Bergische Universität
Wuppertal

Umfangreiche C-XSC-Langzahlpakete für beliebig genaue reelle und komplexe Intervallrechnung

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Preprint
BUW-WRSWT 2012/2
Nachfolger des Preprints BUW-WRSWT 2011/1

Wissenschaftliches Rechnen/
Softwaretechnologie

Diese Arbeit ist entstanden im Rahmen des Projektes
C-XSC Schnittstelle zur MPFR- und MPFI-Bibliothek

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-Zugang

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 [34, 36, 37, 65] ist eine auf **C++**basierende Programmierumgebung, mit deren selbstverifizierenden numerischen Algorithmen die exakten mathematischen Lösungen von linearen oder nichtlinearen Gleichungssystemen, Integralen, Ableitungen von praktisch beliebig komplizierten Ausdrücken, Differential- und Integralgleichungen, oder Nullstellen von differenzierbaren Funktionen, ... durch Intervalle oder Intervallvektoren Funktionsschläuche, etc. nahezu optimal eingeschlossen werden können. In C-XSC sind optimale/semimorphe (siehe [46]) 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 ca. 16 Dezimalstellen und ein Zehnerexponentenbereich von −324 bis +308 zur Verfügung stehen. Rechnungen in höherer Präzision können in C-XSC 2.5.0 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, 46]. 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 2002 entwickelten MPFR- [21] und MPFI-Bibliotheken [60] 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 3.1.0 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 [42, 49, 14, 15] 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 [16] 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 beschriebenen 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 aber überschaubar.

2. Installation

Die Installation der MPFR- und MPFI-Bibliotheken ist nur unter einem Linux/Unix-System möglich. Dazu müssen unter OpenSUSE 11.4 zunächst mit Yast2 u.a. installiert sein:

- `gcc44-c` 4.4.1_20090817-2.3.4 Der GNU **C++**-Compiler
- `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 GMP-, MPFR- und MPFI-Bibliotheken erfolgt unter OpenSUSE 11.4 durch folgende Schritte:

1. Die Dateien `mpfr-3.1.1.tar.gz, mpfi-1.5.1.tar.gz, gmp-5.0.5.tar.xz` aus dem Netz laden und ins Heimatverzeichnis kopieren. Die letzte Datei findet man zum Download unter `http://gmplib.org/`
2. Mit Yast2 `gmp-devel` deinstallieren und `m4` neu installieren. Auf meinem Rechner waren neben `make 3.82-140.1` und `gcc 4.5.1` zusätzlich noch installiert:

   ```
   libgmp10 5.0.1-4.5-x86_64    libgmp10-32bit 5.0.1-4.5-x86_64
   libgmpxx4 5.0.1-4.5-x86_64
   ```

   Ob die drei letzten Dateien auch wirklich gebraucht werden, wurde nicht getestet.
3. Die GMP-, MPFR- und MPFI-Bibliotheken deinstallieren und anschließend die entsprechenden Verzeichnisse löschen. Am Beispiel der MPFR-Bibliothek sind dazu folgende Schritte nötig
   - `cd ~/mpfr-3.1.0`
   - `make clean`
   - `gmake uninstall` (als `root`!)
   - Das Verzeichnis `~/mpfr-3.1.0` löschen

   Die GMP- und MPFI-Bibliotheken sind analog zu deinstallieren und die Ordner entsprechend zu löschen.

4. Die aktuelle gmp-Version neu installieren mit:
   - Wechseln ins Heimatverzeichnis, wo sich `gmp-5.0.5.tar.xz` befindet.
   - `gmp-5.0.5.tar.xz` entpacken, wobei das Unterverzeichnis `~/gmp-5.0.5` entsteht.
   - `cd ~/gmp-5.0.5`
   - `./configure`
   - `make`
   - `make check` **dringend** empfohlen!
   - `make install` (als `root`)
5. Die neueste Version `MPFR-3.1.1` installieren mit:
   - Wechseln in Heimatverzeichnis, wo sich `mpfr-3.1.1.tar.gz` befindet.
   - `mpfr-3.1.1.tar.gz` entpacken, wobei das Verzeichnis `~/mpfr-3.1.1` entsteht.
   - `cd ~/mpfr-3.1.1`
   - `./configure`
   - `make`
   - `make check` **dringend** empfohlen!
   - `make install` (als `root`)

   Die Installation der neuesten Version `MPFI-1.5.1` erfolgt ganz analog!

Anmerkungen:

1. Nach der beschriebenen Neuinstallation von `mpfr-3.1.1` konnte die Übersetzung eines Programms problemlos und ohne Warnungen durchgeführt werden. Beim Programmstart erfolgte jedoch die Fehlermeldung

   ```
   error while loading shared libraries: libmpfi.so.0:
   cannot open shared object file: No such file or directory
   ```

   Diese Fehlermeldung wurde jedoch beseitigt durch den `root`-Programmaufruf

   ```
   ldconfig  <enter>
   ```

   Das Programm `ldconfig` aktualisiert die Links zu allen Bibliotheken und muss nach der **manuellen** Installation von Bibliotheken als root ausgeführt werden.

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 maximale Präzision ist durch `MPFR_PREC_MAX` = 9223372036854775807 vorgegeben. Man sollte diesen großen Wert jedoch auch nicht annähernd verwenden, da Laufzeit und Speicherkapazität überfordert wären. Außerdem kann bei internen Rechnungen, z.B. zur Garantie optimaler Rundungen, die Current-Präzision etwa verdoppelt werden, so dass bei der Präzision nach oben genug Spielraum vorhanden sein sollte. 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)

Der Rundungsmodus kann global mit `SetCurrRndMode(rnd)` gesetzt werden, sonst wird mit dem Standard-Rundungsmodus `RoundNearest` gerundet. Soll abweichend von diesem globalen Rundungsmodus, z.B. bei einer Multiplikation, anders gerundet werden, so kann dies mit den entsprechenden C-Funktionen und ihrem eigenen Rundungsparameter `rnd` erfolgen, siehe etwa das Beispiel auf Seite 30.

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: $2^{+1073741822} < \texttt{MaxFloat(prec)} < 2^{+1073741823}$.
  $\texttt{expo(MaxFloat(prec))} = +1073741823$. $\texttt{mant(MaxFloat(prec))} < 1$.

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 $2^{\texttt{prec}}$ 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 $x = \texttt{m} \cdot 2^{\texttt{ex}} \geq 0$ 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 $x = \texttt{m} \cdot 2^{\texttt{ex}} \geq 0$ gilt die einfachere Darstellung:

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

Für $x \leq 0$ gilt: $\texttt{succ}(x) = -\texttt{pred}(-x)$ und $\texttt{pred}(x) = -\texttt{succ}(-x)$.

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 Standardeingabestrom "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 voreingestellten Current-Rundungsmodus, siehe das Programm auf Seite 21.

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 `RoundToNearest`, 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 30.

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 18. **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 so viele 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 25. 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 33. 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 34.

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 28

```
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 28

```
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 28

```
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 20.

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 Auf- und Abrunden die entsprechenden Anweisungen:

```
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(1.234567890123456, RoundNearest, 20),
17             b(1.234567890123456, 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 voreingestellte 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 Ergebnis `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.

3.10.6. Arithmetik-Funktionen zum Auf- und Abrunden

Um abweichend vom voreingestellten Rundungsmodus bei den arithmetischen Grundoperationen auch das Auf- oder Abrunden zu ermöglichen, werden die folgenden acht Funktionen bereitgestellt, welche die gerundeten Ergebnisse in der CurrentPrecision zurückgeben:

```
MpfrClass addd(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass addu(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass subd(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass subu(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass muld(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass mulu(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass divd(const MpfrClass& op1, const MpfrClass& op2);
MpfrClass divu(const MpfrClass& op1, const MpfrClass& op2);
```

Die Funktion `MpfrClass addd(const MpfrClass& op1, const MpfrClass& op2)` liefert also die optimal abgerundete Summe `(op1 + op2)` in der mit `void SetCurrPrecision(prec)` voreingestellten CurrentPrecision.

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 18. 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 Funktionswert, 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 32 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 32. 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: $|\texttt{m}| \in [0.5, 1)$, vgl. auch Seite 24.
`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 $\texttt{x} \cdot 2^{\texttt{k}}$ 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: $|\texttt{x}| \notin [0.5, 1)$ oder $|\texttt{k}| > 1073741823$, 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 Number_Scaling (MpfrClass& x, long int& k);
```

Obige Funktion liefert mit dem Eingabewert `x` = `a` den Rückgabewert `x` = `A` vom Typ `MpfrClass` in der Präzision von `a`, zusätzlich wird k zurückgegeben. Ist `a` unendlich oder ein NaN, so wird NaN und $k = 0$ zurückgegeben. Im Fall `a` = 0 wird ebenfalls $k = 0$ zurückgegeben. Die Berechnung von $\mathtt{A} = \mathtt{a} \cdot 2^{-k}$ wird stets **rundungsfehlerfrei** ausgeführt, wobei $2 \cdot (\mathtt{A} \cdot \mathtt{A})$ garantiert ohne Überlauf berechnet werden kann. Es gilt also stets: $\mathtt{a} = \mathtt{A} \cdot 2^k$.

```
void Number_Scaling_S (MpfrClass& x, long int& k);
```

Obige Funktion liefert mit dem Eingabewert `x` = `a` den Rückgabewert `x` = `A` vom Typ `MpfrClass` in der Präzision von `a`, zusätzlich wird k zurückgegeben. Ist `a` unendlich oder ein NaN, so wird NaN und $k = 0$ zurückgegeben. Im Fall `a` = 0 wird ebenfalls $k = 0$ zurückgegeben. Die Berechnung von $\mathtt{A} = \mathtt{a} \cdot 2^{-k}$ wird stets **rundungsfehlerfrei** ausgeführt, wobei $4 \cdot \mathtt{A}^4$ garantiert ohne Überlauf berechnet werden kann. Es gilt also stets: $\mathtt{a} = \mathtt{A} \cdot 2^k$.

```
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.

```
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 = 6.14544775142452e-1
x = 9.88050499009929e-1
```

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

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

Konstanten:

```
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.

3.11.3. Elementarfunktionen

Tabelle 3.1.: Elementarfunktionen mit `x,y,a,b` vom Typ `MpfrClass`, `n`: `unsigned long int`;

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(1+x)$	`lnp1(x)`
x^2	`sqr(x)`	$\log_2(x)$	`log2(x)`
x^2+y^2	`x2py2(x,y)`	$\log_{10}(x)$	`log10(x)`
x^2-y^2	`x2my2(x,y)`	$\ln(\sin(x))$	`ln_sin(x)`
$1/x$	`reci(x)`	$\ln(\cos(x))$	`ln_cos(x)`
$x/(x^2+y^2)$	`x_div_x2py2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
$(x^2-y^2)/(x^2+y^2)^2$	`Re_rz2(x,y)`	$\ln(\sqrt{(1+x)^2+y^2})$	`ln_sqrtxp1_2y2(x,y)`
$2xy/(x^2+y^2)^2$	`mIm_rz2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$\dfrac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2}$	`Re_r1pz2(x,y)`	x^y	`pow(x, y)`
$\dfrac{2xy}{4x^2y^2+(1+x^2-y^2)^2}$	`Im_r1pz2(x,y)`	$\sin(x)$	`sin(x)`
$\sqrt{x}$	`sqrt(x)`	$1/\sin(x)$	`csc(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_n(n)`	$\cos(x)$	`cos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$1/\cos(x)$	`sec(x)`
$\sqrt[3]{x},\quad x\in\mathbb{R}$	`cbrt(x)`	$\tan(x)$	`tan(x)`
$\sqrt[n]{x},\ n=2,3,\ldots$	`sqrt(x,n)`	$\cot(x)$	`cot(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\arctan(y/x)$	`atan2(y,x)`
$x/\sqrt{1+x^2}$	`xdsqrt1px2(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$x/\sqrt{1-x^2}$	`xdsqrt1mx2(x)`	$\sinh(x)$	`sinh(x)`
$x/\sqrt{x^2-1}$	`xdsqrtx2m1(x)`	$1/\sinh(x)$	`csch(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$\cosh(x)$	`cosh(x)`
e^x	`exp(x)`	$1/\cosh(x)$	`sech(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)`	AGM(x,y)	`agm(x,y)`

Anmerkungen:

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

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

2. 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 32.

3. Mit `sqr(x, rnd);` wird wie unter 2. verfahren, jedoch wird der exakte Funktionswert $\mathtt{x}^2$ jetzt mittels `rnd` in die Current-Precision gerundet, vgl. ebenfalls Seite 32.

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

5. Bei allen Funktionen aus obiger Tabelle wird vorausgesetzt, dass x, y, a, b exakte Maschinenzahlen sind, die also nicht schon durch vorhergehende Rechnungen gerundet worden sind. Beispielsweise kann daher eine Maschinenzahl a nicht den Wert $\sqrt{3}$ annehmen, wohl aber eine geeignete Näherung, die dann aber als exakt anzusehen ist.

6. `abs(x)` kann zusätzlich noch mit einem Präzisionsparameter `prec` aufgerufen werden. Mit `abs(x, rnd, x.GetPrecision());` wird dabei der exakte Absolutbetrag $|\mathtt{x}|$, und zwar unabhängig von `rnd`, zurückgegeben, vgl. auch Seite 33.

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

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

9. Die Funktion `agm`(x, y) rundet das exakte Arithmetisch-Geometrische Mittel $\mathrm{AGM}(x, y)$ 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:

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

Das Arithmetisch-geometrische Mittel AGM spielt bei der Auswertung elliptischer Integrale eine zentrale Rolle.

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

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()`$/32 \le |r| <$ `MaxFloat()`$/8$.
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 $(x, y \ne 0$ `&& 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 $x, y \ne 0$ 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. 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 zweimal in die Current-Präzision gerundet werden muss. Mit `rnd = RoundDown` und `rnd = RoundUp` erhält man damit für $a \cdot b + c \cdot d$ eine nahezu 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 i.a. nur 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 $\texttt{x} = 2.996\ldots \cdot 10^{322094988}$, 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` **verschiedene** 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
$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	`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 obiger Tabelle 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 32.

2. Die Funktion 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 komplexen 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 13. 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-Rundungsmodus, 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 und man erhält: `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 voreingestellten 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. Leeres Intervall

In seltenen Situationen benötigt man ein Intervall, dessen untere Grenze größer ist als sein obere Grenze. Dies wird realisiert durch die Funktion

```
MpfiClass EmptyIntVal ();
```

welche das Intervall `[9999999999.0, -9999999999.0]` erzeugt, vgl. auf Seite 58 die Member-Funktion `bool isEmpty( )`.

4.5. 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);
```

Anmerkungen:

- 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 Punktintervall. 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.

4.6. 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.

4.6.1. 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);
```

4.6.2. Subtraktion

Beachten Sie den Hinweis auf Seite 51.

```
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);
```

4.6.3. Multiplikation

Beachten Sie den Hinweis auf Seite 51.

```
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);
```

4.6.4. Division

Beachten Sie den Hinweis auf Seite 51.

```
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);
```

4.7. 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.

4.8. 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);
```

Diese Memberfunktion schließt das aktuelle Objekt mit der neuen Präzision 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.

```
const int MpfrClass::GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die Funktion ist definiert in `mpfrclass.cpp`.

```
void MpfrClass::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! Die Funktion ist definiert in `mpfrclass.cpp`.

4.9. Typ-Umwandlungen

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

4.9.1. `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.

4.9.2. 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.

4.9.3. 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`.

4.9.4. 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 30, in dem die analoge Übergabe an referenzierte Parameter vom Typ `mpfr_t` gezeigt wird.

4.9.5. `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.

4.9.6. `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 49. Dort findet man auch Hinweise auf mögliche Intervall-String-Formate.

4.9.7. 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 so viele 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.

4.9.8. 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 50. 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_Mpfi (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.

4.10. 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 MpfiClass& x);
```

Überprüft, ob x das Null-Intervall ist.

```
bool hasZero (const MpfiClass& 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.

4.11. Vergleiche

Mit den folgenden Vergleichsfunktionen werden die üblichen Vergleichsoperatoren implementiert.

4.11.1. 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.

4.11.2. 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.

4.12. 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.

4.13. 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.

4.14. Mathematische Intervall-Funktionen

Zu einer gegebenen Funktion $f : \mathbb{R} \to \mathbb{R}$ liefert eine Intervall-Funktion $f[x]$ zu einem vorgegebenen Intervall $[x] = [a, b]$ 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 `x` $\supseteq [x]$, so liefert eine implementierte Intervallfunktion, z.B. `exp(x)`, eine garantierte Einschließung aller Funktionswerte $y = f(x) = e^x$, mit $x \in [x] = [a, b] \subseteq$ `x`.

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

4.14.1. 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 49. Danach wird mit dem Argument `x` und seiner Präzision das exakte Intervall `y0` $= e^{\texttt{x}}$ 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.

4.14.2. 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`. Ohne 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 Memberfunktionen 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 analog zur gleichnamigen Funktion in C-XSC definiert.

```
int swap_endpoints();
```

Die obige Memberfunktion 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 = " << MpfrClass::GetCurrRndMode() << endl;
11   MpfiClass::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   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 = [6.14544775142451e-1,9.88050499009929e-1]
x = [2.04581622647135e-1,4.38359857288966e-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.

4.14.3. Elementarfunktionen

Tabelle 4.1.: Elementarfunktionen mit `x,y` vom Typ `MpfiClass`, `a` vom Typ `MpfrClass`;

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(x)$	`ln(x)`
x^2	`sqr(x)`	$\ln(1+x)$	`lnp1(x)`
x^2+y^2	`x2py2(x,y)`	$\log_2(x)$	`log2(x)`
x^2-y^2	`x2my2(x,y)`	$\log_{10}(x)$	`log10(x)`
$1/x$	`reci(x)`	$\ln(\sin(x))$	`ln_sin(x)`
$x/(x^2+y^2)$	`x_div_x2py2(x,y)`	$\ln(\cos(x))$	`ln_cos(x)`
$(x^2-y^2)/(x^2+y^2)^2$	`Re_rz2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
$2xy/(x^2+y^2)^2$	`mIm_rz2(x,y)`	$\ln(\sqrt{(1+x)^2+y^2})$	`ln_sqrtxp1_2y2(x,y)`
$\dfrac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2}$	`Re_r1pz2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$\dfrac{2xy}{4x^2y^2+(1+x^2-y^2)^2}$	`Im_r1pz2(x,y)`	x^y	`pow(x, y)`
$\sqrt{x}$	`sqrt(x)`	$\sin(x)$	`sin(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_N(n)`	$1/\sin(x)$	`csc(x)`
$a/\sqrt{n},\ n\in\mathbb{N}$	`xdivsqrtn(a,n)`	$\cos(x)$	`cos(x)`
$x/\sqrt{n},\ n\in\mathbb{N}$	`xdivsqrtn(x,n)`	$1/\cos(x)$	`sec(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)`	$\arg(x+i\cdot y)$	`atan2(y,x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$x/\sqrt{1+x^2}$	`xdsqrt1px2(x)`	$\sinh(x)$	`sinh(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$x/\sqrt{1-x^2}$	`xdsqrt1mx2(x)`	$1/\sinh(x)$	`csch(x)`
$x/\sqrt{x^2-1}$	`xdsqrtx2m1(x)`	$\cosh(x)$	`cosh(x)`
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$1/\cosh(x)$	`sech(x)`
e^x	`exp(x)`	$\tanh(x)$	`tanh(x)`
2^x	`exp2(x)`	$\coth(x)$	`coth(x)`
10^x	`exp10(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
e^x-1	`expm1(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
e^{x^2}	`expx2(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
$e^{x^2}-1$	`expx2m1(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
e^{-x^2}	`expmx2(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$e^{-x^2}-1$	`expmx2m1(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 64. Beachten Sie, dass z.B. für $x = [-4, 1]$ die normale Multiplikation $x \diamond x = [-4, +16]$ und `sqr`$(x) = [0, +16]$ **verschiedene** Ergebnisse liefern!

2. Alle Funktionen der Tabelle auf Seite 68 sind nach Punkt 2. 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 Arithmetisch-geometrische Mittel AGM spielt bei der Auswertung elliptischer Integrale eine zentrale Rolle.

7. Mit `u = xdivsqrtn(a,n)` wird $a/\sqrt{n}$ durch `u` in der Current-Präzision **optimal** eingeschlossen. Dabei ist a vom Typ `MpfrClass` und n vom Typ `unsigned long int`.

8. Mit `u = xdivsqrtn(x,n)` wird $x/\sqrt{n}$ durch `u` in der Current-Präzision **optimal** eingeschlossen. Dabei ist x vom Typ `MpfiClass` und n vom Typ `unsigned long int`.

9. Das Argument `n` der Funktion `sqrt_N(n)` ist vom Typ `unsigned long int`.

10. Die Funktion `MpfiClass atan2(const MpfiClass& Y, const MmpfiClass& X)` berechnet eine Einschließung der Argumente `arg(x+i*y)` für alle `x` $\in$ `X` und für alle `y` $\in$ `Y`. Falls das Intervall `(X,Y)` die negative reelle Achse schneidet, wird das Intervall $[-\pi, +\pi]$ zurückgegeben.

    Die Funktion `MpfiClass ATAN2(const MpfiClass& Y, const MmpfiClass& X)` berechnet ebenfalls eine Einschließung der Argumente `arg(x+i*y)` für alle `x` $\in$ `X` und `y` $\in$ `Y`. Falls das Intervall `(X,Y)` jedoch die negative reelle Achse von unten berührt oder schneidet, erfolgt eine entsprechende Fehlermeldung mit Programmabbruch.

4.14.4. Funktionen der Mathematischen Physik

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

Funktionsterm	Aufruf	Anmerkung
$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	`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 aus obiger Tabelle 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.

4.14.5. 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

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

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

   Abschließend noch ein Hinweis auf einen möglichen Nachteil dieses Verfahrens. Wenn einer der Randpunkte von $\texttt{a} \subseteq (\texttt{a}\cdot 2^{-k_A})\cdot 2^{k_A} = \texttt{A}\cdot 2^{k_A}$ schon selbst dicht vor dem Überlauf liegt, so wird k_A positiv sein, so dass es bei der Skalierung $\texttt{a}\cdot 2^{-k_A}$ 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 $\texttt{a} \subseteq \texttt{A}\cdot 2^{k_A}$ wird mit folgender 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 $\texttt{a} \subseteq \texttt{A}\cdot 2^{k_A}$, wobei $2\cdot(\texttt{A}\cdot\texttt{A})$ garantiert ohne Überlauf berechnet werden kann. $\texttt{a} \subset \texttt{A}\cdot 2^{k_A}$ gilt nur in den ganz seltenen Fällen, wenn bei der Berechnung von $\texttt{a}\cdot 2^{-k_A}$ ein Unterlauf eintritt, d.h. wenn gilt $\texttt{a}\cdot 2^{-k_A} \subset \texttt{A}$. Weitere Einzelheiten 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 Einzelheiten siehe Seite 72 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 Skalarproduktes $r := \mathtt{a}\cdot\mathtt{b}+\mathtt{c}\cdot\mathtt{d}$ 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.

4.15. Optimale Intervall-Einschließungen

Mit den Intervallfunktionen $f(\boldsymbol{x})$ aus den Tabellen auf Seite 68 und 71 erhält man zu einem vorgegebenen, reellen Eingangsintervall $\boldsymbol{x}$ i.a. maximalgenaue Einschließungen des entsprechenden Wertebereichs

$$W_{\boldsymbol{x}} := \{y \in \mathbb{R} \,|\, y = f(x) \wedge x \in \boldsymbol{x}\}.$$

Mit dem Funktionsaufruf $\boldsymbol{y} = f(\boldsymbol{x})$, also z.B. $\boldsymbol{y} = \mathtt{exp}(\boldsymbol{x})$, erhält man die i.a. maximalgenaue Einschließung

$$W_{\boldsymbol{x}} \subseteq \boldsymbol{y}.$$

In der Praxis müssen aber oft auch Intervallausdrücke ausgewertet werden, die aus den vier Grundoperationen zusammen mit den oben genannten Intervallfunktionen $f(\boldsymbol{x})$ aufgebaut sind. Als Beispiel betrachten wir nach [6, S. 28] die beiden vereinfachten Funktionen

$$f_1(x) = \frac{x}{1-x}, \quad x \neq 0,\ x \neq 1; \qquad f_2(x) = \frac{1}{\dfrac{1}{x} - 1}, \quad x \neq 0,\ x \neq 1.$$

Man bestätigt leicht, dass mit $\boldsymbol{x} = [2,3]$ beide Funktionen den gleichen Wertebereich $W_{\boldsymbol{x}} = [-2,-1.5]$ besitzen. Wertet man dann beide Funktionen intervallmäßig aus, so erhält man mit den Einschließungen $\boldsymbol{y}_1, \boldsymbol{y}_2$ das Ergebnis

$$\boldsymbol{y}_2 := \frac{1}{1/[2,3]-1} = \frac{1}{[-2/3,-1/2]} = [-2,-3/2] = W_{\boldsymbol{x}} \subset \boldsymbol{y}_1 := \frac{[2,3]}{1-[2,3]} = \frac{[2,3]}{[-2,-1]} = [-3,-1].$$

$f_2(x)$ liefert damit die **optimale** Einschließung $\boldsymbol{y}_2$, da die Variable x im Funktionsterm von $f_2(x)$ explizit **nur einmal** vorkommt, [6, S. 32]. Wir fassen zusammen:

> Wird ein aus stetigen Funktionen zusammengesetzter Funktionsterm über einem Intervall $\boldsymbol{x}$ intervallmäßig ausgewertet, so ist die berechnete Einschließung $\boldsymbol{y}$ i.a. **nicht optimal**, d.h. es gilt $W_{\boldsymbol{x}} \subset \boldsymbol{y}$.
> Wenn jedoch die Variable $\boldsymbol{x}$ im Funktionsterm explizit **nur einmal** vorkommt, so gilt $W_{\boldsymbol{x}} = \boldsymbol{y}$, d.h. die Einschließung ist dann **optimal**.

Anmerkungen:

1. Damit die Variable $\boldsymbol{x}$ im Funktionsterm nur einmal vorkommt, muss der ursprüngliche Funktionsterm entsprechend äquivalent umgeformt werden, vgl. obiges Beispiel und das Beispiel in [6, S. 36]. Eine solche Umformung ist jedoch nicht immer möglich!

2. In [6, S. 32] findet man für Polynome eine weitere Bedingung für eine optimale Einschließung.

3. In [6] findet man ab Seite 33 noch qualitative Aussagen über die Güte der Einschließung $\boldsymbol{y}$ im Vergleich zum Wertebereich $W_{\boldsymbol{x}}$.

4. Für die komplexe Intervallmultiplikation gilt:

   $$\boldsymbol{z} \cdot \boldsymbol{w} = (X_{\boldsymbol{z}} + i \cdot Y_{\boldsymbol{z}}) \cdot (X_{\boldsymbol{w}} + i \cdot Y_{\boldsymbol{w}}) = X_{\boldsymbol{z}} \cdot X_{\boldsymbol{w}} - Y_{\boldsymbol{z}} \cdot Y_{\boldsymbol{w}} + i \cdot (X_{\boldsymbol{z}} \cdot Y_{\boldsymbol{w}} + Y_{\boldsymbol{z}} \cdot X_{\boldsymbol{w}}).$$

   Betrachtet man z.B. den Realteil $X_{\boldsymbol{z}} \cdot X_{\boldsymbol{w}} - Y_{\boldsymbol{z}} \cdot Y_{\boldsymbol{w}}$, so kommt dort jedes der vier reellen Intervalle jeweils nur einmal vor, so dass die Intervallauswertung automatisch eine optimale Einschließung des Realteils von $\boldsymbol{z} \cdot \boldsymbol{w}$ liefert, entsprechendes gilt für den Imaginärteil. Aber beachten Sie, dass für die komplexe Division $\boldsymbol{z}/\boldsymbol{w}$ diese Überlegungen nicht mehr gelten, so dass für die optimale komplexe Intervalldivision ein neuer Algorithmus gebraucht wird.

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.

```
const int MpfrClass::GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die Funktion ist definiert in `mpfrclass.cpp`.

```
void MpfrClass::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! Die Funktion ist definiert in `mpfrclass.cpp`.

```
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 Präzision des aktuellen Objekts wird gesetzt auf das Maximum der Präzisionen 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 Präzision des aktuellen Objekts wird gesetzt auf das Maximum der Präzisionen 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 so viele 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 79.
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 80. 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 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);
```

5.9.2. Subtraktion

Beachten Sie den Hinweis auf Seite 87.

```
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 87.

```
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 87.

```
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 32.

> 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 32. 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) = atan2(y,x) i.a. gerundet zurückgegeben. Zur Definition von atan2(y,x) vgl. auch Seite 38. Für die Funktion gibt es ganz analog zur `abs(z)`-Funktion drei verschiedene Aufrufmöglichkeiten, siehe dazu Seite 90.

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

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

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

Mit $z = x + i \cdot y$ wird 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 (`±Inf`, `±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` = $x + i \cdot y$, a, b vom Typ `MpfcClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	$e^z - 1$	`expm1(z)`
$\mathrm{Re}(z) = x$	`Re(z)`	e^{z^2}	`expx2(z)`
$\mathrm{Im}(z) = y$	`Im(z)`	$e^{z^2} - 1$	`expx2m1(z)`
$\lvert z \rvert$	`abs(z)`	e^{-z^2}	`expmx2(z)`
$\arg(z)$	`arg(z)`	$e^{-z^2} - 1$	`expmx2m1(z)`
$\arg(1 + z)$	`argp1(z)`	2^z	`exp2(z)`
z^2	`sqr(z)`	10^z	`exp10(z)`
$z^2 + a \cdot z + b$	`poly2(z,a,b)`	$z^n,\ n \in \mathbb{Z}$	`power(z,n)`
$1/z$	`reci(z)`	$z^r,\ r \in \mathbb{R}$	`pow(z,r)`
$1/z^2$	`reci_z2(z)`	$z^w,\ w \in \mathbb{C}$	`pow(z,w)`
$1/(1 + z^2)$	`reci_1pz2(z)`	$\sin(z)$	`sin(z)`
$1/(1 - z^2)$	`reci_1mz2(z)`	$\cos(z)$	`cos(z)`
$\sqrt{z}$	`sqrt(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\cot(z)$	`cot(z)`
$\sqrt{z + 1} - 1$	`sqrtp1m1(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{1 + z^2}$	`sqrt1px2(z)`	$\arccos(z)$	`acos(z)`
$\sqrt{1 - z^2}$	`sqrt1mx2(z)`	$\arctan(z)$	`atan(z)`
$\sqrt{z^2 - 1}$	`sqrtx2m1(z)`	$\mathrm{arccot}(z)$	`acot(z)`
$\sqrt{-z^2 - 1}$	`sqrtmx2m1(z)`	$\sinh(z)$	`sinh(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt(z,n)`	$\cosh(z)$	`cosh(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt_all(z,n)`	$\tanh(z)$	`tanh(z)`
$\log(z)$	`ln(z)`	$\coth(z)$	`coth(z)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\log(1+z)$	`lnp1(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
$\log_2(z)$	`log2(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$\log_{10}(z)$	`log10(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
e^z	`exp(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`

Anmerkungen:

1. Bei den Funktionen `Re`, `Im` und `conj` werden die Rückgabewerte rundungsfehlerfrei in der Präzision des jeweiligen Eingabewertes zurückgegeben. Bei allen anderen Funktionen aus der Tabelle kann mit dem Rundungsparameter `rnd` entsprechend gerundet werden. Wird dieser Parameter nicht gesetzt, so wird nach dem Current-Rundungsmodus gerundet. Ist dieser nicht gesetzt, so erfolgt die Rundung mit `RoundNearest`.

2. Bei den Funktionen `abs`, `arg` und `argp1` 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.

Das nachfolgende Programm `MPFR-10.cpp` berechnet in einer Liste alle abgerundeten dritten Wurzeln aus $z = -1 + i$ und gibt den Inhalt dieser Liste auf dem Bildschirm aus.

```
// 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, 7.937005259840997373 74e-1)
        Praezision = 70
        (-1.0842150814913511818 8, 2.9051455550725144449 5e-1)
        Praezision = 70
        (2.9051455550725144449 9e-1, -1.0842150814913511818 8)
        Praezision = 70
```

mit den drei abgerundeten (`RoundDown`) dritten 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.

```
const int MpfrClass::GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die Funktion ist definiert in `mpfrclass.cpp`.

```
void MpfrClass::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! Die Funktion ist definiert in `mpfrclass.cpp`.

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 `MpfciClass 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);
```

Anmerkungen:

- 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 Punktintervall. 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, MpfciClass& 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.

```
const int MpfrClass::GetBase ();
```

Gibt die aktuelle Basis zurück, diese hat nur Einfluss auf die Ein- und Ausgabeoperatoren und auf String-Manipulationen. Die Funktion ist definiert in `mpfrclass.cpp`.

```
void MpfrClass::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! Die Funktion ist definiert in `mpfrclass.cpp`.

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 Präzision des aktuellen Objekts wird gesetzt auf das Maximum der Präzisionen 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 Präzision des aktuellen Objekts wird gesetzt auf das Maximum der Präzisionen 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 98. 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 so viele 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 99. 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 $([\texttt{NaN},\texttt{NaN}],[\texttt{NaN},\texttt{NaN}])$ 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 109 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 109 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 109 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 64.

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` = $x + i \cdot y$, a, b vom Typ `MpfciClass`; x, y vom Typ `MpfiClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	2^z	`exp2(z)`
$\mathrm{Re}(z) = x$	`Re(z)`	10^z	`exp10(z)`
$\mathrm{Im}(z) = y$	`Im(z)`	e^z	`exp(z)`
$\lvert z \rvert$	`abs(z)`	$e^z - 1$	`expm1(z)`
$\mathrm{Arg}(z)$	`Arg(z)`	e^{z^2}	`expx2(z)`
$\arg(z)$	`arg(z)`	$e^{z^2} - 1$	`expx2m1(z)`
$\arg(1 + z)$	`argp1(z)`	e^{-z^2}	`expmx2(z)`
z^2	`sqr(z)`	$e^{-z^2} - 1$	`expmx2m1(z)`
$z^2 + a \cdot z + b$	`poly2(z,a,b)`	z^n, $n \in \mathbb{Z}$	`power_fast(z,n)`
$1/z$	`reci(z)`	z^n, $n \in \mathbb{Z}$	`power(z,n)`
$1/z^2$	`reci_z2(z)`	z^p, p : `MpfiClass`	`pow(z,p)`
$1/(1 + z^2)$	`reci_1pz2(z)`	z^p, p : `MpfiClass`	`pow_all(z,p)`
$1/(1 - z^2)$	`reci_1mz2(z)`	z^w, w : `MpfciClass`	`pow(z,w)`
$\sqrt{z}$	`sqrt(z)`	$\sin(z)$	`sin(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\cos(z)$	`cos(z)`
$1/\sqrt{z}$	`sqrt_r(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z + 1} - 1$	`sqrtp1m1(z)`	$\cot(z)$	`cot(z)`
$\sqrt{1 + z^2}$	`sqrt1px2(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{1 - z^2}$	`sqrt1mx2(z)`	$\arccos(z)$	`acos(z)`
$\sqrt{z^2 - 1}$	`sqrtx2m1(z)`	$\arctan(z)$	`atan(z)`
$\sqrt{-z^2 - 1}$	`sqrtmx2m1(z)`	$\mathrm{arccot}(z)$	`acot(z)`
$\sqrt[n]{z}$	`sqrt(z,n)`	$\sinh(z)$	`sinh(z)`
$\sqrt[n]{z}$	`sqrt_all(z,n)`	$\cosh(z)$	`cosh(z)`
$\log(z)$	`Ln(z)`	$\tanh(z)$	`tanh(z)`
$\log(1 + z)$	`Lnp1(z)`	$\coth(z)$	`coth(z)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$\log(z)$	`ln(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
$\log(1+z)$	`lnp1(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$\log_2(z)$	`log2(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
$\log_{10}(z)$	`log10(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`

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, d.h. das exakte, reelle Ergebnisintervall, durch ein Intervall der Präzision `prec` garantiert eingeschlossen. Ohne `prec` wird das exakte, reelle Argumentintervall durch ein Intervall mit der Current-Präzision eingeschlossen. Enthält das komplexe Eingangsintervall `z` eine negative, reelle Zahl -auch auf dem Intervallrand!-, so erfolgt eine entsprechende Fehlermeldung. Nicht erlaubte Intervalle sind demnach
   `z` = $[-2,-1]+i\cdot[-1,0]$ oder `z` = $[-1,-1]+i\cdot[0,1]$ oder `z` = $[-1,+1]+i\cdot[0,0]$.
   `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, d.h. das exakte, reelle Ergebnisintervall, 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 `z` $\in \mathbb{IC}$ eine Einschließung des exakten Argumentintervalls, dabei gilt: `arg(z)` $\subset [-\pi, 3\pi/2]$. Hier einige Beispiele:
   `arg`$([-2,-1]+i\cdot[-2,-1]) = [-2.67795,-2.03444]$,
   `arg`$([-2,+1]+i\cdot[-2,+2]) = [-3.14160,+3.14160]$,
   `arg`$([-0,+0]+i\cdot[-0,+0]) = [-0.00000,+0.00000]$,
   `arg`$([-2,-1]+i\cdot[+0,+1]) = [+2.35619,+3.14160]$,
   `arg`$([-2,-1]+i\cdot[-1,-0]) = [-3.14160,-2.35619]$,
   `arg`$([-2,-1]+i\cdot[-0,+0]) = [+3.14159,+3.14160]$,
   `arg`$([-0,+0]+i\cdot[-1,+1]) = [-1.57080,+1.57080]$,
   `arg`$([-1,+0]+i\cdot[-1,+1]) = [+1.57079,+4.71239]$.

5. `argp1(z)` berechnet $\arg(1+z)$ und kann mit einem zusätzlichen Präzisionsparameter `prec` aufgerufen werden. Dann wird das exakte, reelle Argumentintervall, d.h. das exakte, reelle Ergebnisintervall, durch ein Intervall der Präzision `prec` garantiert eingeschlossen. Ohne `prec` wird das exakte, reelle Argumentintervall durch ein Intervall mit der Current-Präzision eingeschlossen. `argp1(z)` liefert für jedes `z` $\in \mathbb{IC}$ eine Einschließung des exakten Argumentintervalls, dabei gilt: `argp1(z)` $\subset [-\pi, 3\pi/2]$. Hier einige Beispiele:
   `argp1`$([-3,-2]+i\cdot[-2,-1]) = [-2.67795,-2.03444]$,
   `argp1`$([-3,+0]+i\cdot[-2,+2]) = [-3.14160,+3.14160]$,

$$\texttt{argp1}([-1,-1]+i\cdot[-0,+0]) = [-0.00000,+0.00000],$$
$$\texttt{argp1}([-3,-2]+i\cdot[+0,+1]) = [+2.35619,+3.14160],$$
$$\texttt{argp1}([-3,-2]+i\cdot[-1,-0]) = [-3.14160,-2.35619],$$
$$\texttt{argp1}([-3,-2]+i\cdot[-0,+0]) = [+3.14159,+3.14160],$$
$$\texttt{argp1}([-1,-1]+i\cdot[-1,+1]) = [-1.57080,+1.57080],$$
$$\texttt{argp1}([-2,-1]+i\cdot[-1,+1]) = [+1.57079,+4.71239].$$

6. Beim Aufruf von `sqr(z)`, mit `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 287. 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 $\texttt{sqr}(z) = [-25,9] + i\cdot[-20,30]$.

7. `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, [54].
   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.

8. 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 Verzweigungsschnitt ganz im Innern enthalten oder diesen von oben oder unten berühren, vgl. die Abbildungen C.17, C.18 auf den Seiten 294 und 295.

9. 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.

10. Mit `pow_all(Z,P)` und $z = |z|\cdot e^{i\varphi},\ -\pi/2 < \varphi < +3\pi/2$ wird eine Einschließung der komplexen Menge

$$\{y \in \mathbb{C} \,|\, y = e^{p\cdot\ln(z)} = e^{p\cdot\ln|z|}\cdot e^{ip(\varphi+2\pi k)},\ z \in \texttt{Z:MpfciClass},\ p \in \texttt{P:MpfiClass},\ k \in \mathbb{Z}\}$$

    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 367.

11. 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.

12. Bei der Auswertung von $z^2 + a \cdot z + b$ mit `poly2(..)` sollte für a möglichst nur ein Punktintervall gewählt werden, um eine optimale Einschließung zu gewährleisten, vgl. S. 291.

13. Die mit Magenta gedruckten Funktionsaufrufe, wie z.B. expx2 und expmx2, beziehen sich auf Implementierungen, die mithilfe der *Globalen Optimierung* realisiert wurden. Dadurch erhält man zwar optimale Einschließungen der Wertemengen, aber bei zu breiten Eingangsintervallen oder bei zu groß gewählten Präzisionen muss mit größeren Laufzeiten gerechnet werden, vgl. Abschnitt 7.4.1.

6.12.4. Optimale, komplexe Intervall-Einschließungen

Ab Seite 243 wird ausführlich beschrieben, wie ein Algorithmus zu realisieren ist, mit dem man für eine vorgegebene holomorphe Funktion $f(z),\ z \in \mathbb{C}$, zu einem komplexen Rechteck-Intervall Z den zugehörigen Wertebereich $W_Z := \{w \in \mathbb{C} \,|\, w = f(z),\ z \in Z\}$ durch $F(Z)$ **optimal** einschließt. Die bisher realisierten Funktionen mit optimaler Einschließung sind in der Tabelle auf Seite 115 zusammengestellt.

Am Beispiel der Funktion $f(z) = \sqrt{-z^2 - 1}$ soll hier noch ausdrücklich betont werden, dass zu einem vorgegebenen Intervall Z die berechnete Einschließung $F_1(Z)$ bei naiver intervallmäßiger Auswertung des Funktionsterms $\sqrt{-z^2 - 1}$ durch `sqrt(-sqr(`z`) ⟠ 1)` i.a. viel gröber ausfällt, als bei Auswertung der Funktion `sqrtmx2m1(Z)` aus der Tabelle auf Seite 115, welche die optimale Einschließung $W_Z \subseteq F(Z) \subset F_1(Z)$ des Wertebereichs W_Z liefert, vgl. Seite 320.

In der Praxis müssen aber oft auch Einschließungen komplexer Intervallausdrücke berechnet werden, die aus den vier Grundoperationen zusammen mit den Intervallfunktionen aus der Tabelle von Seite 115 aufgebaut sind. Im Gegensatz zu einem reellen Intervallausdruck reicht es jetzt jedoch nicht aus, den komplexen Ausdruck äquivalent so umzuformen, dass dieser die Variable Z nur einmal enthält, um danach eine optimale Einschließung des Wertebereichs berechnen zu können, vgl. Seite 74. Als Beispiel betrachten wir wieder die obige Funktion $f(z) = \sqrt{-z^2 - 1}$, in deren Funktionsterm die Variable z zwar nur einmal vorkommt, die aber bei naiver Intervallauswertung nach Seite 320 **keine optimale** Einschließung liefert.

Wie kompliziert schon bei der scheinbar einfachen Funktion $f(z) := e^{z^2} = u(x,y) + i \cdot v(x,y)$ die Realteilfunktion $u(x,y)$ über dem Eingangsintervall $Z = [2,4] + i \cdot [0,3]$ ausfällt, zeigt die folgende mit *Mathematica* erstellte Abbildung 6.1

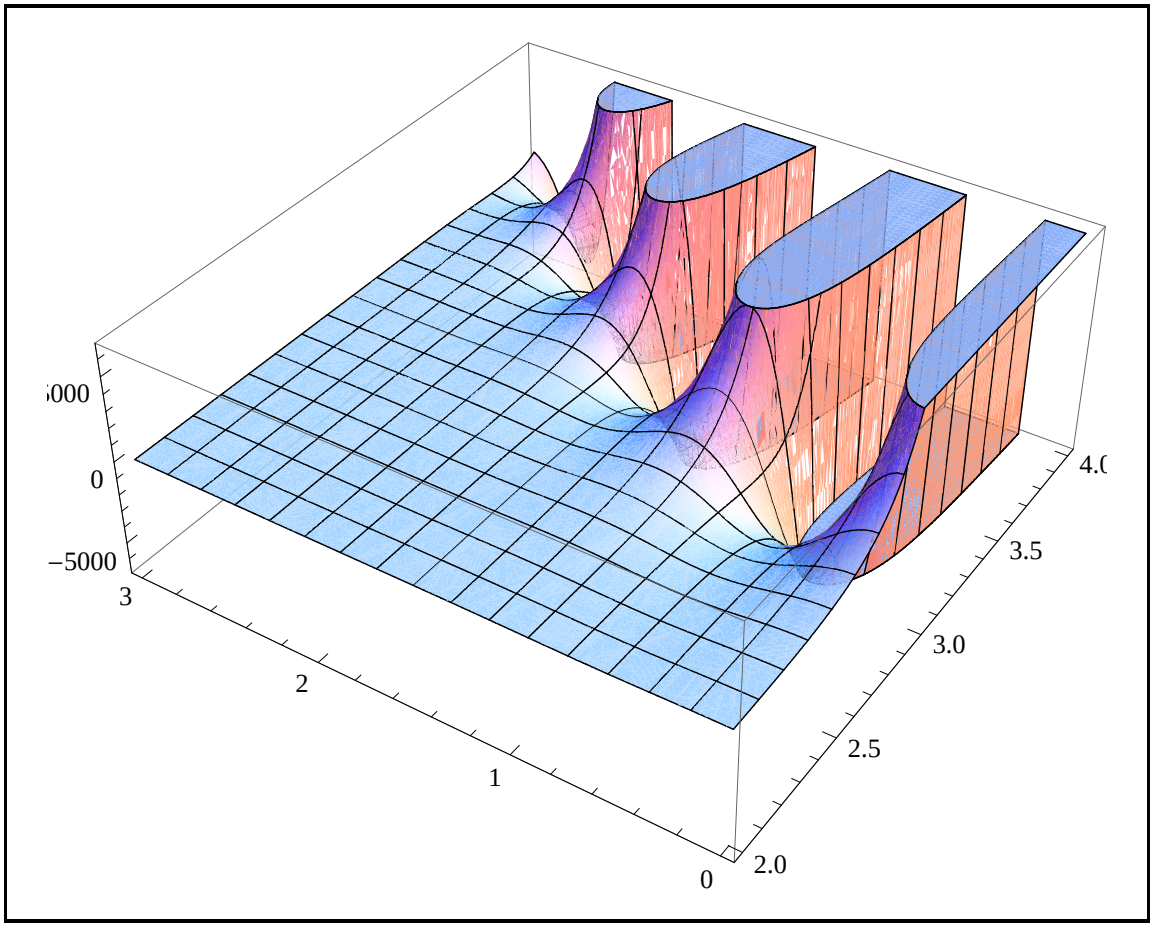


Abbildung 6.1.: Realteil-Funktion $u(x,y)$ von $f(z) := e^{z^2}$ über $Z = [2,4] + i \cdot [0,3]$

.

Man erkennt mit wachsenden x, y-Werten aufeinanderfolgende Maxima und Minima, bedingt durch die 2π-periodischen trigonometrischen Funktionen der komplexen Exponentialfunktion. Es wird daher sehr aufwendig oder sogar unmöglich sein, auf dem Rand eines gegebenen Eingangsintervalls Z die Lage des Maximums und Minimums, z.B. der Realteilfunktion $u(x, y)$, zu bestimmen. Bei komplizierten Intervallausdrücken wird es daher kaum möglich sein, eine optimale Einschließung des Wertebereichs W mit vertretbarem Aufwand zu berechnen. Wir fassen zusammen:

Um den Wertebereich W eines aus holomorphen Funktionen zusammengesetzten Intervallausdrucks möglichst optimal einzuschließen, sollte man auf möglichst viele Teilfunktionen zurückgreifen können, deren Wertebereiche selbst jeweils optimal eingeschlossen werden, vgl. dazu die Tabelle auf Seite 115.

Die Einschließung des Wertebereichs W eines komplexen Intervallausdrucks ist i.a. ein eigenständiges Problem, das nur in Ausnahmefällen optimal gelöst werden kann. Selbst wenn die Variable Z im Funktionsterm nur einmal vorkommt, wird bei naiver Intervallauswertung für W i.a. keine optimale Einschließung berechnet. Eine optimale Einschließung von W wird jedoch erreicht, wenn in den Real- und Imaginärteilfunktionen von $f(z) = u(x, y)+i\cdot v(x, y)$ die Variablen x, y jeweils nur einmal vorkommen, vgl. z.B. die separable, komplexe Exponentialfunktion auf Seite 244.

Abschließend noch ein beruhigender Hinweis aus der numerischen Praxis:

Die erfolgreiche Einschließung einer komplexen Nullstelle z_0, z.B. nach dem Newton-Verfahren, liefert eine Folge von Einschließungsintervallen $Z_k \ni z_0$, deren Durchmesser für $k \to \infty$ gegen Null streben, so dass die Intervalle Z_k selbst nicht wirklich optimal berechnet werden müssen, da die auftretenden Überschätzungen mit $k \to \infty$ vernachlässigbar klein werden, vgl. Seite 155.

7. Anwendungen

7.1. 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 $J_0(x)$, [20], [37]. 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 $\texttt{x} \subseteq \alpha$ 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);
```

```
42 }
43
44 int main(void)
45 {
46   PrecisionType prec = 6000;
47   MpfrClass::SetCurrPrecision(prec);
48   MpfiClass x, xOld, xMid, fxMid, fx, dfx;
49   x = Start_Interval();
50   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.

7.2. Einschließung reeller arithmetischer Ausdrücke

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

$$f(x) := \tan(x) - \sin(x) \equiv 2 \cdot \tan(x) \cdot \sin^2(x/2), \qquad \text{mit:} \quad x_0 = 2^{-16000}.$$

```
// MPFR-12.cpp
// Inclusion of f(x) := tan(x) - sin(x); x = 2^(-16000);
//                  f(x)  = 2*tan(x)*sqr( sin(x/2) );
#include "mpficlass.hpp"

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

int main(void)
{
  PrecisionType prec = 600, prec_old = prec;
  MpfrClass::SetCurrPrecision(prec);
  MpfiClass x(exp2(MpfiClass(-16000))), y; // x = 2^(-16000)
  // Evaluating the difference expression
  y = tan(x) - sin(x);
  cout.precision(prec/3.321928095);
  cout << "f(x) included by " << y << endl;
  // Evaluation with fifty-fivefold Precision
  MpfrClass::SetCurrPrecision(55*prec);
  y = tan(x) - sin(x);
  y.RoundPrecision(prec_old);
  cout << "f(x) included by " << y << endl;
  // Evaluating the simplified expression
  MpfrClass::SetCurrPrecision(prec_old);
  y = 2*tan(x)*sqr( sin(x/2) ); // product expression
  cout << "f(x) included by " << y << endl;

  return 0;
}
```

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 auftretender 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!

7.3. Automatische Differentiation

Die Grundlagen der Automatischen Differentiation werden beschrieben in [20], so dass hier nur eine kurze Zusammenfassung angegeben wird.

Bei vielen wissenschaftlichen Anwendungen ist die Berechnung von Ableitungen komplizierter Funktionsausdrücke ein zentrales Problem. Beispiele sind die Einschließung der Nullstellen oder der relativen Extrema solcher Ausdrücke. Es gibt dabei drei verschieden Methoden, um diese Ableitungen zu berechnen: Die *numerische* Differentiation, die *symbolische* Differentiation und die *automatische* Differentiation.

Die *numerische* Differentiation benutzt Differenzapproximationen zur Berechnung der Ableitungen, die *symbolische* Differentiation berechnet explizite Formeln für diese Ableitungen und die *automatische* Differentiation benutzt ebenfalls diese Formeln, die jedoch automatisch intervallmäßig ausgewertet werden, so dass garantierte Einschließungen dieser Ableitungen berechnet werden. Der Vorteil der *automatischen* Differentiation besteht darin, dass nur der aktuelle Funktionsausdruck selbst, nicht aber die Formeln für die oft sehr komplizierten Ableitungen dieser Ausdrücke explizit angegeben werden müssen.

In der in `mpficlass.cpp` implementierten Klasse `MPDeriveType` sind alle notwendigen arithmetischen Operatoren und Funktionen definiert, um für einen entsprechenden Funktionsausdruck den Funktionswert selbst und die beiden ersten Ableitungen berechnen zu können. Das folgende Programm zeigt, wie man für $f(x) := e^{\cos(x)} \cdot (1 + \sin(x))$ z.B. an der Stelle $x_0 = \sqrt{2}$ die Einschließungen des Funktionswertes $f(x_0)$ und der beiden Ableitungen $f'(x_0)$, $f''(x_0)$ mit der Präzision `prec` = 7000, d.h. also mit nahezu 2107 korrekten Dezimalstellen, sehr einfach berechnen kann.

