     x);
MpfcClass double2Mpfc    (const double&         x);
MpfcClass complex2Mpfc   (const cxsc::complex&  x);
MpfcClass int2Mpfc       (const int&            x);
```

Obige Funktionen liefern mit dem jeweiligen Eingabewert `x` einen Rückgabewert vom Typ `MpfcClass` in einer Präzision, die gewährleistet, dass der Rückgabewert genau dem Wert von `x` entspricht. Die Präzision des Rückgabewertes wird also i.a. nicht mit der Current-Precision übereinstimmen!

5.6.2 MPFC → `complex`

```
cxsc::complex to_complex (const MpfcClass& z , RoundingMode rnd);
```

Obige Funktion rundet mit `rnd` den Eingabewert `z` in einen C-XSC Rückgabewert vom Typ `complex`.

5.6.3 MPFC → `mpfr_t`

```
mpfr_t& MpfcClass::GetValueRe();
```

Die Memberfunktion `GetValueRe()` liefert für das jeweils aktuelle Objekt vom Typ `MpfcClass` eine Referenz auf seinen Realteilwert `mpfr_re` vom Typ `mpfr_t`. Damit kann dieser Realteil z.B. an einen nicht-konstanten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
mpfr_t& MpfcClass::GetValueIm();
```

Die Memberfunktion `GetValueIm()` liefert für das jeweils aktuelle Objekt vom Typ `MpfcClass` eine Referenz auf seinen Imaginärteil `mpfr_im` vom Typ `mpfr_t`. Damit kann dieser Imaginärteil z.B. an einen nicht-konstanten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
const mpfr_t& getvalueRe(const MpfcClass& z)
```

Die Funktion `getvalueRe(...)` liefert für das als `const` definierte Objekt `z` vom Typ `MpfcClass` eine Referenz auf seinen Realteil `mpfr_re` vom Typ `mpfr_t`. Damit kann dieser Realteil z.B. an einen als `const` deklarierten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
const mpfr_t& getvalueIm(const MpfcClass& z)
```

`getvalueIm(...)` liefert für das als `const` definierte Objekt `z` vom Typ `MpfcClass` eine Referenz auf seinen Imaginärteil `mpfr_im` vom Typ `mpfr_t`. Damit kann dieser Imaginärteil z.B. an einen als `const` deklarierten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

5.6.4 `mpfr_t` → MPFC

```
void MpfcClass::SetValueRe(const mpfr_t& v)
```

Mit dieser Memberfunktion wird der Realteil des aktuellen Objekts exakt auf v gesetzt. Der Imaginärteil des aktuellen Objekts bleibt dabei erhalten. Die Praezision des aktuellen Objekts wird gesetzt auf das Maximum der Praezisionen von `v` und `mpfr_im`, d.h. die Präzisionen von Real- und Imaginärteil sind nach dem Funktionsaufruf wieder gleich.

```
void MpfcClass::SetValueIm(const mpfr_t& v)
```

Mit dieser Memberfunktion wird der Imaginärteil des aktuellen Objekts exakt auf v gesetzt. Der Realteil des aktuellen Objekts bleibt dabei erhalten. Die Praezision des aktuellen Objekts wird gesetzt auf das Maximum der Praezisionen von `v` und `mpfr_re`, d.h. die Präzisionen von Real- und Imaginärteil sind nach dem Funktionsaufruf wieder gleich.

```
void MpfcClass::SetValue(const mpfr_t& re, const mpfr_t& im)
```

Mit dieser Memberfunktion werden Real- und Imaginärteil des aktuellen Objekts exakt auf re bzw. im gesetzt. Das aktuelle Objekt erhält als Präzision das Maximum der Präzisionen von re und im, d.h. die Präzisionen von Real- und Imaginärteil sind nach dem Funktionsaufruf wieder gleich.

5.6.5 MPFC → `string`

```
std::string to_string(const MpfcClass& z, RoundingMode rnd,
                                          PrecisionType prec);
```

$z = x + i \cdot y$ wird mittels rnd in einen String `s` mit prec Dezimalstellen gerundet, wenn Base gleich 10 ist. Der String besitzt das Format

```
(Number,Number)
```

wobei intern keine Leerzeichen auftreten. Wählt man prec hinreichend groß, so stellt der String den Wert von z **exakt** dar, weil eine Binärzahl stets exakt in eine Dezimalzahl umgewandelt werden kann.

Wird prec nicht angegeben, so werden x, y mittels rnd in einen String gerundet, der bei Base=10 soviele Dezimalstellen besitzt, wie es der Präzision von `z` entspricht. Ist die Präzision von `z` z.B. 302, so werden im String Real- und Imaginärteil mit $302/\log_2(10) \approx 91$ Dezimalstellen generiert. Wird neben prec auch rnd nicht angegeben, so wird mit dem Current-Rundungsmodus in den String mit gleicher Dezimalstellenzahl gerundet.

5.6.6 `string` → MPFC

```
MpfcClass string2Mpfc(const std::string& s, Roundingmode rnd,
                                            PrecisionType prec);
```

Der Aufruf `string2Mpfc(s);` rundet den String s mittels der voreingestellten Current-Precision und mit dem Current-Rundungsmodus in ein Klassenobjekt vom Typ `MpfcClass`. Mit dem Aufruf `string2Mpfc(s, RoundUp);` wird der String s mit der Current-Precision in ein Klassenobjekt vom Typ `MpfcClass` aufgerundet. Der Aufruf `string2Mpfc(s, RoundNearest, 140);` rundet s in ein Klassenobjekt vom Typ `MpfcClass` mit der Präzision 140 Bits. Gerundet wird dabei zur nächsten Rasterzahl dieses Formats. Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in eine binäre Zahl konvertiert werden kann, so dass daher Rundungen i.a. nicht zu vermeiden sind. Eine weitere Möglichkeit, einen String in ein `MpfcClass`-Objekt zu verwandeln, besteht in einem entsprechenden Konstruktor-Aufruf, vgl. Seite 75.
Der String s muss das Format `(Number,Number)` ohne Leerzeichen besitzen.

5.6.7 MPFC → MPFC

Beachten Sie bitte, dass bei einer Wertzuweisung an eine `MpfcClass`-Variable mit Hilfe des Operators `=` der linke Operand mit Real- und Imaginärteil stets auf die Current-Precision gesetzt wird und dass der rechte `MpfrClass`-Operand dabei **stets** bez. des Current-Rundungsmodus in den linken Operanden gerundet wird, vgl. dazu auch Seite 76. Will man jedoch abweichend von dieser Rundung einen anderen Rundungsmodus benutzen, so kann dies mit folgender Funktion ohne Rückgabewert realisiert werden:

```
void set_Mpfc (MpfcClass& op, const MpfcClass& op1, RoundingMode rnd,
                                                    PrecisionType prec);
```

Folgende Funktionsaufrufe sind möglich:

1. `set_Mpfc (op, op1, RoundUp, prec);`
   op erhält die Präzision prec und den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls aufgerundet wird. Setzt man prec gleich der Präzision von op1, so erhält op den **exakten** Wert von op1, und zwar unabhängig vom gewählten Rundungsmodus rnd.

2. `set_Mpfc (op, op1, RoundDown);`
   op wird auf die Current-Precision gesetzt und erhält den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls abgerundet wird.

3. `set_Mpfc (op, op1);`
   op wird auf die Current-Precision gesetzt und erhält den i.a. gerundeten Wert von op1, wobei hier gegebenenfalls bez. des Current-Rundungsmodus gerundete wird.

5.7 Abfragen

Bei allen folgenden Abfragefunktionen braucht die Präzision von x nicht mit der Current-Precision übereinzustimmen.

```
bool isNan    (const MpfcClass& x);
bool isInf    (const MpfcClass& x);
bool isNumber(const MpfcClass& x);
bool isZero   (const MpfcClass& x);
```

`isNan` und `isInf` prüfen, ob Real- oder Imaginärteil von x gleich NaN bzw. ±Inf sind. `isNumber` überprüft, ob Real- und Imaginärteil von x normale `MpfrClass` Zahlen ungleich NaN und ungleich ±Inf sind, und `isZero` überprüft, ob Real- und Imaginärteil von x gleich Null sind.

5.8 Vergleiche

Es werden die üblichen Vergleichsoperatoren implementiert, wobei wenigstens ein Operand vom Typ `MpfcClass` sein muss. Die Operanden können unterschiedliche Präzisionen besitzen.

5.8.1 Vergleichsoperatoren =, !=

```
bool operator == (const MpfcClass& y, const MpfcClass&          x);
bool operator == (const MpfcClass& y, const cxsc::complex&      x);
bool operator == (const MpfcClass& y, const MpfrClass&          x);
bool operator == (const MpfcClass& y, const mpfr_t&             x);
bool operator == (const MpfcClass& y, const double&             x);
bool operator == (const MpfcClass& y, const cxsc::real&         x);
bool operator == (const MpfcClass& y, const int                 x);
bool operator == (const MpfrClass&          y, const MpfcClass& x);
bool operator == (const cxsc::complex&      y, const MpfcClass& x);
bool operator == (const mpfr_t&             y, const MpfcClass& x);
bool operator == (const double&             y, const MpfcClass& x);
bool operator == (const cxsc::real&         y, const MpfcClass& x);
bool operator == (const int                 y, const MpfcClass& x);
```

```
bool operator != (const MpfcClass& y, const MpfcClass&          x);
bool operator != (const MpfcClass& y, const cxsc::complex&      x);
bool operator != (const MpfcClass& y, const MpfrClass&          x);
bool operator != (const MpfcClass& y, const mpfr_t&             x);
bool operator != (const MpfcClass& y, const double&             x);
bool operator != (const MpfcClass& y, const cxsc::real&         x);
bool operator != (const MpfcClass& y, const int                 x);
bool operator != (const MpfrClass&          y, const MpfcClass& x);
bool operator != (const cxsc::complex&      y, const MpfcClass& x);
bool operator != (const mpfr_t&             y, const MpfcClass& x);
bool operator != (const double&             y, const MpfcClass& x);
bool operator != (const cxsc::real&         y, const MpfcClass& x);
bool operator != (const int                 y, const MpfcClass& x);
```

5.9 Arithmetische Operatoren

Für alle arithmetischen Operatoren gilt:

> Das exakte Ergebnis einer arithmetischen Operation wird unabhängig von der Präzision der Operanden mit dem voreingestellten Current-Rundungsmodus optimal gerundet. Die Ergebnis-Präzision ist dabei stets gleich der voreingestellten Current-Precision.
> Die Operatoren $\circledcirc=$, mit $\circledcirc \in \{+, -, \cdot, /\}$, bedeuten $u \circledcirc = v \iff u = u \circledcirc v$. Dabei wird $u \circledcirc v$ mit dem Current-Rundungsmodus in die Current-Precision gerundet und in u gespeichert, wobei u als Präzision die Current-Precision erhält.

5.9.1 Addition

```
MpfcClass operator + (const MpfcClass& x, const MpfcClass&       y);
MpfcClass operator + (const MpfcClass& x, const cxsc::complex&   y);
MpfcClass operator + (const MpfcClass& x, const MpfrClass&       y);
MpfcClass operator + (const MpfcClass& x, const mpfr_t&          y);
MpfcClass operator + (const MpfcClass& x, const double&          y);
MpfcClass operator + (const MpfcClass& x, const cxsc::real&      y);
MpfcClass operator + (const MpfcClass& x, const int              y);
MpfcClass operator + (const cxsc::complex&   x, const MpfcClass& y);
MpfcClass operator + (const MpfrClass&       x, const MpfcClass& y);
MpfcClass operator + (const mpfr_t&          x, const MpfcClass& y);
MpfcClass operator + (const double&          x, const MpfcClass& y);
MpfcClass operator + (const cxsc::real&      x, const MpfcClass& y);
MpfcClass operator + (const int              x, const MpfcClass& y);
```

```
MpfcClass& operator += (MpfcClass&, const MpfcClass&);
MpfcClass& operator += (MpfcClass&, const cxsc::complex&);
MpfcClass& operator += (MpfcClass&, const MpfrClass&);
MpfcClass& operator += (MpfcClass&, const mpfr_t&);
MpfcClass& operator += (MpfcClass&, const double&);
MpfcClass& operator += (MpfcClass&, const cxsc::real&);
MpfcClass& operator += (MpfcClass&, const int);
```

5.9.2 Subtraktion

Beachten Sie den Hinweis auf Seite 83.

```
MpfcClass operator - (const MpfcClass& x, const MpfcClass&          y);
MpfcClass operator - (const MpfcClass& x, const cxsc::complex&      y);
MpfcClass operator - (const MpfcClass& x, const MpfrClass&          y);
MpfcClass operator - (const MpfcClass& x, const mpfr_t&             y);
MpfcClass operator - (const MpfcClass& x, const double&             y);
MpfcClass operator - (const MpfcClass& x, const cxsc::real&         y);
MpfcClass operator - (const MpfcClass& x, const int                 y);
MpfcClass operator - (const cxsc::complex&     x, const MpfcClass& y);
MpfcClass operator - (const MpfcClass&         x, const MpfcClass& y);
MpfcClass operator - (const mpfr_t&            x, const MpfcClass& y);
MpfcClass operator - (const double&            x, const MpfcClass& y);
MpfcClass operator - (const cxsc::real&        x, const MpfcClass& y);
MpfcClass operator - (const int                x, const MpfcClass& y);
```

```
MpfcClass& operator -= (MpfcClass&, const MpfcClass&);
MpfcClass& operator -= (MpfcClass&, const cxsc::complex&);
MpfcClass& operator -= (MpfcClass&, const MpfrClass&);
MpfcClass& operator -= (MpfcClass&, const mpfr_t&);
MpfcClass& operator -= (MpfcClass&, const double&);
MpfcClass& operator -= (MpfcClass&, const cxsc::real&);
MpfcClass& operator -= (MpfcClass&, const int);
```

5.9.3 Multiplikation

Beachten Sie den Hinweis auf Seite 83.

```
MpfcClass operator * (const MpfcClass& x, const MpfcClass&       y);
MpfcClass operator * (const MpfcClass& x, const cxsc::complex&   y);
MpfcClass operator * (const MpfcClass& x, const MpfrClass&       y);
MpfcClass operator * (const MpfcClass& x, const mpfr_t&          y);
MpfcClass operator * (const MpfcClass& x, const double&          y);
MpfcClass operator * (const MpfcClass& x, const cxsc::real&      y);
MpfcClass operator * (const MpfcClass& x, const int              y);
MpfcClass operator * (const cxsc::complex&   x, const MpfcClass& y);
MpfcClass operator * (const MpfcClass&       x, const MpfcClass& y);
MpfcClass operator * (const mpfr_t&          x, const MpfcClass& y);
MpfcClass operator * (const double&          x, const MpfcClass& y);
MpfcClass operator * (const cxsc::real&      x, const MpfcClass& y);
MpfcClass operator * (const int              x, const MpfcClass& y);
```

```
MpfcClass& operator *= (MpfcClass&, const MpfcClass&);
MpfcClass& operator *= (MpfcClass&, const cxsc::complex&);
MpfcClass& operator *= (MpfcClass&, const MpfrClass&);
MpfcClass& operator *= (MpfcClass&, const mpfr_t&);
MpfcClass& operator *= (MpfcClass&, const double&);
MpfcClass& operator *= (MpfcClass&, const cxsc::real&);
MpfcClass& operator *= (MpfcClass&, const int);
```

5.9.4 Division

Beachten Sie den Hinweis auf Seite 83.

```
MpfcClass operator / (const MpfcClass& x, const MpfcClass&       y);
MpfcClass operator / (const MpfcClass& x, const cxsc::complex&   y);
MpfcClass operator / (const MpfcClass& x, const MpfrClass&       y);
MpfcClass operator / (const MpfcClass& x, const mpfr_t&          y);
MpfcClass operator / (const MpfcClass& x, const double&          y);
MpfcClass operator / (const MpfcClass& x, const cxsc::real&      y);
MpfcClass operator / (const MpfcClass& x, const int              y);
MpfcClass operator / (const cxsc::complex&   x, const MpfcClass& y);
MpfcClass operator / (const MpfcClass&       x, const MpfcClass& y);
MpfcClass operator / (const mpfr_t&          x, const MpfcClass& y);
MpfcClass operator / (const double&          x, const MpfcClass& y);
MpfcClass operator / (const cxsc::real&      x, const MpfcClass& y);
MpfcClass operator / (const int              x, const MpfcClass& y);
```

```
MpfcClass& operator /= (MpfcClass&, const MpfcClass&);
MpfcClass& operator /= (MpfcClass&, const cxsc::complex&);
MpfcClass& operator /= (MpfcClass&, const MpfrClass&);
MpfcClass& operator /= (MpfcClass&, const mpfr_t&);
MpfcClass& operator /= (MpfcClass&, const double&);
MpfcClass& operator /= (MpfcClass&, const cxsc::real&);
MpfcClass& operator /= (MpfcClass&, const int);
```

5.10 Mathematische Funktionen

5.10.1 Standard-Implementierung

Die Implementierung komplexwertiger Funktionen mit komplexen Punktargumenten der Klasse `MpfcClass` erfolgt ganz analog zu den Funktionen der Klasse `MpfrClass`, vgl. Seite 30.

> Unabhängig von der Präzision des Eingangsarguments werden Real- und Imaginärteil des Funktionswertes mit dem Current-Rundungsmodus oder mit dem Rundungsparameter `rnd` in die Current-Precision gerundet.

Es gibt jedoch Funktionen, wie z.B. die komplexe Konjugation, bei denen man von dieser Standard-Implementierung abweichen sollte. Im folgenden Abschnitt werden diese Funktionen kurz beschrieben.

5.10.2 Davon abweichende Funktionen

```
MpfcClass conj (const MpfcClass& z);
```

Mit $z = x + i \cdot y$ ist der Rückgabewert von `conj(z)` gegeben durch $x - i \cdot y$, wobei die Präzision von x und y nicht geändert wird und daher mit der Current-Precision auch nicht übereinstimmen muss.

```
MpfrClass abs (const MpfcClass& z, RoundingMode rnd, PrecisionType prec);
```

Mit $z = x + i \cdot y$ wird $|z| = \sqrt{x^2 + y^2}$ i.a. gerundet zurückgegeben. Für die Funktion gibt es drei verschiedene Aufrufmöglichkeiten:

1. `abs (z);`
   $|z|$ wird bez. des Current-Rundungsmodus in die Current-Precision gerundet und zurückgegeben.
2. `abs (z, RoundUp);`
   $|z|$ wird in die Current-Precision aufgerundet zurückgegeben.
3. `abs (z, RoundDown, prec);`
   $|z|$ wird in ein Format mit der Präzision prec abgerundet zurückgegeben. Die Präzision des zurückgegebenen Wertes wird also i.a. von der voreingestellten Current-Precision verschieden sein!
   Will man jedoch $|z|$ **rundungsfehlerfrei** mit der gleichen Präzision von z zurückgeben, so gelingt dies in allen Fällen, unabhängig von der voreingestellten Current-Precision, mit dem Funktionsaufruf:
   ```
   abs (z, RoundNearest, z.GetPrecision());
   ```
   wobei der Rundungsmodus (hier `RoundNearest`) natürlich beliebig gesetzt werden kann. Stimmt die Präzision von z mit der Current-Precision überein, so liefert der Aufruf `abs (z);` ebenfalls den **rundungsfehlerfreien** Wert von $|z|$. In der Praxis wird die rundungsfehlerfreie Rückgabe von $|z|$ vermutlich immer im Vordergrund stehen.

Die Funktion `abs` kann also sehr flexibel eingesetzt werden und funktioniert nach 1. und 2. wie bei der Standard-Implementierung von Seite 30. Lediglich der Punkt 3. weicht von dieser Standard-Implementierung ab, um den exakten Wert von $|z|$ in jedem Fall garantieren zu können.

```
MpfrClass arg (const MpfcClass& z, RoundingMode rnd, PrecisionType prec);
```

Mit $z = x + i \cdot y$ wird $\arg(z) = \text{atan2(y,x)}$ i.a. gerundet zurückgegeben. Zur Definition von atan2(y,x) vgl. auch Seite 36. Für die Funktion gibt es ganz analog zur `abs(z)`-Funktion drei verschiedene Aufrufmöglichkeiten, siehe dazu Seite 86.

```
MpfrClass Re (const MpfcClass& z);
```

Mit $z = x + i \cdot y$ wird $\text{Re}(z) = x$ ohne Rundung in der Präzision von `z` zurückgegeben.

```
MpfrClass Im (const MpfcClass& z);
```

Mit $z = x + i \cdot y$ wird $\text{Im}(z) = y$ ohne Rundung in der Präzision von `z` zurückgegeben.

```
void times2pown (MpfcClass& op, long int k, RoundingMode rnd);
```

Die Funktion liefert mit dem Eingabewert op den Wert $op \cdot 2^k$ mit der ursprünglichen Präzision zurück, d.h. Real- und Imaginärteil werden mit 2^k multipliziert. Solange kein Über- oder Unterlauf entsteht, wird $op \cdot 2^k$ **exakt**, d.h. rundungsfehlerfrei berechnet. Tritt jedoch z.B. ein Überlauf ein, so wird gemäß rnd gerundet. Wenn rnd nicht gesetzt wird, so erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Ist dieser nicht gesetzt, so erfolgt die Rundung weg von der Null.

```
void set_nan (MpfcClass& x);
```

Setzt Real- und Imaginärteil von x auf NaN, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_inf (MpfcClass& x, const int k);
```

Setzt x auf $(\pm\texttt{Inf}, \pm\texttt{Inf})$, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss. Das Vorzeichen beim Real- und Imaginärteil wird durch k festgelegt.

```
void set_zero (MpfcClass& x);
```

Setzt x auf $(0, 0)$, wobei die Präzision von x erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

5.10.3 Elementarfunktionen

Tabelle 5.1: Elementarfunktionen mit `z` = $x + i \cdot y$ vom Typ `MpfcClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	$z^w,\ w \in \mathbb{C}$	`pow(z,w)`
$\mathrm{Re}(z) = x$	`Re(z)`	$\sin(z)$	`sin(z)`
$\mathrm{Im}(z) = y$	`Im(z)`	$\cos(z)$	`cos(z)`
$\lvert z \rvert$	`abs(z)`	$\tan(z)$	`tan(z)`
$\arg(z)$	`arg(z)`	$\cot(z)$	`cot(z)`
z^2	`sqr(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{z}$	`sqrt(z)`	$\arccos(z)$	`acos(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\arctan(z)$	`atan(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt(z,n)`	$\mathrm{arccot}(z)$	`acot(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt_all(z,n)`	$\sinh(z)$	`sinh(z)`
$\log(z)$	`ln(z)`	$\cosh(z)$	`cosh(z)`
$\log_2(z)$	`log2(z)`	$\tanh(z)$	`tanh(z)`
$\log_{10}(z)$	`log10(z)`	$\coth(z)$	`coth(z)`
e^z	`exp(z)`	$\mathrm{arsinh}(z)$	`asinh(z)`
2^z	`exp2(z)`	$\mathrm{arcosh}(z)$	`acosh(z)`
10^z	`exp10(z)`	$\mathrm{artanh}(z)$	`atanh(z)`
$z^n,\ n \in \mathbb{Z}$	`power(z,n)`	$\mathrm{arcoth}(z)$	`acoth(z)`
$z^r,\ r \in \mathbb{R}$	`pow(z,r)`		

Anmerkungen:

1. Bei den Funktionen `Re`, `Im` und `conj` werden die Rückgabewerte rundungsfehlerfrei in der Präzision des jeweiligen Eingabewertes zurückgegeben.

2. Bei den Funktionen `abs` und `arg` kann mit dem zusätzlichen Parameter `prec` die Präzision festgelegt werden, in den der Rückgabewert zu runden ist. Ohne `prec` wird in die Current-Präzision gerundet.

3. Bei den mehrdeutigen Funktionen, wie z.B `ln(z)`, `asin(z)` oder `atan(z)`, werden nur die Funktionswerte des Hauptzweiges berechnet.

4. Auf der folgenden Seite findet man das Programm `MPFR-10.cpp`, das alle dritten Wurzeln aus $z = -1 + i$ gerundet in einer Liste berechnet und den Inhalt dieser Liste auf dem Bildschirm ausgibt.

```
// MPFR-10.cpp
#include "mpfcclass.hpp"

using namespace MPFR;
using namespace cxsc;
using namespace std;

int main(void)
{
  MPFR::MpfrClass::SetCurrPrecision(70);
  cout << "GetCurrPrecision() = " << MPFR::MpfrClass::GetCurrPrecision() << endl;
  int n = 3; // Alle n-ten Wurzeln gerundet berechnen
  real re(-1), im(1);
  MpfcClass z(re, im, RoundNearest, 53);
  cout.precision(70/3.321928095); // Ausgabe mit 21 Dez.-Stellen
  cout << "z = " << z << endl;
  cout << "z.GetPrecision() = " << z.GetPrecision() << endl;
  cout << "Berechnung aller " << n << "-ten Wurzeln aus z" << endl;

  list<MpfcClass> res;
  res = sqrt_all(z, n, RoundDown);

  list<MpfcClass>::iterator pos;
  // Ausgabe der n n-ten Wurzeln:
  for (pos = res.begin(); pos != res.end(); ++pos )
  {
    cout << *pos << endl; // Jede Einschliessung in neue Zeile
    cout << "Praezision = " << (*pos).GetPrecision() << endl;
  }

  return 0;
}
```

Das Programm liefert die Ausgabe

```
GetCurrPrecision() = 70

z = (-1.0000000000000000000, 1.0000000000000000000)

z.GetPrecision() = 53
Berechnung aller 3-ten Wurzeln aus z
(7.937005259840997373 74e-1, 7.9370052598409973737 4e-1)
Praezision = 70
(-1.0842150814913511818 8, 2.9051455550725144449 5e-1)
Praezision = 70
(2.9051455550725144449 9e-1, -1.0842150814913511818 8)
Praezision = 70
```

mit den Einschließungen der drei abgerundeten (`RoundDown`) Wurzeln aus $z = -1 + i$.

6 MpfciClass-Interface für komplexe Langzahl-Intervallrechnungen in C-XSC

6.1 Grundlegendes

Das MpfciClass-Interface ist eine in `mpfciclass.hpp`, `mpfciclass.cpp` implementierte **C++**-Wrapper-Klasse MpfciClass für die C-Bibliotheken MPFR und MPFI, deren C-Funktionen über die implementierten Operatoren und Funktionen aufgerufen werden. Alle Funktionen mit der möglichen Übergabe eines PrecisionTypes besitzen als Standard den Wert von CurrPrecision, der beliebig gesetzt werden kann. Dies gilt auch für alle Konstruktoren.

6.1.1 Allgemein

Um in C-XSC das Interface verwenden zu können, muss der Header mpfciclass.hpp eingebunden werden. In ihm sind die benötigten Header-Dateien der MPFI-Bibliothek enthalten. Die MpfciClass-Klasse liegt im Namensraum "MPFI".

6.1.2 Aufbau

Die Klasse besteht intern aus zwei "mpfi_t"-Variablen. Diese dienen zum Speichern der Real- und Imaginärteil-Intervalle. Zusätzlich gibt es ein static Elemente, um die aktuelle Basis für die Ein- und Ausgabe zu speichern.

6.1.3 Präzision

Die Präzision gibt die Anzahl der binären Mantissenstellen einer MpfciClass-Variablen an. Die Real- und Imaginärteil-Intervalle erhalten stets die gleiche Präzision. Der Wert `prec` muss mindestens 2 betragen. Die Current-Precision kann global gesetzt werden; wenn dies nicht geschieht, so wird mit der Default-Precision von 53 Bits gerechnet. Unabhängig davon kann die Präzision für jede MpfciClass-Variable auch einzeln festgelegt werden.

6.1.4 Variablentyp PrecisionType

Mithilfe des Variablentyps `PrecisionType` (Name der Variablen meist prec) kann der Präzisionswert einer MpfciClass-Variablen eingestellt werden. Der Variablentyp ist ein typedef für eine "mp_prec_t"-Variable der MPFR-Bibliothek.

6.2 Rundungs und Precision Handling

```
PrecisionType GetPrecision () const;
```

Diese Memberfunktion gibt für das aktuelle Objekt die maximale Präzision seines Real- und Imaginärteil-Intervalls in Bits zurück, wobei hier für beide Intervalle sogar verschiedene Präzisionen erlaubt sind. 302 Bits entsprechen dabei $302/\log_2(10) \approx 91$ Dezimalstellen.

```
void SetPrecision (PrecisionType prec);
```

Diese Memberfunktion setzt die Präzision des aktuellen Objektes auf prec. Die Intervalle für Real- und Imaginärteil bleiben dabei nicht erhalten.

```
void RoundPrecision (PrecisionType prec);
```

Diese Memberfunktion schließt die ursprünglichen Real- und Imaginärteil-Intervalle des aktuellen Objekts durch entsprechende Intervalle mit der neuen Präzision prec ein.

```
static const PrecisionType GetCurrPrecision ();
```

Gibt die aktuelle Current-Precision in Bits zurück.

```
static void SetCurrPrecision (PrecisionType prec);
```

Setzt die Current-Precision in `MpfrClass, MpfiClass, MpfcClass, MpfciClass` auf prec. Wird die Current-Precision nicht gesetzt, so kommt die Default-Precision von 53 Bits zur Anwendung. Durch das Setzen der Current-Precision werden die Präzisionen der bis dahin benutzten Variablen jedoch **nicht** geändert.

6.3 Konstruktoren / Destruktor

6.3.1 Konstruktoren

```
MpfciClass ();
```

Der Default-Konstruktor legt ein neues Element mit der Current-Precision an.
Mit `MpfciClass y;` initialisiert man den Wert: `y` = $([NaN, NaN], [NaN, NaN])$.

```
MpfciClass (const MpfciClass& x,                          PrecisionType prec);
MpfciClass (const cxsc::cinterval& x,                     PrecisionType prec);
MpfciClass (const mpfi_t& x, const mpfi_t& y,             PrecisionType prec);
MpfciClass (const mpfi_t& x,                              PrecisionType prec);
MpfciClass (const MpfiClass& x, const MpfiClass& y,       PrecisionType prec);
MpfciClass (const MpfiClass& x,                           PrecisionType prec);

MpfciClass (const cxsc::interval& x, const cxsc::interval& y,
                                                          PrecisionType prec);

MpfciClass (const cxsc::interval& x,                      PrecisionType prec);
MpfciClass (const MPFR::MpfcClass& x,                     PrecisionType prec);
MpfciClass (const cxsc::complex& x,                       PrecisionType prec);
MpfciClass (const mpfr_t& x, const mpfr_t& y,             PrecisionType prec);
MpfciClass (const mpfr_t& x,                              PrecisionType prec);
MpfciClass (const MpfrClass& x, const MpfrClass& y,       PrecisionType prec);
MpfciClass (const MpfrClass& x,                           PrecisionType prec);
MpfciClass (const double& x, const double& y,             PrecisionType prec);
MpfciClass (const double& x,                              PrecisionType prec);

MpfciClass (const cxsc::real& x, const cxsc::real& y,
                                                          PrecisionType prec);

MpfciClass (const cxsc::real& x,                          PrecisionType prec);
MpfciClass (const int& x, const int& y,                   PrecisionType prec);
MpfciClass (const int& x,                                 PrecisionType prec);
```

Mit den Konstruktoraufrufen `MpfciClass w(z);` oder `MpficClass w(x, y);` werden die Objekte `z` bzw. `x,y` durch `w` mit der Current-Präzision eingeschlossen. Mit dem zusätzlichen Parameter `prec` lassen sich die Objekte auch mit einer anderen Präzision einschließen. Die Real- und Imaginärteil-Intervalle von `w` erhalten stets die **gleiche** Präzision.

Mögliche Konstruktor-Aufrufe sind:

1. `MpfciClass w(z);` `MpfciClass w(x, y);`
2. `MpfciClass w(z, 3);` `MpfcClass w(x, y, 3);`

Zu 1. Die Objekte `z,x,y` werden mit der Current-Precision durch die Real- und Imaginärteil-Intervalle des Objekts `w` eingeschlossen.

Zu 2. Die Objekte `z,x,y` werden durch die Real- und Imaginärteil-Intervalle des Objekts `w` mit der (sehr kleinen) Präzision $prec = 3$ entsprechend grob eingeschlossen.

Die oberen 19 Konstruktoren erlauben also eine sehr flexible Initialisierung von `MpfciClass`-Objekten.

```
MpfciClass::MpfciClass (const std::string& s, PrecisionType prec);
```

Der Aufruf `MpfciClass w(s);` liefert ein Objekt `w` vom Typ `MpfciClass` mit Real- und Imaginärteil-Intervallen, welche die entsprechenden Intervalle des String s in der Current-Precision garantiert einschließen.

Der Aufruf `MpfcClass w(s, 140);` liefert ein Objekt `w` vom Typ `MpfciClass` mit Real- und Imaginärteil-Intervallen der gleichen Präzision $prec = 140$, welche die entsprechenden Intervalle des String s garantiert einschließen. Der Präzision von 140 Bits entsprechen dabei $140/\log_2(10) = 140/3.32192809\ldots \approx 42$ Dezimalstellen. Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in eine binäre Zahl konvertiert werden kann, so dass daher bei beiden Konstruktor-Aufrufen Rundungen nach außen i.a. nicht zu vermeiden sind.
Der String s muss das Format `([Number,Number],[Number,Number])` besitzen.

6.3.2 Destruktor

```
~MpfciClass ();
```

Der Speicher für das Objekt wird freigegeben. Dieser Destruktor muss nicht explizit aufgerufen werden!

6.4 Zuweisungs-Operatoren

Unabhängig von der Präzision des rechten Operanden erhalten bei allen folgenden Zuweisungsoperatoren die Real- und Imaginärteil-Intervalle des linken Operanden `y` stets die Current-Precision und schließen die Zahlen oder Intervalle des jeweiligen rechten Operanden op optimal ein.

```
MpfciClass& operator = (const MpfciClass& op);
MpfciClass& operator = (const cxsc::cinterval& op);
MpfciClass& operator = (const MPFR::MpfcClass& op);
MpfciClass& operator = (const cxsc::complex& op);
MpfciClass& operator = (const mpfi_t& op);
MpfciClass& operator = (const MPFI::MpfiClass& op);
MpfciClass& operator = (const cxsc::interval& op);
MpfciClass& operator = (const mpfr_t& op);
MpfciClass& operator = (const MPFR::MpfrClass& op);
MpfciClass& operator = (const double& op);
MpfciClass& operator = (const cxsc::real& op);
MpfciClass& operator = (const int& op);
MpfciClass& operator = (const std::string& op);
```

Anmerkunken:

- Ist z.B. op ein `real`-Wert und wurde die Current-Precision mit `SetCurrPrecision` zu klein gewählt, so ist das Realteil-Intervall des linken Operanden `y` i.a. kein Punktinterval. Ist die Current-Precision jedoch größer oder gleich 53, so ist das einschließende Realteil-Intervall `y` stets ein Punktintervall, das op optimal einschließt.
- Ist op ein String, so muss dieser die Form `([Number,Number],[Number,Number])` haben. Ist z.B. *op* = `([0.1,0.1],[-1.1,-1.1])` so können die Real- und Imaginärteil-Intervalle von `y` bei noch so großer Current-Precision keine Punktintervalle sein, da 0.1 und 1.1 im vorliegenden Binärsystem nicht exakt darstellbar sind.

6.5 Eingabe / Ausgabe

```
std::ostream& operator << (std::ostream& os, const MpfciClass& z);
```

Ermöglicht die Ausgabe einer MpfciClass-Variablen z über den Standard-Ausgabestrom "cout". Die Anzahl der Nachkommastellen ist identisch mit dem Wert, der in cout.precision() eingestellt ist. Für die dezimale Ausgabe liefert

```
cout.precision(z.GetPrecision()/3.32192809...);
```

die entsprechende Dezimalstellenzahl. Die Real- und Imaginärteil-Intervalle von z werden in der mit `SetBase(k)` voreingestellten Basis im Format

```
([Number,Number],[Number,Number])
```

ausgegeben, wobei die Real- und Imaginärteil-Intervalle von z beide durch die Ausgabeintervalle mit der durch `cout.precision(...)` voreingestellten Ausgabe-Präzision eingeschlossen werden.
Für die meist dezimale Ausgabe ist natürlich mit `SetBase(10)` die richtige Ausgabebasis `k` = 10 zu wählen.

```
std::istream& operator >> (std::istream& is, MpfiClass& z);
```

Ermöglicht das Einlesen eines MpfciClass-Objekts über den Standard-Eingabestrom "cin" in das Objekt z. Die eingegebenen Real- und Imaginärteil-Intervalle sind auf keine Stellenzahl begrenzt. Nur das folgende Eingabeformat ist zulässig:

```
([Number,Number],[Number,Number])
```

Die obigen Real- und Imaginärteil-Intervalle werden durch die binären Real- und Imaginärteil-Intervalle des Objekts z in der Current-Präzision optimal eingeschlossen.
Vorsicht: Leerzeichen sind nicht erlaubt, und die runden Klammern müssen beide gesetzt werden, sonst erfolgt eine entsprechende Fehlermeldung.

```
static const int GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen.

```
static void SetBase (int b);
```

Setzt die aktuelle Basis auf b. Dies muss ein Wert zwischen 2 und 36 sein. Die Basis hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die internen Rechnungen erfolgen stets im Binärsystem!

6.6 Typ-Umwandlungen

Um ein flexibles Arbeiten zwischen C-XSC und der Klasse `MpfciClass` zu ermöglichen, wurden möglichst viele Typ-Umwandlungs-Funktionen bereitgestellt:

6.6.1 `real, interval, ...` → MPFCI

```
MpfciClass cinterval2Mpfci (const cxsc::cinterval& op);
MpfciClass mpfi_t2Mpfci    (const mpfi_t& op);
MpfciClass MpfiClass2Mpfci (const MpfiClass& op);
MpfciClass interval2Mpfci  (const cxsc::interval& op);
MpfciClass mpfr_t2Mpfci    (const mpfr_t& op);
MpfciClass MpfrClass2Mpfci (const MPFR::MpfrClass::MpfrClass& op)
MpfciClass real2Mpfci      (const cxsc::real& op);
MpfciClass double2Mpfci    (const double& op);
MpfciClass int2Mpfci       (const int& op);
```

Obige Funktionen liefern mit dem jeweiligen Eingabewert op einen Rückgabewert vom Typ `MpfciClass` in einer Präzision, die gewährleistet, dass der Rückgabewert genau dem Wert von op entspricht. Die Präzision des Rückgabewertes wird also i.a. nicht mit der Current-Precision übereinstimmen! Die obigen zwölf Funktionen kommen u.a. bei den Vergleichsoperatoren zur Anwendung.

6.6.2 MPFCI → `cinterval`

Die folgende Funktion liefert mit einem Objekt op vom Typ `MpfciClass` eine i.a. **gerundete** Einschließung von op vom C-XSC Typ `cinterval;`

```
cxsc::cinterval to_cinterval(const MpfciClass& op);
```

Eine optimale Einschließung von op wird nur erreicht, wenn op im IEEE-`double`-Format darstellbar ist.

6.6.3 MPFCI → `mpfi_t`

```
mpfi_t& MpfciClass::GetValueRe();
```

Die Memberfunktion `GetValueRe()` liefert für das jeweils aktuelle Objekt vom Typ `MpfciClass` eine Referenz auf sein Realteilintervall `mpfi_re` vom Typ `mpfi_t`. Damit kann dieses Realteilintervall z.B. an einen nicht-konstanten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
mpfr_t& MpfcClass::GetValueIm();
```

Die Memberfunktion `GetValueIm()` liefert für das jeweils aktuelle Objekt vom Typ `MpfciClass` eine Referenz auf sein Imaginärteilintervall `mpfi_im` vom Typ `mpfi_t`. Damit kann dieses Imaginärteilintervall z.B. an einen nicht-konstanten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
const mpfi_t& getvalueRe(const MpfciClass& z)
```

Die Funktion `getvalueRe(...)` liefert für das als `const` definierte Objekt `z` vom Typ `MpfciClass` eine Referenz auf sein Realteilintervall `mpfi_re` vom Typ `mpfi_t`. Damit kann dieses Realteilintervall z.B. an einen als `const` deklarierten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

```
const mpfi_t& getvalueIm(const MpfciClass& z)
```

`getvalueIm(...)` liefert für das als `const` definierte Objekt `z` vom Typ `MpfciClass` eine Referenz auf sein Imaginärteilintervall `mpfi_im` vom Typ `mpfi_t`. Damit kann dieses Imaginärteilintervall z.B. an einen als `const` deklarierten Referenz-Parameter gleichen Typs in einer Parameterliste an eine Funktion übergeben werden.

6.6.4 `mpfi_t` → MPFCI

```
void MpfciClass::SetValueRe(const mpfi_t& v)
```

Mit dieser Memberfunktion wird das Realteilintervall des aktuellen Objekts exakt auf v gesetzt. Das Imaginärteilintervall des aktuellen Objekts bleibt dabei erhalten. Die Praezision des aktuellen Objekts wird gesetzt auf das Maximum der Praezisionen von `v` und `mpfi_im`, d.h. die Präzisionen von Real- und Imaginärteilintervall sind nach dem Funktionsaufruf wieder gleich.

```
void MpfciClass::SetValueIm(const mpfi_t& v)
```

Mit dieser Memberfunktion wird das Imaginärteilintervall des aktuellen Objekts exakt auf v gesetzt. Das Realteilintervall des aktuellen Objekts bleibt dabei erhalten. Die Praezision des aktuellen Objekts wird gesetzt auf das Maximum der Praezisionen von `v` und `mpfi_re`, d.h. die Präzisionen von Real- und Imaginärteilintervall sind nach dem Funktionsaufruf wieder gleich.

```
void MpfciClass::SetValue(const mpfi_t& re, const mpfi_t& im)
```

Mit dieser Memberfunktion werden Real- und Imaginärteilintervall des aktuellen Objekts exakt auf re bzw. im gesetzt. Das aktuelle Objekt erhält als Präzision das Maximum der Präzisionen von re und im, d.h. die Präzisionen von Real- und Imaginärteilintervall sind nach dem Funktionsaufruf wieder gleich.

6.6.5 `string` → MPFCI

```
MpfciClass string2Mpfci(const std::string& op, PrecisionType prec);
```

Der Aufruf `string2Mpfci(op);` liefert ein Ojekt vom Typ `MpfciClass` dessen Real- und Imaginärteil-Intervalle mit der voreingestellten Current-Präzision die Real- und Imaginärteil-Intervalle von op einschließt.
Der zweite mögliche Aufruf `string2Mpfci(op,140);` liefert ganz analog eine Einschließung der String-Intervalle mit einer Präzision von jetzt 140 Bits.
Zu beachten ist, dass ein dezimaler String i.a. nicht rundungsfehlerfrei in ein Binärformat konvertiert werden kann, so dass i.a. Überschätzungen nicht zu vermeiden sind, es werden jedoch stets garantierte Einschließungen von op zurückgegeben. Eine weitere Möglichkeit, einen String in ein `MpfciClass`-Objekt zu verwandeln, besteht in einem entsprechenden Konstruktor-Aufruf, vgl. Seite 94. Dort findet man auch Hinweise auf die möglichen Intervall-String-Formate.

6.6.6 MPFCI → `string`

```
std::string to_string(const MpfciClass& z, PrecisionType prec);
```

Die Real- und Imaginärteilintervalle des Objekts $z = x + i \cdot y$ werden durch den zurückgegebenen String `s` mit prec Dezimalstellen eingeschlossen, wenn Base gleich 10 ist. Der String besitzt das Format

```
([Number,Number],[Number,Number])
```

wobei intern keine Leerzeichen auftreten. Wählt man prec hinreichend groß, so stellt der String die Real- und Imaginärteil-Intervalle von z **exakt** dar, weil eine Binärzahl stets exakt in eine Dezimalzahl umgewandelt werden kann.

Wird prec nicht angegeben, so werden x, y durch String-Intervalle eingeschlossen, die bei Base=10 soviele Dezimalstellen besitzen, wie es der Präzision von `z` entspricht. Ist die Präzision von `z` z.B. 302, so werden im String Real- und Imaginärteil-Intervalle mit $302/\log_2(10) = 302/3.32192809... \approx 91$ Dezimalstellen generiert.

6.6.7 MPFCI → MPFCI

Beachten Sie bitte, dass bei einer Wertzuweisung an eine `MpfciClass`-Variable mit Hilfe des Operators `=` der linke Operand mit seinem Real- und Imaginärteilintervall die entsprechenden Real- und Imaginärteilwerte des rechten Operanden mit der Current-Precision stets einschließt, vgl. dazu auch Seite 95. Will man jedoch abweichend von dieser Current-Precision mit einer anderen Präzision einschließen, so kann dies mit folgender Funktion ohne Rückgabewert realisiert werden:

```
void set_Mpfci (MpfciClass& op, const MpfciClass& op1, PrecisionType prec);
```

Folgende Funktionsaufrufe sind möglich:

1. `set_Mpfci (op, op1, prec);`
   op schließt die Real- und Imaginärteilintervalle von op1 mit der Präzision prec ein. Setzt man prec gleich der Präzision von op1, so erhält op den **exakten** Wert von op1.

2. `set_Mpfci (op, op1);`
   op schließt die Real- und Imaginärteilintervalle von op1 mit der Current-Precision ein.

6.7 Abfragen

Bei allen folgenden Abfragefunktionen muss die Präzision von x nicht mit der Current-Precision übereinstimmen.

```
bool isNan      (const MpfciClass& x);
bool isInf      (const MpfciClass& x);
bool isBounded  (const MpfciClass& x);
bool isZero     (const MpfciClass& x);
bool isPoint    (const MpfciClass& x);
```

`isNan` und `isInf` überprüfen, ob Real- oder Imaginärteil von x gleich NaN bzw. ±Inf sind. `isBounded` überprüft, ob Real- und Imaginärteil von x ein normales `MpfiClass` Intervall ungleich NaN und ungleich ±Inf ist. `isZero` überprüft, ob das Real- und Imaginärteil-Intervall von x gleich Null ist, und `isPoint` überprüft, ob das Real- und Imaginärteil-Intervall von x jeweils ein Punktintervall ist.

6.8 Vergleiche

Es werden die üblichen Vergleichsoperatoren implementiert, wobei wenigstens ein Operand vom Typ `MpfciClass` sein muss. Die Operanden können unterschiedliche Präzisionen besitzen.

6.8.1 Vergleichsoperatoren =, !=, <, <=,

```
bool operator == (const MpfciClass& x, const MpfciClass&       y);
bool operator == (const MpfciClass& x, const cxsc::cinterval&  y);
bool operator == (const MpfciClass& x, const MPFR::MpfcClass&  y);
bool operator == (const MpfciClass& x, const cxsc::complex&    y);
bool operator == (const MpfciClass& x, const mpfi_t&           y);
bool operator == (const MpfciClass& x, const MpfiClass&        y);
bool operator == (const MpfciClass& x, const cxsc::interval&   y);
bool operator == (const MpfciClass& x, const mpfr_t&           y);
bool operator == (const MpfciClass& x, const MPFR::MpfrClass&  y);
bool operator == (const MpfciClass& x, const double&           y);
bool operator == (const MpfciClass& x, const cxsc::real&       y);
bool operator == (const MpfciClass& x, const int               y);
bool operator == (const cxsc::cinterval&  x, const MpfciClass& y);
bool operator == (const MPFR::MpfcClass&  x, const MpfciClass& y);
bool operator == (const cxsc::complex&    x, const MpfciClass& y);
bool operator == (const mpfi_t&           x, const MpfciClass& y);
bool operator == (const MpfiClass&        x, const MpfciClass& y);
bool operator == (const cxsc::interval&   x, const MpfciClass& y);
bool operator == (const mpfr_t&           x, const MpfciClass& y);
bool operator == (const MpfrClass&        x, const MpfciClass& y);
bool operator == (const double&           x, const MpfciClass& y);
bool operator == (const cxsc::real&       x, const MpfciClass& y);
bool operator == (const int               x, const MpfciClass& y);
```

```
bool operator != (const MpfciClass& x, const MpfciClass&       y);
bool operator != (const MpfciClass& x, const cxsc::cinterval&  y);
bool operator != (const MpfciClass& x, const MPFR::MpfcClass&  y);
bool operator != (const MpfciClass& x, const cxsc::complex&    y);
bool operator != (const MpfciClass& x, const mpfi_t&           y);
bool operator != (const MpfciClass& x, const MpfiClass&        y);
bool operator != (const MpfciClass& x, const cxsc::interval&   y);
bool operator != (const MpfciClass& x, const mpfr_t&           y);
bool operator != (const MpfciClass& x, const MPFR::MpfrClass&  y);
bool operator != (const MpfciClass& x, const double&           y);
bool operator != (const MpfciClass& x, const cxsc::real&       y);
bool operator != (const MpfciClass& x, const int               y);
bool operator != (const cxsc::cinterval&  x, const MpfciClass& y);
bool operator != (const MPFR::MpfcClass&  x, const MpfciClass& y);
bool operator != (const cxsc::complex&    x, const MpfciClass& y);
bool operator != (const mpfi_t&           x, const MpfciClass& y);
bool operator != (const MpfiClass&        x, const MpfciClass& y);
bool operator != (const cxsc::interval&   x, const MpfciClass& y);
bool operator != (const mpfr_t&           x, const MpfciClass& y);
bool operator != (const MpfrClass&        x, const MpfciClass& y);
bool operator != (const double&           x, const MpfciClass& y);
bool operator != (const cxsc::real&       x, const MpfciClass& y);
bool operator != (const int               x, const MpfciClass& y);
```

```
bool operator < (const MpfciClass& x, const MpfciClass& y);
bool operator < (const MpfciClass& x, const cxsc::cinterval& y);
bool operator < (const MpfciClass& x, const mpfi_t& y);
bool operator < (const MpfciClass& x, const MpfiClass& y);
bool operator < (const MpfciClass& x, const cxsc::interval& y);
bool operator < (const cxsc::cinterval& x, const MpfciClass& y);
bool operator < (const mpfi_t& x,          const MpfciClass& y);
bool operator < (const MpfiClass& x,       const MpfciClass& y);
bool operator < (const cxsc::interval& x,  const MpfciClass& y);
```

'interval' < 'MpfciClass': 'interval' liegt ganz im Innern von 'MpfciClass', wobei $[0,0] \subset$ `Im(y)` erfüllt sein muss.

```
bool operator < (const MPFR::MpfcClass& x, const MpfciClass& y);
bool operator < (const cxsc::complex& x,   const MpfciClass& y);
bool operator < (const mpfr_t& x,          const MpfciClass& y);
bool operator < (const MPFR::MpfrClass& x, const MpfciClass& y);
bool operator < (const double& x,          const MpfciClass& y);
bool operator < (const cxsc::real& x,      const MpfciClass& y);
bool operator < (const int x,              const MpfciClass& y);
```

'real' < 'MpfciClass': 'real' liegt ganz im Innern von `Re(y)`, wobei 0 im Innern von `Im(y)` liegen muss.

```
bool operator <= (const MpfciClass& x, const MpfciClass& y);
bool operator <= (const MpfciClass& x, const cxsc::cinterval& y);
bool operator <= (const MpfciClass& x, const mpfi_t& y);
bool operator <= (const MpfciClass& x, const MpfiClass& y);
bool operator <= (const MpfciClass& x, const cxsc::interval& y);
bool operator <= (const cxsc::cinterval& x, const MpfciClass& y);
bool operator <= (const mpfi_t& x,          const MpfciClass& y);
bool operator <= (const MpfiClass& x,       const MpfciClass& y);
bool operator <= (const cxsc::interval& x,  const MpfciClass& y);
```

'interval' <= 'MpfciClass': Es gilt $x \subseteq$ `Re(y)` und $[0,0] \subseteq$ `Im(y)`.

```
bool operator <= (const MPFR::MpfcClass& x, const MpfciClass& y);
bool operator <= (const cxsc::complex& x,   const MpfciClass& y);
bool operator <= (const mpfr_t& x,          const MpfciClass& y);
bool operator <= (const MPFR::MpfrClass& x, const MpfciClass& y);
bool operator <= (const double& x,          const MpfciClass& y);
bool operator <= (const cxsc::real& x,      const MpfciClass& y);
bool operator <= (const int x,              const MpfciClass& y);
```

'real' <= 'MpfciClass': Es gilt $x \in$ `Re(y)` und $0 \in$ `Im(y)`.

```
bool operator > (const MpfciClass& x, const MpfciClass& y);
bool operator > (const MpfciClass& x, const cxsc::cinterval& y);
bool operator > (const MpfciClass& x, const mpfi_t& y);
bool operator > (const MpfciClass& x, const MpfiClass& y);
bool operator > (const MpfciClass& x, const cxsc::interval& y);
bool operator > (const cxsc::cinterval& x, const MpfciClass& y);
bool operator > (const mpfi_t& x,          const MpfciClass& y);
bool operator > (const MpfiClass& x,       const MpfciClass& y);
bool operator > (const cxsc::interval& x,  const MpfciClass& y);
```

'interval' > 'MpfciClass': 'MpfciClass' liegt ganz im Innern von 'interval', wobei `Im(y)` $= [0,0]$ erfüllt sein muss.

```
bool operator > (const MpfciClass& x, const MPFR::MpfcClass& y);
bool operator > (const MpfciClass& x, const cxsc::complex& y);
bool operator > (const MpfciClass& x, const mpfr_t& y);
bool operator > (const MpfciClass& x, const MPFR::MpfrClass& y);
bool operator > (const MpfciClass& x, const double& y);
bool operator > (const MpfciClass& x, const cxsc::real& y);
bool operator > (const MpfciClass& x, const int y);
```

'MpfciClass' > 'real': Es gilt y liegt ganz im Innern von `Re(x)` und 0 liegt ganz im Innern von `Im(x)`.

```
bool operator >= (const MpfciClass& x, const MpfciClass& y);
bool operator >= (const MpfciClass& x, const cxsc::cinterval& y);
bool operator >= (const MpfciClass& x, const mpfi_t& y);
bool operator >= (const MpfciClass& x, const MpfiClass& y);
bool operator >= (const MpfciClass& x, const cxsc::interval& y);
bool operator >= (const cxsc::cinterval& x, const MpfciClass& y);
bool operator >= (const mpfi_t& x,          const MpfciClass& y);
bool operator >= (const MpfiClass& x,       const MpfciClass& y);
bool operator >= (const cxsc::interval& x,  const MpfciClass& y);
```

'interval' >= 'MpfciClass': Es gilt `Re(y)` $\subset x$ und `Im(y)` $= [0,0]$.

```
bool operator >= (const MpfciClass& x, const MPFR::MpfcClass& y);
bool operator >= (const MpfciClass& x, const cxsc::complex& y);
bool operator >= (const MpfciClass& x, const mpfr_t& y);
bool operator >= (const MpfciClass& x, const MPFR::MpfrClass& y);
bool operator >= (const MpfciClass& x, const double& y);
bool operator >= (const MpfciClass& x, const cxsc::real& y);
bool operator >= (const MpfciClass& x, const int y);
```

'MpfciClass' >= 'real': Es gilt $y \in$ `Re(y)` und $0 \in$ `Im(y)`.

```
int in (const MpfciClass& x, const MpfciClass& y);
int in (const cxsc::cinterval& x, const MpfciClass& y);
int in (const MPFR::MpfcClass::MpfcClass& x, const MpfciClass& y);
int in (const cxsc::complex& x, const MpfciClass& y);
int in (const MpfiClass& x, const MpfciClass& y);
int in (const mpfi_t& x, const MpfciClass& y);
int in (const cxsc::interval& x, const MpfciClass& y);
int in (const MPFR::MpfrClass::MpfrClass& x, const MpfciClass& y);
int in (const mpfr_t& x, const MpfciClass& y);
int in (const double& x, const MpfciClass& y);
int in (const cxsc::real& x, const MpfciClass& y);
int in (const int& x, const MpfciClass& y);
```

Zurückgegeben wird die Eins, wenn `x` ganz im Innern von `y` enthalten ist, sonst wird die Null zurückgegeben. Ist einer der Operanden ein `NaN` oder sind beide Operanden unbegrenzt, so wird ebenfalls die Null zurückgegeben. Die Präzisionen beider Operanden können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen.