```
1  // MPFR-14.cpp
2  // f(x) = exp(cos(x))*(1+sin(x));
3  // Berechnung der Einschliessungen von
4  // f(x0), f'(x0), f''(x0) fuer x0 = sqrt(2)
5  // mit einer Praezisison von 7000 bits, d.h.
6  // mit nahezu 2107 Dezimalstellen
7
8  #include "mpficlass.hpp"
9
10 using namespace MPFR;
11 using namespace MPFI;
12 using namespace cxsc;
13 using namespace std;
14
15 MPDerivType f(const MPDerivType& x) // f(x)
16 {
17   return exp(cos(x))*(1+sin(x));
18 }
19
20 int main(void)
21 {
22   PrecisionType prec = 7000;
23   MpfrClass::SetCurrPrecision(prec);
24   MpfiClass x(2), fx, dfx, ddfx;
25   x = sqrt(x);
26   ddfEval(f, x, fx, dfx, ddfx);
27
28   cout.precision(x.GetPrecision()/3.321928095);  // output format
29   cout << "x0   in " << x << endl;
30   cout << "fx   in " << fx << endl;
31   cout << "dfx  in " << dfx << endl;
32   cout << "ddfx in " << ddfx << endl;
33
34   return 0;
35 }
```

Das Programm liefert die hier auf nur wenige Dezimalstellen verkürzte Ausgabe

$$x_0 \in [1.41421356237309504880168872 \ldots 88360, 1.41421356237309504880168872 \ldots 88361]$$
$$fx \in [2.32322211003733719499299837 \ldots 68783, 2.32322211003733719499299837 \ldots 68785]$$
$$dfx \in [-2.11253887115066778267776671 \ldots 41006, -2.11253887115066778267776671 \ldots 41003]$$
$$ddfx \in [0.38990937530215173154383054 \ldots 39325, 0.38990937530215173154383054 \ldots 39359]$$

Anmerkungen:

1. Beachten Sie, dass $x_0 = \sqrt{2}$ nicht darstellbar ist und daher oben durch das echte Intervall $\mathtt{x} = [1.4142 \ldots 88360, 1.4142 \ldots 88361]$ eingeschlossen werden muss.

2. Aber auch bei Punktintervallen `x` werden die berechneten Einschließungen von `fx` = $f(x_0)$, `dfx` = $f'(x_0)$, `ddfx` = $f''(x_0)$ wegen der internen Intervallauswertungen grundsätzlich leicht überschätzt sein. Wenn jedoch, wie z.B. im Fall `x` $\approx 1.5\pi$, stärkere Auslöschungen auftreten, so werden die berechneten Einschließungen entsprechend grob ausfallen. Die damit verbundene geringere Genauigkeit kann jedoch durch die Wahl einer höheren Präzision `prec` stets korrigiert werden. Mit dieser höheren Präzision darf natürlich der Werte des Eingangsintervalls `x` $\approx 1.5\pi$ *nicht* geändert werden.

Mit den folgenden Funktionen können zu einer vordefinierten Funktion f vom Typ `MPDerivType` und zu einem gegebenen Eingangsintervall `x` vom Typ `MpfiClass` garantierte Einschließungen von $f(\mathtt{x}), f'(\mathtt{x})$ oder $f''(\mathtt{x})$ berechnet werden:

```
void fEval   ( MPddf_FctPtr f, MpfiClass x, MpfiClass& fx );

void dfEval  ( MPddf_FctPtr f, MpfiClass x, MpfiClass& fx, MpfiClass& dfx );

void ddfEval( MPddf_FctPtr f, MpfiClass x, MpfiClass& fx, MpfiClass& dfx,
                                                          MpfiClass& ddfx );
```

`MPddf_FctPtr f` ist dabei ein vordefinierter Zeiger auf eine Funktion `f` vom Typ `MPDerivType`.

Hat man mit den Deklarationen

```
MPDerivType x, fx;
MpfiClass y, fy, dfy, ddfy;
```

und mit den Anweisungen `y = 123.0; x = DerivVar(y); fx = f(x);` eine Funktion `f` vom Typ `MPDerivType` z.B. an der Stelle `y = 123.0` ausgewertet, so kann man mit den Funktionen

```
const MpfiClass fValue   ( const MPDerivType& fx );

const MpfiClass dfValue  ( const MPDerivType& fx );

const MpfiClass ddfValue( const MPDerivType& fx );
```

und den Anweisungen `fy = fValue(fx); dfy = dfValue(fx); ddfy = ddfValue(fx);` Einschließungen von $f(\mathtt{x}), f'(\mathtt{x})$ und $f''(\mathtt{x})$ berechnen.

Die nachfolgende Funktion liefert mit einem Objekt vom Typ `MpfiClass` eine Variable vom Typ `MPDerivType`. Ein Objekt vom Typ `MPDerivType` ist ein Tripel von `MpfiClass`-Intervallen, welche den Funktionswert und die beiden ersten Ableitungen einschließen. Bei einer Variablen `u` vom Typ `MPDerivType` lautet daher das Tripel: `u = (y, [1,1], [0,0])`.

```
MPDerivType DerivVar( const MpfiClass& y );
```

Für `y` sind folgende Typen möglich: `MpfiClass, MpfrClass, interval, real, double.`

Mit der folgenden Funktion wird aus einem Objekt vom Typ `MpfiClass` eine Konstante vom Typ `MPDerivType` generiert. Bei einer Konstanten `c` vom Typ `MPDerivType` lautet daher mit der Anweisung `c = DerivConst(y)` das Tripel: `c = (y, [0,0], [0,0])`.

```
MPDerivType DerivConst( const MpfiClass& y);
```

Für `y` sind folgende Typen möglich: `MpfiClass, MpfrClass, interval, real, double`. Die beiden letzten Funktionen liefern `MPDerivType`-Objekte, d.h. die oben beschriebenen Tripel von `MpfiClass`-Objekten, jeweils in der vordefinierten Current-Precision.

Wertzuweisungen an ein Objekt vom Typ `MPDerivType` erfolgen mit Hilfe der Operatoren

```
MPDerivType& operator = ( const MPDerivType& u);

MPDerivType& operator = ( const MpfiClass& u);

MPDerivType& operator = ( const MpfrClass& u);

MPDerivType& operator = ( const interval& u);

MPDerivType& operator = ( const double& u);

MPDerivType& operator = ( const real& u);

MPDerivType& operator = ( int u);
```

Der linke Operand vom Typ `MPDerivType` wird dabei stets in der Current-Precision generiert. Beachten Sie, dass bis auf den 1. Operator stets **Konstanten** vom Typ `MPDerivType` erzeugt werden, d.h. mit `u = 1.25;` erhält man das Tripel `( [1.25,1.25], [0,0], [0,0] )` von `MpfiClass`-Intervallen in der voreingestellten Current-Precision.

Mit den Funktionen

```
static void  SetCurrPrecision (PrecisionType prec);

static const PrecisionType GetCurrPrecision ();
```

kann die Current-Precision mit `prec` gesetzt oder abgerufen werden.

Die Memberfunktion

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

setzt die Präzision des aktuellen Objekts vom Typ `MPDerivType` auf `prec` und löscht dabei das aktuelle Tripel der `MpfiClass`-Intervalle.

Die Memberfunktion

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

setzt die Präzision des aktuellen Objekts vom Typ `MPDerivType` auf `prec` und rundet dabei das aktuelle Tripel der `MpfiClass`-Intervalle in die neue Präzision.

Mit den beiden folgenden Funktionen wird die Basis nur für die Ein- und Ausgabe gesetzt oder abgerufen.

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

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

Der voreingestellte Standardwert ist $b = 10$.

In der nachfolgenden Tabelle sind alle Funktionen vom Typ `MPDerivType` aufgelistet, für die Einschließungen ihres Funktionswertes und ihrer beiden ersten Ableitungen berechnet werden können. Wie im Programm `MPFR-14.cpp` können diese Funktionen beliebig verschachtelt und mit den vier Grundoperationen verknüpft werden. Damit kann man fast beliebig komplizierte Funktionen generieren, um deren Funktionswerte und die beiden ersten Ableitungen in fast beliebiger Genauigkeit einzuschließen.

Tabelle 7.1.: Funktionen vom Typ `MPDerivType` zur Automatischen Differentiation

Funktion	Aufruf	Funktion	Aufruf
x^2	`sqr(x)`	$\ln(\cos(x))$	`ln_cos(x)`
$x^n,\ n \in \mathbb{Z}$	`power(x,n)`	$\sin(x)$	`sin(x)`
x^y	`pow(x, y)`	$\cos(x)$	`cos(x)`
$1/x$	`reci(x)`	$\tan(x)$	`tan(x)`
$\sqrt{x}$	`sqrt(x)`	$\cot(x)$	`cot(x)`
$\sqrt[3]{x},\ \ x \in \mathbb{R}$	`cbrt(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt[n]{x},\ n = 2, 3, \ldots$	`sqrt(x,n)`	$\arccos(x)$	`acos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\sinh(x)$	`sinh(x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\cosh(x)$	`cosh(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\tanh(x)$	`tanh(x)`
e^x	`exp(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)`
2^x	`exp2(x)`	$1/\sin(x)$	`csc(x)`
10^x	`exp10(x)`	$1/\cos(x)$	`sec(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$\ln(x)$	`ln(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\cosh(x)$	`sech(x)`
$\log_2(x)$	`log2(x)`	$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$	`erf(x)`
$\log_{10}(x)$	`log10(x)`	$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,dt$	`erfc(x)`
$\ln(\sin(x))$	`ln_sin(x)`		

Wenn man bei der praktischen Anwendung der Funktionen aus obiger Tabelle neue Ausdrücke generiert, so sollte man unbedingt Teilausdrücke, die in obiger Tabelle bereits zu finden sind, auch **direkt** benutzen, um starke Überschätzungen bei den Einschließungen von Funktionswert und den beiden ersten Ableitungen zu vermeiden. Als Beispiel betrachten wir analog zum Programm `MPFR-14.cpp` auf Seite 125 den Ausdruck `g`

```
MPDerivType g(const MPDerivType& x) // g(x)
{
  return sqrt(1-x*x);
}
```

der unbedingt zu ersetzen ist durch den äquivalenten Ausdruck `f`

```
MPDerivType f(const MPDerivType& x) // f(x)
{
  return sqrt1mx2(x);
}
```

Für `g(x)` erhält man mit den Anweisungen

```
int main(void)
{
  PrecisionType prec = 100;
  MpfrClass::SetCurrPrecision(prec);
  MpfrClass r(1);
  r = pred(r);
  MpfiClass x(r), gx, dgx, ddgx;
  ddfEval(g, x, gx, dgx, ddgx);

  cout.precision(x.GetPrecision()/3.321928095);  // output format
  cout << "x0   in " << x << endl;
  cout << "gx   in " << gx << endl;
  cout << "dgx  in " << dgx << endl;
  cout << "ddgx in " << ddgx << endl;

  return 0;
}
```

für das Argument `x0 = pred(1)` = $1 - 2^{-100}$ die folgenden sehr groben Einschließungen:

```
x0   in [9.99999999999999999999999999999e-1,1.00000000000000000000000000000]
gx   in [8.88178419700125232338905334472e-16,1.25607396694702004751470589758e-15]
dgx  in [-1.12589990684262400000000000000e15,-7.96131459065721570577976794814e14]
ddgx in [-1.42724769270595988105828596945e45,-5.04608260972618997395690666018e44]
```

Für den optimalen Ausdruck `f(x)` erhält man mit den analogen Anweisungen

```
int main(void)
{
  PrecisionType prec = 100;
  MpfrClass::SetCurrPrecision(prec);
  MpfrClass r(1);
  r = pred(r);
  MpfiClass x(r), fx, dfx, ddfx;
  ddfEval(f, x, fx, dfx, ddfx);

  cout.precision(x.GetPrecision()/3.321928095);  // output format
  cout << "x0   in " << x << endl;
  cout << "fx   in " << fx << endl;
  cout << "dfx  in " << dfx << endl;
  cout << "ddfx in " << ddfx << endl;

  return 0;
}
```

für das Argument `x0 = pred(1)` = $1 - 2^{-100}$ die folgenden nahezu optimalen Einschließungen:

```
x0   in [9.99999999999999999999999999999e-1,1.00000000000000000000000000000]
fx   in [1.25607396694702004751470589756e-15,1.25607396694702004751470589758e-15]
dfx  in [-7.96131459065721570577976794816e14,-7.96131459065721570577976794813e14]
ddfx in [-5.04608260972618997395690666022e44,-5.04608260972618997395690666018e44]
```

Ganz analog sollte man zur Vermeidung von Auslöschungen z.B. den Ausdruck $g(x) = e^{2x} - e^x$ ersetzen durch $f(x) = e^x \cdot (e^x - 1)$, wobei $e^x - 1$ mit der Funktion `expm1(x)` aus der Tabelle von Seite 128 auszuwerten ist. Die Auswahl des geeigneten Ausdrucks ist daher für eine optimale Einschließung des Funktionswertes und der beiden ersten Ableitungen von entscheidender Bedeutung. In Ausnahmefällen müssen die Algorithmen für den Funktionswert und die Ableitungen neu erstellt werden. Wie dabei vorzugehen ist, kann man an einfachen Beispielen, wie z.B. $\sin(x)$, $\cos(x)$, $\sqrt{1-x^2}$, in der Datei `mpficlass.cpp` ablesen.

7.4. Globale Optimierung

Bei der eindimensionalen globalen Optimierung werden für eine zweimal stetig differenzierbare Funktion $f : \mathbb{R} \to \mathbb{R}$ über einem Intervall $[x] \in \mathbb{IR}$ alle Lösungen x^* berechnet von

$$\min_{x \in [x]} f(x). \tag{7.1}$$

Dabei werden die Minimumstellen x^* und der minimale Funktionswert $f^* := f(x^*)$ garantiert eingeschlossen. Der verwendete Algorithmus basiert auf Hansen, [27], [28], [29], Ratscheck, [31] und Ratz, [59], weitere Einzelheiten findet man in [20]. Die Folgenden Abbildungen zeigen die

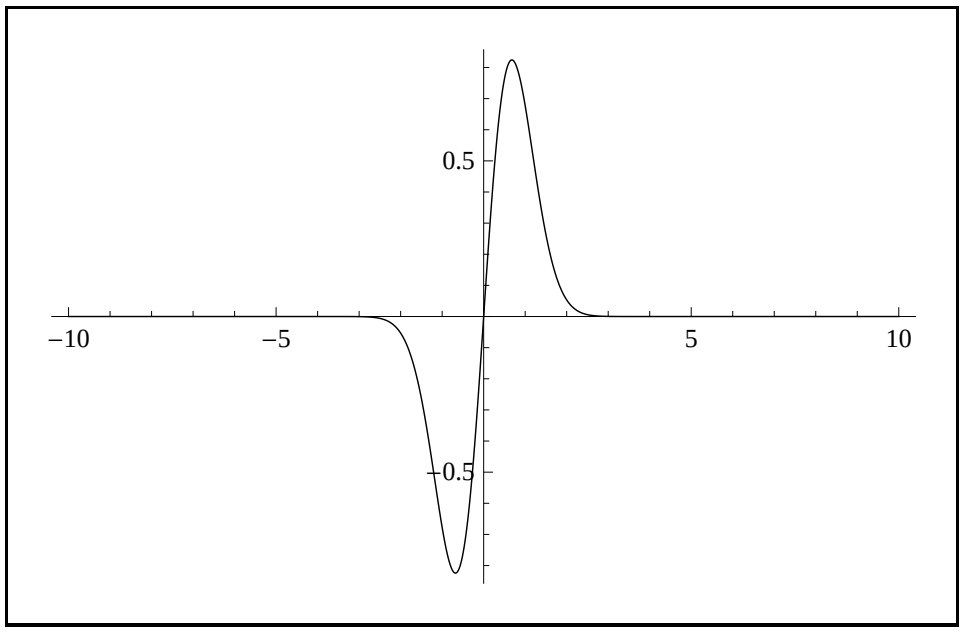


Abbildung 7.1.: $f(x) := (x + \sin(x)) \cdot e^{-x^2},\ x \in [-10, 10]$

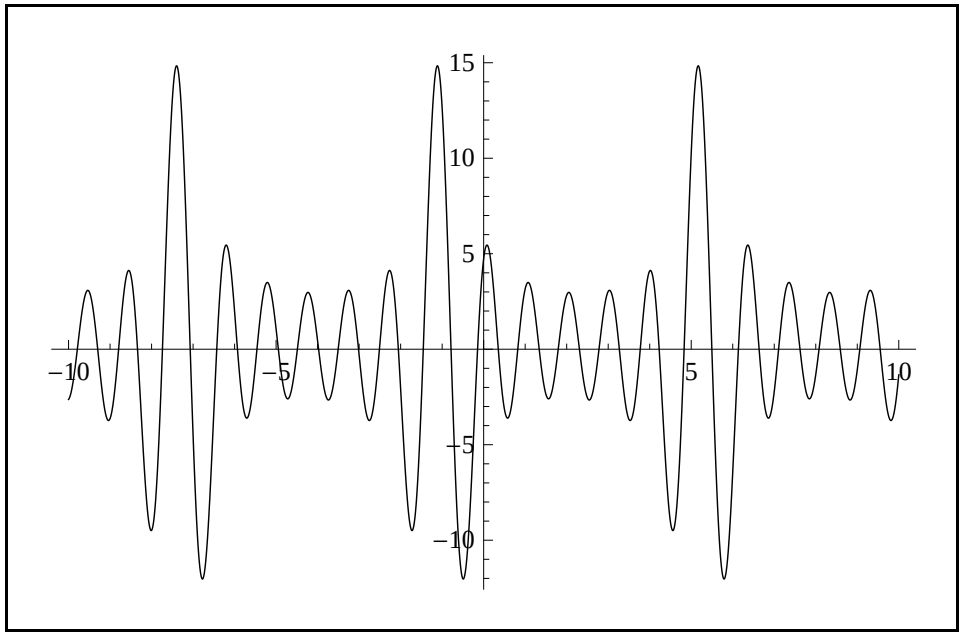


Abbildung 7.2.: $g(x) := -\sum_{k=1}^{5} k \cdot \sin((k+1) \cdot x + k),\ x \in [-10, 10]$

Graphen der Funktionen

$$f(x) = (x + \sin(x)) \cdot e^{-x^2},$$

$$g(x) = -\sum_{k=1}^{5} k \cdot \sin((k+1) \cdot x + k),$$

für die im Intervall $[-10, 10]$ jeweils der oder die minimalen Funktionswerte f^* einzuschließen sind. Die Einschließung von f^* und der lokalen Minimumstellen x^* wird realisiert durch das nachfolgende Programm `MPR-16.cpp`:

```
//--------------------------------------------------------------------------
// Program MPFR-16.cpp: Global optimization (one-dimensional)
// This program uses module 'MPgop1' to compute the global optimizers of the
// functions
//
//     f(x) = (x + sin(x)) * exp(-sqr(x)).
//
// and
//                  5
//     g(x) = - sum ( k * sin((k+1)*x + k) )
//                k=1
//
// A starting interval and a tolerance must be entered.
//--------------------------------------------------------------------------

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

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

MPDerivType f ( const MPDerivType &x )
  { return (x + sin(x)) * exp(-sqr(x)); }

MPDerivType g ( const MPDerivType &x )
{
  MPDerivType  s;
  int          k;

  s = DerivConst(0.0);
  for (k = 1; k <= 5; k++)
    s = s + k * sin ( (k+1) * x + k );
  return -s;
}

//--------------------------------------------------------------------------
// Function for printing and reading information to call the function
// 'AllGOp1'. It must be called with the function parameter 'f' and a
// string 'Name' containing a textual description of that function.
//--------------------------------------------------------------------------
void compute ( MPddf_FctPtr f, char *Name )
{
  MpfiClass  SearchInterval, Minimum;
  MpfrClass  Tolerance;
  MPivector    Opti;
  intvector  Unique;
  int        NumberOfOptis, i, Error;
  PrecisionType prec;

  cout << "Computing all global minimizers of the function  " << Name << endl;
  cout << "Search interval      : "; cin >> SearchInterval;
```

```
54   cout << "Tolerance (relative) : "; cin >> Tolerance;
55   cout << endl;
56
57   if (Tolerance>1e-8) Tolerance = 1e-8;
58   // With the entered 'Tolerance' the corresponding output format is calculated:
59   prec = -expo(Tolerance) + 1;
60   if (prec < 53) prec = 53;
61   cout.precision(prec/3.321928095); // output format;
62
63   AllGOp1(f,SearchInterval,Tolerance,Opti,Unique,NumberOfOptis,Minimum,Error);
64
65   for (i = 1; i <= NumberOfOptis; i++) {
66     cout << Opti[i] << endl;
67     if (Unique[i])
68       cout << "encloses a locally unique candidate for a global minimizer!";
69     else
70       cout << "may contain a local or global minimizer!";
71     cout << endl;
72   }
73
74   if (NumberOfOptis != 0)
75     cout << endl << Minimum << endl
76          << "encloses the global minimum value!" << endl;
77   cout << endl << NumberOfOptis << " interval enclosure(s)" << endl;
78
79   if (Error)
80     cout << endl << AllGOp1ErrMsg(Error) << endl;
81   else if ( (NumberOfOptis == 1) && Unique[1] )
82     cout << endl << "We have validated that there is "
83                  "a unique global optimizer!" << endl;
84 }
85
86 int main(void)
87 {
88   compute(f,(char*)"(x + SIN(x))*EXP(-x^2)");
89   cout << endl << endl;
90   compute(g,(char*)"-SUM(k*SIN((k+1)*x+k),k,1,5)");
91   return 0;
92 }
```

Mit dem Eingabeintervall `[-10,10]` und mit `Tolerance (relative) : 1e-34` liefert das obige Programm für die Funktion $f(x)$ die Ausgabe:

```
Search interval      : [-10,10]
Tolerance (relative) : 1e-34

[-6.795786600198815397288381602301209e-1,-6.795786600198815397288381602301207e-1]
encloses a locally unique candidate for a global minimizer!

[-8.242393984760766542477600525163374e-1,-8.242393984760766542477600525163372e-1]
encloses the global minimum value!

1 interval enclosure(s)

We have validated that there is a unique global optimizer!
```

Mit dem Eingabeintervall `[-10,10]` und mit `Tolerance (relative) : 1e-34` liefert das obige Programm für die Funktion $g(x)$ die Ausgabe:

```
Search interval      : [-10,10]
Tolerance (relative) : 1e-34
```

```
[-6.774576143438901030986146658809774,-6.774576143438901030986146658809773]
encloses a locally unique candidate for a global minimizer!
[-4.913908362593145540608598922507676e-1,-4.913908362593145540608598922507675e-1]
encloses a locally unique candidate for a global minimizer!
[5.791794470920271922864426874308238,5.791794470920271922864426874308239]
encloses a locally unique candidate for a global minimizer!

[-1.203124944216713894806863173737987e1,-1.203124944216713894806863173737986e1]
encloses the global minimum value!

3 interval enclosure(s)
```

Anmerkungen:

1. Die Funktionen $f(x), g(x)$ werden am Anfang des Programms als Funktionen vom Typ `MPDerivType` definiert, wobei alle Funktionen aus der Tabelle auf Seite 128 zur Verfügung stehen. Dabei sollte z.B. $1/\sqrt{x}$ durch `sqrt_r(x)` und nicht mit dem Divisionsoperator "/" durch `1/sqrt(x)` definiert werden.

2. Durch Eingabe von `Tolerance (relative) : 1e-34` wird die relative Genauigkeit der zu berechnenden Einschließungen auf 10^{-34} festgelegt. Damit wird die entsprechende Präzision für die internen Berechnungen festgelegt und zusätzlich das Ausgabeformat, das etwa 34 Dezimalstellen zur Verfügung stellt.

3. Die Funktion $f(x)$ besitzt nur eine lokale Minimumstelle bei $x^* = -0.679578\ldots$ mit dem Minimum $f^* = -0.82423\ldots$, wobei durch das Programm nachgewiesen wird, dass in der berechneten Einschließung von x^* auch keine weitere lokale Minimumstelle liegen kann.

4. Shubert's Funktion $g(x)$, [64], besitzt im Intervall $[-10, +10]$ drei lokale Minimumstellen x^*, an denen das Minimum f^* von $g(x)$ über $[-10, 10]$ angenommen wird, vgl. die Abb. 7.2 auf Seite 131. Das Programm liefert Einschließungen der drei x^* und zeigt, dass jede Einschließung nur eine lokale Minimumstelle enthält. Mit $[-1.203...987e1, -1.203...986e1]$ wird das globale Minimum f^* garantiert und nahezu optimal eingeschlossen.

5. Die Funktion $r(x) := 2x^2 - (3 \cdot e^{-(a(x-b))^2})/c$ nimmt für $a = 1\,000\,000\,000$, $b = 0.0675$ und $c = 100$ bei $x^* = 0.067499\ldots$ ein sehr schmales Minimum an, das bei üblichen Algorithmen ohne Intervallarithmetik leicht übersehen wir. Zeigen Sie, dass unser Programm mit [0,10] und `Tolerance (relative) : 1e-40000` das Minimum $f^* = -2.08874999999999994\ldots \cdot 10^{-2}$ problemlos und mit nahezu optimaler Genauigkeit einschließt.

6. Bei der Auswertung der zu untersuchenden Funktion $f(x)$ mit der Differentiationsarithmetik kann es zu Fehlermeldungen kommen, wenn ein Intervallargument nicht mehr im Definitionsbereich einer verwendeten Elementarfunktion liegt oder wenn durch ein Intervall, das die Null enthält, zu dividieren ist. In diesen Fällen sollte man das Eingangsintervall in mehrere geeignete Teilintervalle unterteilen, um diese Fehlermeldungen zu vermeiden.

7.4.1. Optimale Einschließung des Wertebereichs analytischer Funktionen

Um den Wertebereich einer analytischen Funktion $f(z) = u(x,y) + i \cdot v(x,y),\ z = x + i \cdot y \in \mathbb{C}$, über einem Rechteck Z aus dem Definitionsbereich von $f(z)$ *optimal* einzuschließen, genügt es, die Extrema der harmonischen Funktionen $u(x,y),\ v(x,y)$ jeweils auf dem Rand von Z zu bestimmen. Bei den vergleichsweise einfachen Funktionen $f(z) = 1/\sqrt{z}$ und $f(z) = 1/(1+z^2)$ haben wir gesehen, dass die Berechnung der gesuchten Extrema auf dem Rand von Z jeweils zu aufwendigen Algorithmen führt, die zwar auch bei großen Präzisionen schnell ausführbar sind aber bei komplizierteren Funktionen kaum noch realisierbar sind, vergleichen Sie dazu z.B. die Abschnitte B.6, B.7.

Um eine optimale Einschließung z.B. für $f(z) = e^{z^2} = u(x,y) + i \cdot v(x,y)$ zu realisieren, kann man die gesuchten Extrema von $u(x,y) = e^{x^2-y^2} \cdot \cos(2xy)$ und $v(x,y) = e^{x^2-y^2} \cdot \sin(2xy)$ durch globale Optimierung berechnen, indem man die jeweiligen Extrema auf den vier Rändern von Z für $u(x,y)$ und $v(x,y)$ einzeln bestimmt. Der Vorteil dieses Verfahrens ist die vergleichsweise einfache Realisierung des Algorithmus, der Nachteil ist aber die verhältnismäßig große Laufzeit, die z.B. bei einer Präzision von `prec` = 4000 schon in der Größenordnung mehrerer Sekunden liegt. In der folgenden Klasse `f_expx2` werden u.a. die Realteil- und die Imaginärteilfunktionen u, v von $f(z) = e^{z^2}$ definiert, und mit der Funktion `MpfciClass expx2(const MpfciClass& z)` wird die optimale Einschließung des Wertebereichs von $f(z) = e^{z^2}$ für $z \in Z$ berechnet.

```
class f_expx2
{
  typedef MPDerivType T;        // T wird hier nur zur Abkuerzung eingefuehrt
  enum minORmax {mini, maxi};   // im Namensraum dieser Klasse f_expx2

  template <typename T1, typename T2>
  static T u(const T1& x, const T2& y)
  { // Real part u(x,y) of f(Z)
    T res = exp((x-y)*(x+y))*cos(2*x*y);
    if (mM==maxi) res= -res;  // Es soll -Maximum berechnet werden
    return res;
  }
  static T u_x(const T& x)
  {  // Funktion in x bei festgehaltenem y
     MpfrClass y(p);
     return u(x,y);
  }
  static T u_y(const T& y)
  {  // Funktion in y bei festgehaltenem x
     MpfrClass x(p);
     return u(x,y);
  }
  template <typename T1, typename T2>
  static T v(const T1& x, const T2& y)
  { // Imaginary part v(x,y) of f(z)
    T res = exp((x-y)*(x+y))*sin(2*x*y);
    if (mM==maxi) res = -res;  // Es soll -Maximum berechnet werden
    return res;
  }
  static T v_x(const T& x)
  {  // Funktion in x bei festgehaltenem y
     MpfrClass y(p);
     return v(x,y); }
```

```
 static T v_y(const T& y)
 {  // Funktion in y bei festgehaltenem x
    MpfrClass x(p);
    return v(x,y);
 }

 static MpfiClass OPT(T (*f)(const T&), const MpfiClass& S,
                                        MpfrClass fix, minORmax m)
 {
   PrecisionType prec;
   MpfrClass Tolerance;
   MpfiClass minimum;
   MPivector Opti;
   intvector Unique;
   int       NumberOfOptis, Error;
   mM = m;
   p = fix;

   // Berechnung der 'Tolerance' aus der CurrentPrecision:
   prec = MPFI::MpfciClass::GetCurrPrecision();
   Tolerance = 0.5;
   Tolerance = comp(Tolerance, -prec+1);

   AllGOp1(f, S, Tolerance, Opti, Unique, NumberOfOptis, minimum, Error);
   if (Error)
   {
     cout << endl << AllGOp1ErrMsg(Error) << endl;
     exit(1);
   }
   return minimum; // von f bzw -f ueber Intervall S (= eine Rechteckseite)
 }

public:
 static MpfciClass EinschliessungUeberRechteck(MpfiClass X, MpfiClass Y)
 {
   // Realteil u(x,y):
   MpfiClass u = OPT(u_x, X, Inf(Y), mini); // Einschliessung Minimum
   u|= -OPT(u_x, X, Inf(Y), maxi);         // Einschliessung Maximum
   u|= -OPT(u_x, X, Sup(Y), maxi);
   u|=  OPT(u_x, X, Sup(Y), mini);

   u|=  OPT(u_y, Y, Inf(X), mini);
   u|= -OPT(u_y, Y, Inf(X), maxi);
   u|= -OPT(u_y, Y, Sup(X), maxi);
   u|=  OPT(u_y, Y, Sup(X), mini);
   // Imaginaerteil v(x,y):
   MpfiClass v= OPT(v_x, X, Inf(Y), mini);
   v|= -OPT(v_x, X, Inf(Y), maxi);
   v|= -OPT(v_x, X, Sup(Y), maxi);
   v|=  OPT(v_x, X, Sup(Y), mini);
   v|=  OPT(v_y, Y, Inf(X), mini);
```

```
    v|= -OPT(v_y, Y, Inf(X), maxi);
    v|= -OPT(v_y, Y, Sup(X), maxi);
    v|=  OPT(v_y, Y, Sup(X), mini);
    return MpfciClass(u,v);
  }

  MpfciClass operator()(const MpfciClass& Z)
  {
    return EinschliessungUeberRechteck(Re(Z), Im(Z));
  }

 private:
  static MpfrClass p;  // Wird mit festgehaltenem x- bzw. y-Wert belegt
  static minORmax mM;  // AllGOp1 soll Minimum bzw. -Maximum berechnen
                       // -Maximum ist gerade Minimum von -f
}; // class f_expx2

// Definition der static Variablen der Klasse f_expx2
MpfrClass f_expx2::p(real(12), RoundNearest, 53);
f_expx2::minORmax f_expx2::mM (f_expx2::mini);

MpfciClass expx2(const MpfciClass& z)
// Intervallhuelle von f(z) ueber komplexem Intervall z
{
  int kx, ky, k;
  f_expx2 f; // Funktionsobjekt f der Klasse f_expx2 erzeugen
  PrecisionType prec = z.GetPrecision(),
           prec_old = MPFR::MpfrClass::GetCurrPrecision();
  if (prec<prec_old) prec = prec_old;
  MPFI::MpfiClass::SetCurrPrecision(prec);

  MpfciClass res;
  MpfiClass x(Re(z)), y(Im(z));   // Realteil-/Imaginaerteilintervall
  kx = expo(Inf(x));
  k  = expo(Sup(x));
  if (k > kx) kx = k;
  ky = expo(Inf(y));
  k  = expo(Sup(y));
  if (k > ky) ky = k;
  if (kx+ky > 10000)
  {
      std::cerr << "MpfciClass expx2(const MpfciClass& z);
                    Values of the interval bounds too great!"
                << std::endl;
      exit(1);
  }
  res = f( MpfciClass(x,y) );
  res.RoundPrecision(prec_old);
  MPFI::MpfiClass::SetCurrPrecision(prec_old);
  return res;
}
```

Um eine neue Funktion $f(z)$ mit dem Namen `abcd` durch *Globale Optimierung* zu realisieren, sind folgende Schritte durchzuführen:

1. Kopieren des obigen kompletten Codes aus der Datei `mpfciclass.cpp` in ein neues Arbeitsverzeichnis
2. Ersetzen aller Worte `expx2` durch `abcd`
3. Definitionen von $u(x,y)$ und $v(x,y)$ in den beiden Zeilen der Klasse `f_abcd` realisieren, dazu vorher eventuell die *Mathematica*-Funktion `ComplexExpand[...]` benutzen.
4. In der Funktion `MpfciClass abcd(const MpfciClass& z)` die Zeilen `kx=expo(Inf(x));` bis `exit(1); }` entfernen, die zur Vermeidung eines Überlaufs bei der Funktion e^{z^2} notwendig waren, FERTIG!

Mit der obigen Funktion `MpfciClass expx2(const MpfciClass& Z)` wird für das komplexe Intervall `Z` $= [0.125, 0.35] + i \cdot [1, 3.25]$ mit $f(z) := e^{+z^2}$, $z \in \mathbb{C}$, eine optimale und garantierte Einschließung des Wertebereichs $W_{f,Z} := \{f(z) \,|\, z \in Z\}$ berechnet. Zum Vergleich wird bei gleicher Präzision `prec` $= 74$ der gleiche Wertebereich $W_{f,Z}$ durch die Intervallauswertung des komplexwertigen Ausdrucks `exp(sqr(Z))` eingeschlossen. Man erhält die Ergebnisse:

$$W_{f,Z} \subseteq ([-1.4918459347e-6, 3.6205612011e-1], [1.9076213940e-5, 1.8774573850e-1]),$$
$$W_{f,Z} \subset ([-2.1216070916e-2, 3.7943155613e-1], [6.5006537073e-6, 3.9160562669e-1]).$$

Die Einschließung von $W_{f,Z}$ durch den Intervallausdruck `exp(sqr(Z))` ist deutlich gröber als die optimale Einschließung mit Hilfe der *Globalen Optimierung*.

Wir zeigen jetzt, dass bei viel schmaleren Intervallen, z.B. `Z`$^* = [0.125, 0.126] + i \cdot [1, 1.01]$, die Einschließung von W_{f,Z^*} durch die Intervallauswertung des Ausdrucks `exp(sqr(Z`*`))` i.a. sehr viel besser gelingt, man erhält die Ergebnisse

$$W_{f,Z^*} \subseteq ([3.5452724316e-1, 3.6205612011e-1], [9.1495293392e-2, 9.3195421280e-2]),$$
$$W_{f,Z^*} \subset ([3.5443826798e-1, 3.6214700764e-1], [9.0608448859e-2, 9.4107266993e-2]).$$

Bei hinreichend schmalen Intervallen `Z`* sollte man also insbesondere bei hohen Präzisionen die Auswertung von `exp(sqr(Z`*`))` aus Laufzeitgründen dem Verfahren der *Globalen Optimierung* vorziehen, da dann bei naiver Intervallauswertung die Intervallüberschätzungen praktisch keine Rolle mehr spielen!

7.5. Reelle, eindimensionale Taylor Arithmetik

Die eindimensionale Taylorarithmetik liefert Einschließungen aller Taylorkoeffizienten und Ableitungen einer beliebig oft differenzierbaren Funktion $f : D_f \to \mathbb{R}$ einer Variablen an der Stelle $x_0 \in D_f \subseteq \mathbb{R}$. Dabei kann x_0 durch ein Intervall eingeschlossen werden, wenn x_0 im verwendeten Zahlenraster nicht darstellbar ist. Die in **C-XSC** implementierte Taylorarithmetik kombiniert die symbolische Differentiation mit der Intervallrechnung und liefert dadurch sehr effektiv auch für Taylorkoeffizienten sehr hoher Ordnungen enge Einschließungen, wenn die interne Präzision hinreichend groß gewählt wird. $f(x)$ besitzt für den Entwicklungspunkt x_0 die Taylorreihe

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!} \cdot (x - x_0)^k, \qquad (f)_k := \frac{f^{(k)}(x_0)}{k!},$$

wobei mit $(f)_k$ der Taylorkoeffizient der Ordnung k bezeichnet wird. Die Taylorarithmetik wurde entwickelt in den 60ziger Jahren von Moore, [50], Rall, [57], Griewank, [25], Corliss, [19] und vielen anderen. Eine Zusammenfassung der wichtigsten Rekursionsformeln zur Berechnung der $(f)_k$ und weitere Anwendungen findet man in [11]. In diesem Abschnitt werden die theoretischen Grundlagen nicht weiter behandelt. Beschrieben werden nur die zur Verfügung stehenden Werkzeuge, mit denen in der Programmierumgebung **C-XSC** die Taylorkoeffizienten und Ableitungen in fast beliebiger Präzision berechnet werden können.

Ein Objekt der Klasse `MPitaylor` besitzt einen Vektor `tayl` vom Typ `MPivector` mit den Indices von 0 bis p, wobei $0 \le p \le 500$ die gewählte Ordnung der zu berechnenden Taylorkoeffizienten ist. Die Taylorkoeffizienten $(f)_k$ werden dabei durch die Komponenten vom Typ `MpfiClass` dieses Vektors eingeschlossen. Der maximale Wert $p = 500$ kann bei Bedarf auch größer gewählt werden. Mit $p = 0$ erhält man lediglich eine Einschließung für $f(z_0) = (f)_0$. Die Präzision kann für alle Intervall-Taylorkoeffizienten nach Bedarf einheitlich festgelegt werden. Der Vektor `tayl` einer unabhängigen Variablen `x` vom Typ `MPitaylor` hat die Form

$$\texttt{tayl} = (\texttt{x}, [1], [0], [0], [0], [0], [0]), \quad \text{hier mit } p = 6,$$

und der entsprechende Vektor einer Konstanten `c` vom Typ `MpfiClass` ist gegeben durch

$$\texttt{tayl} = (\texttt{c}, [0], [0], [0], [0], [0]), \quad \text{hier mit } p = 5.$$

Werden die Vektoren für Summe, Differenz, Produkt und Quotient dieser Objekte berechnet, so müssen die Ordnungen p aller dieser Objekte übereinstimmen. Für alle Elementarfunktionen der nachfolgenden Tabelle gibt es entsprechende Formeln, [11], mit denen alle Taylorkoeffizienten $(f)_k,\ k = 0, 1, \ldots, p$, z.B für $f(x) = \sin(x)$, berechnet werden können. Mithilfe dieser Funktionen und den vier Grundoperationen lassen sich dann beliebige Ausdrücke $T(x)$ definieren, für die an Entwicklungspunkten x_0 alle Taylorkoeffizienten oder Ableitungen bis zur Ordnung p berechnet werden können, falls $T(x)$ an der Stelle x_0 mindestens p-mal differenzierbar ist. Wählt man für x_0 ein einschließendes Intervall $z \ni x_0$, z.B. weil x_0 im Zahlenraster nicht darstellbar ist, so ist die berechnete Einschließung für $(T)_k$ auch die Einschließung aller Taylorkoeffizienten der Ordnung k für jedes andere $\hat{x_0} \in z$, mit $\hat{x_0} \neq x_0$. Um möglichst gute Einschließungen der Taylorkoeffizienten zu berechnen, sollte z daher möglichst eng gewählt werden. Bei hoher Ordnung p muss mit einer hinreichend großen Präzision `prec` gerechnet werden, um die bekannten Intervallüberschätzungen bei der internen Intervallauswertung klein zu halten.

7.5.1. Elementatfunktionen

Tabelle 7.2.: Funktionen vom Typ `MPitaylor` zur Taylor-Arithmetik

Funktion	Aufruf	Funktion	Aufruf
x^2	`sqr(x)`	$\cos(x)$	`cos(x)`
$\sqrt{x}$	`sqrt(x)`	$\tan(x)$	`tan(x)`
$\sqrt[n]{x},\ n = 2, 3, \ldots$	`sqrt(x,n)`	$\cot(x)$	`cot(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`
x^y	`pow(x, y)`	$\sinh(x)$	`sinh(x)`
$x^n,\ n \in \mathbb{Z}$	`power(x,n)`	$\cosh(x)$	`cosh(x)`
e^x	`exp(x)`	$\tanh(x)$	`tanh(x)`
2^x	`exp2(x)`	$\coth(x)$	`coth(x)`
10^x	`exp10(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
$e^x - 1$	`expm1(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
$e^{x^2} - 1$	`expx2m1(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
$e^{-x^2} - 1$	`expmx2m1(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$\ln(x)$	`ln(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
$\log_2(x)$	`log2(x)`	$1/\sin(x)$	`csc(x)`
$\log_{10}(x)$	`log10(x)`	$1/\cos(x)$	`sec(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(\sin(x))$	`ln_sin(x)`	$1/\cosh(x)$	`sech(x)`
$\ln(\cos(x))$	`ln_cos(x)`	$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$	`erf(x)`
$\sin(x)$	`sin(x)`	$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,dt$	`erfc(x)`

7.5.2. Konstruktoren / Destruktoren

7.5.2.1. Konstruktoren

```
MPitaylor ();
```

Der Default-Konstruktor legt ein neues `MPitaylor`-Objekt in der Current-Precision an. Aufruf: `MPitaylor f;`
Es wird kein Vektor `tayl` und damit auch keine Ordnung `p` dieses Vektors initialisiert.

```
MPitaylor(const MPitaylor& g, PrecisionType prec);
```

Mit dem Aufruf: `MPitaylor f(g, prec);` legt der Copy-Konstruktor mithilfe von `g` ein neues `MPitaylor`-Objekt `f` an, wobei die Komponenten des Vektors von `f` die Komponenten des Vektors von `g` in der neuen Präzision prec einschließen. Wird prec nicht gesetzt, so werden die Komponenten des Vektors von `g` in der voreingestellten Current-Precision eingeschlossen.

Alle folgenden Konstruktoren erzeugen ein `MPitaylor`-Objekt `f`. Der Vektor `tayl` enthält `p` + 1 Komponenten vom Typ `MpfiClass` mit der Präzision prec.
Aufruf z.B.: `MPitaylor f(5, MpfiClass(1)/10, 400);` dies erzeugt den Vektor `([9.99999e-2,1.00001e-1],[1.00000,1.00000],[0,0],[0,0][0,0],[0,0])`, wobei hier die Dezimalstellen aus Platzgründen verkürzt angegeben sind. Die 1. Komponente ist eine Einschließung des nicht-darstellbaren Entwicklungspunktes $x = 0.1$ mit ca. $400/3.321928095 \approx 120$ Dezimalstellen.

Wird beim Konstruktoraufruf prec nicht angegeben, so erhalten alle Komponenten von `tayl` die voreingestellte Current-Precision. Falls diese mit `SetCurrPrecision` nicht gesetzt wurde, erhalten die Komponenten die Präzision prec = 53.

```
MPitaylor (int p, const MpfiClass& x, PrecisionType prec);
MPitaylor (int p, const interval&  x, PrecisionType prec);
MPitaylor (int p, const MpfrClass& x, PrecisionType prec);
MPitaylor (int p, const real&      x, PrecisionType prec);
MPitaylor (int p, const double&    x, PrecisionType prec);
MPitaylor (int p, int              x, PrecisionType prec);
```

7.5.2.2. Destruktor

```
~MPitaylor(){;};
```

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

7.5.3. Zuweisungs-Operatoren

Unabhängig von der Präzision des rechten Operanden op erhalten bei allen folgenden Zuweisungsoperatoren alle Vektorkomponenten des linken Operanden stets die Current-Precision und schließen die op-Werte optimal ein.

```
MPitaylor operator = (const MPitaylor& op);
MPitaylor operator = (const MpfiClass& op);
MPitaylor operator = (const interval&  op);
MPitaylor operator = (const MpfrClass& op);
MPitaylor operator = (const real&      op);
MPitaylor operator = (const double&    op);
MPitaylor operator = (int              op);
```

Nur beim ersten Operator wird die Ordnung des linken Operanden gleich der Ordnung von op, sonst bleibt die Ordnung des linken Operanden erhalten.

7.5.4. Arithmetische Operatoren

Für alle arithmetischen Operationen mit Operanden vom Typ `MPitaylor` gilt:

> Die exakten Ergebnisse für die Komponenten des Vektors `tayl` werden unabhängig von der Präzision der `tayl`-Komponenten der beiden rechten Operanden in der voreingestellten Current-Präzision optimal eingeschlossen.
> Sind die rechten Operanden beide vom Typ `MPitaylor`, so müssen deren Ordnungen übereinstimmen, und der Ergebnisvektor `tayl` übernimmt diese Ordnung. Sind die Ordnungen der beiden rechten Operanden verschieden, so erfolgt eine entsprechende Fehlermeldung.
> Ist einer der beiden rechten Operanden nicht vom Typ `MPitaylor`, so sind für diesen Operanden nur die folgenden Typen zugelassen:
> `MpfiClass, interval, MpfrClass, real, double, int.`
> Alle Operanden dieser Typen werden dabei stets als Konstanten interpretiert!

Für die Addition existieren beispielsweise die Operatoren:

```
MPitaylor operator + (const MPitaylor& op1, const MPitaylor& op2);
MPitaylor operator + (const MPitaylor& op1, int              op2);
MPitaylor operator + (double&           op1, const MPitaylor& op2);
```

7.5.5. Unabhängige Variablen und Konstanten

Zur Definition von unabhängigen Variablen und Konstanten siehe Seite 139. Unabhängige Variablen werden generiert durch die Funktion

```
MPitaylor var_MPitaylor(int ord, const MpfiClass& x, PrecisionType prec);
```

Für den 2. Parameter stehen folgende Datentypen zur Verfügung:
`MpfiClass, MpfrClass, interval, real, double, int.`

Konstanten werden generiert durch die Funktion

```
MPitaylor const_MPitaylor(int ord, const MpfiClass& x, PrecisionType prec);
```

Für den 2. Parameter stehen folgende Datentypen zur Verfügung:
`MpfiClass, MpfrClass, interval, real, double, int.`

Für beide obigen Funktionen gilt:
Die Vektorkomponenten des Rückgabewertes erhalten die Präzision `prec`. Wird dieser 3. Parameter nicht gesetzt, so erhalten die Vektorkomponenten die Current-Präzision. Die Ordnung des Rückgabevektors `tayl` wird durch den 1. Parameter `ord` festgelegt.

7.5.6. Precision Handling

```
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 `class MPitaylor` 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 Objekte **keinerlei** Einfluss.

```
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 der aktuellen Komponenten von `tayl` auf prec. Die Komponentenwerte bleiben dabei nicht erhalten.

```
void RoundPrecision (PrecisionType prec);
```
Diese Memberfunktion schließt die Komponenten des aktuellen Vektors `tayl` mit der neuen Präzision prec ein. Sollte die Präzision der ursprünglichen Komponenten 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 ursprünglichen Werte der Taylor-Komponenten erhalten bleiben.

7.5.7. Zugriff auf die Vektorkomponenten

```
MPivector get_all_coef(const MPitaylor& x);
```
Zurückgegeben wird ein Vektor vom Typ `MPivector` der Dimension $\mathtt{p}+1$, welcher alle $\mathtt{p}+1$ Taylor-Koeffizienten des Objekts `x` enthält.

```
MpfiClass get_j_coef(const MPitaylor& x, int j);
```
Zurückgegeben wird ein Intervall vom Typ `MpfiClass`, welches den j-ten Taylorkoeffizienten einschließt. $j \leq \mathtt{p}$ muss erfüllt sein.

```
MPivector get_all_derivative(const MPitaylor& x);
```
Zurückgegeben wird ein Vektor vom Typ `MPivector` der Dimension $\mathtt{p}+1$, welcher alle $\mathtt{p}+1$ Ableitungen bezüglich des Objekts `x` enthält. Beachten Sie, dass `x` in seinem Vektor `tayl` nur die Taylorkoeffizienten enthält. Die Ableitungen müssen daher aus diesen Taylor-Koeffizienten neu berechnet werden.

```
MpfiClass get_j_derivative(const MPitaylor& x, int j);
```
Zurückgegeben wird ein Intervall vom Typ `MpfiClass`, welches die j-te Ableitung einschließt. $j \leq \mathtt{p}$ muss erfüllt sein.

```
int get_order(const MPitaylor& x);
```
Zurückgegeben wird die maximale Ordnung `p` der zu berechnenden Taylorkoeffizienten. Die Dimension des Vektors `tayl` ist damit $\mathtt{p}+1$.

7.5.8. Ausgabe

```
void print_MPitaylor(const MPitaylor& x);
```
Gibt alle $\mathtt{p}+1$ Taylorkoeffizienten des Objekts `x` auf dem Bildschirm aus.

```
void print_MPderivatives(const MPitaylor& x);
```
Gibt alle $\mathtt{p}+1$ Ableitungen des Objekts `x` auf dem Bildschirm aus.

```
static void MpfrClass::SetBase (int b);
```
Setzt die Basis $2 \leq b \leq 36$ NUR für das Ausgabeformat. Gerechnet wird stets im Binärformat!

```
static const int MpfrClass::GetBase ();
```
Liefert die Basis $2 \leq b \leq 36$ NUR für das Ausgabeformat. Gerechnet wird stets im Binärformat!

7.5.9. Programme

Die Anwendung der Taylorarithmetik zeigt das nachfolgende Programm `MPFR-17.cpp`, in dem für ein Polynom 4. Grades alle Taylorkoeffizienten und Ableitungen an einer gewählten Stelle x_0 berechnet und auf dem Bildschirm ausgegeben werden.

```
//================================================================
// MPFR-17.cpp;  Example: Taylor Arithmetic;
//----------------------------------------------------------------
//  Inclusions of the Taylor coefficients and derivatives up to order p=5 for
//                P(x) = 2x^4 + x^3 + 4x^2 - 3x + 2
//  at points of expansion x0 included by the interval z.
//  A starting interval including the point of expansion x0 must be entered.
//================================================================

#include "mpficlass.hpp"
#include <iostream>       // Input | output

using namespace cxsc;
using namespace MPFI;

int main()
{
  PrecisionType prec=100;
  MPitaylor::SetCurrPrecision(prec);
  int p = 5;     // Order of expansion
  MpfiClass z;   // Interval to include the point of expansion
  MPitaylor P;   // Default constructor

  while(1) {
    cout << endl << "Inclusion of point of expansion x0; [x0,x0] = ? ";
    cin >> z;
    MPitaylor x(p,z);   // Constructor
    P = (((2.0*x + 1)   // Polynomial of order 4.
              *x + 4)
              *x - 3.0)
              *x + 2.0;
    cout.precision(P.GetPrecision()/3.321928095);
    print_MPitaylor(P);   // Output of Taylor coefficients.

    MPivector derivative(0,p);   // interval vector with
    for(int i=0; i<=p; i++)
      derivative[i] = get_j_derivative(P,i);   // Derivatives
    cout << "Inclusions of the derivatives up to order 5 " << endl;
    for(int i=0; i<=p; i++)      // Output of the derivatives
      cout << i<<"th derivative:  " << derivative[i] << endl;
  }
} // main
```

Das Programm liefert die Ausgabe:

```
Inclusion of point of expansion x0; [x0,x0] = ? [2,2]
Output MPitaylor of order 5
i = 0  component: [5.2000000000000000000000000000000e1,5.2000000000000000000000000000000e1]
i = 1  component: [8.9000000000000000000000000000000e1,8.9000000000000000000000000000000e1]
i = 2  component: [5.8000000000000000000000000000000e1,5.8000000000000000000000000000000e1]
i = 3  component: [1.7000000000000000000000000000000e1,1.7000000000000000000000000000000e1]
i = 4  component: [2.0000000000000000000000000000000,2.0000000000000000000000000000000]
i = 5  component: [0,-0]

Inclusions of the derivatives up to order 5
0th derivative:  [5.2000000000000000000000000000000e1,5.2000000000000000000000000000000e1]
1th derivative:  [8.9000000000000000000000000000000e1,8.9000000000000000000000000000000e1]
2th derivative:  [1.1600000000000000000000000000000e2,1.1600000000000000000000000000000e2]
3th derivative:  [1.0200000000000000000000000000000e2,1.0200000000000000000000000000000e2]
4th derivative:  [4.8000000000000000000000000000000e1,4.8000000000000000000000000000000e1]
5th derivative:  [0,-0]
```

Das nachfolgende Programm `MPFR-18.cpp` liefert Taylorkoeffizienten von Funktionen.

```
//=============================================================================
//-----------------------------------------------------------------------------
// MPFR-18.cpp;  Example: Taylor Arithmetic;
//-----------------------------------------------------------------------------
//   Evaluation of the equivalent functions
//          g(x) = ln(1+e^x);      h(x) = x + ln(1+e^(-x));
//   The selection of the appropriate expression is essential
//   for getting tight enclosures of the Taylor coefficients.
//-----------------------------------------------------------------------------
//=============================================================================

#include "mpficlass.hpp" // Header file of class itaylor
#include <iostream>      // Input | output

using namespace cxsc;
using namespace std;
using namespace MPFI;

MPitaylor f1(const MPitaylor& x)
{
  MPitaylor w;
  MpfiClass z( get_j_coef(x,0) );
  if (Inf(z)>0) w = x + lnp1( exp(-x) );
  else w = lnp1( exp(x) );

  return w;
}

int main()
{
  PrecisionType prec = 100;
  MPitaylor::SetCurrPrecision(prec);
  int p = 4;          // Order of expansion
  MPitaylor f,g,h;   // Default constructor
  MPitaylor x(p,interval(70000));

  g = lnp1( exp(x) );
  h = x + lnp1( exp(-x) );
  f = f1(x);

  cout.precision(x.GetPrecision()/3.321928095);
  cout << "Taylor coefficients of g(x) are included by:" << endl;
  print_MPitaylor(g);  // Output of Taylor coefficients.

  cout << "Taylor coefficients of h(x) are included by:" << endl;
  print_MPitaylor(h);

  cout << "Taylor coefficients of f1(x) are included by:" << endl;
  print_MPitaylor(f);
} // main
```

Das Programm liefert die Ausgabe:

```
Taylor coefficients of g(x) are included by:
Output MPitaylor of order 4
i = 0  Taylor coefficient:
              [6.99999999999999999999999999998e4,7.00000000000000000000000000002e4]
i = 1  Taylor coefficient:
              [9.99999999999999999999999999997e-1,1.00000000000000000000000000001]
i = 2  Taylor coefficient:
              [-1.50518740230346203068684003538e-30,2.00691653640461604091578671384e-30]
i = 3  Taylor coefficient:
              [-2.25778110345519304603026005307e-30,2.00691653640461604091578671384e-30]
i = 4  Taylor coefficient:
              [-2.38321338698048154858749672268e-30,2.44592952874312579986611505749e-30]
```

```
Taylor coefficients of h(x) are included by:
Output MPitaylor of order 4
i = 0  Taylor coefficient:
       [7.0000000000000000000000000000e4,7.0000000000000000000000000002e4]
i = 1  Taylor coefficient:
       [9.9999999999999999999999999999e-1,1.0000000000000000000000000000]
i = 2  Taylor coefficient:
       [1.2168492442353023869353645975 2e-30401,1.2168492442353023869353645975 4e-30401]
i = 3  Taylor coefficient:
       [-4.0561641474510079564512153251 2e-30402,-4.0561641474510079564512153251 0e-30402]
i = 4  Taylor coefficient:
       [1.0140410368627519891128038312 7e-30402,1.0140410368627519891128038312 8e-30402]

Taylor coefficients of f1(x) are included by:
Output MPitaylor of order 4
i = 0  Taylor coefficient:
       [7.0000000000000000000000000000e4,7.0000000000000000000000000002e4]
i = 1  Taylor coefficient:
       [9.9999999999999999999999999999e-1,1.0000000000000000000000000000]
i = 2  Taylor coefficient:
       [1.2168492442353023869353645975 2e-30401,1.2168492442353023869353645975 4e-30401]
i = 3  Taylor coefficient:
       [-4.0561641474510079564512153251 2e-30402,-4.0561641474510079564512153251 0e-30402]
i = 4  Taylor coefficient:
       [1.0140410368627519891128038312 7e-30402,1.0140410368627519891128038312 8e-30402]
```

Anmerkungen:

1. Für die obigen Funktionen gilt für alle $x \in \mathbb{R}$: $\quad g(x) \equiv h(x) \equiv f1(x)$.

2. Mit den obigen Ergebnissen ist für eine nahezu optimale Berechnung der Taylorkoeffizienten höherer Ordnung ($p \geq 2$) an der Stelle $x_0 = 70\,000$ folgende Fallunterscheidung nötig:

$$f1(x) = \begin{cases} h(x) := x + \ln(1 + e^{-x}), & x > 0, \\ g(x) := \ln(1 + e^{x}), & x \leq 0, \end{cases}$$

   dabei sind $\ln(1 + e^{-x})$, $\ln(1 + e^{x})$ mithilfe der vordefinierten Funktionen `lnp1(exp(-x))` bzw. `lnp1(exp(x))` auszuwerten.

3. Zeigen Sie, dass für $x_0 = -70\,000$ die Fallunterscheidung sogar für $p \geq 0$ notwendig ist.

Das Programm-Beispiel zeigt, dass zur optimalen Berechnung der Taylorkoeffizienten einer vorgegebenen Funktion die Wahl eines geeigneten Funktionsterms von entscheidender Bedeutung sein kann. Wenn im Funktionsterm Teilausdrücke auftauchen, die durch Funktionen aus der Tabelle von Seite 140 ersetzt werden können, z.B. $\ln(1 + e^{x})$ durch `lnp1( exp(x) )`, so sollte dies auch unbedingt realisiert werden!
Wenn zu grobe Einschließungen berechnet werden, so ist dies oft ein deutlicher Hinweis dafür, dass der Funktionsterm noch nicht optimal gewählt wurde. Oft genügt es aber auch, einfach die Präzision z.B. zu verdoppeln.

7.6. Komplexe, eindimensionale Taylor Arithmetik

Die komplexe, eindimensionale Taylorarithmetik liefert Einschließungen aller Taylorkoeffizienten und Ableitungen einer analytischen Funktion $f : D_f \to \mathbb{C}$ einer Variablen an der Stelle $z_0 \in D_f \subseteq \mathbb{C}$. Dabei kann $z_0 = x_0 + i \cdot y_0$ durch ein Intervall eingeschlossen werden, wenn z_0 im verwendeten Zahlenraster nicht darstellbar ist. Die in **C-XSC** implementierte Taylorarithmetik kombiniert die symbolische Differentiation mit der Intervallrechnung und liefert dadurch sehr effektiv auch für Taylorkoeffizienten sehr hoher Ordnungen enge Einschließungen, wenn die interne Präzision hinreichend groß gewählt wird. $f(z)$ besitzt für den Entwicklungspunkt z_0 die Taylorreihe

$$f(z) = \sum_{k=0}^{\infty} \frac{f^{(k)}(z_0)}{k!} \cdot (z - z_0)^k, \qquad (f)_k := \frac{f^{(k)}(z_0)}{k!} \in \mathbb{C},$$

wobei mit $(f)_k$ der Taylorkoeffizient der Ordnung k bezeichnet wird. Literaturhinweise findet man im Abschnitt 7.5 auf Seite 139. Da die Rekursionsformeln zur Berechnung der $(f)_k$ für beliebig oft differenzierbare reelle und für die entsprechenden analytischen Funktionen identisch sind, können die entsprechenden reellen Algorithmen direkt für die komplexen Berechnungen übernommen werden. Man muss lediglich darauf achten, dass die den Entwicklungspunkt z_0 einschließenden Intervalle im Definitionsbereich D_f liegen. In diesem Abschnitt werden die theoretischen Grundlagen nicht weiter behandelt. Beschrieben werden nur die zur Verfügung stehenden Werkzeuge, mit denen in der Programmierumgebung **C-XSC** die Taylorkoeffizienten und Ableitungen in fast beliebiger Präzision berechnet werden können.

Ein Objekt der Klasse `MPcitaylor` besitzt einen Vektor `tayl` vom Typ `MPcivector` mit den Indices von 0 bis p, wobei $0 \le p \le 500$ die gewählte Ordnung der zu berechnenden Taylorkoeffizienten ist. Die Taylorkoeffizienten $(f)_k$ werden dabei durch die Komponenten vom Typ `MpfciClass` dieses Vektors eingeschlossen. Der maximale Wert $p = 500$ kann bei Bedarf auch noch größer gewählt werden. Mit $p = 0$ erhält man lediglich eine Einschließung für $f(z_0) = (f)_0$. Die Präzision kann für alle Intervall-Taylorkoeffizienten nach Bedarf einheitlich festgelegt werden. Der Vektor `tayl` einer unabhängigen Variablen[1] `z` vom Typ `MPcitaylor` hat die Form

$$\texttt{tayl} = (\texttt{z}, [1], [0], [0], [0], [0], [0]), \quad \text{hier mit } p = 6,$$

und der entsprechende Vektor einer Konstanten `c` vom Typ `MpfciClass` ist gegeben durch

$$\texttt{tayl} = (\texttt{c}, [0], [0], [0], [0], [0]), \quad \text{hier mit } p = 5.$$

Werden die Vektoren für Summe, Differenz, Produkt und Quotient dieser Objekte berechnet, so müssen die Ordnungen p aller dieser Objekte übereinstimmen. Für alle Elementarfunktionen der nachfolgenden Tabelle gibt es entsprechende Formeln, [11], mit denen alle Taylorkoeffizienten $(f)_k,\ k = 0, 1, \ldots, p$, z.B für $f(z) = \sin(z)$, berechnet werden können. Mithilfe dieser Funktionen und den vier Grundoperationen lassen sich dann beliebige Ausdrücke $T(z)$ definieren, für die an Entwicklungspunkten $z_0 \in \mathbb{C}$ alle Taylorkoeffizienten oder Ableitungen bis zur Ordnung p garantiert eingeschlossen werden können, falls $T(z)$ an der Stelle z_0 analytisch ist. Wählt man für z_0 ein einschließendes Intervall $[z] \ni z_0$, z.B. weil z_0 im Zahlenraster nicht darstellbar ist, so ist die berechnete Einschließung für $(T)_k$ auch die Einschließung aller Taylorkoeffizienten der Ordnung k für jedes andere $\hat{z_0} \in [z]$, mit $\hat{z_0} \neq z_0$. Um möglichst gute Einschließungen der Taylorkoeffizienten zu berechnen, sollte $[z]$ daher möglichst eng gewählt werden. Bei höherer Ordnung p muss oft mit einer hinreichend großen Präzision `prec` gerechnet werden, um die bekannten Intervallüberschätzungen bei der internen Intervallauswertung klein zu halten.

[1] Beachten Sie, dass `z` hier vom Typ `MPcitaylor` ist, während $z \in \mathbb{C}$ eine komplexe Variable bezeichnet.

7.6.1. Elementarfunktionen

Tabelle 7.3.: Funktionen vom Typ `MPcitaylor` zur Taylor-Arithmetik, z vom Typ `MPcitaylor`, w vom Typ `MpfciClass`;

Funktion	Aufruf	Funktion	Aufruf
z^2	`sqr(z)`	$\log_{10}(z)$	`log10(z)`
$\sqrt{z}$	`sqrt(z)`	$\ln(1+z)$	`lnp1(z)`
$\sqrt[n]{z},\ n = 2,3,\ldots$	`sqrt(z,n)`	$\sin(z)$	`sin(z)`
$\sqrt{1+z^2}$	`sqrt1px2(z)`	$\cos(z)$	`cos(z)`
$\sqrt{1-z^2}$	`sqrt1mx2(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z^2-1}$	`sqrtx2m1(z)`	$\cot(z)$	`cot(z)`
$\sqrt{z+1}-1$	`sqrtp1m1(z)`	$\arcsin(z)$	`asin(z)`
z^w	`pow(z,w)`	$\arccos(z)$	`acos(z)`
$z^n,\ n \in \mathbb{Z}$	`power(z,n)`	$\arctan(z)$	`atan(z)`
e^z	`exp(z)`	$\operatorname{arccot}(z)$	`acot(z)`
2^z	`exp2(z)`	$\sinh(z)$	`sinh(z)`
10^z	`exp10(z)`	$\cosh(z)$	`cosh(z)`
e^z-1	`expm1(z)`	$\tanh(z)$	`tanh(z)`
e^{z^2}	`expx2(z)`	$\coth(z)$	`coth(z)`
$e^{z^2}-1$	`expx2m1(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
e^{-z^2}	`expmx2(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$e^{-z^2}-1$	`expmx2m1(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
$\ln(z)$	`ln(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`
$\log_2(z)$	`log2(z)`		

7.6.2. Konstruktoren / Destruktoren

7.6.2.1. Konstruktoren

```
MPcitaylor ();
```

Der Default-Konstruktor legt ein neues `MPcitaylor`-Objekt in der Current-Precision an. Aufruf: `MPcitaylor f;`
Es wird kein Vektor `tayl` und damit auch keine Ordnung `p` dieses Vektors initialisiert.

```
MPcitaylor(const MPcitaylor& g, PrecisionType prec);
```

Mit dem Aufruf: `MPcitaylor f(g, prec);` legt der Copy-Konstruktor mithilfe von `g` ein neues `MPcitaylor`-Objekt `f` an, wobei die Komponenten des Vektors von `f` die Komponenten des Vektors von `g` in der neuen Präzision prec einschließen. Wird prec nicht gesetzt, so werden die Komponenten des Vektors von `g` in der voreingestellten Current-Precision eingeschlossen.

Alle folgenden Konstruktoren erzeugen ein `MPcitaylor`-Objekt `f`. Der zugehörige Vektor `tayl` enthält `p` + 1 Komponenten vom Typ `MpfciClass` mit der Präzision prec.
Der Aufruf: `MPcitaylor f(5, MpfciClass( MpfiClass(1)/10, MpfiClass(-1) ), 400);`
erzeugt den Vektor
`tayl = {([9.999e-2,1.001e-1],[-1,-1]),([1,1],[0,0]),(0,0),(0,0),(0,0),(0,0)}`,
mit 5 + 1 = 6 komplexen Komponenten vom Typ `MpfciClass` , wobei hier die Dezimalstellen aus Platzgründen verkürzt angegeben sind. Die 1. Komponente ist eine Einschließung des nicht-darstellbaren komplexen Entwicklungspunktes $z_0 = 0.1 - i$ mit ca. 400/3.321928095 ≈ 120 Dezimalstellen.

Wird beim Konstruktoraufruf prec nicht angegeben, so erhalten alle Komponenten von `tayl` die voreingestellte Current-Precision. Falls diese mit `SetCurrPrecision` nicht gesetzt wurde, erhalten die Komponenten die Präzision prec = 53.

```
MPcitaylor (int p, const MpfciClass& x, PrecisionType prec);
MPcitaylor (int p, const cinterval&  x, PrecisionType prec);
MPcitaylor (int p, const MpfcClass&  x, PrecisionType prec);
MPcitaylor (int p, const complex&    x, PrecisionType prec);
MPcitaylor (int p, const MpfiClass&  x, PrecisionType prec);
MPcitaylor (int p, const interval&   x, PrecisionType prec);
MPcitaylor (int p, const MpfrClass&  x, PrecisionType prec);
MPcitaylor (int p, const real&       x, PrecisionType prec);
MPcitaylor (int p, const double&     x, PrecisionType prec);
MPcitaylor (int p, int               x, PrecisionType prec);
```

7.6.2.2. Destruktor

```
~MPcitaylor(){;};
```

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

7.6.3. Zuweisungs-Operatoren

Unabhängig von der Präzision des rechten Operanden op erhalten bei allen folgenden Zuweisungsoperatoren alle Vektorkomponenten des linken Operanden stets die Current-Precision und schließen die op-Werte optimal ein.

```
MPcitaylor operator = (const MPcitaylor& op);
MPcitaylor operator = (const MpfciClass& op);
MPcitaylor operator = (const cinterval&  op);
MPcitaylor operator = (const MpfcClass&  op);
MPcitaylor operator = (const complex&    op);
MPcitaylor operator = (const MpfiClass&  op);
MPcitaylor operator = (const interval&   op);
MPcitaylor operator = (const MpfrClass&  op);
MPcitaylor operator = (const real&       op);
MPcitaylor operator = (const double&     op);
MPcitaylor operator = (int               op);
```

Nur beim ersten Operator wird die Ordnung des linken Operanden gleich der Ordnung von op, sonst bleibt die Ordnung des linken Operanden erhalten.

7.6.4. Arithmetische Operatoren

Für alle arithmetischen Operationen mit Operanden vom Typ `MPcitaylor` gilt:

> Die exakten Ergebnisse für die Komponenten des Vektors `tayl` werden unabhängig von der Präzision der `tayl`-Komponenten der beiden rechten Operanden in der voreingestellten Current-Präzision optimal eingeschlossen.
> Sind die rechten Operanden beide vom Typ `MPcitaylor`, so müssen deren Ordnungen übereinstimmen, und der Ergebnisvektor `tayl` übernimmt diese Ordnung. Sind die Ordnungen der beiden rechten Operanden verschieden, so erfolgt eine entsprechende Fehlermeldung.
> Ist einer der beiden rechten Operanden nicht vom Typ `MPcitaylor`, so sind für diesen Operanden nur die folgenden Typen zugelassen:
> `MpfciClass, cinterval, MpfcClass, complex,`
> `MpfiClass, interval, MpfrClass, real, double, int.`
> Alle Operanden dieser Typen werden dabei stets als Konstanten interpretiert!

Für die Addition existieren beispielsweise die Operatoren:

```
MPcitaylor operator + (const MPcitaylor& op1, const MPcitaylor& op2);
MPcitaylor operator + (const MPcitaylor& op1, int               op2);
MPcitaylor operator + (complex&          op1, const MPcitaylor& op2);
```

7.6.5. Unabhängige Variablen und Konstanten

Zur Definition von unabhängigen Variablen und Konstanten siehe Seite 147. Unabhängige Variablen werden generiert durch die Funktion

```
MPcitaylor var_MPcitaylor(int ord,const MpfciClass& x, PrecisionType prec);
```

Für den 2. Parameter stehen folgende Datentypen zur Verfügung:

`MpfciClass, cinterval, MpfcClass, complex,`
`MpfiClass, interval, MpfrClass, real, double, int.`

Konstanten werden generiert durch die Funktion

```
MPcitaylor const_MPcitaylor(int ord,const MpfiClass& x,PrecisionType prec);
```

Für den 2. Parameter stehen folgende Datentypen zur Verfügung:

`MpfciClass, cinterval, MpfcClass, complex,`
`MpfiClass, interval, MpfrClass, real, double, int.`

Für beide obigen Funktionen gilt:
Die Vektorkomponenten des Rückgabewertes erhalten die Präzision `prec`. Wird dieser 3. Parameter nicht gesetzt, so erhalten die Vektorkomponenten die Current-Präzision. Die Ordnung des Rückgabevektors `tayl` wird durch den 1. Parameter `ord` festgelegt.

7.6.6. Precision Handling

```
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 `class MPcitaylor` 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 Objekte **keinerlei** Einfluss.

```
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 der aktuellen Komponenten von `tayl` auf prec. Die Komponentenwerte bleiben dabei nicht erhalten.

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

Diese Memberfunktion schließt die Komponenten des aktuellen Vektors `tayl` mit der neuen Präzision prec ein. Sollte die Präzision der ursprünglichen Komponenten 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 ursprünglichen Werte der Taylor-Komponenten erhalten bleiben.

7.6.7. Zugriff auf die Vektorkomponenten

```
MPcivector get_all_coef(const MPcitaylor& x);
```

Zurückgegeben wird ein Vektor vom Typ `MPcivector` der Dimension `p` + 1, welcher alle `p` + 1 Taylor-Koeffizienten des Objekts `x` enthält.

```
MpfciClass get_j_coef(const MPcitaylor& x, int j);
```

Zurückgegeben wird ein komplexes Intervall vom Typ `MpfciClass` , welches den j-ten Taylorkoeffizienten einschließt. $j \leq$ `p` muss erfüllt sein.

```
MPcivector get_all_derivative(const MPcitaylor& x);
```

Zurückgegeben wird ein Vektor vom Typ `MPcivector` der Dimension `p` + 1, welcher alle `p`+1 Ableitungen bezüglich des Objekts `x` enthält. Beachten Sie, dass `x` in seinem Vektor `tayl` nur die Taylorkoeffizienten enthält. Die Ableitungen müssen daher aus diesen Taylor-Koeffizienten neu berechnet werden.

```
MpfciClass get_j_derivative(const MPcitaylor& x, int j);
```

Zurückgegeben wird ein komplexes Intervall vom Typ `MpfciClass`, welches die j-te Ableitung einschließt. $j \leq$ `p` muss erfüllt sein.

```
int get_order(const MPcitaylor& x);
```

Zurückgegeben wird die maximale Ordnung `p` der zu berechnenden Taylorkoeffizienten. Die Dimension des Vektors `tayl` ist damit `p` + 1.

7.6.8. Ausgabe

```
void print_MPcitaylor(const MPcitaylor& x);
```

Gibt die Einschließungen aller `p` + 1 komplexen Taylorkoeffizienten des Objekts `x` auf dem Bildschirm aus.

```
void print_MPderivatives(const MPcitaylor& x);
```

Gibt die Einschließungen aller `p` + 1 komplexen Ableitungen des Objekts `x` auf dem Bildschirm aus.

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

Setzt die Basis $2 \leq b \leq 36$ NUR für das Ausgabeformat. Gerechnet wird stets im Binärformat!

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

Liefert die Basis $2 \leq b \leq 36$ NUR für das Ausgabeformat. Gerechnet wird stets im Binärformat!

7.6.9. Programme

Die Anwendung der komplexen Taylorarithmetik zeigt das folgende Programm `MPFR-19.cpp`, in dem für ein komplexes Polynom 4. Grades alle Taylorkoeffizienten und Ableitungen an einer gewählten Stelle z_0 eingeschlossen und auf dem Bildschirm ausgegeben werden.

```
//=============================================================================
// MPFR-19.cpp;  Example: Complex Taylor Arithmetic;
//  Enclosures of the Taylor coefficients and derivatives up to order p=5 for
//                P(z) = 2*z^4 + z^3 + (4+i)*z^2 - 3z + 2-i
//  at points of expansion z0=x+i*y included by an interval [z].
//  A starting interval including the point of expansion z0 must be entered.
//=============================================================================

#include "mpfciclass.hpp"
#include <iostream>       // Input | output

using namespace cxsc;
using namespace MPFI;

int main() {
  PrecisionType prec=70;
  MPcitaylor::SetCurrPrecision(prec);
  int p = 5;       // Order of expansion
  MpfciClass z;  // Interval to include the point of expansion
  MPcitaylor P;  // Default constructor

  while(1) {
    cout << endl << "Inclusion of point of expansion x0; ([x,x],[y,y]) = ? ";
    cin >> z;
    MPcitaylor x(p,z);  // Constructor

    P = (((2.0*x + 1)  // Polynomial of order 4.
             *x + complex(4,1))
             *x - 3.0)
             *x + complex(2,-1);
    cout.precision(P.GetPrecision()/3.321928095);
    print_MPcitaylor(P);  // Output of Taylor coefficients.

    MPcivector derivative(0,p);  // interval vector with
                                 // components from index 0 up to the order p.
    for(int i=0; i<=p; i++)
      derivative[i] = get_j_derivative(P,i);  // Derivatives
    cout << "Enclosures of the derivatives up to order 5 " << endl;
    for(int i=0; i<=p; i++)     // Output of the derivatives
      cout << i<<"th derivative:  " << derivative[i] << endl; }
} // main
```

Das Programm liefert die Ausgabe:

```
Inclusion of point of expansion x0; ([x,x],[y,y]) = ? ([2,2],[-3,-3])
Output MPcitaylor of order 5
i = 0  Taylor coefficient: ([-2.96000000000000000000e2,-2.96000000000000000000e2],
                            [1.86000000000000000000e2,1.86000000000000000000e2])
i = 1  Taylor coefficient: ([-3.64000000000000000000e2,-3.64000000000000000000e2],
                            [-1.28000000000000000000e2,-1.28000000000000000000e2])
i = 2  Taylor coefficient: ([-5.00000000000000000000e1,-5.00000000000000000000e1],
                            [-1.52000000000000000000e2,-1.52000000000000000000e2])
i = 3  Taylor coefficient: ([1.70000000000000000000e1,1.70000000000000000000e1],
                            [-2.40000000000000000000e1,-2.40000000000000000000e1])
i = 4  Taylor coefficient: ([2.00000000000000000000,2.00000000000000000000], [0,-0])
i = 5  Taylor coefficient: ([0,-0], [0,-0])

Enclosures of the derivatives up to order 5
0th derivative:  ([-2.96000000000000000000e2,-2.96000000000000000000e2],
                  [1.86000000000000000000e2,1.86000000000000000000e2])
1th derivative:  ([-3.64000000000000000000e2,-3.64000000000000000000e2],
                  [-1.28000000000000000000e2,-1.28000000000000000000e2])
2th derivative:  ([-1.00000000000000000000e2,-1.00000000000000000000e2],
                  [-3.04000000000000000000e2,-3.04000000000000000000e2])
3th derivative:  ([1.02000000000000000000e2,1.02000000000000000000e2],
                  [-1.44000000000000000000e2,-1.44000000000000000000e2])
4th derivative:  ([4.80000000000000000000e1,4.80000000000000000000e1], [0,-0])
5th derivative:  ([0,-0], [0,-0])
```

Mit Hilfe der Identität

$$T_1(z) := \cot(4z) \equiv \frac{\cot^4 z - 6\cot^2 z + 1}{4\cot^3 z - 4\cot\ z} =: T_2(z)$$

zeigt das folgende Programm `MPFR-20`, dass bei ungünstiger Wahl des Funktionsterms $T_2(z)$ und zu kleiner Präzision `prec=200` z.B. die 5. Ableitung an der Stelle $z_0 = 2^{-200} + i \cdot 2^{-100}$ viel zu grob eingeschlossen wird, während mit $T_1(z)$ die gleiche Ableitung bei gleicher Präzision fast optimal eingeschlossen wird.

```
//=============================================================================
//-----------------------------------------------------------------------------
// MPFR-20.cpp;  Example: Complex Taylor Arithmetic;
//-----------------------------------------------------------------------------
// To get best enclosures of the derivatives (Taylor coefficients) the
// choice of the optimal function term is essential:
// cot(4z) = (cot^4(z) - 6*cot^2(z) + 1)/(4*cot^3(z)-4*cot(z));
//-----------------------------------------------------------------------------
//=============================================================================

#include "mpfciclass.hpp"
#include <iostream>       // Input | output

using namespace cxsc;
using namespace MPFI;

int main()
{
  PrecisionType prec=200;
  MPcitaylor::SetCurrPrecision(prec);
  int p = 5;      // Order of expansion
  MpfiClass re(exp2(-200)), im(exp2(-100));
  MpfciClass z(re,im);
  MPcitaylor y;       // Default constructor
  MPcitaylor x(p,z); // x: Variable of type MPcitaylor

  y = cot(4*x);
  cout.precision(y.GetPrecision()/3.321928095);
  cout << "5-th derivative:  " << get_j_derivative(y,5) << endl;

  y = (power(cot(x),4)-6*power(cot(x),2)+1)/(4*power(cot(x),3)-4*cot(x));
  cout << "5-th derivative:  " << get_j_derivative(y,5) << endl;
} // main
```

Das Programm liefert die Ausgabe:

```
5-th derivative:  ([1.2448546706642978875537223591073483453037338696727310699985 0e182,
                    1.2448546706642978875537223591073483453037338696727310699988 7e182],
                   [5.8921030942130553660237414542896784739899556828775166210682 1e152,
                    5.8921030942130553660237414542896784739899556828775166210715 7e152])
5-th derivative:  ([-1.1418171638918676928777003842218244832731196291348163212906 1e775,
                    1.0996097342255268125350146771244859896694073054691119837614 7e775],
                   [-1.1416837936489774349220441445630134392253495813355446456572 7e775,
                    1.1416837936489774349220441445615739324660648720672949519847 1e775])
```

Anmerkungen:

1. Erst mit `prec` = 400 erhält man mit Hilfe von $T_2(z)$ eine Einschließung der 5. Ableitung mit hinreichend vielen übereinstimmenden Dezimalstellen.

2. Ein wesentlicher Vorteil einer Taylorarithmetik mit beliebiger Präzision besteht aber darin, dass man bei ungeschickter Wahl des Funktionsterms immer noch brauchbare Einschließungen erhält, wenn die Präzision nur hinreichend groß gewählt wird.

7.7. Nullstellen komplexer Ausdrücke

In der Dissertation von W. Krämer wird ein Algorithmus zur Einschließung von Nullstellen komplexwertiger Ausdrücke angegeben, [38],[13]. 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

$$f(z) = \arctan\big((z-a)\cdot\ln(z^2 - 5\cdot z + 8 + i)\big)$$

garantiert eingeschlossen werden, wobei zur Realisierung von $z_3 = 4i$ der Parameter $a = 4i$ zu setzten ist. Im Programm muss die 1. Ableitung

$$f'(z) = \frac{\ln(z^2 - 5\cdot z + 8 + i) + \dfrac{(z-a)\cdot(2\cdot z - 5)}{z^2 - 5\cdot z + 8 + i}}{1 + (z-a)^2\cdot(\ln(z^2 - 5\cdot z + 8 + i))^2}$$

für das Newton-Verfahren **nicht** explizit definiert werden, da $f(z)$ und $f'(z)$ mit Hilfe der durch komplexe Taylor-Arithmetik definierten Funktion

```
MPcitaylor F(const MPcitaylor Z)
{
   MpfcClass a(0,4);
   return atan( (Z-a)*ln(sqr(Z) - 5*Z + MpfcClass(8,1)) );
}
```

direkt eingeschlossen werden können.

```
//================================================================================
//--------------------------------------------------------------------------------
// MPFR-11.cpp;  Complex (interval) Newton method using complex interval
// Automatic Taylor Arithmetic (ATA) to compute first derivatives
//--------------------------------------------------------------------------------
//================================================================================

#include "mpfciclass.hpp"

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

// The zeros of the function F() are to be enclosed. Zeros are:
// a (see parameter in function definition), 2+i, and 3-i.

MPcitaylor F(const MPcitaylor& Z) // Definition of F(z) and F'(z) via ATA
{
    MpfcClass a(0,4);    // The parameter a is set to 0 + 4i
    return atan( (Z-a)*ln(sqr(Z) - 5*Z + MpfcClass(8,1)) );
};


int main(void)
{
  PrecisionType prec= 99; // number of bits to be used in arithmetic operations
  MPcitaylor::SetCurrPrecision(prec);
  cout << "Current-precision: about " << Round(prec/3.321928095)
       << " decimals" << endl;
  prec= prec/3.321928095; // precision in decimal digits
  cout.precision(prec);   // perform output with prec decimal digits
```

```
33
34    MpfciClass Y, Z0, Zb, Z, D, CH, Zi, C; // complex intervals
35    MpfcClass approx; // complex point
36    int k, kmax(9);   // iteration counter, maximum number of iteration steps
37    MpfrClass eps= MpfrClass(0.125); // used for epsilon inflation
38
39    while (true) {
40      cout << "The zeros of function F are: 4i, 2+i, and 3-i" << endl;
41      cout << "Approximation of a complex zero
42                ( try e.g. (0.1,4.3) or (-0.2,3.8) for the zero 4i )? ";
43      cin >> approx;
44      Zi= MpfciClass(approx);
45      k= 0;
46
47      MPcitaylor FD;   // function values and values of derivative using ATA
48      MpfciClass FZi;  // function values
49      MpfciClass DFZi; // values of first derivative
50      do { // try to improve approximation
51        k++;
52        cout << "k: " << k << endl;
53        Z0   = Zi;
54        FD   = F( MPcitaylor(1, Zi) ); // compute function value
55                                       // and first derivative via ATA
56        FZi  = get_j_derivative(FD,0); // take function value
57        DFZi = get_j_derivative(FD,1); // take first derivative
58
59        Zi= MpfciClass( mid(Zi - FZi/DFZi) ); // perform Newton step
60        Z = MpfciClass( MpfiClass(Re(Sup(Z0)),Re(Sup(Zi))),
61                        MpfiClass(Im(Sup(Z0)),Im(Sup(Zi))) );
62      } while (common_decimals(Re(Z))<prec/3 && common_decimals(Im(Z))<prec/3
63                                             && k<kmax );
64      cout << "Improved approximation of complex zero: " << endl << Sup(Zi) << endl;
65
66      C = MpfciClass(1/Sup(DFZi));    // C is a point interval
67      FD = F( MPcitaylor(1, Zi) );    // compute function value and first derivative
68      FZi =  get_j_derivative(FD,0); // take function value
69      Y = -C*FZi;
70      Z = Y; // Y,C are the initial values of the interval iteration (verification)
71      k = 0;
72
73      do
74      {
75        k++;
76        cout << "  k: " << k << endl;
77        Zb = Blow(Z, eps); // epsilon inflation of Z
78        CH = Zi | Zi+Zb;   // convex hull
79
80        FD = F( MPcitaylor(1, CH) ); // function value and 1. derivative via ATA
81
82        D = 1 - C*get_j_derivative(FD,1);
83        Z = Y + D*Zb;
84      } while ( (k<kmax) && !in(Z,Zb) );
85
86      if ( in(Z,Zb) )
87      { // inclusion verified
88        cout << "Verification successful, inclusion of a simple zero: "
89             << endl << Zi + Z << endl;
90      }
91      else cout << "Verification failed" << endl;
92    }
93
94    return 0;
95  }
```

Das Programm liefert die Ausgabe:

```
Current-precision: about 30 decimals
The zeros of function F are: 4i, 2+i, and 3-i
Approximation of a complex zero
    ( try e.g. (0.1,4.3) or (-0.2,3.8) for the zero 4i )? (-0.2,3.8)
k: 1
k: 2
k: 3
k: 4
k: 5
k: 6
Improved approximation of complex zero:
(4.39518159174201567128142086 64e-35, 4.000000000000000000000000000000)
  k: 1
  k: 2
Verification successful, inclusion of a simple zero:
([-8.35609961626862289777645319 73e-64,8.35609961626862289777645319 73e-64],
[3.999999999999999999999999999999,4.000000000000000000000000000001])
```

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 50 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 73 der Einschließungstest.

7.8. Einschließung aller Nullstellen nichtlinearer Funktionen

Die Nullstellenberechnung für differenzierbare, nichtlineare Funktionen ist seit Generationen in der Numerik ein klassisches Arbeitsgebiet. In den meisten Fällen geht man aus von einer Nullstellennäherung und versucht dann, diese Näherung mit iterativen Methoden (Newtonverfahren, Halley's Methode, Fixpunktiteration, Bisektion, ...) nach Möglichkeit weiter zu verbessern. Bis in die 70er Jahre des 20. Jahrhunderts war man dabei der Auffassung, dass es keinen Algorithmus geben kann, mit dem *alle* Nullstellen einer nichtlineare Funktion über einem Intervall mit garantierten Fehlerschranken berechnet werden können, wobei zusätzlich auch noch die Existenz und Eindeutigkeit dieser Nullstellen nachzuweisen ist.

Die Aufgabe besteht darin, zu einer gegebenen, stetig differenzierbaren Funktion $f : \mathbb{R} \to \mathbb{R}$ und $x \in [x_1, x_2] \subseteq \mathbb{R}$ **alle** ihre Nullstellen möglichst eng einzuschließen und die Existenz und Eindeutigkeit dieser Nullstellen nachzuweisen. Den theoretischen Hintergrund findet man in [20], [50], [5], [29]. In diesem Abschnitt werden nur die zur Verfügung stehenden Hilfsmittel beschrieben, mit denen die Nullstellen mit praktisch beliebiger Genauigkeit eingeschlossen werden können. Das zentrale Werkzeug ist dabei die in `mpficlass.cpp` definierte Funktion

```
void AllZeros ( MPddf_FctPtr f,
                MpfiClass    SearchInterval,
                MpfrClass    Tolerance,
                MPivector&   Zero,
                intvector&   Unique,
                int&         NumberOfZeros,
                int&         Err,
                int          MaxNumberOfZeros )
```

Anmerkungen:

1. `f` : Funktion vom Typ `MPDerivType`. Zur Definition siehe z.B. Seite 125 und die Tabelle auf Seite 128.

2. `SearchInterval` : Das Intervall, in dem alle Nullstellen von `f` zu berechnen sind.

3. `Tolerance` : Die relative Genauigkeit für die Einschließung der Nullstellen. Mit `Tolerance` wird die Präzision bestimmt, mit der intern gerechnet werden muss, um die vorgegebene relative Genauigkeit zu erreichen.

4. `Zero` : Dieser Vektor liefert die Einschließungen aller Nullstellenkandidaten.

5. `Unique` : `Unique[i]==0` : Existenz und Eindeutigkeit sind nicht garantiert, d.h `Zero[i]` kann entweder keine Nullstelle oder eine oder mehrere einfache oder mehrfache Nullstellen enthalten.
   `Unique[i]==1` : In `Zero[i]` sind Existenz und Eindeutigkeit garantiert.

6. `NumberOfZeros` : Anzahl der gefundenen Nullstellenkandidaten, wobei aber die Eindeutigkeit nicht bei allen Einschließungen garantiert sein muss. Bei symmetrischer Lage der Nullstellen in `SearchInterval` kann die gleiche Nullstelle im Vektor `Zero` in verschiedenen Komponenten mehrfach eingeschlossen und entsprechend gezählt werden. Um diese Symmetrie zu vermeiden, kann bei einigen Programmversionen (Funktion `compute(...)`) das eingegebene Startintervall nach Bedarf nach rechts oder links zum `SearchInterval` aufgebläht werden, so dass dann in nahezu allen Fällen die gleiche Nullstelle nur durch eine einzige Vektorkomponente `Zero[i]` eingeschlossen wird. Durch diese möglichen Aufblähungen können Nullstellen außerhalb von `SearchInterval` eingefangen werden, die aber anschließend automatisch wieder aussortiert werden.

7. Da ein Nullstellenkandidat `Zero[k]` mit `Unique[k]` = 0 keine Nullstelle einschließen muss, wird bei einigen Programmversionen (Funktion `compute(...)`) durch einen nochmaligen Aufruf von `AllZeros(f,Zero[k],Tolerance,ZeroV,UNIQUE,NOZ,Error)` getestet, ob `NOZ` = 0 erfüllt ist. In diesem Fall ist garantiert, dass `Zero[k]` keine Nullstelle enthält. In den Vektoren `Zero` und `Unique` werden dann die Komponenten `Zero[k], Unique[k]` entsprechend aussortiert.

8. Um die Anzahl der Nullstellenkandidaten weiter zu minimieren, betrachten wir den Fall, dass sich zwei aufeinanderfolgende Nullstellenkandidaten `Zero[k]` und `Zero[k+1]` überlappen. Unter der Bedingung `(Unique[k]==Unique[k+1] && Unique[k]==0)` kann deren Vereinigung `tmp` = `Zero[k]` $\bigcup$ `Zero[k+1]` keine oder mehrere einfache oder mehrfache Nullstellen einschließen. Nach dem folgenden Aufruf

   `AllZeros(f,tmp,Tolerance,ZeroV,UNIQUE,NOZ,Error)`

   wird dann die Bedingung `(NOZ==1 && UNIQUE[1]==1)` getestet. Ist diese erfüllt, so setzt man `Zero[k]=ZeroV[1]` und `Unique[k]=1`, und in den Vektoren `Zero` und `Unique` werden die Komponenten `Zero[k+1]` und `Unique[k+1]` gelöscht. Ist obige Bedingung nicht erfüllt, so setzt man `Zero[k]=tmp` und löscht in den Vektoren `Zero` und `Unique` ebenfalls die Komponenten `Zero[k+1]` und `Unique[k+1]`. In jedem Fall reduziert sich damit die Anzahl der Nullstellen-Kandidaten um eins.
   Wenn bei den beiden sich überlappenden Nullstellenkandidaten `Zero[k]` und `Zero[k+1]` die obige Bedingung `(Unique[k]==Unique[k+1] && Unique[k]==0)` nicht erfüllt ist, so wird nach dem Aufruf von `AllZeros(f,tmp,Tolerance,ZeroV,UNIQUE,NOZ,Error)` die Bedingung `(NOZ==1 && UNIQUE[1]==1)` getestet. Und nur, wenn diese erfüllt ist, setzt man `Zero[k]=ZeroV[1]` und `Unique[k]=1`, und zusätzlich werden in den Vektoren `Zero` und `Unique` die Komponenten `Zero[k+1]` und `Unique[k+1]` gelöscht.
   Durch diese Vorgehensweise wird bei einigen Programmversionen (Funktion `compute()`) die Anzahl der Nullstellenkandidaten minimiert. Den vollständigen Programmcode findet man z.B. ab Seite ....

9. `Err` : Bestimmt die Ausgabe einer möglichen Fehlermeldung.

Der in `AllZeros()` realisierte Algorithmus ist sehr leistungsfähig, und ohne das Auftreten einer Fehlermeldungen werden alle Nullstellen aus dem Startintervall erkannt. Mehrfache Nullstellen werden zwar erkannt und eingeschlossen, die Eindeutigkeit der Nullstelle(n) kann in dieser Einschließung jedoch grundsätzlich nicht nachgewiesen werden. Für diese Einschließungen erfolgt die Meldung *may contain a zero (not verified unique)!* Die gleiche Meldung kann aber auch dann erfolgen, wenn der Kandidat keine Nullstelle enthält oder wenn er eine oder mehrere dicht zusammenliegende einfache Nullstellen enthält, deren Eindeutigkeit, z.B. wegen fast waagerechter Tangente, nicht mehr nachgewiesen werden kann.

Wird bei einer einfachen Nullstelle die Eindeutigkeit nachgewiesen, so erfolgt für diese Nullstelleneinschließung die Meldung *encloses a locally unique zero!*, d.h. außer dieser einfachen Nullstelle gibt es in diese Einschließung keine weitere einfache oder mehrfache Nullstelle.

Die bisherigen Anmerkungen beziehen sich auf die Anwendung der Funktion `AllZeros(...)` bzw. auf eine möglichst optimale Ausgabe der Nullstelleneinschließungen, d.h. auf eine möglichst minimale aber vollständige Anzahl der Nullstellenkandidaten. Die folgenden Anmerkungen beziehen sich auf eine möglichst geschickte Darstellung der zu untersuchenden Funktion, deren Nullstellen zu bestimmen sind. Dabei zeigt sich, dass bei komplizierten Funktionen deren optimale Darstellung oft ein eigenständiges Problem ist.

Anmerkungen:

1. Wenn ein Funktionsterm beispielsweise den Ausdruck $\sqrt{1+[x]\cdot[x]}$ enthält, so sollte dieser durch $\sqrt{1+\mathrm{sqr}([x])}$ ersetzt werden, da sonst mit $[x]=[-2,2]$ der Radikand $1+[x]\cdot[x]$ die Null enthält und die Quadratwurzel damit eine Fehlermeldung erzeugt. Um zusätzlich einen vorzeitigen Überlauf zu vermeiden, benutzt man am besten die Funktion `sqrt1px2`(x).

2. Ein Funktionsterm sollte die unabhängige Variable x nur in minimaler Anzahl enthalten. Daher sollte z.B. $(x^2-1)/(x-1)$ durch den äquivalenten Ausdruck $x+1$ ersetzt werden.

3. Ein Polynom (1) $P_N(x) := a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_N \cdot x^N$ sollte nur nach dem Horner-Schema

```
r = a[N]*x + a[N-1];
for (int i=N-2; i>=0; i--)
  r = r*x + a[i];
```

   mit $P_N(x) = r$ ausgewertet werden, da das Horner-Schema weniger Rechenoperationen benötigt und die Potenzen x^i bei der Intervallauswertung zu große Überschätzungen verursachen. Bei aufwendigen Polynomen kann die Darstellung (1) bei der Nullstellenberechnung mit `AllZeros(...)` zum Programmabbruch führen. Dies zeigt, dass die geschickte Darstellung der auszuwertenden Funktion für die erfolgreiche Nullstellenberechnung oft entscheidend ist.

4. Bei komplizierten Nullstellenproblemen (z.B. fast waagerechte oder senkrechte Nullstellen-Tangenten, oder sehr dicht benachbarte einfache Nullstellen) führt oft auch eine geschickte Wahl des Funktionsterms nicht zum Ziel. Der Grund ist dann oft, dass im Funktionsterm die unabhängige Variable x zu häufig vorkommt und damit bei der Intervallauswertung zu große Überschätzungen verursacht, die auch mit noch so großer Präzision nicht beseitigt werden kann. Abhilfe schafft bei nicht zu breiten Intervallen $[x]$ bei der Auswertung von $f[x]$ dann oft die Anwendung der zentralen Form (*centered form*), die für differenzierbare Funktionen f gegeben ist durch [20],[6],[56],[30]

   (7.2) $$f(x) \in f(c) + f'([x]) \cdot ([x]-c), \quad c, x \in [x].$$

   Wählt man $c := \texttt{mid}([x]$, so spricht man von der Mittelwertform (*mean-value form*). Zu beachten ist, dass die Güte der Einschließung von $f(x)$, $x \in [x]$, nur von $f'([x])$ abhängt, so dass auch $f'([x])$ wieder mit Hilfe der Mittelwertform dargestellt werden sollte und damit auch $f''([x])$ benötigt wird. Die höheren Ableitungen, etwa bis zur Ordnung k, können dabei mit Hilfe der implementierten Taylorarithmetik bequem berechnet werden.

5. Bei einigen Problemstellungen, z.B. bei der Bestimmung aller relativen Extrema eine zweimal differenzierbaren Funktion $f(x)$, benötigt man alle Nullstellen von $f'(x)$. Mithilfe der Taylorarithmetik ist es grundsätzlich kein Problem, zu einer n-mal differenzierbaren Funktion $f(x)$ die n-te Ableitung $f^{(n)}(x)$ zu berechnen und dann mit `AllZeros()` alle Nullstellen von $f^{(n)}(x)$ für $x \in [x]$ zu bestimmen. Bei einigen Programmversionen (Funktion `compute()`) kann dies durch die entsprechende interaktive Eingabe von n realisiert werden.

Die folgenden vier Beispielprogramme zeigen die geschilderten, verschiedenen Anwendungsmöglichkeiten zur Nullstellenbestimmung, dabei werden auch die Grenzbereiche angesprochen, in denen die gesuchten Nullstellen nicht oder nicht mehr vollständig angegeben werden können. Das erste Programm `MPFR-15.cpp` besitzt die einfachste Funktion `void compute(...)`, in der keine Mittelwertform, keine Intervall Aufblähung und keine optimierte Kandidatenausgabe zur Anwendung kommt.

Das folgende Programm `MPFR-15.cpp` berechnet für $f(x) = e^{-3x} - \sin^3(x)$ die Einschließungen aller auftretenden einfachen Nullstellen im vorgegebenen Startintervall $[0, 20]$. Die relative Genauigkeit für die einzuschließenden Nullstellen muss eingegeben werde. Mit z.B. `Tolerance` = $1e-400$ wird die interne Präzision auf `prec` = $-1.5 \cdot$ `expo(Tolerance)` $+ 53 \approx 2045$ festgesetzt, so dass Einschließungen mit etwa $2045/3.321928095 \approx 616$ korrekten Dezimalstellen berechnet werden. Falls `prec` kleiner ist als die CurrentPrecision, wird `prec` auf die CurrentPrecision gesetzt.