6.9 Durchschnitt

Berechnet wird der Durchschnitt zweier komplexer Intervalle. Ist einer der nachfolgenden Operanden eine reelle oder komplexe Zahl, so ist diese als Punktintervall zu interpretieren. Die Präzisionen der Intervalle oder Zahlen können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen. Die Präzision des zurückgegebenen komplexen Durchschnitt-Intervalls ist das Maximum der Präzisionen beider Operanden und muss daher mit der Current-Präzision nicht übereinstimmen. Dadurch wird der Durchschnitt stets **rundungsfehlerfrei** berechnet. Ist der Durchschnitt leer, so wird ([NaN,NaN],[NaN,NaN]) zurückgegeben.

```
MpfciClass operator & (const MpfciClass& z, const MpfciClass& x);

MpfciClass operator & (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator & (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator & (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator & (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator & (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator & (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator & (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator & (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator & (const MpfciClass& z, const double&          x);
MpfciClass operator & (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator & (const MpfciClass& z, const int              x);

MpfciClass operator & (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator & (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator & (const cxsc::complex&   x, const MpfciClass& z);
MpfciClass operator & (const mpfi_t&          x, const MpfciClass& z);
MpfciClass operator & (const MpfiClass&       x, const MpfciClass& z);
MpfciClass operator & (const cxsc::interval&  x, const MpfciClass& z);
MpfciClass operator & (const mpfr_t&          x, const MpfciClass& z);
MpfciClass operator & (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator & (const double&          x, const MpfciClass& z);
MpfciClass operator & (const cxsc::real       x, const MpfciClass& z);
MpfciClass operator & (const int&             x, const MpfciClass& z);
```

```
MpfciClass & operator &= (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator &= (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator &= (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator &= (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator &= (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator &= (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator &= (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator &= (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator &= (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator &= (MpfciClass& z, const double&          x);
MpfciClass & operator &= (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator &= (MpfciClass& z, const int              x);
```

Die Anweisung `z &= x;` liefert an `z` den Durchschnitt (`z & x`), wobei die neue Präzision von `z` gleich dem Maximum der Präzisionen der ursprünglichen Operanden `z` und `x` ist. Auch hier wird erreicht, dass der Durchschnitt stets **rundungsfehlerfrei** an `z` zurückgegeben wird.

Hinweis:

Wenn beide Operanden als Präzision die Current-Präzision besitzen, so wird der Durchschnitt stets **rundungsfehlerfrei** in der gleichen Current-Präzision zurückgegeben.

6.10 Konvexe Hülle

Berechnet wird die konvexe Hülle zweier komplexer Intervalle. Ist einer der nachfolgenden Operanden eine reelle oder komplexe Zahl, so ist diese jeweils als Punktintervall zu interpretieren. Die Präzisionen der Intervalle oder Zahlen können unterschiedlich sein und müssen mit der Current-Präzision nicht übereinstimmen. Die Präzision des zurückgegebenen Hüllen-Intervalls ist das Maximum der Präzisionen beider Operanden und muss mit der Current-Präzision nicht übereinstimmen. Dadurch wird die konvexe Hülle stets **rundungsfehlerfrei** berechnet.

```
MpfciClass operator | (const MpfciClass& z, const MpfciClass& w);
MpfciClass operator | (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator | (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator | (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator | (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator | (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator | (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator | (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator | (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator | (const MpfciClass& z, const double&          x);
MpfciClass operator | (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator | (const MpfciClass& z, const int             x);

MpfciClass operator | (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator | (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator | (const cxsc::complex& x,   const MpfciClass& z);
MpfciClass operator | (const mpfi_t& x,          const MpfciClass& z);
MpfciClass operator | (const MpfiClass& x,       const MpfciClass& z);
MpfciClass operator | (const cxsc::interval& x,  const MpfciClass& z);
MpfciClass operator | (const mpfr_t& x,          const MpfciClass& z);
MpfciClass operator | (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator | (const double& x,          const MpfciClass& z);
MpfciClass operator | (const cxsc::real& x,      const MpfciClass& z);
MpfciClass operator | (const int x,             const MpfciClass& z);
```

```
MpfciClass & operator |= (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator |= (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator |= (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator |= (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator |= (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator |= (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator |= (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator |= (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator |= (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator |= (MpfciClass& z, const double&          x);
MpfciClass & operator |= (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator |= (MpfciClass& z, const int             x);
```

Die Anweisung `z |= x;` liefert an `z` die konvexe Hülle (`z | x`), wobei die neue Präzision von `z` gleich dem Maximum der Präzisionen der ursprünglichen Operanden `z` und `x` ist. Auch hier wird erreicht, dass die konvexe Hülle stets **rundungsfehlerfrei** an `z` zurückgegeben wird.

Hinweis:

Wenn beide Operanden als Präzision die Current-Präzision besitzen, so wird die konvexe Hülle stets **rundungsfehlerfrei** in der gleichen Current-Präzision zurückgegeben.

6.11 Arithmetische Operatoren

Für alle arithmetischen Operationen mit komplexen Intervall-Operanden gilt:

> Das exakte Ergebnis einer arithmetischen Operation mit komplexen Intervall-Operanden wird unabhängig von der Präzision dieser Operanden mit der voreingestellten Current-Präzision außer bei Multiplikation und Division optimal eingeschlossen.
> Die Operatoren $\circledcirc=$, mit $\circledcirc \in \{+,-,\cdot,/\}$, bedeuten: $u \circledcirc= v \iff u = u \circledcirc v$.
> Dabei wird $u \circledcirc v$ durch u stets eingeschlossen, wobei u als Präzision die Current-Präzision erhält, d.h. die Präzision von u kann sich ändern.

6.11.1 Addition

```
MpfciClass operator + (const MpfciClass& z, const MpfciClass& w);
MpfciClass operator + (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator + (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator + (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator + (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator + (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator + (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator + (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator + (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator + (const MpfciClass& z, const double&          x);
MpfciClass operator + (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator + (const MpfciClass& z, const int              x);

MpfciClass operator + (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator + (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator + (const cxsc::complex& x,   const MpfciClass& z);
MpfciClass operator + (const mpfi_t& x,          const MpfciClass& z);
MpfciClass operator + (const MpfiClass& x,       const MpfciClass& z);
MpfciClass operator + (const cxsc::interval& x,  const MpfciClass& z);
MpfciClass operator + (const mpfr_t& x,          const MpfciClass& z);
MpfciClass operator + (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator + (const double& x,          const MpfciClass& z);
MpfciClass operator + (const cxsc::real& x,      const MpfciClass& z);
MpfciClass operator + (const int x,              const MpfciClass& z);
```

```
MpfciClass & operator += (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator += (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator += (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator += (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator += (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator += (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator += (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator += (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator += (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator += (MpfciClass& z, const double&          x);
MpfciClass & operator += (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator += (MpfciClass& z, const int              x);
```

6.11.2 Subtraktion

Beachten Sie die Bemerkungen auf Seite 105 oben.

```
MpfciClass operator - (const MpfciClass& z, const MpfciClass& w);
MpfciClass operator - (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator - (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator - (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator - (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator - (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator - (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator - (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator - (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator - (const MpfciClass& z, const double&          x);
MpfciClass operator - (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator - (const MpfciClass& z, const int              x);

MpfciClass operator - (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator - (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator - (const cxsc::complex& x,   const MpfciClass& z);
MpfciClass operator - (const mpfi_t& x,          const MpfciClass& z);
MpfciClass operator - (const MpfiClass& x,       const MpfciClass& z);
MpfciClass operator - (const cxsc::interval& x,  const MpfciClass& z);
MpfciClass operator - (const mpfr_t& x,          const MpfciClass& z);
MpfciClass operator - (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator - (const double& x,          const MpfciClass& z);
MpfciClass operator - (const cxsc::real& x,      const MpfciClass& z);
MpfciClass operator - (const int x,              const MpfciClass& z);
```

```
MpfciClass & operator -= (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator -= (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator -= (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator -= (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator -= (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator -= (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator -= (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator -= (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator -= (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator -= (MpfciClass& z, const double&          x);
MpfciClass & operator -= (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator -= (MpfciClass& z, const int              x);
```

6.11.3 Multiplikation

Beachten Sie die Bemerkungen auf Seite 105 oben.

```
MpfciClass operator * (const MpfciClass& z, const MpfciClass& w);
MpfciClass operator * (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator * (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator * (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator * (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator * (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator * (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator * (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator * (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator * (const MpfciClass& z, const double&          x);
MpfciClass operator * (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator * (const MpfciClass& z, const int              x);

MpfciClass operator * (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator * (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator * (const cxsc::complex& x,   const MpfciClass& z);
MpfciClass operator * (const mpfi_t& x,          const MpfciClass& z);
MpfciClass operator * (const MpfiClass& x,       const MpfciClass& z);
MpfciClass operator * (const cxsc::interval& x,  const MpfciClass& z);
MpfciClass operator * (const mpfr_t& x,          const MpfciClass& z);
MpfciClass operator * (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator * (const double& x,          const MpfciClass& z);
MpfciClass operator * (const cxsc::real& x,      const MpfciClass& z);
MpfciClass operator * (const int x,              const MpfciClass& z);
```

```
MpfciClass & operator *= (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator *= (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator *= (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator *= (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator *= (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator *= (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator *= (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator *= (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator *= (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator *= (MpfciClass& z, const double&          x);
MpfciClass & operator *= (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator *= (MpfciClass& z, const int              x);
```

6.11.4 Division

Beachten Sie die Bemerkungen auf Seite 105 oben.

```
MpfciClass operator / (const MpfciClass& z, const MpfciClass& w);
MpfciClass operator / (const MpfciClass& z, const cxsc::cinterval& x);
MpfciClass operator / (const MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass operator / (const MpfciClass& z, const cxsc::complex&   x);
MpfciClass operator / (const MpfciClass& z, const mpfi_t&          x);
MpfciClass operator / (const MpfciClass& z, const MpfiClass&       x);
MpfciClass operator / (const MpfciClass& z, const cxsc::interval&  x);
MpfciClass operator / (const MpfciClass& z, const mpfr_t&          x);
MpfciClass operator / (const MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass operator / (const MpfciClass& z, const double&          x);
MpfciClass operator / (const MpfciClass& z, const cxsc::real&      x);
MpfciClass operator / (const MpfciClass& z, const int              x);

MpfciClass operator / (const cxsc::cinterval& x, const MpfciClass& z);
MpfciClass operator / (const MPFR::MpfcClass& x, const MpfciClass& z);
MpfciClass operator / (const cxsc::complex& x,   const MpfciClass& z);
MpfciClass operator / (const mpfi_t& x,          const MpfciClass& z);
MpfciClass operator / (const MpfiClass& x,       const MpfciClass& z);
MpfciClass operator / (const cxsc::interval& x,  const MpfciClass& z);
MpfciClass operator / (const mpfr_t& x,          const MpfciClass& z);
MpfciClass operator / (const MPFR::MpfrClass& x, const MpfciClass& z);
MpfciClass operator / (const double& x,          const MpfciClass& z);
MpfciClass operator / (const cxsc::real& x,      const MpfciClass& z);
MpfciClass operator / (const int x,              const MpfciClass& z);
```

```
MpfciClass & operator /= (MpfciClass& z, const MpfciClass&       x);
MpfciClass & operator /= (MpfciClass& z, const cxsc::cinterval& x);
MpfciClass & operator /= (MpfciClass& z, const MPFR::MpfcClass& x);
MpfciClass & operator /= (MpfciClass& z, const cxsc::complex&   x);
MpfciClass & operator /= (MpfciClass& z, const mpfi_t&          x);
MpfciClass & operator /= (MpfciClass& z, const MpfiClass&       x);
MpfciClass & operator /= (MpfciClass& z, const cxsc::interval&  x);
MpfciClass & operator /= (MpfciClass& z, const mpfr_t&          x);
MpfciClass & operator /= (MpfciClass& z, const MPFR::MpfrClass& x);
MpfciClass & operator /= (MpfciClass& z, const double&          x);
MpfciClass & operator /= (MpfciClass& z, const cxsc::real&      x);
MpfciClass & operator /= (MpfciClass& z, const int              x);
```

6.12 Mathematische Funktionen

6.12.1 Standard-Implementierung

Die Implementierung komplexwertiger Intervallfunktionen mit komplexen Intervallargumenten der Klasse `MpfciClass` erfolgt ganz analog zu den Funktionen der Klasse `MpfiClass`, vgl. dazu die Seite 62.

> Unabhängig von der Präzision des Eingangsarguments werden die Real- und Imaginärteil-Intervalle des exakten Funktionswertes durch die entsprechenden Ergebnisintervalle in der voreingestellten Current-Präzision garantiert und nahezu optimal eingeschlossen.

Es gibt jedoch einige Funktionen, wie z.B. die komplexe Konjugation oder die Real- und Imaginärteil-Funktionen, bei denen man von dieser Standard-Implementierung abweichen sollte, d.h. die Präzision der Ergebnisintervalle sollte nicht mit der Current-Präzision, sondern mit der Präzision der Eingangsintervalle genau übereinstimmen. Im folgenden Abschnitt werden diese Funktionen kurz zusammengestellt.

6.12.2 Davon abweichende Funktionen

```
MpfiClass Re (const MpfciClass& z);
```

Mit $z = x + i \cdot y$ wird das Realteil-Intervall $\mathrm{Re}(z) = x$ ohne Rundung in der Präzision von `z` zurückgegeben.

```
MpfiClass Im (const MpfciClass& z);
```

Mit $z = x + i \cdot y$ wird das Imaginärteil-Intervall $\mathrm{Im}(z) = y$ ohne Rundung in der Präzision von `z` zurückgegeben.

```
MpfcClass Inf (const MpfciClass& z, PrecisionType prec);
```

Mit $z = x + i \cdot y$ wird eine komplexe Zahl c in der Präzision `prec` zurückgegeben. $\mathtt{Re}(c)$ und $\mathtt{Im}(c)$ sind dabei die jeweils größten Maschinenzahlen, für die gilt: $\mathtt{Re}(c) \le \mathtt{Inf}(x)$ und $\mathtt{Im}(c) \le \mathtt{Inf}(y)$. Wird `prec` nicht angegeben, so wird c ganz entsprechend in der Current-Präzision berechnet. Ist speziell `prec` die Präzision von z, so gilt in den beiden oberen Ungleichungen das Gleichheitszeichen.

```
MpfcClass Sup (const MpfciClass& z, PrecisionType prec);
```

Mit $z = x + i \cdot y$ wird eine komplexe Zahl c in der Präzision `prec` geliefert. $\mathtt{Re}(c)$ und $\mathtt{Im}(c)$ sind dabei die jeweils kleinsten Maschinenzahlen, für die gilt: $\mathtt{Re}(c) \ge \mathtt{Sup}(x)$ und $\mathtt{Im}(c) \ge \mathtt{Sup}(y)$. Wird `prec` nicht angegeben, so wird c ganz entsprechend in der Current-Präzision berechnet. Ist speziell `prec` die Präzision von z, so gilt in den beiden oberen Ungleichungen das Gleichheitszeichen.

```
MpfcClass mid(const MpfciClass& z);
```

Mit $z = x + i \cdot y$ wird eine komplexe Zahl c in der gleichen Präzision von `z` geliefert. $\mathtt{Re}(c)$ und $\mathtt{Im}(c)$ sind dabei die jeweiligen Mittelpunkte von `x` und `y`. Beachten Sie, dass bei möglichen späteren Rundungen von c diese Mittelpunktseigenschaft verloren gehen kann.

```
MpfciClass Blow(const MpfciClass& op1, const MPFR::MpfrClass& op2);
```

Der Rückgabewert ist ein mit op2 aufgeblähtes Intervall op1 mit gleicher Präzision. `Blow(...)` ist genauso definiert wie die gleichnamige Funktion in C-XSC.

```
void times2pown (MpfciClass& z, long int k)
```

Obige Funktion liefert mit dem Eingabewert z eine optimale Einschließung von $z \cdot 2^k$ mit gleicher Präzision zurück, d.h. die mit 2^k multiplizierten Real- und Imaginärteil-Intervalle werden optimal eingeschlossen. Solange kein Über- oder Unterlauf entsteht, wird $z \cdot 2^k$ sogar **exakt**, d.h. rundungsfehlerfrei eingeschlossen.

```
void set_nan (MpfciClass& z);
```

Setzt z auf ([NaN, NaN], [NaN, NaN]), wobei die Präzision von z erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_inf (MpfciClass& z);
```

Setzt z auf ([−Inf, +Inf], [−Inf, +Inf]), wobei die Präzision von z erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
void set_zero (MpfciClass& z);
```

Setzt z auf ([0, 0], [0, 0]), wobei die Präzision von z erhalten bleibt und deshalb mit der Current-Precision nicht übereinstimmen muss.

```
MpfcClass conj (const MpfciClass& z);
```

Mit $z = x + i \cdot y$ ist der Rückgabewert von `conj(z)` gegeben durch $x - i \cdot y$, wobei die Präzision von x und y nicht geändert wird und daher mit der Current-Precision auch nicht übereinstimmen muss.

6.12.3 Elementarfunktionen

Tabelle 6.1: Funktionen mit `z` = $x + i \cdot y$ vom Typ `MpfciClass` und x, y vom Typ `MpfiClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	z^p, p : `MpfiClass`	`pow(z,p)`
$\mathrm{Re}(z) = x$	`Re(z)`	z^p, p : `MpfiClass`	`pow_all(z,p)`
$\mathrm{Im}(z) = y$	`Im(z)`	z^w, w : `MpfciClass`	`pow(z,w)`
$\lvert z \rvert$	`abs(z)`	$\sin(z)$	`sin(z)`
$\mathrm{Arg}(z)$	`Arg(z)`	$\cos(z)$	`cos(z)`
$\arg(z)$	`arg(z)`	$\tan(z)$	`tan(z)`
z^2	`sqr(z)`	$\cot(z)$	`cot(z)`
$\sqrt{z}$	`sqrt(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\arccos(z)$	`acos(z)`
$\sqrt[n]{z}$	`sqrt(z,n)`	$\arctan(z)$	`atan(z)`
$\sqrt[n]{z}$	`sqrt_all(z,n)`	$\mathrm{arccot}(z)$	`acot(z)`
$\log(z)$	`Ln(z)`	$\sinh(z)$	`sinh(z)`
$\log(z)$	`ln(z)`	$\cosh(z)$	`cosh(z)`
$\log_2(z)$	`log2(z)`	$\tanh(z)$	`tanh(z)`
$\log_{10}(z)$	`log10(z)`	$\coth(z)$	`coth(z)`
10^z	`exp10(z)`	$\mathrm{arsinh}(z)$	`asinh(z)`
e^z	`exp(z)`	$\mathrm{arcosh}(z)$	`acosh(z)`
2^z	`exp2(z)`	$\mathrm{artanh}(z)$	`atanh(z)`
z^n, $n \in \mathbb{Z}$	`power_fast(z,n)`	$\mathrm{arcoth}(z)$	`acoth(z)`
z^n, $n \in \mathbb{Z}$	`power(z,n)`		

Anmerkungen:

1. Bei den Funktionen `conj(z)`, `Re(z)` und `Im(z)` werden die Ergebnisse in der Präzision von `z` zurückgegeben, die also nicht mit der Current-Präzision übereinstimmen muss.

2. `abs(z)` kann mit einem zusätzlichen Präzisionsparameter `prec` aufgerufen werden. Dann wird das exakte, reelle Ergebnisintervall durch ein Intervall der Präzision `prec` garantiert eingeschlossen. Ohne `prec` wird das exakte, reelle Ergebnisintervall durch ein Intervall mit der Current-Präzision eingeschlossen.

3. `Arg(z)` kann mit einem zusätzlichen Präzisionsparameter `prec` aufgerufen werden. Dann wird das exakte, reelle Argumentintervall durch ein Intervall der Präzision `prec` garantiert eingeschlossen. Ohne `prec` wird das exakte, reelle Argumentintervall durch ein Intervall mit der Current-Präzision eingeschlossen. Enhält das komplexe Eingangsintervall `z` eine negative, reelle Zahl -auch auf dem Intervallrand!-, so erfolgt eine entsprechende Fehlermeldung. Nicht erlaubte Intervalle sind demnach
$\texttt{z} = [-2,-1] + i \cdot [-1,0]$ oder $\texttt{z} = [-1,-1] + i \cdot [0,1]$ oder $\texttt{z} = [-1,+1] + i \cdot [0,0]$.
$\texttt{z} = [-2,-1] + i \cdot [-2,-1]$ liefert: `Arg(z)` $= [-2.67795, -2.03444]$.

4. `arg(z)` kann mit einem zusätzlichen Präzisionsparameter `prec` aufgerufen werden. Dann wird das exakte, reelle Argumentintervall durch ein Intervall der Präzision `prec` garantiert eingeschlossen. Ohne `prec` wird das exakte, reelle Argumentintervall durch ein Intervall mit der Current-Präzision eingeschlossen. `arg(z)` liefert für jedes $\texttt{z} \in \mathbb{IC}$ eine Einschließung des exakten Argumentintervalls, dabei gilt: `arg(z)` $\subset [-\pi, 3\pi/2]$. Hier einige Beispiele:
$$\begin{aligned}
\texttt{arg}([-2,-1] + i \cdot [-2,-1]) &= [-2.67795, -2.03444],\\
\texttt{arg}([-2,+1] + i \cdot [-2,+2]) &= [-3.14160, +3.14160],\\
\texttt{arg}([-0,+0] + i \cdot [-0,+0]) &= [-0.00000, +0.00000],\\
\texttt{arg}([-2,-1] + i \cdot [+0,+1]) &= [+2.35619, +3.14160],\\
\texttt{arg}([-2,-1] + i \cdot [-1,-0]) &= [-3.14160, -2.35619],\\
\texttt{arg}([-2,-1] + i \cdot [-0,+0]) &= [+3.14159, +3.14160],\\
\texttt{arg}([-0,+0] + i \cdot [-1,+1]) &= [-1.57080, +1.57080],\\
\texttt{arg}([-1,+0] + i \cdot [-1,+1]) &= [+1.57079, +4.71239].
\end{aligned}$$

5. Beim Aufruf von `sqr(z)`, mit $\texttt{z} = X + i \cdot Y$, wird der Realteil von `sqr(z)` anstelle von X, Y mithilfe der reellen Intervalle $|X|, |Y|$ berechnet, weitere Einzelheiten findet man auf Seite 155. Mit $z = [-3,2] + i \cdot [-5,3]$ liefert daher $z \cdot z = [-31,24] + i \cdot [-20,30]$ eine erhebliche Überschätzung des Realteils von $\texttt{sqr}(z) = [-25,9] + i \cdot [-20,30]$.

6. `power_fast(z,n)` liefert für zu breite Intervalle z und für betragsmäßig zu große n-Werte erhebliche Überschätzungen von z^n. Für sehr schmale Intervalle werden auch bei größeren n-Werten brauchbare Einschließungen in recht kurzen Laufzeiten berechnet. Bei breiteren Intervallen z sollte man daher die Funktion `power(z,n)` benutzen, [43].
Mit $Z = [1,1] + i[1,1]$ erhält man für $\alpha := \{y \in \mathbb{C} \,|\, y = z^8,\ z \in Z\}$ die Einschließungen:
$$\begin{aligned}
\texttt{power_fast(Z,8)} &= ([1.599999e1, 1.600001e1], [-3.918870e-15, 1.029199e-14])\\
\texttt{power(Z,8)} &= ([1.599999e1, 1.600001e1], [-3.918870e-15, 1.029199e-14]).
\end{aligned}$$
Mit $Z = [1,1.125] + i[1,1.25]$ erhält man für $\beta := \{y \in \mathbb{C} \,|\, y = z^8,\ z \in Z\}$ die Einschließungen:
$$\begin{aligned}
\texttt{power_fast(Z,8)} &= ([1.012943e1, 6.397266e1], [-2.897500e1, 4.951985e1])\\
\texttt{power(Z,8)} &= ([1.599999e1, 5.839539e1], [-1.193387e1, 3.337647e1]).
\end{aligned}$$
Das letzte Beispiel zeigt deutlich die mit `power(Z,8)` berechnete bessere Einschließung von β.

7. Mit `Ln(z)` werden nur Funktionswerte des Hauptzweiges des natürlichen Logarithmus eingeschlossen. Die Funktion wird auch als *analytische* Funktion bezeichnet, da ein Rechteckintervall `z` den Verzweigungsschnitt nur von oben berühren darf. Im Gegensatz dazu darf bei der *nicht-analytische* Funktion `ln(z)` das Rechteckintervall `z` den Verzweigugsschnitt ganz im Innern enthalten oder diesen von oben oder unten berühren, vgl. die Abbildungen B.11, B.12 auf den Seiten 157 und 158.

8. Mit `pow(Z,P)` wird eine Rechteck-Einschließung der komplexen Menge

$$\{y \in \mathbb{C} \,|\, y = e^{p \cdot \ln(z)},\ z \in \texttt{Z:MpfciClass},\ p \in \texttt{P:MpfiClass}\}$$

berechnet, wobei nur die Funktionswerte des Hauptzweiges eingeschlossen werden, d.h. das Rechteckintervall `Z` darf den Verzweigungsschnitt nur von oben berühren und nicht in seinem Innern enthalten. $\ln(z)$ bedeutet den Hauptwert des komplexen Logarithmus.

9. Mit `pow_all(Z,P)` und $z = |z| \cdot e^{i\varphi},\ -\pi/2 < \varphi < +3\pi/2$ wird eine Einschließung der komplexen Menge

$$\{y \in \mathbb{C} \,|\, y = e^{p \cdot \ln(z)} = y = e^{p \cdot \ln|z|} \cdot e^{ip(\varphi + 2\pi k)},\ z \in \texttt{Z:MpfciClass},\ p \in \texttt{P:MpfiClass},\ k \in \mathbb{Z}\}$$

berechnet, wobei aber jetzt `Z` den Verzweigungsschnitt auch in seinem Innern enthalten darf. Die Einschließung besteht entweder aus einem einzelnen Rechteckintervall oder aus vier Rechtecken, die einen Kreisring optimal einschließen. Weitere Einzelheiten finde man ab Seite 204.

10. Mit `pow(Z,W)` wird eine Rechteck-Einschließung der komplexen Menge

$$\{y \in \mathbb{C} \,|\, y = e^{w \cdot \ln(z)},\ z \in \texttt{Z:MpfciClass},\ w \in \texttt{W:MpfciClass}\}$$

berechnet, wobei unter $\ln(z)$ der Hauptwert des komplexen Logarithmus zu verstehen ist. Das Rechteckintervall `Z` darf den Verzweigungsschnitt nur von oben berühren und nicht in seinem Innern enthalten.

7 Anwendungen

7.1 Nullstellen komplexer Ausdrücke

In der Dissertation von W. Krämer wird ein Algorithmus zur Einschließung von Nullstellen komplexwertiger Ausdrücke angegeben, [28],[11]. Benötigt wird eine möglichst gute Nullstellenapproximation, die mithilfe eines vereinfachten Newton-Verfahrens weiter verbessert wird, um danach die eindeutig bestimmte Nullstelle garantiert einzuschließen. *Eindeutig* bedeutet hier, dass das einschließende Intervall **genau eine** Nullstelle enthält. Mit dem folgenden Programm können mit geeigneten Näherungen die drei einfachen Nullstellen $z_1 = 2 + i,\ z_2 = 3 - i,\ z_3 = 4i$ der Funktion

$$f(z) = \arctan\big((z-a)\cdot\ln(z^2-5\cdot z+8+i)\big),$$
$$f'(z) = \frac{\ln(z^2-5\cdot z+8+i) + \dfrac{(z-a)\cdot(2\cdot z-5)}{z^2-5\cdot z+8+i}}{1+(z-a)^2\cdot(\ln(z^2-5\cdot z+8+i))^2}$$

garantiert eingeschlossen werden, wobei $a = 4i$ zu setzten ist. Im Programm müssen die beiden Funktionen $f(z)$ und $f'(z)$ explizit definiert werden.

```
1  // MPFR-11.cpp
2  // Inclusion of zeros of a complex expression
3
4  #include "mpfciclass.hpp"
5
6  using namespace MPFR;
7  using namespace MPFI;
8  using namespace cxsc;
9  using namespace std;
10
11 // The zeros of function F() are to be enclosed.
12 MpfcClass a(0,4);  // a = 0 + 4i, parameter of F().
13
14 MpfciClass F(const MpfciClass& Z) // Definition of f(z)
15 {
16     MpfciClass res;
17     res = atan( (Z-a)*Ln(sqr(Z) - 5*Z + MpfcClass(8,1)) );
18     return res;
19 };
20
21 //  First derivative of F():
22 MpfciClass DERIV(const MpfciClass& Z) // Definition of f'(z)
23 {
24     MpfciClass U,res;
25     U = sqr(Z) - 5*Z + MpfcClass(8,1);
26     res = (Ln(U)+(Z-a)*(2*Z-5)/U)/(1+sqr(Z-a)*sqr(Ln(U)));
27     return res;
28 };
29
30 int main(void)
31 {
32   MpfciClass Y,Z0,Zb,Z,D,CH,Zi,C;
33   MpfcClass zeta0;
34   int kmax(12),incl,k; // kmax: Maximum number of iteration steps
```

```
35  MpfrClass eps = MpfrClass(0.125);
36  int prec;
37
38  cout << "Current-Precision(Bits) = ? "; cin >> prec;
39  MpfrClass::SetCurrPrecision(prec);
40  cout << "Current-Precision = " << Round(prec/3.321928095)
41       << " Decimal digits." << endl;
42  prec = prec/3.321928095; // prec: Precision in decimal digits;
43  cout.precision(prec); // Output with prec decimal digits;
44
45  while (1)
46  {
47    cout << "Complex zero approximation (x,y) = ? ";
48    cin >> zeta0;
49    Zi = MpfciClass(zeta0); k=0;
50    do // Improve zero approximation
51    {
52      k++;  Z0 = Zi;
53      Zi = MpfciClass( mid(Zi - F(Zi)/DERIV(Zi)) );
54      Z  = MpfciClass( MpfiClass(Re(Sup(Z0)),Re(Sup(Zi))),
55           MpfiClass(Im(Sup(Z0)),Im(Sup(Zi))) );
56    } while (common_decimals(Re(Z))<prec/3 && common_decimals(Im(Z))<prec/3
57                                           && k<30 );
58
59    cout << "Improved zero approximation: " << Sup(Zi) << endl;
60
61    C = MpfciClass(1/Sup(DERIV(Zi))); // C: point interval
62    Y = -C*F(Zi);
63    Z = Y; // Y,C: initial values of iteration
64    k=0;
65
66    do
67    {
68      k++;
69      Zb = Blow(Z,eps); // epsilon inflation of Z
70      CH = Zi | Zi+Zb;  // CH: complex hull
71      D = 1 - C*DERIV(CH);
72      Z = Y + D*Zb;
73      incl = in(Z,Zb);  // Inclusion test
74    }  // kmax: Maximum number of iteration steps
75    while ( (k<kmax) && (incl==0) );
76
77    if (incl)
78    { // incl=1 <--> Inclusion verified
79      Z0 = Zi + Z; // Inclusion of the unique zero
80      cout << "Inclusion: Z0 = " << Z0 << endl;
81    }
82    else cout << "No inclusion found" << endl;
83  }
84
85  return 0;
86 }
```

Anmerkungen:

- Mit den Näherungen $\zeta_1 = 1.9 + 0.9i$, $\zeta_2 = 2.9 - 0.9i$, $\zeta_3 = 0.1 + 3.9i$ erhält man z.B. mit der Präzision `prec` = 3000 Bits $\approx 3000/3.321928095 \approx 903$ Dezimalstellen nahezu optimale Einschließungen der drei Nullstellen $z_1 = 2 + i$, $z_2 = 3 - i$, $z_3 = 4i$, wobei der Realteil 0 von z_3 natürlich nur vergleichsweise grob eingeschlossen werden kann.
- In Zeile 53 wird das vereinfachte Newton-Verfahren so lange angewandt, bis zwei aufeinanderfolgende Nullstellen-Approximationen in mindestens einem Drittel ihrer dezimalen Präzisionsstellen übereinstimmen. Danach erfolgt ab Zeile 66 der Einschließungstest.

7.2 Erste Nullstelle von $J_0(x)$

Das nachfolgende Programm liefert mit dem Intervall-Newton-Verfahren eine Einschließung der ersten Nullstelle $x_1 = 2.4048\ldots$ der Besselfunktion 1. Art $J_0(x)$, [17], [27]. In einer Umgebung von x_1 benötigt man dazu Intervallfunktionen von $J_0(x)$ und seiner ersten Ableitung $J_0'(x) = -J_1(x)$. Da beide Funktionen als Intervallfunktionen nicht zur Verfügung stehen, im Umgebungsintervall $\alpha = [2,3] \ni x_1$ jedoch monoton fallend bzw. wachsend sind, kann man $J_0(x)$ und $J_0'(x) - J_1(x)$ in `MpfiClass f(const MpfiClass& x)` bzw. in `MpfiClass DERIV(const MpfiClass& x)` sehr einfach implementieren. Die Funktion `bool criter(const MpfiClass& x)` untersucht in `x` den Vorzeichenwechsel und die Monotonie von $J_0(x)$, wodurch die Eindeutigkeit der Nullstelle in `x` verifiziert ist. Zusätzlich wird untersucht, ob das Startintervall `x` die Bedingung $\mathtt{x} \subseteq \alpha$ erfüllt.

```
// MPFR-13.cpp
// Newton-Verfahren zur Einschliessung der 1. Nst. x1 von J0(x);
// J0'(x) = -J1(x);  x1 = 2.4048....;
// In [2,3] sind J0 und -J1(x) monoton fallend bzw. monoton wachsend,
// so dass deren Intervallfunktionen dort einfach zu implementieren sind.
// Als Start-Intervall sollte [2,3] gewaehlt werden.
#include "mpficlass.hpp"

using namespace MPFR;
using namespace MPFI;
using namespace cxsc;
using namespace std;

MpfiClass f(const MpfiClass& x) // J0(x)
{
  MpfrClass left ( J0(Sup(x),RoundDown) );
  MpfrClass right( J0(Inf(x),RoundUp)   );
  return MpfiClass(left, right);
}

MpfiClass DERIV(const MpfiClass& x) // -J1(x)
{
  MpfrClass left ( J1(Sup(x),RoundDown) );
  MpfrClass right( J1(Inf(x),RoundUp)   );
  return -MpfiClass(left, right);
}

MpfiClass Start_Interval(void)
{
  MpfrClass a, b;
  cout << endl << "Left boundary point =  ? "; cin >> a;
  cout << "Right boundary point = ? "; cin >> b; cout << endl;
  return MpfiClass(a,b);
}

bool criter(const MpfiClass& x) // Computing: J0(Inf(x))*f(Sup(x)) < 0
                                // and not 0 in J0'([x]) and x <= [2,3]?
{
  MpfrClass a(2), b(3); // Start-interval <= [2,3] must be verified!
  return ( Sup( f(MpfiClass(Inf(x))) * f(MpfiClass(Sup(x))) ) < 0 )
         && !( 0 <= DERIV(x) ) && x <= MpfiClass(a,b);
}

int main(void)
{
  PrecisionType prec = 6000;
  MpfrClass::SetCurrPrecision(prec);
  MpfiClass x, xOld, xMid, fxMid, fx, dfx;
  x = Start_Interval();
  cout << "Starting interval is: " << x << endl;
```

```
51
52    if (criter(x))
53    {  // There is exactly one zero of f in the interval x
54      do
55      {
56        xOld = x;
57        cout << "Actual enclosure is " << x
58             << ",  Absolute diameter: " << diam(x) << endl;
59        xMid = MpfiClass(mid(x));
60        fxMid = f(xMid);
61        dfx = DERIV(x);
62        x = ( xMid - fxMid / dfx ) & x;
63      } while (x != xOld);
64      cout.precision(prec/3.321928095);
65      cout << "Final enclosure of the zero: " << x << endl
66           << "Absulute diameter = " << diam(x) << endl;
67      cout << "Correct decimal digits = " << common_decimals(x) << endl;
68      cout << "Enclosure of J0(x): " << f(x) << endl;
69      cout << "J0(Inf(x)) = " << f(MpfiClass(Inf(x))) << endl;
70    }
71    else
72      cout << "Criterion not satisfied!" << endl;
73
74    return 0;
75  }
```

Das Programm liefert nach Zeile 46 mit `prec` = 6000 Bits ≈ 1806 Dezimalstellen die verkürzte Ausgabe

```
Left boundary point =  ? 2.3
Right boundary point = ? 2.5

Starting interval is: [2.29999,2.50000]
Actual enclosure is [2.29999,2.50000],  Absolute diameter: 2.00000e-1
Actual enclosure is [2.40464,2.40505],  Absolute diameter: 3.99731e-4
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 3.20189e-9
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 1.03227e-19
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 5.36453e-41
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 7.24403e-84
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 6.60462e-170
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 2.74507e-342
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 2.37102e-687
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 8.84440e-1378
Actual enclosure is [2.40482,2.40483],  Absolute diameter: 2.64293e-1806
Final enclosure of the zero:
[2.40482555769577276862 16...1643831,2.40482555769577276862 16...1643832]
Absulute diameter = 2.6429...4131431e-1806
Correct decimal digits = 1805

Enclosure of J0(x):
[-8.081224285204...20008648601e-1807, 5.6394916205927...970488619e-1807]

J0(Inf(x)) = [5.63949162059...488618e-1807,5.63949162059...488619e-1807]
```

Anmerkungen:

- Die Nullstelle $x_1 = 2.40482\ldots$ wird mit 1805 korrekten Dezimalstellen eingeschlossen. Anstelle von `prec` = 6000 Bits kann in Zeile 46 auch eine sehr viel größere Präzision gewählt werden.
- Prinzipiell können ganz analog auch die Nullstellen $x_3, x_5, x_7, \ldots$ eingeschlossen werden, da $\mathrm{J}_0(x)$ und $\mathrm{J}_0'(x)$ in deren Umgebung das gleiche Monotonieverhalten aufweisen.

7.3 Einschließung reeller arithmetischer Audrücke

Das folgende Programm zeigt wesentliche Punkte, die zu beachten sind, wenn ein arithmetischer Ausdruck an einer speziellen Stelle x_0 einzuschließen ist. Als Beispiel betrachten wir die Funktion

$$f(x) := \tan(x) - \sin(x) \equiv 2 \cdot \tan(x) \cdot \sin^2(x/2), \qquad \text{mit: } x_0 = 2^{-16000}.$$

```
1  // MPFR-12.cpp
2  // Inclusion of f(x) := tan(x) - sin(x); x = 2^(-16000);
3  //                  f(x)  = 2*tan(x)*sqr( sin(x/2) );
4  #include "mpficlass.hpp"
5
6  using namespace MPFR;
7  using namespace MPFI;
8  using namespace cxsc;
9  using namespace std;
10
11 int main(void)
12 {
13   PrecisionType prec = 600, prec_old = prec;
14   MpfrClass::SetCurrPrecision(prec);
15   MpfiClass x(exp2(MpfiClass(-16000))), y; // x = 2^(-16000)
16   // Evaluating the difference expression
17   y = tan(x) - sin(x);
18   cout.precision(prec/3.321928095);
19   cout << "f(x) included by " << y << endl;
20   // Evaluation with fifty-fivefold Precision
21   MpfrClass::SetCurrPrecision(55*prec);
22   y = tan(x) - sin(x);
23   y.RoundPrecision(prec_old);
24   cout << "f(x) included by " << y << endl;
25   // Evaluating the simplified expression
26   MpfrClass::SetCurrPrecision(prec_old);
27   y = 2*tan(x)*sqr( sin(x/2) ); // product expression
28   cout << "f(x) included by " << y << endl;
29
30   return 0;
31 }
```

Das Programm liefert die verkürzte Ausgabe

```
f(x) included by [0,2.3943808622739626 83...288266110563e-4997]
f(x) included by [1.81626048...4826128e-14450,1.81626048...4826129e-14450]
f(x) included by [1.81626048...4826128e-14450,1.81626048...4826129e-14450]
```

Anmerkungen:

- Die erste Einschließung ist wegen auftreteder Auslöschung sehr grob, da die Differenz in der Nähe ihrer Nullstelle 0 auszuwerten ist.
- Eine optimale Einschließung dieser Differenz erhält man erst bei 55-facher Präzision.
- Die Einschließung des Produkts ist schon bei einfacher Präzision nahezu optimal.

> Ist eine Summe oder Differenz in der Nähe einer Nullstelle auszuwerten, so muss zur Kompensation möglicher Auslöschungen eine mehrfache Präzision gewählt werden. Summen oder Differenzen sind daher möglichst in Produkte umzuformen!

A Neue (Hilfs-)Funktionen vom Typ `MpfrClass`

Mit Hilfe der bereits in den beiden MPFR- und MPFI-Bibliotheken implementierten Funktionen aus Tabelle 3.1 und 3.2 sollen noch weitere (Hilfs-)Funktionen (insbesondere hilfreich im Zusammenhang mit der Realisierung weiterer komplexer Intervallfunktionen) für den Datentyp `MpfrClass` realisiert werden, wobei darauf zu achten ist, dass mit einem zusätzlichen Rundungsparameter `rnd` der Funktionswert, abweichend vom Current-Rundungsmodus, gerundet werden kann. Damit stellt sich die Aufgabe, einen gegebenen Funktionsterm so zu programmieren, dass dieser entweder zur nächsten Rasterzahl oder garantiert auf- bzw. abgerundet werden kann. Ist eine solche Funktion dann z.B. in einem Maschinenintervall $[a, b]$ monoton wachsend, so ist eine Einschließung aller Funktionswerte $f(x)$, $x \in [a, b]$ durch das Intervall $[f_d(a), f_u(b)]$ gegeben, wobei $f_d(a)$ und $f_u(b)$ die ab- bzw. aufgerundeten Funktionswerte bedeuten.

A.1 Grundregeln für garantierte Rundungen

Soll ein neu implementierter Ausdruck A z.B. zur nächsten Rasterzahl gerundet werden, so ist der Rundungsmodus `rnd` auf `RoundNearest` zu setzen und A ist auszuwerten. Ist dann $\widetilde{A}$ das Ergebnis dieser Auswertung, so wird $\widetilde{A}$ i.a. nicht der Vorgänger oder Nachfolger von A sein. Entsprechende Aussagen gelten auch für die auf- oder abgerundeten Maschinenergebnisse. Wir fassen zusammen:

> Wird eine neu implementierte Funktion mit `rnd = RoundNearest` mit der Maschinenzahl x_0 ausgewertet, so muss der Maschinenwert $\widetilde{f}(x_0)$ nicht die zum exakten Funktionswert $f(x_0)$ nächstgelegene Rasterzahl sein. Ist $f_u(x_0)$ der mit `rnd = RoundUp` aufgerundete Funktionswert, so wird i.a. auch $f_u(x_0)$ nicht der Nachfolger von $f(x_0)$ sein; $f(x_0) \le f_u(x_0)$ ist aber stets garantiert. Für den abgerundeten Funktionswert $f_d(x_0)$ gelten entsprechende Aussagen.

Bezeichnungen:

A, B sind exakte Ausdrücke, die im Zahlenformat nicht exakt darstellbar sein müssen.

- A_d bezeichnet den abgerundeten Maschinenwert: $A_d \le A$.
- A_u bezeichnet den aufgerundeten Maschinenwert: $A_u \ge A$.
- $\oplus_d$ bezeichnet die abrundende Maschinen-Addition.
- $\oplus_u$ bezeichnet die aufrundende Maschinen-Addition.
- $\ominus_d$ bezeichnet die abrundende Maschinen-Subtraktion.
- $\ominus_u$ bezeichnet die aufrundende Maschinen-Subtraktion.
- $\odot_d$ bezeichnet die abrundende Maschinen-Multiplikation.
- $\odot_u$ bezeichnet die aufrundende Maschinen-Multiplikation.
- $\oslash_d$ bezeichnet die abrundende Maschinen-Division.

- $\oslash_u$ bezeichnet die aufrundende Maschinen-Division.

Auch jetzt wird nicht verlangt, dass $A_d = \texttt{pred}(A)$ oder $A_u = \texttt{succ}(A)$ erfüllt sind, aber die Beziehungen $A_d \leq A$ bzw. $A_u \geq A$ werden stets garantiert.

A.1.1 Unitäre Operatoren

Unabhängig davon, ob A positiv, negativ oder gleich Null ist, gilt

$$+(A_d) \leq A; \tag{A.1}$$
$$+(A_u) \geq A; \tag{A.2}$$
$$-(A_d) \geq -A; \tag{A.3}$$
$$-(A_u) \leq -A; \tag{A.4}$$

A.1.2 Addition

Unabhängig davon, ob A oder B positiv, negativ oder gleich Null sind, gilt

$$A_d \oplus_d B_d \leq A + B; \tag{A.5}$$
$$A_u \oplus_u B_u \geq A + B; \tag{A.6}$$

A.1.3 Subtraktion

Unabhängig davon, ob A oder B positiv, negativ oder gleich Null sind, gilt

$$A_d \ominus_d B_u \leq A - B; \tag{A.7}$$
$$A_u \ominus_u B_d \geq A - B; \tag{A.8}$$

Die Beweise für die obigen Sätze sind trivial und bleiben dem Leser überlassen.

A.1.4 Multiplikation

Um bei Multiplikation und Division die Bedingungen an beide Operanden für ein Auf- bzw. Abrunden möglichst übersichtlich formulieren zu können, geben wir noch zwei einschränkende Eigenschaften[1] von A an, die aber nur in speziellen Fällen erfüllt sein müssen:

Für Ausdrücke A, die auf dem Rechner gezielt auf- bzw. abzurunden sind, muss zusätzlich erfüllt sein:

$$A_u > 0 \implies A_u \geq A \geq 0; \tag{A.9}$$
$$A_d < 0 \implies A_d \leq A \leq 0; \tag{A.10}$$

Wir formulieren jetzt noch Bedingungen für einen arithmetischen Ausdruck A, mit denen die Eigenschaften (A.9) bzw. (A.10) abgesichert werden:

Wird ein Ausdruck A nur durch eine einzige Rechenoperation erzeugt, so gilt (A.9) bzw. (A.10). Diese Eigenschaften sind auch dann garantiert, wenn A nur durch einen einzigen Funktionsaufruf mit Funktionen aus Tabelle 3.1 und 3.2 realisiert wird.

[1] Genauer gesagt sind es die Eigenschaften des Algorithmus, mit dem der arithmetische Ausdruck A ausgewertet wird.

Das folgende Beispiel soll zeigen, dass z.B die Eigenschaft (A.9) nicht immer erfüllt sein muss. Mit den Rasterzahlen a, b, c, d definieren wir dazu den exakten Ausdruck A durch:

$$A := a \cdot b + c \cdot d - c \cdot d = a \cdot b,$$

wobei $-\texttt{minfloat()} < a{\cdot}b = A < 0$ gelten soll. Die Rasterzahlen $c, d > 0$ seien so gewählt, dass $c{\cdot}d \approx 1$ im vorliegenden Zahlenraster nicht exakt darstellbar ist, so dass bei der Produktberechnung eine Rundung notwendig wird. Der aufzurundende Ausdruck A ist wie folgt auszuwerten:

$$\begin{aligned} A_u &:= (a \odot_u b) \oplus_u (c \odot_u d) \ominus_u (c \odot_d d) \\ &= (c \odot_u d) \ominus_u (c \odot_d d) > 0; \end{aligned}$$

Zunächst gilt wegen der geforderten Aufrundung $(a \odot_u b) = 0$, und wegen der notwendigen Rundung bei den Produktberechnungen gilt: $0 < (c \odot_u d) - (c \odot_d d) \leq (c \odot_u d) \ominus_u (c \odot_d d) = A_u$, so dass jetzt im Gegensatz zu (A.9) aus $A_u > 0$ nicht $A \geq 0$ gefolgert werden kann.

Wir kommen jetzt zur Formulierung der Operandenbedingungen, mit denen bei der Multiplikation gezielt auf- bzw. abgerundet werden kann. Bei der gerundeten Multiplikation muss man dabei unterscheiden, ob die Operanden positiv, negativ oder gleich Null sind. Wir betrachten zunächst das Abrunden, dabei bedeutet $*$, dass nur einer der beiden gerundeten Operanden verschwinden darf:

$$\begin{aligned}
&\text{(A.11)} && A_d \geq 0; \quad B_d \geq 0 && \Longrightarrow && A_d \odot_d B_d \leq A \cdot B; \\
&\text{(A.12)} && * \; A_u \geq 0; \quad B_d \leq 0 && \overset{(A.9)(A.10)}{\Longrightarrow} && A_u \odot_d B_d \leq A \cdot B; \\
&\text{(A.13)} && * \; A_d \leq 0; \quad B_u \geq 0 && \overset{(A.9)(A.10)}{\Longrightarrow} && A_d \odot_d B_u \leq A \cdot B; \\
&\text{(A.14)} && A_u \leq 0; \quad B_u \leq 0 && \Longrightarrow && A_u \odot_d B_u \leq A \cdot B;
\end{aligned}$$

Bei der Multiplikation betrachten wir jetzt das Aufrunden, und nachfolgend bedeutet $*$, dass nur einer der beiden gerundeten Operanden verschwinden darf:

$$\begin{aligned}
&\text{(A.15)} && * \; A_u \geq 0; \quad B_u \geq 0 && \overset{(A.9)}{\Longrightarrow} && A_u \odot_u B_u \geq A \cdot B; \\
&\text{(A.16)} && A_d \geq 0; \quad B_u \leq 0 && \Longrightarrow && A_d \odot_u B_u \geq A \cdot B; \\
&\text{(A.17)} && A_u \leq 0; \quad B_d \geq 0 && \Longrightarrow && A_u \odot_u B_d \geq A \cdot B; \\
&\text{(A.18)} && * \; A_d \leq 0; \quad B_d \leq 0 && \overset{(A.10)}{\Longrightarrow} && A_d \odot_u B_d \geq A \cdot B;
\end{aligned}$$

A.1.5 Division

Wir betrachten zunächst das Abrunden, wobei $B \neq 0$ vorausgesetzt wird:

$$\begin{aligned}
&\text{(A.19)} && A_d \geq 0; \quad B_u > 0 && \overset{(A.9)}{\Longrightarrow} && A_d \oslash_d B_u \leq A/B; \\
&\text{(A.20)} && A_d \leq 0; \quad B_d > 0 && \overset{(A.10)}{\Longrightarrow} && A_d \oslash_d B_d \leq A/B; \\
&\text{(A.21)} && A_u \geq 0; \quad B_u < 0 && \overset{(A.9)}{\Longrightarrow} && A_u \oslash_d B_u \leq A/B; \\
&\text{(A.22)} && A_u \leq 0; \quad B_d < 0 && \overset{(A.10)}{\Longrightarrow} && A_u \oslash_d B_d \leq A/B;
\end{aligned}$$

Bei der Division betrachten wir jetzt das Aufrunden, wobei wieder $B \neq 0$ vorausgesetzt wird:

$$\begin{aligned}
&\text{(A.23)} && A_u \geq 0; \quad B_d > 0 && \overset{(A.9)}{\Longrightarrow} && A_u \oslash_u B_d \geq A/B; \\
&\text{(A.24)} && A_u \leq 0; \quad B_u > 0 && \overset{(A.9)}{\Longrightarrow} && A_u \oslash_u B_u \geq A/B; \\
&\text{(A.25)} && A_d \geq 0; \quad B_d < 0 && \overset{(A.10)}{\Longrightarrow} && A_d \oslash_u B_d \geq A/B; \\
&\text{(A.26)} && A_d \leq 0; \quad B_u < 0 && \overset{(A.10)}{\Longrightarrow} && A_d \oslash_u B_u \geq A/B;
\end{aligned}$$

Anmerkungen:

1. Zum Verständnis wird zunächst (A.12) bewiesen. Dabei zeigt sich, dass beide Bedingungen (A.9) und (A.10) von Seite 122 auch tatsächlich benötigt werden. Zu zeigen ist also:

   (A.27) $$A_u \geq 0, \quad B_d \leq 0 \quad \Longrightarrow \quad A_u \odot_d B_d \leq A \cdot B;$$

   Der Fall $A_u = B_d = 0$ ist auszuschließen, da sonst folgt: $A \leq 0$ und $B \geq 0$, d.h. $A \cdot B \leq 0$, und dies ist wegen $A_u \odot_d B_d = 0$ ein Widerspruch zur Behauptung.