```cpp
// Programm  MPFR-15.cpp  zur Einschliessung aller Nullstellen
// einer Funktion f vom Typ MPDerivType
#include <iostream>
#include "mpficlass.hpp"
#include <stacksz.hpp>

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

MPDerivType f (const MPDerivType& x)
{
  // return exp(-30.0*sqr(x)) - power(sin(x), 3); // Tolerance = 1e-4000 works
  return exp(-3.0*x) - power(sin(x), 3);
}

//------------------------------------------------------------------------
// Function for prompting and reading information to call the function
// 'AllZeros()'. This function must be called with the function 'f'
// and a string 'Name' containing a textual description of that function.
//------------------------------------------------------------------------

void compute ( MPddf_FctPtr f, const char *Name )
{
  MpfiClass    SearchInterval;
  MpfrClass    Tolerance;
  MPivector    Zero;
  intvector    Unique;
  int          NumberOfZeros, i, Error;
  PrecisionType prec;

  cout << "Computing all zeros of the function  " << Name << endl;
  cout << "Search interval      : ";
  cin  >> SearchInterval;
  cout << "Tolerance (relative) : ";
  cin  >> Tolerance;
  cout << endl;

  if (Tolerance>1e-8) Tolerance = 1e-8;
  // With the entered 'Tolerance' the corresponding output format is calculated:
  prec = -expo(Tolerance) + 1;
  if (prec < 53) prec = 53;
  cout.precision(prec/3.321928095); // Only for the output format;

  AllZeros(f,SearchInterval,Tolerance,Zero,Unique,NumberOfZeros,Error);

  if (Error) cout << endl << AllZerosErrMsg(Error) << endl;
  else
  {
    for ( i = 1; i <= NumberOfZeros; i++)
    {
      cout << Zero[i] << endl;
      if (Unique[i])
        cout << "encloses a locally unique zero!" << endl;
      else
        cout << "may contain a zero (not verified unique)!" << endl;
    }
    cout << "Calculated " << NumberOfZeros << " interval enclosure(s)" << endl;
  }
} // compute()
```

```
62 int main(void)
63 {
64   int base;
65   MpfrClass::SetBase (10);
66   base = MpfrClass::GetBase();
67   cout << "Base = " << base << endl;
68
69   compute(f, "EXP(-3x) - POWER(SIN(x),3)");
70
71   return 0;
72 }
```

Beim Programmstart erhält man den folgenden Ausdruck:

```
Base = 10

Computing all zeros of the function  EXP(-3x) - POWER(SIN(x),3)

Search interval      : [0,20]
Tolerance (relative) : 1e-34

[5.8853274398186107743245204570290 36e-1,5.8853274398186107743245204570290 37e-1]
encloses a locally unique zero!
[3.0963639324106461156258408499040 24,3.0963639324106461156258408499040 25]
encloses a locally unique zero!
[6.2850492733825865338483009697841 56,6.2850492733825865338483009697841 57]
encloses a locally unique zero!
[9.4246972547385212191158651839181 11,9.4246972547385212191158651839181 12]
encloses a locally unique zero!
[1.2566374101689367676822016042515 79e1,1.2566374101689367676822016042515 80e1]
encloses a locally unique zero!
[1.5707963117247215942290368284687 96e1,1.5707963117247215942290368284687 97e1]
encloses a locally unique zero!
[1.8849555928051171524444249658063 80e1,1.8849555928051171524444249658063 81e1]
encloses a locally unique zero!
Calculated 7 interval enclosure(s)
```

Anmerkungen:

1. Wir erhalten also 7 eindeutig bestimmte Nullstellen, die mit einer Genauigkeit von etwa 221 Dezimalstellen eingeschlossen werden, und der verwendete Algorithmus garantiert wegen der nachgewiesenen Eindeutigkeit, dass es im Innern von $[0, 20]$ keine weiteren Nullstellen geben kann. Die obige Ausgabe ist auf 34 Dezimalstellen beschränkt.

2. Die Funktion $g(x) = -\sin^3(x)$ besitzt bei $x_k = k \cdot \pi,\ k \in \mathbb{Z}$, nur dreifache Nullstellen mit waagerechter Tangente. Solche mehrfachen Nullstellen werden vom Algorithmus zwar eingeschlossen, die Eindeutigkeit kann aber nicht nachgewiesen werden. Der Nachweis der Eindeutigkeit gelingt nur bei einfachen Nullstellen!

3. Addiert man zu $g(x) = -\sin^3(x)$ den kleinen positiven Funktionswert e^{-3x}, so werden die mehrfachen Nullstellen von $g(x)$ zu einfachen Nullstellen von $f(x) = e^{-3x} - \sin^3(x)$, die mit obigem Programm erfolgreich eingeschlossen werden.

4. Wählt man in $f(x) = e^{-3x} - \sin^3(x)$ den für $x > 0$ noch viel kleineren Summanden e^{-30x^2}, so besitzt $h(x) := e^{-30x^2} - \sin^3(x)$ ebenfalls einfache Nullstellen, jetzt jedoch mit nahezu waagerechten Tangenten, und der Nachweis der Eindeutigkeit dieser Nullstellen in ihren Einschließungen gelingt in $[0, +20]$ nur mit der viel kleineren Toleranz $10^{-2\,000}$, wodurch aber erhebliche Laufzeiten erforderlich werden.

5. $u(x) = \sin^2(x) - 2^{-20\,000}+$ besitzt in $[3,4]$ zwei sehr eng benachbarte, einfache Nullstellen. Der Algorithmus findet jedoch mit `Tolerance = 1e-34` nur *eine* Einschließung mit der Meldung *may contain a zero (not verified unique)!* Mit dem sehr viel kleineren Wert `Tolerance = 1e-2000` wird eine Einschließung wegen zu vieler interner rekursiver Funktionsaufrufe jedoch nicht gefunden, (Speicherzugriffsfehler). Die gleiche Fehlermeldung erhält man aber auch mit der sehr viel kleineren Fehlerschranke `Tolerance = 1e-40000`, vgl. dazu auch das nächste Beispiel und Seite 170.

6. $v(x) = \cos(x) \cdot \cosh(x) + 1$ besitzt in $[1e9, 1.00000001e9]$ drei einfache Nullstellen mit fast senkrechten Tangenten. Mit `Tolerance = 1e-34` erhält man das Ergebnis:

```
[1.000000000993400903293511449821674e9,1.000000000993400903293511449821675e9]
may contain a zero (not verified unique)!
[1.000000004134993556883304688284317e9,1.000000004134993556883304688284318e9]
may contain a zero (not verified unique)!
[1.000000007276586210473097926746961e9,1.000000007276586210473097926746962e9]
may contain a zero (not verified unique)!
Calculated 3 different interval enclosure(s)
```

   Mit `Tolerance = 1e-4000` erhält man die Fehlermeldung: `Speicherzugriffsfehler` und mit `Tolerance = 1e-6600` erhält man das Ergebnis:

```
[1.0000000009934...92174077065973028e9,1.0000000009934...92174077065973029e9]
encloses a locally unique zero!
[1.0000000041349...87813976055331239e9,1.0000000041349...87813976055331240e9]
encloses a locally unique zero!
[1.0000000072765...83453875044689451e9,1.0000000072765...83453875044689452e9]
encloses a locally unique zero!

Calculated 3 different interval enclosure(s)
```

   Bei einfachen Nullstellen mit nahezu senkrechten Tangenten erhält man also bei diesem Beispiel den Beweis der Eindeutigkeit erst mit hinreichend kleinen relativen Fehlerschranken. Es ist sogar möglich, mit `Tolerance = 1e-2000000` die drei Nullstellen mit jeweils etwa 3 000 000 korrekten Dezimalstellen einzuschließen. Mit wachsender Stellenzahl wächst die Laufzeit jedoch erheblich.

Die nächsten drei Beispielprogramme verlangen innerhalb der Funktion `void compute(...)` die folgenden Eingaben:

1. *n-th derivative of the function, n = ?*, d.h. die Ordnung $n >= 0$ der Ableitung von $f(x)$, deren Nullstellen zu berechnen sind. Die Funktion $f(x)$ selbst ist jeweils vor der Funktion `void compute(...)` zu definieren.

2. *Start intervall [x1,x2] = ?*, d.h. das Intervall in der Form `[1.234e-500,1.23e-490]`, in dem die Nullstellen einzuschließen sind.

3. *The precision in bits of the start interval ?* Die Eingabe von 53 ist meist völlig ausreichend. Wenn jedoch das Startintervall schon sehr schmal ist und noch weiter verbessert werden soll, so muss seine Präzision in Bits eingegeben werden, d.h. besteht z.B. die Intervallunterschranke aus 100 Dezimalstellen, so sollten wenigstens $100 \cdot 3.321928095 \approx 332$ Bits eingegeben werden. Innerhalb der Funktion `void AllZeros(....)` wird dann mit einer CurrentPrecision von wenigsten 332 Bits gerechnet.

4. *Number k of derivatives for mean value forms (e.g. 1 <= k <= 10), k = ?* Hier ist die Anzahl $k \geq 1$ der Ableitungen anzugeben, die bei der benutzten Mittelwertform zur Anwendung kommen sollen.

5. *No inflation: i=0, inflation to left: i=1, inflation to right: i=2; i = ?* Bei symmetrischer Lage der Nullstellen im Startintervall können einige Nullstellenkandidaten mehrfach gezählt werden. Um dies zu vermeiden, kann das Startintervall wahlweise nach rechts oder links minimal aufgebläht werden. Dadurch möglicherweise eingefangene Nullstellenkandidaten werden jedoch anschließend wieder aussortiert.

6. *(Relative) Tolerance for the enclosures (e.g. 1e-20) = ?* Sollen die Nullstellen mit z.B. 20 korrekten Dezimalstellen eingeschlossen werden, so sollte man hier `1e-20` eingeben. Innerhalb von `void AllZeros(...)` wird dann die CurrentPrecision auf mindestens $-1.5 \cdot$ `expo(Tolerance)` $+ 53$ gesetzt. Sollte dieser Wert immer noch zu klein sein, so kann bei der nächsten Eingabeaufforderung ein noch größerer Wert in Bits eingegeben werden.

7. *How many decimals should have the output format = ?* Hier kann die Anzahl der Dezimalstellen im Ausgabeformat festgelegt werden.

Das folgende Programm `MPFR-21` berechnet für ein gegebenes Startintervall $[x]$ alle Nullstellen der n-ten Ableitung $f^{(n)}(x)$ der recht komplizierten Funktion

$$f(x) := \cosh(x-1) - \mathrm{erf}((x+3)/(3+\cos(9x)) + \exp(-\mathrm{sqr}(x)) \cdot \sin(5.5 \cdot x - 2)), \quad x \in [x].$$

Versuchen Sie (lieber nicht!) etwa $f^{(5)}(x)$ mit Papier und Bleistift zu berechnen! Löst man diese Aufgabe mit einem Algebrasystem, so wird man von seitenlangen Funktionstermen erschlagen, und es ist völlig ausgeschlossen, für die Nullstellenbestimmung den optimalen Funktionsterm zu bestimmen. Aber mit `AllZeros(...)` und mithilfe der Mittelwertform (mea-value form, $k = 8$) lassen sich im Intervall $[-14, 14]$ alle 823 einfache Nullstellen erfolgreich einschließen!

```
//================================================================
//   MPFR-21.cpp
//   Calculating enclosures of all zeros of the n-th derivative of the
//   basic expression
//================================================================

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

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

namespace{ int n; } // n-th derivative of the basic expression
                    // is used in function definition of f
namespace{ int k; } // derivatives up to this order are used
                    // by the centered forms
class BasicExpression
{
 public:
  MPitaylor operator()(const MPitaylor& x)
  { // Evaluation of expresssion:
    MPitaylor r = cosh(x-1)-erf((x+3)/(3+cos(9*x)) + exp(-sqr(x))*sin(5.5*x-2));
    // MPitaylor r = 1-cos((x-0.992)*(x-0.993)); // Test for many candidates
    // MpfrClass t(0.5);
    // MPitaylor r = sqr(sin(x)) - comp(t,-19999);
    return r;
  } //operator

} basic_f; // class BasicExpression
           // with one global object basic_f (used in function f)
MPDerivType f (const MPDerivType& xx)
{
  MPitaylor x(::n+::k, fValue(xx));        // independent variable x
```

```
36    MPitaylor xmid(n+k, mid(fValue(xx)));  // independent variable xmid
37    MPitaylor rinf;
38    MPitaylor rsup;
39
40    MPitaylor rx = basic_f(x);  // function evaluated with interval argument
41
42    MpfrClass::SetCurrPrecision(MpfrClass::GetCurrPrecision()+10);
43    MPitaylor rmidx = basic_f(xmid); // function evaluated with
44                                     // point interval argument
45    MpfrClass::SetCurrPrecision(MpfrClass::GetCurrPrecision()-10);
46
47    MpfiClass y = fValue(xx) - mid(fValue(xx)); // used in mean-value forms
48
49    MpfiClass g[k+1]; // g[0]= f, g[1]= df, g[2]= ddf, ...
50
51    g[k] = get_j_derivative(rx,n+k); // highest derivative
52    for(int i = k-1; i>=0; i--)
53      g[i] = get_j_derivative(rmidx,n+i) + g[i+1]*y; // central form
54
55    MpfiClass dummy;
56    return MPDerivType(g[0], g[1], dummy);
57  } // MPDerivType f
58
59  //-------------------------------------------------------------------------
60  // Function for prompting and reading information to call the function
61  // 'AllZeros()'. This function must be called with the function 'f'
62  // and a string 'Name' containing a textual description of that function.
63  //-------------------------------------------------------------------------
64
65  void compute ( MPddf_FctPtr f, const char *Name )
66  {
67    MpfiClass    SearchInterval;
68    MpfrClass    Tolerance,
69                 SupStart, InfStart, EpsBlow;
70    MPivector    Zero, ZeroV;
71    intvector    Unique, UNIQUE;
72    int          NumberOfZeros, NOZ, i, Error;
73    long int K;
74    PrecisionType prec;
75    string str;
76
77    cout << endl << " ************************************************" << endl;
78    cout << "   ***" << endl;
79    cout << "   *** Computing all zeros of the "
80         << "n-th derivative of the function  " << Name << endl;
81    cout << "   *** n = ?  "; cin >> n;
82    cout << "   *** The zeros of the " << n << "-th derivative of the function "
83                                                              << endl;
84    cout << "   *** " << Name << " are to be calculated." << endl;
85    cout << "   *** Start intervall  [x1,x2] = ?  "; cin >> str;
86    cout << "   *** The precision in bits of the start interval must be entered."
87                                                              << endl;
88    cout << "   *** (e.g. 10 decimals require a precision of 10*3.3219280 bits) ? ";
89    cin >> prec;
90    if (prec<53) prec = 53;
91    MpfiClass Start(str,prec); // All zeros in Start are to be enclosured.
92    set_Mpfi(SearchInterval, Start, Start.GetPrecision());
93    cout << "   *** Number k of derivatives for mean-value forms (e.g. 1<=k<=10),
94    if (k<1) k=1;
95    cout << "   *** k for mean value forms: " << ::k << endl;
96    cout << "   *** To avoid a possible symmetric location of the zeros in " << endl;
97    cout << "   *** the start interval a small inflation can be realized. " << endl;
98
99    do {
```

```
100   cout << "  *** No inflation: i=0, inflation to left: i=1,
101                                    inflation to right: i=2; i = ?  "; cin >> i;
102   } while (i<0 || i>3);
103
104   if (i>0)
105   {
106     K = -0.86*SearchInterval.GetPrecision();
107     EpsBlow = 1;
108     times2pown(EpsBlow, K);
109     MpfiClass::SetCurrPrecision(SearchInterval.GetPrecision());
110     if (i==1)
111       SearchInterval = Blow_L(SearchInterval, EpsBlow);
112     else // i=2
113       SearchInterval = Blow_R(SearchInterval, EpsBlow);
114   }
115   if (i==0)
116     cout << "  *** SearchInterval = " << SearchInterval << endl;
117   else cout << "  *** Blown up SearchInterval = " << SearchInterval << endl;
118   cout << "  *** (Relative) Tolerance for the enclosures (e.g. 1e-20) = ?  ";
119   cin >> str;
120   Tolerance = string2Mpfr(str);
121   cout << "  *** Tolerance = " << Tolerance << endl;
122   cout << "  *** With Tolerance the CurrentPrecision is calculated in" << endl;
123   cout << "  *** AllZeros(...) by:  -1.5*expo(Tolerance)+53 = "
124        << -1.5*expo(Tolerance)+53 << endl;
125   cout << "  *** In some cases this value can be too small.
126                                          So a greater value in bits" << endl;
127   cout << "  *** for the CurrentPrecision can be neccessary: (e.g. 300) = ?  ";
128   cin >> prec;
129   if (prec < 53) prec = 53;
130   MpfrClass::SetCurrPrecision(prec);
131   cout << "  *** Minimum CurrentPrecision in bits: "
132        << MpfrClass::GetCurrPrecision() << endl;
133   prec = -expo(Tolerance)/3.321928095 + 1;
134   cout << "  *** With respect to Tolerance each of the zero enclosures " << endl;
135   cout << "  *** should have " << prec << " common decimals." << endl;
136   cout << "  *** How many decimals should have the output format = ? ";
137   cin >> prec;
138   if (prec < 10) prec = 10;
139   cout.precision(prec); // only for the output format
140   cout << endl << "  *** Computing all zeros of the " << n
141        << "-th derivative of " << endl;
142   cout << "       " << Name << endl;
143   cout << "  *** please wait some moments:" << endl;
144
145   AllZeros(f, SearchInterval, Tolerance, Zero, Unique, NumberOfZeros, Error);
146
147   cout << endl << "  *** All zero enclosures are calculated ... "
148        << endl << endl;
149   if (Error)
150     cout << endl << AllZerosErrMsg(Error) << endl;
151   else
152   {
153     SupStart = Sup(Start);  InfStart = Inf(Start);
154     if (i==2) // Inflation to right
155     // In Zero and Unique those zero enclosures are sorted out
156     // with Inf(Zero[k])>Sup(Start).
157     { i = 0;
158       for (int k=1; k<=NumberOfZeros; k++)
159       {
160         if (Inf(Zero[k-i])<=SupStart)
161         {
162           Zero[k-i]   = Zero[k];
163           Unique[k-i] = Unique[k];
```

```
164         }
165         else i++;
166       }
167     }
168     else
169     {
170       if (i==1) // Inf(SearchInterval) < Inf(Start)
171       // In Zero and Unique those zero enclosures are sorted out
172       // with Sup(Zero[k])<Inf(Start).
173       { i = 0;
174         for (int k=1; k<=NumberOfZeros; k++)
175         {
176           if (Sup(Zero[k-i])>=InfStart)
177           {
178             Zero[k-i]   = Zero[k];
179             Unique[k-i] = Unique[k];
180           }
181           else i++;
182         }
183       }
184     }
185     NumberOfZeros = NumberOfZeros - i;
186     // Candidates Zero[k] with Unique[k] = 0 are sorted out if it is guaranted
187     // that no zero is enclosed.    However, it is possible that the remaining
188     // candidates, with Unique[k] = 0, enclose NO zero!
189
190     i=0;
191     for (int k=1; k<=NumberOfZeros; k++)
192     {
193       if (Unique[k] == 0)
194       {
195         AllZeros(f, Zero[k], Tolerance, ZeroV, UNIQUE, NOZ, Error);
196         if (NOZ==0) // No zero found
197           i++;      // Elimination of Zero[k] including no zero
198         else
199         {
200           Zero[k-i]   = Zero[k];
201           Unique[k-i] = Unique[k];
202         }
203       }
204       else
205       {
206         Zero[k-i]   = Zero[k];
207         Unique[k-i] = Unique[k];
208       }
209     }
210     NumberOfZeros = NumberOfZeros - i;
211     // Now successive and overlapping enclosures are tested and
212     // under certain conditions one of these are sorted out:
213     i=0;
214     MpfiClass tmp;
215     for (int k=1; k<=NumberOfZeros-1; k++)
216     {
217       if (!Disjoint(Zero[k-i], Zero[k-i+1])) // intersection is not empty
218       {
219         tmp = Zero[k-i] | Zero[k-i+1];
220         if (Unique[k-i]==Unique[k+1-i] && Unique[k-i]==0)
221         {
222           AllZeros(f, tmp, Tolerance, ZeroV, UNIQUE, NOZ, Error);
223           if (NOZ==1 && UNIQUE[1]==1)
224           {
225             Zero[k-i]   = ZeroV[1]; Unique[k-i] = 1;
226           }
227           else
```

```
228             Zero[k-i] = tmp;
229           MPivec_del(Zero, k-i+1);
230           intvec_del(Unique, k-i+1);
231           NumberOfZeros--;
232           i++;
233         }
234         else
235         {
236           AllZeros(f, tmp, Tolerance, ZeroV, UNIQUE, NOZ, Error);
237           if (NOZ==1 && UNIQUE[1]==1)
238           {
239             Zero[k-i]   = ZeroV[1];
240             Unique[k-i] = 1;
241             MPivec_del(Zero, k-i+1);
242             intvec_del(Unique, k-i+1); NumberOfZeros--;
243             i++;
244           }
245         }
246       } // !Disjoint( )
247     }   // for (...)
248     cout << "  ********************************************" << endl << endl;
249     for (int i=1; i<=NumberOfZeros; i++)
250     {
251       cout << Zero[i] << endl;
252       if (Unique[i])
253         cout << "encloses a locally unique zero!" << endl;
254       else
255         cout << "may contain a zero (not verified unique)!" << endl;
256     }
257     cout << endl << "Calculated " << NumberOfZeros << " interval enclosure(s)"
258          << endl;
259     bool empty(true);
260     for (int i=1; i<=NumberOfZeros; i++)
261     {
262       for (int j=i+1; j<=NumberOfZeros; j++)   // pairwise empty intersections?
263         empty = empty && Disjoint(Zero[i], Zero[j]);
264     }
265     cout << boolalpha << "Different enclosures do NOT overlap: "
266          << empty << endl;
267     bool allUnique(true);
268     for (int i=1; i<=NumberOfZeros; i++)
269       allUnique = allUnique && Unique[i];
270     cout << "Every enclosure contains a locally unique zero: "
271          << allUnique << endl;
272     cout.precision(Start.GetPrecision()/3.321928095);
273     if (empty && allUnique)
274     {
275       cout << "There is/are exactly " << NumberOfZeros
276            << " zero(s) in the interval " << endl;
277       cout << "   " << Start << endl;
278     }
279     cout << "The zeros of the *** " << n
280          << "-th derivative *** of the basic expression have been computed."
281          << endl << endl;
282   }
283   return;
284 } // compute
285
286 int main(void) {
287   compute(f, "cosh(x-1)-erf((x+3)/(3+cos(9*x)) + exp(-sqr(x))*sin(5.5*x-2))");
288   return 0;
289 }
```

Mit den folgenden Eingaben

$$n = 20,\ [-14,14],\ 53,\ k = 8,\ i = 0,\ 1e-20,\ 152,\ 20; \tag{7.3}$$

erhält man die Ausgabe, wobei hier aus Platzgründen nur die letzte Einschließung angegeben wird:

```
*** Computing all zeros of the 20-th derivative of
    cosh(x-1)-erf((x+3)/(3+cos(9*x)) + exp(-sqr(x))*sin(5.5*x-2))
*** please wait some moments:
                          .
                          .
[1.39863986165066814584144317220e1,1.39863986165066814584144317221e1]
encloses a locally unique zero!

Calculated 823 interval enclosure(s)
Different enclosures do NOT overlap: true
Every enclosure contains a locally unique zero: true
There is/are exactly 823 zero(s) in the interval
  [-1.40000000000000e1,1.40000000000000e1]
The zeros of the *** 20-th derivative ***
of the basic expression have been computed.
```

Die 823 Nullstelleneinschließungen werden intern mit etwa 45 korrekten Dezimalstellen berechnet und oben mit 30 Dezimalstellen ausgegeben. Will man die letzte Nullstelle mit z.B. 5000 Dezimalstellen berechnen, so benutzt man die obige Einschließung und startet das Programm `MPFR-21` mit den Eingaben

$$n = 20,\ [1.39863986165066814584144317220e1, 1.39863986165066814584144317221e1],$$
$$100,\ k = 8,\ i = 0,\ 1e-5000,\ 24967,\ 5000;$$

und erhält so die Einschließung der letzten Nullstelle mit 5000 korrekten Dezimalziffern:

```
[1.398639861650668...818636349627e1,1.398639861650668...818636349628e1
encloses a locally unique zero!
```

In einer `for`-Schleife kann man mit Hilfe des Vektors `Zero`, der die 823 Einschließungen mit 30 korrekten Dezimalstellen enthält, die Funktion `AllZeros(...)` für jeden Kandidaten `Zero[k]` mit den obigen Eingabedaten nochmals aufrufen und damit alle 823 Einschließungen mit 5000 korrekten Dezimalziffern berechnen. Ganz analog können die 823 Einschließungen dann auch mit einer Genauigkeit von z.B. 20 000 Dezimalstellen eingeschlossen werden.

Es sei noch angemerkt, dass auch alle 1045 Nullstellen von $f^{(25)}$ mit den Eingaben

$$n = 25,\ [-14,14],\ 53,\ k = 8,\ i = 0,\ 1e-20,\ 152,\ 20;$$

erfolgreich eingeschlossen werden.

Für die Funktion $f(x) := 1 - \cos((x-0.992)\cdot(x-0.993))$ erhält man mit den Eingabedaten

$$n = 0,\ [0.75,1],\ 53,\ k = 1,\ i = 0,\ 1e-20,\ 152,\ 20;$$

die folgenden Einschließungen der beiden doppelten Nullstellen:

```
[9.919999999999999288e-1,9.919999999999999291e-1]
may contain a zero (not verified unique)!
[9.929999999999999377e-1,9.929999999999999379e-1]
may contain a zero (not verified unique)!
```

Wählt man statt `1e-20` den `Tolerance`-Wert `1e-100`, so erhält man optimal nur die zwei Nullstellenkandidaten, wenn die obige Eingabe 152 durch 1000 ersetzt wird. Bei zu kleinen Werten erhält man zusätzliche Nullstellenkandidaten mit *may contain a zero (not verified unique)!*. Das Beispiel zeigt, wie nützlich es ist, dass man in Zeile 128, nach der Eingabe von `Tolerance`, den Mindestwert der `CurrentPrecision` in `AllZeros(...)` weiter vergrößern kann.

Die Funktion $f(x) := \sin^2(x) - 2^{-20\,000}$, vgl. Seite 163, besitzt in $[3.1, 3.2]$ zwei einfache aber sehr dicht benachbarte Nullstellen mit fast waagerechter Tangente, die auch jetzt z.B. mit den Eingabedaten

$$n = 0,\ [3.1415...3751, 3.1415...3752],\ 180,\ k = 1,\ i = 0,\ 1e-25\,000,\ 124650,\ 50;$$

wegen der Fehlermeldung *Speicherzugriffsfehler* nicht getrennt werden können. Mit den Eingabedaten

$$n = 0,\ [3.1, 3.2],\ 53,\ k = 1,\ i = 0,\ 1e-30,\ 202,\ 30;$$

erhält man lediglich die Einschließung

```
[3.14159265358979323846264338327,3.14159265358979323846264338329]
may contain a zero (not verified unique)!
```

Das prinzipiell einfache Beispiel zeigt, dass auch die Funktion `AllZeros(...)` nicht alle Nullstellenprobleme lösen kann, aber vorhandene Nullstellen werden grundsätzlich nicht übersehen, wenn das Programm ohne Fehlermeldung beendet wird. Das obige spezielle Problem lässt sich natürlich sehr einfach lösen, wenn $f(x)$ als Produkt $f(x) = (\sin(x) + 2^{-10\,000}) \cdot (\sin(x) - 2^{-10\,000})$ geschrieben wird und anschließend die einfache Nullstelle jedes einzelnen Faktors separat berechnet wird. Auch hier ist die Wahl des günstigsten Funktionsterms wieder entscheidend.

Mit dem folgenden Programm `MPFR-22.cpp` werden die Nullstellen von $w_{20}^{(n)}(x)$ berechnet für $0 \le n \le 20$, wobei das Wilkinson-Polynom $w_{20}(x)$ definiert ist durch [66]

$$w_{20}(x) := (x-1)\cdot(x-2)\cdot\ldots\cdot(x-20) = \sum_{i=0}^{20} a_i \cdot x^i, \quad a_{20} = 1,\ a_{19} = -210,\ \ldots$$

Da die Funktionswerte $w_{20}(x)$ selbst und damit auch Lage der Nullstellen $x_i = i,\ i = 1, 2, ..., 20$, sehr empfindlich ist bezüglich nur kleiner Änderungen der ganzzahligen Koeffizienten a_i, muss zunächst sichergestellt werden, dass die a_i im Rechner exakt gespeichert werden. Dies wird im folgenden Quellcode realisiert von Zeile 34 bis 65. Der Quellcode ab Zeile 80 stimmt mit dem Quellcode von `MPFR-21.cpp`, dort ab Zeile 33, genau überein und wird hier aus Platzgründen weggelassen.

```
1  //=============================================================================
2  //    MPFR-22.cpp
3  //    Calculating enclosures of all zeros of the n-th derivative of the
4  //    basic expression
5  //    Wilkinson polynomial of degree 20
6  //=============================================================================
7
8  #include <iostream>
9  #include "mpficlass.hpp"
10
11 using namespace cxsc;
12 using namespace std;
13 using namespace MPFR;
14 using namespace MPFI;
15
16 namespace{ int n; } // n-th derivative of the basic expression
17                     // is used in function definition of f
18 namespace{ int k; } // derivatives up to this order are used
19
20 class BasicExpression {
21 #define MP MpfiClass
22  public:
23   static const int degree = 20;
24   MpfiClass a[degree+1]; // coeff.: a[degree] ... a[0]
25
```

```
26  public:
27   BasicExpression() // constructor
28   {
29     PrecisionType oldPrec(MP::GetCurrPrecision());
30     MP::SetCurrPrecision(57);
31     for (int i=degree; i>=0; i--) a[i] = 0;
32
33     // Wilkinson polynomial of degree 20
34     // Initialization of the coefficients:
35       a[20]= MP( 1);
36       a[19]= MP(-210);
37       a[18]= MP( 20615);
38       a[17]= MP(-1256850.0);
39       a[16]= MP( 53327946.0);
40       a[15]= MP(-1672280820.0);
41       a[14]= MP( 40171771630.0);
42       a[13]= MP("-756111184500.0");
43       a[12]= MP(" 11310276995381.0");
44       a[11]= MP("-135585182899530.0");
45       a[10]= MP(" 1307535010540395.0");
46       a[9]=  MP("-10142299865511450.0");
47       a[8]=  MP(" 63030812099294896.0");
48       a[7]=  MP("-311333643161390640.0");
49       a[6]=  MP(" 1206647803780373360.0");
50       a[5]=  MP("-3599979517947607200.0"),
51       a[4]=  MP(" 8037811822645051776.0");
52       a[3]=  MP("-12870931245150988800.0");
53       a[2]=  MP(" 13803759753640704000.0");
54       a[1]=  MP("-8752948036761600000.0");
55       a[0]=  MP(" 2432902008176640000.0");
56
57       for (int i=degree; i>=0; i--) // check whether coeff are exactly repr.
58       {
59         if (diam(a[i])!=0)
60         {
61           cout << "  *** coeff a[" << i << "] not representable,
62                        increase the CurrenPrecision!" << endl;
63           exit(1);
64         }
65       }
66     MP::SetCurrPrecision(oldPrec); // restore precision setting
67     return;
68   }  // constructor
69
70   MPitaylor operator()(const MPitaylor& x)
71   {
72     // Compute Taylor coefficients of basic expression r(x) via Horner's scheme
73     MPitaylor r= a[degree]*x + a[degree-1]; //expanded form of polynomial is used
74     for (int i=degree-2; i>=0; i--)
75       r = r*x+a[i];
76     return r;
77   } // operator
78
79 } basic_f; // class with one global object basic_f (used in function f)
```

Mit den Eingabedaten

$$n = 0,\ [1,20],\ 53,\ k = 4,\ i = 0,\ 1e-30,\ 202,\ 30;$$

erhält man die Ausgabe der Nullstelleneinschließungen mit 30 korrekten Dezimalstellen:

```
*** Computing all zeros of the 0-th derivative of
    Wilkinson polynomial of degree 20
*** please wait some moments:

*** All zero enclosures are calculated ...

*******************************************************************************

[9.99999999999999999999999999999e-1,1.00000000000000000000000000001]
encloses a locally unique zero!
[1.99999999999999999999999999999,2.00000000000000000000000000001]
encloses a locally unique zero!
[2.99999999999999999999999999999,3.00000000000000000000000000001]
encloses a locally unique zero!
[3.99999999999999999999999999999,4.00000000000000000000000000001]
encloses a locally unique zero!
[4.99999999999999999999999999999,5.00000000000000000000000000001]
encloses a locally unique zero!
[5.99999999999999999999999999999,6.00000000000000000000000000001]
encloses a locally unique zero!
[6.99999999999999999999999999999,7.00000000000000000000000000001]
encloses a locally unique zero!
[7.99999999999999999999999999999,8.00000000000000000000000000001]
encloses a locally unique zero!
[8.99999999999999999999999999999,9.00000000000000000000000000001]
encloses a locally unique zero!
[9.99999999999999999999999999999,1.00000000000000000000000000001e1]
encloses a locally unique zero!
[1.09999999999999999999999999999e1,1.10000000000000000000000000001e1]
encloses a locally unique zero!
[1.19999999999999999999999999999e1,1.20000000000000000000000000001e1]
encloses a locally unique zero!
[1.29999999999999999999999999999e1,1.30000000000000000000000000001e1]
encloses a locally unique zero!
[1.39999999999999999999999999999e1,1.40000000000000000000000000001e1]
encloses a locally unique zero!
[1.49999999999999999999999999999e1,1.50000000000000000000000000001e1]
encloses a locally unique zero!
[1.59999999999999999999999999999e1,1.60000000000000000000000000001e1]
encloses a locally unique zero!
[1.69999999999999999999999999999e1,1.70000000000000000000000000001e1]
encloses a locally unique zero!
[1.79999999999999999999999999999e1,1.80000000000000000000000000001e1]
encloses a locally unique zero!
[1.89999999999999999999999999999e1,1.90000000000000000000000000001e1]
encloses a locally unique zero!
[1.99999999999999999999999999999e1,2.00000000000000000000000000001e1]
encloses a locally unique zero!

Calculated 20 interval enclosure(s)
Different enclosures do NOT overlap: true
Every enclosure contains a locally unique zero: true
There is/are exactly 20 zero(s) in the interval
  [1.00000000000000,2.00000000000000e1]
The zeros of the *** 0-th derivative *** of the basic expression
have been computed.
```

Das Tschebyscheff-Polynom $T_N(x)$ vom Grad $N = 40$ ist definiert durch

$$T_{40}(x) := \cos(40 \cdot \arccos(x)) = \sum_{k=0}^{20} a_{2k} \cdot x^{2k}; \quad a_0 = 1,\ a_2 = -800, \ldots, a_{40} = 549755813888.$$

Mit folgendem Programm `MPFR-23.cpp` werden für die n-te Ableitung $T_{40}^{(n)}(x)$, mit $0 \le n \le 40$, alle Nullstellen für $x \in [-1, 1]$ berechnet. Dazu muss im folgenden Quellcode zunächst ab Zeile 31 bis 67 dafür gesorgt werden, dass die ganzzahligen Koeffizienten a_{2k} exakt gespeichert werden. Die Abbildung 7.3 zeigt den zur y-Achse symmetrischen Graphen von $T_{40}(x)$ für $x \in [0, 1]$. Man erkennt, dass sich die einfachen Nullstellen am Intervallrand zusammendrängen und die x-Achse fast senkrecht schneiden. Das numerische Problem besteht also darin, auch noch diese Nullstellen erfolgreich einzuschließen.

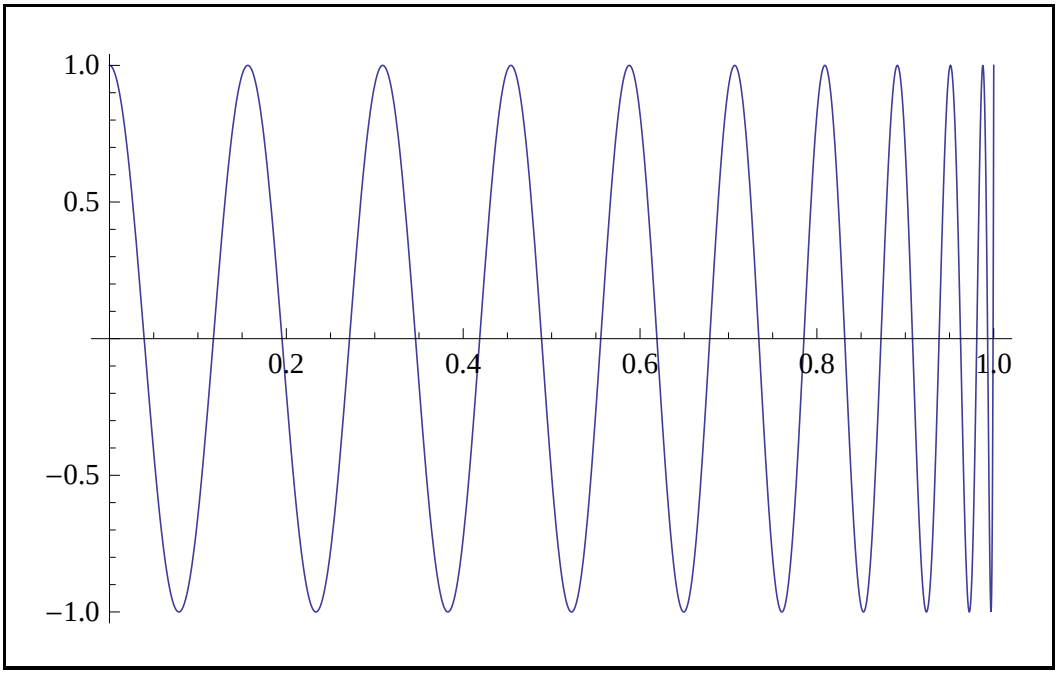


Abbildung 7.3.: Tschebyscheff-Polynom $T_{40}(x),\ x \in [0, +1]$

```
1
2  //================================================================
3  //    MPFR-23.cpp
4  //    Calculating enclosures of all zeros of the n-th derivative of the
5  //    basic expression
6  //    Tschebyscheff polynomial of degree 40
7  //================================================================
8
9  #include <iostream>
10 #include "mpficlass.hpp"
11
12 using namespace cxsc;
13 using namespace std;
14 using namespace MPFR;
15 using namespace MPFI;
16
17 namespace{ int n; } // n-th derivative of the basic expression
18                     // is used in function definition of f
19 namespace{ int k; } // derivatives up to this order are used
20
21 class BasicExpression {
22 #define MP MpfiClass
```

```
23  public:
24   static const int degree = 40;
25   MpfiClass a[degree+1]; // coeff. a[degree] ... a[0]
26
27  public:
28   BasicExpression() // constructor
29   {
30     PrecisionType oldPrec(MP::GetCurrPrecision());
31     MP::SetCurrPrecision(53);
32     for (int i=degree; i>=0; i--) a[i]= 0;
33
34     // Tschebyscheff polynomial of degree 40
35     // Initialization of the coefficients:
36     a[40]= MP(" 549755813888.0");
37     a[38]= MP("-5497558138880.0");
38     a[36]= MP(" 25426206392320.0");
39     a[34]= MP("-72155450572800.0");
40     a[32]= MP(" 140552804761600.0");
41     a[30]= MP("-199183403319296.0");
42     a[28]= MP(" 212364657950720.0");
43     a[26]= MP("-173752901959680.0");
44     a[24]= MP(" 110292369408000.0");
45     a[22]= MP("-54553214976000.0");
46     a[20]= MP(" 21002987765760.0");
47     a[18]= MP("-6254808268800.0");
48     a[16]= MP(" 1424085811200.0");
49     a[14]= MP("-243433472000.0");
50     a[12]= MP(" 30429184000.0");
51     a[10]= MP("-2677768192.0"),
52     a[8]=  MP(" 156900480.0");
53     a[6]=  MP("-5617920.0");
54     a[4]=  MP(" 106400.0");
55     a[2]=  MP("-800.0");
56     a[0]=  MP(" 1.0");
57
58     for (int i=degree; i>=0; i--) // check whether coeff are exactly repr.
59     {
60       if (diam(a[i])!=0)
61       {
62         cout << "  *** coeff a[" << i << "] not representable,
63                   increase the CurrenPrecision!"
64              << endl;
65         exit(1);
66       }
67     }
68     MP::SetCurrPrecision(oldPrec); // restore precision setting
69     return;
70   }  //constructor
71
72   MPitaylor operator()(const MPitaylor& x)
73   { // Compute Taylor coefficients of basic expression r(x) via Horner's scheme
74     MPitaylor r = a[degree]*x + a[degree-1]; // expanded form of polynomial
75     for (int i=degree-2; i>=0; i--)
76       r = r*x+a[i];
77     // MPitaylor r = cos(degree*acos(x));  // polynomial via cos and acos
78     return r;
79   } //operator
80
81 } basic_f; // class with one global object basic_f (used in function f)
```

Der Rest des Programmcodes stimmt überein mit dem Code des Programms `MPFR-21.cpp` auf Seite 164 und wird hier aus Platzgründen nicht angegeben.

Mit den Eingabedaten

$$n = 0,\ [-1,+1],\ 53,\ k = 5,\ i = 0,\ 1e-30,\ 202,\ 30;$$

erhält man die Einschließungen der folgenden 40 einfachen Nullstellen:

```
[-9.9922903624072293473712626034 2e-1,-9.9922903624072293473712626034 1e-1]
enclosesalocallyuniquezero!
[-9.9306845695492629563743724781 1e-1,-9.9306845695492629563743724781 0e-1]
enclosesalocallyuniquezero!
[-9.8078528040323044912618223613 5e-1,-9.8078528040323044912618223613 4e-1]
enclosesalocallyuniquezero!
[-9.6245523645364728763026640518 6e-1,-9.6245523645364728763026640518 5e-1]
enclosesalocallyuniquezero!
[-9.3819133592248413445233972668 7e-1,-9.3819133592248413445233972668 6e-1]
enclosesalocallyuniquezero!
[-9.0814317382508129925808583657 2e-1,-9.0814317382508129925808583657 1e-1]
enclosesalocallyuniquezero!
[-8.7249600707279711452516109922 3e-1,-8.7249600707279711452516109922 2e-1]
enclosesalocallyuniquezero!
[-8.3146961230254523707878837761 8e-1,-8.3146961230254523707878837761 7e-1]
enclosesalocallyuniquezero!
[-7.8531693088074492747034027894 8e-1,-7.8531693088074492747034027894 7e-1]
enclosesalocallyuniquezero!
[-7.3432250943568553563612622218 8e-1,-7.3432250943568553563612622218 6e-1]
enclosesalocallyuniquezero!
[-6.7880074553294174139385555424 2e-1,-6.7880074553294174139385555424 1e-1]
enclosesalocallyuniquezero!
[-6.1909394930983398694156085624 7e-1,-6.1909394930983398694156085624 6e-1]
enclosesalocallyuniquezero!
[-5.5557023301960222474283081394 9e-1,-5.5557023301960222474283081394 8e-1]
enclosesalocallyuniquezero!
[-4.8862124149695494742019088783 9e-1,-4.8862124149695494742019088783 8e-1]
enclosesalocallyuniquezero!
[-4.1865973753742808667556520512 2e-1,-4.1865973753742808667556520512 1e-1]
enclosesalocallyuniquezero!
[-3.4611705707749297646821499492 9e-1,-3.4611705707749297646821499492 8e-1]
enclosesalocallyuniquezero!
[-2.7144044986507425334378740129 6e-1,-2.7144044986507425334378740129 5e-1]
enclosesalocallyuniquezero!
[-1.9509032201612826784828486847 8e-1,-1.9509032201612826784828486847 7e-1]
enclosesalocallyuniquezero!
[-1.1753739745783764410556826684 1e-1,-1.1753739745783764410556826684 0e-1]
enclosesalocallyuniquezero!
[-3.9259815759068609020803363798 4e-2,-3.9259815759068609020803363798 3e-2]
enclosesalocallyuniquezero!
[3.9259815759068609020803363798 3e-2,3.9259815759068609020803363798 4e-2]
enclosesalocallyuniquezero!
[1.1753739745783764410556826684 0e-1,1.1753739745783764410556826684 1e-1]
enclosesalocallyuniquezero!
[1.9509032201612826784828486847 7e-1,1.9509032201612826784828486847 8e-1]
enclosesalocallyuniquezero!
[2.7144044986507425334378740129 5e-1,2.7144044986507425334378740129 6e-1]
enclosesalocallyuniquezero!
[3.4611705707749297646821499492 8e-1,3.4611705707749297646821499492 9e-1]
enclosesalocallyuniquezero!
[4.1865973753742808667556520512 1e-1,4.1865973753742808667556520512 2e-1]
enclosesalocallyuniquezero!
```

```
[4.88621241496954947420190887838e-1,4.88621241496954947420190887839e-1]
encloses a locally unique zero!
[5.55570233019602224742830813948e-1,5.55570233019602224742830813949e-1]
encloses a locally unique zero!
[6.19093949309833986941560856246e-1,6.19093949309833986941560856247e-1]
encloses a locally unique zero!
[6.78800745532941741393855554241e-1,6.78800745532941741393855554242e-1]
encloses a locally unique zero!
[7.34322509435685535636126222186e-1,7.34322509435685535636126222188e-1]
encloses a locally unique zero!
[7.85316930880744927470340278947e-1,7.85316930880744927470340278948e-1]
encloses a locally unique zero!
[8.31469612302545237078788377617e-1,8.31469612302545237078788377618e-1]
encloses a locally unique zero!
[8.72496007072797114525161099222e-1,8.72496007072797114525161099223e-1]
encloses a locally unique zero!
[9.08143173825081299258085836571e-1,9.08143173825081299258085836572e-1]
encloses a locally unique zero!
[9.38191335922484134452339726686e-1,9.38191335922484134452339726687e-1]
encloses a locally unique zero!
[9.62455236453647287630266405185e-1,9.62455236453647287630266405186e-1]
encloses a locally unique zero!
[9.80785280403230449126182236134e-1,9.80785280403230449126182236135e-1]
encloses a locally unique zero!
[9.93068456954926295637437247810e-1,9.93068456954926295637437247811e-1]
encloses a locally unique zero!
[9.99229036240722934737126260341e-1,9.99229036240722934737126260342e-1]
encloses a locally unique zero!

Calculated 40 interval enclosure(s)
Different enclosures do NOT overlap: true
Every enclosure contains a locally unique zero: true
There is/are exactly 40 zero(s) in the interval
  [-1.00000000000000,1.00000000000000]
The zeros of the *** 0-th derivative *** of the basic expression
have been computed.
```

Mit den folgenden Eingabedaten

$$n = 0,\ [-1,+1],\ 53,\ k = 9,\ i = 0,\ 1e-300,\ 1547,\ 30;$$

werden die 40 einfachen Nullstellen mit einer Genauigkeit von 300 Dezimalstellen eingeschlossen. Mit noch höherer relativer Genauigkeit, z.B. `1e-500`, können die Einschließungen jedoch nicht mehr berechnet werden. Will man z.B. die letzte Nullstelle mit noch größerer Genauigkeit, z.B. mit 10000 Dezimalstellen, einschließen, so berechnet man alle Einschließung in $[-1,1]$ zunächst mit 300 Dezimalstellen und wählt dann die Eingabedaten für die letzte Nullstelle:

$$n = 0,\ [9.99229036...2009244248e-1, 9.99229036...2009244248e-1],$$
$$53,\ k = 9,\ i = 0,\ 1e-10000,\ 50000,\ 30;$$

Nach dem neuen Programmstart erhält man dann die gewünschte hohe Genauigkeit. Dies lässt sich (in einer **for**-Schleife) mit jeder anderen Nullstelle analog durchführen. Natürlich wird man bei diesem Beispiel auch wieder an die Grenzen der Funktion `AllZeros(...)` stoßen, wenn nur der Polynomgrad hinreichend erhöht wird. Es bleibt dem Anwender überlassen zu testen, bis zu welcher Genauigkeit die Nullstellen eingeschlossen werden können, wenn man an Stelle der Polynomdarstellung die alternative Definition $T_{40}(x) := \cos(40 \cdot \arccos(x)))$ wählt. Zum Testen der Leistungsfähigkeit von `AllZeros()` sollte natürlich immer die Polynomdarstellung gewählt werden.

Abschließend noch einige Empfehlungen zur Anwendung der Funktion `AllZeros(...)`:

Bei einem einfachen Problem verwende man zunächst das Programm `MPFR-15`. Wenn dies scheitert, verwende man eines der Programme `MPFR-21` oder `MPFR-22`, wobei `k` ≥ 1 schrittweise um eins zu vergrößern ist. Achten Sie darauf, nach Möglichkeit den günstigsten Funktionsterm zu wählen.
Scheitert auch dies, so ist das Startintervall so zu unterteilen, dass die Nullstellensuche in jedem Teilintervall erfolgreich ist, dabei sollte die relative Genauigkeit (`Tolerance`) mit etwa `1e-20` zunächst nicht zu groß gewählt werden. Nullstelleneinschließungen mit: *may contain a zero (not verified unique)!* sollten in einem einzelnen Intervall zusammengefasst, und nach Seite 169 unten sollte der Mindestwert der `CurrenPrecision` hinreichend vergrößert werden.
Falls die geforderte Genauigkeit einer Einschließung nach der ersten Anwendung von `AllZeros(..)` noch nicht ausreicht, muss mit dieser Einschließung als neues Startintervall das entsprechende Programm mit der gewünschten größeren Genauigkeit nochmals gestartet werden, vgl. Seite 176.
Mit diesen Empfehlungen können die gängigsten Nullstellenprobleme gelöst werden. Wenn ein Programm ohne Fehlermeldung beendet wird, so werden im Vektor `Zero` garantiert alle existierenden Nullstellen aus dem Startintervall erfasst. Eine Einschließung mit: *may contain a zero (not verified unique)!* kann entweder keine oder eine oder auch mehrere einfache oder mehrfache Nullstellen enthalten.
Bei einer Einschließung mit: `encloses a locally unique zero!` ist die Einschließung selbst mathematisch garantiert und sichergestellt, dass die Einschließung nur eine einzige Nullstelle enhält!

A. Neue Funktionen vom Typ `MpfrClass`

Mit Hilfe der bereits in den beiden MPFR- und MPFI-Bibliotheken implementierten Funktionen aus den Tabellen auf Seite 37 und 44 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 $A_d \le A$ bzw. $A_u \ge A$ werden stets garantiert.

A.1.1. Unitäre Operatoren

Unabhängig davon, ob A positiv, negativ oder gleich Null ist, gilt

$$+(A_d) \le A; \tag{A.1}$$
$$+(A_u) \ge A; \tag{A.2}$$
$$-(A_d) \ge -A; \tag{A.3}$$
$$-(A_u) \le -A; \tag{A.4}$$

A.1.2. Addition

Unabhängig davon, ob A oder B positiv, negativ oder gleich Null ist, gilt

$$A_d \oplus_d B_d \le A + B; \tag{A.5}$$
$$A_u \oplus_u B_u \ge A + B; \tag{A.6}$$

A.1.3. Subtraktion

Unabhängig davon, ob A oder B positiv, negativ oder gleich Null ist, gilt

$$A_d \ominus_d B_u \le A - B; \tag{A.7}$$
$$A_u \ominus_u B_d \ge A - B; \tag{A.8}$$

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:
>
> $$A_u > 0 \implies A_u \ge A \ge 0; \tag{A.9}$$
> $$A_d < 0 \implies A_d \le A \le 0; \tag{A.10}$$

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 eine einzige Funktion aus den Tabellen auf Seite 37 und 44 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 180 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 180 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 184.

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 Funktionswerte $f_u(x),\ f_d(x)$

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 180 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. $x/(x^2+y^2)$

Für $x, y \in \mathbb{R}\backslash\{0\}$ sei $f(x,y)$ definiert durch

$$f(x,y) := \frac{x}{x^2+y^2}, \quad x \neq 0 \wedge y \neq 0. \tag{A.43}$$

Für Präzisionen $\texttt{prec} \geq 2$ erfolgt die Auswertung von $f(x,y)$ auf der Maschine durch

```
MpfrClass x_div_x2py2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

wobei bez. des Rundungsmodus $\texttt{rnd} \in \{\texttt{RoundUp}, \texttt{RoundDown}, \texttt{RoundNearest}\}$ nahezu optimal gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus.

$f(x,y)$ wird mit $x \neq 0 \wedge y \neq 0$ auf der Maschine berechnet durch:

$$f(x,y) = \begin{cases} 0, & x = 0 \\ 1/x, & y = 0 \\ \dfrac{x}{x^2+y^2}, & \text{sonst.} \end{cases} \tag{A.44}$$

Die Auswertung von $1/x$ erfolgt dabei mithilfe der Funktion

```
MpfrClass reci(const MpfrClass& x , RoundingMode rnd);
```

die in `mpfrclass.cpp` bereits definiert ist.

Um bei der Auswertung der Summe x^2+y^2 keinen vorzeitigen Überlauf zu erzeugen, suchen wir eine Konstante c, so dass für $|x| \leq c$ und $|y| \leq c$ auch im ungünstigsten Fall x^2+y^2 auf der Maschine ohne Überlauf berechnet werden kann. Der ungünstigste Fall ist dabei definiert durch $\texttt{prec} = 2$ und durch den eingestellten Rundungsmodus `RoundUp`. Man erhält damit $c := 2^{+536870910}$, und um $|x| = m \cdot 2^{\texttt{ex\_x}} \leq 2^{+536870910}$ mit $m \in [0.5, +1)$ garantieren zu können, verlangen wir daher mit $\texttt{ex\_x} = \texttt{expo(x)}$ die Beziehung $\texttt{ex\_x} \leq +536870910$. Ganz entsprechend verlangen wir $\texttt{ex\_y} \leq c_1 := +536870910$. Nur im Fall $\texttt{ex\_x} > c_1$ oder $\texttt{ex\_y} > c_1$ kann also ein vorzeitiger Überlauf eintreten. Mit $\texttt{ex\_m} := \text{Max}(\texttt{ex\_x}, \texttt{ex\_y})$ wird dieser Überlauf verhindert, wenn im Nenner von $x/(x^2+y^2)$ die Beträge $|x|, |y|$ mit $2^{\texttt{ex\_s}}$ und der Zähler x mit $2^{2\cdot\texttt{ex\_s}}$ skaliert wird, wobei $\texttt{ex\_s} < 0$ definiert ist durch $\texttt{ex\_m} + \texttt{ex\_s} = c_1$.

Um bei der Auswertung von x^2+y^2 keinen vorzeitigen Unterlauf zu erhalten[2], suchen wir z.B. für $|x|$ eine möglichst kleine Oberschranke c, so dass im ungünstigsten Fall für $|x| < c$ die Bedingung $x^2 > m := \texttt{minfloat()}$ erfüllt ist. Der ungünstigste Fall ist dabei definiert durch $\texttt{prec} = 2$ und durch den voreingestellten Rundungsmodus `RoundDown`. Man erhält damit $c := 2^{-536870912}$, so dass aus $|x| \geq c \vee |y| \geq c$ stets folgt $x^2+y^2 > m$, oder anders ausgedrückt:

Nur im Fall $|x| < c \wedge |y| < c$ kann bei der Auswertung von x^2+y^2 ein Unterlauf eintreten.

Mit $|x| = m_x \cdot 2^{\texttt{ex\_x}}$, $|y| = m_y \cdot 2^{\texttt{ex\_y}}$, $m_x, m_y \in [0.5, 1)$, $\texttt{ex\_x} = \texttt{expo}(x)$, $\texttt{ex\_y} = \texttt{expo}(y)$ und mit $c_2 := -536870911$ erhält man die etwas gröbere Formulierung:

Nur für $\texttt{ex\_x} < c_2 \wedge \texttt{ex\_y} < c_2$ kann bei der Auswertung von x^2+y^2 ein Unterlauf eintreten.

Mit $\texttt{ex\_m} := \text{Min}(\texttt{ex\_x}, \texttt{ex\_y})$ wird dann ein solcher vorzeitiger Unterlauf vermieden, wenn man den Zähler x mit $2^{2\cdot\texttt{ex\_s}}$ und im Nenner x, y jeweils mit $2^{\texttt{ex\_s}}$ skaliert, wobei `ex_s` definiert ist durch: $\texttt{ex\_m} + \texttt{ex\_s} = c_2 + 1$, um $|x| > c$ und $|y| > c$ garantieren zu können.

Als Beispiel wählen wir $x = y = m := \texttt{minfloat(prec)} = 2^{-1073741824}$ und erhalten für den exakten Wert $f(m,m) = 1/(2 \cdot m) = 2^{+1073741823} > \texttt{MaxFloat(prec)}$. `x_div_x2py2(..)` liefert für den aufgerundeten Funktionswert das korrekte Ergebnis `+infinity`, und für den optimal abgerundeten Funktionswert erhält man: $\texttt{MaxFloat(prec)} = 2.0985787164673\ldots \cdot 10^{+323228496}$.

[2] Ein vorzeitiger Unterlauf würde bei der Auswertung von $x/(x^2+y^2)$ eine Division durch Null erzeugen.

A.4. $(x^2 - y^2)/(x^2 + y^2)^2$

Für $x, y \in \mathbb{R}\backslash\{0\}$ sei $f(x,y)$ definiert durch

$$f(x,y) := \frac{x^2 - y^2}{(x^2 + y^2)^2}, \quad x \neq 0 \wedge y \neq 0. \tag{A.45}$$

Für Präzisionen `prec` ≥ 2 erfolgt die Auswertung von $f(x,y)$ auf der Maschine durch

```
MpfrClass Re_rz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

wobei bez. des Rundungsmodus `rnd` $\in$ {`RoundUp, RoundDown, RoundNearest`} nahezu optimal gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Die interne Auswertung von $f(x,y)$ erfolgt beim Zähler und Nenner mittels geeigneter Skalierungen ohne einen vorzeitigen Über- oder Unterlauf. Im Fall $x = y = 0$ erfolgt eine entsprechende Fehlermeldung. $f(x,y)$ kommt u.a. zur Anwendung als Realteil der komplexen Funktion $w(z) = 1/z^2$, $z = x + i \cdot y \in \mathbb{C}$.

Der Zähler $Z(x,y) = x^2 - y^2$ wird realisiert mit Hilfe der internen Funktion

```
void Zae_x2my2(MpfrClass& d, long int& k, const MpfrClass& x,
               const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^k \cdot d = x^2 - y^2$, wobei intern bei der Berechnung von d bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|d|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt.

Ganz entsprechend wird der Nenner $N(x,y) = (x^2 + y^2)^2$ realisiert durch die interne Funktion

```
void Ne_x2py2p2(MpfrClass& r, long int& kn, const MpfrClass& x,
                const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^{kn} \cdot r = (x^2 + y^2)^2$, wobei intern bei der Berechnung von r bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|r|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt. Anschließend kann die notwendige Division d/r ohne vorzeitigen Über- oder Unterlauf durchgeführt werden. Nur bei der letzten Multiplikation mit 2^{k-kn} kann dann ein nicht vermeidbarer Über- oder Unterlauf eintreten.

A.4.1. Numerische Beispiele

Im **1. Beispiel** wähle wir $x =$ `MaxFloat()`, $y =$ `MaxFloat()/3`, d.h. $0 < f(x,y) <$ `minfloat()` ist sicher erfüllt. Mit dem Funktionsaufruf `res = Re_rz2 (x, y, RoundDown);` erhalten wir daher `res` $= 0$ und `res = Re_rz2 (x, y, RoundUp);` liefert `res` $=$ `minfloat()`.

Im **2. Beispiel** wähle wir $x =$ `minfloat()`, $y =$ `3*minfloat()`, d.h. $f(x,y) <$ `-MaxFloat()` ist sicher erfüllt. Mit dem Funktionsaufruf `res = Re_rz2 (x, y, RoundDown);` erhalten wir daher `res` $=$ `-@Inf@`, und der Funktionsaufruf `res = Re_rz2 (x, y, RoundUp);` liefert folglich: `res` $= -$`MaxFloat()` $= -2.098\ldots \cdot 10^{323228496}$.

Im **3. Beispiel** wähle wir $x = 1/3$, $y = 2/3$ und erhalten $f(1/3, 2/3) = -27/25 = -1.08000\ldots$. Mit dem Funktionsaufruf `res = Re_rz2 (x, y, RoundDown);` erhalten wir mit der Präzision `prec` $= 300$ das nahezu optimal abgerundete Ergebnis

```
res = -1.08000000000000000...000000000000000388
```

wobei insgesamt 93 Dezimalstellen ausgegeben wurden. Beachten Sie, dass sowohl Zähler als auch Nenner mit den Werten $-1/3$ bzw. $25/81$ jeweils periodische Dezimalbrüche sind, die daher bei der internen Berechnung entsprechend gerundet werden müssen.

A.5. $2xy/(x^2+y^2)^2$

Für $x, y \in \mathbb{R}\backslash\{0\}$ sei $f(x,y)$ definiert durch

$$f(x,y) := \frac{2xy}{(x^2+y^2)^2}, \quad x \neq 0 \wedge y \neq 0. \tag{A.46}$$

Für Präzisionen `prec` ≥ 2 erfolgt die Auswertung von $f(x,y)$ auf der Maschine durch

```
MpfrClass mIm_rz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

wobei bez. des Rundungsmodus `rnd` $\in$ {`RoundUp, RoundDown, RoundNearest`} nahezu optimal gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Die interne Auswertung von $f(x,y)$ erfolgt beim Zähler und Nenner mittels geeigneter Skalierungen ohne einen vorzeitigen Über- oder Unterlauf. Im Fall $x = y = 0$ erfolgt eine entsprechende Fehlermeldung. $-f(x,y)$ kommt zur Anwendung als Imaginärteil der komplexen Funktion $w(z) = 1/z^2,\ z = x + i \cdot y \in \mathbb{C}$.

Der Zähler $Z(x,y) = 2xy$ wird realisiert mit Hilfe der internen Funktion

```
void Zae_2xy(MpfrClass& z, long int& k, const MpfrClass& x,
             const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^k \cdot z = 2 \cdot (xy)$, wobei intern bei der Berechnung von z bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|z|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt.

Ganz entsprechend wird der Nenner $N(x,y) = (x^2+y^2)^2$ realisiert durch die interne Funktion

```
void Ne_x2py2p2(MpfrClass& r, long int& kn, const MpfrClass& x,
                const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^{kn} \cdot r = (x^2+y^2)^2$, wobei intern bei der Berechnung von r bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|r|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt. Anschließend kann die notwendige Division z/r ohne vorzeitigen Über- oder Unterlauf durchgeführt werden. Nur bei der letzten Multiplikation mit 2^{k-kn} kann dann ein nicht vermeidbarer Über- oder Unterlauf eintreten.

Der Ausdruck $-2xy/(x^2+y^2)^2$ ist der Imaginärteil von $1/z^2,\ z = x + i \cdot y$, und wird analog realisiert durch die Funktion `Im_rz2(x,y,rnd)`, die in `mpfrclass.cpp` definiert ist.

A.5.1. Numerische Beispiele

Im **1. Beispiel** wähle wir $x =$ `MaxFloat()`, $y =$ `MaxFloat()/3`, d.h. $0 < f(x,y) <$ `+minfloat()` ist sicher erfüllt. Mit dem Funktionsaufruf `res = mIm_rz2 (x, y, RoundUp);` erhalten wir daher `res` = `+minfloat()` und `res = mIm_rz2(x,y,RounDown);` liefert `res` = 0.

Im **2. Beispiel** wähle wir $x =$ `-minfloat()`, $y =$ `3*minfloat()`, d.h. $f(x,y) <$ `-MaxFloat()` ist sicher erfüllt. Mit dem ersten Funktionsaufruf `res = mIm_rz2 (x, y, RoundUp);` erhalten wir folgerichtig `res` = `-MaxFloat()` $= -2.098\ldots \cdot 10^{323228496}$, und der nächste Funktionsaufruf `res = mIm_rz2 (x, y, RoundDown);` liefert wie erwartet: `res` = `-@Inf@`.

Im **3. Beispiel** wähle wir $x = 2,\ y = 3$ und erhalten $f(2,3) = 12/169 = 7.10059116\ldots \cdot 10^{-2}$. Mit dem Funktionsaufruf `res = mIm_rz2 (x, y, RoundDown);` erhalten wir mit der Präzision `prec` = 300 das nahezu optimal abgerundete Ergebnis

$$\texttt{res} = 7.10059171597633136094674556213 0\ldots 171597379 \cdot 10^{-2}$$

wobei insgesamt 93 Dezimalstellen ausgegeben wurden.

A.6. $2xy/(4x^2y^2 + (1 + x^2 - y^2)^2)$

Für $(x, y) \in \mathbb{R}^2 \backslash \{(0, \pm 1)\}$ sei $f(x, y)$ definiert durch

$$f(x,y) := \frac{2xy}{4x^2y^2 + (1 + x^2 - y^2)^2}, \quad (x,y) \neq (0, \pm 1). \tag{A.47}$$

Für Präzisionen `prec` ≥ 2 erfolgt die Auswertung von $f(x, y)$ auf der Maschine durch

```
MpfrClass Im_r1pz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

wobei bez. des Rundungsmodus `rnd` $\in$ {`RoundUp, RoundDown, RoundNearest`} nahezu optimal gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Die interne Auswertung von $f(x, y)$ erfolgt beim Zähler und Nenner mittels geeigneter Skalierungen ohne einen vorzeitigen Über- oder Unterlauf. $(x, y) = (0, \pm 1)$ liefert eine entsprechende Fehlermeldung. $-f(x, y)$ kommt zur Anwendung als Imaginärteil der komplexen Funktion $w(z) = 1/(1 + z^2)$, $z = x + i \cdot y \in \mathbb{C}$.

Der Zähler $Z(x, y) = 2xy$ wird realisiert mit Hilfe der internen Funktion

```
void Zae_2xy(MpfrClass& z, long int& k, const MpfrClass& x,
             const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^k \cdot z = 2 \cdot (xy)$, wobei intern bei der Berechnung von z bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|z|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt.

Ganz entsprechend wird der Nenner $N(x, y) = 4x^2y^2(1 + x^2 - y^2)^2$ realisiert durch die interne Funktion

```
void Ne_r1px2(MpfrClass& r, long int& kn, const MpfrClass& x,
              const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^{kn} \cdot r = 4x^2y^2(1 + x^2 - y^2)^2$, wobei intern bei der Berechnung von r bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|r|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt. Anschließend kann die notwendige Division z/r ohne vorzeitigen Über- oder Unterlauf durchgeführt werden. Erst bei der letzten Multiplikation mit 2^{k-kn} kann dann ein nicht vermeidbarer Über- oder Unterlauf eintreten.

A.6.1. Numerische Beispiele

Im **1. Beispiel** wähle wir $x =$ `MaxFloat()`, $y =$ `MaxFloat()`, d.h. $0 < f(x, y) <$ `+minfloat()` ist sicher erfüllt. Mit dem Funktionsaufruf `res = Im_r1pz2(x, y, RoundUp);` erhalten wir daher `res` = `+minfloat()` und `res = Im_r1pz2(x,y,RounDown);` liefert `res` = 0.

Im **2. Beispiel** wähle wir $x =$ `-minfloat()`, $y =$ `minfloat()`, d.h. `-minfloat()` $< f(x, y) < 0$ ist sicher erfüllt. Mit dem Funktionsaufruf `res = Im_r1pz2 (x, y, RoundDown)` erhalten wir folgerichtig `res` = `-minfloat()` $= -2.382564\ldots \cdot 10^{-323228497}$, und der nächste Funktionsaufruf `res = Im_r1pz2(x,y,RoundUp)` liefert wie erwartet: `res` = 0.

Im **3. Beispiel** wähle wir $x = 2$, $y = -3$ und erhalten $f(2, -3) = -3/40 = -7.5 \cdot 10^{-2}$. Mit dem Funktionsaufruf `res = Im_r1pz2(x,y,RoundUp);` erhalten wir mit `prec` = 300 das nahezu optimal aufgerundete Ergebnis

$$\texttt{res} = -7.49999999999999999999999999999999\ldots 99999631 \cdot 10^{-2}$$

wobei insgesamt 93 Dezimalstellen ausgegeben wurden.

A.7. $(1 + x^2 - y^2)/(4x^2y^2 + (1 + x^2 - y^2)^2)$

$f : \mathbb{R}^2 \backslash \{(0, \pm 1)\} \to \mathbb{R}$ sei definiert durch

$$f(x, y) := \frac{1 + x^2 - y^2}{4x^2y^2 + (1 + x^2 - y^2)^2}$$

wobei man sich bei der Auswertung auf den 1. Quadranten beschränken kann, d.h. wir betrachten nur die Funktion

$$f(x, y) := \frac{1 + x^2 - y^2}{4x^2y^2 + (1 + x^2 - y^2)^2} \qquad \text{mit } x \geq 0, y \geq 0 \text{ und } (x, y) \neq (0, +1), \tag{A.48}$$

und mit $\varepsilon > 0$ gilt zusätzlich: $\lim_{\varepsilon \to 0} f(0, 1 \pm \varepsilon) = \mp\infty, \quad \lim_{\varepsilon \to 0} f(\varepsilon, +1) = 1/4.$

Für die Präzisionen `prec` ≥ 2 erfolgt die Auswertung von $f(x, y)$ auf der Maschine durch

```
MpfrClass Re_r1pz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

wobei bez. des Rundungsmodus `rnd` $\in$ {`RoundUp, RoundDown, RoundNearest`} nahezu optimal gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Die interne Auswertung von $f(x, y)$ erfolgt beim Zähler und Nenner mittels geeigneter Skalierungen ohne einen vorzeitigen Über- oder Unterlauf. Im Fall $x = 0, y = +1$ erfolgt eine entsprechende Fehlermeldung. $f(x, y)$ kommt u.a. zur Anwendung als Realteil der komplexen Funktion $w(z) = 1/(1 + z^2),\ z = x + i \cdot y \in \mathbb{C}$.

Der Zähler $Z(x, y) := 1 + x^2 - y^2$ wird realisiert mit Hilfe der internen Funktion

```
void x2p1my2(MpfrClass& d, long int& k, const MpfrClass& x,
             const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^k \cdot d = 1 + x^2 - y^2$, wobei intern bei der Berechnung von d bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|d|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt. Die interne Berechnung von d erfolgt in zwei Schritten.

Im Fall $x \approx y$ wird $x^2 - y^2$ über sein Produkt $(x - y) \cdot (x + y)$ ausgewertet, da jetzt wegen der exakten Maschinenwerte x, y ihre Differenz $(x - y)$ **ohne** Auslöschung, d.h. bis auf den normalen Rundungsfehler **ohne** Informationsverlust, berechnet wird. Anschließend erfolgt die Addition der Eins, wobei im Fall $x^2 - y^2 \approx -1$ bzw. $y \approx \sqrt{1 + x^2}$ eine jetzt wirkliche Auslöschung mit Informationsverlust durch die schrittweise Auswertung mit wachsender Präzision garantiert vermieden wird. Beachten Sie, dass selbst bei einer auftretenden Auslöschung bez. der mit `rnd` gewählten Rundungen **stets korrekt** auf- bzw. abgerundet wird!

Falls sich x, y deutlicher unterscheiden, wird $1 - y^2$ über sein Produkt $(1 - y) \cdot (1 + y)$ ausgewertet und danach erfolgt die Addition von x^2, wobei im Fall $y \approx \sqrt{1 + x^2}$ eine Auslöschung mit Informationsverlust durch die schrittweise Auswertung mit wachsender Präzision garantiert vermieden wird.

Ganz analog wird der Nenner $N(x, y) := 4x^2y^2 + (1 + x^2 - y^2)^2$ realisiert durch die interne Funktion

```
void Ne_r1px2(MpfrClass& r, long int& kn, const MpfrClass& x,
              const MpfrClass& y, RoundingMode rnd);
```

dabei gilt exakt, d.h. ohne Rundung: $2^{kn} \cdot r = 4x^2y^2 + (1 + x^2 - y^2)^2$, wobei intern bei der Berechnung von r bezüglich `rnd` gerundet wird. Mit geeigneter Skalierung erreicht man, dass $|r|$ nur wenig kleiner als `MaxFloat(prec)` ausfällt. Anschließend kann die notwendige Division d/r ohne vorzeitigen Über- oder Unterlauf durchgeführt werden. Nur bei der letzten Multiplikation mit 2^{k-kn} kann dann ein nicht vermeidbarer Über- oder Unterlauf eintreten. Beachten Sie, dass in $N(x, y)$ beide Summanden nicht negativ sind, so dass eine Auslöschung nicht auftreten kann.

A.7.1. Numerische Beispiele

Im **1. Beispiel** wähle wir $y = 0$ und erhalten $f(x,0) = 1/(1+x^2) > 0$. Mit $x =$ `MaxFloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{MaxFloat}(),0) = 0$$
$$f(\texttt{MaxFloat}(),0) = 2.38256490488795107321616978173267452041\ldots513701\cdot 10^{-323228497},$$

wobei $300\cdot\log_{10}(2) \approx 90$ Dezimalstellen ausgegeben wurden.

Im **2. Beispiel** wähle wir $y = 0$ und erhalten $f(x,0) = 1/(1+x^2) > 0$. Mit $x =$ `minfloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{minfloat}(),0) = 9.99999999999999999999999999999999999999\ldots99999999999999998\cdot 10^{-1}$$
$$f(\texttt{minfloat}(),0) = 1.00000000000000000000000000000000000000\ldots00000000000000001,$$

wobei $300\cdot\log_{10}(2) \approx 90$ Dezimalstellen ausgegeben wurden.

Im **3. Beispiel** wähle wir $y = 1$ und erhalten $f(x,+1) = 1/(4+x^2) > 0,\ x > 0$. Mit $x =$ `minfloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{minfloat}(),0) = 2.49999999999999999999999999999999999999\ldots9999999999999\cdot 10^{-1}$$
$$f(\texttt{minfloat}(),0) = 2.50000000000000000000000000000000000000\ldots0000000000000\cdot 10^{-1},$$

wobei $300\cdot\log_{10}(2) + 1 \approx 91$ Dezimalstellen ausgegeben wurden.

Im **4. Beispiel** wähle wir $y = 1$ und erhalten $f(x,+1) = 1/(4+x^2) > 0,\ x > 0$. Mit $x =$ `MaxFloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{MaxFloat}(),0) = 0$$
$$f(\texttt{MaxFloat}(),0) = 2.38256490488795107321616978173267452041\ldots513701\cdot 10^{-323228497},$$

wobei $300\cdot\log_{10}(2) \approx 90$ Dezimalstellen ausgegeben wurden.

Im **5. Beispiel** wähle wir $x = 0$ und erhalten $f(0,y) = 1/(1-y^2),\ y \neq 1$. Mit $y =$ `minfloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{minfloat}(),0) = 9.99999999999999999999999999999999999999\ldots99999999999999998\cdot 10^{-1}$$
$$f(\texttt{minfloat}(),0) = 1.00000000000000000000000000000000000000\ldots00000000000000001,$$

wobei $300\cdot\log_{10}(2) \approx 90$ Dezimalstellen ausgegeben wurden.

Im **6. Beispiel** wähle wir $x = 0$ und erhalten $f(0,y) = 1/(1-y^2),\ y \neq 1$. Mit $y =$ `MaxFloat()` und `prec` $= 300$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{MaxFloat}(),0) = -2.38256490488795107321616978173267452041\ldots513701\cdot 10^{-323228497},$$
$$f(\texttt{MaxFloat}(),0) = 0,$$

wobei $300\cdot\log_{10}(2) \approx 90$ Dezimalstellen ausgegeben wurden.

Im **7. Beispiel** wähle wir $x = 0$ und erhalten $f(0,y) = 1/(1-y^2),\ y \neq 1$. Mit `prec` $= 300$ und $y =$ `pred`$(1) = 1 - 2^{-300}$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert

$$f(\texttt{MaxFloat}(),0) = 1.01851798816724304313422284420468908052573419\ldots1698688\cdot 10^{90},$$
$$f(\texttt{MaxFloat}(),0) = 1.01851798816724304313422284420468908052573419\ldots1698690\cdot 10^{90},$$

wobei $300\cdot\log_{10}(2) + 1 \approx 91$ Dezimalstellen ausgegeben wurden.

A.8. $x^2 + a \cdot x + b$

$f : \mathbb{R} \to \mathbb{R}$ sei definiert durch

$$f(x,a,b) := x^2 + a \cdot x + b; \quad x,a,b \in \mathbb{R}.$$

Für die Präzisionen `prec` ≥ 2 erfolgt die Auswertung von $f(x,a,b)$ auf der Maschine zunächst durch die Funktion

```
void poly2(const MpfrClass& r, lomg int& k, const MpfrClass& x,
  const MpfrClass& a, const MpfrClass& b, RoundingMode rnd);
```

Dabei gilt ohne Berücksichtigung einer Rundung $2^k \cdot r = x^2 + a \cdot x + b$, wobei aber r bei der internen Berechnung bez. des Parameters `rnd` $\in$ {`RoundUp, RoundDown, RoundNearest`} **optimal** gerundet wird. Ohne den obigen Parameter `rnd` erfolgt die Rundung nach dem voreingestellten Current-Rundungsmodus. Durch geeignete Skalierungen wird $r \neq 0$ so berechnet, dass $|r|$ nur etwas kleiner als `MaxFloat()` ausfällt, wobei ein vorzeitiger Überlauf oder Unterlauf vermieden wird. Im Fall $r = 0$ gilt $k = 0$. Die optimale Rundung wird durch eine schrittweise Vergrößerung der internen Präzision erreicht. Im letzten Schritt wird dann in der Funktion

```
MpfrClass poly2(const MpfrClass& x, const MpfrClass& a,
        const MpfrClass& b, RoundingMode rnd);
```

das bez. `rnd` gerundete Produkt $2^k \cdot r$ ausgewertet, wobei erst dabei ein dann unvermeidbarer Überlauf oder Unterlauf eintreten kann.

A.8.1. Numerische Beispiele

Zunächst ein allgemeiner **Hinweis**: Bei der Funktion $f(x,a,b)$, wie auch bei allen Funktionen aus Tabelle 3.11.3, wird vorausgesetzt, dass x,a,b exakte Maschinenzahlen sind, die also nicht schon durch vorhergehende Rechnungen gerundet worden sind. Beispielsweise kann daher eine Maschinenzahl a nicht den Wert $\sqrt{3}$ annehmen, wohl aber eine geeignete Maschinennäherung, die dann aber als **exakt** anzusehen ist.

Im **1. Beispiel** wählen wir $x = 10,\ a = 3,\ b = -3$ und erhalten mit z.B. `prec` $= 300$ für das auf- und abgerundete Ergebnis jeweils den gleichen, exakten Wert $f(x,a,b) = 127$.

Im **2. Beispiel** wählen wir $a = 0,\ b = -1$ und werten mit der Präzision `prec` $= 300$ den Ausdruck $f(x,a,b) = x^2 - 1$ in der unmittelbaren Nähe seiner Nullstelle $x_0 = +1$ für $x =$ `pred`$(1) = 1 - 2^{-300}$ aus, so dass bei der internen Auswertung von

$$f(\texttt{pred}(1),0,-1) = 2^{-300} \cdot (-2 + 2^{-300}) = -9.818186930595453106\ldots 2019576566474\ldots \cdot 10^{-91}$$

maximale Auslöschung auftreten muss. Mit der Funktion `poly2(x,a,b,rnd)` erhält man für den abgerundeten und aufgerundeten Maschinenwert das Ergebnis

$$\texttt{poly2(x,a,b,rnd)} = -9.818186930595453106\ldots 2019576\overset{325}{\underset{807}{}}566474\ldots \cdot 10^{-91},$$

wobei die gerundeten Werte den exakten Wert einschließen und in den ersten 90 Dezimalziffern übereinstimmen. Wegen $300 \cdot \log_{10}(2) = 90.308998\ldots$ erhalten wir daher trotz maximaler Auslöschung mit 90 die **Maximalzahl** übereinstimmender Dezimalziffern.

A.9. $2\sin(x)\cosh(y)/(\cosh(2y)-\cos(2x))$

Die reelle Funktion $f:\mathbb{R}^2\to\mathbb{R}$, mit

$$f(x,y) := \frac{2\sin(x)\cdot\cosh(y)}{\cosh(2y)-\cos(2x)} = \frac{\sin(x)\cdot\cosh(y)}{\sin^2(x)+\sinh^2(y)} \tag{A.49}$$

ist der Realteil der komplexen Funktion $1/\sin(z)$, $z = x+i\cdot y\in\mathbb{C}$. Die Funktionswerte $f(x,y)$ werden in nahezu allen Fällen optimal approximiert durch

`MpfrClass Re_csc(const MpfrClass& x, const MpfrClass& y, const RoundingMode rnd)`,

wobei für den Rundungsparameter `rnd` die folgenden Rundungsmodi zur Verfügung stehen:

`rnd ∈ {RoundUp, RoundDown, RoundNearest}`.

$f(x,y)$ besitzt Singularitäten an den Stellen $S_k = (x_k,0) := (k\pi,0)$, $k\in\mathbb{Z}$. Parallel zur y-Achse im Abstand x_k verschwinden für $y\neq 0$ alle Funktionswerte $f(x_k,y)=0$, wobei die Funktion an den Stellen $S_k=(x_k,0)$ nicht definiert ist. Auf der x-Achse selbst besitzt $f(x,0)$ an den Stellen x_k Polstellen mit Vorzeichenwechsel, d.h. für $x\to 0$ gilt beispielsweise

$$\lim_{x\to 0-} f(x,0) = \lim_{x\to 0-}\frac{1}{\sin(x)} = -\infty, \qquad \lim_{x\to 0+} f(x,0) = \lim_{x\to 0+}\frac{1}{\sin(x)} = +\infty.$$

Bezüglich der Variablen x ist $f(x,y)$ 2π-periodisch, d.h. $f(x+k\cdot 2\pi,y)=f(x,y)$, $k\in\mathbb{Z}$. Wegen $f(x,y)\equiv f(x,|y|)$ kann man sich auf die obere Halbebene, d.h. $y\geq 0$, beschränken.

Die interne Auswertung erfolgt intervallmäßig mithilfe von Punktintervallen $x\in[x], y\in[y]$ in hinreichend hoher Präzision. Ist beispielsweise `Fxy` eine optimale Einschließung für $f(x,y)$, so ist z.B. der aufgerundete Funktionswert gegeben durch `Sup(Fxy)` $\geq f(x,y)$, wobei x,y jetzt als Maschinenzahlen zu verstehen sind.

A.9.1. Optimale Einschließung

Für eine optimale Einschließung von $f(x,y)$ muss verhindert werden, dass bei intervallmäßiger Auswertung von (A.49) im Zähler und Nenner ein vorzeitiger Unter- oder Überlauf entsteht. Um bei $\cosh(2y)$ oder $\sinh^2(y)$ einen Überlauf zu vermeiden, betrachten wir zunächst den

Fall 1. $0\leq y\leq y_0 := 372130555$, wodurch ein solcher Überlauf verhindert wird. Ausgewertet wird der zweite Ausdruck in (A.49), weil sein Nenner ohne Auslöschung berechnet werden kann. Bei der intervallmäßigen Auswertung von $\sin([x])$, $[x]=[\text{minfloat}()]$, entsteht ein vorzeitiger Unterlauf, der vermieden wird, wenn zunächst mit 2 erweitert wird:

$$f(x,y) = \frac{2\cdot\sin(x)\cdot\cosh(y)}{2\cdot(\sin^2(x)+\sinh^2(y))} \tag{A.50}$$

und im Zähler im Fall $|x|\leq 2\cdot\text{minfloat}()$ der Term $2\cdot\sin(x)$ ersetzt wird durch

$$2\cdot\sin(x) = \frac{\sin(2x)}{\cos(x)},$$

wobei $\sin(2x)$ jetzt auch für $[x]=[\text{minfloat}()]$ ohne Unterlauf berechnet werden kann. $1/\cos(x)$ lässt sich mithilfe der geometrischen Reihe abschätzen durch

$$1\leq\frac{1}{\cos(x)} = 1+\frac{x^2}{2}+\frac{5x^4}{24}\ldots\leq\frac{1}{1-x^2}\leq 1+2x^2, \quad |x|\leq 2\cdot\text{minfloat}(),$$

und wir verlangen zusätzlich: $1+2x^2\leq\text{succ}(1)=1+2^{1-\texttt{prec}} \iff x^2\leq 2^{-\texttt{prec}}$.
Mit $\texttt{minfloat()}=2^{-1073741824}$ und $x^2\leq 4\cdot\texttt{minfloat()}^2$ lautet die letzte Bedingung:

$2^{-2147483646} \le 2^{-\texttt{prec}}$ bzw. $\texttt{prec} \le +2147483646$, und diese Bedingung wird in der Praxis schon aus Laufzeitgründen stets erfüllt sein! $1/\cos(x)$ kann damit sehr einfach eingeschlossen werden durch:

$$1 \le \frac{1}{\cos(x)} \le \mathrm{succ}(1), \quad \text{falls} \quad |x| \le 2 \cdot \mathrm{minfloat}(). \tag{A.51}$$

Der Zähler in (A.50) kann daher effektiv ohne Über- und Unterlauf eingeschlossen werden durch

$$2 \cdot \sin(x) \cdot \cosh(y) \in \sin(2 \diamond [x]) \diamond [1, \texttt{succ(1)}] \diamond \cosh([y]).$$

Um in (A.50) den Nenner $2 \cdot (\sin^2(x) + \sinh^2(y)) \ge 2 \cdot \sinh^2(y) \ge 2y^2$ ohne vorzeitigen Unterlauf berechnen zu können, verlangen wir zunächst $2y^2 \ge 32 \cdot \texttt{minfloat()} = 2^{-1073741819}$, und diese Bedingung[3] ist für $y \ge 2^{-536870910}$ erfüllt. Nur im Fall $y < 2^{-536870910}$ bzw. $\texttt{expo}(y) < -536870909$ kann daher bei intervallmäßiger Auswertung die Einschließung des Nenners die Null enthalten, wodurch bei der nachfolgenden Intervalldivision starke Überschätzungen auftreten[4]. Um dies zu vermeiden, werden Zähler und Nenner im Fall $\texttt{expo}(y) < -536870909$ skaliert, d.h. Zähler und Nenner erhalten die Form $m_{1,2} \cdot 2^{k_{1,2}}$, wobei die schmalen Intervalle $m_{1,2}$ in der Nähe von ± 1 liegen und die Division dieser Intervalle ohne Unter- oder Überlauf möglich ist. Erst bei der Multiplikation mit der Zweierpotenz $2^{k_1 - k_2}$ kann dann ein unvermeidbarer Unter- oder Überlauf auftreten.

Beachten Sie, dass wegen der Taylorreihe $\sinh(y) = y + y^3/3! + y^5/5! + \ldots$ ein Unterlauf bei der Berechnung von $\sinh([\mathrm{minfloat}()])$ nicht auftreten kann, aber $2 \cdot \sin^2(x)$ durch

$$2 \cdot \sin^2(x) = \frac{\sin^2(2x)}{2 \cdot \cos^2(x)} = \frac{1}{2} \cdot \left(\frac{\sin(2x)}{\cos(x)} \right)^2$$

zu ersetzen ist, um $\sin(2[\mathrm{minfloat}()])$ intervallmäßig ohne Unterlauf berechnen zu können. Das Quadrieren erfolgt mit skalierten Intervallen für $\sin(2[x])$ und $\sec([x]) = 1/\cos([x])$, um einen möglichen Unterlauf zu vermeiden. Im Fall $2^{-536870910} \le y \le 372130555$ kann der rechte Term in (A.49) direkt ohne Unter- oder Überlauf intervallmäßig ausgewertet werden. Weitere Einzelheiten findet man im Quellcode der Funktion `Re_csc(x,y,rnd)` in `mpfrclass.cpp`. Es bleibt jetzt noch der

Fall 2. $y > y_0 = 372130555$, in dem ein Überlauf bei der Auswertung von $\cosh(2y)$ in (A.49) zu vermeiden ist. Wir schreiben (A.49) zunächst in der Form

$$f(x,y) = 2 \cdot \frac{\cosh(y)}{\cosh(2y)} \cdot \frac{\sin(x)}{1 - \dfrac{\cos(2x)}{\cosh(2y)}}, \quad \text{mit} \quad \frac{\cosh(y)}{\cosh(2y)} = e^{-y} \cdot \frac{1 + e^{-2y}}{1 + e^{-4y}}, \quad \text{und} \tag{A.52}$$

$$1 < \frac{1 + e^{-2y}}{1 + e^{-4y}} < 1 + e^{-2y}, \text{ d.h.} \quad \frac{\cosh(y)}{\cosh(2y)} \in e^{-y} \cdot \left[1, 1 + e^{-2y}\right]. \tag{A.53}$$

Mit $1/\cosh(2y) = 2/(e^{2y} + e^{-2y})$ folgt weiter

$$1 - \frac{\cos(2x)}{\cosh(2y)} \in 1 - \frac{[-1,+1]}{\cosh(2y)} = 1 - \frac{[-2,+2]}{e^{2y} + e^{-2y}} = \left[1 - \frac{2}{e^{2y} + e^{-2y}}, 1 + \frac{2}{e^{2y} + e^{-2y}}\right] \tag{A.54}$$

$$\subset [1 - 2e^{-2y}, 1 + 2e^{-2y}]. \tag{A.55}$$

Mit diesen Abschätzungen erhält man schließlich nach einfachen Rechnungen:

$$\frac{\cosh(y)}{\cosh(2y) \cdot \left(1 - \dfrac{\cos(2x)}{\cosh(2y)}\right)} \in \frac{e^{-y} \cdot [1, 1 + e^{-2y}]}{[1 - 2e^{-2y}, 1 + 2e^{-2y}]} = e^{-y} \cdot \left[\frac{1}{1 + 2e^{-2y}}, \frac{1 + e^{-2y}}{1 - 2e^{-2y}}\right]$$

$$\subset e^{-y} \cdot [1 - 2e^{-2y}, 1 + 4e^{-2y}].$$

[3] Der Faktor 32 berücksichtigt mögliche Überschätzungen bei der intervallmäßigen Auswertung des Nenners.

[4] Im Fall $x = y = 0$ ist die Division durch Null natürlich unvermeidbar, aber `Re_csc(0,0,rnd)` erzeugt an dieser Singularität eine entsprechende Fehlermeldung.

Zur Vereinfachung verlangen wir noch zusätzlich

$$[1-2e^{-2y}, 1+4e^{-2y}] \subseteq [\texttt{pred}(1), \texttt{succ}(1)] = [1-2^{-\texttt{prec}}, 1+2^{1-\texttt{prec}}], \quad \text{d.h. wegen } y \geq y_0$$
$$1-2^{-\texttt{prec}} < 1-2\cdot e^{-2y_0} \quad \text{und} \quad 1+4\cdot e^{-2y_0} < 1+2^{1-\texttt{prec}}. \tag{A.56}$$

Die erste Ungleichung in (A.56) ist erfüllt für $\texttt{prec} < 2y_0/\ln(2) - 1 = 1073741811,523\ldots$, und diese Bedingung wird in der Praxis schon aus Laufzeitgründen immer erfüllt sein. Es bleibt dem Leser überlassen, dass auch die zweite Ungleichung in (A.56) mit der gleichen Bedingung erfüllt wird. $f(x,y)$ kann damit durch folgenden Intervallausdruck eingeschlossen werden:

$$f(x,y) \in \left([\texttt{pred}(1), \texttt{succ}(1)] \diamond e^{-[y]+\ln(2)}\right) \diamond \sin([x]), \quad y \geq y_0 = 372130555.$$

Beachten Sie dabei, dass jetzt mit $x = \text{minfloat}()$ der Unterlauf bei $\sin(x) \in [0, \text{minfloat}()]$ kein Unglück ist, da das Infimum des obigen Intervallausdrucks $(\ldots)$ sicher kleiner 1 ist, so dass das Infimum des ganzen Intervallausdrucks 0 sein muss. Ganz Entsprechendes gilt auch im Fall $x = -\text{minfloat}()$.

A.9.2. Numerische Beispiele

Im **1. Beispiel** wählen wir mit $x_1 = \texttt{minfloat()} = 2^{-1073741824}$ und $y_1 = 0$ einen Punkt ganz in der Nähe der Singularität $S_0 = (0,0)$ und erhalten mit $\texttt{prec} = 200$ für den optimal ab- und aufgerundeten Funktionswert

```
f(x1,y1,RoundDown) = 2.0985787164673876924043581168 8...335262778664e323228496
f(x1,y1,RoundUp)   = @Inf@
```

Im **2. Beispiel** wählen wir $x_2 = 3\cdot\texttt{minfloat()} = 3\cdot 2^{-1073741824}$ und $y_2 = 0$ und erhalten mit $\texttt{prec} = 200$ für den optimal ab- und aufgerundeten Funktionswert

```
f(x2,y2,RoundDown) = 1.3990524776449251282695720779 22...890175185776e323228496
f(x2,y2,RoundUp)   = 1.3990524776449251282695720779 22...890175185777e323228496
```

Im **3. Beispiel** wählen wir $x_3 = \texttt{minfloat()} = 2^{-1073741824}$ und $y_3 = \texttt{Maxfloat()}$ und erhalten mit $\texttt{prec} = 200$ für den optimal ab- und aufgerundeten Funktionswert

```
f(x3,y3,RoundDown) = 0
f(x3,y3,RoundUp)   = 2.3825649048879510732161697817 32...23978795503e-323228497
```

Im **4. Beispiel** wählen wir $x_4 = 2$ und $y_4 = 0.25$ und erhalten mit $\texttt{prec} = 200$ für den optimal ab- und aufgerundeten Funktionswert

```
f(x4,y4,RoundDown) = 1.0530256453589318310512458485 994...69109247482859536177
f(x4,y4,RoundUp)   = 1.0530256453589318310512458485 994...69109247482859536178
```

Im **5. Beispiel** wählen wir $x_5 = -2$ und $y_5 = 372130600$ und erhalten mit $\texttt{prec} = 200$ für den optimal ab- und aufgerundeten Funktionswert

```
f(x5,y5,RoundDown) = -1.3564060343501086287177564349 78...813293104e-161614266
f(x5,y5,RoundUp)   = -1.3564060343501086287177564349 78...813293103e-161614266
```

A.10. $2\cos(x)\sinh(y)/(\cos(2x)-\cosh(2y))$

Die reelle Funktion $f:\mathbb{R}^2\to\mathbb{R}$, mit

$$f(x,y) := \frac{2\cos(x)\cdot\sinh(y)}{\cos(2x)-\cosh(2y)} = \frac{-\cos(x)\cdot\sinh(y)}{\sin^2(x)+\sinh^2(y)} \tag{A.57}$$

ist der Imaginärteil der komplexen Funktion $1/\sin(z)$, $z = x + i\cdot y \in \mathbb{C}$. $f(x,y)$ wird in nahezu allen Fällen optimal approximiert durch

`MpfrClass Im_csc(const MpfrClass& x, const MpfrClass& y, const RoundingMode rnd)`,

wobei für den Rundungsparameter `rnd` die folgenden Rundungsmodi zur Verfügung stehen:

`rnd ∈ {RoundUp, RoundDown, RoundNearest}`.

$f(x,y)$ besitzt Singularitäten an den Stellen $S_k = (x_k,0) := (k\pi,0)$, $k\in\mathbb{Z}$. Parallel zur y-Achse im Abstand x_k gilt $f(x_k,y) = (-1)^{k+1}/\sinh(y)$, d.h. es liegen Polstellen mit Vorzeichenwechsel vor. Wegen $f(x,0) = 0/\sin^2(x)$ verschwinden die Funktionswerte auf der x-Achse für $x\neq x_k$. Bezüglich der Variablen x ist $f(x,y)$ 2π-periodisch, d.h. $f(x+k\cdot 2\pi,y) = f(x,y)$, $k\in\mathbb{Z}$. Außerdem gilt $f(x,-y) = -f(x,y)$. Die interne Auswertung erfolgt intervallmäßig mithilfe von Punktintervallen $x\in[x], y\in[y]$ in hinreichend hoher Präzision. Ist beispielsweise `Fxy` eine optimale Einschließung für $f(x,y)$, so ist z.B. der aufgerundete Funktionswert gegeben durch `Sup(Fxy)` $\geq f(x,y)$, wobei x,y jetzt als Maschinenzahlen zu verstehen sind.

A.10.1. Optimale Einschließung

Für eine optimale Einschließung von $f(x,y)$ muss verhindert werden, dass bei intervallmäßiger Auswertung von (A.57) im Zähler und Nenner ein vorzeitiger Unter- oder Überlauf entsteht. Um bei $\sinh^2(y)$ einen Überlauf zu vermeiden, betrachten wir zunächst den

Fall 1. $0\leq|y|\leq y_0 := 372130555$, wodurch ein solcher Überlauf verhindert wird. Ausgewertet wird der zweite Ausdruck in (A.57), weil sein Nenner ohne Auslöschung berechnet werden kann.

Um bei der Auswertung von (A.57) einen vorzeitigen Über- oder Unterlauf zu vermeiden, werden Zähler und Nenner skaliert, d.h. Zähler und Nenner erhalten die Form $m_{1,2}\cdot 2^{k_{1,2}}$, wobei die schmalen Intervalle $m_{1,2}$ in der Nähe von ±1 liegen und die Division dieser Intervalle ohne Unter- oder Überlauf möglich ist. Erst bei der Multiplikation mit der Zweierpotenz $2^{k_1-k_2}$ kann dann ein unvermeidbarer Unter- oder Überlauf auftreten.

Beachten Sie, dass wegen der Taylorreihe $\sinh(y) = y + y^3/3! + y^5/5! + \ldots$ ein Unterlauf bei der Berechnung von sinh([minfloat()]) nicht auftreten kann, aber $\sin^2(x)$ durch

$$\sin^2(x) = \frac{1}{4}\cdot\left(\frac{\sin(2x)}{\cos(x)}\right)^2$$

zu ersetzen ist, um für $x =$ `minfloat()` den rechten Term sin(2[minfloat()]) intervallmäßig ohne Unterlauf berechnen zu können. Das Quadrieren erfolgt mit skalierten Intervallen für sin(2[x]) und cos([x]), um einen möglichen Unterlauf zu vermeiden. Weitere Einzelheiten findet man im Quellcode der Funktion `Im_csc(x,y,rnd)` in `mpfrclass.cpp`. Es bleibt jetzt noch der

Fall 2. $|y| > y_0 = 372130555$, in dem ein Überlauf bei der Auswertung von $\sinh(2y)$ in (A.57) zu vermeiden ist. Wir schreiben mit $\sinh(y) = \mathrm{sign}(y)\cdot\sinh(|y|)$ (A.57) zunächst in der Form

$$f(x,y) = \frac{-\cos(x)}{\sinh(y)\cdot\left(1+\dfrac{\sin^2(x)}{\sinh^2(y)}\right)} = -\cos(x)\cdot\frac{2\cdot\mathrm{sign}(y)}{e^{|y|}-e^{-|y|}}\cdot\frac{1}{1+\dfrac{\sin^2(x)}{\sinh^2(y)}} \tag{A.58}$$

und werden versuchen, die beiden letzten Brüche rechts weiter zu vereinfachen. Es gilt zunächst

(A.59) $$\frac{2}{e^{|y|}-e^{-|y|}} = 2e^{-|y|}\cdot\frac{1}{1-e^{-2|y|}},\quad \text{mit:}\quad 1+e^{-2|y|} \le \frac{1}{1-e^{-2|y|}} \le 1+2e^{-2|y|},$$

und daraus ergibt sich für $f(x,y)$ die Einschließung

(A.60) $$f(x,y) \in -2\cdot\operatorname{sign}(y)\cdot\cos([x])\cdot e^{-|y|}\cdot\frac{[1+e^{-2|y|},1+2e^{-2|y|}]}{1+[0,1]/\sinh^2(y)}$$

Für $1/\sinh(y)$ und $|y| > 372130555$ beweist man elementar die folgenden Ungleichungen

$$\begin{aligned}\frac{1}{\sinh(y)} &= \frac{4}{e^{2|y|}-2+e^{-2|y|}} < \frac{4}{e^{2|y|}-2} < 5\cdot e^{-2|y|}\\ \frac{1}{\sinh(y)} &= \frac{4}{e^{2|y|}-2+e^{-2|y|}} > \frac{4}{e^{2|y|}+e^{-2|y|}} > 3\cdot e^{-2|y|}\end{aligned}\quad \text{und erhält mit (A.60)}$$

$$\begin{aligned} f(x,y) &\in -2\cdot\operatorname{sign}(y)\cdot\cos([x])\cdot e^{-|y|}\cdot\frac{[1+e^{-2|y|},1+2e^{-2|y|}]}{[1,1+5\cdot e^{-2|y|}]}\\ &= -2\cdot\operatorname{sign}(y)\cdot\cos([x])\cdot e^{-|y|}\cdot\left[\frac{1+e^{-2|y|}}{1+5e^{-2|y|}},1+2e^{-2|y|}\right]\\ &\subset -2\cdot\operatorname{sign}(y)\cdot\cos([x])\cdot e^{-|y|}\cdot[1-4e^{-2|y|},1+2e^{-2|y|}]. \end{aligned} \tag{A.61}$$

Zur weiteren Vereinfachung verlangen wir: $[1-4e^{-2|y|},1+2e^{-2|y|}] \subseteq [\mathtt{pred}(1),\mathtt{succ}(1)]$, d.h.

(A.62) $$1-4e^{-2y_0} > 1-2^{-\mathtt{prec}},\qquad 1+2e^{-2y_0} < 1+2^{1-\mathtt{prec}},\quad \text{d.h. wir verlangen}$$

(A.63) $$\mathtt{prec} < \frac{2y_0}{\ln(2)} - 2 = 1\,073\,741\,810{,}523\ldots;\quad \mathtt{prec} < \frac{2y_0}{\ln(2)} = 1\,073\,741\,812{,}523\ldots,$$

und beide Ungleichungen in (A.63) sind schon aus Laufzeitgründen sicher erfüllt. d.h es gilt

(A.64) $$f(x,y) \in -\operatorname{sign}(y)\cdot\cos([x])\cdot e^{-|y|+\ln(2)}\cdot[\mathtt{pred}(1),\mathtt{succ}(1)].$$

Damit kann $f(x,y)$ für $|y| > 372130555$ ohne vorzeitigen Unter- oder Überlauf nahezu optimal und sehr effektiv eingeschlossen werden.

A.11. $\sqrt{x^2-1}$

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:

$$D(x) := |x| - \frac{0.5}{|x| - \frac{1}{|x|}} < \sqrt{x^2-1} < |x| - \frac{0.5}{|x|} =: U(x), \qquad |x| \gg 1; \tag{A.65}$$

Für $|x| \gg 1$ sind alle drei Terme in (A.65) 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.65) 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.65) 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.12. $x/\sqrt{1+x^2}$

Da die monoton wachsende Funktion $f(x) := x/\sqrt{1+x^2}$ punktsymmetrisch ist, kann man sich bei der Auswertung auf $x > 0$ beschränken. In diesem Bereich gelten mit $m :=$ `minfloat()` die Abschätzungen

$$x \cdot (1-m) < x \cdot (1-x^2/2) < \frac{x}{\sqrt{1+x^2}} \equiv \frac{1}{\sqrt{1+(1/x)^2}} < x, \quad x > 0.$$

Die erste Abschätzung $x \cdot (1-m) < x \cdot (1-x^2/2)$ gilt für $0 < x < \sqrt{2m}$ und wird beim Abrunden im Fall $x \le 4m$ benutzt. Falls der aufgerundete Funktionswert für $x \ll 1$ größer ist als x, wird x selbst als Oberschranke gewählt. Die Auswertung von $f(x)$ erfolgt mit Hilfe von $f(x) = 1/\sqrt{1+(1/x)^2}$, da die Wurzel jetzt für $x \to$ `MaxFloat()` keinen vorzeitigen Überlauf verursacht.

Die nahezu optimale Auswertung von $f(x)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfrClass xdsqrt1px2 (const MpfrClass& x, RoundingMode rnd)
```

wobei die Rundung mittels `rnd` $\in$ `{RoundUp, RoundDown, RoundNearest}` gesteuert wird. Falls `rnd` nicht gesetzt wird, erfolgt die Rundung nach dem Current-Rundungsmodus.