   Im Fall $A_u = 0$ und $B_d < 0$ folgt zunächst $A_u \odot_d B_d = 0$ und $A \leq 0$. Um damit $A \cdot B \geq 0$ zu garantieren, benötigt man $B \leq 0$, und dies folgt nur, wenn für die gerundete Größe B_d die Forderung (A.10) erfüllt ist.

   Im Fall $B_d = 0$ und $A_u > 0$ folgt zunächst $A_u \odot_d B_d = 0$ und $B \geq 0$. Um damit $A \cdot B \geq 0$ zu garantieren, benötigt man $A \geq 0$, und dies folgt nur, wenn für die gerundete Größe A_u die Forderung (A.9) erfüllt ist.

   Jetzt bleibt noch: $A_u > 0$ und $B_d < 0$. Nach Definition von $\odot_d$ gilt ganz allgemein: $A_u \odot_d B_d \leq A_u \cdot B_d$. Weiter ergibt sich:

   $$A_u \geq A \overset{B_d<0}{\Longrightarrow} A_u \cdot B_d \leq A \cdot B_d, \quad \text{und} \quad B_d \leq B \overset{A\geq 0}{\Longrightarrow} B_d \cdot A \leq A \cdot B \blacksquare$$

   Zum Beweis benötigen wir damit $A \geq 0$, und diese Bedingung wird nur erfüllt, wenn für die gerundete Größe $A_u > 0$ die Forderung (A.9) auch tatsächlich erfüllt ist. Man kann den Beweis auch etwas abändern, muss dann aber auf (A.10) zurückgreifen. Die restlichen Beweise bez. der gerundeten Multiplikation können ganz analog durchgeführt werden.

2. Als Beispiel für die Division wird jetzt noch (A.24) bewiesen. Dabei zeigt sich, dass eine der beiden Bedingungen (A.9) und (A.10) von Seite 122 auch tatsächlich benötigt wird. Zu zeigen ist:

   (A.28) $$A_u \leq 0; \quad B_u > 0 \implies A_u \oslash_u B_u \geq A/B;$$

   Im Fall $A_u = 0$ und $B_u > 0$ folgt zunächst $A_u \oslash_u B_u = 0$ und $A \leq 0$. Um $A/B \leq 0$ zu garantieren, benötigt man $B > 0$, was wegen $B_u > 0$ mithilfe von (A.9) gesichert ist.

   Wir betrachten jetzt den Fall $A_u < 0,\ B_u > 0$. Zunächst gilt: $A_u \oslash_u B_u \geq A_u/B_u$;
   Wegen $B_u > 0$ gilt nach (A.9): $B_u \geq B > 0 \implies 1/B_u \leq 1/B \overset{A_u<0}{\Longrightarrow} A_u/B_u \geq A_u/B$;
   Es gilt nach Voraussetzung: $0 > A_u \geq A \overset{B>0}{\Longrightarrow} A_u/B \geq A/B \blacksquare$
   Die restlichen Beweise bez. der gerundeten Division können ganz analog durchgeführt werden.

3. Es sei noch einmal darauf hingewiesen, dass im Gegensatz zur Multiplikation und Division bei der gerichteten Addition und Subtraktion nicht untersucht werden muss, ob die Operanden positiv, negativ oder gleich Null sind.

4. Ein erstes einfaches Anwendungsbeispiel für gerichtete Rundungen findet man für die neu installierte Funktion $f(x, y) = x^2 - y^2$ auf Seite 126.

Nachdem wir für die gerichteten Rundungen bei der Multiplikation und Division die entsprechenden Bedingungen in (A.11) bis (A.18) bzw. in (A.19) bis (A.26) beschrieben haben, stellt sich jetzt noch die Frage, wie z.B. in (A.12) die Bedingungen $A_u \geq 0$ und $B_d \leq 0$ garantiert werden können. Dazu formulieren wir zunächst:

Ist $f(x)$ eine Funktion der MPFR-Bibliothek und bedeutet $f_a(x)$ den von der Null weggerundeten Maschinenwert, so gilt:

(A.29) $f_a(x) = 0 \implies f(x) = 0;$

(A.30) $f_a(x) > 0 \implies f(x) > 0;$

(A.31) $f_a(x) < 0 \implies f(x) < 0;$

Mithilfe von $f_a(x)$ erhält man daher gesicherte Aussagen über den exakten Funktionswert $f(x)$, und mit den folgenden Sätzen erhält man schließlich Aussagen bez. der auf- bzw. abgerundeten Funktonswerte $f_u(x)$, $f_d(x)$

Ist $f(x)$ eine Funktion der MPFR-Bibliothek und bedeuten $f_u(x)$ und $f_d(x)$ die auf- bzw. abgerundeten Funktionswerte, so gilt:

(A.32) $f(x) \geq 0 \implies f_u(x),\ f_d(x) \geq 0;$

(A.33) $f(x) \leq 0 \implies f_u(x),\ f_d(x) \leq 0;$

Wir fassen zusammen:

Ist $f(x)$ eine Funktion der MPFR-Bibliothek und bedeuten $f_u(x)$ und $f_d(x)$ die auf- bzw. abgerundeten Funktionswerte und ist $f_a(x)$ der von der Null weggerundete Funktionswert, so gilt:

(A.34) $f_a(x) \geq 0 \implies f_u(x),\ f_d(x) \geq 0;$

(A.35) $f_a(x) \leq 0 \implies f_u(x),\ f_d(x) \leq 0;$

Damit erhalten wir mithilfe von $f_a(x)$ die gewünschten Aussagen bezüglich der auf- bzw. abgerundeten Funktionswerte $f_u(x)$, $f_d(x)$. Wir formulieren noch zusätzlich:

Ist $f(x)$ eine Funktion der MPFR-Bibliothek und bedeuten $f_u(x)$ und $f_d(x)$ die auf- bzw. abgerundeten Funktionswerte, so gilt:

(A.36) $f_u(x) > 0 \iff f(x) > 0,$ d.h. (A.9) ist erfüllt.

(A.37) $f_u(x) = 0 \implies f(x) \leq 0,$

(A.38) $f_u(x) < 0 \implies f(x) < 0,$

(A.39) $f_d(x) > 0 \implies f(x) > 0,$

(A.40) $f_d(x) = 0 \implies f(x) \geq 0,$

(A.41) $f_d(x) < 0 \iff f(x) < 0,$ d.h. (A.10) ist erfüllt.

A.2 $x^2 - y^2,\ x^2 + y^2$

Um einen vorzeitigen Überlauf zu vermeiden, benutzen wir $x^2 - y^2 \equiv (|x| - |y|)(|x| + |y|)$. Bei hinreichend großen Werten von $|x|$ und $|y|$ kann im Fall $|x| \approx |y|$ auch jetzt noch die Summe $(|x| + |y|)$ einen vorzeitigen Überlauf erzeugen, der jedoch mit der Skalierung $|x| \cdot 2^{-2} + |y| \cdot 2^{-2}$ vermieden werden kann.

Es soll jetzt $f(x, y) = x^2 - y^2$ abgerundet werden, wobei die Differenz $A := |x| - |y| < 0$ als negativ vorausgesetzt wird. Die Summe $B := |x| \cdot 2^{-2} + |y| \cdot 2^{-2} > 0$ ist positiv und wegen der Skalierung nur wenig kleiner als `MaxFloat()`. Nach (A.13) ist das Abrunden garantiert durch

$$(\text{A.42}) \qquad * \quad A_d \le 0; \quad B_u \ge 0 \overset{(A.9)(A.10)}{\Longrightarrow} A_d \odot_d B_u \le A \cdot B,$$

wobei $*$ bedeutet, dass $A_d = B_u = 0$ nicht eintreten darf. Um das Abrunden von $f(x, y)$ zu gewährleisten, muss also $A_d \le 0$ und $B_u \ge 0$ nachgewiesen werden.

Ganz allgemein gilt $A_d \le A$, und wegen der Voraussetzung $A < 0$ ist die Bedingung $A_d \le 0$ schon erfüllt, wobei A_d selbst mithilfe der MPFR-Funktion `mpfr_sub(..,..,..,RoundDown)` berechnet wird.

Da $B > 0$ aufzurunden ist, müssen nach (A.6) die beiden Summanden $|x| \cdot 2^{-2}$ und $|y| \cdot 2^{-2}$ selbst aufgerundet werden durch:

$$\begin{aligned} \texttt{times2pown}(|x|, -2, \texttt{RoundUp}) \quad &\longrightarrow \quad |x|_u \ge |x| \cdot 2^{-2} > 0 \\ \texttt{times2pown}(|y|, -2, \texttt{RoundUp}) \quad &\longrightarrow \quad |y|_u \ge |y| \cdot 2^{-2} \ge 0; \end{aligned}$$

Da $B > 0$ ist, gilt: $B_u := |x|_u \oplus_u |y|_u \ge B > 0$, d.h. $B_u \ge B > 0$, so dass damit auch die zweite Bedingung $B_u \ge 0$ erfüllt ist und auch $A_d = B_u = 0$ nicht eintreten kann. Zu beachten ist außerdem, dass bez. B_u die Bedingung (A.9) und bez. A_d die Bedingung (A.10) erfüllt ist, womit die gerichtete Abrundung des Funktionswertes $f(x, y)$ gesichert ist. Die Berechnung von B_u erfolgt wieder mithilfe der MPFR-Funktion `mpfr_add`(B_u, $|x|_u$, $|y|_u$, `RoundUp`);

Im Fall $A := |x| - |y| \ge 0$ kann der Nachweis für eine gesicherte Abrundung analog geführt werden, und auch für die gesicherte Aufrundung von $f(x, y)$ erfolgt der Nachweis ganz analog.

Wir betrachten noch die implementierte Funktion $g(x, y) = x^2 + y^2$. Da eine Skalierung einen Überlauf jetzt nicht verhindern kann, wird die Summe der Quadrate direkt ausgewertet. Um z.B. die Aufrundung von $g(x, y)$ zu garantieren, muss nach (A.6) $x \odot_u x \oplus_u y \odot_u y$ berechnet werden, wobei also beide Operanden von $\oplus_u$ durch nur eine Rechenoperation realisiert werden, was nach den Bemerkungen von Seite 122 die garantierte Rundung beider Operanden gewährleistet. Das Aufrunden der Summe $x \odot_u x \oplus_u y \odot_u y$ wird wieder realisiert mithilfe der MPFR-Funktion `mpfr_add`(..., ..., ..., `RoundUp`);

A.3 $\sqrt{x^2-1}$

Da $f(x) = \sqrt{x^2-1}$ für $|x| \gg 1$ einen vorzeitigen Überlauf erzeugt, benutzen wir in diesem Fall für das gerichtete Auf- und Abrunden die Abschätzungen:

$$D(x) := |x| - \frac{0.5}{|x| - \frac{1}{|x|}} < \sqrt{x^2-1} < |x| - \frac{0.5}{|x|} =: U(x), \qquad |x| \gg 1; \tag{A.43}$$

Für $|x| \gg 1$ sind alle drei Terme in (A.43) positiv, so dass die Beweise der zwei Ungleichungen nach dem Quadrieren mit einfachen Umformungen leicht durchgeführt werden können.

Wenn also ein aufgerundeter Funktionswert $f_u(x) \geq f(x)$ zu berechnen ist, so muss $U(x)$ aufgerundet werden. Nach (A.8) muss dazu $0.5/|x|$ abgerundet werden, was mithilfe der MPFR-Funktion `mpfr_div (..., ..., ..., RoundDown)` direkt realisiert werden kann, und die aufzurundende Differenz selbst wird berechnet mit `mpfr_sub (..., ..., ..., RoundUp)`.

Wenn jedoch ein abgerundeter Funktionswert $f_d(x) \leq f(x)$ zu berechnen ist, so muss $D(x)$ abgerundet und damit der Doppelbruch aufgerundet werden. Dazu muss $N := |x| - 1/|x|$ abgerundet werden, wobei $1/|x|$ mithilfe von `mpfr_div (..., ..., ..., RoundUp)` aufzurunden ist. Die aufzurundende Division $0.5 \oslash_u N_d$ kann dann nach (A.23) problemlos durchgeführt werden, da der Zähler 0.5 rundungsfehlerfrei vorliegt und daher die geforderte Eigenschaft (A.9) automatisch erfüllt ist.

Man kann jetzt noch die Frage stellen, ob die Abschätzungen in (A.43) nicht zu grob sind, so dass bei großen Präzisionen die Fehler $U(x) - \sqrt{x^2-1}$ bzw. $\sqrt{x^2-1} - D(x)$ zu groß ausfallen. Da beide Fehler von der Größenordnung $O(1/x)^3$ sind und weil (A.43) erst zur Anwendung kommt, wenn $\texttt{expo}(x) > 1073741813$ erfüllt ist, was etwa 323228493 Dezimalstellen entspricht, so wird sich der Fehler erst nach insgesamt $323228493+3\cdot323228493 = 1292913973$ Dezimalstellen bemerkbar machen, was in der numerischen Praxis absolut keine Rolle spielt.

Wenn ein vorzeitiger Überlauf nicht eintreten kann, wird $x^2 - 1$ nach Bedarf auf- bzw. abgerundet, und weil die Wurzelfunktion eine monoton steigende Funktion ist, können $f_u(x)$ oder $f_d(x)$ problemlos berechnet werden.

A.4 $\ln(\sin(x))$

Da $f(x) := \ln(\sin(x))$ für $x \in \mathbb{R}$ nur für $\sin(x) > 0$ definiert ist, muss $\sin(x) \le 0$ bei der Auswertung von $f(x)$ ausgeschlossen werden. Dies erreicht man durch

```
mpfr_sin(res.mpfr_rep, x.mpfr_rep, RoundFromZero);
// res: sin(x), gerundet weg von der Null;
if (res <= 0)
//  ==>  sin(x) <= 0:
{
  set_nan(res);
  return res;
}
// Fuer den exakten Funktionswert gilt jetzt:  sin(x) > 0:
```

Da $\sin(x)$ für $x = \pi/2$ eine waggerechte Tangente besitzt, kann für $x_0 \approx \pi/2$ der exakte Wert $f(x_0)$ mithilfe des Funktionsterms $\ln(\sin(x))$ nur grob approximiert werden. So erhält man z.B. $f_n(x_0) = 0$, obwohl $f(x_0) \neq 0$ ist[2]. Benutzt man daher für $\sin(x) \ge 0.5$ den Funktionsterm

$$f(x) = 0.5 \cdot \ln(1 - \cos^2(x)), \quad \sin(x) \ge 0.5, \tag{A.44}$$

so wird man nach Abb. A.1 eine sehr viel bessere Auswertung erwarten können, wenn man mit dem Argument $-\cos^2(x)$ die rechte Seite von (A.44) mithilfe der Funktion `lnp1(...)` auswertet.

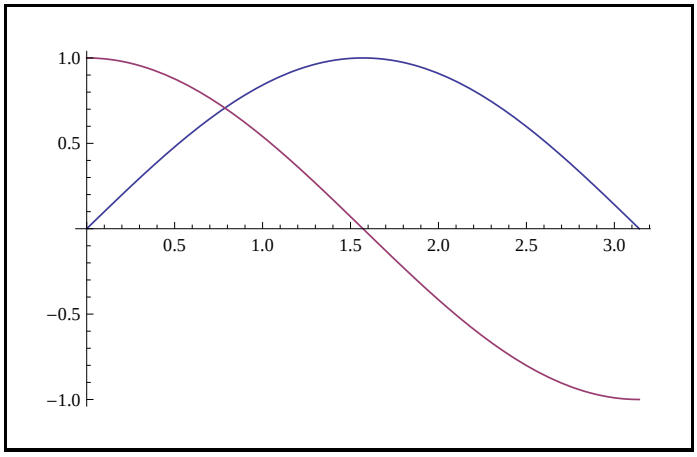


Abbildung A.1: $y = \sin(x), \cos(x)$

Für $x_0 \approx \pi/2$ kann jetzt das Argument $-\cos^2(x_0) \neq 0$, im Gegensatz zu $\sin(x_0)$, sehr viel effektiver ausgewertet werden, da die Tangente der cos-Funktion bei $x = \pi/2$ die Steigung -1 besitzt.

Jetzt wird gezeigt, wie man mit (A.44) einen garantiert abgerundeten Wert $f_d(x)$ berechnet. Wegen der monoton wachsenden Funktion $\ln(1 + x)$ muss das Argument $-\cos^2(x)$ abgerundet, d.h. $\cos^2(x)$ muss aufgerundet werden. Für die folgende Fallunterscheidung muss $\cos(x) < 0$ und $\cos(x) > 0$ unterschieden werden, und dies gelingt, wenn man vorher wie oben

```
mpfr_cos(res.mpfr_rep, x.mpfr_rep, RoundFromZero);
```

aufruft und dann bez. des von Null weggerundeten Funktionswertes `(res < 0)` bzw. `(res > 0)` abfragt. Aus `(res < 0)` folgt dann für den exakten Funktionswert die Aussage $\cos(x) < 0$ und aus `(res > 0)` folgt entsprechend $\cos(x) > 0$[3].

[2] $f_n(x_0)$ bedeutet den zur nächsten Rasterzahl gerundeten Funktionswert.

[3] Diese Aussagen gelten auch dann, wenn die betrachtete Funktion, hier $\cos(x)$, mit `RoundFromZero` viel gröber von der Null weggerundet wird. Wenn der exakte Funktionswert jedoch verschwindet, so muss dies auch für den auf- oder abgerundeten Funktionswert zutreffen, was bei allen implementierten Funktionen realisiert ist!

Sei also: $\mathbf{\cos(x) < 0}$.
Dann gilt für den abgerundeten Funktionswert $\cos_d(x)$

$$\cos_d(x) \leq \cos(x) < 0, \tag{A.45}$$

und mit (A.18) folgt dann

$$\cos_d(x) \odot_u \cos_d(x) \geq \cos^2(x), \tag{A.46}$$

wobei wegen (A.45) die Bedingung (A.18) automatisch erfüllt ist und auch $\cos_d(x)$ nicht verschwinden kann.

Sei jetzt: $\mathbf{\cos(x) > 0}$.
Dann gilt für den aufgerundeten Funktionswert $\cos_u(x)$

$$\cos_u(x) \geq \cos(x) > 0, \tag{A.47}$$

und mit (A.15) folgt dann

$$\cos_u(x) \odot_u \cos_u(x) \geq \cos^2(x), \tag{A.48}$$

wobei wegen (A.47) die Bedingung (A.9) automatisch erfüllt ist und auch $\cos_u(x)$ nicht verschwinden kann.

Jetzt wird gezeigt, wie man mit (A.44) einen garantiert aufgerundeten Wert $f_u(x)$ berechnet, dabei werden die Bedingungen $\cos(x) > 0$ bzw. $\cos(x) < 0$ ganz analog zur Berechnung von $f_d(x)$ abgefragt. Der Ausdruck $\cos^2(x)$ muss jetzt abgerundet werden.
Sei also: $\mathbf{\cos(x) < 0}$.
Dann gilt für den aufgerundeten Funktionswert $\cos_u(x)$

$$\cos(x) \leq \cos_u(x) \leq 0,$$

und mit (A.14) folgt dann die gewünschte garantierte Abrundung:

$$\cos_u(x) \odot_d \cos_u(x) \leq \cos^2(x).$$

Sei jetzt: $\mathbf{\cos(x) > 0}$.
Dann gilt für den abgerundeten Funktionswert $\cos_d(x)$

$$0 \leq \cos_d(x) \leq \cos(x),$$

und mit (A.11) folgt dann wieder die gewünschte garantierte Abrundung:

$$\cos_d(x) \odot_d \cos_d(x) \leq \cos^2(x).$$

Im unkritischen Bereich $0 < \sin(x) < 0$ wird $f(x) := \ln(\sin(x))$ direkt ausgewertet, wobei wegen der Monotonie der ln-Funktion zur Berechnung von $f_u(x)$ auch $\sin(x)$ aufzurunden ist. Für die Berechnung von $f_d(x)$ ist entsprechend $\sin(x)$ abzurunden, was mit der MPFR-Funktion `mpfr_sin(...,RoundDown)` einfach realisiert werden kann.

Die Funktion $f(x) = \ln(\cos(x))$ ist in `mpfrclass.cpp` ganz analog implementiert und muss daher nicht weiter behandelt werden.

A.5 $\ln(\sqrt{x^2+y^2})$

Wir betrachten

$$f(x) := \ln(\sqrt{x^2+y^2}), \quad \text{mit:} \quad |x| \geq |y| \text{ und } |x|, |y| > 0. \tag{A.49}$$

Um im Fall $|x| \gg 1$ einen vorzeitigen Überlauf zu vermeiden, wird folgende Darstellung benutzt:

$$f(x) := \ln(|x|) + \frac{1}{2}\ln\left(1 + (|y|/|x|)^2\right) > 0, \quad |y|/|x| > 0,\ |x| \gg 1. \tag{A.50}$$

Zur Berechnung von $f_u(x)$ bzw. $f_d(x)$ muss $\ln(|x|)$ auf- bzw. abgerundet werden, was mithilfe der MPFR-Funktion `mpfr_log(...)` direkt realisiert werden kann. Da auch der zweite Summand in (A.50) auf- bzw. abzurunden ist und die `lnp1`-Funktion monoton wächst, muss das Quadrat $(|y|/|x|)^2$ ebenfalls auf- bzw. abgerundet werden und damit auch der Quotient $A := |y|/|x|$. Da die Operanden von A rundungsfehlerfreie Rasterzahlen sind, können A_u und A_d direkt mithilfe der MPFR-Funktion `mpfr_div(...)` berechnet werden. Es gilt dann:

$$A_u \geq A > 0 \tag{A.51}$$

$$0 \leq A_d < A. \tag{A.52}$$

Es soll jetzt $(|y|/|x|)^2$ aufgerundet werden. Mit (A.15) folgt dann

$$A_u \odot_u A_u \geq (|y|/|x|)^2,$$

wobei (A.9) wegen (A.51) automatisch erfüllt ist und $A_u > 0$ garantiert ist.

Es soll jetzt $(|y|/|x|)^2$ abgerundet werden. Mit (A.11) folgt dann

$$A_d \odot_d A_d \leq (|y|/|x|)^2,$$

wobei $A_d \geq 0$ durch (A.52) garantiert wird.

Wenn $f_u(x)$ und $f_d(x)$ nach (A.49) zu berechnen sind, so ist dies kein Problem, da $\sqrt{x^2+y^2}$ mithilfe der Funktion `sqrtx2y2(..., rnd)` direkt auf- oder abgerundet werden kann und weil die ln-Funktion monoton wachsend ist.

A.6 $\mathbf{arcosh}(1+x)$

Wir betrachten für $x \geq 0$

$$\operatorname{arcosh}(1+x) = \ln(1 + x + \sqrt{x\cdot(2+x)}), \quad x \geq 0, \tag{A.53}$$

$$= \ln(x) + \ln(1 + 1/x + \sqrt{1+2/x}), \quad x \gg 1; \tag{A.54}$$

In (A.54) ist die Rundung auch des zweiten Summanden unproblematisch, da bei den zwei auftretenden Divisionen beide Operanden jeweils rundungsfehlerfreie Rasterzahlen sind und neben der ln-Funktion auch die Quadratwurzel monoton wachsende Funktionen sind.

In (A.53) muss bei der Berechnung von $f_u(x)$ und $f_d(x)$ das Produkt $P := x\cdot(2+x) = A\cdot B$ auf- bzw. abgerundet werden, wobei $B := 2 + x \geq 2$ ebenfalls auf- bzw. abzurunden ist. Es gilt:

$$0 \leq A = A_u = A_d, \tag{A.55}$$

$$2 \leq B_d \leq B, \tag{A.56}$$

$$2 \leq B \leq B_u; \tag{A.57}$$

Nach (A.15) gilt: $P_u := A_u \odot_u B_u \geq x \cdot (2 + x)$, wobei die Bedingung (A.9) wegen (A.55) automatisch erfüllt ist und auch $A_u = A$ und $B_u \geq 2$ nicht beide verschwinden können.

Nach (A.11) gilt wegen (A.55) und (A.56) unmittelbar: $P_d := A_d \odot_d B_d \leq x \cdot (2 + x)$, womit für P_d und P_u die entsprechenden Rundungen garantiert sind. Den vollständigen Algorithmus findet man in `mpfrclass.cpp`.

A.7 $\Gamma'(x)$

$\Gamma'(x)$ wird mithilfe der Digamma-Funktion $\psi(x) = \Gamma'(x)/\Gamma(x)$ über das folgende Produkt berechnet:

$$\Gamma'(x) = \psi(x) \cdot \Gamma(x), \quad x \neq 0, -1, -2, \ldots \tag{A.58}$$

Die Funktionen $\psi(x)$ und $\Gamma(x)$ stehen dabei als `digamma(x)` bzw. `gamma(x)` zur Verfügung. Um das Produkt $\psi(x) \cdot \Gamma(x)$ gesichert auf- bzw. abzurunden, werden $\psi(x)$ und $\Gamma(x)$ jeweils durch die Intervalle `u` und `v` eingeschlossen. Mit dem Intervallprodukt `u = u ⋄ v` ist dann das auf- bzw. abgerundete Produkt $\psi(x) \cdot \Gamma(x)$ gegeben durch `Sup(u)` bzw. `Inf(u)`, und der nächstgelegene Wert ist gegeben durch `mid(u)`. Der Vorteil dieser Methode ist einmal die kurze Laufzeit, da zur Berechnung von `v` nur ein Funktionsaufruf `Inf(v) = gamma(x,RoundDown)` notwendig ist, wobei dann `Sup(v)` einfach durch den Nachfolger `succ(Inf(v)) = Sup(x)` zu berechnen ist. Die geschilderte Intervallmethode hat zusätzlich den Vorteil, dass die sonst notwendigen Fallunterscheidungen jetzt entfallen, wodurch ein übersichtlicher Programmcode gewährleistet wird. Aber jedes Verfahren besitzt auch einen Nachteil, denn wenn in seltenen Ausnahmefällen für `Inf(v) := gamma(x,RoundDown)` gilt: `Inf(v)` $= \Gamma(x)$, so wird mit `Sup(x) = succ(Inf(v))` das Intervall `v` leicht nach oben überschätzt, da bei diesem Beispiel das ideale Intervall `v` ein Punktintervall ist. Den vollständigen Algorithmus findet man in `mpfrclass.cpp`.

A.8 $1/\Gamma(x)$

$f(x) := 1/\Gamma(x)$ ist in der ganzen komplexen Ebene analytisch und besitzt an den Polstellen $x_k = 0, -1, -2, \ldots$ der Γ-Funktion die Funktionswerte Null. Für alle anderen reellen x-Werte wird $f(x)$ approximiert durch den Quotienten $1/\Gamma(x)$ der auf der Maschine mittels `gamma(x)` ausgewertet wird.

Um $f_u(x)$ zu berechnen, ist nach (A.23) und (A.25) $\Gamma(x)$ unabhängig von seinem Wert stets abzurunden und die Division muss aufgerundet werden.

Um $f_d(x)$ zu berechnen, ist nach (A.19) und (A.21) $\Gamma(x)$ unabhängig von seinem Wert stets aufzurunden und die Division muss abgerundet werden.

A.9 $(1/\Gamma(x))'$

Für die überall differenzierbare Funktion $f(x) := (1/\Gamma(x))'$ gilt:

$$f(x) = \left(\frac{1}{\Gamma(x)}\right)' = \begin{cases} \dfrac{-\Gamma'(x)}{\Gamma^2(x)} = -\dfrac{\Gamma'(x)}{\Gamma(x)}/\Gamma(x), & x \neq 0, -1, -2, \ldots \\ |x|!, & x = 0, -2, -4, \ldots \\ -|x|!, & x = -1, -3, -5, \ldots \end{cases} \tag{A.59}$$

$|x|!$ wird mithilfe der `faktorial(...)`-Funktion berechnet und die erste Zeile in (A.59) mit dem Quotienten `-digamma(x)/gamma(x)`, wobei zur Vermeidung von Fallunterscheidungen Zähler und Nenner und auch die Division selbst intervallmäßig ausgewertet werden. In der Umgebung von Null, d.h. für $|x| \leq 2 \cdot \mathtt{succ}(0)$ und $x \neq 0$, entsteht jedoch das Problem, dass `digamma(x)` und `gamma(x)` beide einen Überlauf erzeugen, obwohl ihr Quotient von der Größenordnung 1 ist. Um diesen Überlauf zu vermeiden, wird ausgenutzt, dass $f'(x)$ für $x \in [-1, 0.3]$ positiv ist, so dass $f(x)$ in obiger Umgebung von Null monoton wächst. Für $x = \mathtt{succ}(0)$ und $x = 2 \cdot \mathtt{succ}(0)$ ist damit wegen $f(0) = 1$ eine Einschließung von $f(x)$ gegeben durch:

$$t = 3 \cdot \mathtt{succ}(0), \quad u = \mathtt{digamma(t,\ RoundDown)}, \quad v = \mathtt{gamma(t,\ RoundDown)},$$
$$u = -u \diamondsuit v;$$
$$f(x) \in [1,\ \mathtt{Sup}(u)], \quad x = \mathtt{succ}(0) \text{ oder } x = 2 \cdot \mathtt{succ}(0)$$

wobei wegen $t = 3 \cdot \mathtt{succ}(0)$ die Punktintervalle u und v jetzt keinen Überlauf mehr erzeugen. Natürlich ist das Einschließungsintervall $[1, \mathtt{Sup}(u)]$ eine gewisse Überschätzung von $f(x)$, aber dies wird in der numerischen Praxis kaum eine Rolle spielen.

Für die negativen Argumente $x = \mathtt{pred}(0)$ und $x = 2 \cdot \mathtt{pred}(0)$ gelten ganz entsprechende Aussagen. Den ausführlichen Algorithmus findet man in `mpfrclass.cpp`.

B Elementarfunktionen für komplexe Punkt- und Intervallargumente

Die Aufgabenstelung wird wie folgt beschrieben:

Zu einem vorgegebenen Intervallargument

$$Z = X + i \cdot Y; \quad X = [x_1, x_2], \quad Y = [y_1, y_2]; \tag{B.1}$$

mit reellen Intervallen X, Y ist zu einer gegebenen komplexwertigen Funktion $w = f(z)$, mit $z, w \in \mathbb{C}$, eine möglichst optimale Einschließung aller Funktionswerte w gesucht, wenn $z \in Z$, d.h. einzuschließen ist

$$W = f(Z) := \{w \,|\, w = f(z) \wedge z \in Z\}, \; z = x + i \cdot y. \tag{B.2}$$

Zu beachten ist, dass das Bild $W = f(Z)$ des Argumentintervalls Z i.a. kein achsenparalleles Rechteck ist, so dass die Einschließung von W durch ein Intervall $F(Z) = U(Z) + i \cdot V(Z)$ auch dann Überschätzungen liefert, wenn das Rechteck F die Menge W **optimal** einschließt, wobei die Überschätzungen in Abb. B.1 rechts durch die vier weißen Rechteckspitzen realisiert werden, [28, S.16].

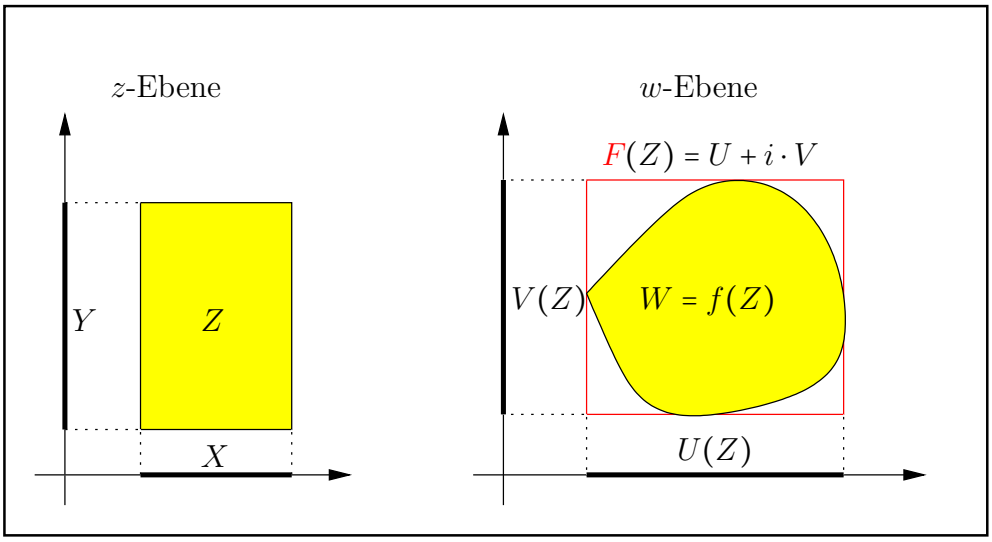


Abbildung B.1: Intervall-Funktion

Jetzt muss noch geklärt werden, wie die reellen Intervalle $U(Z)$ und $V(Z)$ zu berechnen sind.

Dazu betrachten wir zunächst den einfachsten Fall, dass mit $f(z) = u + i \cdot v$ die Funktionen $u(x, y)$ und $v(x, y)$ separabel sind, d.h. für z.B. u soll im Falle einer Multiplikation gelten:

$$u(x, y) = u_1(x) \cdot u_2(y), \quad x \in X, \; y \in Y.$$

Eine optimale Einschließung von $u(x, y)$ erhält man aber nur dann, wenn in den auszuwertenden Ausdrücken u_1 und u_2 die Variablen x bzw. y jeweils **nur einmal** vorkommen und wenn das Produkt $u_1(x) \cdot u_2(y)$ intervallmäßig ausgewertet wird, [4]. Es gilt dann auch für $v(x, y)$

$$\begin{aligned} u(x, y) &= u_1(x) \cdot u_2(y) \in u_1(X) \diamond u_2(Y) =: U(Z), \\ v(x, y) &= v_1(x) \cdot v_2(y) \in v_1(X) \diamond v_2(Y) =: V(Z). \end{aligned}$$

Ein Beispiel für eine separable Funktion ist die komplexe Exponentialfunktion, bei ihr gilt:

$$u(x,y) = e^x \cdot \cos(y), \qquad v(x,y) = e^x \cdot \sin(y), \quad x \in X,\ y \in Y. \tag{B.3}$$

Weitere Beispiele für separable Funktionen sind: $\sin(z)$, $\cos(z)$, $\sinh(z)$ und $\cosh(z)$, wobei deren Implementierung besonders einfach ist, da die dabei benötigten reellen Intervall-Funktionen $\sin(X)$, $\cos(X)$, $\cosh(Y)$ und $\sinh(Y)$ bereits in `mpficlass.cpp` definiert sind, vgl. auch Tab. 4.1 auf Seite 66.

Wir kommen jetzt zu den **nicht-separablen** Funktionen $f(z) = u(x,y) + i \cdot v(x,y)$, bei denen die Berechnung von $U(Z) \ni u(x,y)$ und $V(Z) \ni v(x,y)$ ausnahmslos sehr viel schwieriger ist. Eine Basis zur Berechnung der reellen Intervalle U, V liefert jetzt der folgende Satz, [6].

Schreibt man mit $z = x + i \cdot y \in \mathbb{C}$ die Funktion $w = f(z)$ in der Form

$$w = f(z) = u(x,y) + i \cdot v(x,y), \quad z = x + i \cdot y \in Z \subset \mathbb{C}, \quad \text{so gilt:}$$

Ist $f = u + i \cdot v : Z \subset G \longrightarrow \mathbb{C}$ holomorph im Gebiet G, so nehmen sowohl $u(x,y)$ als auch $v(x,y)$ als harmonische Funktionen ihr Maximum und Minimum auf dem Rand von Z an.

Die Extrema von $u(x,y)$ und $v(x,y)$ müssen also nur auf dem **Rand** von Z gesucht werden.

Wir betrachten auch jetzt wieder den einfachsten Fall, wenn die Koordinaten der Randpunkte m und M, in denen das Minimum bzw. Maximum angenomen wird, Maschinenzahlen des durch `prec` bestimmten Zahlenformats sind. In der folgenden Abb. A.2 sind für verschiedene Argumentintervalle Z die Punkte m und M angegeben, in denen für die Realteilfunktion $u(x,y)$ der $\arcsin(z)$-Funktion das Minimum bzw. Maximum angenommen wird.

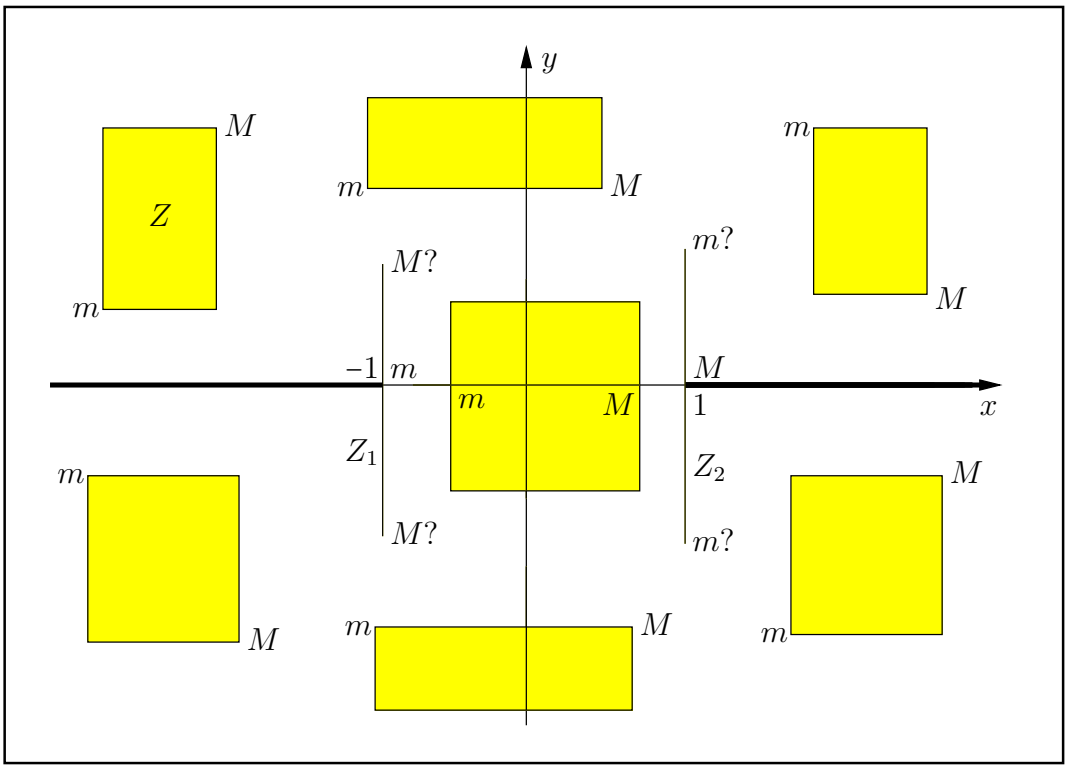


Abbildung B.2: Die Lage der Punkte m, M auf Z beim Realteil von $\arcsin(z)$.

Bei den mit '?' gekennzeichneten Punkten ist deren y-Koordinate als Maximum von $-y_1$ und y_2 definiert, mit $Y := [y_1, y_2]$. Für die durch -1 bzw. $+1$ laufenden Argumentintervalle Z_1, Z_2 gilt:

$$m = (1, \mathtt{max}(-y_1, y_2)); \qquad M = (-1, \mathtt{max}(-y_1, y_2));$$

Aus Abb. B.2 erkennt man, dass die Koordinaten von m und M alles Maschinenzahlen sind. Bedeutet $u(x,y)$ die Realteilfunktion der $\arcsin(z)$-Funktion und ist $u_d(m)$ der abgerundete und $u_u(M)$ der aufgerundete Funktionswert, jeweils an den Punkten m und M, so ist $U(Z)$ gegeben durch

$$U(Z) := [u_d(m), u_u(M)] \ni u(x,y), \quad z = x + i \cdot y \in Z.$$

Es stellt sich jetzt noch die Frage, wie die Werte $u_d(m)$ und $u_u(M)$ zu berechnen sind. Dazu gibt es im Wesentlichen drei Möglichkeiten:

Methode 1. Nach [15],[16],[28],[29] wird eine garantierte Fehlerschranke für $u(x,y)$ berechnet, mit der dann die Werte $u_d(m)$ und $u_u(M)$ bestimmt werden können. Dies ist jedoch programmiertechnisch sehr aufwendig, so dass meist das folgende Verfahren zur Anwendung kommt.

Methode 2. Wenn zur intervallmäßigen Auswertung von $u(x,y)$ die entsprechenden Intervallfunktionen bereits implementiert sind, so kann $u(x,y)$ an den Stellen m und M durch die Intervalle $\boldsymbol{u}(m)$ und $\boldsymbol{u}(M)$ eingeschlossen werden, und $U(Z)$ ist gegeben durch, [43],[44]

$$U(Z) = [\texttt{Inf}(\boldsymbol{u}(m)), \texttt{Sup}(\boldsymbol{u}(M))].$$

Dieses Verfahren kann programmiertechnisch vergleichsweise einfach realisiert werden, es erzeugt aber wegen der notwendigen Intervall-Auswertungen deutlich größere Laufzeiten.

Methode 3. Wenn $u(x,y)$ für punktförmige Maschinenzahlen x, y so implementiert ist, dass man mit u(x, y, `RoundDown`) bzw. u(x, y, `RoundUp`) garantiert ab- bzw. aufgerundetet Funktionswerte im Punkt $P(x,y)$ erhält, so kann $U(Z)$ realisiert werden durch

$$U(Z) = [\texttt{u}(m, \texttt{RoundDown}), \texttt{u}(M, \texttt{RoundUp})].$$

Der Nachteil ist jetzt, dass u(x, y, `rnd`) mit den Rundungsparametern `rnd = RoundDown` und `rnd = RoundUp` mit Hilfe der MPFR- und MPFI-Bibliotheken sehr sorgfältig zu implementieren ist, [20],[21],[41],[45]. Der Vorteil ist aber die nahezu optimale Laufzeit, da Intervallauswertungen jetzt nicht zur Anwendung kommen.

Bei der Realisierung von $U(Z)$ und $V(Z)$ sollte daher nur die Methode 3. zur Anwendung kommen. Nur in wirklich komplizierten Fällen wird man daher auf die Methode 2. zurückgreifen.

Abschließend noch einige Bemerkungen zur Lage der achsenparallelen Argumentintervalle Z in der komplexen Ebene.

- Ist $f(z)$ holomorph in der ganzen komplexen Ebene, so kann $Z = X + i \cdot Y$ beliebig gewählt werden, wobei aber die Randpunkte der reellen Intervalle X und Y Maschinenzahlen sind, so dass damit das komplexe Intervall Z stets endlich sein wird.

- Der Hauptwert der komplexen Funktion $f(z) = \arcsin(z)$ besitzt nach Abb. B.2 zwei Verzweigungsschnitte auf der reellen Achse von 1 bis $+\infty$ und von $-\infty$ bis -1. In diesem Fall darf ein Verzweigungsschnitt das Intervall Z nicht im Innern durchlaufen, vielmehr darf Z einen Verzweigungsschnitt höchstens von oben oder von unten berühren. In der CoStLy-Bibliothek von M. Neher, [43],[44], darf Z keinen Punkt der beiden Verzweigungsschnitte enthalten; lediglich Argumentintervalle Z der Breite Null durch die Verzweigungspunkte $P_1(-1,0)$ und $P_2(+1,0)$ sind zugelassen.

- Bei mehrdeutigen Funktionen werden stets Einschließungen des jeweiligen Hauptwertes berechnet, wobei die erlaubten bzw. verbotenen Lagen der Argumentintervalle Z stets anzugeben sind.

Wir kommen jetzt zum kompliziertesten Fall, wenn die Extremalpunkte m, M nicht mehr Eckpunke oder Schnittpunkte von Z mit den Achsen sind. In diesem Fall ist dann i.a. nur noch eine Koordinate von m oder M eine Maschinenzahl, so dass die Auswertung der Real- bzw. Imaginärteilfunktion $u(x, y)$ bzw. $v(x, y)$ auf der Maschine nicht mehr so ohne Weiteres möglich ist. Als Beispiel betrachten wir in Abb. B.3 für den Hauptwert der $\arctan(z)$-Funktion in der oberen Halbebene einige Intervalle Z, bei denen nur die y-Koordinate des jeweiligen Punktes m eine Maschinenzahl ist, [28].

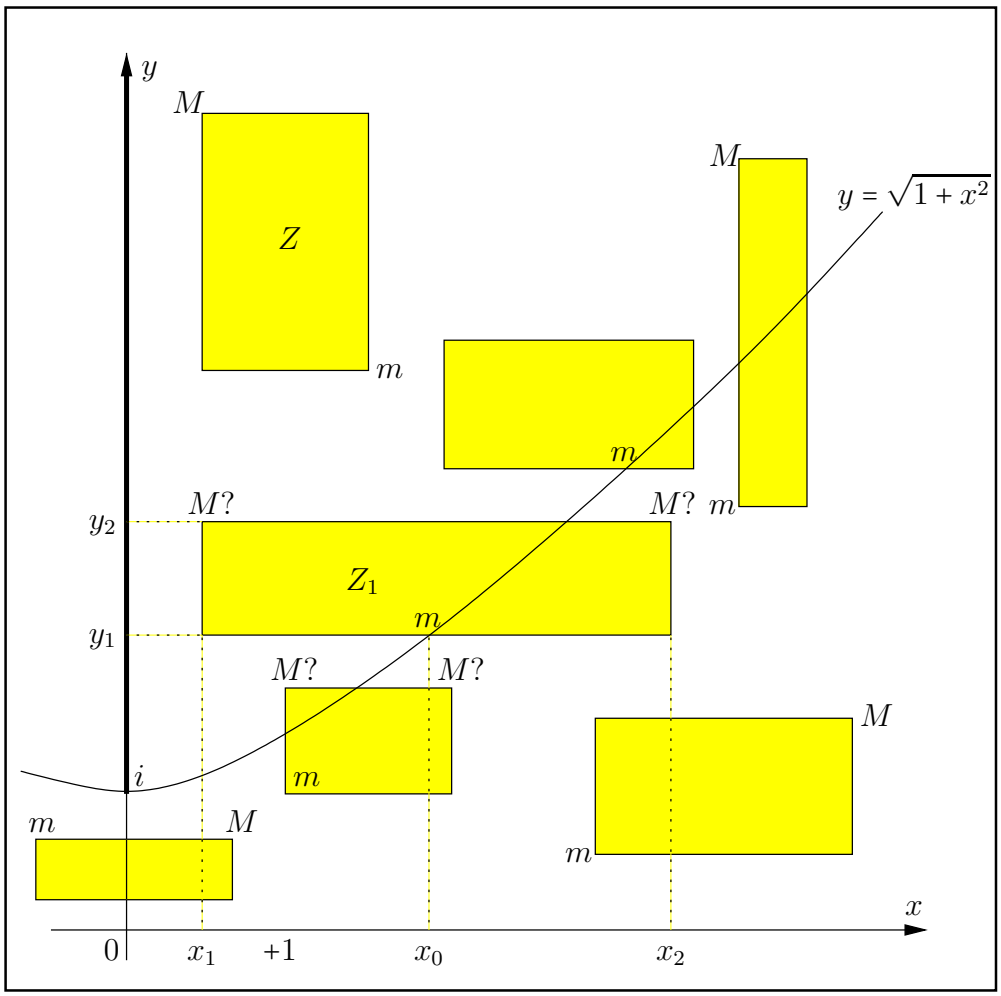


Abbildung B.3: Die Lage der Punkte m, M auf Z beim Realteil von $\arctan(z)$.

Wir betrachten das Argumentintervall Z_1 im 1. Quadranten außerhalb des Einheitskreises um 0. Für die Realteilfunktion $u(x, y)$ von $\arctan(z)$ gilt in diesem Bereich

$$u(x,y) := \frac{1}{2}\arctan\frac{2x}{1-x^2-y^2} + \frac{\pi}{2}, \tag{B.4}$$

und wenn für die x-Koordinate x_0 von m gelten soll $x_1 < x_0 < x_2$, so muss für ein lokales Minimum in m die notwendige Bedingung

$$\frac{\partial u(x,y)}{\partial x} = 0 \quad \Longleftrightarrow \quad y = +\sqrt{1+x^2}$$

erfüllt sein, d.h. es muss gelten $y_1 = +\sqrt{1+x_0^2}$, wobei $x_0 = \sqrt{y_1^2 - 1}$ i.a. keine Maschinenzahl ist. Glücklicherweise wird aber x_0 nicht explizit benötigt, da zur Einschließung von $u(x, y)$ in Z ein

abgerundeter Wert von $u(x_0, y_1)$ benötigt wird. Dazu setzt man in (B.4) $x = \sqrt{y_1^2 - 1}$ und $y = y_1$ und erhält

$$u(x_0, y_1) := -\frac{1}{2} \arctan \frac{1}{\sqrt{y_1^2 - 1}} + \frac{\pi}{2} =: u(y_1), \tag{B.5}$$

wobei jetzt $u(y_1)$ mit der Maschinenzahl y_1 problemlos ausgewertet werden kann. Ein abgerundeter Wert von $u(y_1)$ kann jetzt wieder analog zu Seite 137 mit drei verschiedenen Methoden berechnet werden.

Methode 1. Nach [15],[16],[28],[29] wird eine garantierte Fehlerschranke für $u(y)$ berechnet, mit der dann der abgerundeter Wert von $u(y_1)$ bestimmt werden kann. Dies ist jedoch programmiertechnisch wieder sehr aufwendig, so dass meist das folgende Verfahren zur Anwendung kommt.

Methode 2. Da zur intervallmäßigen Auswertung von $u(y)$ die entsprechenden Intervallfunktionen bereits implementiert sind, kann $u(y_1)$ durch das Intervall $\boldsymbol{u}(y_1)$ eingeschlossen werden, und der gesuchte abgerundete Wert ist gegeben durch, [43],[44]

$$\texttt{Inf}(\boldsymbol{u}(y_1)) \le u(y_1) \le u(x, y), \quad z = x + i \cdot y \in Z..$$

Dieses Verfahren kann programmiertechnisch vergleichsweise einfach realisiert werden, es erzeugt aber wegen der notwendigen Intervall-Auswertungen deutlich größere Laufzeiten.

Methode 3. Wenn $u(y)$ für die punktförmigen Maschinenzahlen y so implementiert ist, dass man mit u(y, `RoundDown`) einen garantiert abgerundeten Funktionswert erhält, so ist dieser die gesuchte Unterschranke

$$u(y_1, \texttt{RoundDown}) \le u(y_1) \le u(x, y), \quad z = x + i \cdot y \in Z.$$

Der Nachteil ist jetzt, dass $u(y, \texttt{RoundDown})$ mit Hilfe der MPFR- und MPFI-Bibliotheken sehr sorgfältig zu implementieren ist, [20],[21],[41],[45]. Der Vorteil ist aber die nahezu optimale Laufzeit, da Intervallauswertungen jetzt nicht zur Anwendung kommen.

Anmerkungen:

1. Da die Bedingung $\partial u(x_0, y_1)/\partial x = 0$ im Punkt m für ein lokales Minimum nur notwendig ist, muss die Existenz des lokales Minimums zusätzlich abgesichert werden. In [28, Seite 145] erfolgt dies durch eine allgemeine Monotoniebetrachtung der Funktion $u(x, y)$.

2. Nach Abb. B.3 schneidet die Hyperbel $y = +\sqrt{1 + x^2}$ auch die obere Parallele von Z_1, so dass auch dort ein lokales Minimum auftrete könnte. Wegen $0 < y_1 < y_2$ gilt jedoch nach (B.5) $u(y_1) < u(y_2)$, so dass das lokale Minimum der Realteilfunktion $u(x, y)$ wirklich in m angenommen wird.

3. Für die Realteilfunktion $u(x, y)$ liefert $\partial u(x, y)/\partial x = 0$ den geometrischen Ort aller derjenigen Punkte $P(x, y)$, in denen auf einer Parallelen zur x-Achse, z.B. durch $y = y_1$, ein lokaler Extremwert auftreten kann. Dieser geometrische Ort wird auch als Extremalkurve bezeichnet und ist bei der $\arctan(z)$-Funktion für den Realteil die Hyperbel $y^2 - x^2 = 1$. Schneidet also einer der beiden Hyperbeläste $y = \pm\sqrt{1 + x^2}$ einen zur reellen Achse parallelen Rand von Z, so können relative Extrema auf diesen Rändern nur in diesen Schnittpunkten angenommen werden. Ganz entsprechend liefert ein Schnittpunkt der Extremalkurve $\partial u(x, y)/\partial y = 0$ mit einer Randparallelen von Z zur imaginären Achse einen Punkt, in dem ein lokales Extremum auf dieser Parallelen zur y-Achse auftreten kann.

4. Für die Imaginärteilfunktion $v(x, y)$ sind die beiden Extremalkurven $\partial v(x, y)/\partial x = 0$ und $\partial v(x, y)/\partial y = 0$ und ihre Schnittpunkte mit dem Rand von Z ganz entsprechend zu betrachten.

B.1 Elementarfunktionen für komplexe Punktargumente

Alle nachfolgenden Funktionen mit Punktargumenten z vom Typ `MpfcClass` können mit einem zusätzlichen Rundungsparameter `rnd` aufgerufen werden, wobei nur die Rundungen

$$\texttt{rnd = RoundNearest}, \quad \texttt{rnd = RoundDown}, \quad \texttt{rnd = RoundUp}$$

zur Verfügung stehen. Die obigen Rundungen werden aber nicht immer optimal ausgeführt, d.h. mit z.B. `rnd = RoundUp` wird nicht die nächst-größere Rasterzahl berechnet, sondern nur eine der benachbarten Rasterzahlen, die garantiert rechts vom exakten Funktionswert liegt. Wird `rnd` nicht gesetzt, so wird ebenfalls nur in die unmittelbare Nähe des exakten Funktionswertes gerundet. Die für die Implementierung der neuen Funktionen benötigten Bedingungen für das korrekte Runden bei den vier Grundoperationen werden ausführlich beschrieben im Abschnitt A.1 ab Seite 121. Die Elementarfunktionen für komplexe Punktargumente sind in Tabelle 5.1 auf Seite 88 zusammengestellt. Im ersten sehr einfachen Beispiel betrachten wir nur den Realteil der komplexen Exponentialfunktion, wobei nur die korrekte Rundung bei den Operanden und bei der Multiplikation zu beachten ist.

B.1.1 Exponentialfunktion, Realteil

Nach (B.3) von Seite 136 gilt mit $z = x + i \cdot y \in \mathbb{C}$ für den Realteil $u(x, y)$ der Exponentialfunktion

$$\Re(e^z) := u(x, y) = e^x \cdot \cos(y). \tag{B.6}$$

Zunächst sollen nur **aufgerundete** Funktionswerte von $u(x, y)$ berechnet werden. Nach (A.15) und (A.16) von Seite 123 muss dazu der Funktionswert $\cos(y)$ stets aufgerundet werden. Der nachfolgende Code zeigt die entsprechenden Anweisungen.

```
1   MpfrClass re(0), tmp(0);
2   // Calculating the real part:
3   if (rnd==MPFR_RNDU)
4   {
5     mpfr_cos (tmp.GetValue(), z.mpfr_im, rnd);   // tmp = cos( Im(z) )
6     if (tmp < 0)
7       mpfr_exp (re.GetValue(), z.mpfr_re, MPFR_RNDD);
8     else mpfr_exp (re.GetValue(), z.mpfr_re, rnd);
9   };
10  mpfr_mul (re.GetValue(), re.GetValue(), tmp.GetValue(), rnd);
```

In Zeile 5 wird der aufgerundete Funktionswert $\cos(y)$ brechnet und in `tmp` gespeichert. Nach (A.15) und (A.16) muss e^x im Fall `tmp` < 0 abgerundet und andernfalls aufgerundet werden, was in den Zeilen 7 und 8 realisiert wird. Nach (A.15) wird noch verlangt

$$\texttt{tmp} > 0 \implies \cos(y) > 0 \quad \text{und} \quad \texttt{re} > 0 \implies e^x > 0.$$

Die beiden Bedingungen sind wegen der optimalen Rundung der Funktionen aus der MPFR-Bibliothek sicher erfüllt. Mit (A.15) wird zusätzlich noch verlangt, dass nur einer der beiden aufgerundeten Funktionswerte verschwindet. Wegen der positiven Exponentialfunktion ist auch diese Forderung sicher erfüllt. Nach (A.15) und (A.16) ist bei der Multiplikation der gerundeten Werte `tmp` und `re` jeweils aufzurunden, was in Zeile 10 realisiert wird, wobei der aufgerundete Realteil von e^z in `re` gespeichert wird. Die Berechnung eines abgerundeten Realteils erfolgt nach (A.11) und (A.12) ganz analog. Der entsprechende Quelltext, auch für die Berechnung des Imaginärteils, befindet sich in der Datei `mpfcclass.cpp`.

B.1.2 $\sin(z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist der komplexe Sinus definiert durch

$$\sin(z) = \sin(x) \cdot \cosh(y) + i \cdot \cos(x) \cdot \sinh(y).$$

Die Berechnung der gerundeten Realteilwerte erfolgt ganz analog zum Realteil der Exponentialfunktion, da einer der Faktoren stets positiv ist. Im Vergleich dazu ist die Auswertung des Imaginärteils etwas komplizierter, da neben $\cos(x)$ auch der zweite Faktor $\sinh(y)$ sowohl positiv als auch negativ werden kann. Beachten Sie, dass die vier obigen Funktionen aus der MPFR-Bibliothek direkt zur Verfügung stehen und auf der Maschine **optimal** gerundet werden. Daher gelten für diese Funktionen die Aussagen (A.36) bis (A.41) auf Seite 125. Weitere Einzelheiten findet man in der Funktion `MpfcClass sin(const MpfcClass& z, RoundingMode rnd);` die in der Datei `mpfcclass.cpp` definiert ist.

B.1.3 $\cos(z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist der komplexe Cosinus definiert durch

$$\cos(z) = \cos(x) \cdot \cosh(y) - i \cdot \sin(x) \cdot \sinh(y).$$

Die Berechnung der gerundeten Real- und Imaginärteilwerte erfolgt analog zur $\sin(z)$-Funktion, wobei zusätzlich das Minus-Zeichen beim Imaginärteil zu beachten ist. Weitere Einzelheiten findet man in der Funktion `MpfcClass cos(const MpfcClass& z, RoundingMode rnd);` die in der Datei `mpfcclass.cpp` definiert ist.

B.1.4 $\tan(z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist der komplexe Tangens definiert durch

$$\tan(z) = \frac{\sin(2x)}{2 \cdot \{\cos^2(x) + \sinh^2(y)\}} + i \cdot \frac{\sinh(2y)}{2 \cdot \{\cos^2(x) + \sinh^2(y)\}}.$$

Die Argumente $2x$ und $2y$ werden rundungsfehlerfrei berechnet, falls kein Überlauf eintritt. Der Nenner $2 \cdot \{\cos^2(x) + \sinh^2(y)\}$ wird wegen der Quadrate ohne Auslöschung berechnet, wobei jedoch durch $\sinh^2(y)$ ein Überlauf erzeugt werden kann. Dadurch treten bei den gerundeten Real- und Imaginärteilwerten starke Überschätzungen ein. Weitere Einzelheiten findet man in der Funktion `MpfcClass tan(const MpfcClass& z, RoundingMode rnd);` die in der Datei `mpfcclass.cpp` definiert ist.

B.1.5 $\cot(z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist der komplexe Cotangens definiert durch

$$\cot(z) = \frac{\sin(2x)}{2 \cdot \{\sin^2(x) + \sinh^2(y)\}} + i \cdot \frac{\sinh(-2y)}{2 \cdot \{\sin^2(x) + \sinh^2(y)\}}.$$

Es gelten die gleichen Überlegungen wie bei der $\tan(z)$-Funktion. Real- und Imaginärteil sind implementiert in den Funktionen

```
MpfrClass Re_cot(const MpfcClass& z, RoundingMode rnd);
MpfrClass Im_cot(const MpfcClass& z, RoundingMode rnd);
```

die in der Datei `mpfcclass.cpp` definiert sind.

B.1.6 arg(z)

Für $z = x + i \cdot y \in \mathbb{C}$ wird das Argument von z, d.h. arg(z), definiert durch das Bogenmaß des Winkels, den der Fahrstrahl vom Ursprung nach z mit der positiven reellen Achse einschließt. Die komplexe Ebene ist dabei längs der negativen reellen Achse von 0 bis $-\infty$ aufgeschnitten.

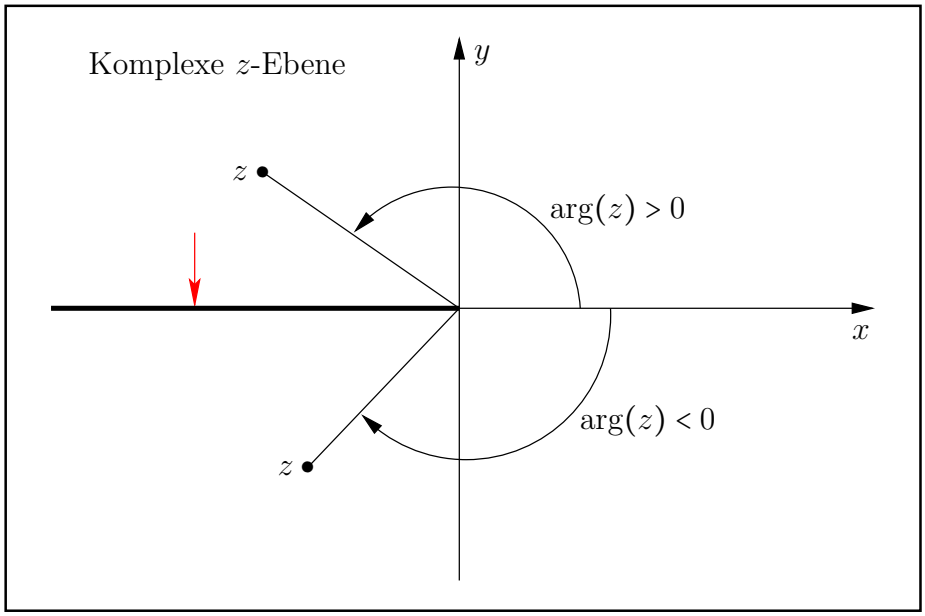


Abbildung B.4: Verzweigungsschnitt von arg(z), $z \in \mathbb{C}$.

Der Pfeil zeigt die Richtung an, aus der arg(z) auf den Verzweigungsschnitt stetig ergänzt wird, d.h. es gilt z.B. $\arg(-1 + 0 \cdot i) = +\pi$. arg($z$) wird intern mithilfe der Funktion `mpfr_atan2(...)` aus der MPFR-Bibliothek direkt ausgewertet, so dass die gewünschten Rundungen sogar optimal berechnet werden können. Der gerundete Funktionswert arg(z) wird bestimmt mit dem Funktionsaufruf

```
MpfrClass arg(const MpfcClass& z, RoundingMode rnd, PrecisionType prec);
```

Wird `prec` gesetzt, so wird arg(z) mittels `rnd` in die gewählte Präzision gerundet und dann zurückgegeben. Ohne `prec` wird mittels `rnd` in die Current-Präzision gerundet. Ohne `prec` und ohne `rnd` wird arg(z) mit dem Current-Rundungsmodus in die Current-Präzision gerundet.

In der folgenden Tabelle sind für einige spezielle $z = x + i \cdot y$-Werte die Funktionswerte arg(z) zusammengestellt.

x	y	arg(z)	x	y	arg(z)	x	y	arg(z)
0	0	0	0	$+\infty$	$\pi/2$	$-\infty$	+1	π
+1	0	0	0	−1	$-\pi/2$	$+\infty$	$+\infty$	$+\pi/4$
$+\infty$	0	0	0	$-\infty$	$-\pi/2$	$+\infty$	$-\infty$	$-\pi/4$
$+\infty$	+1	0	−1	0	π	$-\infty$	$+\infty$	$+3\pi/4$
$+\infty$	−1	0	$-\infty$	0	π	$-\infty$	$-\infty$	$-3\pi/4$
0	+1	$\pi/2$	$-\infty$	−1	$-\pi$	x oder y = NaN		NaN

Abbildung B.5: Wertetabelle für arg(z), $z \in \mathbb{C}$.

B.1.7 $|z|$

Für $z = x + i \cdot y \in \mathbb{C}$ ist der Betrag definiert durch

$$|z| := \sqrt{x^2 + y^2}.$$

Die obige Quadratwurzel wird mit der Funktion `mpfr_hypot(...)` aus der MPFR-Bibliothek direkt ausgewertet, so dass die gewünschten Rundungen sogar optimal berechnet werden können. Der Betrag von z wird bestimmt mit dem Funktionsaufruf

```
MpfrClass abs(const MpfcClass& z, RoundingMode rnd, PrecisionType prec);
```

Wird `prec` gesetzt, so wird $|z|$ mittels `rnd` in die gewählte Präzision gerundet und dann zurückgegeben. Ohne `prec` wird mittels `rnd` in die Current-Präzision gerundet. Ohne `prec` und ohne `rnd` wird $|z|$ mit dem Current-Rundungsmodus in die Current-Präzision gerundet.

B.1.8 $\log(z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist $\log(z)$ definiert durch

$$\log(z) := \log\left(\sqrt{x^2 + y^2}\right) + i \cdot \arg(z),$$

wobei der Realteil $\log\left(\sqrt{x^2 + y^2}\right)$ mit der vordefinierten Funktion

```
MpfrClass ln_sqrtx2y2(const MpfrClass&x, const MpfrClass&y, RoundingMode rnd);
```

aus `mpfrclass.cpp` direkt ausgewertet werden kann.

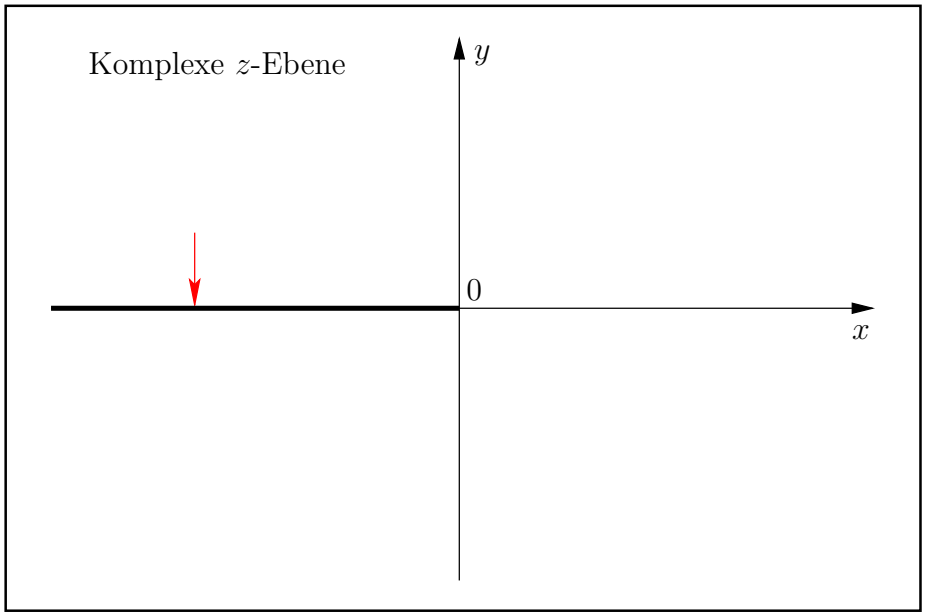


Abbildung B.6: Verzweigungsschnitt von $\log(z)$, $z \in \mathbb{C}$.

Der Pfeil gibt die Richtung an, aus der $\log(z)$ auf den Verzweigungsschnitt analytisch fortgesetzt wird. Die Auswertung von $\log(z)$ erfolgt mit

```
MpfcClass ln(const MpfcClass& z, RoundingMode rnd);
```

aus `mpfcclass.cpp`. In C-XSC wird die Logarithmusfunktion zur Basis e traditionsgemäß mit `ln(...)` bezeichnet.

B.1.9 z^2

Für $z = x + i \cdot y \in \mathbb{C}$ ist z^2 definiert durch

$$z^2 := x^2 - y^2 + 2i \cdot x \cdot y.$$

Der Realteil $x^2 - y^2$ wird direkt mit der vordefinierten Funktion

```
MpfrClass x2my2 (const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

aus `mpfrclass.cpp` ausgewertet. Um beim Imaginärteil einen vorzeitigen Unterlauf bei der Auswertung von $x \cdot y$ zu vermeiden, wird zunächst das betragsmäßige Minumum von x, y mit 2 multipliziert, und erst dann erfolgt die Multiplikation mit dem betragsmäßig größeren zweiten Faktor. Um dies zu testen, wähle man z.B. $x =$ `minfloat()` und $y = 0.75$ und berechne damit den abgerundeten Wert z^2 mit der Funktion

```
MpfcClass sqr (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist. Man erhält für den Imaginärteil von z^2 den positiven Wert $3.573\ldots \cdot 10^{-323228497}$, während schon das Produkt $x \cdot y$ in $2i \cdot (x \cdot y)$ einen Unterlauf verursacht.

B.1.10 $\sqrt{z}$

Mit $z = x + i \cdot y \in \mathbb{C}$ ist der Hauptwert der komplexen Quadratwurzel definiert durch

$$\sqrt{z} := \begin{cases} \sqrt{x} + i \cdot 0, & \text{falls } \ y = 0 \wedge x \geq 0 \\ 0 + i \cdot \sqrt{|x|}, & \text{falls } \ y = 0 \wedge x < 0 \\ \dfrac{\sqrt{2 \cdot (|z| + x)}}{2} + i \cdot \dfrac{y}{\sqrt{2 \cdot (|z| + x)}}, & \text{falls } \ y \neq 0. \end{cases} \tag{B.7}$$

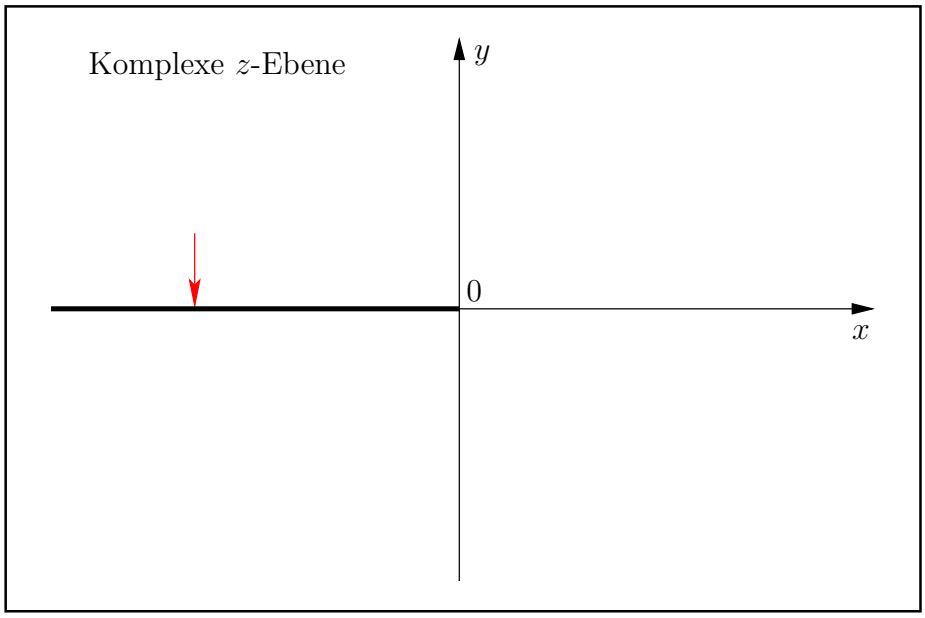


Abbildung B.7: Verzweigungsschnitt von $\sqrt{z}$, $z \in \mathbb{C}$.

Der Pfeil gibt die Richtung an, aus der $\sqrt{z}$ auf den Verzweigungsschnitt analytisch fortgesetzt wird, d.h. z.B. $\sqrt{-1 + 0 \cdot i} = +i$.

Nach (B.7) können im Fall $y \neq 0$ und $x < 0$ bei der Auswertung der gerundeten Wurzelausdrücke $\sqrt{2 \cdot (|z| + x)}$ wegen drohender Auslöschung starke Überschätzungen entstehen. Um diese Überschätzungen zu vermeiden, schreibt man im Fall negativer x-Werte $x = -|x|$, und das Erweitern mit $(|z| + |x|)$ liefert

$$\Re(\sqrt{z}) = \begin{cases} \dfrac{\sqrt{2 \cdot (|z| + x)}}{2}, & \text{falls } \; y \neq 0 \wedge x \geq 0 \\ \dfrac{|y|}{\sqrt{2 \cdot (|z| + |x|)}}, & \text{falls } \; y \neq 0 \wedge x < 0. \end{cases} \tag{B.8}$$

$$\Im(\sqrt{z}) = \begin{cases} \dfrac{y}{\sqrt{2 \cdot (|z| + x)}}, & \text{falls } \; y \neq 0 \wedge x \geq 0 \\ \dfrac{\operatorname{sign}(y)}{2} \cdot \sqrt{2 \cdot (|z| + |x|)}, & \text{falls } \; y \neq 0 \wedge x < 0. \end{cases} \tag{B.9}$$

Die reellen Wurzeln in (B.8) und (B.9) können jetzt ohne Auslöschung mit Hilfe der Funktion

```
MpfrClass Sqrt_zpx(const MpfrClass& x, const MpfrClass& y, const RoundingMode rnd);
```

aus der Datei `mpfcclass.cpp` ausgewertet werden, wobei mit dem Parameter `x` ≥ 0 jetzt keine negativen Werte übergeben werden. Es besteht aber noch ein weiteres Problem, denn mit der Abkürzung $M :=$ `MaxFloat()` gilt für die reelle Wurzel die Abschätzung

$$A := \sqrt{2 \cdot (|z| + x)} \leq \sqrt{2 \cdot (\sqrt{2M^2} + M)} = \sqrt{2(\sqrt{2} + 1) \cdot M} < 3 \cdot \sqrt{M} < M,$$

so dass A im Zahlenformat stets darstellbar ist, während $2 \cdot (|z| + x)$ durchaus einen vorzeitigen Überlauf verursachen kann. Um diesen Überlauf zu vermeiden, wird bei zu großem $|x|$ oder $|y|$ skaliert, d.h. x und y werden bei korrekter Rundung durch 2^4 dividiert, und die ausgewertete Wurzel wird dann am Ende zum Ausgleich wieder mit $2^2 = 4$ rundungsfehlerfrei multipliziert, da wegen der obigen Abschätzung kein Überlauf eintreten kann.

Die Auswertung der gerundeten Werte von $\sqrt{z}$ erfolgt mit Hilfe der Funktion

```
MpfcClass sqrt(const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist.

B.1.11 $\sinh(z)$

Mit $z = x + i \cdot y \in \mathbb{C}$ kann der hyperbolische Sinus wie folgt definiert werden

$$\sinh(z) := \Im(\sin(y + i \cdot x)) + i \cdot \Re(\sin(y + i \cdot x)).$$

Mit der bereits definierten $\sin(z)$-Funktion wird daher $\sinh(z)$ wie folgt implementiert:

```
1  MpfcClass sinh (const MpfcClass& z, RoundingMode rnd)
2  // z = x + i*y;
3  {
4    MpfcClass z_(Im(z), Re(z), rnd, z.GetPrecision());  // z_ = y + i*x; Exakt!
5    z_ = sin(z_, rnd);
6    return MpfcClass( Im(z_), Re(z_) );
7  }
```

Zunächst wird in Zeile 4 das transformierte Argument $z\_ = y + i \cdot x$ mit der gleichen Präzision von z bestimmt. Mit diesem Argument wird in Zeile 5 zuerst der mittels `rnd` gerundete Wert $\sin(z\_)$ berechnet und in die Current-Präzision gerundet. Anschließend erfolgt die Wertübergabe an $z\_$ in der gleichen Current-Präzision. In Zeile 6 wird der gerundete Funktionswert $\sinh(z)$ zurückgegeben.

B.1.12 $\cosh(z)$

Mit $z = x + i \cdot y \in \mathbb{C}$ kann der hyperbolische Cosinus wie folgt definiert werden

$$\cosh(z) := \cos(i \cdot z) = \cos(-y + i \cdot x).$$

Die Auswertung der gerundeten Werte von $\cosh(z)$ erfolgt mit Hilfe der Funktion

```
MpfcClass cosh (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist.

B.1.13 $\tanh(z)$

Mit $z = x + i \cdot y \in \mathbb{C}$ kann der hyperbolische Tangens wie folgt definiert werden

$$\tanh(z) := \Im(\tan(y + i \cdot x)) + i \cdot \Re(\tan(y + i \cdot x)).$$

Die Auswertung der gerundeten Werte von $\tanh(z)$ erfolgt mit Hilfe der Funktion

```
MpfcClass tanh (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist.

B.1.14 $\coth(z)$

Mit $z = x + i \cdot y \in \mathbb{C}$ kann der hyperbolische Cotangens wie folgt definiert werden

$$\coth(z) := -\Im(\cot(y + i \cdot x)) + i \cdot \Re(\cot(y + i \cdot x)).$$

Die Auswertung der gerundeten Werte von $\coth(z)$ erfolgt mit Hilfe der Funktion

```
MpfcClass coth (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist.

B.1.15 arcsin(z)

Die mehrdeutige arcsin-Funktion besitzt auf der reellen Achse die beiden Verzweigungspunkte $(-1,0)$ und $(+1,0)$. Für den Hauptzweig gehen die Verzweigungsschnitte von $-\infty$ bis -1 und von $+1$ bis $+\infty$.

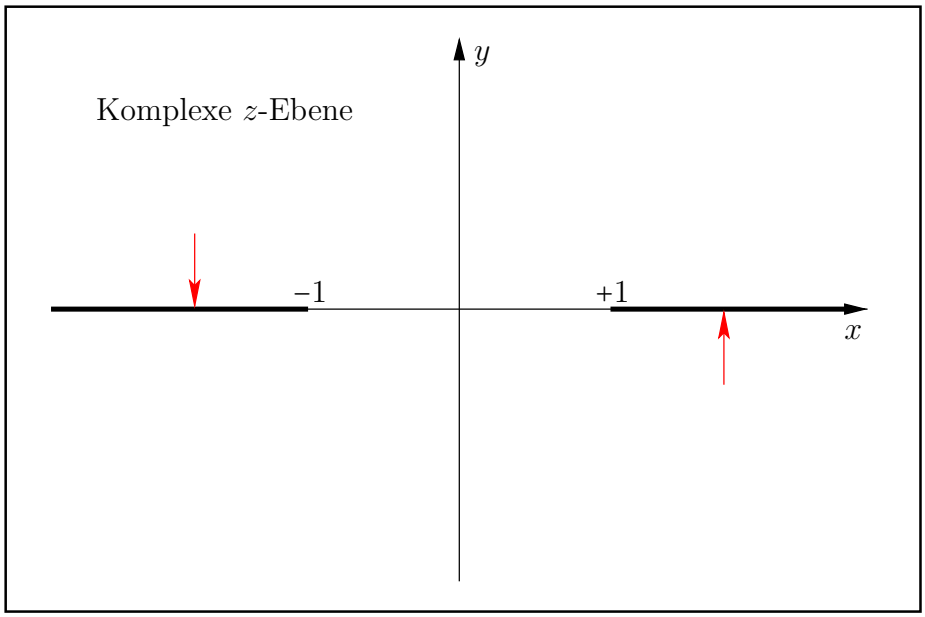


Abbildung B.8: Verzweigungsschnitte von arcsin(z), $z \in \mathbb{C}$.

Die Pfeile geben die jeweilige Richtung an, aus denen die Funktionswerte von arcsin(z) auf den jeweiligen Verzweigungsschnitt analytisch fortgesetzt werden.

B.1.15.1 Realteil

Mit $z = x + i \cdot y \in \mathbb{C}$, $\arcsin(z) = u(x,y) + i \cdot v(x,y)$ und

$$T(x,y) := \frac{1}{2}\left(\sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2}\right) \tag{B.10}$$

gilt z.B. nach [28] für die Realteilfunktion

$$\Re(\arcsin(z)) = u(x,y) = \arcsin\frac{x}{T}. \tag{B.11}$$

Da die reelle arcsin-Funktion rechts in (B.11) streng monoton wächst, müssen für auf- bzw. abgerundete Funktionswerte $u(x,y)$ die Argumente $x/T(x,y)$ selbst auf- bzw. abgerundet werden. Um dies möglichst einfach realisieren zu können, wird $T(x,y)$ zunächst mithilfe der Funktion[1]

```
MpfrClass asin_T(const MpfcClass& z, bool& scal, RoundingMode rnd);
```

so mit einem Rundungsparameter `rnd` implementiert, dass mit ihm auf- bzw. abgerundete Funktionswerte von $T(x,y)$ berechnet werden können. Wegen

$$T(x,y) \equiv T(|x|,|y|) = \frac{1}{2}\left(\sqrt{(|x|+1)^2+|y|^2} + \sqrt{(|x|-1)^2+|y|^2}\right) \geq T(|x|,0), \tag{B.12}$$

$$T(|x|,0) = \frac{1}{2}\{(|x|+1) + ||x|-1|\} = \begin{cases} |x|, & \text{falls } |x| \geq 1, \\ 1, & \text{falls } |x| \leq 1, \end{cases} \tag{B.13}$$

[1] Den Quellcode von `asin_T` findet man in `mpfcclass.cpp`.

folgt direkt $T(x,y) \geq 1$, und wegen $T(x,y) \equiv T(|x|,|y|)$ kann $T(x,y)$ mithilfe der Wurzelsumme in (B.12) ausgewertet werden. Die Voraussetzungen $x \geq 0$ und $y \geq 0$ stellen dabei für die korrekte Rundung von $T(x,y)$ eine große Hilfe dar. Um einen vorzeitigen Überlauf bei der Berechnung der Wurzelsumme in (B.12) zu vermeiden, betrachten wir zunächst mit $M := \texttt{max}(|x|,|y|)$ die folgenden Abschätzungen:

$$\sqrt{(|x|+1)^2+|y|^2} + \sqrt{(|x|-1)^2+|y|^2} \leq 2\sqrt{(|x|+1)^2+|y|^2} \leq 2\sqrt{(M+1)^2+M^2} < 2\sqrt{2}(M+1).$$

Ein Überlauf wird also verhindert durch $2\sqrt{2}(M+1) < \texttt{MaxFloat()}$ bzw. durch die Forderung

$$\texttt{expo}(M) < \texttt{expo}\left(\frac{\texttt{MaxFloat()}}{2\sqrt{2}} - 1\right) = 1073741821 =: \texttt{p}.$$

Der opige minimale Wert $\texttt{p} = 1073741821$ wurde mit der kleinst-möglichen Current-Precision $\texttt{prec} = 2$ berechnet. Schon mit $\texttt{prec} \geq 4$ erhält man $\texttt{p}_g = 1073741822$. Um für alle Präzisionen einen Überlauf zu vermeiden, wird daher im Fall $\texttt{expo}(M) \geq \texttt{p}$ geeignet skaliert, d.h. die Werte $|y|, (|x|+1), (|x|-1)$ werden durch 8 dividiert, wobei 1/8 selbst bei $\texttt{prec} = 2$ exakt gespeichert wird. Im Fall `scal == true` muss daher der Rückgabewert von `asin_T(...)` noch mit 8 multipliziert werden, um den korrekten Wert von $T(x,y)$ nach (B.12) zu erhalten. In (B.12) erfolgt die Auswertung der beiden Wurzeln mit den quadratischen Argumenten mithilfe der bereits implementierten Funktion `sqrtx2y2(...)`.

Die Auswertung der Realteilfunktion $u(x,y)$ erfolgt mithilfe von `asin_T(...)` in der Funktion

```
MpfrClass Re_asin (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist. Im Fall $y = 0$ erhält man mit (B.11) und (B.13)

$$u(x,0) = \begin{cases} \arcsin(x), & |x| \leq 1, \\ \arcsin(+1) = +\pi/2, & x \geq +1, \\ \arcsin(-1) = -\pi/2, & x \leq -1. \end{cases}$$

Im Fall $y = 0$ kann $u(x,0)$ damit direkt, d.h. ohne die Funktion $T(x,y)$, ausgewertet werden. Im Fall $y \neq 0$ wird in (B.11) zunächst der Quotient $x/T(x,y)$ berechnet, wobei auf die korrekten Rundungen zu achten ist. Wenn bei der Auswertung von $T(x,y)$ skaliert worden ist, so wird T jetzt nicht mit 8 multipliziert, weil dies zum Überlauf führen würde, sondern der Zähler x ist dafür durch 8 zu dividieren.

Mit (B.12) und (B.13) folgt für den Quotienten $|x|/T(x,y)$ die Abschätzung

$$\frac{|x|}{T(x,y)} \leq \frac{|x|}{T(x,0)} = \begin{cases} 1, & \text{falls} |x| \geq 1, \\ |x|, & \text{falls} |x| < 1, \end{cases} \qquad \Longrightarrow \qquad \frac{|x|}{T(x,y)} \leq 1,$$

die natürlich erfüllt sein muss, wenn die reelle arcsin-Funktion in (B.11) mit diesem Quotienten ohne Fehlermeldung ausgewertet werden soll. Die gewünschten Rundungen bei der Auswertung des Quotienten können aber auf der Maschine $|x|/T > 1$ zur Folge haben. Um die damit verbundenen Fehlermeldungen zu vermeiden, müssen daher vor Auswertung der reellen arcsin-Funktion die folgenden Abfragen erfolgen. Zur Abkürzung wird dabei der auf der Maschine berechnete und gerundete Quotient jetzt mit `x` bezeichnet.

```
if (x>1)
  x = 1;
if (x<-1)
  x = -1;
```

Alle weiteren Einzelheiten findet man im Quellcode der Funktion `Re_asin(...)` in der Datei `mpfcclass.cpp`.

B.1.15.2 Imaginärteil

Mit der in (B.10) bereits definierten Funktion $T(x,y)$ gilt nach [28] für die Imaginärteilfunktion

$$v(x,y) := \begin{cases} +\log(T+\sqrt{T^2-1}), & \text{falls } y>0 \\ +\log(T+\sqrt{T^2-1}), & \text{falls } y=0 \text{ und } x\le -1 \\ 0 \quad , & \text{falls } y=0 \text{ und } |x|\le +1 \\ -\log(T+\sqrt{T^2-1}), & \text{falls } y=0 \text{ und } x\ge +1 \\ -\log(T+\sqrt{T^2-1}), & \text{falls } y<0. \end{cases} \tag{B.14}$$

Wegen $T(x,y)\ge 1$ gilt zusätzlich

$$\operatorname{arcosh}(T) \equiv \log(T+\sqrt{T^2-1}), \tag{B.15}$$

so dass in (B.14) die Logarithmus-Funktionen mit dem komplizierten Argument $T+\sqrt{T^2-1}$ im Bedarfsfall durch $\operatorname{arcosh}(T)$ ersetzt werden können. Bei der Auswertung von $\operatorname{arcosh}(T)$ treten jedoch zwei grundsätzliche Probleme auf:

1. Bei der Auswertung von $T(x,y)$ mithilfe der Funktion `asin_T(...)` wird bei zu großem $|x|$ oder $|y|$ geeignet skaliert, so dass der Rückgabewert T noch mit 8 zu multiplizieren ist. In diesem Fall wäre $\operatorname{arcosh}(8\cdot T)$ auszuwerten, wodurch ebenfalls ein Überlauf entstehen würde. In diesem Fall muss man auf die rechte Seite von (B.15) zurückgreifen und den komplizierteren Ausdruck $\log(8\cdot T+\sqrt{64\cdot T^2-1})$ weiter umformen, um den z.B. durch $8\cdot T$ drohenden Überlauf zu vermeiden.

2. Im Fall $T(x,y)\to +1$ werden beide Funktionen in (B.15) in der Nähe ihrer gemeinsamen Nullstellen $T_0(x,y)=1$ ausgewertet, wobei wegen starker Auslöschung bei den Rundungen zu große Überschätzungen entstehen. Diese Überschätzungen werden vermieden, wenn man in $\operatorname{arcosh}(T)$ das Argument $T=1+r$ in zwei Summanden zerlegt und $r:=T-1$ so umformt, dass bei seiner Auswertung nur minimale Überschätzungen entstehen können. Damit folgt

   $$\operatorname{arcosh}(T) = \operatorname{arcosh}(1+r) = \texttt{acoshp1}(r),$$

   wobei die letzte Funktion `acoshp1`(r) bereits implementiert ist, vgl. Tab. 3.1 auf Seite 35.

Wir behandeln zunächst das **Problem 1.**

Im Skalierungsfall gilt mit $T=8\cdot T$

$$\begin{aligned} \log(8\cdot T+\sqrt{64\cdot T^2-1}) &= \log(8\cdot T+\sqrt{64\cdot (T^2-1/64)} \\ &= \log(8\cdot T+8\cdot\sqrt{(T^2-1/64)} \\ &= \log(8)+\log(T+\sqrt{(T^2-1/64)}) \\ &= 3\cdot\log(2)+\log(T+\sqrt{T-1/8}\cdot\sqrt{T+1/8}). \end{aligned}$$

Jetzt kann die ganze letzte rechte Seite ohne Überlauf ausgewertet werden, wobei $\log(2)$ als Konstante zur Verfügung steht. T ist der mit Skalierung berechnete Rückgabewert der Funktion `asin_T(...)`, und 1/8 kann auch noch bei der minimalen Current-Präzision `prec` = 2 exakt gespeichert werden. Die Auswertung des letzten Ausdrucks oben rechts erfolgt mit der Funktion

```
MpfrClass acosh_T(const MpfcClass& z, RoundingMode rnd);
```

die in der Datei `mpfcclass.cpp` definiert ist.

Wir betrachten jetzt das **Problem 2.**

Der geometrische Ort aller Punkte mit $T_0(x,y) = 1$ ist auf der x-Achse das Intervall $[-1,+1]$. In seiner Umgebung

$$U := \{(x,y) \in \mathbb{R}^2 \,\big|\, |x| < 1.125 \wedge |y| < 0.125\}$$

wird T zerlegt in $T = 1 + r$, womit die Beziehung $\operatorname{arcosh}(T) = \operatorname{arcosh}(1+r) = \texttt{acoshp1}(r)$ zur Anwendung kommt. Die Schranken 1.125 und 0.125 wurden so gewählt, dass mit `prec` = 53 beim Überschreiten der Umgebungsgrenze die Genauigkeit höchstens um eine Dezimalstelle abnimmt. Die nachfolgenden Rechnungen zeigen, dass die Auswertung innerhalb der Umgebung U deutlich aufwendiger sind als außerhalb. Die gewählten Schranken sollten daher möglichst klein sein, so dass die Wahl der obigen Werte einen guten Kompromiss darstellt. Bei größeren Präzisionen `prec` > 53 wird beim Überschreiten der Umgebungsgrenze die Genauigkeit ebenfalls höchstens nur um eine Dezimalstelle kleiner.

Mit $T = 1 + r,\ r \geq 0$ gilt

$$\begin{aligned}
2r &= 2T(x,y) - 2 \\
&= \sqrt{(|x|+1)^2 + y^2} + \sqrt{(|x|-1)^2 + y^2} - 2 \\
&= (|x|+1)\sqrt{1 + \left(\frac{y}{|x|+1}\right)^2} - 2 + \sqrt{(|x|-1)^2 + y^2} \\
&= (|x|+1)\left\{\sqrt{1 + \left(\frac{y}{|x|+1}\right)^2} - 1 + 1\right\} - 2 + \sqrt{(|x|-1)^2 + y^2} \\
&= (|x|+1)\left\{\sqrt{1 + (\,)^2} - 1\right\} + (|x|-1) + \sqrt{(|x|-1)^2 + y^2} \\
&= (|x|+1)\cdot g\left(\left(\frac{|y|}{|x|+1}\right)^2\right) + B, \quad g(t) := \sqrt{1+t} - 1, \\
&= A + B.
\end{aligned}$$

Der Ausdruck A wird mithilfe der Funktion `asin_rA(...)` ausgewertet, wobei die Funktion $g(t)$ mithilfe der bereits imlementierten, monoton wachsenden Funktion `sqrtp1m1(...)` direkt ausgewertet werden kann, vgl. Tab. 3.1 auf Seite 35. Das Argument $(|y|/(|x|+1))^2$ kann jetzt mit nur ganz minimalen Überschätzungen auf- bzw. abgerundet werden. Der Ausdruck B wird am Anfang der Funktion `acoshp1_r(...)` wie folgt ausgewertet:

$$B = \begin{cases} |y|, & \text{falls} \quad |x| = 1 \\ (|x|-1) + \sqrt{(|x|-1)^2 + y^2}, & \text{falls} \quad |x| > 1 \\ (1-|x|)\left\{\sqrt{1 + \left(\frac{|y|}{1-|x|}\right)^2} - 1\right\}, & \text{falls} \quad |x| < 1, \end{cases}$$

wobei auch jetzt der Ausdruck {...} mithilfe der Funktion `sqrtp1m1(...)` berechnet wird. Es ist zu beachte, dass in den Fällen $|x| > 1$ und $|x| < 1$ die entsprechenden Terme ohne Auslöschung berechnet werden. Lediglich im letzten Fall $|x| < 1$ könnte man annehmen, dass für $|x| \to +1$ das Argument $(|y|/(1-|x|))^2$ einen vorzeitigen Überlauf erzeugt. Erfreulicherweise ist diese Gefahr jedoch rein theoretisch, da sie nur eintritt, wenn die Current-Präzision in die Größenordnung von ca. `pred` = 500 000 000 Bits kommt, was schon aus Laufzeitgründen völlig unrealistisch ist!

B.1.16 $\arccos(z)$

Die mehrdeutige arccos-Funktion besitzt auf der reellen Achse die beiden Verzweigungspunkte $(-1,0)$ und $(+1,0)$. Für den Hauptzweig gehen die Verzweigungsschnitte wie bei der arcsin-Funktion von $-\infty$ bis -1 und von $+1$ bis $+\infty$, vgl. Abb. B.8 auf Seite 147. Für den Hauptwert gelten mit $T(x,y)$ von Seite 147 die Beziehungen

$$\Re(\arccos(z)) = \arccos(x/T), \tag{B.16}$$

$$\Im(\arccos(z)) = -\Im(\arcsin(z)). \tag{B.17}$$

Der Realteil von $\arccos(z)$ wird im Vergleich zu (B.11) ganz analog zum Realteil von $\arcsin(z)$ berechnet. Im Gegensatz zur reellen arcsin-Funktion ist die reelle arccos-Funktion jedoch monoton fallend, so dass für die korrekten Rundungen das Argument x/T jetzt im Vergleich zur arcsin-Funktion genau entgegengesetzt zu runden ist. Weitere Einzelheiten findet man in der Funktion `Re_acos(...)`, die in der Datei `mpfcclass.cpp` definiert ist.

Nach (B.17) unterscheidet sich der Imaginärteil von $\arccos(z)$ vom Imaginärteil der arcsin-Funktion nur durch das Vorzeichen, so dass für die korrekten Rundungen nach (A.3) und (A.4) von Seite 122 die Rundungsparameter bei den entsprechenden Funktionsaufrufen `Im_asin(...)` nur zu vertauschen sind. Weitere Einzelheiten findet man in der Funktion `Im_acos(...)`, die in der Datei `mpfcclass.cpp` definiert ist.

B.1.17 $\log(1+z)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist $\log(1+z)$ definiert durch

$$\log(1+z) := \log\left(\sqrt{(1+x)^2+y^2}\right) + i \cdot \arg(1+z),$$

wobei der Realteil $\log\left(\sqrt{(1+x)^2+y^2}\right)$ mit der vordefinierten Funktion

```
MpfrClass ln_sqrtxp1_2y2(const MpfrClass& x, const MpfrClass& y,
                                             RoundingMode rnd);
```

aus `mpfrclass.cpp` direkt ausgewertet werden kann.

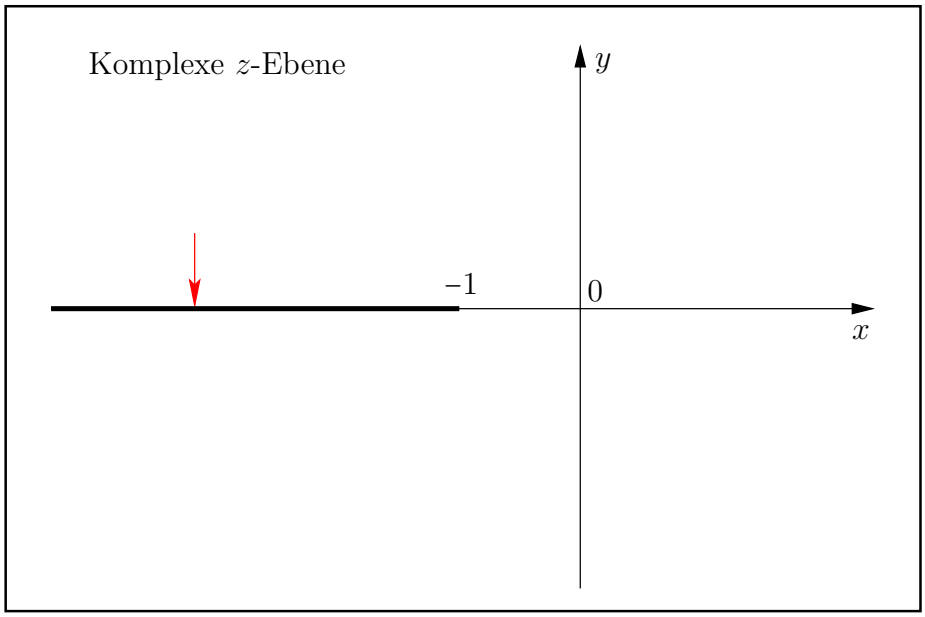


Abbildung B.9: Verzweigungsschnitt von $\log(1+z)$, $z \in \mathbb{C}$.

Der Pfeil gibt die Richtung an, aus der $\log(1+z)$ auf den Verzweigungsschnitt analytisch fortgesetzt wird. Die Auswertung von $\log(1+z)$ erfolgt mit der Funktion

```
MpfcClass lnp1(const MpfcClass& z, RoundingMode rnd);
```

aus `mpfcclass.cpp`. In C-XSC wird die Logarithmusfunktion zur Basis e traditionsgemäß mit `ln(...)` bezeichnet.

Mithilfe des Rundungsmodus `rnd` kann im Vergleich zum exakten Funktionswert auf- oder abgerundet bzw. in Richtung des exakten Funktionswerts gerundet werden. Diese Rundungen werden jedoch nicht immer optimal ausgeführt, d.h. der z.B. aufgerundete Wert ist nicht immer die erste Maschinenzahl rechts vom exakten Funktionswert[2]. Es wird jedoch garantiert, dass der zurückgegebene Wert rechts vom exakten Funktionswert liegt. Ganz analog wird mit `rnd = RoundNearest` nur garantiert, dass der Rückgabewert in der unmittelbaren Nähe des exakten Funktionswertes liegt. Beachten Sie, dass bei einer **optimalen** Rundung mit z.B. `rnd = RoundNearest` der Rückgabewert die zum exakten Funktionswert nächstgelegene Maschinenzahl ist. Beachten Sie außerdem, dass bei den beiden oberen Funktionen der Rundungsmodus `rnd` nicht gesetzt werden muss. In diesem Fall wird bez. des voreingestellten Current-Rundungsmodus gerundet.

[2]Bei diesem Beispiel wird angenommen, dass der exakte Funktionswert keine Maschinenzahl ist.

B.1.17.1 Realteilfunktion

Die auszuwertende Realteilfunktion lautet

$$u(x,y) := \frac{1}{2}\cdot\ln\big((1+x)^2+y^2\big) = \ln\big(\sqrt{(1+x)^2+y^2}\big); \tag{B.18}$$

Zur Vermeidung eines vorzeitigen Überlaufs bei der Auswertung von $\beta := (1+x)^2+y^2$ berechnen wir die folgende Oberschranke

$$(1+x)^2+y^2 \le (1+|x|)^2+|y|^2 < (1+|x|)^2+(1+|y|)^2.$$

Mit $M :=$ `MaxFloat(prec)` wird ein vorzeitiger Überlauf vermieden, wenn die beiden folgenden Bedingungen erfüllt sind:

$$\begin{gathered}(1+|x|)^2 < M/2 \ \wedge\ (1+|y|)^2 < M/2 \quad\Longleftrightarrow\\ |x| < \sqrt{M/2}-1 \ \wedge\ |y| < \sqrt{M/2}-1.\end{gathered}$$

Wegen

$$2^{\mathtt{k1}} < \mathtt{pred}(\sqrt{M/2}) \le \sqrt{M/2}-1, \quad k1 = 536870909 \tag{B.19}$$

wird damit ein vorzeitiger Überlauf vermieden, wenn gilt

$$\mathtt{expo}(x) < \mathtt{k1} \ \wedge\ \mathtt{expo}(y) < \mathtt{k1}, \quad \mathtt{k1} = 536870909. \tag{B.20}$$

Zu beachten ist, dass in (B.19) M eine von `prec` abhängige Größe ist und dass $2^{\mathtt{k1}}$ eine Unterschranke für alle `prec` ≥ 2 ist. Nach (B.20) kann für alle `prec` ≥ 2 ein Überlauf nur auftreten, wenn gilt:

$$\mathtt{expo}(x) \ge \mathtt{k1} \ \text{ oder } \ \mathtt{expo}(y) \ge \mathtt{k1}, \quad \mathtt{k1} = 536870909. \tag{B.21}$$

Wir betrachten jetzt den Fall, dass mit (B.21) ein Überlauf bei der Auswertung von $(1+x)^2+y^2$ eintreten kann. Zur Vermeidung eines solchen Überlaufs betrachten wir die Umformung

$$\begin{aligned}\ln\big[(1+x)^2+y^2\big] &= \ln\big[2^{2k}\cdot 2^{-2k}\big((1+x)^2+y^2\big)\big], \quad k\in\mathbb{N}\\ &= 2k\cdot\ln(2)+\ln\big[(2^{-k}+x\cdot 2^{-k})^2+(2^{-k}\cdot y)^2\big].\end{aligned}$$

Für das Argument der letzten ln-Funktion gilt die Abschätzung

$$\alpha := (2^{-k}+x\cdot 2^{-k})^2+(2^{-k}\cdot y)^2 < (1+|x|\cdot 2^{-k})^2+(1+|y|\cdot 2^{-k})^2.$$

Wählt man jetzt nach Seite 15 für x,y die für alle `prec` ≥ 2 gültige Oberschrane der größten positiven Zahl `MaxFloat(prec)` $< 2^{1073741824}$, so folgt

$$\alpha < 2\cdot(1+2^{1073741824-k})^2.$$

Ein vorzeitiger Überlauf wird also vermieden, wenn man wieder nach Seite 15 verlangt

$$2\cdot(1+2^{1073741824-k})^2 < 2^{1073741823} \quad\Longleftrightarrow\quad 1+2^{1073741824-k} < 2^{536870911}, \tag{B.22}$$

und wegen $1+2^{1073741824-k} < 2^{1073741825-k}$ ist die letzte Ungleichung in (B.22) erfüllt, wenn gilt

$$2^{1073741825-k} < 2^{536870911} \quad\Longleftrightarrow\quad k > 536870914.$$

Zusammenfassung:
Im Fall (B.21) wird ein Überlauf bei der Auswertung von β vermieden, wenn $u(x,y)$ mithilfe der Konstanten `Ln2(rnd)` wie folgt berechnet wird, vgl. Seite 33:

$$u(x,y) = k\cdot\ln(2)+\frac{1}{2}\cdot\ln\big[(2^{-k}+2^{-k}\cdot x)^2+(2^{-k}\cdot y)^2\big], \quad k = 536870915. \tag{B.23}$$

Wir betrachten jetzt den Fall, dass bei der Berechmumg des ln-Arguments $\beta := (1+x)^2 + y^2$ **kein Überlauf** eintreten kann. Bei der Auswertung von $\ln(\beta)$ sind dann aber in Verbindung mit Abb. B.10 noch folgende Punkte zu beachten:

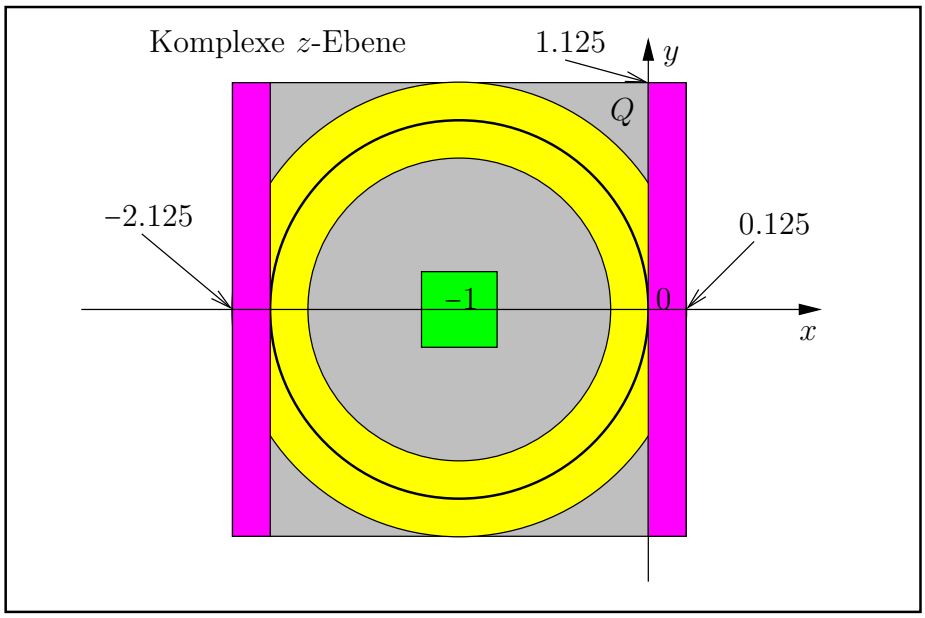


Abbildung B.10: Verschiedene Bereiche zur Auswertung von $u(x,y)$.

1. Im grünen Quadrat mit dem Mittelpunkt $M = (-1,0)$ und der Kantenlänge $2 \cdot 0.125 = 0.25$ gilt mit $x = -1 \pm \varepsilon,\ 0 \le \varepsilon \le 0.125$, die Beziehung $\ln(\beta) = \ln(\varepsilon^2 + y^2)$, so dass für $\varepsilon, y \to 0$ die Logarithmusfunktion in der Nähe des Ursprungs mit dort nahezu vertikalen Tangenten auszuwerten ist. Um die dabei auftretenden großen Überschätzungen zu vermeiden, wird die Current-Präzision schrittweise verdoppelt, bis das Argument $\varepsilon^2 + y^2$ exakt berechnet werden konnte. Die Auswertung der ln-Funktion erfolgt dann mit dem exakten Argument in der vergrößerten Präzision.

2. Die in der Nähe der Nullstelle $\beta = 1$ auftretende Auslöschung lässt sich vermeidet, wenn man $\ln(\beta)$ wie folgt umformt $\ln(\beta) = \ln(1 + \{x(2+x) + y^2\}) = \ln(1+\delta)$ und $\ln(1+\delta)$ mit Hilfe der bereits implementierten Funktion `lnp1`(δ) auswertet. Die Umgebung von $\beta(x,y) = 1$ ist in Abb. B.10 der durch die beiden roten Rechtecke teilweise überdeckte gelbe Kreisring, der durch das schraffierte Quadrat Q mit dem Mittelpunkt $M = (-1,0)$ und der Kantenlänge 2.25 eingeschlossen wird. Die Punktmenge $\beta(x,y) = 1$ ist der Kreis um M mit dem Radius 1 und berührt die y-Achse im Ursprung.

3. In der Teilmenge $-2 \le x < 0 \wedge |y| < 1.125$ von Q besteht wegen $x \cdot (2+x) < 0$ bei der Auswertung von $\delta = x \cdot (2+x) + y^2$ die Gefahr starker Auslöschung, die auch jetzt wieder durch schrittweise Verdoppelung der Current-Präzision bis zur exakten Berechnung von δ vermieden wird.

4. In den beiden roten Teilmengen $0 < x < 0.125 \wedge |y| < 1.125$ bzw. $-2.125 < x < -2 \wedge |y| < 1.125$ von Q kann bei der Auswertung von $\delta = x \cdot (2+x) + y^2$ keine Auslöschung auftreten, so dass `lnp1`(δ) direkt ausgewertet werden kann. In beiden Teilmengen gilt: $x \cdot (2+x) > 0$.

5. Außerhalb des Quadrates Q und außerhalb des Überlaufbereiches wird $\ln\{(1+x)^2 + y^2\}$ direkt ausgewertet. Weitere Einzelheiten findet man in der Datei `mpfrclass.cpp`.

B.2 Elementarfunktionen für komplexe Intervallargumente

B.2.1 z^2

Für $z = x + i \cdot y \in \mathbb{C}$ ist z^2 definiert durch

$$z^2 := x^2 - y^2 + 2i \cdot x \cdot y \equiv |x|^2 - |y|^2 + 2i \cdot x \cdot y.$$

Ist z jetzt ein Element aus einem komplexen Intervall $Z = X + iY$ mit reellen Intervallen X, Y, so ist der Realteil von Z^2 wie folgt zu berechnen:

$$\Re(Z^2) := \{(|x|^2 - |y|^2) \,\big|\, |x| \in |X| \wedge |y| \in |Y|\} = [\texttt{Inf}(|X|)^2 - \texttt{Sup}(|Y|)^2,\ \texttt{Sup}(|X|)^2 - \texttt{Inf}(|Y|)^2],$$

wobei z.B. $\texttt{Inf}(|X|)^2 - \texttt{Sup}(|Y|)^2$ mithilfe der vordefinierten Funktion

```
MpfrClass x2my2 (const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

mit `rnd = RoundDown` auszuwerten ist.