A.13. $x/\sqrt{1-x^2}$

Da die monoton wachsende Funktion $f(x) := x/\sqrt{1-x^2}$ punktsymmetrisch ist, kann man sich bei der Auswertung auf $0 < x < +1$ beschränken. Die nahezu optimale Auswertung von $f(x)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfrClass xdsqrt1mx2 (const MpfrClass& x, RoundingMode rnd)
```

wobei die Rundung mittels `rnd` $\in$ `{RoundUp, RoundDown, RoundNearest}` gesteuert wird. Falls `rnd` nicht gesetzt wird, erfolgt die Rundung nach dem Current-Rundungsmodus. Weitere Einzelheiten findet man in der Datei `mpfrclass.cpp`.

A.14. $x/\sqrt{x^2-1}$

Da die monoton fallende Funktion $f(x) := x/\sqrt{x^2-1}$ punktsymmetrisch ist, kann man sich bei der Auswertung auf $x > +1$ beschränken. Die nahezu optimale Auswertung von $f(x)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfrClass xdsqrtx2m1 (const MpfrClass& x, RoundingMode rnd)
```

wobei die Rundung mittels `rnd` $\in$ `{RoundUp, RoundDown, RoundNearest}` gesteuert wird. Falls `rnd` nicht gesetzt wird, erfolgt die Rundung nach dem Current-Rundungsmodus. Weitere Einzelheiten findet man in der Datei `mpfrclass.cpp`.

A.15. $\ln(\sin(x))$

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 waagerechte 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[5]. Benutzt man daher für $\sin(x) \ge 0.5$ den Funktionsterm

$$f(x) = 0.5 \cdot \ln(1 - \cos^2(x)), \quad \sin(x) \ge 0.5, \tag{A.66}$$

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.66) mithilfe der Funktion `lnp1(...)` auswertet.

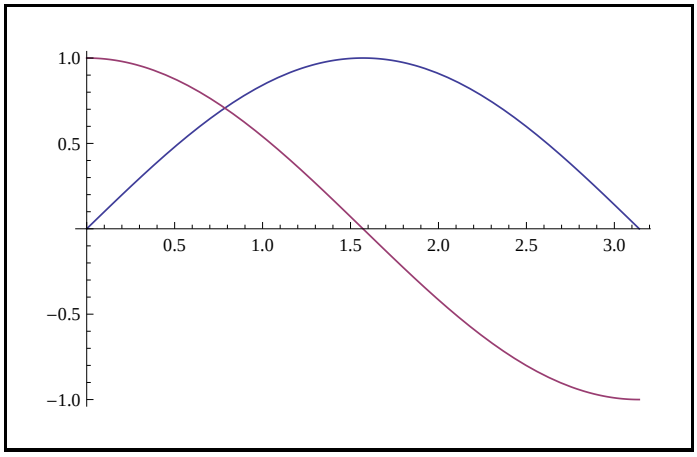


Abbildung A.1.: $y = \sin(x), \cos(x)$

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.66) 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 $\cos(x) > 0$[6].

[5] $f_n(x_0)$ bedeutet den zur nächsten Rasterzahl gerundeten Funktionswert.

[6] 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 $\cos_d(x)$

$$\cos_d(x) \le \cos(x) < 0, \tag{A.67}$$

und mit (A.18) folgt dann

$$\cos_d(x) \odot_u \cos_d(x) \ge \cos^2(x), \tag{A.68}$$

wobei wegen (A.67) 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 $\cos_u(x)$

$$\cos_u(x) \ge \cos(x) > 0, \tag{A.69}$$

und mit (A.15) folgt dann

$$\cos_u(x) \odot_u \cos_u(x) \ge \cos^2(x), \tag{A.70}$$

wobei wegen (A.69) die Bedingung (A.9) automatisch erfüllt ist und auch $\cos_u(x)$ nicht verschwinden kann.

Jetzt wird gezeigt, wie man mit (A.66) 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 $\cos_u(x)$

$$\cos(x) \le \cos_u(x) \le 0,$$

und mit (A.14) folgt dann die gewünschte garantierte Abrundung:

$$\cos_u(x) \odot_d \cos_u(x) \le \cos^2(x).$$

Sei jetzt: $\mathbf{\cos(x) > 0}$.
Dann gilt für den abgerundeten Funktionswert $\cos_d(x)$

$$0 \le \cos_d(x) \le \cos(x),$$

und mit (A.11) folgt dann wieder die gewünschte garantierte Abrundung:

$$\cos_d(x) \odot_d \cos_d(x) \le \cos^2(x).$$

Im unkritischen Bereich $0 < \sin(x) < 0.5$ wird $f(x) := \ln(\sin(x))$ direkt ausgewertet, wobei wegen der Monotonie der ln-Funktion zur Berechnung von $f_u(x)$ bzw. $f_d(x)$ auch $\sin(x)$ entsprechend auf- bzw. abzurunden ist. Dabei ist jedoch der Bereich $0 < x < 5 * \texttt{minfloat()}$ gesondert zu behandeln, um beim Abrunden zu große Überschätzungen zu vermeiden. So erhält man beispielsweise für $x = \texttt{minfloat()}$ den abgerundeten Funktionswert $f_d(x) = \texttt{-@Inf@}$, der jedoch eine viel zu 'kleine' Unterschranke darstellt. Um eine größere und optimale Unterschranke zu berechnen, betrachten wir die Reihe:

$$\begin{aligned}
\ln(\sin(x)) &= \ln(x) - \frac{x^2}{6} - \frac{x^4}{180} - \ldots = -\sum_{k=1}^{\infty} \frac{2^{2k-1}}{k \cdot (2k)!} |B_{2k}| x^{2k}, \quad 0 < x < \pi, \\
&> \ln(x) - \frac{x^2}{6}\left(1 + x^2 + x^4 + x^6 + \ldots\right) = \ln(x) - \frac{x^2}{6} \cdot \frac{1}{1-x^2}, \quad 0 < x < 1, \\
&> \ln(x) - \frac{x}{6} \cdot \frac{1}{1-x^2} > \ln(x) - \frac{x}{5} > \ln(x) - \texttt{minfloat()}, \text{ falls } x < 5 \cdot \texttt{minfloat()}.
\end{aligned}$$

Für $x < 5 \cdot \texttt{minfloat()}$ erhalten wir damit die Unterschranke: $\ln(x) - \texttt{minfloat()} < \ln(\sin(x))$.

Die Funktion $f(x) = \ln(\cos(x))$ ist in `mpfrclass.cpp` ganz analog implementiert und muss deshalb hier nicht weiter diskutiert werden.

A.16. $\ln(\sqrt{x^2+y^2})$

Wir betrachten

$$f(x) := \ln(\sqrt{x^2+y^2}), \quad \text{mit:} \quad |x| \geq |y| \text{ und } |x|, |y| > 0. \tag{A.71}$$

Um im Fall $|x| \gg 1$ einen vorzeitigen Überlauf zu vermeiden, wird folgende Darstellung benutzt:

$$f(x) := \ln(|x|) + \frac{1}{2}\ln\left(1 + (|y|/|x|)^2\right) > 0, \quad |y|/|x| > 0,\ |x| \gg 1. \tag{A.72}$$

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.72) 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:

$$A_u \geq A > 0 \tag{A.73}$$

$$0 \leq A_d < A. \tag{A.74}$$

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.73) 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.74) garantiert wird.

Wenn $f_u(x)$ und $f_d(x)$ nach (A.71) 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.17. $\ln(\sqrt{(1+x)^2+y^2}\,)$

Da $u(x,y) = \ln(\sqrt{(1+x)^2+y^2})$ auch als Realteil der komplexen Funktion $\ln(1+z)$, mit $z \in \mathbb{C}$, benötigt wird, findet man eine ausführliche Beschreibung des Algorithmus ab Seite 267.

A.18. arcosh$(1+x)$

Wir betrachten für $x \geq 0$

$$\text{(A.75)} \qquad \operatorname{arcosh}(1+x) = \ln(1+x+\sqrt{x\cdot(2+x)}), \quad x \geq 0,$$

$$\text{(A.76)} \qquad = \ln(x) + \ln(1+1/x+\sqrt{1+2/x}), \quad x \gg 1;$$

In (A.76) 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 eine monoton wachsende Funktion ist.

In (A.75) 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:

$$\text{(A.77)} \qquad 0 \leq A = A_u = A_d,$$

$$\text{(A.78)} \qquad 2 \leq B_d \leq B,$$

$$\text{(A.79)} \qquad 2 \leq B \leq B_u;$$

Nach (A.15) gilt: $P_u := A_u \odot_u B_u \geq x\cdot(2+x)$, wobei die Bedingung (A.9) wegen (A.77) automatisch erfüllt ist und auch $A_u = A$ und $B_u \geq 2$ nicht beide verschwinden können.

Nach (A.11) gilt wegen (A.77) und (A.78) 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.19. $\Gamma'(x)$

$\Gamma'(x)$ wird mithilfe der Digamma-Funktion $\psi(x) = \Gamma'(x)/\Gamma(x)$ über das folgende Produkt berechnet:

$$\Gamma'(x) = \psi(x) \cdot \Gamma(x), \quad x \neq 0, -1, -2, \ldots \tag{A.80}$$

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.20. $1/\Gamma(x)$

$f(x) := 1/\Gamma(x)$ 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.21. $(1/\Gamma(x))'$

Für die überall differenzierbare Funktion $f(x) := (1/\Gamma(x))'$ gilt:

$$f(x) = \left(\frac{1}{\Gamma(x)}\right)' = \begin{cases} \dfrac{-\Gamma'(x)}{\Gamma^2(x)} = -\dfrac{\Gamma'(x)}{\Gamma(x)}/\Gamma(x), & x \neq 0, -1, -2, \ldots \\ |x|!, & x = 0, -2, -4, \ldots \\ -|x|!, & x = -1, -3, -5, \ldots \end{cases} \tag{A.81}$$

$|x|!$ wird mithilfe der `faktorial(...)`-Funktion berechnet und die erste Zeile in (A.81) 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 $|x| \leq 2 \cdot \texttt{succ}(0)$ 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 $x = \texttt{succ}(0)$ und $x = 2 \cdot \texttt{succ}(0)$ ist damit wegen $f(0) = 1$ eine Einschließung von $f(x)$ gegeben durch:

$$t = 3 \cdot \texttt{succ}(0), \quad u = \texttt{digamma(t, RoundDown)}, \quad v = \texttt{gamma(t, RoundDown)},$$
$$u = -u \diamond v;$$
$$f(x) \in [1, \texttt{Sup}(u)], \quad x = \texttt{succ}(0) \text{ oder } x = 2 \cdot \texttt{succ}(0)$$

wobei wegen $t = 3 \cdot \texttt{succ}(0)$ die Punktintervalle u und v jetzt keinen Überlauf mehr erzeugen. Natürlich ist das Einschließungsintervall $[1, \texttt{Sup}(u)]$ eine gewisse Überschätzung von $f(x)$, aber dies wird in der numerischen Praxis kaum eine Rolle spielen.

Für die negativen Argumente $x = \texttt{pred}(0)$ und $x = 2 \cdot \texttt{pred}(0)$ gelten ganz entsprechende Aussagen. Den ausführlichen Algorithmus findet man in `mpfrclass.cpp`.

B. Neue Funktionen vom Typ `MpfiClass`

B.1. $\ln(\sqrt{(1+x)^2+y^2}\,)$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass ln_sqrtxp1_2y2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$u(x,y) = \ln(\sqrt{(1+x)^2+y^2}) = \frac{1}{2}\cdot\ln((1+x)^2+y^2), \quad \text{mit:}\ \ x \in \mathtt{X},\ y \in \mathtt{Y}.$$

Eine ausführliche Beschreibung von $u(x,y)$ findet man ab Seite 267. $u(x,y)$ ist als Realteilfunktion der holomorphen Funktion $\log(1+z),\ z \in \mathbb{C}$, eine harmonische Funktion, wenn der singuläre Punkt $P(-1,0)$ nicht ein Element des zweidimensionalen reellen Intervalls Z ist.

$$Z := \left\{(x,y)\,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad (-1,0) \notin Z. \tag{B.1}$$

$u(x,y)$ nimmt dann für alle $(x,y) \in Z$ als harmonische Funktion ihre relativen Extrema auf dem Rand von Z an, vgl. dazu auch Seite 244. Die Berechnung dieser Extremstellen, $m(x,y)$ für das relative Minimum und $M(x,y)$ für das relative Maximum, kann wesentlich vereinfacht werden, wenn man $u(x,y) \equiv u(x,|y|)$ beachtet und daher das Eingangsintervall `Y` durch das Intervall $Y = \mathtt{abs(Y)} = \{|y|\,|\,y \in \mathtt{Y}\}$ seiner Absolutbeträge ersetzt. Die zu betrachtenden Intervalle Z aus (B.1) liegen damit alle in der oberen Halbebene, und für die interne Berechnung gilt mit $X = \mathtt{X}$:

$$Z := \left\{(x,y)\,\middle|\, x \in X = [x_1,x_2] \wedge y \in Y = [y_1,y_2]\right\}, \quad (-1,0) \notin Z,\ y_1 \geq 0. \tag{B.2}$$

Um die Koordinaten der Extremalpunkte m, M bestimmen zu können, berechnen wir zunächst die partiellen Ableitungen von $u(x,y)$ bezüglich x, y:

$$\begin{aligned} \frac{\partial u(x,y)}{\partial x} &= \frac{1+x}{(1+x)^2+y^2} = 0, \qquad \text{d.h.}\ \ x = -1, \\ \frac{\partial u(x,y)}{\partial y} &= \frac{y}{(1+x)^2+y^2} = 0, \qquad \text{d.h.}\ \ y = 0. \end{aligned}$$

Auf einer Parallelen zur x-Achse wächst bzw. fällt $u(x,y)$ monoton für $x > -1$ bzw. für $x < -1$, und auf einer Parallelen zur y-Achse ist $u(x,y)$ (in der oberen Halbebene) monoton wachsend. Die Extremalkurven sind wegen $x = -1$ einmal die Parallele zur y-Achse durch $x = -1$ und zum anderen wegen $y = 0$ die x-Achse, wobei letztere keine Rolle spielt, da wir nur Intervalle Z in der oberen Halbebene betrachten. Schneidet die Extremalkurve $x = -1$ das Rechteck Z, so besitzt m die Koordinaten $m(-1,y_1)$. In Abb. B.1 geben die Pfeile auf dem Rand von Z die Richtung wachsender $u(x,y)$-Werte an, so dass die Extremalpunkte m und M die folgenden Koordinaten besitzen: $m(x_1,y_1),\ M(x_2,y_2)$. Die breiten Pfeile links und rechts neben der Extremalkurve $x = -1$ geben das Monotonieverhalten der Funktion $u(x,y)$ an. Das zweidimensionale, grüne Intervall ist nicht erlaubt, da es den singulären Punkt $P(-1,0)$ enthält. Die Koordinaten der Extremalpunkte m, M der erlaubten gelben, zweidimensionalen Intervalle Z sind stets Maschinenzahlen, so dass die Berechnung der Funktionswerte $u(x,y)$ an diesen Stellen einfach realisiert werden kann.

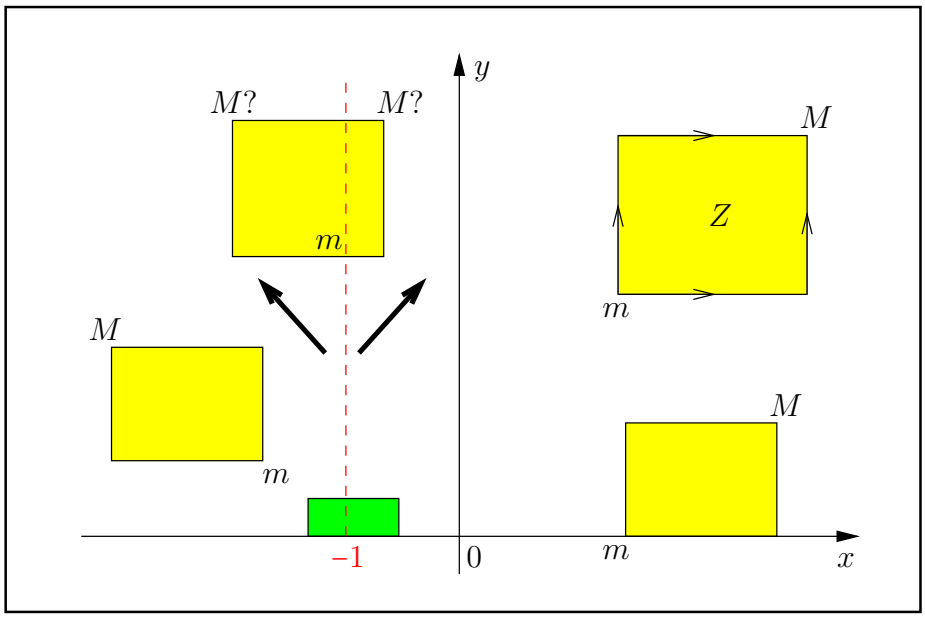


Abbildung B.1.: Zweidimensionale Intervalle Z mit Extremalpunkten m, M

Bei einem Intervall Z, das die Extremalkurve $x = -1$ schneidet, muss noch entschieden werden, welcher der beiden möglichen Punkte $M?$ das Maximum von $u(x, y)$ wirklich liefert. Nach der obigen Abbildung ergibt sich

$$|1 + x_1| \geq |1 + x_2| \quad \Longleftrightarrow \quad -1 - x_1 \geq x_2 - (-1) \quad \Longleftrightarrow \quad x_1 + 2 \leq -x_2,$$

so dass nach obiger Abbildung für den Maximumpunkt gilt: $M(x_1, y_2)$. Die letzte Ungleichung muss auf der Maschine ausgewertet werden. Dazu berechnet man zunächst das reelle Intervall $z = x_1 \oplus 2 = [z_1, z_2]$, das wegen $x_1 \leq -1$ keinen Überlauf erzeugt. Dabei ist z entweder ein Punktintervall oder in seinem Innern befindet sich keine Maschinenzahl, d.h. es gilt $z_2 = z_1$ oder $z_2 = \texttt{succ}(z_1)$. Aus der Doppelungleichung $x_1 + 2 \leq z_2 \leq -x_2$, d.h. aus $z_2 \leq -x_2$ folgt dann $x_1 + 2 \leq -x_2$ und damit $M(x_1, y_2)$. Ganz entsprechend folgt aus $z_1 \geq -x_2$ für den Maximumpunkt $M(x_2, y_2)$.

Nach diesen Überlegungen sind die **beiden** Abfragen $z_2 \leq -x_2$ und $z_1 \geq -x_2$ nötig, um die richtige x-Koordinate von M zu bestimmen. Wir zeigen jetzt, dass dazu **nur eine** Abfrage, z.B. $z_2 \leq -x_2$, notwendig ist und dabei nur der rechte Randpunkt $z_2 \geq x_1 + 2$ bekannt sein muss. In der folgenden Abbildung findet man auf der oberen x-Achse die Situation, wenn $z_2 \leq -x_2$ und $z_2 = \texttt{succ}(z_1)$ erfüllt sind. Im grünen Bereich liegen dort alle möglichen $-x_2$-Werte, so dass die

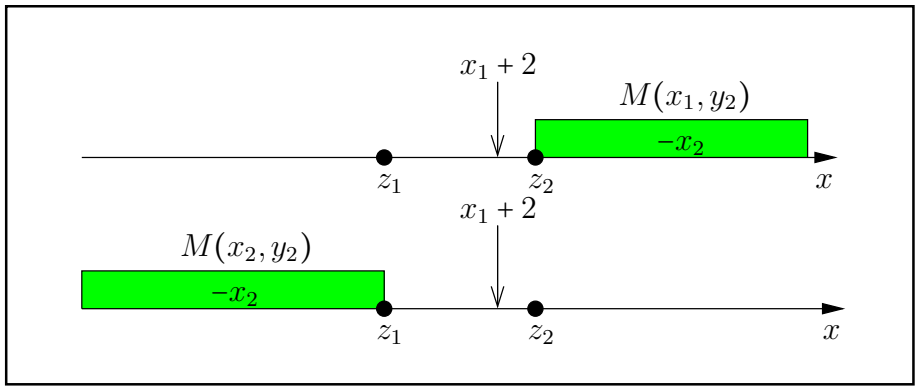


Abbildung B.2.: x-Koordinate von $M(?, y_2)$ im Fall: $-1 \in [x_1, x_2]$

Bedingung $x_1 + 2 \le -x_2$ stets erfüllt ist und damit M gegeben ist durch $M(x_1, y_2)$. Wenn jedoch $z_2 \le -x_2$ nicht erfüllt ist, so gilt $z_2 > -x_2$ und da zwischen z_1 und z_2 keine Maschinenzahl liegt, können die möglichen $-x_2$-Werte jetzt nur in dem grünen Bereich auf der unteren x-Achse liegen, d.h. es gilt stets $x_1 + 2 \ge -x_2$, so dass jetzt M gegeben ist durch $M(x_2, y_2)$.

Zusammenfassung:

- $x_1 \ge -1 \quad \Longrightarrow \quad m = m(x_1, y_1), \quad M = M(x_2, y_2);$
- $x_2 \le -1 \quad \Longrightarrow \quad m = m(x_2, y_1), \quad M = M(x_1, y_2);$
- $-1 \in [x_1, x_2], \quad z_2 := x_1 \oplus_u 2 \quad \Longrightarrow$
  1. $m = m(-1, y_1);$
  2. $z_2 \le -x_2 \quad \Longrightarrow \quad M = M(x_1, y_2);$ Nur diese Abfrage muss realisiert werden!
  3. $z_2 > -x_2 \quad \Longrightarrow \quad M = M(x_2, y_2);$

Beachten Sie, dass die Maschinenzahl $z_2 := x_1 \oplus_u 2$ die optimal aufgerundete Summe $(x_1 + 2)$ ist.

Numerische Ergebnisse:

Bezeichnungen: $t = \ln(\sqrt{(1+x)^2 + y^2}), \quad x \in X = [x_1, x_2], \quad y \in Y = [y_1, y_2];$
Vergleichen Sie für Punktargumente auch die auf- und abgerundeten Ergebnisse von Seite 269.

1. Mit $X = Y =$ [`MaxFloat(prec)`,`MaxFloat(prec)`] und `prec` = 3000 erhält man für t die nahezu optimale Einschließung mit 902 gemeinsamen Dezimalstellen:

   $$t \in [7.4426111760831942759393...664834e8, 7.4426111760831942759393...664836e8].$$

2. Mit $X = Y =$ [`minfloat(prec)`,`minfloat(prec)`] und `prec` = 3 000 000 erhält man für t die notwendigerweise grobe Einschließung:

   $$t \in [0, 2.38256490488795107321616978173267 45...34539016124e - 323228497].$$

3. Mit $X = Y =$ [$2 \cdot$`minfloat(prec)`, $2 \cdot$`minfloat(prec)`] und `prec` = 30 000 erhält man für t die nahezu optimale Einschließung mit 9030 gemeinsamen Dezimalstellen:

   $$t \in [4.765129809775...95148232e - 323228497, 4.765129809775...95148233e - 323228497].$$

4. Mit $X = [-2, -2]$, $Y = [2^{-10000}, 2^{-10000}]$ und `prec` = 30 000 erhält man für t die nahezu optimale Einschließung mit 9030 gemeinsamen Dezimalstellen:

   $$t \in [1.2561940288493722925...505035e - 6021, 1.2561940288493722925...505036e - 6021].$$

   Die gleiche Einschließung erhält man natürlich auch mit $X = [0, 0]$, $Y = [2^{-10000}, 2^{-10000}]$.

5. Mit $X =$ [`succ(-1)`, `succ(-1)`], `succ(-1)` $= -1 + 2^{-\texttt{prec}}$, $Y = [2^{-30000}, 2^{-30000}]$ und der Präzision `prec` = 30 000 erhält man für t die nahezu optimale Einschließung mit 9030 gemeinsamen Dezimalstellen:

   $$t \in [-2.079406884320807930...15528832738e4, -2.079406884320807930...15528832737e4].$$

   Beachten Sie, dass eine Präzision von 30 0000 Bits einer Präzision von $30\,0000/\log_2(10) = 30\,000/3.32192809... \approx 9031$ Dezimalstellen entspricht. Bezüglich `succ(-1)` $= -1 + 2^{-\texttt{prec}}$ vergleiche Seite 17.

B.2. $x/(x^2+y^2)$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass x_div_x2py2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x,y) = \frac{x}{x^2+y^2} \quad \text{mit:} \;\; x \in \mathtt{X},\; y \in \mathtt{Y} \;\; \text{und} \;\; x \neq 0 \wedge y \neq 0.$$

Die Berechnung von $f(x,y)$ wird ausführlich beschrieben auf Seite 185. $f(x,y)$ ist als Realteilfunktion der holomorphen Funktion $1/z,\; z \in \mathbb{C}\backslash\{0\}$, eine harmonische Funktion, wenn der singuläre Punkt $P(0,0)$ nicht ein Element des zweidimensionalen reellen Intervalls Z ist.

$$Z := \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad 0 \notin \mathtt{X} \wedge 0 \notin \mathtt{Y}. \tag{B.3}$$

$f(x,y)$ nimmt daher für alle Punkte $(x,y) \in Z$ als harmonische Funktion ihre Extrema auf dem Rand von Z an, vgl. dazu auch die Seite 244. Die Berechnung dieser Extremstellen, $m(x,y)$ für das Minimum und $M(x,y)$ für das Maximum, kann wesentlich vereinfacht werden, wenn man $f(x,y) \equiv f(x,|y|)$ beachtet und daher das Eingangsintervall $\mathtt{Y}$ durch das Intervall $Y = \mathtt{abs(Y)} = \{|y| \,|\, y \in \mathtt{Y}\}$ seiner Absolutbeträge ersetzt. Die zu betrachtenden Intervalle Z aus (B.3) liegen damit alle in der oberen Halbebene, und für die interne Berechnung gilt mit $X = \mathtt{X}$:

$$Z := \left\{(x,y) \,\middle|\, x \in X = [x_1,x_2] \wedge y \in Y = [y_1,y_2]\right\}, \quad (0,0) \notin Z,\; y_1 \geq 0. \tag{B.4}$$

Wegen $f(-x,y) \equiv -f(x,y)$ kann die Berechnung der Extrema m, M noch weiter auf den ersten Quadranten reduziert werden, wobei das Minimum m und das Maximum M berechnet werden mithilfe der Funktionen:

```
MpfrClass x_div_m(const MpfiClass& x, const MpfiClass& y);   Inf(x)>=0, Inf(y)>=0,
MpfrClass x_div_M(const MpfiClass& x, const MpfiClass& y);   Inf(x)>=0, Inf(y)>=0.
```

Um die Extrema m, M mit den obigen Funktionen richtig berechnen zu können, sind drei Fälle zu unterscheiden:

1. $x_1 \geq 0 \wedge y_1 \geq 0$: `m = x_div_m(X,Y); M = x_div_M(X,Y);`
2. $x_2 < 0 \wedge y_1 \geq 0$: `res = -x; m = x_div_m(res,Y); M = x_div_M(res,Y);`
   `swap(m,M); m = -m; M = -M;`
3. $x_1 < 0 \wedge x_2 \geq 0 \wedge y_1 \geq 0$: `res` $= [0,-x_1]$; `m = -x_div_M(res,Y);`
   `res` $= [0,x_2]$; `M = x_div_M(res,Y);`

Um jetzt im 1. Quadranten die beiden Funktionen `x_div_m(...), x_div_M(...)` realisieren zu können, benötigen wir zunächst für $x, y \geq 0$ die beiden partiellen Ableitungen von $f(x,y)$:

$$\frac{\partial f(x,y)}{\partial x} = \frac{y^2-x^2}{(x^2+y^2)^2}, \tag{B.5}$$

$$\frac{\partial f(x,y)}{\partial y} = \frac{-2xy}{(x^2+y^2)^2} \leq 0. \tag{B.6}$$

Oberhalb bzw. unterhalb der ersten Winkelhalbierenden gilt damit: $\partial f/\partial x > 0$ bzw. $\partial f/\partial x < 0$, und auf einer Parallelen zur y-Achse nehmen die Funktionswerte von $f(x,y)$ mit wachsenden y-Werten monoton ab, und wegen

$$f(x_1,x_1) = \frac{x_1}{x_1^2+x_1^2} \geq \frac{x_2}{x_2^2+x_2^2} \iff \frac{1}{2x_1} \geq \frac{1}{2x_2} \iff 0 < x_1 \leq x_2 \tag{B.7}$$

nehmen die Funktionswerte von $f(x,y)$ monoton zu, wenn man sich im 1. Quadranten auf der 1. Winkelhalbierenden dem Ursprung nähert. Mit diesen Ergebnissen erhält man in folgender Abbildung die Lage der Extremstellen m, M zu gegebenen zweidimensionalen Intervallen Z_i, wobei fünf grundsätzliche Lagen dieser Z_i zu unterscheiden sind.

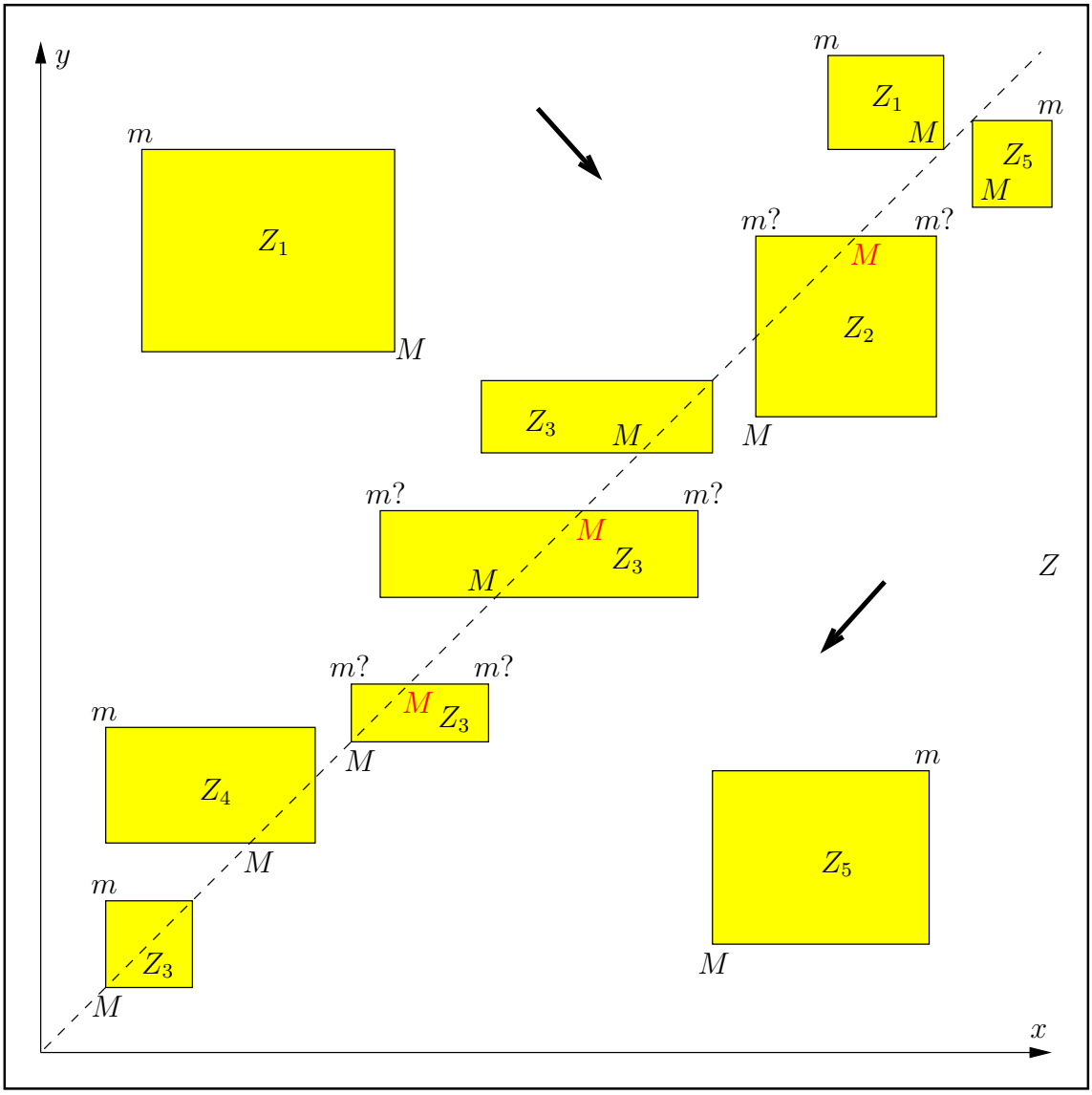


Abbildung B.3.: Zweidimensionale Intervalle Z_i mit Extremalpunkten m, M

Die fünf grundsätzlichen Lagen der Z_i sind definiert durch:

1. $x_2 \leq y_1$, Z_1;
2. $y_1 < x_1 < y_2$, Z_2;
3. $x_1 \leq y_1 \leq x_2 \wedge x_1 \leq y_2 \leq x_2$, Z_3;
4. $y_1 < x_2 < y_2$, Z_4;
5. $y_2 \leq x_1$, Z_5;

Die Punkte M auf der 1. Winkelhalbierenden sind Punkte mit relativen Maxima, wobei jedoch gilt $M \geq M$. Bei den Punktepaaren $m?, m?$ ist jeweils der Punkt mit dem kleinsten Funktionswert auszuwählen. Die beiden Pfeile oberhalb und unterhalb der ersten Winkelhalbierenden geben die Richtung wachsender Funktionswerte von $f(x, y)$ an.

B.3. $(x^2-y^2)/(x^2+y^2)^2$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass Re_rz2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x,y) = \frac{x^2-y^2}{(x^2+y^2)^2} \quad \text{mit:} \quad x \in \mathtt{X},\ y \in \mathtt{Y} \quad \text{und} \quad x \neq 0 \wedge y \neq 0.$$

Die Berechnung von $f(x,y)$ wird ausführlich beschrieben auf Seite 186. $f(x,y)$ ist als Realteilfunktion der holomorphen Funktion $1/z^2,\ z \in \mathbb{C}\backslash\{0\}$, eine harmonische Funktion, wenn der singuläre Punkt $P(0,0)$ nicht ein Element des zweidimensionalen reellen Intervalls Z ist.

$$Z := \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad 0 \notin \mathtt{X} \wedge 0 \notin \mathtt{Y}. \tag{B.8}$$

$f(x,y)$ nimmt daher für alle Punkte $(x,y) \in Z$ als harmonische Funktion ihre Extrema auf dem Rand von Z an, vgl. dazu auch die Seite 244. Die Berechnung dieser Extremstellen, $m(x,y)$ für das Minimum und $M(x,y)$ für das Maximum, kann wesentlich vereinfacht werden, wenn man $f(x,y) \equiv f(|x|,|y|)$ beachtet und die Eingangsintervalle `X,Y` durch die jeweiligen Intervalle, z.B. $X = \mathtt{abs(X)} = \{|x| \,|\, x \in \mathtt{X}\}$, seiner Absolutbeträge ersetzt. Damit reduziert sich die Berechnung der Extremstellen $m(x,y), M(x,y)$ auf ein Intervall $Z := \left\{(x,y) \,\middle|\, x \in X \wedge y \in Y\right\},\ x \geq 0, y \geq 0$, das nur im 1. Quadranten liegt und den Ursprung nicht enthalten darf. $X = [x_1, x_2],\ Y = [y_1, y_2]$.

Zur Berechnung der Extremstellen m, M benötigt man zunächst die partiellen Ableitungen:

$$\frac{\partial f(x,y)}{\partial x} = \frac{2x \cdot (-x^4 + 2x^2y^2 + 3y^4)}{(x^2+y^2)^4} = \frac{2x \cdot \left(4y^4 - (x^2-y^2)^2\right)}{(x^2+y^2)^4}, \tag{B.9}$$

$$\frac{\partial f(x,y)}{\partial y} = \frac{-2y \cdot (-y^4 + 2x^2y^2 + 3x^4)}{(x^2+y^2)^4} = \frac{-2y \cdot \left(4x^4 - (x^2-y^2)^2\right)}{(x^2+y^2)^4}. \tag{B.10}$$

Die Extremalkurven für $\partial f/\partial x$ ergeben sich aus der Forderung $\partial f/\partial x = 0$, siehe Seite 247. Die erste Extremalkurve ist wegen $x = 0$ die positive y-Achse, und die zweite ergibt sich aus der Forderung $4y^4 = (x^2-y^2)^2 \iff 2y^2 = |x^2-y^2|$, aus der $y = x/\sqrt{3}$ folgt.

Die beiden Extremalkurven für $\partial f/\partial y$ ergeben sich analog aus der Forderung $\partial f/\partial y = 0$. Die erste Extremalkurve ist dann wegen $y = 0$ die positive x-Achse und die zweite Extremalkurve ist gegeben durch $y = \sqrt{3} \cdot x$.

Anmerkungen:

1. Da beide partiellen Ableitungen auf ihren Extremalkurven das Vorzeichen wechseln, wenn man parallel zu den beiden Koordinatenachsen diese Extremalkurven schneidet, besitzt $f(x,y)$ an den Schnittpunkten der Intervallränder von Z mit den Extremalkurven längs dieser Intervallränder tatsächlich relative Extrema, d.h. in diese Fällen müssen die Extremalpunkte m oder M nicht auf den Eckpunkten von Z liegen.

2. Es gilt folgendes: $x > y \implies f(x,y) > 0$ und $x < y \implies f(x,y) < 0$, d.h. wird Z von der 1. Winkelhalbierenden geschnitten, so kann das Minimum nur auf dem Rand des oberen und das Maximum nur auf dem Rand des unteren Z-Abschnitts liegen.

3. In den folgenden Abbildungen sind alle typischen Lagen der Intervalle Z mit den entsprechenden Lagen der Extremstellen $m(x,y), M(x,y)$ angegeben. Dabei bedeutet das Symbol (M), dass hier zwar ein relatives Maximum vorliegt, dass aber das wirkliche Maximum an der anderen Stelle M angenommen wird. Die zwei Symbole $m?$ bedeuten, dass das wirkliche Minimum nur an einer dieser beiden Stellen angenommen werden kann, so dass zwei Berechnungen und ein Vergleich erforderlich sind. Die 1. Winkelhalbierende und die beiden Extremalkurven $y = \sqrt{3} \cdot x$ und $y = x/\sqrt{3}$ sind leicht verdreht dargestellt. Die farbigen Pfeile geben die Richtung wachsender Funktionswerte von $f(x,y)$ an.

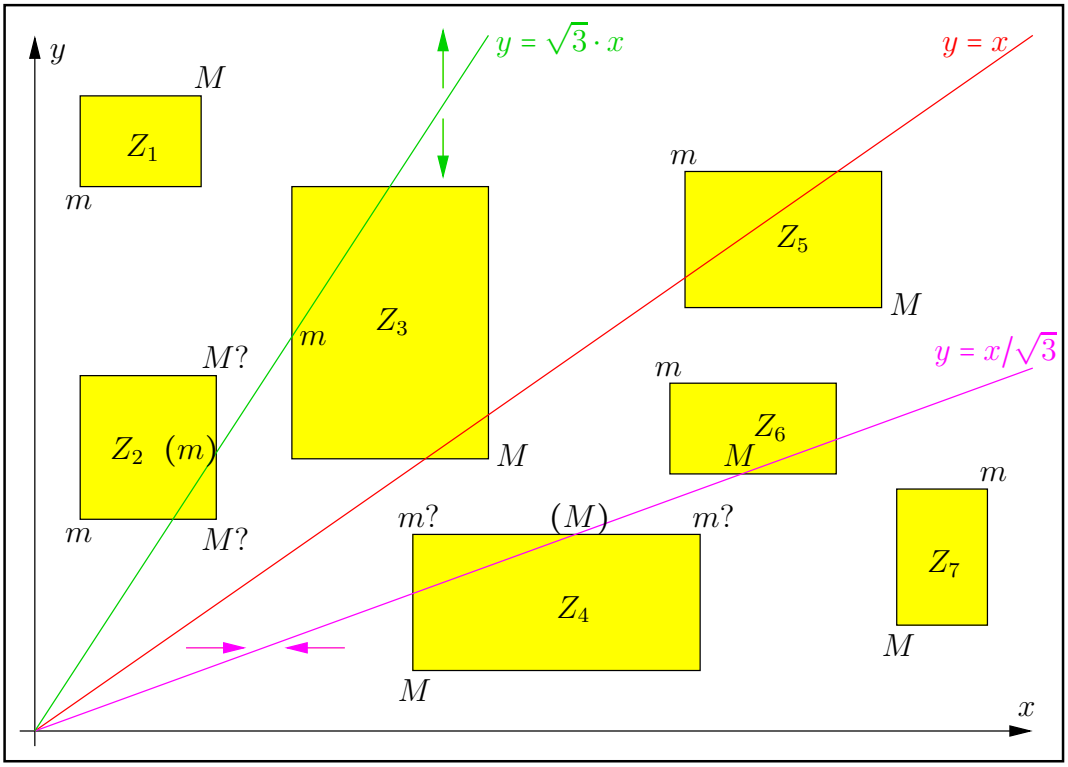


Abbildung B.4.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

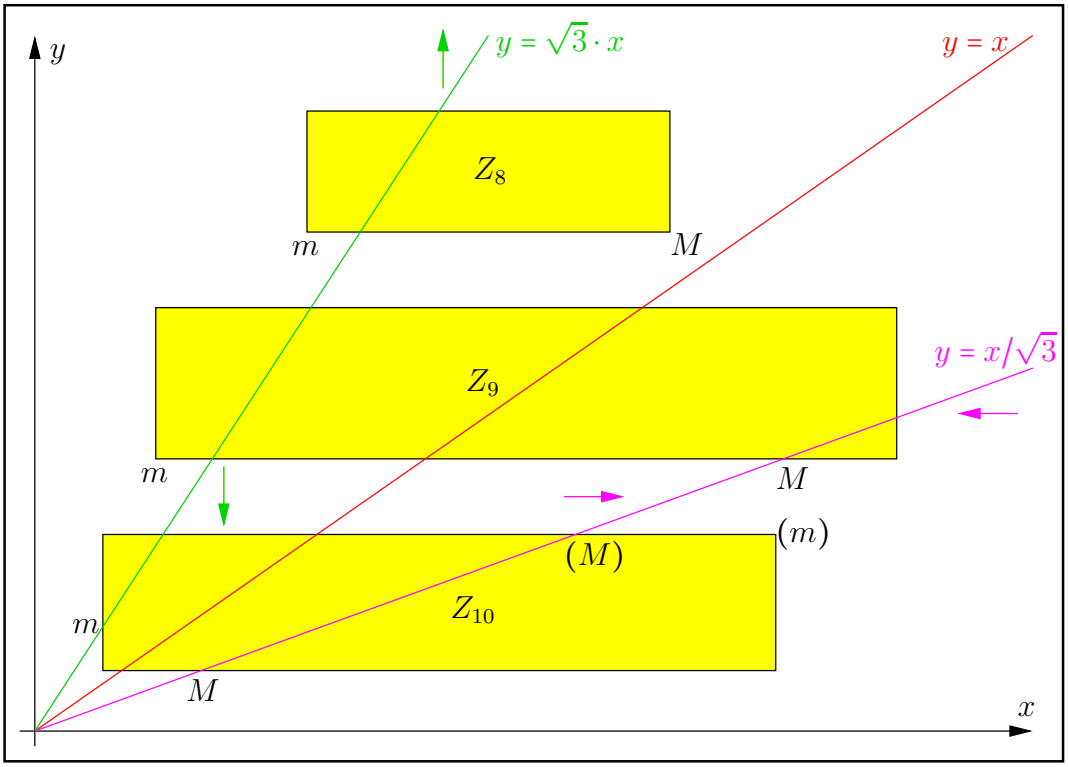


Abbildung B.5.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

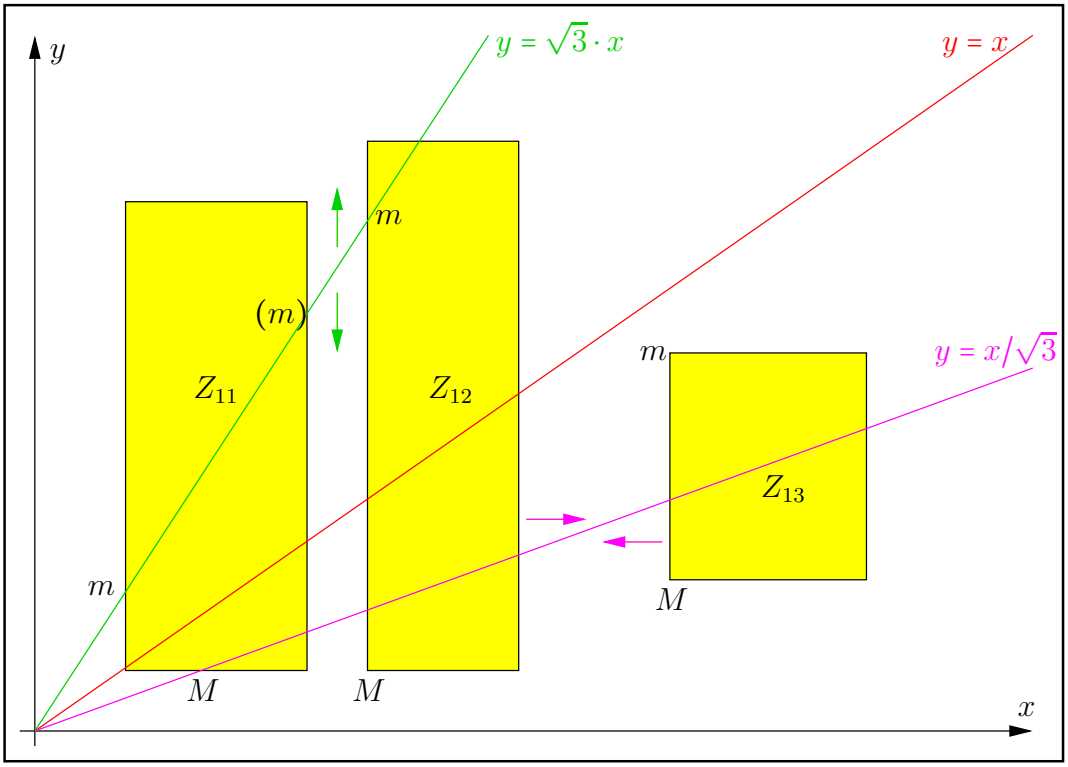


Abbildung B.6.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

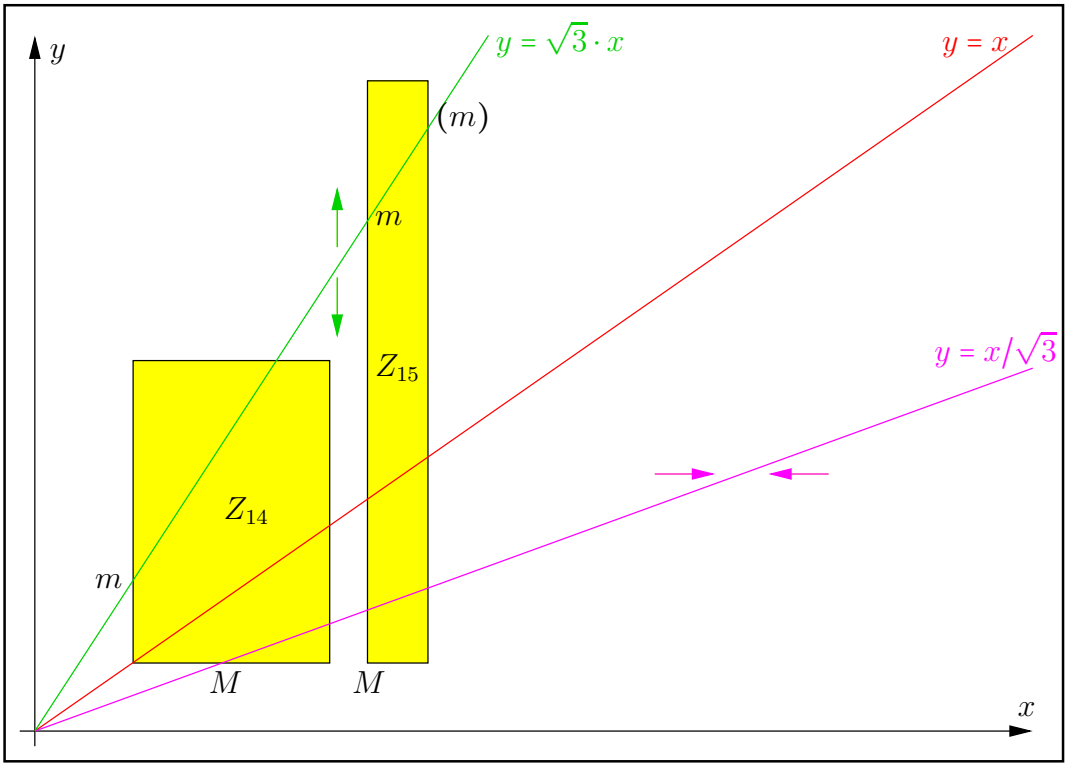


Abbildung B.7.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Anmerkungen:

1. Bezüglich des zweidimensionalen Intervalls Z_2 muss noch geklärt werden, dass das Minimum nicht bei (m), sondern bei m angenommen wird. Unter der Voraussetzung $x_2 > x_1$ muss daher gezeigt werden: $f(x_1, y_1) = (x_1^2 - y_1^2)/(x_1^2 + y_1^2)^2 < f(x_2, \sqrt{3} \cdot x_2) = -1/(8x_2^2)$.