Der Imaginärteil von Z^2 ist wie folgt zu berechnen:

$$\Im(Z^2) := \{2 \cdot x \cdot y \,\big|\, x \in X \wedge y \in Y\} = 2 \cdot (X \cdot Y).$$

Um bei der Auswertung von $(X \cdot Y)$ einen vorzeitigen Unterlauf zu vermeiden, berechnet man zuerst $2 \cdot X$ bzw. $2 \cdot Y$, falls $2 \cdot X$ zum Überlauf führt, und multipliziert dann mit dem Intervall Y bzw. mit X. Weitere Einzelheiten findet man in der Quelltext-Datei `mpfciclass.cpp`.

B.2.2 Logarithmusfunktionen

Der Hauptwert der komplexen Logarithmusfunktion zur Basisi e ist für $z = x + i \cdot y \in \mathbb{C}$ definniert durch

$$\log(z) := \ln(|z|) + i \cdot \arg(z), \quad |z| := \sqrt{x^2 + y^2}, \quad -\pi < \arg(z) \leq +\pi.$$

Der Verzweigungsschnitt ist wie üblich die negative reelle Achse, wobei der Ursprung $(0, 0)$ selbst der Verzweigungspunkt ist. $\ln(|z|)$ bedeutet dabei den reellen Logarithmus zur Basis e mit dem Argument $|z|$. Ist nun Z ein komplexes Rechteckintervall, das die Null nicht enthält[3] und die negative reelle Achse nur von oben berühren darf, so enthält die komplexe Zahlenmenge

$$\log(Z) := \{y \in \mathbb{C} \,|\, y = \log(z) \wedge z \in Z\}$$

nur Funktionswerte des Hauptwertes der Logarithmusfunktion. Beachten Sie, dass $\log(Z)$ i.a. kein Rechteckintervall ist. Beispielsweise ist das Bild $\log(Z)$ der Parallelen $Z = [1, 1] + i \cdot [1, 2]$ zur y-Achse in der komplexen Ebenen eine glatte Kurve, deren Einschließung durch ein achsenparalleles Rechteck mit der üblichen Überschätzung verbunden ist, vgl. dazu auch Seite 135.

Bezeichnen wir mit $\mathrm{Ln}(Z)$ die Einschließung von $\log(Z)$ durch ein achsenparalleles Rechteck, so enthält auch dieses Rechteck nur Funktionswerte des Hauptwertes $\log(z)$ und es gilt

$$\log(Z) := \{y \in \mathbb{C} \,|\, y = \log(z) \wedge z \in Z\} \subseteq \mathrm{Ln}(Z).$$

Wir bezeichnen daher $\mathrm{Ln}(Z)$ als **Analytische Logarithmusfunktion**, wobei das Rechteckintervall Z die Null nicht enthalten darf und die negative reelle Achse nur von oben berühren darf. Man sollte noch beachten, dass z.B. im Fall $Z = [-2, -1] + i \cdot [0, 1]$ die Einschließung

$$\mathrm{Ln}(Z) = \texttt{([0,8.04719e-1], [2.35619,3.14160])}$$

mit dem Supremum ihres Imaginärteils einen Wert liert, der nicht mehr zum Hauptwert des Logarithmus gehört, da das maximale Argument π auf der Maschine durch einen etwas gößeren Zahlenwert eingeschlossen werden muss!

Durch analytische Fortsetzung können Funktionswerte der Logarithmusfunktion auch für Rechteckintervalle Z definiert werden, die den Verzweigungsschnitt in ihrem Innern enthalten, wobei aber die Null weder in Z noch auf seinem Rand liegen darf. Durch analytische Fortsetzung ist es jetzt auch möglich, dass Z die negative reelle Achse von unten berührt, wobei dann aber die Funktionswerte auf dem Verzweigungsschnitt den Imaginärteil $-\pi$ erhalten. Bezeichnen wir mit $\log(Z)$ auch jetzt wieder die Menge aller dieser Funktionswerte, mit $z \in Z$, die also auch durch analytische Fortsetzung definiert sein können und wird deren Einschließung durch Rechteckintervalle mit $\ln(Z)$ bezeichnet, so gilt

$$\log(Z) \subseteq \ln(Z), \; z \in Z.$$

Da $\ln(Z)$, je nach Lage von Z, Funktionswerte einschließen kann, die nicht zum Hauptwert der Logarithmusfunktion gehören, wird $\ln(Z)$ auch als **Nicht-analytische Logarithmusfunktion** bezeichnet. Weitere Einzelheiten zu den erlaubten Intervallen Z findet man auf Seite 158.

[3] Die Null darf auch nicht auf dem Rand von Z liegen!

B.2.2.1 Analytische Logarithmusfunktion

Bedeutet $\log(z)$ den auf Seite 156 definierten Hauptwert der komplexen Logarithmusfunktion und ist Z ein achsenparalleles Rechteckintervall, wobei die Null nicht in seinem Innern oder auf dem Rand von Z liegen darf und Z die negative reelle Achse auch nur von oben berühren darf, so liefert die analytische Logarithmusfunktion $\mathrm{Ln}(Z)$ eine nahezu optimale achsenparallele Rechteck-Einschließung der komplexen Zahlenmenge

$$\log(Z) := \{y \in \mathbb{C} \,|\, y = \log(z) \wedge z \in Z\} \subseteq \mathrm{Ln}(Z).$$

In Abbildung B.11 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z angegeben.

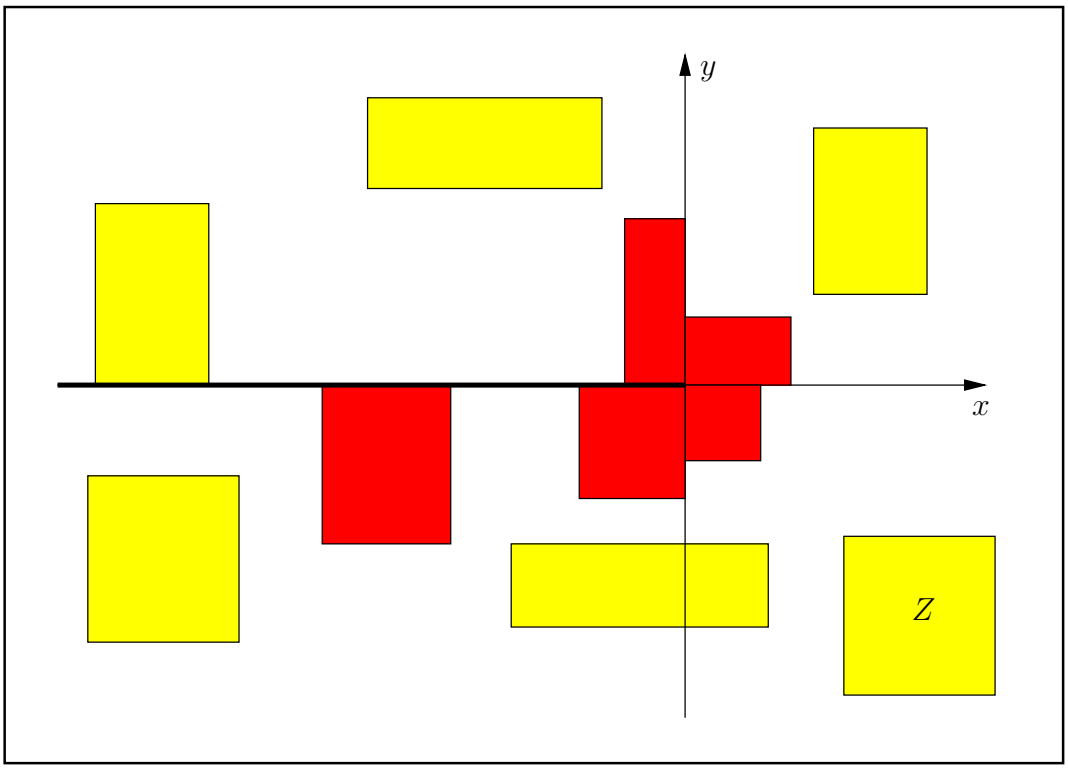


Abbildung B.11: Erlaubte und nicht erlaubte Intervalle Z von Ln(Z).

Hier einige numerische Beispiele in der niedrigen Präzision `prec` = 30, womit etwa $30/\log_2(10) \approx 30/3.321928095 \approx 9$ Dezimalstellen berechnet werden:

$\mathrm{Ln}([+2,+4] + i\cdot[+0,+0]) = ([6.93147180e-1, 1.38629437], [0.00000000, 0.00000000])$,
$\mathrm{Ln}([-4,-2] + i\cdot[+0,+0]) = ([6.93147180e-1, 1.38629437], [3.14159265, 3.14159266])$,
$\mathrm{Ln}([-2,-1] + i\cdot[+0,+1]) = ([0.00000000, 8.04718957e-1], [2.35619449, 3.14159266])$,
$\mathrm{Ln}([-2,+1] + i\cdot[-2,-1]) = ([0.00000000, 1.03972078], [-2.67794505, -7.85398163e-1])$,
$\mathrm{Ln}([+2,+3] + i\cdot[-3,-2]) = ([1.03972077, 1.44518588], [-9.82793724e-1, -5.88002603e-1])$,
$\mathrm{Ln}([+2,+3] + i\cdot[+2,+3]) = ([1.03972077, 1.44518588], [5.88002603e-1, 9.82793724e-1])$,
$\mathrm{Ln}([-2,+2] + i\cdot[+2,+3]) = ([6.93147180e-1, 1.28247468], [7.85398163e-1, 2.35619450])$,
$\mathrm{Ln}([-2,+2] + i\cdot[+0,+1])$ ⇝ `MpfciClass LN( const MpfciClass& z ); z contains 0`.

B.2.2.2 Nicht-analytische Logarithmusfunktion

Ausgangspunkt ist zunächst wieder der auf Seite 156 definierte Hauptwert $\log(z)$ der komplexen Logarithmusfunktion. Ist Z dann ein achsenparalleles komplexes Rechteckintervall, das die Null nicht enthält und die negative reelle Achse nur von oben berühren darf, so enthält die komplexe Zahlenmenge

$$\log(Z) := \{y \in \mathbb{C} \,\big|\, y = \log(z) \wedge z \in Z\}$$

nur Funktionswerte des Hauptwertes der Logarithmusfunktion. Zusätzlich wird aber jetzt noch vorausgesetzt, dass Z den Verzweigungsschnitt in seinem Innern enthalten darf, wobei die Funktionswerte in der unteren Halbebene durch analytische Fortsetzung aus der oberen Halbebene definiert werden. Ist dann $\ln(Z)$ eine Rechteck-Einschließung dieser Zahlenmenge $\log(Z)$, so gilt

$$\log(Z) \subseteq \ln(Z), \quad z \in Z.$$

$\ln(Z)$ wird dabei als nicht-analytische Logarithmusfunktion bezeichnet, da jetzt auch Funktionswerte eingeschlossen werden können, die nicht zum Hauptwert der Logarithmusfunktion gehören. In Abbildung B.12 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $\ln(Z)$ angegeben.

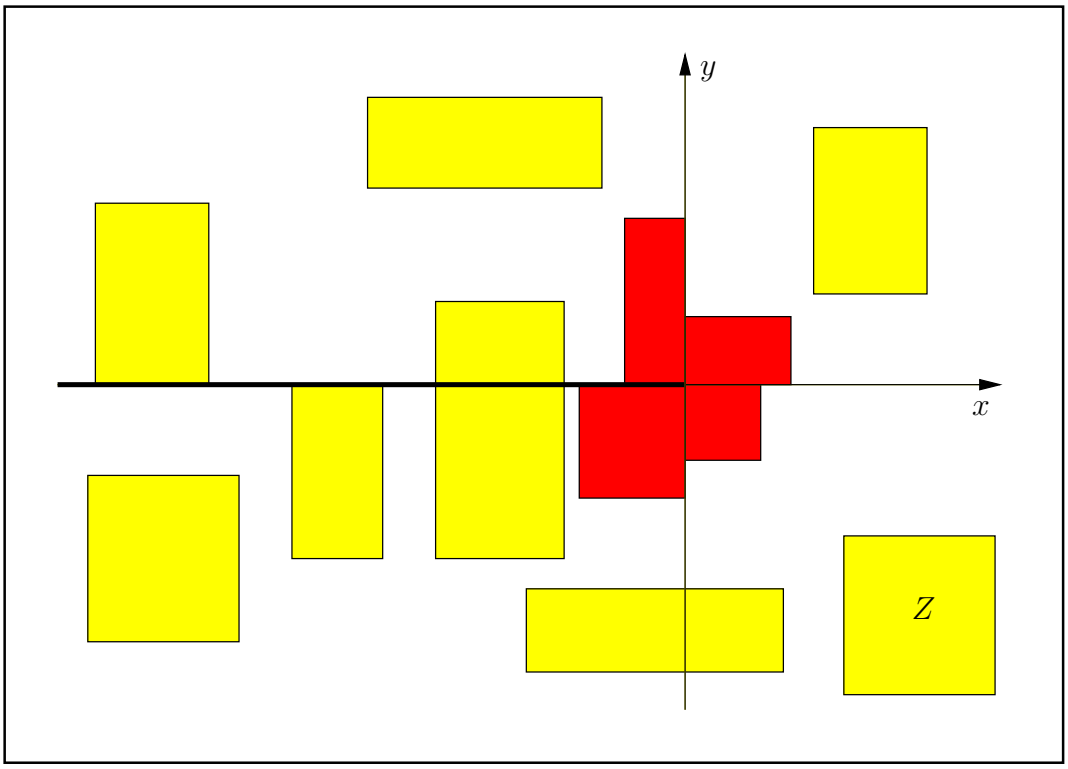


Abbildung B.12: Erlaubte und nicht erlaubte Intervalle Z von ln(Z).

Hier einige numerische Beispiele in der niedrigen Präzision `prec` = 30, womit etwa $30/\log_2(10) \approx 30/3.321928095 \approx 9$ Dezimalstellen berechnet werden:

$\texttt{ln}([+2,+4] + i \cdot [+0,+0]) = ([6.93147180e-1, 1.38629437], [0.00000000, 0.00000000])$,
$\texttt{ln}([-4,-2] + i \cdot [+0,+0]) = ([6.93147180e-1, 1.38629437], [3.14159265, 3.14159266])$,
$\texttt{ln}([-4,-2] + i \cdot [+0,+1]) = ([6.93147180e-1, 1.41660668], [2.67794504, 3.14159266])$,
$\texttt{ln}([-4,-2] + i \cdot [-1,+0]) = ([6.93147180e-1, 1.41660668], [-3.14159266, -2.67794504])$,
$\texttt{ln}([-4,-2] + i \cdot [-1,+1]) = ([6.93147180e-1, 1.41660668], [2.67794504, 3.60524027])$,
$\texttt{ln}([-2,+2] + i \cdot [-3,-2]) = ([6.93147180e-1, 1.28247468], [-2.35619450, -7.85398163e-1])$.

B.2.3 $\sqrt{z}$

Der Verzweigungsschnitt der komplexen Quadratwurzel ist in der komplexen Ebene wie üblich die negative reelle Achse. Ein achsenparalleles Argumentintervall Z darf diesen Verzweigungsschnitt nicht im Innern enthalten und auch nicht von unten berühren. $\sqrt{Z}$ liefert eine achsenparallele Einschließung der komplexen Zahlenmenge

$$\{y \in \mathbb{C} \,|\, y = \sqrt{z},\ z \in Z\} \subseteq \sqrt{Z},$$

wobei $\sqrt{z}$ als Hauptwert der komplexen Quadratwurzel definiert ist.

In Abbildung B.13 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $\sqrt{Z}$ angegeben.

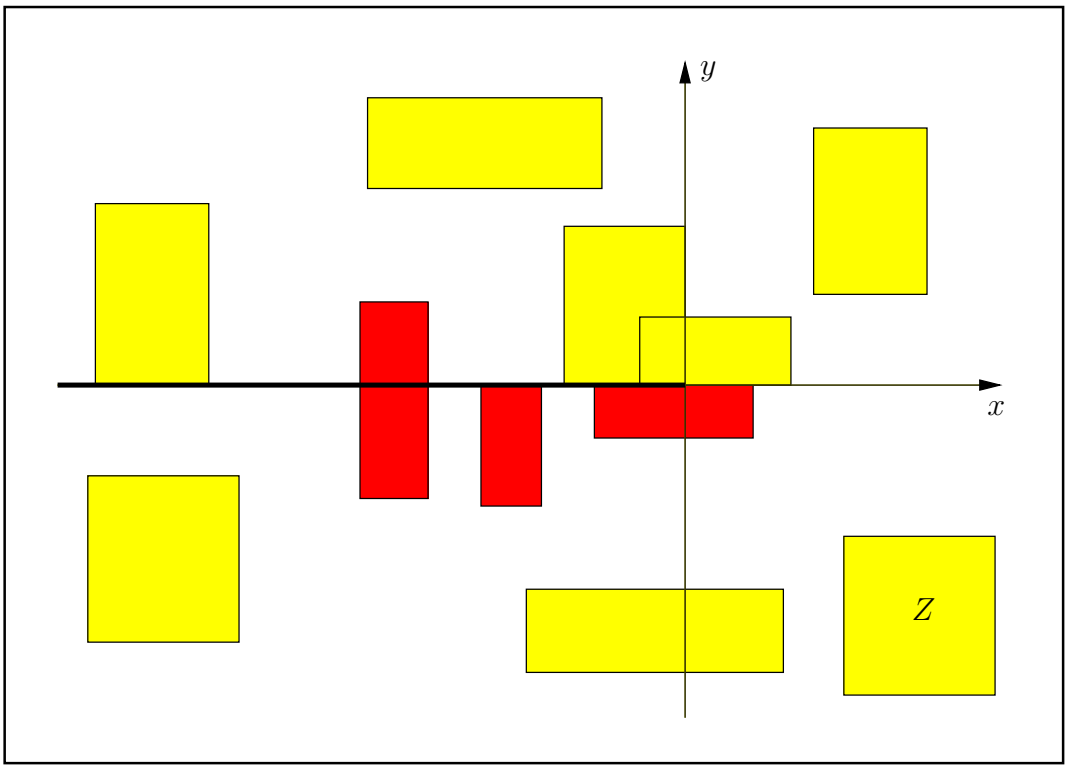


Abbildung B.13: Erlaubte und nicht erlaubte Intervalle Z von $\sqrt{Z}$.

Hier einige numerische Beispiele in der niedrigen Präzision `prec` = 30, womit etwa $30/\log_2(10) \approx 30/3.321928095 \approx 9$ Dezimalstellen berechnet werden:

`sqrt`$([+0,+0]+i\cdot[+0,+0]) = ([0.00000000, 0.00000000],[0.00000000, 0.00000000])$,
`sqrt`$([+0,+0]+i\cdot[+1,+2]) = ([7.07106781e-1, 1.00000001],[7.07106781e-1, 1.00000001])$,
`sqrt`$([+0,+0]+i\cdot[-2,-1]) = ([7.07106781e-1, 1.00000001],[-1.00000001, -7.07106781e-1])$,
`sqrt`$([-4,-2]+i\cdot[+0,+2]) = ([0.00000000, 6.43594253e-1],[1.41421356, 2.05817103])$,
`sqrt`$([-1,+0]+i\cdot[+0,+1]) = ([0.00000000, 7.07106782e-1],[0.00000000, 1.09868412])$,
`sqrt`$([-1,+2]+i\cdot[+0,+1]) = ([0.00000000, 1.45534670],[0.00000000, 1.09868412])$,
`sqrt`$([-1,+2]+i\cdot[-3,-2]) = ([7.86151377e-1, 1.67414923],[-1.44261528, -6.43594252e-1])$,
`sqrt`$([+2,+3]+i\cdot[-2,+1]) = ([1.41421356, 1.81735403],[-6.43594253e-1, 3.43560750e-1])$,
`sqrt`$([-3,-2]+i\cdot[-2,+2])$ ⇝

```
MpfciClass sqrt(const MpfciClass& z); z not in the principal branch.
```

B.2.4 $\sqrt{z}$, Beide Quadratwurzeln

Während mit `sqrt(`Z`)` nach Seite 159 eine achsenparallele Einschließung nur der Hauptwerte $\sqrt{z}$, $z \in Z$, geliefert wird, werden mit `sqrt_all(`Z`)` für **beliebige** achsenparallele Argumentintervalle Z in einer Liste die **beiden** möglichen Einschließungen für $\pm\sqrt{z}$, $z \in Z$, berechnet. Mit dem folgenden Programm

```
// MPFR-07.cpp
#include "mpfciclass.hpp"

using namespace MPFR;
using namespace MPFI;
using namespace cxsc;
using namespace std;

int main(void)
{
  interval re(-4,4), im(0,0);
  MpfciClass Z(re, im, 20);
  cout.precision(30/3.321928095); // Ausgabe mit 9 Dez.-Stellen
  cout << "Z = " << Z << endl;
  cout << "Z.GetPrecision() = " << Z.GetPrecision() << endl;

  list<MpfciClass> res;
  // Berechnung beider Einschliessungen in einer Liste:
  res = sqrt_all(Z);

  list<MpfciClass>::iterator pos;
  // Ausgabe der beiden Einschliessungen:
  for (pos = res.begin(); pos != res.end(); ++pos )
  {
    cout << *pos << endl; // Jede Einschliessung in neue Zeile
    cout << "Praezision = " << (*pos).GetPrecision() << endl;
  }

  return 0;
}
```

erhält man die Ausgabe

```
Z = ([-4.00000000,4.00000000], [0,-0])
Z.GetPrecision() = 20
([0,2.00000000], [0,2.00000000])
Praezision = 53
([-2.00000000,-0], [-2.00000000,-0])
Praezision = 53
```

wobei nach jeder Einschließung, die mit jeweils 9 Dezimalstellen erfolgt, auch noch die Präzision dieser berechneten Einschließung ausgegeben wird. Hier wird mit der Default-Präzision `prec` = 53 gerechnet, da keine Current-Präzision vereinbart wurde.
Es folgen noch weitere Beispiele in der niedrigen Präzision `prec` = 30, womit etwa $30/\log_2(10) \approx 30/3.321928095 \approx 9$ Dezimalstellen berechnet werden:

`sqrt_all`$([-2,-1]+i\cdot[-1,+1]) \rightsquigarrow$
$([-4.55089861e-1, 4.55089861e-1], [1.00000000, 1.45534670])$
$([-4.55089861e-1, 4.55089861e-1], [-1.45534670, -1.00000000])$.
`sqrt_all`$([-4,-2]+i\cdot[-4,+0]) \rightsquigarrow$
$([0.00000000, 1.11178595], [-2.19736823, -1.41421356])$
$([-1.11178595, -0.00000000], [1.41421356, 2.19736823])$.

B.2.5 $\sqrt[n]{z}$

Schreibt man eine komplexe Zahl $z = x + i \cdot y \in \mathbb{C}$ in Polarkoordinaten, so gilt

$$z = r \cdot e^{i \cdot \varphi}, \quad r = \sqrt{x^2 + y^2} \geq 0, \quad -\pi < \varphi \leq +\pi.$$

Die komplexen Lösungen $w_k \in \mathbb{C}$ der Gleichung

$$w^n = z, \quad n \in \mathbb{N}$$

sind gegeben durch die n kompleken Zahlen

$$w_k := \sqrt[n]{r} \cdot \left\{ \cos\left(\frac{\varphi + 2\pi k}{n}\right) + i \cdot \sin\left(\frac{\varphi + 2\pi k}{n}\right) \right\} = \sqrt[n]{r} \cdot e^{i \frac{\varphi + 2\pi k}{n}}, \quad k = 0, 1, 2, \ldots, n-1, \tag{B.24}$$

wobei die w_k auch allgemein als $\sqrt[n]{z}$ oder $z^{1/n}$ bezeichnet werden. In diesem Abschnitt definieren wir den Hauptzweig der n–ten Wurzel aus $z = r \cdot e^{i \cdot \varphi}$ durch:

$$\sqrt[n]{z} := w_0 = \sqrt[n]{r} \cdot \left\{ \cos\left(\frac{\varphi}{n}\right) + i \cdot \sin\left(\frac{\varphi}{n}\right) \right\} = \sqrt[n]{r} \cdot e^{i \frac{\varphi}{n}}, \quad -\pi < \varphi < +\pi.$$

Im **ersten Beispiel** betrachten wir $z = -1 + i = \sqrt{2} \cdot \{\cos(3\pi/4) + i \cdot \sin(3\pi/4)\} = \sqrt{2} \cdot e^{i \cdot (3\pi/4)}$. Dann gilt nach (B.24) für die dritten Wurzeln aus z, [47]

$$\begin{aligned} \sqrt[3]{-1+i} \;:=\; w_0 &= 2^{1/6} \cdot \{\cos(\pi/4) + i \cdot \sin(\pi/4)\}, \\ w_1 &= 2^{1/6} \cdot \{\cos(11\pi/12) + i \cdot \sin(11\pi/12)\}, \\ w_2 &= 2^{1/6} \cdot \{\cos(19\pi/12) + i \cdot \sin(19\pi/12)\}. \end{aligned}$$

Im **zweiten Beispiel** wählen wir $z = -2\sqrt{3} - 2i = 4 \cdot \{\cos(-5\pi/6) + i \cdot \sin(-5\pi/6)\} = 4 \cdot e^{-5\pi/6}$. Beachten Sie, dass jetzt $\varphi = -5\pi/6$ negativ sein muss, da z in der unteren Halbebene liegt. Dann gilt wieder nach (B.24) für die vierten Wurzeln aus z

$$\begin{aligned} \sqrt[4]{-2\sqrt{3} - 2i} \;:=\; w_0 &= \sqrt{2} \cdot \{\cos(-5\pi/24) + i \cdot \sin(-5\pi/24)\}, \\ w_1 &= \sqrt{2} \cdot \{\cos(7\pi/24) + i \cdot \sin(7\pi/24)\}, \\ w_2 &= \sqrt{2} \cdot \{\cos(19\pi/24) + i \cdot \sin(19\pi/24)\}, \\ w_3 &= \sqrt{2} \cdot \{\cos(-17\pi/24) + i \cdot \sin(-17\pi/24)\}, \end{aligned}$$

wobei auch jetzt wieder zu beachten ist, dass für die 4 komplexen Wurzelwerte die Bedingung $-\pi < \varphi \leq +\pi$ erfüllt sein muss.

Für ein vorgegebenes achsenparalles Rechteckintervall $Z \subset \mathbb{C}$ sollen jetzt für alle komplexen Zahlen $z \in Z$ die Funktionswerte des Hauptzweiges der n-ten Wurzel durch ein achsenparalles Rechteckintervall möglichst optimal eingeschlossen werden. Dazu definieren wir die komplexe Zahlenmenge $\sqrt{Z, n}$ durch:

$$\sqrt{Z, n} \;:=\; \begin{cases} [1,1] + i \cdot [0,0], & \text{für } n = 0 \text{ und für alle } Z \subset \mathbb{C}, \\ Z \subset \mathbb{C}, & \text{für } n = 1, \\ \square \left\{ \sqrt[n]{z} \,\middle|\, z \in Z \right\}, & Z \subset \mathbb{C}_0^-,\ n = 2, 3, 4, \ldots, \\ \text{undefiniert}, & Z \cap \mathbb{R}^- \neq \varnothing,\ n = 2, 3, 4, \ldots \end{cases}$$

$\mathbb{C}_0^-$ ist die längs der negativen reellen Achse aufgeschnittene komplexe Ebene $\mathbb{C}$, wobei die Null selbst zu $\mathbb{C}_0^-$ gehört. $\mathbb{R}^-$ bedeutet die Menge der negativen reellen Zahlen, und $\square \left\{ \sqrt[n]{z} \,\middle|\, z \in Z \right\}$ symbolisiert die achsenparallele Rechteckeinschließung der komplexen Menge $\left\{ \sqrt[n]{z} \,\middle|\, z \in Z \right\}$. Die in `mpfciclass.cpp` definierte Funktion

```
MpfciClass sqrt( const MpfciClass& z, int n )
```

liefert eine fast optimale Einschließung von $\sqrt{Z, n}$, d.h. es gilt

$$\sqrt{Z, n} \subseteq \texttt{sqrt(Z,n)}.$$

In der folgenden Abbildung B.14 sind für $n = 2, 3, 4, \ldots$ einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben:

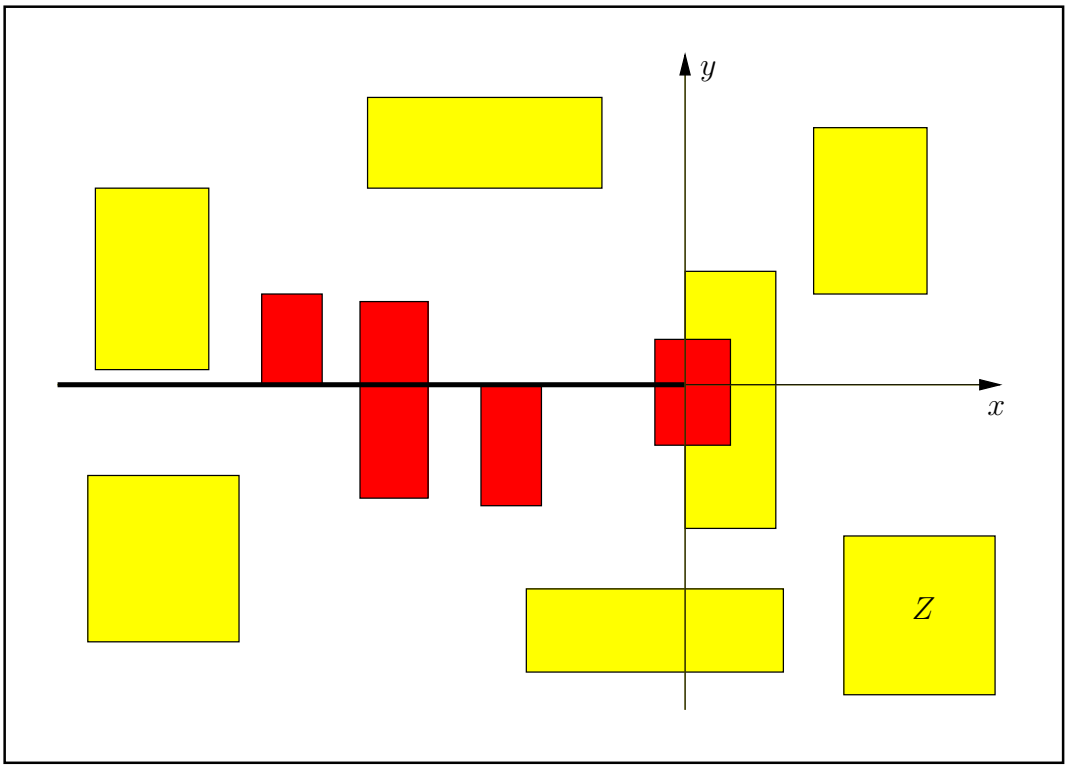


Abbildung B.14: Erlaubte und nicht erlaubte Intervalle Z von `sqrt(Z,n)`.

Es folgen noch einige numerische Beispiele in der niedrigen Präzision `prec` = 30, womit etwa $30/\log_2(10) \approx 30/3.321928095 \approx 9$ Dezimalstellen berechnet werden:

$\texttt{sqrt}([-2,+1] + i \cdot [-1,+1], 0) = ([1.00000000, 1.00000000], [0.00000000, 0.00000000])$
$\texttt{sqrt}([-2,+1] + i \cdot [-1,+1], 1) = ([-2.00000000, 1.00000000], [-1.00000000, 1.00000000]),$
$\texttt{sqrt}([+1,+2] + i \cdot [-1,+0], 2) = ([1.00000000, 1.45534670], [-4.55089861e-1, -0.00000000]),$
$\texttt{sqrt}([-1,-1] + i \cdot [+1,+1], 3) = ([0.793700525, 0.793700526], [0.793700525, 0.793700526]),$
$\texttt{sqrt}([-2\sqrt{3}, -2\sqrt{3}] + i \cdot [-2,-2], 4) =$
$([1.12197105, 1.12197106], [-8.60918670e-1, -8.60918669e-1]),$
$\texttt{sqrt}([-5,-5] + i \cdot [-10,-10], 9) = ([1.27439289, 1.27439290], [-0.293084789, -0.293084788]),$
$\texttt{sqrt}([-2,+1] + i \cdot [-1,+1], 6) \rightsquigarrow$
`MpfciClass sqrt(const MpfciClass& z, const int n ); z contains negative real values.`

Anmerkung:

Der Parameter `n` sollte nicht zu groß gewählt werden. `n` = 100000 erfordert beispielsweise schon große Laufzeiten!

B.2.6 $\sqrt[n]{z}$ Alle Wurzeln

Für ein vorgegebenes achsenparalleles Rechteckintervall $Z \subset \mathbb{C}$ berechnet die Funktion, [11], [44]

```
std::list<MpfciClass> sqrt_all(const MpfciClass& Z, int n);
```

eine Liste von n achsenparallelen Rechteckintervallen, die jeweils alle Lösungen von $w^n = z$ für alle $z \in Z \subset \mathbb{C}$ einschließen. Mit dem folgenden Programm `MPFR-08.cpp` berechnen wir zunächst für das Punktintervall $Z = [-1,-1] + i \cdot [1,1]$ Einschließungen der drei Wurzeln w_0, w_1, w_2 der Gleichung $w^3 = Z$ aus dem ersten Beispiels von Seite 161.

```
// MPFR-08.cpp
#include "mpfciclass.hpp"

using namespace MPFR;
using namespace MPFI;
using namespace cxsc;
using namespace std;

int main(void)
{
  MPFI::MpfiClass::SetCurrPrecision(30);
  cout << "GetCurrPrecision() = " << MPFI::MpfiClass::GetCurrPrecision() << endl;
  int n = 3; // n-te Wurzel berechnen
  interval re(-1,-1), im(1,1);
  MpfciClass Z(re, im, 53);
  cout.precision(30/3.321928095); // Ausgabe mit 9 Dez.-Stellen
  cout << "Z = " << Z << endl;
  cout << "Z.GetPrecision() = " << Z.GetPrecision() << endl;
  cout << "Berechnung aller " << n << "-ten Wurzeln aus Z" << endl;

  list<MpfciClass> res;
  res = sqrt_all(Z, n);

  list<MpfciClass>::iterator pos;
  // Ausgabe der n n-ten Wurzeln:
  for (pos = res.begin(); pos != res.end(); ++pos )
  {
    cout << *pos << endl; // Jede Einschliessung in neue Zeile
    cout << "Praezision = " << (*pos).GetPrecision() << endl;
  }

  return 0;
}
```

Das Programm liefert die Ausgabe

```
GetCurrPrecision() = 30
Z = ([-1.00000000,-1.00000000], [1.00000000,1.00000000])
Z.GetPrecision() = 53
Berechnung aller 3-ten Wurzeln aus Z
([7.93700525e-1,7.93700527e-1], [7.93700525e-1,7.93700527e-1])
Praezision = 30
([-1.08421509,-1.08421508], [2.90514555e-1,2.90514556e-1])
Praezision = 30
([2.90514555e-1,2.90514556e-1], [-1.08421509,-1.08421508])
Praezision = 30
```

mit den Einschließungen der drei Wurzeln w_0, w_1, w_2.

Mit $Z = [-2\sqrt{3}, -2\sqrt{3}] + i \cdot [-2, -2]$ liefert `sqrt_all(Z,4)` Einschließungen der vier Wurzeln w_0, w_1, w_2, w_3 der Gleichung $w^4 = Z$ aus dem zweiten Beispiel von Seite 161. Die Einschließungen sind:

$$
\begin{aligned}
w_0 &\in ([1.12197105, 1.12197106], [-8.60918670e-1, -8.60918668e-1]),\\
w_1 &\in ([8.60918668e-1, 8.60918670e-1], [1.12197105, 1.12197106]),\\
w_2 &\in ([-1.12197106, -1.12197105], [8.60918668e-1, 8.60918670e-1]),\\
w_3 &\in ([-8.60918670e-1, -8.60918668e-1], [-1.12197106, -1.12197105]).
\end{aligned}
$$

Beachten Sie, dass die Intervalle $Z \subset \mathbb{C}$ jetzt die negative reelle Achse nicht nur berühren sondern auch ganz in ihrem Innern enthalten dürfen. Mit $Z = [-1, -1] + i \cdot [0, 0]$ erhält man daher für die beiden Quadratwurzeln $w_0 = i, w_1 = -i$ die Einschließungen

$$
\begin{aligned}
w_0 &\in ([0.00000000, 0.00000000], [+1.00000000, +1.00000000]),\\
w_1 &\in ([0.00000000, 0.00000000], [-1.00000000, -1.00000000]).
\end{aligned}
$$

Mit $Z = [-1, -1] + i \cdot [-1, 1]$ erhält man die folgenden sechs Einschließungen der sechs Wurzeln $w_0, w_1, w_2, w_3, w_4, w_5$ der Gleichung $w^6 = Z$.

$$
\begin{aligned}
w_0 &\in ([7.93353340e-1, 9.78816269e-1], [3.82683432e-1, 6.44960268e-1]),\\
w_1 &\in ([-1.38287684e-1, 1.38287684e-1], [9.91444861e-1, 1.05946310]),\\
w_2 &\in ([-9.78816269e-1, -7.93353340e-1], [3.82683432e-1, 6.44960268e-1]),\\
w_3 &\in ([-9.78816269e-1, -7.93353340e-1], [-6.44960268e-1, -3.82683432e-1]),\\
w_4 &\in ([-1.38287684e-1, 1.38287684e-1], [-1.05946310, -9.91444861e-1]),\\
w_5 &\in ([7.93353340e-1, 9.78816269e-1], [-6.44960268e-1, -3.82683432e-1]).
\end{aligned}
$$

Mit $Z = [-1, +1] + i \cdot [4, 4]$ erhält man die folgenden acht Einschließungen der acht Wurzeln $w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7$ der Gleichung $w^8 = Z$.

$$
\begin{aligned}
w_0 &\in ([1.15870665, 1.17736602], [1.96183044e-1, 2.68620905e-1]),\\
w_1 &\in ([6.30104013e-1, 6.93274511e-1], [9.68097359e-1, 1.01238336]),\\
w_2 &\in ([-2.68620905e-1, -1.96183044e-1], [1.15870665, 1.17736602]),\\
w_3 &\in ([-1.01238336, -9.68097359e-1], [6.30104013e-1, 6.93274511e-1]),\\
w_4 &\in ([-1.17736602, -1.15870665], [-2.68620905e-1, -1.96183044e-1]),\\
w_5 &\in ([-6.93274511e-1, -6.30104013e-1], [-1.01238336, -9.68097359e-1]),\\
w_6 &\in ([1.96183044e-1, 2.68620905e-1], [-1.17736602, -1.15870665]),\\
w_7 &\in ([9.68097359e-1, 1.01238336], [-6.93274511e-1, -6.30104013e-1]).
\end{aligned}
$$

Mit $Z = [+1, +1] + i \cdot [-0, +0]$ erhält man die folgenden fünf Einschließungen der fünf Einheitswurzeln w_0, w_1, w_2, w_3, w_4 der Gleichung $w^5 = 1$.

$$
\begin{aligned}
w_0 &\in ([1.00000000, 1.00000000], [0.00000000, 0.00000000]),\\
w_1 &\in ([3.09016994e-1, 3.09016995e-1], [9.51056516e-1, 9.51056517e-1]),\\
w_2 &\in ([-8.09016995e-1, -8.09016994e-1], [5.87785252e-1, 5.87785253e-1]),\\
w_3 &\in ([-8.09016995e-1, -8.09016994e-1], [-5.87785253e-1, -5.87785252e-1]),\\
w_4 &\in ([3.09016994e-1, 3.09016995e-1], [-9.51056517e-1, -9.51056516e-1]).
\end{aligned}
$$

B.2.7 $\cot(z)$

Mit $z = x + i \cdot y$ gilt nach Abramowitz (4.3.58)

$$\cot(z) = \frac{\sin(2x) - i \cdot \sinh(2y)}{\cosh(2y) - \cos(2x)} = u(x,y) + i \cdot v(x,y) \tag{B.25}$$

$$= \frac{\cos(z)}{\sin(z)}, \quad z \neq k\pi,\ k \in \mathbb{Z} \tag{B.26}$$

$$= \frac{1}{\tan(z)} \tag{B.27}$$

$$= \tan(\pi/2 - z) = \tan(\pi/2 - z + k\pi), \quad k \in \mathbb{Z} \tag{B.28}$$

Die Polstellen der Cotangens-Funktion sind gegeben durch $z_{p,k} = k\pi$, und ihre Nullstellen liegen bei $z_{s,k} = \pi \cdot (k + 1/2),\ k \in \mathbb{Z}$.

Da der Tangens für komplexe Intervallargumente Z bereits implementiert ist, soll der ebenfalls π-periodische Cotangens mit Hilfe des Tangens realisiert werden. Die Gleichung (B.27) ist dabei ungeeignet, denn im Falle $\pi/2 \in Z$ liegt eine Polstelle des Tangens in Z, so dass mit (B.27) Programmabbruch erfolgt, obwohl der Cotangens in der Umgebung von $\pi/2$ existiert und bei $z_{s,0} = \pi/2$ eine Nullstelle besitzt. Die Darstellung (B.26) ist ebenfalls ungeeignet, da zwei Ergebnisintervalle zu dividieren sind, womit eine erhebliche, zusätzliche Überschätzung verbunden ist. Zur Implementierung des Cotangens wählen wir daher (B.28). Ist ein rechteckiges Argumentintervall $Z = X + i \cdot Y$ vorgegeben, so kann es bei der intervallmäßigen Auswertung von $\pi/2 - Z$ zu den bekannten Überschätzungen kommen, wenn $\texttt{Inf}(Z) \approx \pi/2$ oder $\texttt{Sup}(Z) \approx \pi/2$ realisiert werden. Diese Überschätzungen sind prinzipiell unvermeidbar, da $\pi/2$ keine Maschinenzahl ist und daher durch ein echtes Intervall eingeschlossen werden muss. Insbesondere bei Punktintervallen $Z \approx \pi/2$ lassen sich diese Überschätzungen jedoch vermeiden, wenn man die Intervalldifferenz $\pi/2 - Z$ in doppelter Präzision ausführt. Beachten Sie jedoch, dass die Intervalldifferenz $\pi/2 - Z$ selbst dann von $[0,0]$ verschieden ist, wenn Z eine optimale Einschließung von $\pi/2$ ist, so dass dann die Auswertung von $\pi/2 - Z$ zu großen Überschätzungen führen muss.

Im folgenden **1. Beispiel** wählen wir ein Punktintervall $X = [x,x]$, wobei im Zahlenformat mit der Current-Präzision `prec` = 300

$$x = \texttt{Sup(Pi()/2)}$$

die kleinste Maschinenzahl größer $\pi/2$ ist. Mit $Z = ([x,x] + i \cdot [1,1])$ liefert die Version mit doppelter Präzision `prec` = 600 die Einschließung

```
cot(Z) = ([-3.08764047e-91,-3.08764046e-91], [-7.61594156e-1,-7.61594155e-1])
```

während man bei nur einfacher Präzision für den Realteil die praktisch unbrauchbare Einschließung erhält:

```
cot(Z) = ([-4.12338660e-91,0.00000000], [-7.61594156e-1,-7.61594155e-1]).
```

Im **2. Beispiel** wählen wir mit $Z = (\texttt{Pi()}/2 + i \cdot [1,1])$ ein echtes komplexes Maschinenintervall, das im Zahlenformat mit der Current-Präzision `prec` = 300 den Realteil $\pi/2$ optimal einschließt. Aber selbst mit doppelter Präzision `prec` = 600 erhält man jetzt nur die grobe Einschließung

```
cot(Z) = ([-3.08764047e-91,1.03574613e-91], [-7.61594156e-1,-7.61594155e-1]),
```

die auch bei einer drei- oder vierfachen Präzision beim Realteil **grundsätzlich** nicht verbessert werden kann.

B.2.8 arcsin(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{(-\infty, -1) \cup (+1, +\infty)\}$ liefert die Funktion

```
MpfciClass asin(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `asin(Z)` für die komplexen Funktionswerte $\arcsin(z)$, mit $z \in Z$.

$$\{\arcsin(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{asin(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der reellen Achse von -1 bis $-\infty$ bzw. von $+1$ bis $+\infty$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

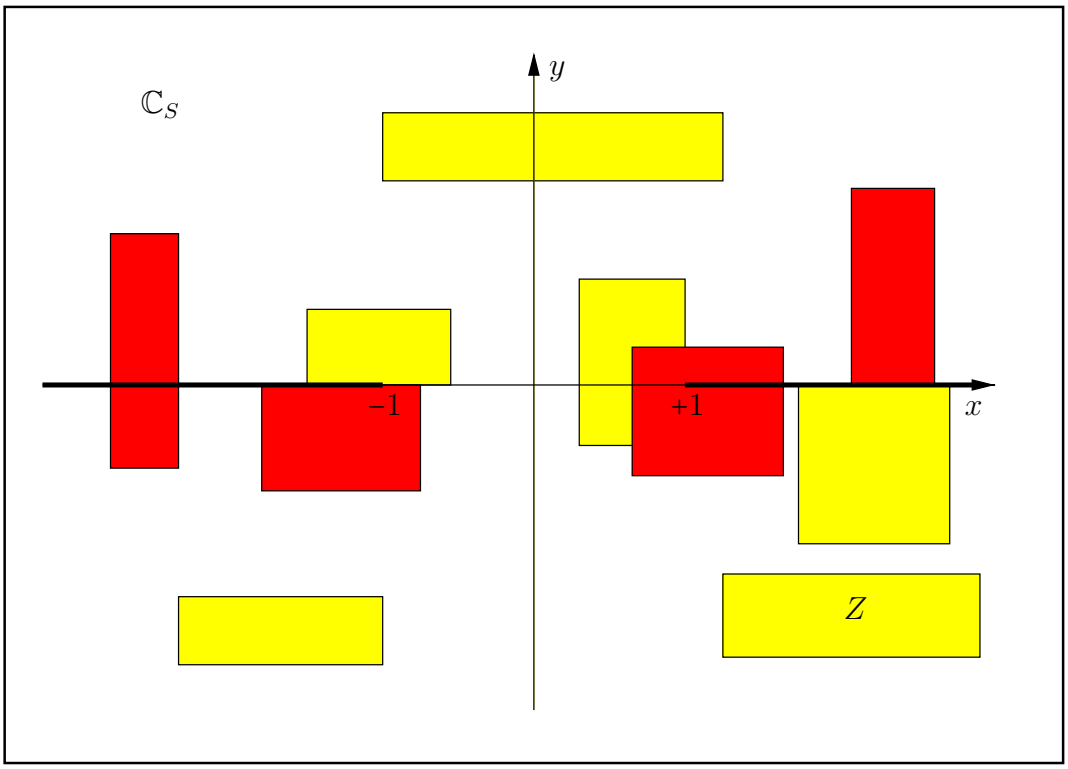


Abbildung B.15: Erlaubte und nicht erlaubte Intervalle Z von `asin(Z)`.

Mit $Z = [1,1] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[4]

$$\begin{aligned}\texttt{asin(Z)} \;=\; &([1.57079632679489661922, 1.57079632679489661924],\\ &[4.88115243040816240520e-161614249, 4.88115243040816240522e-161614249]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{asin(Z)} \;=\; &([3.49439062857213293627411825e-1, 1.57079632679489661923132170],\\ &[-1.26047518779845407285290814, 0.00000000000000000000000000]).\end{aligned}$$

[4] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

B.2.8.1 Algorithmus

Der nachfolgende Algorithmus zur Berechnung der Rechteck-Einschließung `asin(Z)` für ein vorgegebenes achsenparalleles Rechteckintervall Z basiert auf [44], [43], [10], wobei sich die Verbesserungen in [10] auf das dort verwendete IEEE-Format beziehen. Da `asin(Z)` jetzt im MPFCI-Format implementiert wird, müssen die Verbesserungen entsprechend an dieses Format angepasst werden.

Mit $z = x + i \cdot y \in \mathbb{C}$ gelten nach W. Krämer für den komplexen Funktionswert $w = \arcsin(z) = \Re(w) + i \cdot \Im(w)$ die folgenden Beziehungen, [28]:

$$\alpha := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\} \tag{B.29}$$

$$\beta := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} - \sqrt{(x-1)^2 + y^2} \right\} \tag{B.30}$$

$$\beta = \frac{x}{\frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\}} = \frac{x}{\alpha} \tag{B.31}$$

$$\Re(w) := \arcsin(\beta) \tag{B.32}$$

$$\Im(w) = \begin{cases} +\operatorname{arcosh}(\alpha), & \text{falls } y > 0 \\ +\operatorname{arcosh}(\alpha), & \text{falls } y = 0 \text{ und } x \le -1 \\ 0 \quad , & \text{falls } y = 0 \text{ und } -1 \le x \le +1 \\ -\operatorname{arcosh}(\alpha), & \text{falls } y = 0 \text{ und } x \ge +1 \\ -\operatorname{arcosh}(\alpha), & \text{falls } y < 0 \end{cases} \tag{B.33}$$

Wegen $\alpha(x,y) \ge \alpha(x,0) = (|x+1| + |x-1|)/2$ erhält man mit einfachen Fallunterscheidungen ($|x| \le 1,\ x < -1,\ x > +1$):

$$\alpha(x,y) \ge \max(1,|x|) = \begin{cases} 1, & \text{wenn } |x| \le 1, \\ |x|, & \text{wenn } |x| > 1, \end{cases} \qquad \text{oder gröber: } \alpha(x,y) \ge 1. \tag{B.34}$$

Unter den Voraussetzungen $y = 0$ und $-1 \le x \le +1$ gilt:

$$\alpha(x,0) \equiv 1, \quad \text{d.h. } \Im(w) = \pm\operatorname{arcosh}(1) = 0, \quad \Re(w) = \arcsin(x); \tag{B.35}$$

Es gilt außerdem

$$|\beta(x,y)| \le |\beta(x,0)| \le 1$$

Zum Beweis gilt nach (B.31):

$$|\beta(x,y)| \le |\beta(x,0)| = \frac{2 \cdot |x|}{|x+1| + |x-1|} =: R(x),$$

und wegen $R(-x) \equiv R(x)$ kann man sich auf $x \ge 0$ beschränken:
Sei $\boldsymbol{0 \le x \le 1}$, d.h. $|x-1| = -(x-1) \ \rightsquigarrow\ R(x) = \frac{2x}{(x+1)-(x-1)} = x \le 1$;
Sei $\boldsymbol{x > 1}$, d.h. $|x-1| = +(x-1) \ \rightsquigarrow\ R(x) = \frac{2x}{(x+1)+(x-1)} = 1$ ∎

In [43] wird für das komplexe Intervallargument

$$\boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y} = [x_1, x_2] + i \cdot [y_1, y_2]$$

eine Einschließung für $\arcsin(\boldsymbol{z})$ berechnet, wobei die beiden reellen Funktionen $\arcsin(\beta)$ und $\operatorname{arcosh}(\alpha)$ für **Intervallargumente $\boldsymbol{\alpha}$** und **$\boldsymbol{\beta}$** auszuwerten sind. Dabei sind in (B.29) und (B.31) 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.

Bei der Auswertung von α und β nach (B.29) und (B.31) ist folgender Term $T(x,y)$ zu berechnen:

$$T := \sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2} \tag{B.36}$$

Um $T <$ `MaxFloat()` für jede Präzision zu gewährleisten, betrachten wir folgende Abschätzungen:
Zunächst gilt: $T(x,y) = T(|x|,y) = \sqrt{(|x|+1)^2+y^2} + \sqrt{(|x|-1)^2+y^2}$;
Der Beweis ergibt sich direkt mit den Fallunterscheidungen: $x \geq 0$ und $x < 0$. Man findet zusätzlich:

$$\begin{aligned} T(|x|,y) \equiv T(|x|,|y|) &\leq \sqrt{(|x|+1)^2+y^2} + \sqrt{(|x|+1)^2+y^2} \\ &= 2\cdot\sqrt{(|x|+1)^2+y^2} < 2\cdot\sqrt{(|x|+1)^2+(|y|+1)^2} \\ &\leq 2\sqrt{2}\cdot(M+1), \quad \text{falls} \quad M = \max\{|x|,|y|\}. \end{aligned}$$

Bei der Auswertung von $T(x,y)$ wird also Overflow verhindert, falls gilt[5]:

$$2\sqrt{2}\cdot(M+1) < \texttt{MaxFloat(2)}$$

$$\Longleftrightarrow \qquad M < \frac{\texttt{MaxFloat(2)}}{2\sqrt{2}} - 1 = 0.530330085889\ldots\cdot 2^{1073741822}$$

Mit $gr := 0.530330085889\cdot 2^{1073741822}$ und der Anweisung

```
if (|x|>gr or |y|>gr) 'Programm-Abbruch'
```

verhindert man daher einen vorzeitigen Overflow bei der Berechnung von $T(x,y)$, wobei die obige Einschränkung in der Praxis keine Bedeutung haben wird. $\alpha := T(x,y)/2$ wird mit der Hilfsfunktion

```
MpfiClass f_aux_asin(const MpfiClass& x, const MpfiClass& y)
```

zusammen mit `sqrtx2y2()` ausgewertet, wobei für `x,y` nur Punktintervalle übergeben werden. Die oben beschriebene Abfrage `if (|x|>gr or |y|>gr)` erfolgt nicht in `f_aux_asin(x,y)` selbst, sondern in einer übergeordneten Funktion. In `f_aux_asin(x,y)` wird zusätzlich noch die erste Ungleichung in (B.34) überprüft, die durch Intervallüberschätzungen verletzt sein kann.

Eine nahezu optimale Einschließung des Realteils $\Re(w)$ nach (B.32) durch die Auswertung von $\arcsin(\boldsymbol{\beta})$ scheint damit gewährleistet zu sein, wenn man zusätzlich im entsprechenden Algorithmus aus [43] (`2nd: real part`) den offensichtlichen Tippfehler, `hxl` statt `hxu`, beseitigt:

```
if( srez < 0.0 )
    resxu = Sup(asin(hxu/f_aux_asin(hxu,interval(max(-iimz,simz)))));
```

Für $\beta \to -1$ oder $\beta \to +1$ wird jedoch der Realteil $\Re(w)$ durch die Auswertung von $\arcsin(\boldsymbol{\beta})$ noch nicht optimal eingeschlossen, da wegen der nahezu vertikalen Tangenten an den Intervallrändern von $\beta \in [-1,+1]$ nur minimale Überschätzungen beim Argument β große Überschätzungen beim reellen $\arcsin(\boldsymbol{\beta})$ bewirken.

Diese Überschätzungen lassen sich jedoch im Falle $\beta \geq 0.75$, d.h. $\beta \to +1$ durch die folgende Transformation $\beta = 1-\delta$ bzw. $\delta = 1-\beta$ beseitigen:

$$\arcsin(\beta) = \arcsin(1-\delta) = \frac{\pi}{2} - \arcsin(\sqrt{\delta\cdot(2-\delta)}), \quad 0 \leq \delta \leq 2; \tag{B.37}$$

Wertet man jetzt in Gleichung (B.37) den letzten Ausdruck intervallmäßig aus, so wird von der optimalen Einschließung von $\pi/2$ der betragsmäßig kleine Intervallausdruck $\arcsin(\sqrt{\boldsymbol{\delta\cdot(2-\delta)}})$ subtrahiert, so dass keine nennenswerten Überschätzungen auftreten können. Die Berechnung von $\delta(x,y) := 1-\beta(x,y)$ ist auch bei der Einschließung des Realteils von $\arccos(z)$ erforderlich. Für die Fälle $0.75 \leq x < 1$, $x = 1$ und $x > 1$ findet man die entsprechenden Formeln in (B.103), (B.93) und (B.70). Wegen der Identität $\arcsin(-\beta) \equiv -\arcsin(\beta)$ kann der andere Fall $\beta \to -1$ auf den oben behandelten Fall $\beta \to +1$ zurückgeführt werden. Die Auswertung von $\arcsin(\boldsymbol{\beta})$ erfolgt mit der Hilfsfunktion

[5] Für `prec` ≥ 2 gilt: `MaxFloat(2)` $\leq$ `MaxFloat(prec)`, d.h. `MaxFloat(prec)` ist präzisionsabhängig!

```
MpfiClass Asin_beta( const MpfiClass& x, const MpfiClass& y );
```

Bei der intervallmäßigen Auswertung von $\delta(x,y) \geq 0$ kann in seltenen Fällen das Infimum seiner Einschließung negativ werden, wodurch später ein `NaN` erzeugt wird. Diese Fehlerquelle wird in `Asin_beta(x,y)` mit der Abfrage `if (Inf(delta)<0)` beseitigt.

Wir kommen jetzt zur Berechnung einer Einschließung des Imaginärteils, wobei nach (B.33) $\operatorname{arcosh}(\boldsymbol{\alpha})$ für das Intervallargument $\boldsymbol{\alpha}$ auszuwerten ist. Nach (B.29) sind dabei für x, y entsprechende Punktintervalle $\boldsymbol{x} = [x_1, x_1]$, $\boldsymbol{y} = [y_1, y_1]$, $x_1, y_1 \in S(2,53)$ einzusetzen:

$$\boldsymbol{\alpha} := \frac{1}{2} \cdot \left\{ \sqrt{(\boldsymbol{x}+1)^2 + \boldsymbol{y}^2} + \sqrt{(\boldsymbol{x}-1)^2 + \boldsymbol{y}^2} \right\}$$

Bei der Auswertung von $\operatorname{arcosh}(\boldsymbol{\alpha})$ treten jedoch für $\mathtt{Sup}(\boldsymbol{\alpha}) \to +1$ erhebliche Überschätzungen auf, da die arcosh-Funktion dann in der Nähe ihrer Nullstelle 1 zu berechnen ist. Diese Überschätzungen des Imaginärteils lassen sich jedoch vermeiden, wenn man mit der folgenden Transformation $\alpha = 1 + (\alpha - 1) = 1 + \delta,\ \delta := \alpha - 1 \geq 0$ die Identität

$$\operatorname{arcosh}(\alpha) \equiv \operatorname{arcosh}(1+\delta) \tag{B.38}$$

benutzt und dabei $\operatorname{arcosh}(1+\delta)$ mit Hilfe der Funktion `acoshp1`(δ) auswertet.

Bevor wir auf die Berechnung von $\delta := \alpha - 1 \geq 0$ näher eingehen, bestimmen wir zunächst in der komplexen Ebene diejenigen Bereiche, in denen $\operatorname{arcosh}(\alpha)$ direkt oder mit Hilfe von $\operatorname{arcosh}(1+\delta)$ auszuwerten ist. Dazu beweisen wir mit $z = x + i \cdot y$:

$$|x| \geq 2 \ \vee\ |y| \geq 2 \quad \Longrightarrow \quad \alpha(x,y) \geq 2; \tag{B.39}$$

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 $|y| \geq 2$ folgt der zweite Teil ∎

Außerhalb eines Quadrats mit der Seitenlänge 4 und dem Mittelpunkt im Ursprung der komplexen Ebene benutzen wir daher wegen (B.39) $\operatorname{arcosh}(\alpha)$ direkt und innerhalb dieses Quadrats gilt: $\operatorname{arcosh}(\alpha) \equiv \operatorname{arcosh}(1+\delta)$, mit $\delta := \alpha - 1$.

Wir kommen jetzt zur Berechnung von $\delta := \alpha - 1 \geq 0$:

Man findet zunächst

$$x = \pm 1 \quad \rightsquigarrow \quad \delta = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2}, \tag{B.40}$$

wobei der erste Summand $\{\ldots\}$ mit Hilfe der **C-XSC** Funktion `sqrtp1m1(...)` berechnet wird. Für $|x| \neq 1$ gilt mit $K(x) := |x+1| + |x-1| - 2$:

$$\begin{aligned}
2\delta &= |x+1| \cdot \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 2 + \sqrt{(x-1)^2 + y^2} \\
&= |x+1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right\} + |x+1| - 2 + \sqrt{(x-1)^2 + y^2} \\
&= |x+1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right\} + |x-1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right\} \\
&\quad + K(x)
\end{aligned}$$

Mit der folgenden Darstellung für $K(x)$

$$K(x) = \begin{cases} 0 & , \text{falls } \ 0 \leq |x| \leq 1 \\ 2 \cdot (|x| - 1) & , \text{falls } \ |x| > 1 \end{cases} \qquad \text{und mit}$$

$$V(x,y) := |x+1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right\} + |x-1| \left\{ \sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right\}$$

gilt dann insgesamt:

$$\delta = \frac{V(x,y)}{2} \quad \text{, falls } |x| < 1 \tag{B.41}$$

$$\delta = \frac{V(x,y)}{2} + (|x| - 1) \quad \text{, falls } |x| > 1 \tag{B.42}$$

$$\delta = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2} \quad \text{, falls } |x| = 1, \tag{B.43}$$

dabei werden die mit $\{\ldots\}$ gekennzeichneten Ausdrücke mit der **C-XSC** Funktion `sqrtp1m1()` berechnet. Bitte beachten Sie, dass innerhalb des oben beschriebenen Quadrats mit der Kantenlänge 4 die quadratischen Argumente dieser Funktion, z.B. $(y/2)^2$ oder $(y/(x+1))^2$, ohne vorzeitigen Overflow berechnet werden. Beachten Sie außerdem, dass $V(x,y)$ eine Summe aus nur positiven Summanden ist, so dass keine Auslöschungen zu befürchten sind.

Die Auswertung von $\operatorname{arcosh}(\alpha)$ nach (B.38) mit Hilfe der Ausdrücke für δ nach (B.41) und (B.43) können im Falle $\delta \ll 1$ noch wesentlich vereinfacht werden. Dabei sind für x, y Punktintervalle einzusetzen, so dass in (B.41) und (B.43) δ als Intervallargument $\boldsymbol{\delta}$ zu verstehen ist.

Wir beginnen mit dem Fall: $|x| = 1$;

$$\begin{aligned}
\operatorname{arcosh}(\alpha) &\equiv \operatorname{arcosh}(1+\delta), \quad \delta = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2}; \quad \rightsquigarrow \\
&\equiv \operatorname{arcosh}(1+\delta), \quad \delta = \left\{ \sqrt{1 + \left(\frac{t}{2}\right)^2} - 1 \right\} + \frac{t}{2},\ t := |y| \geq 0; \\
&\equiv \operatorname{arcosh}(\sqrt{1 + (t/2)^2} + t/2), \quad t = |y| \geq 0;
\end{aligned}$$

Die wesentliche Vereinfachung besteht nun darin, für $t \ll 1$ die Potenzreihe des obigen Ausdrucks $\operatorname{arcosh}(\sqrt{1+(t/2)^2}+t/2)$ zu benutzen. Um Aussagen über diese Reihe machen zu können, geben wir zunächst die Potenzreihe für $\operatorname{arcosh}(1+t)$ an[6]:

$$\operatorname{arcosh}(1+t) = \sqrt{2t} \cdot Q(t); \tag{B.44}$$

$$Q(t) := \sum_{k=0}^{\infty} a_k \cdot t^k = 1 - \frac{1}{12}t + \frac{3}{160}t^2 - \frac{5}{896}t^3 + \frac{35}{18432}t^4 - \frac{63}{90112}t^5 \pm \ldots$$

$$a_k = \frac{1}{(2k+1)\cdot 2^k} \cdot \binom{-0.5}{k}, \quad a_0 = 1,$$

$$a_{k+1} = -a_k \cdot \frac{(2k+1)^2}{4\cdot(2k+3)(k+1)},\ k = 0, 1, 2, \ldots$$

Mit Hilfe des Majorantenkriteriums und der geometrischen Reihe kann leicht gezeigt werden, dass die Reihe für $Q(t)$ sogar absolut konvergiert, wenn $t < 1$.

Mit `Maple7` und `Mathematica` erhält man:

$$\operatorname{arcosh}(\sqrt{1 + (t/2)^2} + t/2) = \sqrt{t} \cdot H(t), \quad H(t) = \sum_{k=0}^{\infty} b_k,$$

$$H(t) = 1 + \frac{1}{12}t - \frac{3}{160}t^2 - \frac{5}{896}t^3 + \frac{35}{18432}t^4 + \frac{63}{90112}t^5 - - + + \ldots,$$

[6] Vgl. die Ergebnisse zur Funktion $\operatorname{arcosh}(1+x)$ im Buch zur Fehlerabschätzung; dort wird auch die folgende Ungleichung gezeigt: $|a_{k+1}| < |a_k|,\ k = 0, 1, \ldots$

und es gilt vermutlich[7] $|a_k| = |b_k|$. Unter dieser Voraussetzung ist dann auch $H(t)$ absolut konvergent, so dass wie folgt geklammert werden kann:

$$\begin{aligned} H(t) &= (1+\frac{1}{12}t) - (\frac{3}{160}t^2 + \frac{5}{896}t^3) + (\frac{35}{18432}t^4 + \frac{63}{90112}t^5) - + \ldots \\ &= h_1 - h_2 + h_3 - h_4 \pm \ldots, \quad h_j \geq 0; \end{aligned} \tag{B.45}$$

Damit erhalten wir für $H(t)$ eine *alternierende* Reihe. Wegen $|a_{k+1}| < |a_k|$ kann unter der Voraussetzung $t < 1$ leicht gezeigt werden: $h_{k+1} < h_k$, so dass die Reihe in (B.45) eine alternierende Leibniz-Reihe ist. Damit ergeben sich im Falle $|x| = 1$ und $t = |y| < 1$ mit $\operatorname{arcosh}(\alpha) = \operatorname{arcosh}(\sqrt{1+(t/2)^2} + t/2)$ die folgenden Abschätzungen:

$$\sqrt{t} \cdot \left[\left(1 + \frac{t}{12}\right) - t^2 \left(\frac{3}{160} + \frac{5t}{896}\right) \right] \leq \operatorname{arcosh}(\alpha), \quad t = |y| < 1; \tag{B.46}$$

$$\operatorname{arcosh}(\alpha) = \operatorname{arcosh}\left(\sqrt{1+\left(\frac{t}{2}\right)^2} + \frac{t}{2}\right) \leq \sqrt{t} \cdot \left(1 + \frac{t}{12}\right), \; t = |y| < 1; \tag{B.47}$$

Für $0 \leq t = |y| < 1$ gelten noch die folgenden Abschätzungen:

$$\begin{aligned} &\sqrt{t}\left(1 - \frac{3}{80}t^2\right) \leq \sqrt{t}\left[1 - t^2\left(\frac{3}{160} + \frac{3}{160}\right)\right] \leq \sqrt{t}\left[1 - t^2\left(\frac{3}{160} + \frac{5}{896} \cdot 1\right)\right] \leq \\ &\leq \sqrt{t}\left[1 - t^2\left(\frac{3}{160} + \frac{5}{896} \cdot t\right)\right] \leq \sqrt{t}\left[\left(1 + \frac{1}{12} \cdot t\right) - t^2\left(\frac{3}{160} + \frac{5}{896} \cdot t\right)\right] \end{aligned}$$

Mit (B.46) und (B.47) erhalten wir schließlich im Fall $|x| = 1$:

$$\sqrt{t} \cdot \left(1 - \frac{3}{80}t^2\right) \leq \operatorname{arcosh}(\alpha) \leq \sqrt{t} \cdot \left(1 + \frac{t}{12}\right), \qquad t = |y| < 1; \tag{B.48}$$

Die Auswertung dieser Abschätzung erfolgt in der Hilfsfunktion

```
MpfiClass ACOSH_f_aux(const MpfiClass& x, const MpfiClass& y);
```

unter der Voraussetzung `(expo(Inf(delta)) < -GetCurrPrecision())`. Bei hoher Current-Präzision kommt obige Abschätzung also nur bei entsprechend kleinen Werten von $\delta := |y|$ zur Anwendung, um möglichst optimale Einschließungen zu erhalten.

Wir betrachten jetzt nach (B.41) den Fall: $|x| < 1$, mit $\delta := V(x,y)/2$, wobei $V(x,y)$ definiert war durch:

$$V(x,y) := |x+1| \cdot \left\{ \sqrt{1+\left(\frac{y}{x+1}\right)^2} - 1 \right\} + |x-1| \left\{ \sqrt{1+\left(\frac{y}{x-1}\right)^2} - 1 \right\}.$$

Für die Einschließung des Imaginärteils $\Im(\boldsymbol{w})$ von $\boldsymbol{w} = \arcsin(\boldsymbol{z})$ ist dann nach (B.38) der Ausdruck $\operatorname{arcosh}(1+\boldsymbol{\delta})$ für das Intervallargument $\boldsymbol{\delta}$ auszuwerten, wobei in $V(x,y)$ für x, y die entsprechenden Punktintervalle einzusetzen sind. Damit scheint das Problem gelöst zu sein, da man für $y \to 0$ die Ausdrücke $\{\ldots..\}$ mit Hilfe der bereits implementierten Funktion `sqrtp1m1(..)` intervallmäßig auswerten kann. Dabei übersieht man jedoch, dass wegen der Reihenentwicklung

$$\sqrt{1+t} - 1 = t \cdot \sum_{k=0}^{\infty} (-1)^{k+1} \cdot \left(\prod_{n=0}^{k} \frac{2n-1}{2n+2} \right) \cdot t^k \tag{B.49}$$

$$= \frac{t}{2} \cdot \left(1 - \frac{t}{4} + \frac{t^2}{8} - \frac{5 \cdot t^3}{64} \pm \ldots\right), \quad |t| < 1; \tag{B.50}$$

[7] $|a_k| = |b_k|$ konnte z.B. mit `Maple` bis $k = 100$ bestätigt werden.

die obigen Ausdrücke $\{\ldots\}$ für $y \to 0$ von der Größenordnung

$$\frac{1}{2} \cdot \left(\frac{y}{x \pm 1}\right)^2$$

sind und damit bei der späteren Intervallauswertung von $\sqrt{V}$ zu einem vorzeitgen Unterlauf führen, womit dann starke Überschätzungen verbunden sind. Diese Überschätzungen kann man zwar vermeiden, indem man y geeignet skaliert, die entsprechenden Rechnungen sind jedoch aufwendig und lassen sich durch den folgenden sehr einfachen Algorithmus für $|y| \to 0$ ohne Anwendung der Funktion `sqrtp1m1(...)` ganz vermeiden.

Um $V(x,y)$ einzuschließen, berechnen wir zunächst eine **Oberschranke**:
Da die Reihe in (B.50) für $0 \le t < 1$ eine alternierende Leibniz-Reihe ist, gilt die Abschätzung[8]:

$$\frac{t}{2} \cdot \left(1 - \frac{t}{4}\right) \le \sqrt{1+t} - 1 \le \frac{t}{2}, \quad \text{falls } \; 0 \le t < 1; \tag{B.51}$$

Mit

$$v_1 := |x+1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right\}$$

folgt dann direkt

$$v_1 \le |x+1| \cdot \frac{1}{2} \cdot \left(\frac{y}{x+1}\right)^2 = \frac{1}{2} \cdot \frac{y^2}{|x+1|}, \quad \text{falls } \; y^2 < (x+1)^2.$$

Mit

$$v_2 := |x-1| \cdot \left\{ \sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right\}$$

folgt ganz entsprechend

$$v_2 \le |x-1| \cdot \frac{1}{2} \cdot \left(\frac{y}{x-1}\right)^2 = \frac{1}{2} \cdot \frac{y^2}{|x-1|}, \quad \text{falls } \; y^2 < (x-1)^2.$$

Für $V(x,y) = v_1 + v_2$ erhält man unter den Voraussetzungen $|y| < |x+1|$, $|y| < |x-1|$ und $|x| < 1$ die Oberschranke:

$$V(x,y) \le \frac{y^2}{2} \cdot \left[\frac{1}{|x+1|} + \frac{1}{|x-1|}\right] = \frac{y^2}{1-x^2}. \tag{B.52}$$

Unter der Voraussetzung $|x| < 1$ sind die beiden Bedingungen $|y| < |x+1|$, $|y| < |x-1|$ für jede Current-Präzision $\texttt{prec} \ge 2$ erfüllt, wenn gilt: $|y| < 1 - \texttt{pred}(1) = 2^{-\texttt{prec}}$. Damit erhalten wir

$$V(x,y) \le \frac{y^2}{1-x^2}, \qquad \text{falls } |x| < 1 \text{ und } |y| < 2^{-\texttt{prec}}. \tag{B.53}$$

Um $V(x,y)$ einzuschließen, berechnen wir jetzt eine **Unterschranke**. Mit (B.51) folgt jetzt ganz analog:

$$\frac{1}{2} \cdot \frac{y^2}{|x+1|} \cdot \left[1 - \frac{1}{4}\left(\frac{y}{x+1}\right)^2\right] \le v1, \quad \text{falls } \; |y| < 2^{-\texttt{prec}}.$$

Wir stellen zusätzlich noch folgende Bedingung:

$$1 - \frac{1}{4}\left(\frac{y}{x+1}\right)^2 > \texttt{pred}(1) = 1 - 2^{-\texttt{prec}}, \quad \text{d.h.} \quad \left(\frac{y}{x+1}\right)^2 < 2^{-\texttt{prec}+2}.$$

[8] Diese Abschätzung lässt sich sehr einfach auch direkt beweisen.

Es gilt für $|x| < 1$ die Abschätzung $(x+1)^2 \geq (\mathtt{succ}(-1)+1)^2 = 2^{-2\mathtt{prec}}$ und damit:

$$\frac{y^2}{(x+1)^2} \leq \frac{y^2}{2^{-2\mathtt{prec}}}.$$

Die obige Bedingung ist also erfüllt, wenn gilt

$$\frac{y^2}{2^{-2\mathtt{prec}}} < 2^{-\mathtt{prec}+2} \quad \text{bzw. wenn} \quad |y| < 2^{-((3\mathtt{prec})/2)}.$$

Damit erhalten wir dann für $\mathtt{prec} \geq 2$ die Unterschranke:

$$\frac{1}{2} \cdot \frac{y^2}{|x+1|} \cdot \mathtt{pred}(1) \leq v_1, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}. \tag{B.54}$$

Beim obigen Exponenten $(3\mathtt{prec})/2$ ist die spezielle Maschinenrundung des integer-Quotienten in Richtung Null bereits berücksichtigt!

Ganz entsprechend findet man die Abschätzung

$$\frac{1}{2} \cdot \frac{y^2}{|x-1|} \cdot \mathtt{pred}(1) \leq v_2, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}, \quad \text{und damit:} \tag{B.55}$$

$$\frac{y^2}{2} \cdot \left[\frac{1}{|x+1|} + \frac{1}{|x-1|}\right] \cdot \mathtt{pred}(1) = \frac{y^2}{1-x^2} \cdot \mathtt{pred}(1) \leq v_1 + v_2 = V(x,y); \tag{B.56}$$

Für $V(x,y)$ gilt dann zusammen mit (B.52) für $\mathtt{prec} \geq 2$ die garantierte Einschließung

$$\mathtt{pred}(1) \cdot \frac{y^2}{1-x^2} \leq V(x,y) \leq \frac{y^2}{1-x^2}, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}. \tag{B.57}$$

Damit haben wir für hinreichend kleine Werte von $|y|$ eine schon recht einfache Einschließung für $V(x,y)$ gefunden. Um eine Einschließung auch für $\operatorname{arcosh}(1+\delta)$ zu erhalten, betrachten wir die Reihenentwicklung in (B.44) auf Seite 170. Im Buch zur Fehlerabschätzung wird gezeigt, dass $Q(t)$ für $0 \leq t < 2$ eine alternierende Leibniz-Reihe ist, so dass sich für $0 \leq \delta < 2$ die folgende Abschätzung ergibt:

$$\sqrt{2\delta} \cdot \left(1 - \frac{\delta}{12}\right) \leq \operatorname{arcosh}(1+\delta) \leq \sqrt{2\delta}, \quad \delta = V(x,y)/2 < 2 \quad \rightsquigarrow \tag{B.58}$$

$$\sqrt{V(x,y)} \cdot \left(1 - \frac{V(x,y)}{24}\right) \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \sqrt{V(x,y)}, \quad \text{falls } V(x,y) < 4.$$

Wir stellen jetzt noch zusätzlich die Bedingung

$$1 - \frac{V}{24} > \mathtt{pred}(1) = 1 - 2^{-\mathtt{prec}}, \quad \text{d.h. } V(x,y) < 3 \cdot 2^{-\mathtt{prec}+3}$$

und erhalten damit:

$$\sqrt{V} \cdot \mathtt{pred}(1) \leq \operatorname{arcosh}(1 + V/2) \leq \sqrt{V}, \quad \text{falls } V(x,y) < 3 \cdot 2^{-\mathtt{prec}+3}. \tag{B.59}$$

Mit (B.57) und (B.59) kann jetzt unter der Voraussetzung $|y| < 2^{-(3\mathtt{prec})/2}$ für $\mathtt{prec} \geq 2$ eine sehr enge Einschließung für $\operatorname{arcosh}(1 + V/2)$ angegeben werden[9]:

$$\mathtt{pred}(1) \cdot \sqrt{\mathtt{pred}(1)} \cdot \frac{|y|}{\sqrt{1-x^2}} \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \frac{|y|}{\sqrt{1-x^2}}. \tag{B.60}$$

[9] Man kann mit (B.57) leicht zeigen, dass unter der Voraussetzung $|y| < 2^{-(3\mathtt{prec})/2}$ neben $V(x,y) < 4$ auch die Bedingung $V(x,y) < 3 \cdot 2^{-\mathtt{prec}+3}$ für alle $\mathtt{prec} \geq 2$ erfüllt ist.

Wegen $\sqrt{\texttt{pred}(1)} > \texttt{pred}(1)$ gilt im Fall $|y| < 2^{-(3\texttt{prec})/2}$ für $\texttt{prec} \geq 2$ auch die etwas gröbere Einschließung:

$$\texttt{pred}(1) \cdot \texttt{pred}(1) \cdot \frac{|y|}{\sqrt{1-x^2}} \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \frac{|y|}{\sqrt{1-x^2}}. \tag{B.61}$$

Beachten Sie, dass jetzt für $|x| < 1$ bei der Auswertung von $|y|/\sqrt{1-x^2}$ ein vorzeitiger Unterlauf nicht mehr eintreten kann, wobei $\sqrt{1-x^2}$ mit Hilfe von `sqrt_1mx2(x)` intervallmäßig auszuwerten ist. Ein Überlauf kann bei der Auswertung von $|y|/\sqrt{1-x^2}$ für $|x| < 1$ nur eintreten, wenn die Präzision sehr groß und damit $\sqrt{1-x^2}$ sehr klein wird. Diese sehr großen Präzisionen haben aber wegen der damit verbundenen extremen Laufzeiten keinerlei praktische Bedeutung!

Um für $\operatorname{arcosh}(1+V(x,y)/2)$ eine möglichst einfach auszuwertende Einschließung angeben zu können, beweisen wir vorher noch den folgenden Satz:

Für alle Maschinenzahlen $x \geq 0$ des MPFR-Formats gilt:

$$\texttt{pred}(1) \cdot x \geq \texttt{pred}(x). \tag{B.62}$$

Der **Beweis** für $x = 0$ und $x = \texttt{minfloat()}$ ist trivial. Wir betrachten jetzt positive, normalisierte Zahlen $x = m \cdot 2^{ex} > \texttt{minfloat()}$, mit $0.5 \leq m < 1$. Nach (3.1) auf Seite 15 gilt:

$$\begin{aligned} m = 0.5 \quad &\rightsquigarrow \quad \texttt{pred}(x) = 2^{ex-1} \cdot (1 - 2^{-\texttt{prec}}) \\ 0.5 < m < 1 \quad &\rightsquigarrow \quad \texttt{pred}(x) = 2^{ex} \cdot (\texttt{m} - 2^{-\texttt{prec}}). \end{aligned}$$

Zum Beweis von (B.62) sind damit zwei Fälle zu unterscheiden. Für $m = 0.5$ kann (B.62) wie folgt äquivalent umgeformt werden:

$$\begin{aligned} & \texttt{pred}(1) \cdot x \geq \texttt{pred}(x) \\ \iff \quad & \texttt{pred}(1) \cdot 0.5 \cdot 2^{\texttt{ex}} \geq 2^{\texttt{ex}-1} \cdot (1 - 2^{-\texttt{prec}}), \\ \iff \quad & (1 - 2^{-\texttt{prec}}) \cdot 2^{\texttt{ex}-1} \geq 2^{\texttt{ex}-1} \cdot (1 - 2^{-\texttt{prec}}). \end{aligned}$$

Für $0.5 < m < 1$ kann (B.62) wie folgt äquivalent umgeformt werden:

$$\begin{aligned} & \texttt{pred}(1) \cdot x \geq \texttt{pred}(x) \\ \iff \quad & \texttt{pred}(1) \cdot \texttt{m} \cdot 2^{\texttt{ex}} \geq 2^{\texttt{ex}} \cdot (\texttt{m} - 2^{-\texttt{prec}}) \\ \iff \quad & (1 - 2^{-\texttt{prec}}) \cdot \texttt{m} \cdot 2^{\texttt{ex}} \geq 2^{\texttt{ex}} \cdot (\texttt{m} - 2^{-\texttt{prec}}) \\ \iff \quad & (1 - 2^{-\texttt{prec}}) \cdot \texttt{m} \geq \texttt{m} - 2^{-\texttt{prec}} \\ \iff \quad & -2^{-\texttt{prec}} \geq -2^{-\texttt{prec}} \quad \blacksquare \end{aligned}$$

Um jetzt mit (B.61) unter der Voraussetzung $|y| < 2^{-(3\texttt{prec})/2}$ eine Einschließung des Ausdrucks $\operatorname{arcosh}(1+\delta) = \operatorname{arcosh}(1 + V(x,y)/2)$ zu erhalten, berechnen wir zunächst eine intervallmäßige Einschlieung des Quotienten

$$|y|/\sqrt{1-x^2} \in \texttt{abs}(y) \diamond \texttt{sqrt1mx2(x)} = [\texttt{u1}, \texttt{u2}],$$

wobei das Argument $\texttt{x} = [x, x]$ in der Funktion `sqrt1mx2(x)` als Punktintervall aufzufassen ist. Mit `u2` erhalten wir damit auch eine Oberschranke von $A := \operatorname{arcosh}(1 + V(x,y)/2) \leq \texttt{u2}$.

Um für A eine Unterschranke zu berechnen, gilt mit dem Satz (B.62)

$$\begin{aligned} \texttt{pred}(1) \cdot \{\texttt{pred}(1) \cdot |y|/\sqrt{x^2-1}\} \quad &\geq \quad \texttt{pred}(1) \cdot \{\texttt{pred}(1) \cdot \texttt{u1}\} \geq \texttt{pred}(1) \cdot \{\texttt{pred(u1)}\} \\ &\geq \quad \texttt{pred}\{\texttt{pred(u1)}\}. \end{aligned}$$

Unter der Voraussetzung $|y| < 2^{-(3\mathtt{prec})/2}$ erhalten wir schließlich mit (B.61) für $\mathtt{prec} \geq 2$ die sehr effektive Einschließung

$$\mathtt{pred}\,\{\mathtt{pred}(\mathtt{u1})\} \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \mathtt{u2}. \tag{B.63}$$

Mit $y = m \cdot 2^{ex}$, $ex = \mathtt{expo}(y)$ und $0.5 \leq m < 1$ ist die Voraussetzung $|y| < 2^{-(3\mathtt{prec})/2}$ erfüllt, wenn gilt: $ex < -(3 \cdot \mathtt{prec})/2$, dabei ist die Maschinenrundung zur Null bei der integer-Division $(3 \cdot \mathtt{prec})/2$ schon berücksichtigt. Die Auswertung von (B.63) erfolgt in der Hilfsfunktion

```
MpfiClass ACOSH_p1(const MpfiClass& x, const MpfiClass& y);
```

die nur für Punktintervalle auszuwerten ist.

Die beiden nächsten numerischen Beispiele zeigen, wie genau die Einschließungen mit (B.63) im Vergleich zu (B.58) ausfallen. Dazu wählen wir die Current-Präzision $\mathtt{prec} = 70$. Mit den beiden Maschinenzahlen $x_1 = \mathtt{pred}(1) = 1 - 2^{-70}$ und $y_1 = \mathtt{minfloat()} = 2^{-1073741824}$ ist das Eingangsinterval gegeben durch das Punktintervall $Z = [x_1, x_1] + i \cdot [y_1, y_1]$.

Mit (B.58) erhält man für den Imaginärteil die praktisch unbrauchbare Einschließung

$$\begin{aligned}\mathtt{asin}(Z) \;=\; &([1.57079632675373758748, 1.57079632675373758749],\\ &[0.00000000000000000000, 8.45440400895524581723e-161614249]).\end{aligned}$$

Dagegen erhält man mit (B.63) für den Imaginärteil die fast optimale Einschließung

$$\begin{aligned}\mathtt{asin}(Z) \;=\; &([1.57079632675373758748, 1.57079632675373758749],\\ &[5.78868064589607735286e-323228487, 5.78868064589607735289e-323228487]).\end{aligned}$$

B.2.9 $\arccos(z)$

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{(-\infty, -1) \cup (+1, +\infty)\}$ liefert die Funktion

```
MpfciClass acos(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `acos(Z)` für die komplexen Funktionswerte $\arccos(z)$, mit $z \in Z$.

$$\{\arccos(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{acos(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der reellen Achse von -1 bis $-\infty$ bzw. von $+1$ bis $+\infty$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

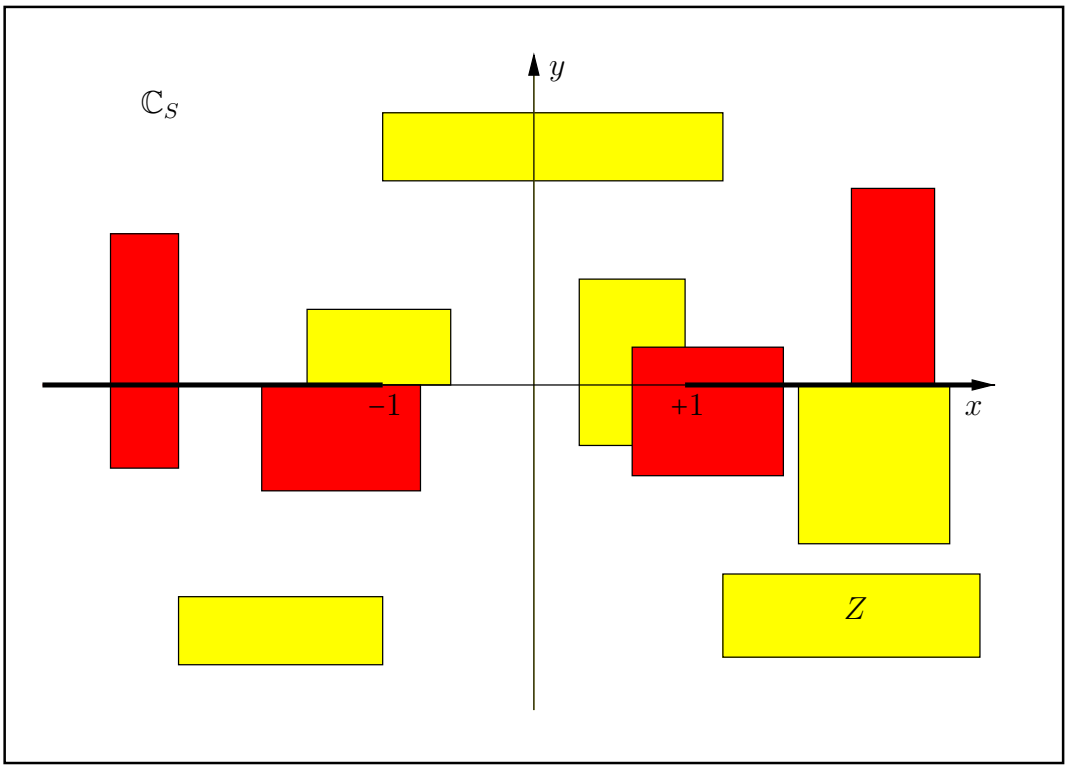


Abbildung B.16: Erlaubte und nicht erlaubte Intervalle Z von `acos(Z)`.

Mit $Z = [1,1] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[10]

$$\begin{aligned}\texttt{acos(Z)} \;=\; &([4.8811524304081624052 0e{-}161614249, 4.8811524304081624052 2e{-}161614249],\\ &[-4.8811524304081624052 2e{-}161614249, -4.8811524304081624052 0e{-}161614249]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{acos(Z)} \;=\; &([0.00000000000000000000000000, 1.2213572639376833256039 2],\\ &[0.00000000000000000000000000, 1.2604751877984540728529 2]).\end{aligned}$$

[10] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

B.2.9.1 Algorithmus

Der in [43] beschriebene Algorithmus verwendet die folgende Identität:

$$\arccos(z) \equiv \frac{\pi}{2} - \arcsin(z), \quad z = x + i \cdot y; \tag{B.64}$$

Mit $w = \arccos(z) = \Re(w) + i \cdot \Im(w)$ wird $\Im(w)$ daher über den inversen Sinus optimal berechnet. Im Falle $\Re(w) \sim 0$ bzw. $\Re(\arcsin(z)) \sim \pi/2$ wird jedoch $\Re(w)$ nach (B.64) wegen starker Auslöschung sehr fehlerhaft berechnet, so dass $\Re(w)$ damit nur grob eingeschlossen werden kann. Als Beispiel betrachten wir $z = 1 + i \cdot 2^{-100}$. Für das entsprechende Punktintervall $\boldsymbol{z}$ erhalten wir mit (B.64) nur die sehr grobe Einschließung

$$\arccos(\boldsymbol{z}) \subset ([-8.437695E{-}015, 2.980234E{-}008], [-2.980233E{-}008, -0.000000E{+}000])$$

während z.B. *Maple7* das folgende Ergebnis liefert:

$$\arccos(z) = 8.881784197001\ldots \cdot 10^{-16} - i \cdot 8.881784197001\ldots \cdot 10^{-16}$$

Um diese Überschätzung zu vermeiden, wird wie in [28] bei vorgegebenem komplexen Argumentintervall

$$Z = X + i \cdot Y = [x_1, x_2] + i \cdot [y_1, y_2], \quad x_i, y_i \text{ sind Maschinenzahlen}$$

zur Einschließung von $\Re(\arccos(Z))$ die reelle arccos-Funktion für den Intervall-Ausdruck $\boldsymbol{\beta}$ ausgewertet:

$$\arccos(\boldsymbol{\beta}), \quad \boldsymbol{\beta} := \frac{2\boldsymbol{x}}{\sqrt{(\boldsymbol{x}+1)^2 + \boldsymbol{y}^2} + \sqrt{(\boldsymbol{x}-1)^2 + \boldsymbol{y}^2}}, \tag{B.65}$$

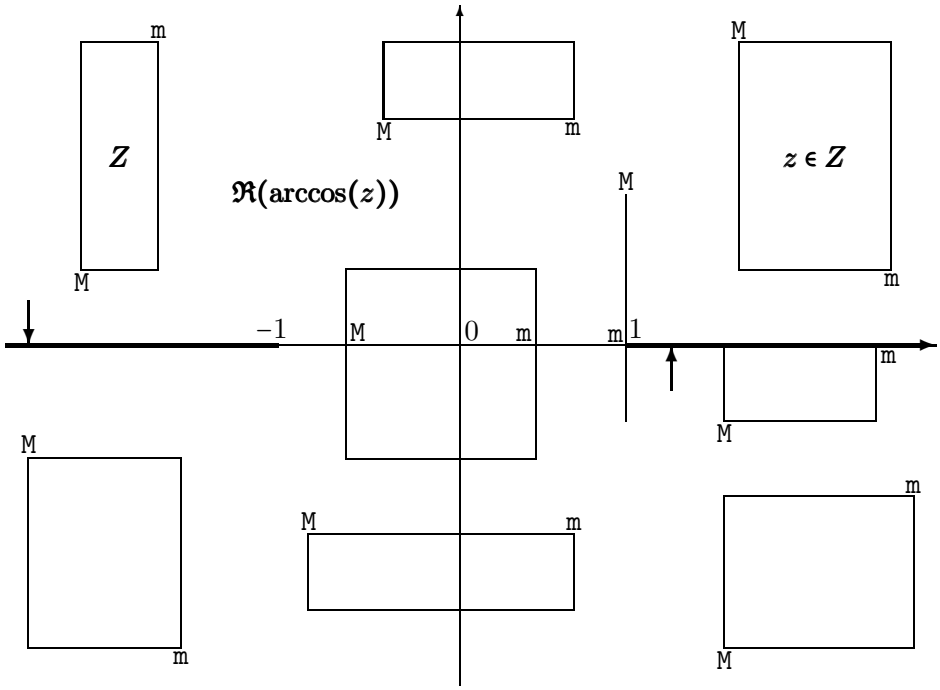
wobei die Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ durch die Extremalpunkte auf dem Rand von Z bestimmt sind, vgl. [28, S. 135].

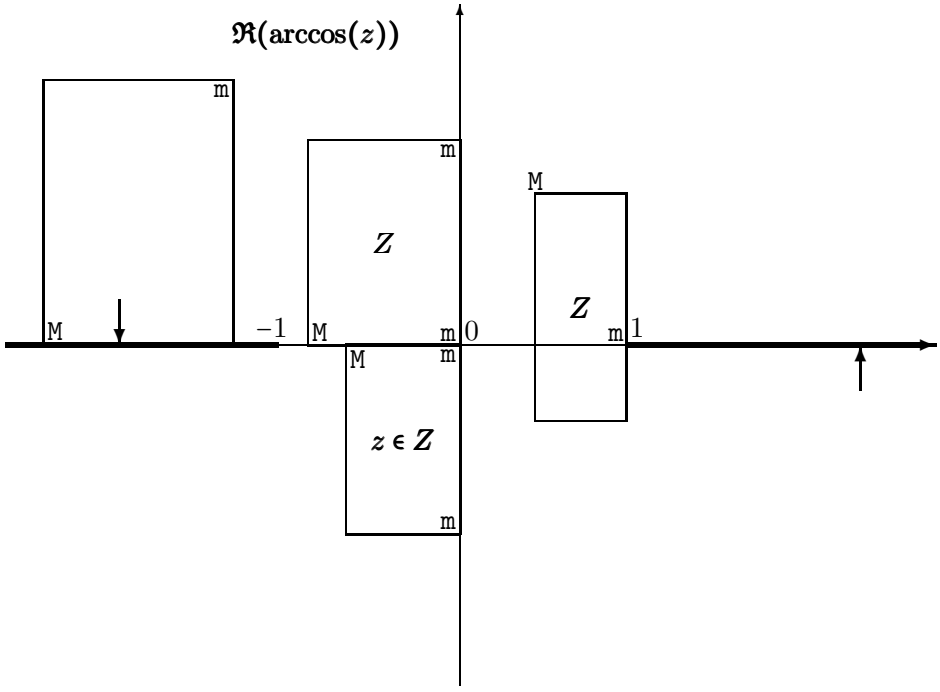
In den folgenden Abbildungen sind zu einigen Intervallen Z, die in der komplexen Ebene als Rechtecke dargestellt werden, die Extremalpunkte angegeben. Dabei sind die Koordinaten von `m` die Koordinaten des Extremalpunktes, an dem das Minimum von $\Re(w)$ angenommen wird. Entsprechend sind die Koordinaten 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 Z z.B. in der rechten Halbebene liegt und die reelle Achse durch das Innere von Z läuft. Dann liegt `M` auf demjenigen linken Eckpunkt von $\boldsymbol{Z}$, der die betragsmäßig maximale y-Koordinate besitzt. Liegt jedoch Z in der linken Halbebene und läuft die reelle Achse wieder durch das Innere von Z, dann liegt `m` auf demjenigen rechten Eckpunkt von 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 $\boldsymbol{Z}$ 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 folgenden Abbildungen sind einige mögliche Lagen der Intervalle Z angegeben. Die Pfeile auf die Verzweigungsschnitte beschreiben, aus welcher Richtung die Funktionswerte des Hauptzweiges auf die Schnitte analytisch fortzusetzen sind.