   Wegen $\sqrt{3} \cdot x_1 \le y_1 \le \sqrt{3} \cdot x_2$ gilt zunächst $\dfrac{x_1^2 - y_1^2}{(x_1^2 + y_1^2)^2} \le \dfrac{x_1^2 - 3 \cdot x_1^2}{(x_1^2 + 3 \cdot x_1^2)^2} = \dfrac{-1}{8 \cdot x_1^2} < \dfrac{-1}{8x_2^2}$ ∎

2. Bezüglich der zweidimensionalen Intervalle Z_4, Z_{10} muss noch geklärt werden, dass das Maximum nicht bei (M), sondern bei M angenommen wird. Der Bewies ist einfach, denn läuft man von (M) senkrecht nach unten und dann parallel zu x-Achse nach links bis M, so wachsen die Funktionswerte jeweils monoton.

3. Bezüglich des zweidimensionalen Intervalls Z_{10} muss noch geklärt werden, dass das Minimum nicht bei (m), sondern bei m angenommen wird. Dies ist jedoch trivial, da die Funktionswerte von $f(x, y)$ bei m negativ und bei (m) positiv sind.

4. Bezüglich der zweidimensionalen Intervalle Z_{11}, Z_{15} muss noch geklärt werden, dass das Minimum nicht bei (m), sondern bei m angenommen wird. Im Fall $x_2 > x_1$ muss daher gezeigt werden: $f(x_1, \sqrt{3}x_1) < f(x_2, \sqrt{3}x_2)$.

   $f(x_1, \sqrt{3}x_1) < f(x_2, \sqrt{3}x_2) \iff \dfrac{x_1^2 - 3x_1^2}{(x_1^2 + 3x_1^2)^2} < \dfrac{x_2^2 - 3x_2^2}{(x_2^2 + 3x_2^2)^2} \iff \dfrac{-1}{x_1^2} < \dfrac{-1}{x_2^2} \iff x_2 > x_1$ ∎

B.3.1. Maximumbestimmung

Zur Bestimmung der Lage von M sind nacheinander folgende Fälle zu unterscheiden:

1. Wenn die untere Parallele zu x-Achse unterhalb der Extremalkurve $y = x/\sqrt{3}$ liegt, so ist die Maximumstelle M die linke untere Ecke, vgl. $Z_4, Z_7, Z_{12}, Z_{13}, Z_{15}$.
   Sei `Ux1` die optimale Einschließung von $x_1/\sqrt{3}$, dann gilt: Die untere Parallele zu x-Achse liegt unterhalb der Extremalkurve $y = x/\sqrt{3}$ $\iff$ $y_1 \le$ `Inf(Ux1)`.

2. Wenn die untere Parallele zu x-Achse die Extremalkurve $y = x/\sqrt{3}$ schneidet, so ist dieser Schnittpunkt gleich M, vgl. $Z_6, Z_9, Z_{10}, Z_{11}, Z_{14}$.
   Sei `Ux1` die optimale Einschließung von $x_1/\sqrt{3}$ und `Ux2` die optimale Einschließung von $x_2/\sqrt{3}$, dann gilt: Die untere Parallele zu x-Achse schneidet die Extremalkurve $y = x/\sqrt{3}$ $\iff$ `Sup(Ux1)` $\le y_1 \le$ `Inf(Ux2)`.
   Es gilt $M(x_S, y_S) = M(\sqrt{3} \cdot y_1, y_1)$, und für den maximalen Funktionswert erhält man: $f(x_S, y_S) = (3y_1^2 - y_1^2)/(3y_1^2 + y_1^2)^2 = 1/(8y_1^2) = (1/(\sqrt{8} \cdot y_1))^2$.

3. Wenn die rechte Parallele zu y-Achse die Extremalkurve $y = \sqrt{3} \cdot x$ schneidet, so ist M der rechte untere oder obere Eckpunkt, vgl. Z_2.
   Sei `Uy1` die optimale Einschließung von $y_1/\sqrt{3}$ und `Uy2` die optimale Einschließung von $y_2/\sqrt{3}$, dann gilt: Die rechte Parallele zu y-Achse schneidet die Extremalkurve $y = \sqrt{3}x$ $\iff$ `Sup(Uy1)` $\le x_2 \le$ `Inf(Uy2)`.

4. Wenn der untere rechte Eckpunkt unterhalb der Extremalkurve $y = \sqrt{3} \cdot x$ liegt, so ist dieser untere rechte Eckpunkt gleich M, vgl. Z_3, Z_5, Z_8.
   Sei `Uy1` die optimale Einschließung von $y_1/\sqrt{3}$, dann gilt: Der untere rechte Eckpunkt liegt unterhalb der Extremalkurve $y = \sqrt{3} \cdot x$ $\iff$ `Sup(Uy1)` $\le x_2$.

5. Wenn der untere rechte Eckpunkt oberhalb der Extremalkurve $y = \sqrt{3} \cdot x$ liegt, so ist M der obere rechte Eckpunkt, vgl. Z_1.
   Sei `Uy1` die optimale Einschließung von $y_1/\sqrt{3}$, dann gilt: Der untere rechte Eckpunkt liegt oberhalb der Extremalkurve $y = \sqrt{3} \cdot x$ $\iff$ `Inf(Uy1)` $\ge x_2$.

Anmerkungen:

1. nacheinander bedeutet, dass z.B. die 3. Abfrage nur dann erfolgen darf, wenn vorher die 1. und 2. Abfrage jeweils negativ ausgefallen sind.

2. Die Schnittpunkte der zweidimensionalen Maschinenintervalle Z_ν mit den Extremalkurven $y = \sqrt{3}\cdot x$, $y = x/\sqrt{3}$ können wegen der irrationalen Zahl $\sqrt{3}$ keine Maschinenzahlen sein.

B.3.2. Minimumbestimmung

Zur Bestimmung der Lage von m sind nacheinander folgende Fälle zu unterscheiden:

1. Wenn die linke Parallele zu y-Achse links der Extremalkurve $y = \sqrt{3}x$ liegt, so ist die Minimumstelle m die linke untere Ecke, vgl. Z_1, Z_2, Z_8, Z_9.
   Sei `Uy1` die optimale Einschließung von $y_1/\sqrt{3}$, dann gilt: Die linke Parallele zu y-Achse liegt links der Extremalkurve $y = \sqrt{3}x \iff x_1 \le$ `Inf(Uy1)`.

2. Wenn die linke Parallele zu y-Achse die Extremalkurve $y = \sqrt{3}x$ schneidet, so ist dieser Schnittpunkt gleich m, vgl. $Z_3, Z_{10}, Z_{11}, Z_{12}, Z_{14}, Z_{15}$.
   Sei `Uy1` die optimale Einschließung von $y_1/\sqrt{3}$ und `Uy2` die optimale Einschließung von $y_2/\sqrt{3}$, dann gilt: Die linke Parallele zu y-Achse schneidet die Extremalkurve $y = \sqrt{3}x$ $\iff$ `Sup(Uy1)` $\le x_1 \le$ `Inf(Uy2)`.
   Es gilt $m(x_S, y_S) = m(x_1, \sqrt{3}\cdot x_1)$, und für den minimalen Funktionswert erhält man: $f(x_S, y_S) = (x_1^2 - 3x_1^2)/(x_1^2 + 3x_1^2)^2 = -1/(8x_1^2) = -(1/(\sqrt{8}\cdot x_1))^2$.

3. Wenn die obere Parallele zu x-Achse die Extremalkurve $y = x/\sqrt{3}$ schneidet, so ist m der obere rechte oder linke Eckpunkt, vgl. Z_4.
   Sei `Ux1` die optimale Einschließung von $x_1/\sqrt{3}$ und `Ux2` die optimale Einschließung von $x_2/\sqrt{3}$, dann gilt: Die obere Parallele zu x-Achse schneidet die Extremalkurve $y = x/\sqrt{3}$ $\iff$ `Sup(Ux1)` $\le y_2 \le$ `Inf(Ux2)`.

4. Wenn der obere linke Eckpunkt oberhalb der Extremalkurve $y = x/\sqrt{3}$ liegt, so ist dieser obere linke Eckpunkt gleich m, vgl. Z_5, Z_6, Z_{13}.
   Sei `Ux1` die optimale Einschließung von $x_1/\sqrt{3}$, dann gilt: Der obere linke Eckpunkt liegt oberhalb der Extremalkurve $y = x/\sqrt{3} \iff$ `Sup(Ux1)` $\le y_2$.

5. Wenn der obere linke Eckpunkt unterhalb der Extremalkurve $y = x/\sqrt{3}$ liegt, so ist m der obere rechte Eckpunkt, vgl. Z_7.
   Sei `Ux1` die optimale Einschließung von $x_1/\sqrt{3}$, dann gilt: Der obere linke Eckpunkt liegt unterhalb der Extremalkurve $y = x/\sqrt{3} \iff y_2 \le$ `Inf(Ux1)`.

Weitere Einzelheiten zum Algorithmus findet man in der Funktion `Re_rz2(X,Y)`, die in der Datei `mpficlass.cpp` definiert ist.

B.3.3. Numerische Beispiele

Im **1. Beispiel** wählen wir $X = [1, 4]$, $Y = [1.5, 2]$, womit die typische Lage des zweidimensionalen Intervalls Z_{10} realisiert wird. Mit dem Funktionsaufruf `T = Re_rz2(X,Y);` erhalten wir mit `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [-1/8, +1/18]$ die nahezu optimale Einschließung $W_f \subseteq$ `T`.

$$\texttt{T} = [-0.12500000\ldots00001, +5.55555\ldots55556\cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen. Rechnet man z.B. mit `prec` = 3 000 000, so erhält man das analoge Ergebnis mit $3\,000\,000\cdot\log_{10}(2) \approx 903\,090$ Dezimalstellen.

Im **2. Beispiel** wählen wir $X = [0,1],\ Y = [2,3]$, womit die typische Lage des zweidimensionalen Intervalls Z_1 realisiert wird. Mit dem Funktionsaufruf `T = Re_rz2(X,Y);` erhalten wir mit `prec` = 300 für den Wertebereich $W_f := \{f(x,y)\,|\,x \in X,\ y \in Y\} = [-1/4, -2/25]$ die nahezu optimale Einschließung $W_f \subseteq$ `T`.

$$\texttt{T} = [-2.500000\ldots000001 \cdot 10^{-1}, -7.999999\ldots999999 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

In der folgenden Tabelle sind für alle typischen, zweidimensionalen Intervalle Z_ν ihre möglichen reellen Intervalle X_ν, Y_ν zusammen mit den jeweiligen exakten Funktionswerten f_m, f_M an den Minimum- bzw. Maximumstellen m, M zusammengestellt:

ν	X_ν	Y_ν	f_m	f_M	ν	X_ν	Y_ν	f_m	f_M
1	$[0,1]$	$[2,3]$	$-1/4$	$-2/25$	2	$[1,2]$	$[3,4]$	$-2/25$	$-5/169$
3	$[1,2]$	$[1.5,2]$	$-1/8$	$28/625$	4	$[2,3]$	$[1,1.5]$	$28/625$	$3/25$
5	$[4,5]$	$[4.5]$	$-9/1681$	$+9/1681$	6	$[4,5]$	$[2.6,3]$	$7/625$	$25/1352$
7	$[1,2]$	$[0.25,0.5]$	$60/289$	$240/289$	8	$[3,4]$	$[6,6.5]$	$-27/2025$	$-5/676$
9	$[1,3.5]$	$[2,2.5]$	$-3/25$	$1/32$	10	$[1,4]$	$[1.5,2]$	$-1/8$	$1/18$
11	$[1,2]$	$[1,4]$	$-1/8$	$+1/8$	12	$[2,3]$	$[1,4]$	$-1/32$	$3/25$
13	$[4,5]$	$[2,3]$	$7/625$	$3/100$	14	$[1,1.75]$	$[1,2]$	$-1/8$	$+1/8$
15	$[2,3]$	$[1,6]$	$-1/32$	$3/25$					

Der Algorithmus zur Berechnung der gesuchten Extrema f_m, f_M wurde u.a. mit allen obigen $Z_\nu := \{(x,y)\,|\,x \in X_\nu \wedge y \in Y_\nu\}$ getestet.

B.4. $2xy/(x^2+y^2)^2$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass mIm_rz2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x,y) = \frac{2xy}{(x^2+y^2)^2} \quad \text{mit:} \quad x \in \mathtt{X} = [x_1, x_2],\ y \in \mathtt{Y} = [y_1, y_2] \quad \text{und} \quad x \neq 0 \wedge y \neq 0.$$

Die Berechnung von $f(x,y)$ wird ausführlich beschrieben auf Seite 187. $f(x,y)$ ist als negative Imaginärteilfunktion der holomorphen Funktion $1/z^2,\ z \in \mathbb{C}\backslash\{0\}$, eine harmonische Funktion, wenn der singuläre Punkt $P(0,0)$ nicht ein Element des zweidimensionalen reellen Intervalls Z ist.

$$Z := \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad 0 \notin \mathtt{X} \wedge 0 \notin \mathtt{Y}. \tag{B.11}$$

$f(x,y)$ nimmt daher für alle Punkte $(x,y) \in Z$ als harmonische Funktion ihre Extrema auf dem Rand von Z an, vgl. dazu die Seite 244. Um die zugehörigen Extremstellen, m für das Minimum und M für das Maximum, möglichst einfach bestimmen zu können, betrachten wir zunächst die folgenden Symmetrieeigenschaften:

$$f(x,y) \equiv f(y,x), \quad \text{Spiegelung an der 1. Winkelhalbierenden,} \tag{B.12}$$

$$f(-x,-y) \equiv f(y,x), \quad \text{Spiegelung am Ursprung,} \tag{B.13}$$

$$f(-x,y) \equiv -f(x,y), \quad \text{Spiegelung an der } y\text{-Achse.} \tag{B.14}$$

Für Rechteckintervalle Z ergeben sich aus (B.12) und (B.13) die folgenden Eigenschaften:

1. Ist $Z_W := \left\{(x,y) \,\middle|\, x \in \mathtt{Y} \wedge y \in \mathtt{X}\right\}$ das an der 1. Winkelhalbierenden gespiegelte Rechteck, so nimmt $f(x,y)$ über den Rechteckintervallen Z_W und Z genau die gleichen Funktionswerte an, d.h. die Wertebereiche über Z_W und Z sind identisch.

2. Bedeutet $Z_U := \left\{(x,y) \,\middle|\, x \in -\mathtt{X} \wedge y \in -\mathtt{Y}\right\}$ das am Ursprung gespiegelte Rechteck, so nimmt $f(x,y)$ über den Rechteckintervallen Z_U und Z genau die gleichen Funktionswerte an.

Mit Hilfe dieser beiden Spiegelungen kann jetzt ein Rechteckintervall $Z = \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}$, das teilweise oder ganz in der unteren Halbebene liegt, ganz in die obere Halbebene so transformiert werden, dass die Wertebereiche von $f(x,y)$ über Z und dem transformierten Rechteck Z_T wieder identisch sind. Z_T ergibt sich aus

```
if (y2<=0) { X=-X;  Y=-Y; }
else
  if (y1<0)
     swap(X,Y);
     if (y1<0) {X=-X;  Y=-Y; }
```

Anmerkungen:

1. Nach dieser Transformation liegt $Z_T = \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}$ ganz in der oberen Halbebene, und im Fall $x_1 >= 0$ liegt Z_T sogar ganz im ersten Quadranten, so dass dann die Bestimmung von m, M auf den 1. Quadranten beschränkt werden kann. Im Fall $y_2 > x_2$ werden X und Y nochmals vertauscht, so dass dann im ersten Quadranten nur noch Rechtecke zu betrachten sind, deren obere rechte Eckpunkte stets unterhalb der 1. Winkelhalbierenden liegen. Hierdurch wird die Anzahl der notwendigen Fallunterscheidungen nochmals deutlich reduziert.

2. Im Fall $x_2 <= 0$ liegt Z_T ganz im 2. Quadranten, und mit der zusätzlichen Transformation $X = -X$ liegt dann das neue Z_{T1} ganz im 1. Quadranten, nach (B.14) jedoch mit den Funktionswerten $-f(x,y)$. Ist dann $[f_m, f_M]$ der berechnete Wertebereich von $f(x,y)$ über Z_{T1}, so ist $-[f_m, f_M] = [-f_M, -f_m]$ der gesuchte Wertebereich von $f(x,y)$ über Z_T.

3. Im Fall $x_1 < 0 \wedge x_2 > 0$ wird mit $X_2 := [x_1, 0]$ und $X_1 := [0, x_2]$ das Rechteck Z_T aufgeteilt in $Z_{T2} = \{(x,y) \,|\, x \in X_2 \wedge y \in Y\}$ und $Z_{T1} = \{(x,y) \,|\, x \in X_1 \wedge y \in Y\}$, so dass in Z_{T2} wegen $f(x,y) \le 0$ nur der Minimumpunkt m und in Z_{T1} wegen $f(x,y) \ge 0$ nur der Maximumpunkt M liegen kann. Daher wird Z_{T1} wie unter Punkt 1. und Z_{T2} wie unter Punkt 2. behandelt.

Zusammenfassung:

Um zu einem gegebenen Rechteckintervall $Z \not\ni (0,0)$ den Wertebereich $[f_m, f_M]$ von $f(x,y)$ zu bestimmen, wird Z zunächst nach Seite 216 als Z_T in die obere Halbebene transformiert. Nach den Punkten 1,2,3 von Seite 216 müssen dann die Extremwerte f_m und f_M nur noch für solche Rechteckintervalle aus dem 1. Quadranten bestimmt werden, deren obere rechte Ecken unterhalb der 1. Winkelhalbierenden liegen.

Die Berechnung der gerundeten Extrema f_m bzw. f_M für ein Rechteckintervall Z, nur aus dem 1. Quadranten mit seinem oberen rechten Eckpunkt unterhalb der 1. Winkelhalbierenden, erfolgt mit den beiden Funktionen

```
MpfrClass mIm_rz2_m (const MpfiClass& X, const MpfiClass& Y);
MpfrClass mIm_rz2_M (const MpfiClass& X, const MpfiClass& Y);
```

die in der Datei `mpficlass.cpp` definiert sind. Zur Berechnung der beiden Extremstellen m, M benötigen wir zunächst die partiellen Ableitungen:

$$\frac{\partial f(x,y)}{\partial x} = \frac{2y \cdot (-3x^2 + y^2)}{(x^2 + y^2)^3} \tag{B.15}$$

$$\frac{\partial f(x,y)}{\partial y} = \frac{2x \cdot (x^2 - 3y^2)}{(x^2 + y^2)^3}. \tag{B.16}$$

Die Extremalkurven für $\partial f/\partial x$ ergeben sich aus der Forderung $\partial f/\partial x = 0$, siehe Seite 247. Die erste Extremalkurve ist wegen $y = 0$ die positive x-Achse, und die zweite ergibt sich aus der Forderung $y^2 = 3x^2$ aus der für den 1. Quadranten $y = \sqrt{3} \cdot x$ folgt.

Die beiden Extremalkurven für $\partial f/\partial y$ ergeben sich analog aus der Forderung $\partial f/\partial y = 0$. Die erste Extremalkurve ist dann wegen $x = 0$ die positive y-Achse und die zweite Extremalkurve ist gegeben durch $y = x/\sqrt{3}$.

Da beide partiellen Ableitungen auf den Extremalkurven $y = \sqrt{3} \cdot x$ und $y = x/\sqrt{3}$ das Vorzeichen wechseln, wenn man parallel zu den beiden Koordinatenachsen diese Extremalkurven schneidet, besitzt $f(x,y)$ an den Schnittpunkten der Intervallränder von Z mit den Extremalkurven längs dieser Intervallränder tatsächlich relative Extrema, d.h. in diese Fällen müssen die Extremalpunkte m oder M nicht auf den Eckpunkten von Z liegen.

In den folgenden Abbildungen sind alle typischen Lagen der Intervalle Z_ν aus dem 1. Quadranten mit den entsprechenden Lagen der Extremstellen $m(x,y), M(x,y)$ angegeben. Dabei bedeutet das Symbol (M), dass hier zwar ein relatives Maximum vorliegt, dass aber das wirkliche Maximum an der anderen Stelle M angenommen wird. Die zwei Symbole $m?$ bedeuten, dass das wirkliche Minimum nur an einer dieser beiden Stellen angenommen werden kann, so dass zwei Berechnungen und ein Vergleich erforderlich sind. Die 1. Winkelhalbierende und die beiden Extremalkurven $y = \sqrt{3} \cdot x$ und $y = x/\sqrt{3}$ sind leicht gedreht dargestellt. Die farbigen Pfeile geben die Richtung wachsender Funktionswerte von $f(x,y) = +2xy/(x^2 + y^2)^2$ an.

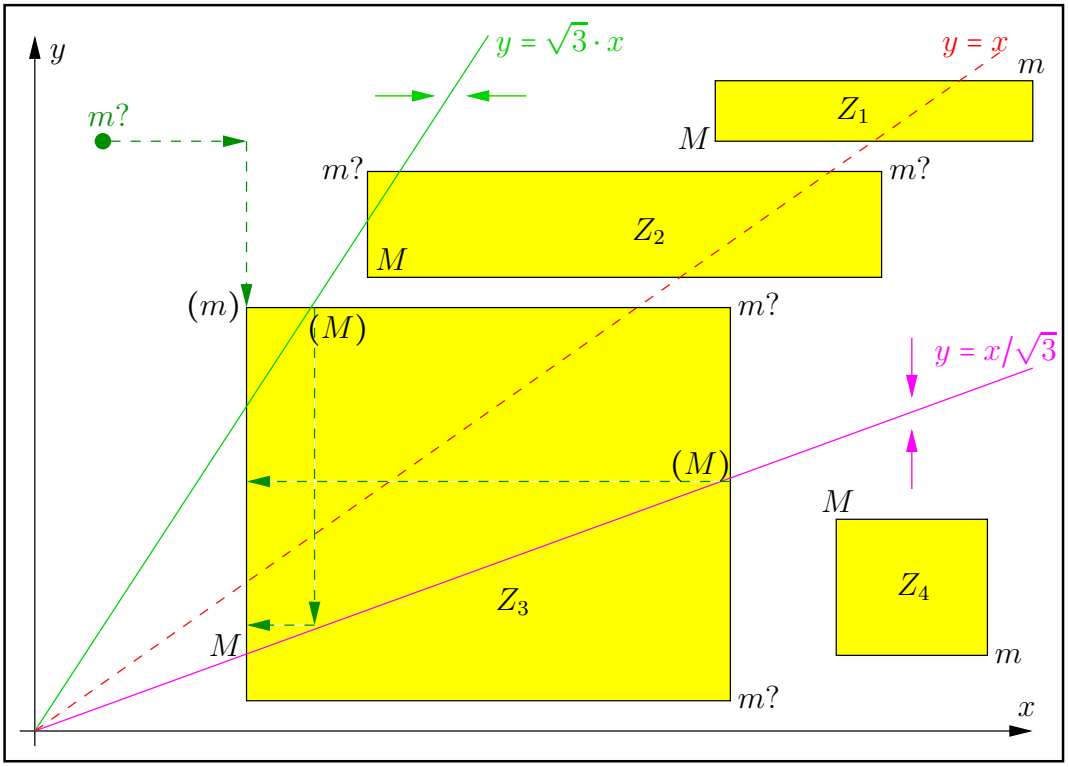


Abbildung B.8.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Anmerkungen:

1. Die 1. Winkelhalbierende und die Extremalkurven $y = \sqrt{3} \cdot x$ bzw. $y = x/\sqrt{3}$ sind etwas gedreht dargestellt, so dass z.B. der an der 1. Winkelhalbierenden gespiegelte untere rechte Eckpunkt $m?$ durch $m?$ nicht ganz maßstabsgetreu dargestellt wird.

2. Bei den Rechtecken Z_1, Z_2, Z_4 liegen die Extremalpunkte m, M auf den Eckpunkten dieser Rechtecke, wobei bez. Z_2 der Minimumpunkt m einer der beiden oberen Eckpunkte ist.

3. Wir zeigen zuerst, dass am oberen linken Eckpunkt (m) von Z_3 die Funktion $f(x, y)$ nicht ihr Minimum annehmen kann. Dazu spiegeln wir den unteren rechten Eckpunkt $m?$ an der 1. Winkelhalbierenden und erhalten den Spiegelpunkt $m?$. Mit $m?(x_2, y_1)$ gilt deshalb $m?(y_1, x_2)$, und nach (B.12) gilt $f_{m?} := f(x_2, y_1) = f_{m?} := f(y_1, x_2)$. Im nächsten Schritt zeigen wir, dass der Spiegelpunkt $m?$ links oberhalb von $(m)(x_1, y_2)$ liegt, d.h. zu zeigen ist $y_1 < x_1$ und $y_2 < x_2$. Dazu benutzen wir die für die Lage von Z_3 gültigen Bedingungen:

$$y_1 < x_1/\sqrt{3} < x_1 \qquad \text{und} \qquad y_2 < x_2,$$

aus denen die Behauptung unmittelbar folgt. In Abb. (B.8) wird mit den Pfeilen `----->` gezeigt, wie man von $m?$ nach (m) kommt, und weil `--->` die Richtung wachsender Funktionswerte angibt, gilt $f_{m?} = f_{m?} < f_{(m)}$, so dass (m) nicht der gesuchte Minimumpunkt m sein kann ∎

4. Wir zeigen jetzt, dass der rechte Schnittpunkt (M) mit der Extremalkurve $y = x/\sqrt{3}$ nicht der gesuchte Maximumpunkt M sein kann. Dazu bewegen wir uns zunächst von (M) aus in Richtung des Pfeils `<---` bis zum Schnittpunkt mit der linken Parallelen zur y-Achse von Z_3 und von dort aus weiter in Richtung wachsender Funktionswerte nach unten bis M, so dass (M) nicht der Maximumpunkt sein kann ∎ Ganz analog zeigt man, dass auch der obere Schnittpunkt (M) mit der Extremalkurve $y = \sqrt{3} \cdot x$ kein Maximumpunkt sein kann.

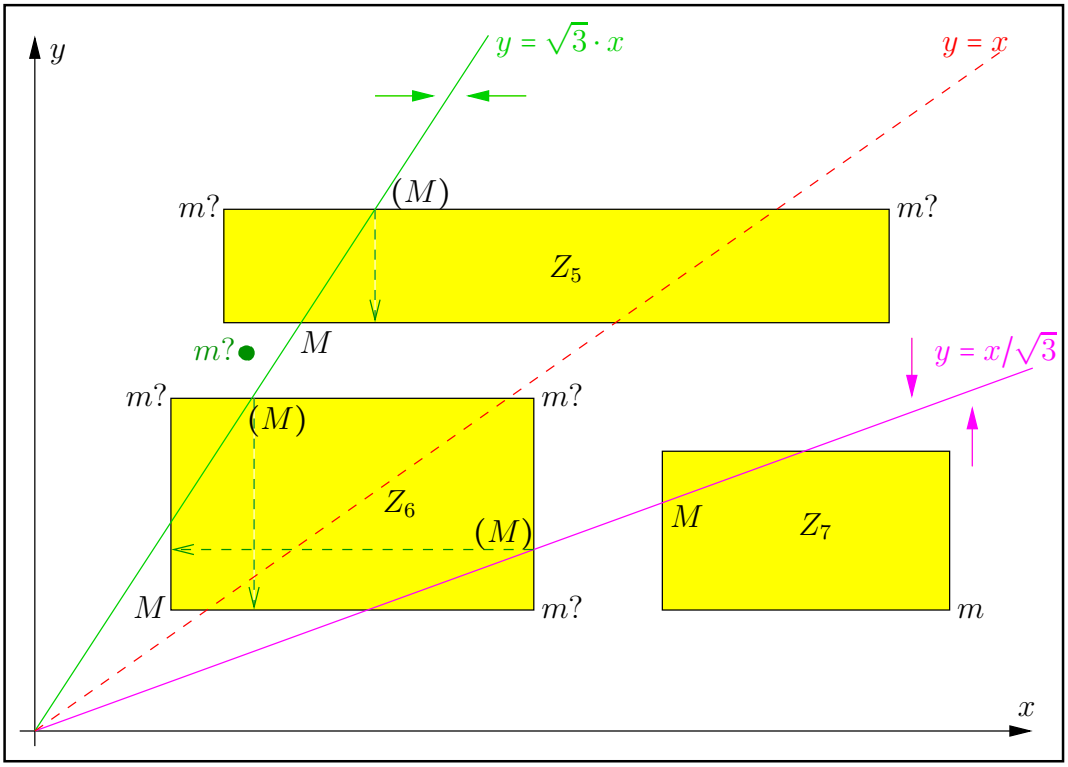


Abbildung B.9.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Da in Z_6 wegen $y_1 > x_1$, im Gegensatz zu Z_3 aus Abb. B.8, der Spiegelpunkt $m?(y_1, x_2)$ rechts von $m?(x_1, y_2)$ liegt, kann $m?(x_1, y_2)$ in Z_6 nicht als Minimumpunkt ausgeschlossen werden.

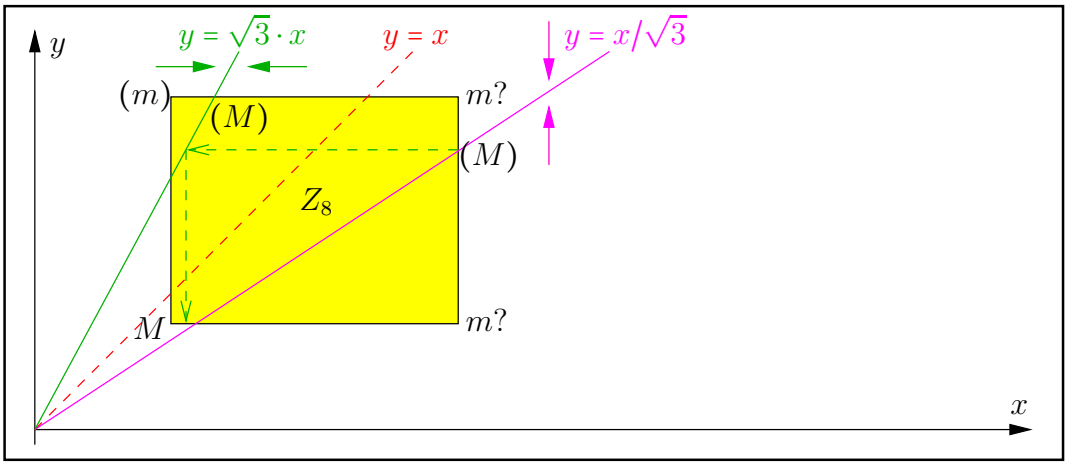


Abbildung B.10.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

In Z_8 liegen jetzt bez. des Minimumpunktes m die gleichen Verhältnisse vor wie in Z_3 aus Abb. (B.8), da wegen $y_1 < x_1$ und $y_2 < x_2$ der Spiegelpunkt $m?(y_1, x_2)$ wieder links oberhalb von $(m)(x1, y2)$ liegt, so dass (m) auch jetzt kein Minimumpunkt sein kann. Ganz analog zeigt man auch jetzt mit Hilfe der Pfeile `<----`, welche die Richtung wachsender Funktionswerte angeben, dass die beiden Punkte (M) keine Maximumpunkte sein können.

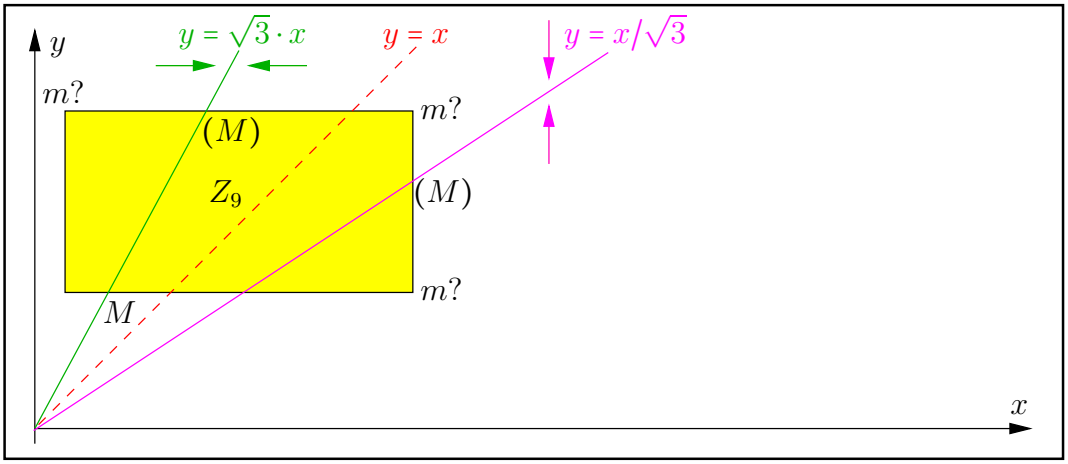


Abbildung B.11.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Da in Z_9 wegen $y_1 > x_1$, im Gegensatz zu Z_3 aus Abb. B.8, der Spiegelpunkt $m?(y_1, x_2)$ rechts von $m?(x_1, y_2)$ liegt, kann $m?(x_1, y_2)$ in Z_9 nicht als Minimumpunkt m ausgeschlossen werden. Damit sind für m drei Berechnungen und zwei Vergleiche notwendig. Ganz analog zeigt man auch jetzt mit Hilfe der Pfeile <----, welche die Richtung wachsender Funktionswerte angeben, dass die beiden Punkte (M) keine Maximumpunkte sein können.

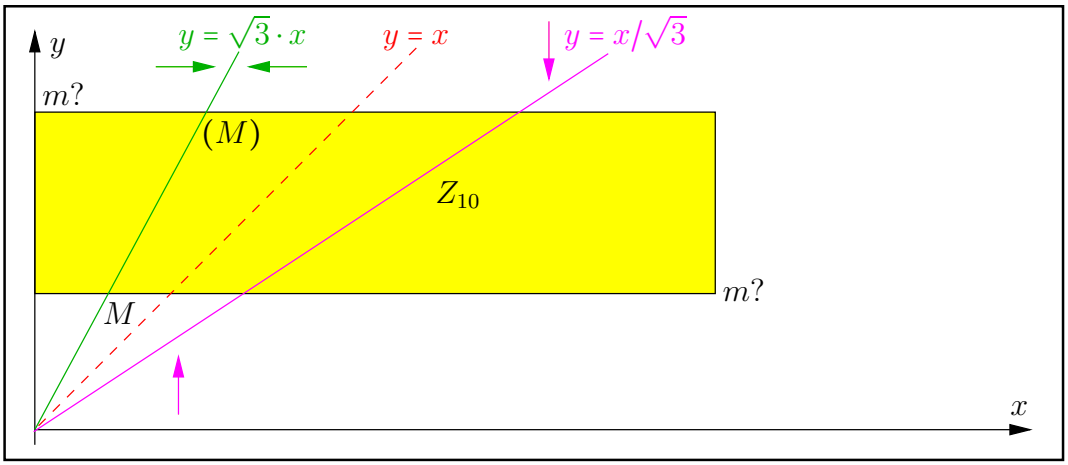


Abbildung B.12.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Da in Z_{10} wegen $y_1 > x_1$, im Gegensatz zu Z_3 aus Abb. B.8, der Spiegelpunkt $m?(y_1, x_2)$ von $m?(x_2, y_1)$ rechts von $m?(x_1, y_2)$ liegt, kann $m?(x_1, y_2)$ in Z_{10} nicht als Minimumpunkt m ausgeschlossen werden. Damit sind für m zwei Berechnungen und ein Vergleiche notwendig. Ganz analog zeigt man auch jetzt mit Hilfe des Pfeils <----, welcher die Richtung wachsender Funktionswerte angibt, dass der Punkte (M) mit seinem relativen Maximum nicht der Punkt M mit dem gesuchten absoluten Maximum auf dem Rand von Z_{10} sein kann.

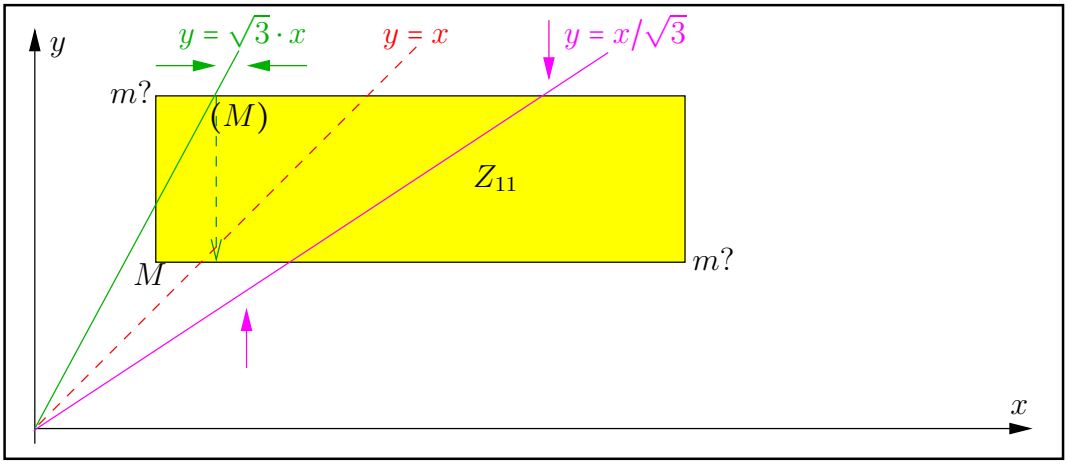


Abbildung B.13.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

Da in Z_{11} wegen $y_1 > x_1$, im Gegensatz zu Z_3 aus Abb. B.8, der Spiegelpunkt $m?(y_1, x_2)$ von $m?(x_2, y_1)$ rechts von $m?(x_1, y_2)$ liegt, kann $m?(x_1, y_2)$ in Z_{11} nicht als Minimumpunkt m ausgeschlossen werden. Damit sind für m zwei Berechnungen und ein Vergleiche notwendig. Ganz analog zeigt man auch jetzt mit Hilfe des Pfeils <----, welcher die Richtung wachsender Funktionswerte angibt, dass der Punkte (M) mit seinem relativen Maximum nicht der Punkt M mit dem gesuchten absoluten Maximum auf dem Rand von Z_{11} sein kann.

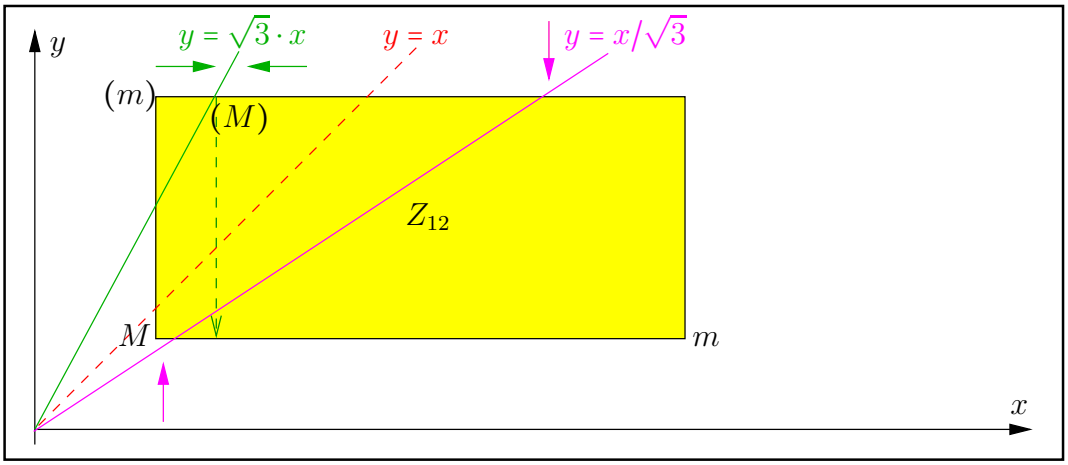


Abbildung B.14.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

In Z_{12} liegen jetzt bez. des Minimumpunktes m die gleichen Verhältnisse vor wie in Z_3 aus Abb. (B.8), da wegen $y_1 < x_1$ und $y_2 < x_2$ der Spiegelpunkt $m(y_1, x_2)$ von $m(x_2, y_1)$ wieder links oberhalb von $(m)(x1, y2)$ liegt, so dass (m) auch jetzt kein Minimumpunkt sein kann. Ganz analog zeigt man auch jetzt mit Hilfe des Pfeils <----, welcher die Richtung wachsender Funktionswerte angibt, dass der Punkte (M) mit seinem relativen Maximum nicht der Punkt M mit dem gesuchten absoluten Maximum auf dem Rand von Z_{12} sein kann.

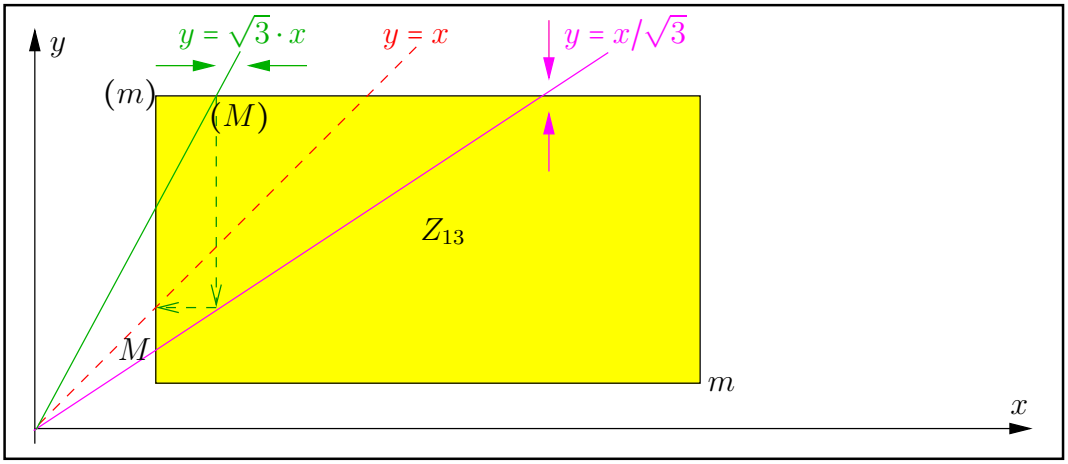


Abbildung B.15.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

In Z_{13} liegen jetzt bez. des Minimumpunktes m die gleichen Verhältnisse vor wie in Z_3 aus Abb. (B.8), da wegen $y_1 < x_1$ und $y_2 < x_2$ der Spiegelpunkt $m(y_1, x_2)$ von $m(x_2, y_1)$ wieder links oberhalb von $(m)(x1, y2)$ liegt, so dass (m) auch jetzt kein Minimumpunkt sein kann. Ganz analog zeigt man auch jetzt mit Hilfe der Pfeile <----, welche die Richtung wachsender Funktionswerte angeben, dass der Punkte (M) mit seinem relativen Maximum nicht der Punkt M mit dem gesuchten absoluten Maximum auf dem Rand von Z_{13} sein kann.

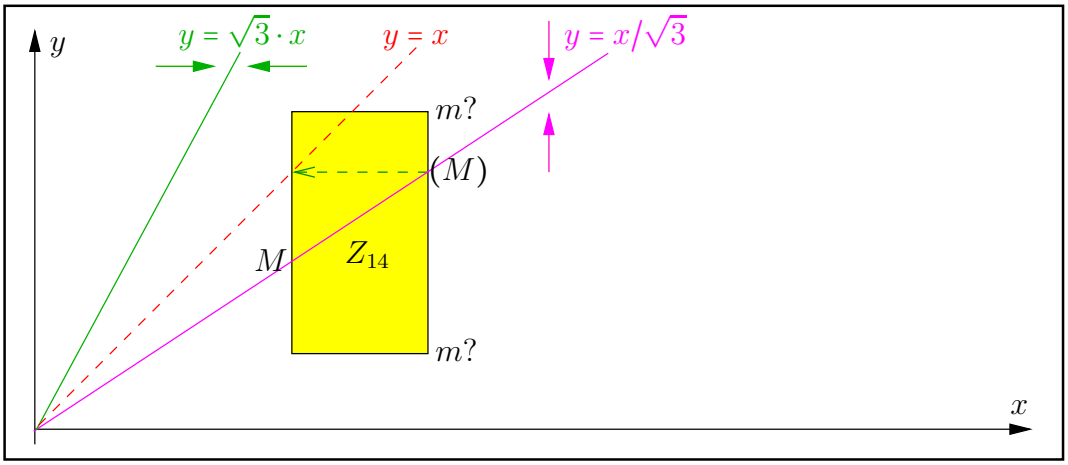


Abbildung B.16.: Typische Lagen der Intervalle Z_ν mit den Extremstellen m, M

In Z_{14} ist der Minimumpunkt der echte obere oder untere Eckpunkt, so dass zwei Berechnungen und ein Vergleich erforderlich sind. (M) kann nicht der Maximumpunkt sein, denn läuft man von (M) aus in Richtung des Pfeils <----, der die Richtung wachsender Funktionswerte angibt, so kommt man nur auf Wegen mit wachsenden Funktionswerten zum Punkt M.

In den Abbildungen B.8 bis B.16 sind alle typischen Lagen der Rechteckintervalle Z_ν, mit $\nu = 1, 2, \ldots, 14$, und $y_2 \le x_2$ zusammengestellt, wobei der rechte obere Eckpunkt eines Z_ν nicht oberhalb der 1. Winkelhalbierenden liegt.

B.4.1. Maximumbestimmung

Es gelten die folgenden äquivalenten Aussagen:

1. Der untere linke Eckpunkt $C(x_1, y_1)$ liegt zwischen den Extremalkurven $y = \sqrt{3} \cdot x$ und $y = x/\sqrt{3} \iff x_1/\sqrt{3} < y_1 < x_1 \cdot \sqrt{3} \iff M = C(x_1, y_1)$, vgl. die Rechtecke $Z_1, Z_2, Z_6, Z_8, Z_{11}, Z_{12}$.

2. S ist Schnittpunkt der linken Rechteck-Parallelen zur y-Achse mit der Extremalkurve $y = x/\sqrt{3} \iff y_1 < x_1/\sqrt{3} < y_2 \iff M = S(x_1, x_1/\sqrt{3})$, vgl. Z_3, Z_7, Z_{13}, Z_{14}.

$$f_M := f(x_1, x_1/\sqrt{3}) = \begin{cases} \left(\frac{1}{x_1}\right)^2 / \frac{8}{\sqrt{27}}, & x_1 > 1 \\ \left(\frac{1}{2x_1}\right)^2 / \frac{2}{\sqrt{27}}, & x_1 < 1. \end{cases}$$

3. Die untere Parallele zur x-Achse schneidet die Gerade $y = \sqrt{3} \cdot x \iff x_1 < y_1/\sqrt{3} < x_2 \iff M = S(y_1/\sqrt{3}, y_1)$, vgl. die Rechtecke Z_5, Z_9, Z_{10}.
   Anmerkung: Die Schnittpunktbedingung $x_1 < y_1/\sqrt{3} < x_2$ kann reduziert werden auf $x_1 < y_1/\sqrt{3}$, da nach Voraussetzung gilt: $y_1/\sqrt{3} < y_1 \le y_2 \le x_2$.
   $f_M := f(y_1/\sqrt{3}, y_1)$ kann nach (B.12) wie unter Punkt 2. berechnet werden, wenn man x_1 durch y_1 ersetzt.