Nach Seite 167 gilt $\texttt{Sup}(\beta) \leq 1$. Für $\texttt{Inf}(\beta) \to 1$ wird daher die reelle arccos-Funktion in der Nähe ihrer Nullstelle $x_0 = +1$ ausgewertet, so dass bei der Einschließung von $\arccos(\beta)$ große Überschätzungen zu erwarten sind. Mit dem komplexen Argument $z = 1 + i \cdot 2^{-100}$ erhält man in der Tat bei Anwendung von (B.65) für den Realteil die sehr grobe Einschließung:

$$\arccos(z) \in ([0.000000E{+}000, 2.980233E{-}008], [-8.881785E{-}016, -8.881784E{-}016])$$

Diese Überschätzungen lassen sich vermeiden, wenn man mit Hilfe der Transformation

$$\beta := \frac{2x}{\sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2}} = 1 - \delta, \quad \delta \geq 0 \tag{B.66}$$

im Falle $\beta \geq 0.75$ für $\arccos(\beta)$ die folgende Darstellung benutzt:

$$\arccos(\beta) = \arccos(1-\delta) = \arcsin(\sqrt{2\delta - \delta^2}) \tag{B.67}$$

Bevor wir für die Fälle: $x < 1,\ x = 1,\ x > 1$ zur Auswertung von $\delta(x,y)$ die jeweils geeigneten Terme angeben, soll zunächst in der komplexen Ebene der Bereich bestimmt werden, in dem $\beta \geq 0.75$ gilt. Aus (B.66) 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 (B.66) erhält man für diese Maximumwerte direkt:

$$\beta(x,0) = \begin{cases} x & \text{falls } \ 0 \leq x \leq 1 \\ 1 & \text{falls } \ x > 1 \end{cases} \quad \rightsquigarrow \quad \delta = 0; \tag{B.68}$$

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 $\beta(x,y) \geq 0.75$ gilt, innerhalb der Höhenlinie

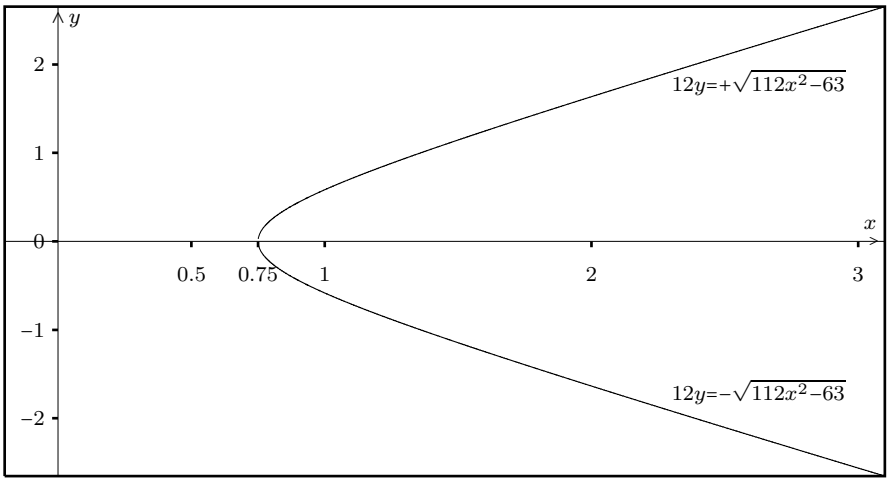
$$\beta(x,y) = \frac{3}{4} \quad \Longleftrightarrow \quad |y| = \frac{\sqrt{112\,x^2 - 63}}{12};$$

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 $\beta(x,y) \geq 0.75$ ist in der folgenden Abbildung innerhalb dieses Hyperbelastes dargestellt:

Bereich in der komplexen Ebene bez. $\beta \geq 0.75$



Um (B.67) im obigen Bereich $\beta(x,y) \geq 0.75$ anwenden zu können, benötigt man zunächst eine Darstellung für $\delta = \delta(x,y)$. Man findet direkt

$$2\delta = 2 - \sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2} \geq 0. \tag{B.69}$$

Aber (B.69) ist offensichtlich für $\delta \to 0$ wegen starker Auslöschung zur Auswertung auf der Maschine nicht geeignet. Wir betrachten daher unter der Voraussetzung $\beta \geq 0.75$ zunächst den Fall $\boldsymbol{x > 1}$:

Wegen $y = 0 \;\rightsquigarrow\; \delta = 0$, vgl. (B.68), kann man sich jetzt auf $y \neq 0$ beschränken. Aus (B.69) ergibt sich zunächst:

$$2\delta = -(x+1)\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\} + (x-1)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right\}$$

Die beiden obigen Summanden liefern jedoch für $x \to +\infty$ ebenfalls starke Auslöschung, so dass weitere Umformungen nötig sind:

$$\begin{aligned}
& -x\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\} + x\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right\} \\
= \;& -x\cdot\sqrt{1+\left(\frac{y}{x+1}\right)^2} + x\cdot\sqrt{1+\left(\frac{y}{x-1}\right)^2} \\
= \;& x\cdot\frac{1+\left(\frac{y}{x-1}\right)^2 - 1 - \left(\frac{y}{x+1}\right)^2}{\sqrt{1+\left(\frac{y}{x+1}\right)^2}+\sqrt{1+\left(\frac{y}{x-1}\right)^2}} \\
= \;& \frac{4y^2\cdot x^2}{(x^2-1)^2\cdot N}, \qquad N := \sqrt{1+\left(\frac{y}{x+1}\right)^2}+\sqrt{1+\left(\frac{y}{x-1}\right)^2};
\end{aligned}$$

Für 2δ erhält man damit:

$$\begin{aligned}
2\delta &= \frac{4y^2\cdot x^2}{(x^2-1)^2\cdot N} + 2 - N = \frac{4y^2\cdot x^2 + (2-N)(x^2-1)^2\cdot N}{(x^2-1)^2\cdot N} \\
&= \frac{Z}{(x^2-1)^2\cdot N}, \quad Z = 4y^2\cdot x^2 + (2-N)(x^2-1)^2\cdot N;
\end{aligned}$$

$$Z = -2\cdot(x^2-1)\cdot F;$$

$$\begin{aligned}
F &= x^2 - y^2 - 1 + \sqrt{1+\left(\frac{y}{x+1}\right)^2}\cdot\sqrt{1+\left(\frac{y}{x-1}\right)^2}\cdot(x^2-1) - \\
&\quad -\left(\sqrt{1+\left(\frac{y}{x+1}\right)^2}+\sqrt{1+\left(\frac{y}{x-1}\right)^2}\right)\cdot(x^2-1) \\
&= (x^2-1)\cdot\left(\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right)\cdot\left(\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right) - y^2
\end{aligned}$$

Für $\delta = \delta(x,y)$ ergibt sich damit schließlich im Fall $x > 1$:

$$2\delta = \frac{\dfrac{y^2}{x^2-1} - \left[\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right]\cdot\left[\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right]}{1+\dfrac{1}{2}\cdot\left(\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right)+\dfrac{1}{2}\cdot\left(\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right)}. \tag{B.70}$$

Der Nenner in (B.70) lässt sich zwar noch weiter vereinfachen zu

$$\frac{1}{2}\cdot\sqrt{1+\left(\frac{y}{x-1}\right)^2}+\frac{1}{2}\cdot\sqrt{1+\left(\frac{y}{x+1}\right)^2},$$

die Darstellung in (B.70) hat jedoch den Vorteil, dass zur Berechnung von $\delta(x,y)$ im Wesentlichen nur zwei Terme mit Hilfe der Funktion `sqrtp1m1()` auszuwerten sind. Außerdem kann 2δ für $y \to 0$ mit Hilfe von (B.70) sehr einfach eingeschlossen werden. Für eine Oberschranke erhält man aus (B.70) direkt:

$$2\cdot\delta(x,y) \le \frac{y^2}{x^2-1}. \tag{B.71}$$

Die Berechnung einer Unterschranke für $2\cdot\delta(x,y)$ ist etwas komplizierter. Nach (B.70) schreiben wir $2\delta = Z/N$ und bestimmen zunächst eine Oberschranke des Nenners N. Mit Hilfe der Reihenentwicklung für $\sqrt{1+t}-1$ von Seite 171 gelten die Abschätzungen:

$$\begin{aligned}
\frac{1}{2}\cdot\left(\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right) &\le \frac{1}{4}\cdot\left(\frac{y}{x+1}\right)^2, \quad \text{falls } \frac{|y|}{x+1}<1\\
\frac{1}{2}\cdot\left(\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right) &\le \frac{1}{4}\cdot\left(\frac{y}{x-1}\right)^2, \quad \text{falls } \frac{|y|}{x-1}<1.
\end{aligned}$$

Für den Nenner N folgt damit:

$$\begin{aligned}
N &\le 1+\frac{y^2}{4}\cdot\left[\frac{1}{(x+1)^2}+\frac{1}{(x-1)^2}\right]\\
&= 1+\frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2}, \quad \text{falls } \frac{|y|}{x-1}<1.
\end{aligned}$$

Beachten Sie bitte, dass wegen $x > 1$ die Bedingung $|y| < x+1$ automatisch erfüllt ist, falls $|y| < x-1$ gilt. Für den Kehrwert von N erhalten wir:

$$\frac{1}{N} \ge \frac{1}{1+\dfrac{y^2}{2}\cdot\dfrac{x^2+1}{(x^2-1)^2}}, \quad \text{falls } \frac{|y|}{x-1}<1.$$

Da die Reihe

$$\frac{1}{1+\varepsilon} = 1-\varepsilon+\varepsilon^2-\varepsilon^3\pm\ldots$$

für $0 \le \varepsilon < 1$ eine alternierende Leibniz-Reihe ist, folgt direkt:

$$\frac{1}{N} \ge 1-\frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2}, \quad \text{falls } \frac{|y|}{x-1}<1 \ \wedge\ \frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2}<1. \tag{B.72}$$

Wir zeigen jetzt, dass wegen unserer Voraussetzung $x > 1$ mit $|y| < x-1$ die obige zweite Bedingung automatisch erfüllt ist. Es gilt

$$\begin{aligned}
&\frac{x^2+1}{(x+1)^2} < \frac{x^2+1}{x^2} = 1+\frac{1}{x^2} < 2 \quad \text{und damit:}\\
&\frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2} = \frac{y^2}{2(x-1)^2}\cdot\frac{x^2+1}{(x+1)^2} < \frac{y^2}{(x-1)^2},
\end{aligned}$$

d.h. die obige zweite Bedingung ist erfüllt, wenn

$$\frac{y^2}{(x-1)^2}<1 \quad\Longleftrightarrow\quad \frac{|y|}{x-1}<1,$$

und die letzte Ungleichung ist gerade die erste Bedingung in (B.72). Es gilt damit:

$$\frac{1}{N} \geq 1 - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2}, \quad \text{falls} \quad \frac{|y|}{x-1} < 1. \tag{B.73}$$

Wir müssen jetzt noch eine Unterschranke des Zählers Z in (B.70) bestimmen. Zunächst gilt wieder die Abschätzung:

$$\begin{aligned} \big[...\big] \cdot \big[...\big] &\leq \frac{1}{4} \cdot \left(\frac{y}{x-1}\right)^2 \cdot \left(\frac{y}{x+1}\right)^2 \\ &= \frac{1}{4} \cdot \frac{y^4}{(x^2-1)^2}, \quad \text{falls} \quad \frac{|y|}{x-1} < 1, \end{aligned}$$

und damit erhalten wir:

$$\begin{aligned} Z &\geq \frac{y^2}{x^2-1} - \frac{1}{4} \cdot \frac{y^4}{(x^2-1)^2} \\ &= \frac{y^2}{x^2-1} \cdot \left[1 - \frac{y^2}{4 \cdot (x^2-1)}\right], \quad \text{falls} \quad \frac{|y|}{x-1} < 1. \end{aligned}$$

Zusammen mit (B.73) ergibt sich für $2 \cdot \delta(x,y)$ die Abschätzung:

$$2\delta \geq \frac{y^2}{x^2-1} \cdot \left\{1 - \frac{y^2}{4(x^2-1)}\right\} \cdot \left\{1 - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2}\right\}, \text{ falls } \frac{|y|}{x-1} < 1. \tag{B.74}$$

$$\begin{aligned} \big\{...\big\} \cdot \big\{...\big\} &\geq 1 - \frac{y^2}{4(x^2-1)} - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2} \\ &= 1 - \frac{y^2}{2(x^2-1)} \left(\frac{1}{2} + \frac{x^2+1}{x^2-1}\right). \end{aligned} \tag{B.75}$$

$$\begin{aligned} \left(\frac{1}{2} + \frac{x^2+1}{x^2-1}\right) &= \frac{3x^2+1}{2(x+1)} \cdot \frac{1}{x-1} \leq \frac{1}{x-1} \cdot \frac{3x^2+1}{2x} \\ &= \frac{1}{x-1} \cdot \left[\frac{3}{2} \cdot x + \frac{1}{2x}\right] \leq \frac{1}{x-1} \cdot \left[\frac{3}{2} \cdot x + \frac{1}{2}\right] \\ &= \frac{3x+1}{2(x-1)} \qquad \rightsquigarrow \end{aligned}$$

$$\big\{...\big\} \cdot \big\{...\big\} \geq 1 - \frac{y^2}{4(x-1)^2} \cdot \frac{3x+1}{x+1}.$$

Wegen $x > 1$ gilt noch

$$\begin{aligned} \frac{3x+1}{x+1} \leq \frac{3x+1}{x} = 3 + \frac{1}{x} < 4 \quad \text{und damit} \\ \big\{...\big\} \cdot \big\{...\big\} \geq 1 - \frac{y^2}{(x-1)^2}. \end{aligned}$$

Zusammen mit (B.71) und (B.74) erhalten wir unter der Voraussetzung $x > 1$ die Einschließung:

$$\left[1 - \frac{y^2}{(x-1)^2}\right] \cdot \frac{y^2}{x^2-1} \leq 2 \cdot \delta(x,y) \leq \frac{y^2}{x^2-1}, \quad \text{falls} \quad \frac{|y|}{x-1} < 1. \tag{B.76}$$

Wir verlangen zusätzlich noch

$$1 - \frac{y^2}{(x-1)^2} > \texttt{pred}(1) = 1 - 2^{-\texttt{prec}} \quad \Longleftrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\texttt{prec}}}$$

und erhalten damit im Falle $x > 1$ für $2 \cdot \delta(x, y)$ die gesuchte Einschließung:

$$\texttt{pred}(1) \cdot \frac{y^2}{x^2-1} \le 2 \cdot \delta(x,y) \le \frac{y^2}{x^2-1}, \quad \text{falls} \quad \frac{|y|}{x-1} < \sqrt{2^{-\texttt{prec}}}; \tag{B.77}$$

Mit Hilfe der gefundenen Doppelungleichung (B.77) kann man unter den Voraussetzungen $x > 1$ und $|y|/(x-1) \ll 1$ eine sehr effektive Einschließung für $2 \cdot \delta$ und damit auch für $\arccos(1-\delta) \approx \sqrt{2\delta}$ berechnen. Im nächsten Schritt wird eine Einschließung für $\arccos(1-\delta)$ angegeben. Nach Abramowitz gilt, [1, S. 81]:

$$\begin{aligned} \arccos(1-\delta) &= \sqrt{2\delta} \cdot \left[1 + \sum_{k=1}^{\infty} \frac{1 \cdot 3 \cdot 5 \cdots \cdot (2k-1)}{2^{2k}(2k+1)k!} \cdot \delta^k \right] \\ &= \sqrt{2\delta} \cdot \left[1 + \frac{1}{12}\delta + \frac{3}{160}\delta^2 + \frac{5}{896}\delta^3 + \ldots \right], \quad 0 \le \delta < 2; \end{aligned}$$

dabei sind die Taylorkoeffizienten bis aufs Vorzeichen identisch mit den Taylorkoeffizienten der Reihe für $\operatorname{arcosh}(1+\delta)$, so dass die vorliegende Reihe mit Hilfe der geometrischen Reihe abgeschätzt werden kann. Man findet unmittelbar:

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \frac{1}{1-\delta}, \quad 0 \le \delta < 1. \tag{B.78}$$

Wir verlangen noch zusätzlich

$$\frac{1}{1-\delta} < \texttt{succ}(1) = 1 + 2^{-\texttt{prec}+1} \quad \Longleftrightarrow \quad \delta < \frac{2^{-\texttt{prec}+1}}{1+2^{-\texttt{prec}+1}}$$

und die letzte Ungleichung ist erfüllt, wenn $\delta < 2^{-\texttt{prec}}$, d.h. es gilt

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \texttt{succ}(1), \quad \text{falls} \quad \delta < 2^{-\texttt{prec}}. \tag{B.79}$$

Zusammen mit (B.77) ergibt sich daraus:

$$\sqrt{\texttt{pred}(1)} \cdot \frac{|y|}{\sqrt{x^2-1}} \le \arccos(1-\delta) \le \frac{|y|}{\sqrt{x^2-1}} \cdot \texttt{succ}(1). \tag{B.80}$$

Die bisherigen Bedingungen für (B.80) lauten:

$$x > 1 \tag{B.81}$$

$$\frac{|y|}{x-1} < \sqrt{2^{-\texttt{prec}}} \tag{B.82}$$

$$\delta < 2^{-\texttt{prec}}. \tag{B.83}$$

Wir zeigen jetzt, dass mit (B.81) und (B.82) die Bedingung (B.83) automatisch erfüllt ist. Nach (B.77) gilt mit (B.82) die Abschätzung:

$$\begin{aligned} \delta &\le \frac{y^2}{2(x^2-1)}, \quad \text{und damit} \\ \delta &\le \frac{y^2}{(x-1)^2} \cdot \frac{x-1}{2(x+1)}, \quad \text{und wegen} \quad \frac{x-1}{2(x+1)} < \frac{1}{2} \quad \text{folgt} \\ \delta &< 2^{-\texttt{prec}} \cdot \frac{1}{2} = 2^{-\texttt{prec}-1} < 2^{-\texttt{prec}} \ \blacksquare \end{aligned}$$

Für $\arccos(1-\delta)$ gilt damit im Falle $x > 1$ und $|y|/(x-1) < \sqrt{2^{-\texttt{prec}}}$ die Einschließung

$$\sqrt{\texttt{pred}(1)} \cdot \frac{|y|}{\sqrt{x^2-1}} \le \arccos(1-\delta) \le \frac{|y|}{\sqrt{x^2-1}} \cdot \texttt{succ}(1). \tag{B.84}$$

Um für $\arccos(1-\delta)$ eine möglichst einfach auszuwertende Einschließung angeben zu können, beweisen wir jetzt noch den folgenden **Satz**:

Für alle Maschinenzahlen $x \geq 0$ gilt:

$$x \cdot \mathtt{succ}(1) \leq \mathtt{succ}(\mathtt{succ}((x))). \tag{B.85}$$

Der **Beweis** für $x = 0$ ist trivial. Wir betrachten daher jetzt positive Zahlen $x = m \cdot 2^{ex}$, mit $0.5 \leq m < 1$. Nach (3.2) von Seite 15 gilt $\mathtt{succ}(x) = x + 2^{ex-\mathtt{prec}}$ und daraus folgt direkt:

$$\begin{aligned}
x \cdot \mathtt{succ}(1) \leq \mathtt{succ}(\mathtt{succ}((x)) \quad &\Longleftrightarrow \quad \mathtt{m} \cdot 2^{\mathtt{ex}} \cdot (1 + 2^{1-\mathtt{prec}}) \leq \mathtt{succ}[x + 2^{-\mathtt{prec}+ex}] = \\
&= \quad x + 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]}. \text{ Zu zeigen ist also} \\
&\quad x \cdot (1 + 2^{1-\mathtt{prec}}) \leq x + 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad x \cdot 2^{1-\mathtt{prec}} \leq 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad \mathtt{m} \cdot 2^{ex-\mathtt{prec}+1} \leq 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad \mathtt{m} \cdot 2^{ex+1} \leq 2^{ex} + 2^{\mathtt{expo}[\ldots]}.
\end{aligned}$$

Mit den zusätzlichen Abschätzungen: $\mathtt{m} \cdot 2^{ex+1} < 2^{ex+1}$ und $2^{ex} + 2^{\mathtt{expo}[\ldots]} \geq 2^{ex} + 2^{ex} = 2^{ex} + 1$ bleibt also zu zeigen $2^{ex+1} \leq 2^{ex+1}$ ∎

Man beweist sehr einfach $\sqrt{\mathtt{pred}(1)} > \mathtt{pred}(1)$, und mit dem **Satz**

$$\mathtt{pred}(1) \cdot x \geq \mathtt{pred}(x), \quad \text{falls } x \geq 0$$

von Seite 174 folgt zusammen mit (B.84) zur Einschließung von $\arccos(1-\delta)$ unter der Bedingung $|y|/(x-1) < \sqrt{2^{-\mathtt{prec}}}$ der folgende sehr einfache **Algorithmus**:

- Berechne im Falle $|y|/(x-1) < \sqrt{2^{-\mathtt{prec}}}$ und $x > 1$ mit `u = abs(y)/sqrtx2m1(x)` eine garantierte Einschließung für: $|y|/\sqrt{x^2-1} \subseteq$ `u := [u1,u2]`;
- Es gilt dann: $\mathtt{pred}(\mathtt{u1}) \leq \arccos(1-\delta) \leq \mathtt{succ}(\mathtt{succ}(\mathtt{u2}))$.

Beachten Sie, dass bei Anwendung der Funktion `sqrtx2m1(....)` ein vorzeitiger Overflow bei der Berechnung von $\sqrt{x^2-1}$ nicht eintreten kann.

Die Bedingung $|y|/(x-1) < \sqrt{2^{-\mathtt{prec}}}$ wird im Programm wie folgt realisiert:
Zunächst wird mit `tm = y/(x-1)` der Quotient $y/(x-1)$ intervallmäßig ausgewertet[11], d.h. es gilt: $y/(x-1) \in$ `tm`. Verlangt man danach `expo(Sup(tm)) <= -prec ⊘ 2 - 1`, so ist die Bedingung $|y|/(x-1) < \sqrt{2^{-\mathtt{prec}}}$ erfüllt. $\oslash$ bedeutet dabei die zur Null gerundete integer-Division durch 2, wenn die Current-Präzision $\mathtt{prec} \geq 2$ ungerade ist.

Für $\mathtt{prec} \geq 2$ und $x > 1$ müssen wir also noch beweisen:

$$\mathtt{expo(Sup(tm))} <= -\mathtt{prec} \oslash 2 - 1 \quad \Longrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}. \tag{B.86}$$

Zum **Beweis** sei zunächst $y \geq 0$. Dann gilt $y/(x-1) \leq \mathtt{Sup(tm)}$ und die Behauptung rechts in (B.86) ist erfüllt, falls

$$\begin{aligned}
&\mathtt{Sup(tm)} < \sqrt{2^{-\mathtt{prec}}} \\
\Longleftrightarrow \quad &m \cdot 2^{ex} < \sqrt{2^{-\mathtt{prec}}}, \qquad 0.5 \leq m < 1, \quad ex := \mathtt{expo(Sup(tm))} \\
\Longleftrightarrow \quad &m^2 \cdot 2^{2ex} < 2^{-\mathtt{prec}},
\end{aligned}$$

und wegen $m^2 < 1$ ist die letzte Ungleichung erfüllt, falls $2^{2ex} < 2^{-\mathtt{prec}}$, bzw. falls $ex < -\mathtt{prec}/2$. Damit ist bewiesen:

$$\mathtt{expo(Sup(tm))} <= -\mathtt{prec}/2 \quad \Longrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}. \tag{B.87}$$

[11] `x,y` sind Punktintervalle, welche die Maschinenzahlen x, y einschließen.

Da auf der Maschine integer-Divisionen zur Null gerundet werden, gilt $-\texttt{prec}/2 \geq -\texttt{prec} \oslash 2 - 1$, womit dann (B.86) für $y \geq 0$ bewiesen ist.
Für den Rest des Beweises sei jetzt $y < 0$. Dann gilt $0 \geq \texttt{Sup(tm)} \geq y/(x-1)$ bzw.

$$-\texttt{Sup(tm)} \leq \frac{-y}{x-1} = \frac{|y|}{x-1}, \tag{B.88}$$

und die Behauptung rechts in (B.86) ist erfüllt, falls

$$\begin{aligned} & -\texttt{Sup(tm)} < \sqrt{2^{-\texttt{prec}}} \\ \iff\ & -(m \cdot 2^{ex}) < \sqrt{2^{-\texttt{prec}}}, \qquad -1 < m \leq -0.5, \quad ex := \texttt{expo(Sup(tm))} \\ \iff\ & m^2 \cdot 2^{2ex} < 2^{-\texttt{prec}}, \end{aligned}$$

und wegen $m^2 < 1$ ist die letzte Ungleichung erfüllt, falls $2^{2ex} < 2^{-\texttt{prec}}$, bzw. falls $ex < -\texttt{prec}/2$. Der Rest des Beweises verläuft dann wie auf Seite 184 ∎

Im Fall $x > 1$ und $|y|/(x-1) \geq \sqrt{2^{-\texttt{prec}}}$ wird δ nach (B.70) ausgewertet:

$$2\delta = \frac{\dfrac{y^2}{x^2-1} - \left[\sqrt{1+\left(\dfrac{y}{x-1}\right)^2} - 1\right] \cdot \left[\sqrt{1+\left(\dfrac{y}{x+1}\right)^2} - 1\right]}{1 + \dfrac{1}{2} \cdot \left(\sqrt{1+\left(\dfrac{y}{x-1}\right)^2} - 1\right) + \dfrac{1}{2} \cdot \left(\sqrt{1+\left(\dfrac{y}{x+1}\right)^2} - 1\right)} \tag{B.89}$$

Dabei entsteht sofort wieder die Frage, ob der Zähler in (B.89) als Differenz zweier positiver Größen ohne merkliche Auslöschung berechnet werden kann. Zur Beantwortung dieser Frage betrachten wir die für $y \neq 0$ äquivalenten Gleichungen

$$\frac{y^2}{x^2-1} - \left[\sqrt{1+\left(\frac{y}{x-1}\right)^2} - 1\right] \cdot \left[\sqrt{1+\left(\frac{y}{x+1}\right)^2} - 1\right] = r \cdot \frac{y^2}{x^2-1}$$

$$1 - r = \frac{1}{y^2} \cdot \left[\sqrt{(x-1)^2+y^2} - (x-1)\right] \cdot \left[\sqrt{(x+1)^2+y^2} - (x+1)\right]$$

und suchen im ganzen Bereich $\beta \geq 0.75$ im Falle $x > 1$ für $r = r(x,y)$ eine möglichst große Unterschranke bzw. für $1-r$ eine möglichst kleine Oberschranke. Lässt sich dies realisieren, so kann 2δ nach (B.89) ohne Auslöschung ausgewertet werden. Zur Berechnung einer möglichst kleinen Oberschranke von $1-r$ kann man sich wegen der Symmetrie zur reellen Achse, d.h. wegen $r(x,y) \equiv r(x,-y)$ auf $y > 0$ beschränken. Es gilt dann:

$$1 - r = \left[\sqrt{\left(\frac{x-1}{y}\right)^2 + 1} - \frac{x-1}{y}\right] \cdot \left[\sqrt{\left(\frac{x+1}{y}\right)^2 + 1} - \frac{x+1}{y}\right] \tag{B.90}$$

Im nächsten Schritt zeigen wir, dass $\partial(1-r)/\partial y$ für $y > 0$ positiv ist, so dass das Maximum von $1-r$ für $x \geq 1$ daher auf dem folgenden Hyperbelast liegt:

$$y = \frac{1}{12} \cdot \sqrt{112\,x^2 - 63} \tag{B.91}$$

Für $y > 0$ gilt mit

$$A := \sqrt{(x-1)^2+y^2} > x-1; \quad B := \sqrt{(x+1)^2+y^2} > x+1;$$

$$\frac{\partial(1-r)}{\partial y} = \frac{[A-(x-1)] \cdot [B-(x+1)] \cdot [B(x-1)+A(x+1)]}{y^3 \cdot A \cdot B}$$

und daraus ergibt sich für $x \geq 1$ die gewünschte Beziehung $\partial(1-r)/\partial y > 0$. Das gesuchte Maximum von $1-r(x,y)$ liegt daher auf dem Hyperbelast. Um dieses Maximum zu berechnen, setzen wir daher y aus (B.91) in (B.90) ein. Nach einigen Umrechnungen erhält man dann das erstaunlicher Ergebnis:

$$1 - r \equiv \frac{1}{7} = 0.142857\ldots, \quad \text{falls } x \geq 1, \tag{B.92}$$

d.h. auf dem Hyperbelast ist $1-r(x,y)$ eine konstante Funktion. Man zeigt nun leicht, dass das Maximum von $1-r(x,y)$ für $x = 1$ und $0 < y \leq 7/12$, d.h. also auf der Parallelen zur imaginären Achse durch $x = 1$ ebenfalls durch den Wert $1/7$ gegeben ist. Der Nachweis bleibt dem Leser überlassen. Für $1-r(x,y) \leq 1/7$ haben wir damit im Bereich $\beta \geq 0.75$ eine hinreichend kleine Oberschranke berechnet, so dass 2δ nach (B.89) trotz der Differenz im Zähler auf der Maschine stabil, d.h. ohne Auslöschung ausgewertet werden kann.

Wir betrachten jetzt den Fall $\boldsymbol{x = +1}$. Mit (B.69) von Seite 180 erhält man:

$$\delta = \left\{1 - \sqrt{1 + \left(\frac{y}{2}\right)^2}\right\} + \frac{|y|}{2}, \quad \text{falls } x = 1; \tag{B.93}$$

Wir zeigen zunächst, dass (B.93) im Bereich $\beta \geq 0.75$ nur für $|y| \leq 7/12$ auszuwerten ist. Im Falle $x = 1$ gilt nach (B.69)

$$\begin{aligned} \beta \quad &= \quad \frac{2}{\sqrt{4+y^2}+|y|} \geq \frac{3}{4} \\ &\iff \quad \sqrt{4+y^2}+|y| \leq \frac{8}{3} \quad \iff \quad \sqrt{4+y^2} = \frac{8}{3} - |y| \\ &\iff \quad |y| \leq \frac{7}{12} = 0.58333\ldots \ \blacksquare \end{aligned}$$

In (B.93) wird der erste Summand $\{.....\} \leq 0$ mit Hilfe der Funktion `sqrtp1m1()` ausgewertet, die Auslöschung innerhalb von $\{\ldots\}$ vermeidet. Man könnte jedoch einwenden, dass in (B.93) die Addition eines negativen und positiven Summanden zur Auslöschung führen kann. Um dies auszuschließen, zeigen wir mit $t := |y|/2$, dass in

$$t - \left\{\sqrt{1+t^2} - 1\right\} = r \cdot t \tag{B.94}$$

der Faktor $r < 1$ für $|y| \leq 7/12$ bzw. für $t \leq 7/24$ hinreichend groß ist. Mit

$$r(t) := 1 - \frac{1}{t} \cdot \left\{\sqrt{1+t^2} - 1\right\} \quad \text{folgt}$$

$$r'(t) = -\frac{\sqrt{1+t^2}-1}{t^2 \cdot \sqrt{1+t^2}} < 0$$

so dass $r(t)$ streng monoton fallend ist. Im Bereich $0 \leq t \leq 7/24$ gilt daher

$$r(t) \geq r(7/24) = \frac{6}{7} = 0.85714\ldots \tag{B.95}$$

und mit dieser Unterschranke für $r(t)$ folgt aus (B.94), dass für $|y| \leq 7/12$ bei der Auswertung von (B.93) keine wesentliche Auslöschung auftreten kann. Damit könnte man den Fall $x = 1$ als erledigt ansehen. Für $|y| \ll 1$ lässt sich die Auswertung nach (B.93) jedoch noch wesentlich vereinfachen. Mit $t := |y|/2$ folgt nach (B.93) zunächst $\delta = 1 + t - \sqrt{1+t^2}$, und damit ergibt sich ganz elementar die folgende Einschließung für δ:

$$t \cdot \left(1 - \frac{t}{2}\right) \leq \delta \leq t, \quad t := \frac{|y|}{2}. \tag{B.96}$$

Mit $0 \le \delta < 1$ gilt nach (B.78) für $\arccos(1-\delta)$ die Einschließung

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \frac{1}{1-\delta}, \quad 0 \le \delta < 1, \tag{B.97}$$

und zusammen mit (B.96) folgt direkt:

$$\sqrt{2t} \cdot \sqrt{1-\frac{t}{2}} \le \arccos(1-\delta) \le \frac{\sqrt{2t}}{1-t}, \quad \text{falls} \quad 0 \le t = \frac{|y|}{2} < 1. \tag{B.98}$$

Wir verlangen zusätzlich:

$$\begin{aligned} & \sqrt{1-\frac{t}{2}} \ge \mathtt{pred}(1) = 1 - 2^{-\mathtt{prec}} \\ \iff \quad & 1 - \frac{t}{2} \ge 1 - 2 \cdot 2^{-\mathtt{prec}} + 2^{-2\mathtt{prec}} \\ \iff \quad & |y| \le 2^{-\mathtt{prec}+3} - 2^{-2\mathtt{prec}+2} = 2^{-\mathtt{prec}}(8 - 2^{-\mathtt{prec}+2}), \end{aligned}$$

Wegen $|y| = m \cdot 2^{ex} < 2^{ex}$ und wegen $8 - 2^{-\mathtt{prec}+2} > 2^2$ ist die letzte Ungleichung erfüllt, wenn gilt: $\mathtt{expo}(y) = ex \le -\mathtt{prec} + 2$. Zusammen mit (B.98) erhält man:

$$\mathtt{pred}(1) \cdot \sqrt{|y|} \le \arccos(1-\delta) \le \frac{\sqrt{|y|}}{1-t}, \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 2. \tag{B.99}$$

Wie auf Seite 183 verlangen wir außerdem

$$\frac{1}{1-t} < \mathtt{succ}(1) = 1 + 2^{1-\mathtt{prec}} \quad \iff \quad t < \frac{2^{1-\mathtt{prec}}}{1 + 2^{1-\mathtt{prec}}},$$

und die letzte Ungleichung ist erfüllt, wenn $t < 2^{-\mathtt{prec}}$, bzw. wenn $|y| < 2^{-\mathtt{prec}+1}$ oder wenn gilt $\mathtt{expo}(y) \le -\mathtt{prec} + 1$. Mit (B.99) ergibt sich daraus die Einschließung:

$$\mathtt{pred}(1) \cdot \sqrt{|y|} \le \arccos(1-\delta) \le \sqrt{|y|} \cdot \mathtt{succ}(1), \quad \text{falls} \quad \mathtt{expo}(y) \le -\mathtt{prec} + 1. \tag{B.100}$$

Um die Einschließung in (B.100) auf der Maschine effizient realisieren zu können, berechnet man zu gegebener Maschinenzahl `y` mit `u = [u1,u2]` zunächst eine Einschließung für $\sqrt{|y|}$:

$$\mathtt{u} = \mathtt{sqrt(\ MpfiClass(abs(y))\ )} \quad \rightsquigarrow \quad \sqrt{|y|} \in \mathtt{[u1,u2]}.$$

Mit (B.100) erhält man daher

$$\mathtt{pred}(1) \cdot \mathtt{u1} \le \arccos(1-\delta) \le \mathtt{u2} \cdot \mathtt{succ}(1), \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 1; \tag{B.101}$$

Bei Anwendung von (B.62) und (B.85) kann man außerdem noch die beiden Multiplikationen in (B.101) vermeiden:

$$\mathtt{pred(u1)} \le \arccos(1-\delta) \le \mathtt{succ(succ(u2))}, \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 1. \tag{B.102}$$

(B.102) liefert damit unter den Voraussetzungen $\beta \ge 0.75$, $x = 1$ und $|y| < 2^{-\mathtt{prec}+1}$, bzw. $\mathtt{expo}(y) \le -\mathtt{prec} + 1$ einen sehr effektiven Algorithmus zur fast optimalen Einschließung von $\arccos(1-\delta)$. Für $\mathtt{expo}(y) > -\mathtt{prec} + 1$ benutzt man (B.93) zusammen mit der Beziehung $\arccos(1-\delta) \equiv \arcsin(\sqrt{\delta \cdot (2-\delta)}\,)$.

Wir müssen jetzt noch im Bereich $\beta \geq 0.75$ den letzten Fall $\mathbf{0.75 \leq x < +1}$ betrachten: Ausgehend von (B.69) auf Seite 180 findet man nach einigen Umrechnungen:

$$\begin{aligned} 2\delta &= -(x+1)\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\}+ \\ &\quad +(1-x)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right\}+2\cdot(1-x) \end{aligned}$$

$$\begin{aligned} 2\cdot\delta(x,y) &= -(x+1)\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\}+ \\ &\quad +(1-x)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}+1\right\}, \quad \text{falls } x<1; \end{aligned} \tag{B.103}$$

Da in (B.103) zwei Summanden mit unterschiedlichen Vorzeichen zu addieren sind, stellt sich auch jetzt wieder die Frage, ob diese Addition in $0.75 \leq x < +1$ ohne wesentliche Auslöschung durchführbar ist. Zur Beantwortung dieser Frage betrachten wir die beiden äquivalenten Gleichungen

$$\begin{aligned} &(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right]-(1+x)\cdot\left[\sqrt{1+\left(\frac{y}{1+x}\right)^2}-1\right] \\ &\qquad = r\cdot(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right] \end{aligned}$$

$$1-r=\frac{(1+x)\cdot\left[\sqrt{1+\left(\frac{y}{1+x}\right)^2}-1\right]}{(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right]} \tag{B.104}$$

und suchen im Bereich $\beta \geq 0.75$ im Falle $0.75 \leq x < 1$ für $r = r(x,y)$ eine möglichst große Unterschranke bzw. für $1-r$ eine möglichst kleine Oberschranke. Lässt sich dies realisieren, so kann 2δ nach (B.103) ohne Auslöschung ausgewertet werden. Zur Berechnung einer möglichst kleinen Oberschranke von $1-r$ kann man sich wegen der Symmetrie zur reellen Achse, d.h. wegen $r(x,y) \equiv r(x,-y)$ auf $y>0$ beschränken. Wir zeigen jetzt wieder, dass $\partial(1-r)/\partial y$ für $y>0$ positiv ist, so dass das Maximum von $1-r$ für $x<1$ auf dem folgenden Hyperbelast liegt:

$$y=\frac{1}{12}\cdot\sqrt{112\,x^2-63} \tag{B.105}$$

Für $0.75 \leq x < 1$ ist die partielle Ableitung $\partial(1-r)/\partial y$ mit

$$A := \sqrt{(1-x)^2+y^2} > 1-x, \quad B := \sqrt{(1+x)^2+y^2} > 1+x \quad \text{gegeben durch:}$$

$$\frac{\partial(1-r)}{\partial y}=\frac{y\cdot\{-4x+A\cdot(1-x)+B\cdot(1+x)\}}{A\cdot B\cdot(A+1-x)^2} \tag{B.106}$$

und für $y>0$ ist $\partial(1-r)/\partial y > 0$, wenn in (B.106) gilt: $\{\ldots\} > 0$, d.h. wenn

$$C := (1-x)\cdot\sqrt{(1-x)^2+y^2}+(1+x)\cdot\sqrt{(1+x)^2+y^2} > 4x. \tag{B.107}$$

Wegen $C \geq (1-x)^2 + (1+x)^2 = 2 + 2x^2$ ist (B.107) erfüllt, wenn gilt

$$2 + 2x^2 > 4x \quad \Longleftrightarrow \quad (1-x)^2 > 0.$$

Das Maximum von $1 - r(x,y)$ liegt damit für $0.75 \leq x < 1$ auf dem Hyperbelast

$$y = \frac{1}{12} \cdot \sqrt{112x^2 - 63}. \tag{B.108}$$

Um das Maximum zu berechnen, setzen wir y aus (B.108) ein in (B.104) und erhalten

$$\begin{aligned} 1 - r(x, y(x)) &= \frac{\sqrt{\frac{16}{9}x^2 + 2x + \frac{9}{16}} - x - 1}{\sqrt{\frac{16}{9}x^2 - 2x + \frac{9}{16}} - x + 1} \\ &= \frac{\sqrt{\frac{1}{144}(16x+9)^2} - (x+1)}{\sqrt{\frac{1}{144}(16x-9)^2} - (x-1)} = \frac{4x-3}{4x+3}. \end{aligned}$$

Da der letzte Term in x monoton wächst, gilt für $0.75 \leq x < 1$

$$1 - r(x, y(x)) \leq \frac{4 \cdot 1 - 3}{4 \cdot 1 + 3} = \frac{1}{7}$$

Im Fall $0.75 \leq x < 1$ haben wir damit im Bereich $\beta(x,y) \geq 0.75$ für $1 - r(x,y)$ mit $1/7$ eine hinreichend kleine Oberschranke gefunden, so dass 2δ nach (B.103) ohne Auslöschung ausgewertet werden kann. Bitte beachten Sie, dass wir in den verschiedenen Fällen $x < 1,\ x = 1,\ x > 1$ für den Ausdruck $1 - r$ jeweils die gleiche, hinreichend kleine Oberschranke $1/7$ gefunden haben, obwohl in (B.103),(B.93),(B.70) verschieden Terme zur Berechnung von δ definiert wurden. Vermutlich ist die Gleichheit der gefundenen Oberschranken ein Hinweis dafür, dass die gewählten Terme zur Auswertung von $\delta(x,y)$ für die numerische Stabilität schon optimal gewählt wurden.

Zur Einschließung von $\Re(\arccos(\boldsymbol{Z}))$ ist nach Seite 177 die reelle arccos-Funktion für den Intervall-Ausdruck $\boldsymbol{\beta}$ auszuwerten:

$$\arccos(\boldsymbol{\beta}), \quad \boldsymbol{\beta} := \frac{2\boldsymbol{x}}{\sqrt{(\boldsymbol{x}+1)^2 + \boldsymbol{y}^2} + \sqrt{(\boldsymbol{x}-1)^2 + \boldsymbol{y}^2}}, \tag{B.109}$$

wobei die Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ durch die Extremalpunkte auf dem Rand von $\boldsymbol{Z}$ bestimmt sind. Dabei haben wir zunächst den Fall $\texttt{Inf}(\boldsymbol{\beta}) \to +1$ betrachtet, um Überschätzungen in der Nähe der Nullstelle der reellen arccos-Funktion zu vermeiden. Überschätzungen treten jedoch auch im Fall $\texttt{Sup}(\boldsymbol{\beta}) \to -1$ auf, da die reelle arccos-Funktion am linken Definitionsrand nahezu senkrechte Tangenten besitzt, so dass kleine Überschätzungen bei der Berechnung des Arguments $\boldsymbol{\beta}$ deutliche Überschätzungen bei den Funktionswerten verursachen. Mit dem Punktargument $z = -1 + i \cdot 2^{-200}$ und $\texttt{prec} = 53$ erhält man z.B. für $\Re(w) := \Re(\arccos(z))$ die folgende recht grobe Einschließung:

$$\Re(w) \in [3.1415926\mathbf{237874627}, 3.1415926\mathbf{535898020}]$$

mit nur 8 korrekten Dezimalziffern. Diese zu groben Einschließungen lassen sich durch Anwendung folgender, für $0 \leq \delta \leq 2$ geltenden Identität vermeiden:

$$\arccos(-1+\delta) \equiv \pi - \arccos(1-\delta) \equiv \pi - \arcsin(\sqrt{\delta \cdot (2-\delta)}). \tag{B.110}$$

Bei vorgegebenem $\beta = -1 + \delta \leq -0.75$ gilt dann $\delta = 1 - (-\beta)$, mit $(-\beta) \geq 0$, so dass $\delta \geq 0$ mit Hilfe der Gleichungen (B.70),(B.93) und (B.103) intervallmäßig berechnet werden kann. Die intervallmäßige Auswertung der rechten Seite von (B.110) liefert dann eine fast optimale

Einschließung des Funktionswertes $\arccos(\beta) = \arccos(-1 + \delta)$. Der beschriebene Algorithmus liefert dann mit dem Punktargument $z = -1 + i \cdot 2^{-200}$ für $\Re(w) := \Re(\arccos(z))$ und $\texttt{prec} = 53$ die folgende fast optimal Einschließung:

$$\Re(w) \in [3.14159265358979\mathbf{2}, 3.14159265358979\mathbf{4}].$$

Damit wird der Realteil $\Re(\arccos(z)) = \arccos(\beta)$ im ganzen Bereich $|\beta| \leq +1$ in ausreichender Genauigkeit eingeschlossen.

B.2.10 $\arctan(z)$

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{i \cdot (-\infty, -1) \cup i \cdot (+1, +\infty)\}$ liefert die Funktion

```
MpfciClass atan(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `atan(Z)` für die komplexen Funktionswerte $\arctan(z)$, mit $z \in Z$.

$$\{\arctan(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{atan(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der imaginären Achse von $-i\infty$ bis $-i$ bzw. von $+i$ bis $+i\infty$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

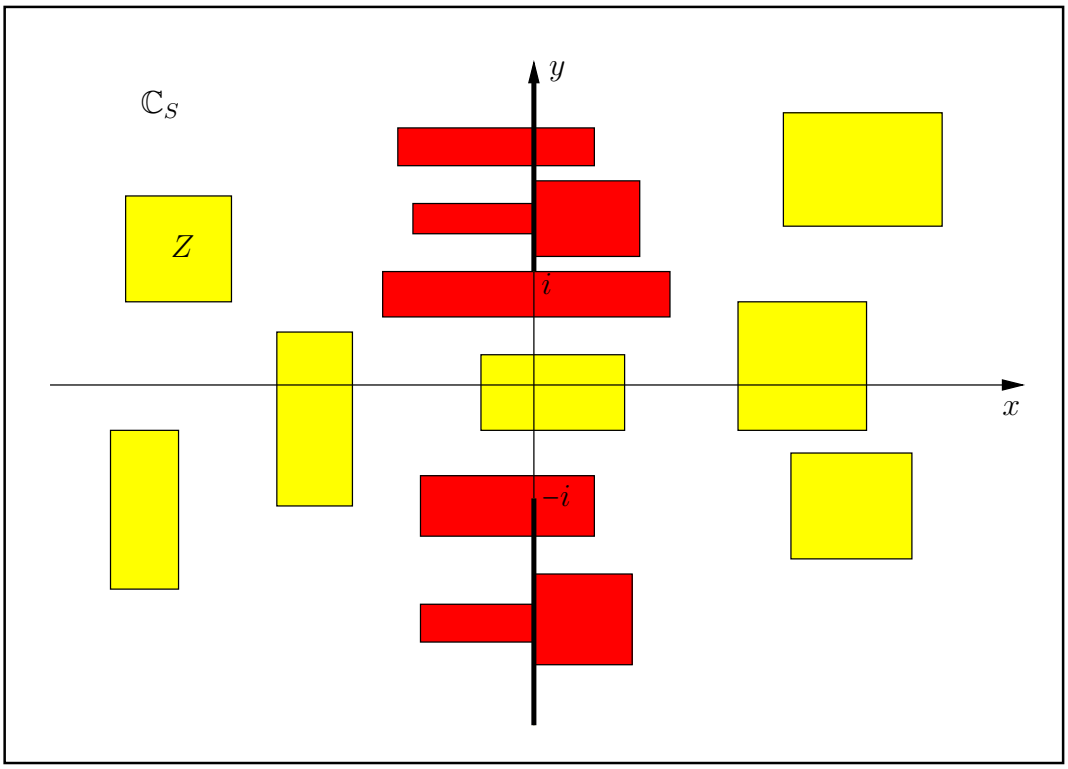


Abbildung B.17: Erlaubte und nicht erlaubte Intervalle Z von `atan(Z)`.

Nach Abbidung B.17 darf also kein Punkt der beiden Verzweigungsschnitte ein Element eines erlaubten, achsenparallelen Intervalls Z sein.
Mit $Z = [2^{-1073741824}, 2^{-1073741824}] + i \cdot [1, 1]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[12]

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([7.85398163397448309613e-1, 7.85398163397448309619e-1],\\ &[3.72130559324020099216e8, 3.72130559324020099218e8]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 1]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([4.6364760900080611621425 6e-1, 1.1071487177940905030170 7],\\ &[-7.0830333601405402006238 5e-1, 7.0830333601405402006238 5e-1]).\end{aligned}$$

[12] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

B.2.10.1 Algorithmus

In [43] wird für den Imaginärteil der arctan-Funktion u.a. folgender Term ausgewertet:

$$T(x) := \ln\left[1 + \frac{4 \cdot \sqrt{1+x^2}}{x^2 + \left(1 - \sqrt{1+x^2}\right)^2}\right], \quad x = [x]: \text{ Punktintervall}$$

dabei wurde vermutlich übersehen, dass sich $T(x)$ noch wesentlich vereinfachen lässt, wodurch sowohl die Güte der Einschließung als auch die Laufzeit deutlich verbessert werden können. Nach entsprechender Vereinfachung sind dann für den Imaginärteil folgende Terme auszuwerten:

$$T(x) := \ln\left[1 + \frac{2}{\sqrt{1+x^2} - 1}\right], \quad x = [x]: \text{ Punktintervall} \tag{B.111}$$

$$Q_{1,2}(x,y) := \ln\left[1 \pm \frac{4y}{x^2 + (1 \mp y)^2}\right], \quad \text{nur } y = [y] \text{ ist ein Punktintervall.} \tag{B.112}$$

In (B.112) bedeuten die Indices 1 bzw. 2 das jeweilige obere bzw. untere Vorzeichen.

Auswertung von $T(x)$

Der Term $T(x)$ ist nur für Punktintervalle $x = [x]$ auszuwerten. Zur Vermeidung von Overflow müssen die beiden Fälle $x \to 0$, d.h. Nenner $\to 0$ und $x \to \infty$, d.h. $x^2 \to \infty$ gesondert betrachtet werden.

Für $x \to 0$ benutzen wir die folgende Darstellung:

$$T(x) = \ln\left[2 + x^2 + 2 \cdot \sqrt{1+x^2}\right] - 2 \cdot \ln(x) \tag{B.113}$$

Der obige Term $T(x)$ kann jetzt intervallmäßig für $x \to 0$ problemlos ausgewertet werden. Die Laufzeit kann jedoch durch die folgende Vereinfachung noch mehr als halbiert werden:

Zunächst gilt:

$$\begin{aligned} \alpha &:= 2 + x^2 + 2 \cdot \sqrt{1+x^2} \\ &= 4 + \left[2x^2 + \sum_{k=2}^{\infty} (-1)^{k+1} \frac{1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2k-3)}{4 \cdot 6 \cdot 8 \cdot \ldots \cdot 2k} \cdot (x^2)^k\right], \end{aligned}$$

und weil die Reihe in $[\ldots]$ für $x^2 < 1$ eine alternierende Leibniz-Reihe ist, erhält man für α die Einschließung:

$$4 < \alpha < 4 + 2x^2, \quad x^2 < 1;$$

Wegen der Monotonie der ln-Funktion folgt daraus direkt die Einschließung:

$$\ln(4) < \ln(\alpha) < \ln(4 + 2x^2) = \ln(4) + \ln(1 + \frac{1}{2} \cdot x^2) < \ln(4) + \frac{1}{2} \cdot x^2, \quad \text{d.h.}$$

$$\ln(4) < \ln(\alpha) < \ln(4) + \frac{1}{2} \cdot x^2, \quad x^2 < 1. \tag{B.114}$$

Es gilt also für das folgende Nicht-Maschinenintervall $\texttt{Nm}(x)$

$$\ln(\alpha) \in \texttt{Nm}(x) := \left[\ln(4)\ ,\ \ln(4) + \frac{1}{2} \cdot x^2\right], \quad x^2 < 1. \tag{B.115}$$

Das Ziel ist nun, $x \in]0,1[$ in (B.115) so klein zu wählen, dass für jede Current-Präzision `prec`, mit $2 \le \texttt{prec} \le \texttt{prec}_0$, ein Maschinenintervall `U` so angegeben werden kann, dass gilt:

$$\texttt{Nm}(x) \subseteq \texttt{U} \quad \text{und} \quad \texttt{sup(U)} = \texttt{succ(Inf(U))}. \tag{B.116}$$

Es ist klar, dass die zweite Bedingung in (B.116) bei festem `prec` nicht erfüllt sein kann, wenn $x = 2^k \in]0,1[$ zu groß gewählt wird. In einer Schleife, d.h. mit der Funktion

```
int Schleife1(const int prec)
{
  int k(-prec/2 -9);
  MpfiClass::SetCurrPrecision(prec);
  MpfiClass U(0);
  MpfiClass ln4(MpfiClass::Ln2(2*prec));
  MpfiClass x, u;
  times2pown(ln4,1); // ln(4) mit Praezision 2*prec;
  MPFR::MpfrClass::MpfrClass LrI, LrS;

  do
  {
    k++;
    MpfiClass::SetCurrPrecision(2*prec);
    x = MpfiClass( comp(MPFR::MpfrClass::MpfrClass(0.5),k) );
    LrI = Inf(ln4);
    LrS = Sup(ln4 + sqr(x)/2);
    u = MpfiClass(LrI, LrS);
    MpfiClass::SetCurrPrecision(prec);
    U = u;
    U.RoundPrecision(prec);
  } while( Sup(U) == succ(Inf(U)) );

  return k-1;
}
```

berechnet man daher, beginnend mit einem hinreichend kleinen $k < 0$, in stark vergrößerter Präzision[13] eine Einschließung von $\ln(4) + x^2/2 \in \mathtt{u}$ und rundet dieses $\mathtt{u}$ in ein Maschinenintervall $\mathtt{U} \supseteq \mathtt{u}$. In der Schleife wird dann das maximale k berechnet, für das die obige Bedingung `sup(U) = succ(Inf(U))` erfüllt ist, und dieses k wird von `Schleife1(prec)` zurückgegeben. Für das in `Schleife1(prec)` in der Current-Präzision `prec` berechnete `U` und für das zurückgegeben k gilt damit:

$$0 \le x \le 0.5 \cdot 2^k \quad \Longrightarrow \quad \ln(4) + \frac{x^2}{2} \in \mathtt{U},\ \ln(4) \in \mathtt{U} \ \text{ und } \ \mathtt{sup(U)} = \mathtt{succ(Inf(U))}.$$

Berechnet man daher in der Current-Präzision `prec` mit $\hat{\mathtt{U}} := 2 \diamond \mathtt{MpfiClass::Ln2(prec)}$ eine nicht notwendig optimale Maschineneinschließung von $\ln(4)$, so gilt $\mathtt{U} \subseteq \hat{\mathtt{U}}$, wobei $\mathtt{U} = \hat{\mathtt{U}}$ nur dann gilt, wenn $\hat{\mathtt{U}}$ eine optimale Einschließung von $\ln(4)$ ist. Bedeutet dann in der Current-Präzision `prec` $[x]$ das Punktintervall, das x einschließt, so gilt

$$0 \le x \le 0.5 \cdot 2^k \quad \Longrightarrow \quad T(x) \in -2 \diamond \ln([x]) \oplus \mathtt{U} \subseteq -2 \diamond \ln([x]) \oplus \hat{\mathtt{U}},$$

wobei man jetzt das obige Intervall $-2 \diamond \ln([x]) \oplus \hat{\mathtt{U}}$ mit minimalem Durchmesser und optimaler Laufzeit berechnen kann.

Da der maximale k-Wert von der gewählten Current-Präzision $\mathtt{prec} \ge 2$ unmittelbar abhängt, wäre es äußerst sinnvoll, wenn man eine von `prec` abhängige Unterschranke $k_0(\mathtt{prec}) < k$ so angeben könnte, dass für einen möglichst großen Bereich $2 \le \mathtt{prec} \le \mathtt{prec}_0$ folgendes gilt:

$$0 \le x \le 0.5 \cdot 2^{k_0} \quad \Longrightarrow \quad T(x) \in -2 \diamond \ln([x]) \oplus \hat{\mathtt{U}}, \quad 2 \le \mathtt{prec} \le \mathtt{prec}_0.$$

[13] Es zeigt sich, dass die berechneten Einschließungen in doppelter Päzision eng genug sind.

Mit einer zweiten Schleife (Programmteil `Schleife2`) findet man dazu[14]

$$(B.117)\quad 0 \le x \le 0.5\cdot 2^{-(\mathtt{prec}\oslash 2+9)},\ 2 \le \mathtt{prec} \le \mathtt{prec}_0 = 677\,370 \quad\Longrightarrow\quad T(x) \in -2 \diamond \ln([x]) \oplus \hat{\mathtt{U}}.$$

```
// Schleife2
int k, prec = 1;
do
{
  prec++;
  cout << "prec == " << prec << endl;
  k = Schleife1(prec);
} while( k >= -(prec/2+9) );
```

Damit können wir in nahezu allen praktischen Fällen mit (B.117) eine optimale Einschließung berechnen. Abschließend geben wir für $x = \mathtt{m}\cdot 2^{ex}$ noch an, wie die erste Bedingung zu realisieren ist. Wegen $\mathtt{m}\cdot 2^{ex} < 2^{ex}$ verlangen wir $2^{ex} \le 0.5\cdot 2^{-(\mathtt{prec}\oslash 2+9)} \iff ex \le -\mathtt{prec} \oslash 2 - 10$, wobei ex durch $\mathtt{expo}(x)$ definiert ist.

Im sehr seltenen Fall $\mathtt{prec} > \mathtt{prec}_0$ wird die Einschließung von T mit (B.113) jedoch etwas aufwendiger realisiert. Der Präzision von $\mathtt{prec}_0 = 677370$ Bits entsprechen ca. 203908 Dezimalstellen, womit alle praktischen Fälle abgedeckt sein sollten.

Wir kommen jetzt zur Auswertung von $T(x)$ für $x \to +\infty$. Bei der normalen Berechnung von $T(x)$ nach (B.111) wird der Nenner $\sqrt{1+x^2}-1$ mit Hilfe der Funktion $\mathtt{sqrt1pm1(sqr}([x]))$ ausgewertet. Um dabei einenÜberlauf zu vermeiden, verlangen wir

$$x^2 = \mathtt{m}^2\cdot 2^{2ex} < 2^{2ex} < 2^{+1073741820} \quad\Longleftrightarrow\quad \mathtt{expo}(x) = ex < 536870910,$$

so dass im Fall $\mathtt{expo}(x)$ >= 536870910 ein Überlauf eintreten kann. Um in diesem Fall einen solchen Überlauf zu vermeiden, schreiben wir den Bruch in (B.111) wie folgt um:

$$\beta := \frac{2}{\sqrt{1+x^2}-1} = \frac{\frac{2}{x}}{\sqrt{1+\frac{1}{x}\cdot\frac{1}{x}}-\frac{1}{x}}$$

Auch jetzt könnte man den Term rechts ohne Overflow für $x \to +\infty$ auf dem Rechner auswerten. Die Laufzeit kann jedoch durch die folgenden Überlegungen wesentlich reduziert werden. Mit $r := 1/x$ erhält man zunächst:

$$\begin{aligned}\beta(r) &:= 2r\cdot\frac{1}{\sqrt{1+r^2}-r} = 2r\cdot\frac{\sqrt{1+r^2}+r}{(\sqrt{1+r^2}-r)\cdot(\sqrt{1+r^2}+r)}\\ &= 2r\cdot(\sqrt{1+r^2}+r)\\ &= 2r+2r^2+\left(r^3-\frac{r^5}{4}+\frac{r^7}{8}-\frac{5}{64}r^9+-\ldots\right)\end{aligned}$$

und da der letzte Klammerausdruck ( ) wegen $\sqrt{1+r^2}$ selbst wieder eine alternierende Leibniz-Reihe ist, ergibt sich für $\beta(r)$ die Einschließung:

$$(B.118)\qquad 2r+2r^2 < \beta(r) < 2r+2r^2+r^3,\quad 0<r<1.$$

Da die Taylorreihe von

$$\ln(1+x) = x-\frac{1}{2}x^2+\frac{1}{3}x^3-+\ldots,\quad -1<x\le 1$$

[14] $\oslash$ bedeutet die integer-Division auf der Maschine, wobei zur Null hin gerundet wird, wenn der exakte Quotient keine integer-Zahl ist.

für $0 < x \le 1$ eine alternierende Leibniz-Reihe ist, folgt die Ungleichung

$$x - \frac{1}{2}x^2 < \ln(1+x) < x, \quad 0 < x \le 1; \tag{B.119}$$

Mit den rechten Ungleichungen aus (B.118) und (B.119) folgt nun direkt:

$$\ln(1+\beta(r)) < 2r + 2r^2 + r^3, \quad \text{falls} \;\; 2r + 2r^2 + r^3 < 1,$$

wobei die Bedingung $2r + 2r^2 + r^3 < 1$ wegen $x \to \infty$, d.h. $r \to 0$ sicher erfüllt sein wird. Wir benötigen jetzt noch eine Unterschranke von $\ln(1+\beta(r))$. Dazu gilt wieder nach (B.118) und (B.119)

$$\ln(1+\beta(r)) > \beta(r) - \frac{1}{2}\beta(r)^2 > 2r + 2r^2 - \frac{1}{2}\beta(r)^2$$

Aus (B.118) folgt für $0 < r < 1$

$$-\frac{1}{2}\beta(r)^2 > -2r^2 - 4r^3 - 4r^4 - 2r^5 - \frac{1}{2}r^6 > -2r^2 - 4r^3 - 4r^4 - 2r^4 - 1r^4$$

und das ergibt dann die gesuchte Einschließung:

$$2r - 4r^3 - 7r^4 < \ln(1+\beta(r)) < 2r + 2r^2 + r^3, \quad 2r + 2r^2 + r^3 < 1;$$
$$2r \cdot (1 - 2r^2 - \frac{7}{2}r^3) < \ln(1+\beta(r)) < 2r \cdot (1 + r + \frac{1}{2}r^2),$$

die noch etwas vereinfacht werden kann:

$$2r \cdot (1 - 2r^2 - 4r^3) < \ln(1+\beta(r)) < 2r \cdot (1 + r + \frac{1}{2}r^2), \quad 2r + 2r^2 + r^3 < 1. \tag{B.120}$$

Für hinreichend kleine r kann diese Einschließung noch wesentlich vereinfacht werden. Dazu verlangen wir für die Unterschranke

$$\begin{aligned} & 1 - 2r^2 - 4r^3 > \mathtt{pred}(1) = 1 - 2^{-\mathtt{prec}} \\ \iff \quad & 2r^2 \cdot (1 + 2r) < 2^{-\mathtt{prec}}. \end{aligned} \tag{B.121}$$

Mit $r = 1/x = (1/m) \cdot 2^{-ex}$, $ex \gg 1$, folgt weiter

$$1 + 2r = 1 + \frac{1}{\mathtt{m}} \cdot 2^{-ex+1} \le 1 + 2^{-ex+2}, \quad \text{d.h. (B.121) ist erfüllt, wenn gilt:} \quad 2r^2 \cdot 2 < 2^{-\mathtt{prec}}.$$

Es gilt zusätzlich

$$2r^2 \cdot 2 = \frac{1}{\mathtt{m}^2} \cdot 2^{-2ex+2} \le 4 \cdot 2^{-2ex+2} = 2^{-2ex+4},$$

d.h. (B.121) ist erfüllt, wenn gilt

$$2^{-2ex+4} < 2^{-\mathtt{prec}} \quad \iff \quad ex > \mathtt{prec}/2 + 2.$$

Symbolisiert $\oslash$ den Operator der integer-Division, der zur Null rundet, wenn der Quotient keine integer-Zahl ist, so gilt $\mathtt{prec}/2 < \mathtt{prec} \oslash 2 + 1$, d.h. (B.121) ist erfüllt, wenn gilt

$$ex > \mathtt{prec} \oslash 2 + 3 \quad \Longrightarrow \quad 1 - 2r^2 - 4r^3 > \mathtt{pred}(1). \tag{B.122}$$

Zur Verbesserung der Oberschranke in (B.120) verlangen wir jetzt

$$\begin{aligned} & 1 + r + \frac{r^2}{2} < \mathtt{succ}(1) = 1 + 2^{-\mathtt{prec}+1} \\ \iff \quad & r + \frac{r^2}{2} < 2^{-\mathtt{prec}+1}. \end{aligned} \tag{B.123}$$

Es gilt zusätzlich noch die folgende Abschätzung

$$r + \frac{r^2}{2} = \frac{1}{\mathtt{m}} \cdot 2^{-ex} + \frac{1}{\mathtt{m}^2} \cdot 2^{-2ex-1} \le 2 \cdot 2^{-ex} + 4 \cdot 2^{-2ex-1} = 2^{-ex+1}(1 + 2^{-ex}) < 2^{-ex+2},$$

d.h. (B.123) ist erfüllt, wenn gilt: $2^{-ex+2} < 2^{-\mathtt{prec}+1} \iff ex > \mathtt{prec} + 1$. Damit erhalten wir

$$ex > \mathtt{prec} + 1 \quad \Longrightarrow \quad 1 + r + \frac{r^2}{2} < \mathtt{succ}(1). \tag{B.124}$$

Zusammen mit (B.120) folgt jetzt unter den Bedingungen $ex > \mathtt{prec} + 1$ und $ex > \mathtt{prec} \oslash 2 + 3$

$$2r \cdot \mathtt{pred(1)} < \ln(1 + \beta(r)) < 2r \cdot \mathtt{succ(1)}, \quad 2r + 2r^2 + r^3 < 1.$$

Man kann noch einfach nachweisen, dass die beiden obigen Bedingungen erfüllt sind, wenn gilt $ex > \mathtt{prec} + 2$. Zusammen mit (B.62) und (B.85) erhalten wir mit $x = \mathtt{m} \cdot 2^{ex}$, $r = 1/x$, $\mathtt{t} = [2r]$:

$$ex > \mathtt{prec} + 2 \quad \Longrightarrow \quad \mathtt{pred(Inf(t))} < \ln(1 + \beta(r)) < \mathtt{succ(succ(Sup(t)))}. \tag{B.125}$$

Im Fall $ex \le \mathtt{prec} + 2$, der wegen $ex \ge 536870910$ in der Praxis kaum auftreten wird, benutzen wir:

$$2r \cdot [1 - 2r^2(1 + 2r)] < \ln(1 + \beta(r)) < 2r \cdot [1 + r(1 + r)]. \tag{B.126}$$

Die Auswertung von $T(x)$ erfolgt mithilfe der Funktion

```
MpfiClass Aux_1_atan(const MPFR::MpfrClass::MpfrClass& x);
```

Auswertung von $Q_{1,2}(x, y)$

Wir beschränken uns auf den Index 1, und nach (B.112) ist auszuwerten

$$Q_1(x, y) := \ln\left[1 + \frac{4y}{x^2 + (1 - y)^2}\right], \tag{B.127}$$

dabei ist $Q_1(x, y)$ einzuschließen, wobei $y = [y]$ als Punktintervall und x=[x] als echtes Intervall aufzufassen ist. Bei der Auswertung des Bruches in (B.127) ist zu vermeiden, dass Zähler oder Nenner oder der Bruch selbst einen Überlauf liefern.

Wir betrachten zunächst den Fall: $\boldsymbol{y = 1}$
Wegen

$$\ln\left(1 + \frac{4}{x^2}\right) = \begin{cases} \ln\left(1 + \frac{2}{x} \cdot \frac{2}{x}\right), & x > 1 \\ \ln(4 + x^2) - 2 \cdot \ln(x), & 0 < x \le 1 \end{cases}$$

sind drei Unterfälle zu behandeln:

1. $Q_1([x]) \subseteq \ln\left(1 + \frac{2}{[x]} \cdot \frac{2}{[x]}\right)$, wenn $\mathtt{Inf([x])} \in [1, \mathtt{MaxFloat()}]$
2. $Q_1([x]) \subseteq \ln(4 + [x]^2) - 2 \cdot \ln([x])$, wenn $\mathtt{Sup([x])} < 1$;
3. Es bleibt der Fall, dass 1 Innen- oder rechter Randpunkt des echten Intervalls $[x] = [x_1, x_2]$ ist. Mit der Auswertung der Funktionen

$$\begin{aligned} H_1([x_2]) &:= \ln\left(1 + \frac{2}{[x_2]} \cdot \frac{2}{[x_2]}\right) \\ H_2([x_1]) &:= \ln(4 + [x_1]^2) - 2 \cdot \ln([x_1]) \end{aligned}$$

erhält man die Einschließung:

$$Q_1([x]) \subseteq [\mathtt{Inf}(H_1([x_2])), \mathtt{Sup}(H_2([x_1]))]$$

Im Falle $y = 1$ kann man so für beliebige Intervalle $[x] = [x_1, x_2]$, mit $x_1 > 0$ eine garantierte Einschließung für $Q_1([x])$ ohne Auftreten eines Überlaufs berechnen.

Es bleibt der Fall: $\boldsymbol{y \neq 1}$.
Zunächst gilt folgendes: Für $x \to 0$ und $y \to 1$, mit $y \neq 1$ bleibt auch bei großen Präzisionen von y

$$b := \frac{y}{x^2 + (1-y)^2} \tag{B.128}$$

beschränkt, d.h. $b <$ `MaxFloat()`, so dass kein Überlauf entstehen kann. Zur Vermeidung eines vorzeitigen Überlaufs im Nenner muss daher noch der Fall: $x \to +\infty$ oder $y \to +\infty$ betrachtet werden. Falls der Nenner $Ne := x^2 + (1-y)^2$ zum Überlauf führt, muss mit einer geeigneten Zweierpotenz $2^s, s < 0$ so multipliziert werden, dass gerade kein Überlauf mehr auftritt:

$$2^{2s} \cdot Ne = (2^s \cdot x)^2 + (s^s - 2^s \cdot y)^2 =: Nes < \texttt{MaxFloat()}$$

Beachten Sie bitte, dass bei großem y und bei zu kleinem Nes bei der Division

$$bs := \frac{y}{Nes}$$

wieder ein Überlauf auftreten kann! Bei der Rückskalierung mit 2^{2s+2} berücksichtigt der Summand +2 den Faktor 4 in (B.127):

$$2^{2s+2} \cdot bs = \frac{4 \cdot y}{x^2 + (1-y)^2}$$

Für $x \to +\infty$ oder $y \to +\infty$ kann so das Argument der `lnp1`-Intervallfunktion ohne Überlauf berechnet werden.
Im Restbereich erhält man dann für $Q_1([x],[y])$ durch Intervallauswertung von

$$Q_1([x],[y]) \subseteq \ln\left[1 + \frac{4[y]}{[x]^2 + (1-[y])^2}\right] \tag{B.129}$$

eine garantierte Einschließung ohne zwischenzeitlichen Overflow. Für $y \to 0$ und $x \to \infty$ muss in (B.129) die Funktion `lnp1` mit dem obigen Bruch als Argument zur Anwendung kommen. Weitere Einzelheiten findet man im Quelltext der folgenden Funktion `Q_atan_UPSIGN(...)`, die bei der Auswertung des inversen Tangens zur Anwendung kommt.

Das folgende Beispiel benutzt mit der Current-Präzision `prec` = 200 das Argumentintervall $Z = [0,0] + i \cdot [\texttt{pred}(1), \texttt{pred}(1)]$, wobei nach (3.1) gilt: $\texttt{pred}(1) = 1 - 2^{-\texttt{prec}} = 1 - 2^{-200}$. Man erhält die Einschließung mit 60 Dezimalstellen:

```
atan(Z)  =  ([0.0, 0.0],
             [6.96612916462745035964318282065467450915877635032056530391283e1,
              6.96612916462745035964318282065467450915877635032056530391284e1]).
```

Erhöht man mit $Z = [0,0] + i \cdot [\texttt{pred}(1), \texttt{pred}(1)]$ die Current-Präzision z.B. auf `prec` = 2000000, so erhält man nach einigen Sekunden die Einschließung:

```
atan(Z)  =  ([0.0, 0.0],
             [6.93147527133535589389886830074237297163784172110322434248307e5,
              6.93147527133535589389886830074237297163784172110322434248308e5, ]),
```

wobei hier bei der Ausgabe die gleiche Dezimalstellenzahl wie im ersten Beispiel gewählt wurde.

Im Algorithmus von `atan`(Z) ist noch folgender Intervallausdruck auszuwerten:

$$\mathbf{D}(s) \coloneqq 2^{2s} - y^2 - x^2, \qquad s \in \mathbb{Z}, \quad x, y \text{ vom Typ } \texttt{MpfiClass},$$

wobei nur x ein Punktintervall ist. Zusätzlich wird vorausgesetzt, dass bei der Auswertung von 2^{2s}, y^2 und x^2 kein Überlauf entsteht. Eine fast optimale Einschließung von $\mathbf{D}(s)$ wird berechnet mithilfe der Funktion

```
void TwoPow2s_y2_x2(const long s, const MpfiClass& y,
                                  const MpfiClass& x, MpfiClass& D),
```

die in `mpfciclass.cpp` implementiert ist. Es gilt dann $\mathbf{D}(s) \subseteq \texttt{D}$.

Um die Problematik einer optimalen Einschließung von $\mathbf{D}(s)$ zu erläutern, betrachten wir den Spezialfall $s = 0$, d.h. einzuschließen ist

(B.130) $\mathbf{D}_0 \coloneqq 1 - y^2 - x^2$, x, y vom Typ MpfiClass, wobei nur x ein Punktintervall ist.

Prinzipiell ist $\mathbf{D}_0$ in (B.130) schon optimal, da nach [5, Seite 32] in einem Intervallterm zur optimalen Einschließung die Intervallvariablen jeweils nur einmal auftreten dürfen. In einigen Sonderfällen kann es jedoch zu Auslöschungseffekten kommen, die dann eine optimale Einschließung von $\mathbf{D}_0$ dennoch verhindern.

Ein **erster Sonderfall** liegt vor, wenn z.B. das Punktintervall x sehr dicht bei 1 liegt. In diesem Fall wird $\mathbf{D}_0$ intervallmäßig fast optimal ausgewertet mit

$$\texttt{D} \coloneqq (1 \diamondminus x) \diamond (1 \diamondplus x) - \texttt{sqr}(y) \supseteq \mathbf{D}_0.$$

Beachten Sie, dass im Intervallausdruck rechts jetzt die Intervallvariable x zweimal vorkommt. Die damit verbundene Überschätzung ist jedoch ganz minimal, da x ein Punktintervall ist.

Ein **zweiter Sonderfall** liegt vor, wenn z.B. das echte Intervall y sehr schmal ist und sehr dicht bei 1 liegt. In diesem Fall wird $\mathbf{D}_0$ intervallmäßig fast optimal ausgewertet mit

$$\texttt{D} \coloneqq (1 \diamondminus y) \diamond (1 \diamondplus y) - \texttt{sqr}(x) \supseteq \mathbf{D}_0.$$

In der obigen Funktion `TwoPow2s_y2_x2(...)` findet man weitere Einzelheiten. Die folgenden Beispiele zeigen die gewonnenen Verbesserungen einiger Einschließungen.

Mit $Z = [-\texttt{minfloat}(), -\texttt{minfloat}()] + i \cdot [1 - 2^{-70}, 1 - 2^{-70}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned} \texttt{atan(Z)} \;=\; & ([-1.4064180...553497e - 323228476, -1.4064180...553495e - 323228476], \\ & [2.4606724909878058484 2e1, 2.4606724909878058484 4e1]). \end{aligned}$$

Mit $Z = [-\texttt{minfloat}(), -\texttt{minfloat}()] + i \cdot [1 - 2^{-70000}, 1 - 2^{-70000}]$ erhält man mit der Current-Präzision `prec` = 70000 die Einschließung

$$\begin{aligned} \texttt{atan(Z)} \;=\; & ([-1.4986879...456363e - 323207425, -1.4986879...456362e - 323207425], \\ & [2.4260497893188365802 2e4, 2.4260497893188365802 3e4]). \end{aligned}$$

`minfloat()` $\coloneqq 2^{-1073741824}$ ist die, von der Current-Präzision unabhängige, kleinste positive Maschinenzahl.

B.2.11 arccot(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - i \cdot [-1, +1]$ liefert die Funktion

```
MpfciClass acot(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `acot(Z)` für die komplexen Funktionswerte arccot(z), mit $z \in Z$.

$$\{\text{arccot}(z) \,\big|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{acot(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der imaginären Achse von $-i$ bis $+i$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

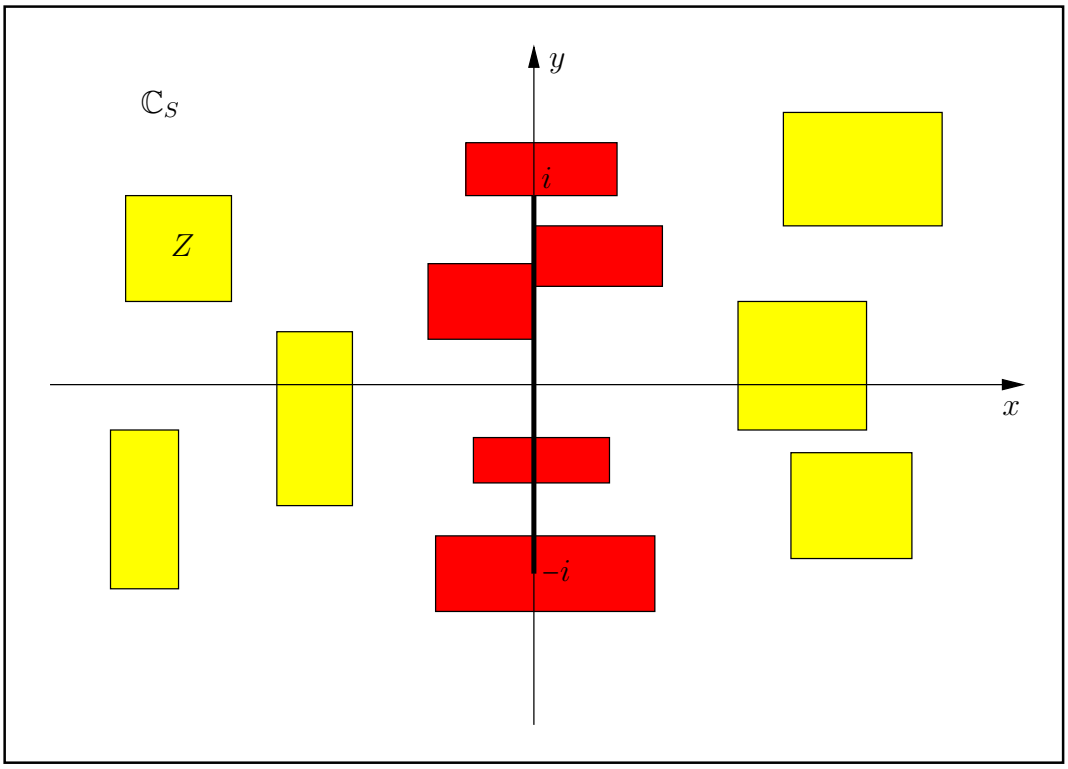


Abbildung B.18: Erlaubte und nicht erlaubte Intervalle Z von `acot(Z)`.

Nach Abbidung B.18 darf also kein Punkt des Verzweigungsschnitts ein Element eines erlaubten, achsenparallelen Intervalls Z sein.
Mit $Z = [2^{-1073741824}, 2^{-1073741824}] + i \cdot [1, 1]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[15]

$$\begin{aligned} \texttt{acot(Z)} \;=\; & ([1.57079632679489661922, 1.57079632679489661924], \\ & [-2.46067249098780584844e1, -2.46067249098780584842e1]). \end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 1]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned} \texttt{acot(Z)} \;=\; & ([4.63647609000806116214256e-1, 1.10714871779409050301707], \\ & [-7.08303336014054020062385e-1, 7.08303336014054020062385e-1]). \end{aligned}$$

[15] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

B.2.12 arsinh(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{(-\infty,-1) \cup (+1,+\infty)\}$ liefert die Funktion

```
MpfciClass asinh(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `asinh(Z)` für die komplexen Funktionswerte arsinh(z), mit $z \in Z$.

$$\{\text{arsinh}(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{asinh(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der reellen Achse von -1 bis $-\infty$ bzw. von $+1$ bis $+\infty$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben. Die Implementierung erfolgt mit

$$\texttt{asinh}(Z) = i \cdot \texttt{asin}(-i \cdot Z);$$

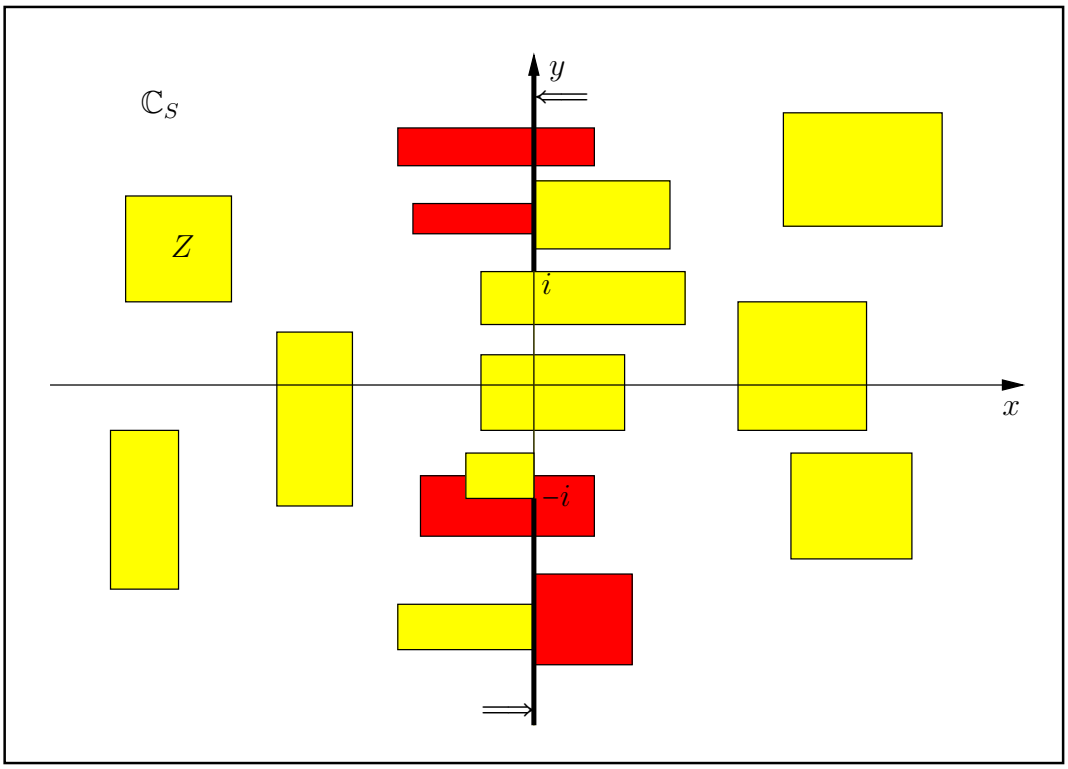


Abbildung B.19: Erlaubte und nicht erlaubte Intervalle Z von `asinh(Z)`.

Mit $Z = [2^{1073741821}, 2^{1073741821}] + i \cdot [2^{1073741821}, 2^{1073741821}]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{asinh(Z)} \;=\; &([7.4426111691517224703398 4e8, 7.4426111691517224703398 6e8],\\ &[7.8539816339744830961565 8e-1, 7.8539816339744830961566 4e-1]).\end{aligned}$$

Mit $Z = [-0.5, 0.5] + i \cdot [-1, -0.5]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{asinh(Z)} \;=\; &([-7.3285767597364526088867 5e-1, 7.3285767597364526088867 5e-1],\\ &[-1.5707963267948966192313 3, -4.5227844715119068206365 7e-1]).\end{aligned}$$

B.2.13 arcosh(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{(-\infty, +1)\}$ liefert die Funktion

```
MpfciClass acosh(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `acosh(Z)` für die komplexen Funktionswerte arcosh(z), mit $z \in Z$.

$$\{\text{arcosh}(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{acosh(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der reellen Achse von $-\infty$ bis $+1$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben. Die Implementierung erfolgt mit

$$\texttt{acosh}(Z) = i \cdot \texttt{acos}(Z) = \pm i \cdot (\pi/2 - \texttt{asin}(Z));$$

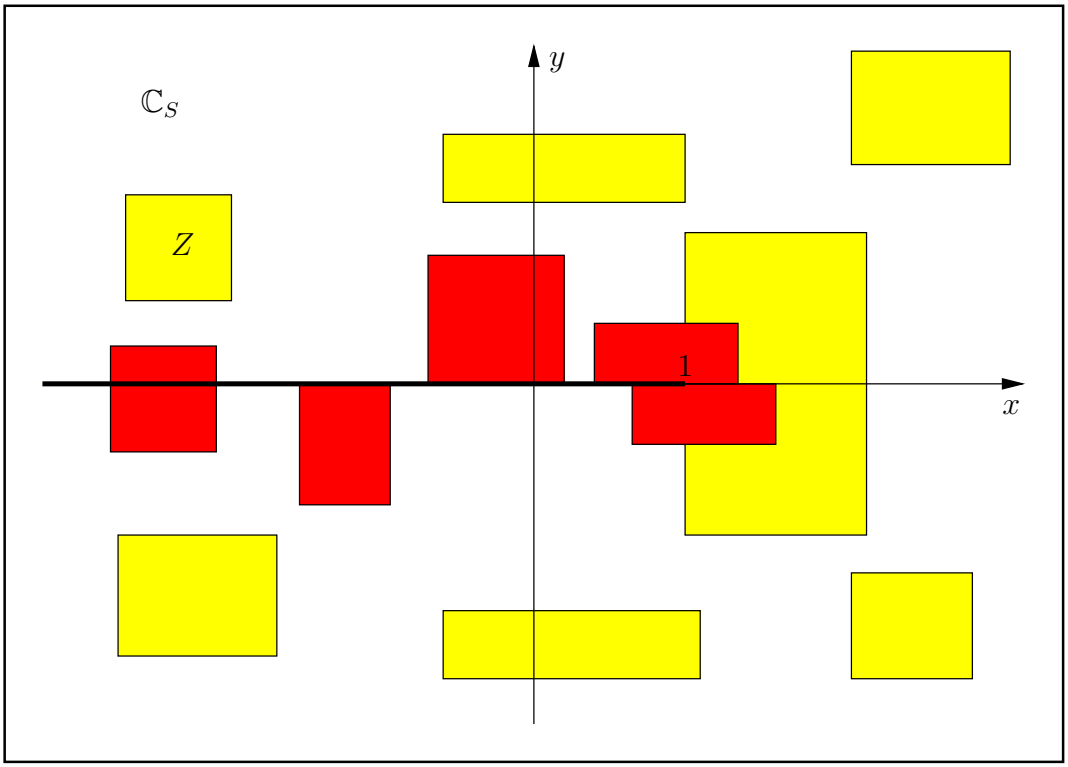


Abbildung B.20: Erlaubte und nicht erlaubte Intervalle Z von `acosh(Z)`.

Mit $Z = [1 - 2^{-70}, 1 - 2^{-70}] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned}\texttt{acosh(Z)} \;=\; &([5.78868064589607735286e - 323228487, 5.78868064589607735289e - 323228487],\\ &[4.11590317489199529168e - 11, 4.11590317489199529171e - 11]).\end{aligned}$$

Mit $Z = [1, 2] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 150 die Einschließung

$$\begin{aligned}\texttt{acosh(Z)} \;=\; &([0, 1.469351744368185273255844317361647616787803357334788 18],\\ &[-9.045568943023813641273167956619587214310945609616050 70e - 1, 0]).\end{aligned}$$

B.2.14 artanh(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - \{(-\infty,-1) \cup (+1,+\infty)\}$ liefert die Funktion

```
MpfciClass atanh(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `atanh(Z)` für die komplexen Funktionswerte artanh(z), mit $z \in Z$.

$$\{\text{artanh}(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{atanh(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der reellen Achse von -1 bis $-\infty$ bzw. von $+1$ bis $+\infty$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

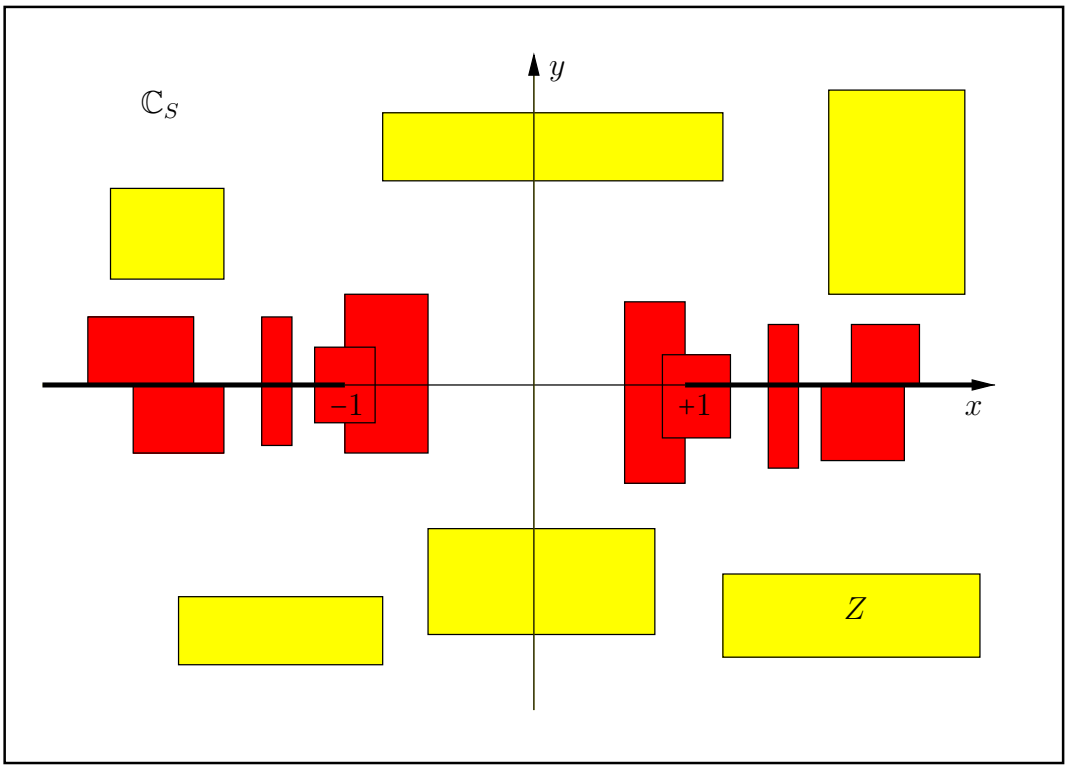


Abbildung B.21: Erlaubte und nicht erlaubte Intervalle Z von `atanh(Z)`.

Mit $Z = [1-2^{-70}, 1-2^{-70}] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned}\texttt{atanh(Z)} \;=\; &([2.46067249098780584842e1, 2.46067249098780584844e1],\\ &[1.4064180...553495e-323228476, 1.4064180...553497e-323228476]).\end{aligned}$$

Mit $Z = [1-2^{-70}, 1-2^{-70}] + i \cdot [2^{3000}, 2^{3000}]$ erhält man mit der Current-Präzision `prec` = 150 die Einschließung

$$\begin{aligned}\texttt{atanh(Z)} \;=\; &([6.60733027580565499208e-1807, 6.60733027580565499209e-1807],\\ &[1.57079632679489661923, 1.57079632679489661924]),\end{aligned}$$

wobei die Ausgabe wie im ersten Beispiel auf 21 Dezimalstellen begrenzt wurde.

B.2.15 arcoth(z)

Mit dem achsenparallelen Rechteckintervall $Z \subset \mathbb{C}_S := \mathbb{C} - i \cdot [-1, +1]$ liefert die Funktion

```
MpfciClass acoth(const MpfciClass& Z);
```

die folgende achsenparallele Rechteckeinschließung `acoth(Z)` für die komplexen Funktionswerte arcoth(z), mit $z \in Z$.

$$\{\text{arcoth}(z) \,|\, z \in Z \subset \mathbb{C}_S\} \subseteq \texttt{acoth(Z)}.$$

$\mathbb{C}_S$ ist dabei die längs der imaginären Achse von $-i$ bis $+i$ aufgeschnittene komplexe Ebene. In der folgenden Abbildung sind einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben.

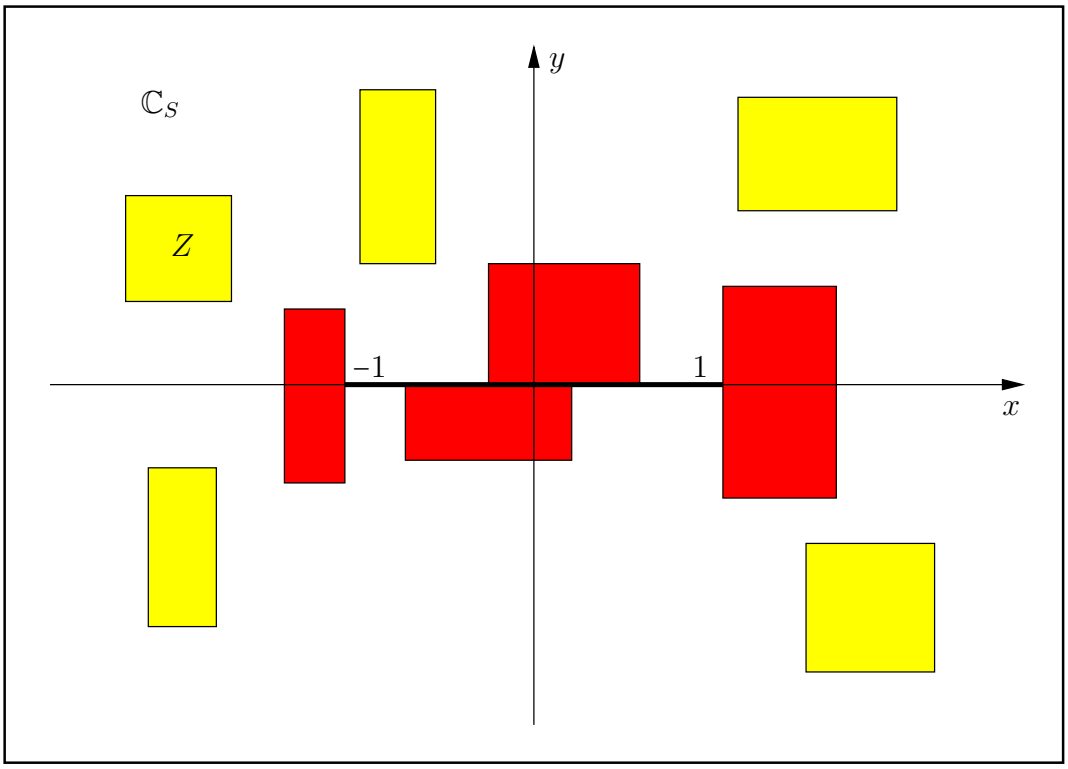


Abbildung B.22: Erlaubte und nicht erlaubte Intervalle Z von `acoth(Z)`.

Mit $Z = [1, 1] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned} \texttt{acoth(Z)} \;=\; & ([3.7213055932402009921 6e8, 3.7213055932402009921 8e8], \\ & [-7.8539816339744830961 7e-1, -7.8539816339744830961 5e-1]). \end{aligned}$$

Mit $Z = [1, 1] + i \cdot [2^{3000}, 2^{3000}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned} \texttt{acoth(Z)} \;=\; & ([6.6073302758056549920 7e-1807, 6.6073302758056549920 9e-1807], \\ & [-8.1285486255577354404 8e-904, -8.1285486255577354404 6e-904]), \end{aligned}$$

wobei die Ausgabe wie im ersten Beispiel auf 21 Dezimalstellen begrenzt wurde.

B.2.16 z^p, $p \in$ P:MpfiClass

Für $z = |z| \cdot e^{i\varphi}$, $-\pi/2 < \varphi < +3\pi/2$ definieren wir mit $z \in$ Z:MpfciClass und $p \in$ P:MpfciClass die Potenz

$$z^p := e^{p \cdot \ln(z)} = e^{p \cdot \ln|z|} \cdot e^{i \cdot p(\varphi + 2\pi k)}, \quad k \in \mathbb{Z}. \tag{B.131}$$

Wegen der Mehrdeutigkeit des komplexen Logarithmus, d.h. wegen $k \in \mathbb{Z}$, gibt es damit für festes $z \neq 0$ und $p \in \mathbb{R} - \mathbb{Q}$ beliebig viele Potenzen z_k^p, die alle auf einem Kreis um den Ursprung mit dem Radius $R = e^{p \cdot \ln|x|}$ liegen. Die Aufgabe besteht nun darin, alle diese Potenzen für alle $z \in$ Z unf für alle $p \in$ P möglichst optimal einzuschließen, d.h. einzuschließen ist die Menge

$$T := \{y \in \mathbb{C} \,\big|\, y = e^{p \cdot \ln|z|} \cdot e^{i \cdot p(\varphi + 2\pi k)}, \quad k \in \mathbb{Z},\ z \in \mathtt{Z},\ p \in \mathtt{P} = [p_1, p_2]\}.$$

Zur Einschließung von T sind drei Fälle zu unterscheiden:

Fall 1: $0 \notin$ Z.
T ist jetzt ein Kreisring , der durch folgende Radien bestimmt ist.

$$r_1 = e^{\mathtt{Inf}(\mathtt{P} \ln|\mathtt{Z}|)}, \quad r_2 = e^{\mathtt{Sup}(\mathtt{P} \ln|\mathtt{Z}|)}.$$

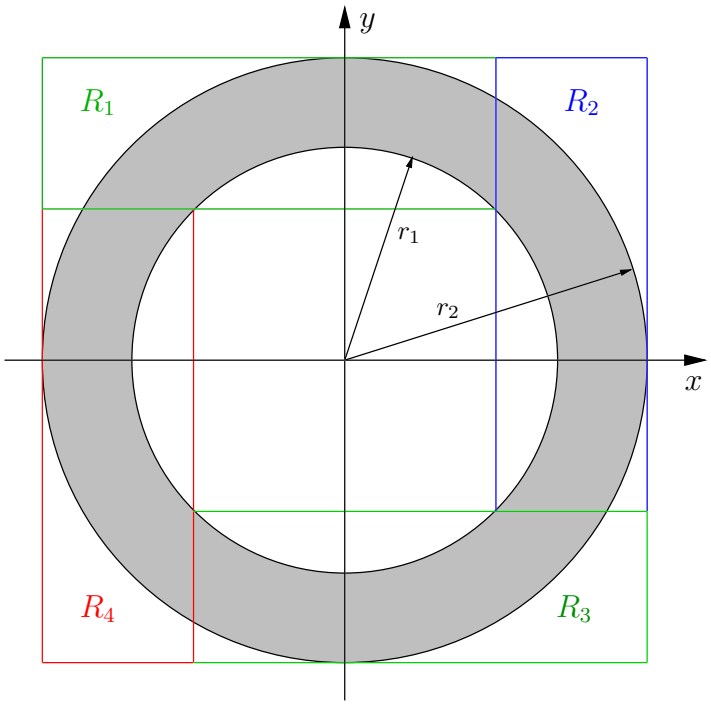


Abbildung B.23: Den Kreisring einschließende Rechtecke R_ν, $\nu = 1, 2, 3, 4$.

Die vier Rechtecke R_ν, $\nu = 1, 2, 3, 4$, schließen den Kreisring T ein und werden von der Funktion

```
std::list<MpfciClass> pow_all( const MpfciClass& Z, const MpfiClass& P )
```

in einer Liste bereitgestellt. Beachten Sie bitte, dass bei kleinen Differenzen $r_2 - r_1$ die Einschließung von T mit einem einzelnen Quadrat der Seitenlänge $2r_2$ zu großen Überschätzungen führen würde. Darüber hinaus wäre die Null ein Element der Einschließung, womit weitere Komplikationen in nachfolgenden Rechnungen verbunden sein könnten.

Das folgende Programm `MPFR-09.cpp` berechnet mit Z $= [1, 1.125] + i[1, 1.25]$ und mit dem Exponentenintervall P $= [1, 1.5]$ die vier einschließenden Intervalle R_ν, $\nu = 1, 2, 3, 4$.

```
// MPFR-09.cpp
#include "mpfciclass.hpp"

using namespace MPFR;
using namespace MPFI;
using namespace cxsc;
using namespace std;

int main(void)
{
  MPFI::MpfiClass::SetCurrPrecision(40);
  cout << "GetCurrPrecision() = " << MPFI::MpfiClass::GetCurrPrecision() << endl;
  MpfciClass Z(interval(1,1.125), interval(1,1.25), 53);
  MpfiClass  P(interval(1,1.5), 53);
  cout.precision(40/3.321928095); // Ausgabe mit 9 Dez.-Stellen
  cout << "Z = " << Z << endl;
  cout << "Z.GetPrecision() = " << Z.GetPrecision() << endl;
  cout << "P = " << P << endl;
  cout << "P.GetPrecision() = " << P.GetPrecision() << endl;
  cout << "Einschliessung aller Potenzen:" << endl;

  list<MpfciClass> res;
  res = pow_all(Z, P);

  list<MpfciClass>::iterator pos;
  // Ausgabe der n n-ten Wurzeln:
  int k(0);
  for (pos = res.begin(); pos != res.end(); ++pos )
  {
    k++;
    cout << "R" << k << " = " << *pos << endl; // Jedes Rechteck R in neue Zeile
    cout << "Praezision = " << (*pos).GetPrecision() << endl;
  }

  return 0;
}
```

Das Programm liefert die Ausgabe

```
GetCurrPrecision() = 40
Z = ([1.00000000000,1.12500000000], [1.00000000000,1.25000000000])
Z.GetPrecision() = 53
P = [1.00000000000,1.50000000000]
P.GetPrecision() = 53

Einschliessung aller Potenzen:

R1 = ([9.99999999999e-1,2.18084073549], [-1.00000000000,2.18084073549])
Praezision = 40
R2 = ([-2.18084073549,1.00000000000], [9.99999999999e-1,2.18084073549])
Praezision = 40
R3 = ([-2.18084073549,-9.99999999999e-1], [-2.18084073549,1.00000000000])
Praezision = 40
R4 = ([-1.00000000000,2.18084073549], [-2.18084073549,-9.99999999999e-1])
Praezision = 40
```

mit den vier Rechteckintervallen R_ν, $\nu = 1,2,3,4$.

Beachten Sie, dass sich nach (B.131) im Fall `P` $= [p,p]$, mit $p \in \mathbb{Q}$ und bei Punktintervallen Z nur endlich viele Potenzwerte z_k^p auf dem Kreis mit dem Radius $e^{p\cdot\ln|z|}$ befinden. Beispielsweise gilt $1^2 = 1$, aber `pow_all`(1,2) liefert wieder die vier einschließenden Rechtecke. Diese Spezialfälle werden also in `pow_all` nicht exktra behandelt!

Fall 2: $0 \in \mathtt{Z},\ p_1 > 0$.
T ist jetzt eine Kreisscheibe, die mit $\mathtt{Z} \to 0$ in den Ursprung übergeht. Diese Kreisscheibe wird jetzt mit `pow_all()` durch ein einziges Quadrat der Ergebnisliste optimal eingeschlossen.

Fall 3: $0 \in \mathtt{Z},\ p_1 \le 0$.
Da 0^p für $p \le 0$ undefiniert ist, liefert `pow_all` jetzt eine Fehlermeldung mit Programmabbruch.

Anmerkungen:

1. `pow_all()` berechnet eine Einschließung aller Potenzen z^p, mit $z \in \mathtt{Z}$ und $p \in \mathtt{P}$, wobei `P` ein reelles Intervall sein muss. Würde man im Gegensatz dazu für den Exponenten einen von Null verschiedenen Imagimärteil zulassen, so wäre in (B.131) mit $p = p_1 + i \cdot p_2$ der erste Faktor rechts gegeben durch

   $$e^{p_1 \cdot \ln|z| - p_2(\varphi + 2\pi k)}, \quad k \in \mathbb{Z},$$

   und wegen $k \in \mathbb{Z}$ würden damit die Potenzen z_k^p über alle Grenzen wachsen und damit eine sinnvolle Einschließung verhindern. Bei der Funktion `pow(Z,W)` wählt man daher $k = 0$ und berechnet damit nur Einschließungen des Hauptwertes.

C Laufzeitvergleiche

Mit der Current-Präzision `prec` = 53 werden die Laufzeiten der Langzahl-Bibliotheken `MPFR`, `MPFI`, `MPFC` und `MPFCI` jeweils verglichen mit den Laufzeiten der Arithmetiken mit den Datentypen `real`, `interval`, `complex` und `cinterval`. Dabei zeigt sich, dass die Laufzeiten mit den letzten vier Datentypen etwa zehnmal günstiger sind als mit den entsprechenden Langzahl-Bibliotheken mit der Präzision `prec` = 53.

Ein zweiter Laufzeitvergleich mit den C-XSC-Datentypen `l_real`, `l_interval`, `l_complex` und `l_cinterval` zeigt die große Überlegenheit der `MPF*`-Bibliotheken, wenn die Präzision größer als `prec` = 53 Bits, d.h. größer als 16 Dezimalstellen gewählt wird. Bei den genannten Staggered Correction Arithmetiken wird die Präzision mit der Variablen `stagprec` festgelegt, wobei z.B. `stagprec` = 15 eine Präzision von etwa $15 \cdot 16 = 240$ Dezimalstellen definiert.

$x = 200000000, \quad f(x) = \sin(x);$				
real	MPFR	`stagprec` = 2	`stagprec` = 5	`stagprec` = 15
1/10	1	177	305	844, 4.89

$x = [0.25, 0.5], \quad f(x) = \arcsin(x);$				
interval	MPFI	`stagprec` = 2	`stagprec` = 5	`stagprec` = 15
1/23	1	39	38	76

$z = 2 + i \cdot 3, \quad f(z) = e^z, \ z \in \mathbb{C}$				
complex	MPFC	`stagprec` = 2	`stagprec` = 5	`stagprec` = 15
1/9	1	125	309	838, 2.75

$Z = [1,2] + i \cdot [2,3], \quad f(z) = \arctan(z), \ z \in Z$				
cinterval	MPFCI	`stagprec` = 2	`stagprec` = 5	`stagprec` = 15
1/13	1	39	56	154

Anmerkungen:

- In der letzten Zeile bedeutet 1/13, dass die Laufzeit mit `cinterval` 13-mal kleiner ist als die Laufzeit mit `MPFCI` und der entsprechenden Präzision von `prec` = 53 Bits.
- In der letzten Zeile bedeutet 154, dass die Laufzeit mit `l_cinterval` und `stagprec` = 15 ≈ 240 Dezimalstellen ganze 154-mal größer ist als die Laufzeit mit der `MPFCI`-Bibliothek mit der Präzision von `prec` = 798 Bits, die etwa 240 Dezimalstellen entspricht.
- Der große Laufzeitvorteil der vier `MPF*`-Bibliotheken beruht auf deren direkten Hardware-Zugriff, während die Staggered-Arithmetiken den langen Akkumulator benutzen, der leider nur software-mäßig simuliert vorhanden ist.

- Die grün unterlegten Werte beziehen sich auf den Laufzeitvergleich zwichen *Mathematica* und `MPFR` bzw. `MPFC`, wobei in *Mathematica* jeweils die zu `stagprec` = 15 entsprechende Dezimalstellenzahl 240 gewählt wurde.

Literaturverzeichnis

[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] Bohlender, G.: What do we need beyond IEEE arithmetic? pp. 1-32 in: Ch. Ullrich: Contributions to Computer Arithmetic and Self-Validating Numerical Methods. J.C. Baltzer AG, Scientific Publishing Co., Basel, 1990.

[4] Alefeld G. and Herzberger J. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[5] Alefeld G. and Herzberger J. *Einführung in die Intervallrechnung.* BI, Reihe Informatik/12, 1983.

[6] Behnke H., Sommer F. *Theorie der analytischen Funktionen einer komplexen Veränderlichen.* Springer, Berlin, 1962.

[7] 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.

[8] 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.

[9] Blomquist, F.: Automatische a priori Fehlerabschätzungen zur Entwicklung optimaler Algorithmen und Intervallfunktionen in C-XSC. Universität Wuppertal, 425 Seiten, Nov. 2005. `http://www2.math.uni-wuppertal.de/~xsc/literatur/a_priori.pdf`

[10] Blomquist F.: Verbesserungen der komplexen Standardfunktionen von Markus Neher (see [43]). Interne Mitteilung, Bergische Universität Wuppertal, 2005.

[11] 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.

[12] Blomquist, F., Hofschuster, W. and Krämer, W.: *A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range.* Lecture Notes in Computer Science LNCS 5492, Springer-Verlag Berlin Heidelberg, 41-67, 2009.

[13] Blomquist, F.: Staggered Correction Computations with Enhanced Accuracy and Extremely Wide Exponent Range. Reliable Computing, Vol. 15, pp. 26-35, May, 2011.

[14] Brand, Hans-Stephan: Integration und Test einer Langzahlintervallbibliothek in C-XSC. Bachelor-Arbeit, Universität Wuppertal, 2010.

[15] 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.

[16] Braune, K.: Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:159-184, 1988.

[17] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer-Verlag, Berlin / Heidelberg / New York, 1995.

[18] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software (TOMS), Vol. 33 Issue 2, pp. June 2007.

[19] Fritzsche, A.: Grundkurs Funktionentheorie. Spektrum Akademischer Verlag Heidelberg, 2009.

[20] GNU MP LIBRARY, `http://gmplib.org/`

[21] GNU MPFR LIBRARY, `http://www.mpfr.org/mpfr-current/mpfr.html`

[22] Herzberger, J. (Ed): Topics in Validated Computations. Proceedings of IMACS-GAMM International Workshop on Validated Numerics, Oldenburg, 1993. North Holland, 1994.

[23] IBM: High-Accuracy Arithmetic Subroutine Library(ACRITH). IBM Deutschland GmbH, third edition, 1986.

[24] 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.

[25] 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.

[26] Hofschuster, W., Krämer, W., and Neher, M.: *C-XSC and Closely Related Software Packages*. Lecture Notes in Computer Science LNCS 5492, Springer-Verlag Berlin Heidelberg, 68-102, 2009.

[27] 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.

[28] Krämer, W.: Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate, Dissertation, Universität Karlsruhe, 1987.

[29] Krämer W.: Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:185-212, 1988.

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

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

[32] Krämer, W.: Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, Bericht des Instituts für Angewandte Mathematik, Universität Karlsruhe, S. 1-80, 1988.

[33] Krämer, W.: Die Berechnung von Funktionen und Konstanten in Rechenanlagen. Habilitationsschrift, Universität Karlsruhe 1993.

[34] Krämer, W.: Multiple-Precision Computations with Result Verification, in: Adams, E., Kulisch, U.(editiors): Scientific Computing with Automatic Result Verification. Academic Press, pp. 325-356, 1993.

[35] Krämer, W.; Kulisch, U.; Lohner, R.: Numerical Toolbox for Verified Computing II. Springer Verlag. Draft version, Karlsruhe, 1994.

[36] Kulisch, U.: Computer Arithmetic and Validity – Theory, Implementation and Applications. de Gruyter, Berlin, 2008.

[37] 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.

[38] Lohner, R., Wolff von Gudenberg, J.: Complex interval division with maximum accuracy. Proc. of the 7th IEEE Symposium on Computer Arithmetic in Urbana (Illinois), pp 332-336, IEEE Comp. Soc., 1985.

[39] 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.

[40] MPFR manual. April 2011.
`http://www.mpfr.org/mpfr-current/mpfr.pdf`

[41] MPFI library for arbitrary precision interval arithmetic.
`http://cadadr.org/fm/package/mpfi.html`

[42] Neher M.: The mean value form for complex analytic functions. *Computing*, 67:255-268, 2001.

[43] Neher M.: Complex Standard Functions and their Implementation in the CoStLy Library. Preprint Nr. 04/18, Universität Karlsruhe, 2004.

[44] Neher, M.: Complex Standard Functions and Their Implementation in the CoStLy Library, ACM Transactions on Mathematical Software, Vol. 33, Number 1, 27 pages, 2007.

[45] Revol, Nathalie and Rouillier, Fabrice: MPFI, a multiple precision interval arithmetic library based on MPFR. `http://perso.ens-lyon.fr/nathalie.revol/software.html`, 2001.

[46] Rotmaier, B.: Die Berechnung der elementaren Funktionen mit beliebiger Genauigkeit. Dissertation, Universität Karlsruhe, 1971.

[47] M. R. Spiegel: *KOMPLEXE VARIABLEN, Theorie und Anwendung*, Schaum's Überblicke, Aufgaben, 1991.

[48] 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.

[49] XSC website on programming languages for scientific computing with validation.
`http://www.xsc.de`

Stichwortverzeichnis