4. Der obere linke Eckpunkt $C(x_1, y_2)$ liegt unterhalb der Extremalkurve $y = x/\sqrt{3} \iff y_2 < x_1/\sqrt{3} \iff M = C(x_1, y_2)$, vgl. Z_4.

Daraus ergibt sich der folgende Algorithmus zur Bestimmung des Maximumpunktes M:

```
if (y2 < x1/sqrt(3))
   M = C;  // nach 4.
else  // x1/sqrt(3) < y2
{
  if (y1 < x1/sqrt(3))
     M = S;  // nach 2.
  else // y1 > x1/sqrt(3),  x1/sqrt(3) < y2;
  {
    if (y1/sqrt(3) < x1)
       M = C;  // nach 1.
    else // sqrt(3)*x1 < y1,  y1 > x1/sqrt(3),  x1/sqrt(3) < y2;
       M = S;  // nach 3.
  }
}
```

Mit dem obigen recht einfachen Algorithmus, der alle typischen Lagen der vierzehn Rechtecke Z_ν zusammenfasst, wird der maximale Funktionswert über dem Punkt M berechnet mit Hilfe der Funktion

```
MpfrClass mIm_rz2_M (const MpfiClass& x, const MpfiClass& y),
```

die in `mpficlass.cpp` definiert ist. Der Rückgabewert ist der nahezu optimal aufgerundete, maximale Funktionswert von $f(x, y)$ über dem Rechteck Z_ν, dessen oberer rechter Eckpunkt unterhalb der 1. Winkelhalbierenden liegen muss. Diese letzte Bedingung wird innerhalb der Funktion `mIm_rz2_M(...)` jedoch nicht überprüft und muss damit vorher abgesichert werden. Beachten Sie, dass bei der internen Auswertung von $f(x, y)$ ein vorzeitiger Überlauf durch geeignete Skalierungen verhindert wird.

B.4.2. Minimumbestimmung

Aus den Abbildungen B.8 bis B.16 ergibt sich, dass die Minimumpunkte m nur auf den Eckpunkten der Z_ν liegen, da die Funktionswerte $f(x,y)$ nur anwachsen, wenn man sich parallel zu den Koordinatenachsen auf die Extremalkurven $y = \sqrt{3}\cdot x$ oder $y = x/\sqrt{3}$ zubewegt. Um die Lage der Minimumpunkte m möglichst einfach, d.h. mit einem Minimum an Abfragen, bestimmen zu können, betrachten wir nacheinander die folgenden drei Fälle:

1. Der untere rechte Eckpunkt $C(x_2, y_1)$ liegt oberhalb $y = x/\sqrt{3}$, d.h. $\quad x_2/\sqrt{3} < y_1$.
2. Der obere rechte Eckpunkt $C(x_2, y_2)$ liegt unterhalb $y = x/\sqrt{3}$, d.h. $\quad y_2 < x_2/\sqrt{3}$.
3. Die rechte Parallele zur y-Achse schneidet die Extremalkurve $y = x/\sqrt{3}$. Dieser Fall tritt ein, wenn die Bedingungen aus 1. und 2. nicht erfüllt sind.

Die oberen drei Fälle werden jetzt genauer betrachtet:

1. Zu diesem Fall gehören die Rechtecke Z_1, Z_2, Z_5.
   Wenn zusätzlich der obere linke Eckpunkt $C(x_1, y_2)$ oberhalb von $y = \sqrt{3}\cdot x$ liegt, d.h. wenn gilt $y_2/\sqrt{3} > x_1$, dann können $C(x_2, y_2)$ oder auch $C(x_1, y_2)$ die Minimumpunkte sein, vgl. Z_2, Z_5, sonst ist nur $C(x_2, y_2)$ der Minimumpunkt, vgl Z_1.

2. Zu diesem Fall gehören die Rechtecke $Z_4, Z_7, Z_{10}, Z_{11}, Z_{12}, Z_{13}$.
   Für die Rechtecke Z_4, Z_7, Z_{12}, Z_{13} gilt: $\quad m = C(x_2, y_1)$. Wenn jedoch der Spiegelpunkt $C(y_1, x_2)$ von $C(x_2, y_1)$ rechts oberhalb von $C(x_1, y_2$ liegt, d.h. wenn gilt $y_1 > x_1 \wedge x_2 \geq y_2$, so kann auch der obere linke Eckpunkt $C(x_1, y_2)$ ein Minimum sein, vgl. Z_{10}, Z_{11}. Beachten Sie, dass die zweite Bedingung $y_2 \leq x_2$ automatisch erfüllt ist, da vorausgesetzt wird, dass der obere rechte Eckpunkt $C(x_2, y_2)$ unterhalb der 1. Winkelhalbierenden liegt. Außerdem ist zu beachten, dass die erste Bedingung $y_1 > x_1$ für die Rechtecke Z_4, Z_7, Z_{12}, Z_{13} nicht erfüllt ist und damit nur für Z_{10}, Z_{11} zutrifft.

3. Zu diesem letzten Fall gehören die Rechtecke $Z_3, Z_6, Z_8, Z_9, Z_{14}$.
   Wenn also die rechte Parallele zur y-Achse die Extremalkurve $y = x/\sqrt{3}$ schneidet, dann ist einer der beiden Eckpunkte $C(x_2, y_1)$ oder $C(x_2, y_2)$ gleich m. Wenn dann zusätzlich der Spiegelpunkt $C(y_1, x_2)$ von $C(x_2, y_1)$ rechts oberhalb von $C(x_1, y_2)$ liegt, d.h. wenn gilt $y_1 > x_1 \wedge y_2 \leq x_2$, so kann auch $C(x_1, y_2)$ der Minimumpunkt m sein. Beachten Sie bitte, dass die letzte Bedingung $y_2 \leq x_2$ entfallen kann, da sie nach Voraussetzung erfüllt ist, denn der rechte obere Eckpunkt $C(x_2, y_2)$ soll unterhalb der 1. Winkelhalbierenden liegen! Vergleichen Sie dazu die Rechtecke Z_3, Z_8, Z_{14} bzw. Z_6, Z_9. Beachten Sie außerdem, dass bei Z_{14} die Bedingung $y_1 > x_1$ nicht erfüllt sein kann, denn der linke untere Eckpunkt $C(x_1, y_1)$ soll nach Voraussetzung unterhalb von $y = x/\sqrt{3}$ liegen, d.h. es muss gelten $y_1 < x_1/\sqrt{3} < x_1$.

Mit Hilfe der obigen drei Hauptfälle werden alle typischen Lagen der vierzehn Rechtecke Z_ν beschrieben, und der minimale Funktionswert über dem Punkt m lässt sich berechnet mit der Funktion

```
MpfrClass mIm_rz2_m (const MpfiClass& x, const MpfiClass& y),
```

die in `mpficlass.cpp` definiert ist. Der Rückgabewert ist der nahezu optimal abgerundete, minimale Funktionswert von $f(x,y)$ über dem Rechteck Z_ν, dessen oberer rechter Eckpunkt unterhalb der 1. Winkelhalbierenden liegen muss. Diese letzte Bedingung wird innerhalb der Funktion `mIm_rz2_m(...)` jedoch nicht überprüft und muss damit vorher abgesichert werden. Beachten Sie, dass bei der internen Auswertung von $f(x,y)$ ein vorzeitiger Überlauf durch geeignete Skalierungen verhindert wird.

B.4.3. Numerische Beispiele

Im **1. Beispiel** wählen wir $X_1 = [5,8]$ und $Y_1 = [5,6]$, womit die typische Lage des zweidimensionalen Intervalls Z_1 realisiert wird. Mit dem Funktionsaufruf `T = mIm_rz2_m(X,Y);` erhalten wir mit `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [+6/625, +1/50]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [9.59999999\ldots99999 \cdot 10^{-3}, 2.00000\ldots00001 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen. Rechnet man z.B. mit `prec` = 3 000 000, so erhält man das analoge Ergebnis mit $3\,000\,000 \cdot \log_{10}(2) \approx 903\,090$ Dezimalstellen.

In der folgenden Tabelle sind für alle typischen, zweidimensionalen Intervalle Z_ν ihre möglichen reellen Intervalle X_ν, Y_ν zusammen mit den jeweiligen exakten Funktionswerten f_m, f_M an den Minimum- bzw. Maximumstellen m, M zusammengestellt:

ν	X_ν	Y_ν	f_m	f_M	ν	X_ν	Y_ν	f_m	f_M
1	$[5,8]$	$[5,6]$	$6/625$	$1/50$	2	$[4,6]$	$[3.5,4.5]$	$32/1875$	$448/12769$
3	$[2,5]$	$[1,4]$	$5/338$	$9/(32\sqrt{3})$	4	$[6,9]$	$[1,2]$	$9/3362$	$3/200$
5	$[2,6]$	$[4,5]$	$60/3721$	$9/(128\sqrt{3})$	6	$[2,6]$	$[3,5]$	$60/3721$	$12/169$
7	$[4,5]$	$[2,2.5]$	$20/841$	$9/(128\sqrt{3})$	8	$[2,4]$	$[1.5,3]$	$192/5329$	$96/625$
9	$[0.5,2]$	$[1,2]$	$32/289$	$9\sqrt{3}/24$	10	$[0,4]$	$[1,2]$	0	$9\sqrt{3}/24$
11	$[1,6]$	$[1.5,3]$	$32/2601$	$48/169$	12	$[2,7]$	$[1.5,4]$	$336/42025$	$96/625$
13	$[2,7]$	$[1,4]$	$14/2500$	$9\sqrt{3}/96$	14	$[2,5]$	$[1,3]$	$5/338$	$9\sqrt{3}/96$
15	$[1,4]$	$[2,7]$	$14/2500$	$9\sqrt{3}/96$	16	$[-7,-2]$	$[1,4]]$	$-9\sqrt{3}/96$	$-14/2500$
17	$[-4,4]$	$[1,2]$	$-9\sqrt{3}/96$	$+9\sqrt{3}/96$	18	$[-2,-0.5]$	$[-2,-1]$	$32/289$	$9\sqrt{3}/24$

Anmerkungen:

1. Die Rechtecke $Z_\nu,\ \nu = 1,2,\ldots,14$, liegen alle mit dem oberen rechten Eckpunkt nicht oberhalb der 1. Winkelhalbierenden, d.h. es gilt $y_2 \le x_2$.

2. Z_{15} entsteht aus Z_{13} durch Spiegelung an der 1. Winkelhalbierenden, so dass $f(x,y)$ über Z_{15} und Z_{13} den gleichen Wertebereich besitzt.

3. Z_{16} entsteht aus Z_{13} durch Multiplikation von X_{13} mit -1. Nach (B.14) ist dann der Wertebereich von $f(x,y)$ über Z_{16} der negative Wertebereich von $f(x,y)$ über Z_{13}.

4. Z_{17} entsteht aus Z_{10} durch Vereinigung von X_{10} mit $-X_{10}$, so dass das Minimum von $f(x,y)$ über Z_{17} das negative Maximum von X_{10} sein muss.

5. Z_{18} entsteht aus Z_9 durch Multiplikation von X_9 und Y_9 durch -1, so dass nach (B.13) $f(x,y)$ über Z_9 und Z_{18} den gleichen Wertebereich besitzt.

Der Algorithmus zur Berechnung der gesuchten Extrema f_m, f_M wurde u.a. mit allen obigen $Z_\nu := \left\{(x,y) \,\middle|\, x \in X_\nu \wedge y \in Y_\nu\right\},\ \nu = 1,2,\ldots,18$, getestet.

B.5. $x^2 + a \cdot x + b$

Für die gegebenen reellen Maschinenintervalle $\mathtt{X} = [x_1, x_2] \ni x$, $\mathtt{A} = [a_1, a_2] \ni a$ und $\mathtt{B} = [b_1, b_2] \ni b$ liefert die Funktion

```
MpfiClass poly2(const MpfiClass& X, const MpfiClass& A, const MpfiClass& B)
```

für

$$f(x, a, b) = x^2 + a \cdot x + b; \quad x, a, b \in \mathbb{R}$$

eine optimale Maschineneinschließung $f(X, A, B)$ der Wertemenge

$$W_{f,X,A,B} := \{f(x, a, b) \,|\, x \in \mathtt{X},\ a \in \mathtt{A},\ b \in \mathtt{B}\} \subset f(X, A, B),$$

wobei die **optimale** Einschließung nur garantiert ist, wenn $A = [a_1, a_2]$ ein Punktintervall[1] ist. Die optimale Einschließung ist gewährleistet, wenn $f(x, a, b)$ mit folgendem Ausdruck

$$f(x, a, b) = (x + a/2)^2 - (a/2)^2 + b \tag{B.17}$$

intervallmäßig ausgewertet wird, denn jetzt kommen beide Variablen x, b explizit nur einmal vor und weil a zweimal auftritt, muss $A = [a_1, a_2]$ ein Punktintervall sein. Um bei der Auswertung von B.17 einen vorzeitigen Über- oder Unterlauf zu vermeiden, wird in der Funktion

```
void poly2(MpfiClass& R, long int& k, const MpfiClass& X,
           const MpfiClass& A, const MpfiClass& B);
```

durch geeignete Skalierungen erreicht, dass das Intervall $2^k \cdot R$ die gewünschte optimale Einschließung $f(X, A, B)$ liefert, wobei $\mathtt{Sup}(|R|)$ nur wenig kleiner als `MaxFloat()` ausfällt. Bei der Berechnung von R werden mögliche Auslöschungen durch schrittweise Vergrößerungen der Präzision `prec` vermieden. Dabei wird `prec` so lange erhöht, bis sich die jeweiligen R-Werte nicht mehr ändern. Beim Vergleich dieser R-Werte müssen diese natürlich jeweils **vorher** nach außen in die Ausgangspräzision gerundet werden. Beachten Sie, dass die berechneten Einschließungen natürlich nicht optimal sein können, wenn mögliche Auslöschungen nicht vermieden werden.

B.5.1. Numerische Beispiele

Im **1. Beispiel** wählen wir die Intervalle $X = [-3, 2]$, $A = [-6, -6]$, $B = [-1, 2]$ und erhalten mit den Zwischenergebnissen

$$X + A/2 = [-6, -1],\ (X + A/2)^2 = [1, 36],\ (X + A/2)^2 - (A/2)^2 = [-8, 27]$$

das exakte Ergebnis $f(X, A, B) = W_{f,X,A,B} = [-9, 29]$, und genau diese optimale Einschließung liefert für $\mathtt{prec} \geq 5$ der Funktionsaufruf `F = poly2(X,A,B);` mit `F,X,A,B` vom Typ `MpfiClass`. Für $\mathtt{prec} = 4$ erhält man die gröbere Einschließung $\mathtt{F} = [-9, 30]$, da bei dieser Präzision das exakte Intervall $[-9, 29]$ nicht mehr darstellbar ist.

Im **2. Beispiel** wählen wir mit der Präzision $\mathtt{prec} = 300$ die nachfolgenden Maschinenintervalle $X = [\mathtt{pred}(1), \mathtt{pred}(1)] = [1 - 2^{-300}, 1 - 2^{-300}]$, $A = [0, 0]$, $B = [-1, -1]$ und werten damit die Funktion $f(x, 0, -1) = x^2 - 1$ in der unmittelbaren Umgebung ihrer Nullstelle $x_0 = +1$ aus, so dass erhebliche Auslöschungen durch eine hinreichend große interne Präzision zu vermeiden ist. Der Funktionsaufruf `F = poly2(X,A,B);` liefert die optimale Einschließung

$$\mathtt{T} = [-9.81818693059545310\ldots 19576808 \cdot 10^{-91}, -9.81818693059545310\ldots 19576325 \cdot 10^{-91}]$$

mit 93 ausgegebenen Dezimalziffern, vergleichen Sie dazu das entsprechende Ergebnis auf Seite 191, das mit einem ganz anderen Algorithmus ebenfalls optimal berechnet wurde.

[1] Gilt $a_2 = \mathtt{succ}(a_1)$, so ist die Einschließung nahezu optimal.

B.6. $(1+x^2-y^2)/(4x^2y^2+(1+x^2-y^2)^2)$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass Re_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x,y) = \frac{1+x^2-y^2}{(4x^2y^2+(1+x^2-y^2)^2)} \quad \text{mit:} \;\; x \in \mathtt{X},\; y \in \mathtt{Y} \;\; \text{und} \;\; (x,y) \neq (0,\pm 1).$$

Die Berechnung von $f(x,y)$ wird ausführlich beschrieben auf Seite 189. $f(x,y)$ ist als Realteilfunktion der holomorphen Funktion $1/(1+z^2),\; z \in \mathbb{C}\backslash\{0 \pm i\}$, eine harmonische Funktion, wenn die singulären Punkte $P(0,\pm 1)$ nicht Element des zweidimensionalen reellen Intervalls Z sind.

$$Z := \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad (x,y) \neq (0,\pm 1). \tag{B.18}$$

$f(x,y)$ nimmt daher für alle Punkte $(x,y) \in Z$ als harmonische Funktion ihre Extrema auf dem Rand von Z an, vgl. dazu auch die Seite 244. Die Berechnung dieser Extremstellen, $m(x,y)$ für das Minimum und $M(x,y)$ für das Maximum, kann wesentlich vereinfacht werden, wenn man $f(x,y) \equiv f(|x|,|y|)$ beachtet und die Eingangsintervalle `X,Y` durch die jeweiligen Intervalle, z.B. $X = \mathtt{abs(X)} = \{|x| \,|\, x \in \mathtt{X}\}$, seiner Absolutbeträge ersetzt. Damit reduziert sich die Berechnung der Extremstellen $m(x,y), M(x,y)$ auf ein Intervall $Z := \left\{(x,y) \,\middle|\, x \in X \wedge y \in Y\right\},\; x \geq 0, y \geq 0$, das nur im 1. Quadranten liegt und den Punkt $(0,1)$ nicht enthalten darf. $X = [x_1, x_2],\; Y = [y_1, y_2]$.

Zur Berechnung der Extremstellen m, M benötigt man zunächst die partiellen Ableitungen:

$$\frac{\partial f(x,y)}{\partial x} = -\frac{2x \cdot (1+2x^2+x^4+2y^2-2x^2y^2-3y^4)}{(1+x^2-2y+y^2)^2 \cdot (1+x^2+2y+y^2)^2}, \tag{B.19}$$

$$\frac{\partial f(x,y)}{\partial y} = -\frac{2y \cdot (-1+2x^2+3x^4+2y^2+2x^2y^2-y^4)}{(1+x^2-2y+y^2)^2 \cdot (1+x^2+2y+y^2)^2}. \tag{B.20}$$

Die Extremalkurven für $\partial f/\partial x$ ergeben sich aus der Forderung $\partial f/\partial x = 0$, siehe Seite 247. Die erste ist wegen des Faktors $x = 0$ die positive y-Achse, und die zweite Extremalkurve ergibt sich aus der Forderung $\quad 1+2x^2+x^4+2y^2-2x^2y^2-3y^4 = 0, \quad$ aus der für den 1. Quadranten folgt[2]

$$y = h_1(x) = \sqrt{(1-x^2+2\cdot\sqrt{1+x^2+x^4}\,)/3}, \quad 0 \leq x \leq 16,$$
$$h_1(x) = x \cdot \sqrt{(u-1+2\cdot\sqrt{1+u+u^2}\,)/3}, \quad u := (1/x)^2,\; x > 16.$$

Die Extremalkurven für $\partial f/\partial y$ ergeben sich analog aus der Forderung $\partial f/\partial y = 0$. Die erste ist wegen des Faktors $y = 0$ die positive x-Achse und die zweite und dritte Extremalkurve ergibt sich aus $\;-1+2x^2+3x^4+2y^2+2x^2y^2-y^4 = 0,\;$ und für den 1. Quadranten folgt wie oben

$$y = h_2(x) := \sqrt{1+x^2+2x\cdot\sqrt{1+x^2}}, \quad 0 \leq x \leq 16, \tag{B.21}$$

$$:= x \cdot \sqrt{1+u+2\cdot\sqrt{1+u}}, \quad u := (1/x)^2,\; x > 16. \tag{B.22}$$

$$y = h_3(x) := \sqrt{1+x^2-2x\cdot\sqrt{1+x^2}}, \quad 0 \leq x \leq 1/\sqrt{3}. \tag{B.23}$$

Für die Bestimmung der Extremalpunkte m, M ist es noch wichtig zu wissen, in welchen Bereichen des 1. Quadranten $f(x,y)$ positiv oder negativ ist. Für $y > h_4(x) := \sqrt{1+x^2}$ gilt $f(x,y) < 0$ und für $y < h_4(x)$ gilt $f(x,y) > 0$.

Die Funktionen $h_i(x), i = 1,2,3,4$ besitzen den gemeinsamen Punkt $P(0,1)$ und erfüllen die Ungleichungen

$$h_3(x) \leq h_1(x), \text{ falls } 0 \leq x \leq 1/\sqrt{3}; \qquad h_1(x) \leq h_4(x) \leq h_2(x), \text{ falls } x \geq 0. \tag{B.24}$$

Der Beweis der Ungleichungen ist trivial und bleibt dem Leser überlassen.

[2] $h_1(x)$ wurde berechnet mit *Mathematica* und der `Solve[...]`-Funktion.

In der folgenden Abbildung sind die Funktionen $h_1(x), h_2(x), h_3(x), h_4(x)$ im 1. Quadranten entsprechend farbig und leicht verzerrt dargestellt.

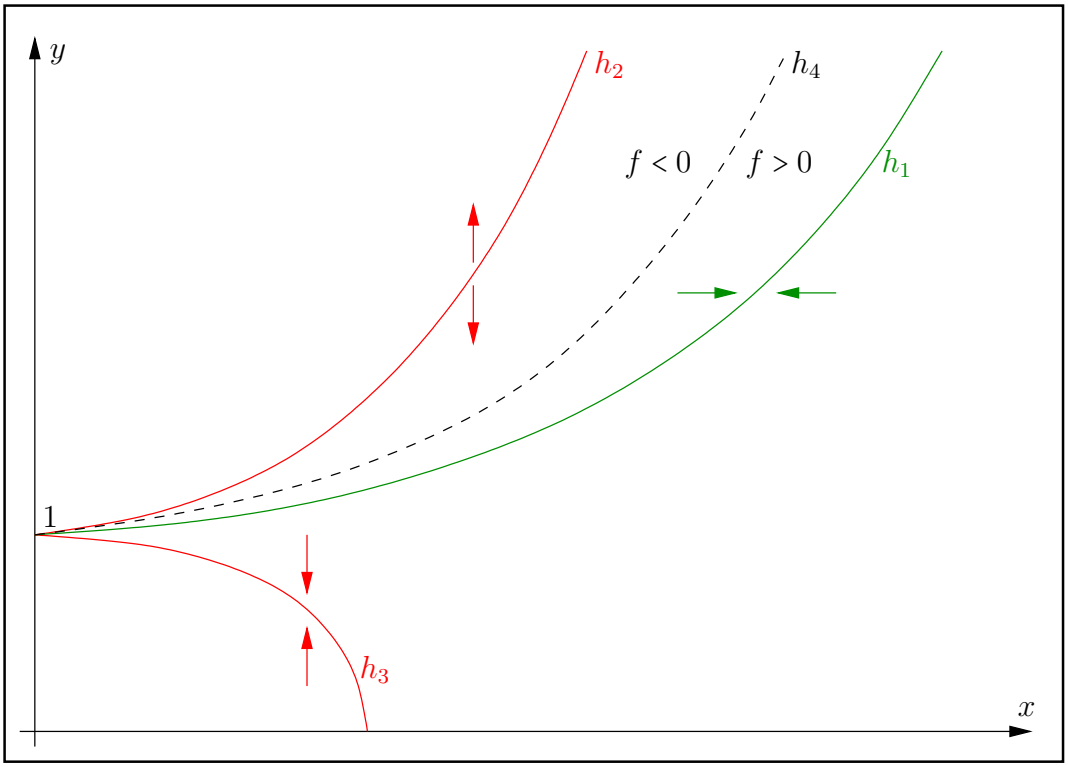


Abbildung B.17.: Extremalkurven und Richtungspfeile für wachsende $f(x,y)$-Werte

Die roten Pfeile geben parallel zur y-Achse und die grünen Pfeile parallel zur x-Achse die Richtung wachsender $f(x,y)$-Werte an.

B.6.1. Maximumbestimmung

Vergleicht man die Extremalkurven der obigen Abbildung z.B. mit denen aus Abbildung B.4, so erkennt man, dass sich die Extremalkurven von $y = h_2(x)$ und $y = \sqrt{3}\cdot x$ bzw. von $y = h_1(x)$ und $y = x/\sqrt{3}$ genau entsprechen, da sie jeweils im gemeinsamen Punkt $(0,0)$ bzw. $(0,+1)$ starten, monoton sind und die gleiche gegenseitige Lage zueinander besitzen. Außerdem stimmen auch die jeweiligen Pfeile, welche die Richtungen wachsender Funktionswerte von $f(x,y)$ angeben, genau überein. Zusätzlich liegen auch die monoton verlaufenden *Nulllinien*, d.h. die Graphen der Funktionen $y = x$ bzw. $y = h_4(x) = \sqrt{1+x^2}$ stets zwischen diesen Extremalkurven.

Prinzipiell **neu** ist in Abbildung B.17 jedoch die zusätzliche Extremalkurve $y = h_3(x)$, die im Definitionsbereich $0 \le x \le 1/\sqrt{3}$ streng monoton fällt. Wenn wir daher zur Bestimmung von M den Algorithmus von Seite 213 sinngemäß übernehmen wollen, so muss in einem vorgeschalteten Punkt 0. untersucht werden, wie der Extremalpunkt M zu bestimmen ist, wenn der untere linke Eckpunkt des Rechtecks Z unterhalb von $h_3(x)$ liegt. In der folgenden Abbildung wird dieser Fall daher genauer untersucht.

Wir müssen noch zeigen, dass die Nachweise bei den Anmerkungen auf Seite 213 auch mit unserer Funktion $f(x,y)$ und den entsprechenden Extremalkurven geführt werden können. Nach **Punkt 1.** ist unter den Voraussetzungen $x_2 > x_1,\ y_1 > \sqrt{1+x_1^2}$ zu zeigen:

$$f(x_1,y_1) < f(x_2,h_2(x_2)) = \frac{-1}{4x_2\cdot(x_2+\sqrt{1+x_2^2})}.$$

Wegen $1 \le h_2(x_1) \le y_1 \le h_2(x_2)$ gilt die Doppelungleichung

$$1 + x_1^2 + 2x_1 \cdot \sqrt{1 + x_1^2} \le y_1^2 \le 1 + x_2^2 + 2x_2 \cdot \sqrt{1 + x_2^2},$$

und damit erhält man nach einfachen Umrechnungen die Abschätzung

$$f(x_1, y_1) = \frac{1 + x_1^2 - y_1^2}{4x_1^2 y_1^2 + (1 + x_1^2 - y_1^2)^2} \le \frac{-1}{4x_1(x_1 + \sqrt{1 + x_1^2})}.$$

Wegen $x_2 > x_1$ folgt daraus unmittelbar die Behauptung ∎ Die Punkte 2. und 3. auf Seite 213 sind auch hier trivial und der Nachweis unter Punkt 4. bleibt dem Leser überlassen.

In der folgenden Abbildungen zeigen wir unter der Voraussetzung *Der linke untere Eckpunkt von Z liegt unterhalb oder auf dem Graphen von $h_3(x)$* drei typische Lagen eines Rechteckintervalls Z.

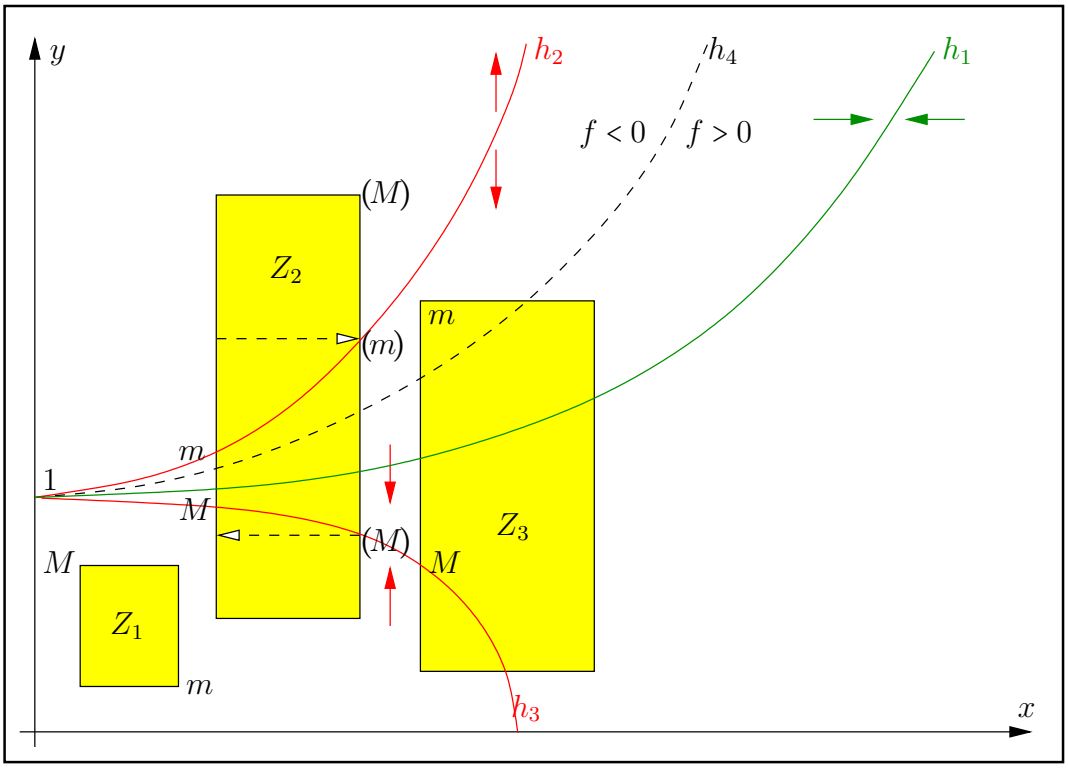


Abbildung B.18.: Linke untere Ecke von Z unterhalb von $h_3(x)$

Anmerkungen:

1. In Z_2 kann der Eckpunkt (M) oben rechts kein Maximumpunkt sein, da er im Bereich $f(x, y) < 0$ liegt. Der Schnittpunkt (M) mit $h_3(x)$ kann ebenfalls kein Maximumpunkt sein, da man längs des punktierten Pfeils in Richtung wachsender Funktionswerte nach M kommt, und ganz entsprechend kann (m) kein Minimumpunkt sein.

2. Bewegt man bei Z_2 und Z_3 die obere Parallele zu x-Achse so, dass der obere rechte Eckpunkt immer oberhalb oder auf dem Graphen von h_3 liegt, so befindet sich M stets auf der linken Parallelen zur y-Achse. Schneidet diese Parallele den Graphen von $h_3(x)$, so ist M dieser Schnittpunkt, sonst ist M die linke obere Ecke von Z. Der Leser möge dies anhand aller möglichen, prinzipiellen Lagen von Z überprüfen!

3. Wenn also der untere linke Eckpunkt von Z unterhalb oder auf dem Graphen von $h_3(x)$ liegt, so liegt M stets auf der linken Parallelen zur y-Achse. Schneidet diese Parallele den Graphen von $h_3(x)$, so ist M dieser Schnittpunkt, sonst ist M der obere linke Eckpunkt von Z.

Wir formulieren jetzt noch die Bedingungen aus dem letzten Punkt 3. in der Sprache der Intervallarithmetik. Dazu benutzen wir die nur für interne Anwendungen definierte Funktion

```
MpfiClass Re_r1pz2_H3(const MpfrClass& x)
```

welche für eine Maschinenzahl $0 \le x < 1/\sqrt{3}$ den Funktionswert $h_3(x)$ optimal einschließt. Eine ebenfalls optimale Einschließung von $1/\sqrt{3}$ liefert mit $x = 1$ und $n = 3$ die Funktion

```
MpfiClass xdivsqrtn (const MPFR::MpfrClass& x, unsigned long int n)
```

welche in `mpficlass.cpp` definiert ist.

Die Bedingung *Die linke untere Ecke von Z liegt unterhalb oder auf dem Graphen von* $h_3(x)$ ist definiert durch:

1. $x_1 < 1/\sqrt{3}$ und wird realisiert durch: $x_1 \le$ `Inf(xdivsqrtn(1,3))`,
2. $y_1 \le h_3(x_1)$ und wird realisiert durch: $y_1 \le$ `Inf(Re_r1pz2_H3(`x_1`))`.

Wenn oben $y_1 =$ `Inf(Re_r1pz2_H3(`x_1`))` erfüllt ist, so kann gelten $y_1 = h_3(x_1)$, aber auch dann ist M der Schnittpunkt der linken Parallelen von Z mit dem Graphen von $h_3(x)$.

Die Bedingung *Die linke Parallele von Z schneidet den Graphen von* $h_3(x)$ ist definiert durch:

$$y_1 \le h_3(x) \le y_2 \quad \text{und wird realisiert durch:} \quad h_3([x_1]) \le [y_1, y_2],$$

wobei $h_3([x_1])$ die mit `MpfiClass Re_r1pz2_H3(const MpfrClass& x)` berechnete **optimale** Einschließung von $h_3(x_1)$ bedeutet.

Zur Bestimmung von M benötigen wir zusätzlich die internen Funktionen

```
MpfiClass Re_r1pz2_H1(const MpfrClass& x)
MpfiClass Re_r1pz2_H2(const MpfrClass& x)
MpfiClass Re_r1pz2_H4(const MpfrClass& x)
```

die für eine Maschinenzahl x **optimale** Einschließungen der Funktionswerte $h_1(x), h_2(x), h_4(x)$ liefern.

Anmerkung:
Um sicherzugehen, dass z.B. für Maschinenzahlen x_1, y_1 die Ungleichung $y_1 \le h_3(x_1)$ erfüllt ist, wenn $y_1 \le$ `Inf(Re_r1pz2_H3(`x_1`))` gilt, muss vorausgesetzt werden, dass mit `Re_r1pz2_H3(`x_1`)` eine **optimale** Einschließung von $h_3(x_1)$ berechnet wird. Der Leser möge mit einem einfachen Beispiel zeigen, dass die **optimale** Einschließung wirklich vorausgesetzt werden muss. Die obigen fünf Funktionen liefern stets die notwendigen optimalen Einschließungen, wenn für Maschinenzahlen x gilt:

$$x > 0; \quad h_i(x) \text{ ist keine Maschinenzahl}; \quad \text{Eine Fehlermeldung tritt nicht auf.}$$

Falls jedoch die Funktionswerte $h_i(x), i = 1, 2, 3$ für Maschinenzahlen x selbst wieder Maschinenzahlen sind, so kann der Fall eintreten, dass $h_i(x)$ nicht durch ein Punktintervall sondern durch ein echtes Intervall eingeschlossen wird, das jedoch in seinem Innern keine Maschinenzahl enthält. In diesem Fall kann z.B. die Ungleichung $y_1 \le h_3(x_1)$ durch die Maschinenabfrage

$$y_1 \le \texttt{Inf(Re\_r1pz2\_H3(}x_1\texttt{))}$$

nicht mehr garantiert werden. Mir ist jedoch kein Fall bekannt, in dem $h_i(x), i = 1, 2, 3$ für $x > 0$ wirklich eine Maschinenzahl ist. Für $h_4(x) = \sqrt{1 + x^2}$ gibt es natürlich Beispiele, bei denen für Maschinenzahlen $x > 0$ die Funktionswerte $h_4(x)$ selbst wieder Maschinenzahlen sind, z.B. gilt $\sqrt{1 + (3/4)^2} = 5/4$. Die Funktion `Re_r1pz2_H4(x)` liefert jedoch für diese ersten Spezialfälle einschließende **Punktintervalle**. Diese Spezialfälle berechnen sich aus der speziellen Pythagoräischen Gleichung

$$(2^{n+1})^2 + (2^{2(n-p)} - 2^{2p})^2 = (2^{2(n-p)} + 2^{2p})^2, \quad n \in \mathbb{N},\ p \in \mathbb{N}_0,\ p = 0, 1, \ldots, n/2 + 1,$$

wobei x gegeben ist durch: $x = (2^{2(n-p)} - 2^{2p})/2^{n+1}$, mit z.B. $x = 3/4$ für $n = 1$ und $p = 0$.

Um die maximalen oder minimalen Funktionswerte $f(x,y)$ an den Schnittpunkten der Rechteckseiten von Z mit den Extremalkurven $h_1(x), h_2(x), h_3(x)$ einschließen zu können, benötigen wir noch die folgenden Funktionen

$$f_{h1}(x) := f(x,h_1(x)) = \begin{cases} \dfrac{3}{4\cdot(2+x^2+\sqrt{1+x^2+x^4})}, & 0 \le x \le 8, \\ u\cdot\dfrac{3}{4\cdot(1+2u+\sqrt{1+u+u^2})}, & u=(1/x)^2,\ x>8. \end{cases}$$

$$f_{h2}(x) := f(x,h_2(x)) = \begin{cases} \dfrac{-1}{4x\cdot(\sqrt{1+x^2}+x)}, & 0 \le x \le 8, \\ u\cdot\dfrac{-1}{4\cdot(1+\sqrt{1+u})}, & u=(1/x)^2,\ x>8. \end{cases}$$

$$f_{h3}(x) := f(x,h_3(x)) = \frac{\sqrt{1+x^2}+x}{4x}, \quad 0 < x \le 1\sqrt{3}.$$

deren Funktionswerte mithilfe der internen Funktionen

```
MpfiClass Re_r1pz2_fH1(const MpfrClass& x)
MpfiClass Re_r1pz2_fH2(const MpfrClass& x)
MpfiClass Re_r1pz2_fH3(const MpfrClass& x)
```

eingeschlossen werden. Beachten Sie dabei, dass diese Einschließungen nicht unbedingt optimal sondern nur nahezu optimal sein müssen. Die obigen Funktionsterme $f_{hi}(x)$ wurden mit dem Algebrasystem *Mathematica* und `FullSimplify[...]` so berechnet, dass bei der Auswertung im 1. Quadranten wegen $x \ge 0$ keine Auslöschungen auftreten können.

Um analog zu Punkt 2. auf Seite 213 den Funktionswert über dem Schnittpunkt der unteren Parallelen von Z mit dem Graphen der Extremalkurve $h_1(x)$ zu berechnen, gilt zunächst für den Schnittpunkt $M(x_S,y_S) = M(x_S,y_1)$, wobei y_1 und x_S gegeben sind durch[3]

$$y_1 = h_1(x_S) = \sqrt{(1-x_S^2+2\cdot\sqrt{1+x_S^2+x_S^4})/3} \quad \rightsquigarrow \quad x_S = \sqrt{y_1^2-1+2y_1\cdot\sqrt{y_1^2-1}}\,.$$

Der Funktionswert über $M(x_S,y_1)$ ergibt sich daraus mit $y = y_1$ zu

$$f(x_S,y_1) = h_5(y) := \frac{1}{4y\cdot(y+\sqrt{y^2-1})} = u\cdot\frac{1}{4\cdot(1+\sqrt{1-u})}, \quad u := (1/y)^2 \le 1,$$

wobei $y^2 \ge 1$ automatisch erfüllt ist. Die Funktionswerte von $h_5(x)$ werden für Maschinenzahlen $x \ge 1$ nahezu optimal eingeschlossen mit Hilfe der internen Funktion

```
MpfiClass Re_r1pz2_H5(const MpfrClass& x);
```

B.6.2. Minimumbestimmung

Analog zur Maximumbestimmung können wir auch jetzt die Lage der Minimumpunkte m analog zum Fall $1/z^2$ bestimmen, wenn alle Rechtecke Z stets oberhalb des Graphen der Extremalkurve $y = h_3(x)$ liegen, und dies gilt auch dann noch, wenn die untere Parallele von Z den Graphen der Extremalkurve $y = h_3(x)$ schneidet. In der folgenden Abbildung wird dies am Beispiel Z_0 verdeutlicht, das dem Rechteck Z_{13} in Abbildung B.6 entspricht, wobei der Minimumpunkt m der linke obere Eckpunkt des jeweiligen Rechtecks ist. Der Leser möge diesen Sachverhalt für alle restlichen typischen Lagen von Z bestätigen, wenn die untere Parallele von Z den Graphen der Extremalkurve $y = h_3(x)$ schneidet.

Zur Bestimmung der Lage von m muss daher nur noch der Fall gesondert behandelt werden, dass der rechte untere Eckpunkt von Z unterhalb des Graphen von $y = h_3(x)$ liegt. In diesem

[3] Die folgenden Umformungen wurden mit *Mathematica* und der Funktion `Solve[...]` durchgeführt.

Fall genügt es, den Minimumpunkt m auch wieder analog zum Fall $1/z^2$ zu bestimmen und den Funktionswert $f(x,y)$ über diesem Punk mit dem Funktionswert über dem rechten unteren Eckpunkt von Z zu vergleichen. In der nachfolgenden Abbildung wird dies mit dem Beispiel Z_1 gezeigt. Der Leser möge anhand weiterer typischer Beispiele für die Lage von Z zeigen, dass so der minimale Funktionswert $f(x,y)$ über Z stets korrekt berechnet wird.

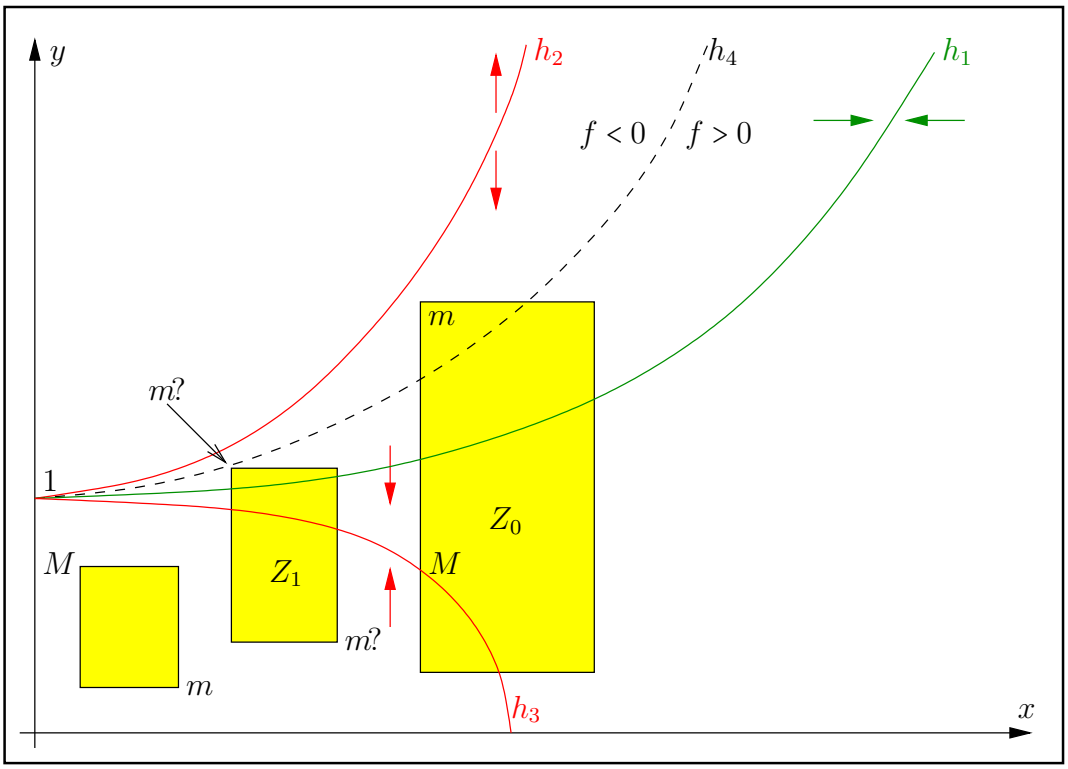


Abbildung B.19.: Zur Lage der Minimumpunkte m

B.6.3. Numerische Beispiele

Im **1. Beispiel** wählen wir Z_1 aus Abb. B.19 mit $X = [1/4, 1/2]$, $Y = [1/4, 5/4]$. Mit einfachen Rechnungen erhält man $f_m = f(1/4, 5/4) = -32/41 = -0.78048...$ und $f_M = f(1/4, h_3(1/4)) = (1 + \sqrt{17})/4 = 1.2807764....$ Mit dem Funktionsaufruf `T = Re_r1pz2(X,Y);` erhalten wir mit `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq$ `T`.

$$\texttt{T} = [-7.80487804878048...78049 \cdot 10^{-1}, +1.2807764064044...96592] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **2. Beispiel** wählen wir $X = [2, 2]$, $Y = [3, 3]$ und erhalten mit `T = Re_r1pz2(X,Y)` für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq$ `T`.

$$\texttt{T} = [-2.5000000000000...00001 \cdot 10^{-2}, -2.4999999999999...99999 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

B.7. $2xy/(4x^2y^2+(1+x^2-y^2)^2)$

Für die vorgegebenen reellen Maschinenintervalle $\mathtt{X} \ni x$ und $\mathtt{Y} \ni y$ liefert die Intervallfunktion

```
MpfiClass Im_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x,y) = \frac{2xy}{4x^2y^2+(1+x^2-y^2)^2} \quad \text{mit:} \quad x \in \mathtt{X},\ y \in \mathtt{Y} \quad \text{und} \quad (x,y) \neq (0,\pm 1).$$

Die Berechnung von $f(x,y)$ wird ausführlich beschrieben auf Seite 188. $-f(x,y)$ ist als Imaginärteil der holomorphen Funktion $1/(1+z^2)$, mit $z \in \mathbb{C}\backslash\{0 \pm i\}$, eine harmonische Funktion, wenn die singulären Punkte $P(0,\pm 1)$ nicht Element des zweidimensionalen reellen Intervalls Z sind.

$$Z := \left\{(x,y) \,\middle|\, x \in \mathtt{X} \wedge y \in \mathtt{Y}\right\}, \quad (x,y) \neq (0,\pm 1). \tag{B.25}$$

$f(x,y)$ nimmt daher für alle Punkte $(x,y) \in Z$ als harmonische Funktion ihre Extrema auf dem Rand von Z an, vgl. dazu auch die Seite 244. Um die zugehörigen Extremstellen, m für das Minimum und M für das Maximum, möglichst einfach bestimmen zu können, betrachten wir zunächst die folgenden Symmetrieeigenschaften:

$$f(-x,-y) \equiv f(y,x), \quad \text{Spiegelung am Ursprung,} \tag{B.26}$$

$$f(-x,y) \equiv -f(x,y), \quad \text{Spiegelung an der } y\text{-Achse,} \tag{B.27}$$

$$f(x,-y) \equiv -f(x,y), \quad \text{Spiegelung an der } x\text{-Achse.} \tag{B.28}$$

Mit den obigen drei Transformationen kann die Bestimmung der Extremalpunkte m, M auf den 1. Quadranten reduziert werden. Wir betrachten dazu ein typisches Beispiel, wobei das gegebene Rechteck $Z = \left\{(x,y) \,\middle|\, x \in \mathtt{X} = [-1,3] \wedge y \in \mathtt{Y} = [0.321, 1.10]\right\} = Z1 \cup Z2$ in der oberen Halbebene liegt.

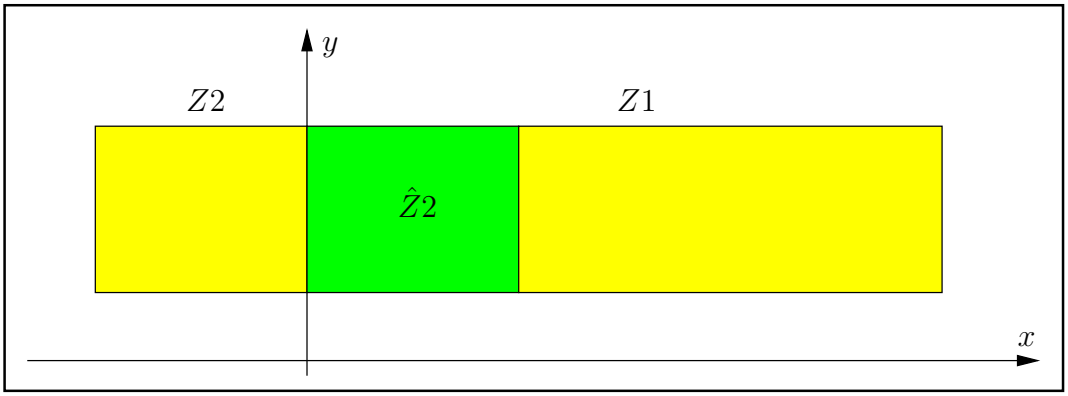


Abbildung B.20.: Transformation in den 1. Quadranten

$Z = Z1 \cup Z2$ wird zuerst in die beiden Intervalle $Z1, Z2$ aus dem 1. und 2. Quadranten zerlegt. Da $f(x,y)$ in $Z2$ nur negative Funktionswerte besitzt, kann $f(x,y)$ über ganz Z sein Minimum nur über $Z2$ annehmen. Ist dann $\hat{Z}2$ das an der y-Achse gespiegelte Rechteck $Z2$ und ist f_M das Maximum über diesem Rechteck, so ist $f_m := -f_M$ das Minimum über ganz Z. Da $f(x,y)$ in $Z1$ nur positive Funktionswerte besitzt, ist das Maximum f_M über $Z1$ auch das Maximum über dem ganzen Rechteck Z. Zur Berechnung von f_m, f_M wurden damit Maximum- und Minimumberechnungen nur über Rechtecken im 1. Quadranten vorgenommen. Ganz analog können auch für alle anderen Lagen von Z die Berechnungen stets nur im 1. Quadranten durchgeführt werden. Weitere Einzelheiten findet man in der Funktion

```
MpfiClass Im_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

die in `mpficlass.cpp` definiert ist.

Zur Berechnung der Extremstellen m, M von Rechtecken Z im 1. Quadranten benötigt man zunächst die partiellen Ableitungen:

$$\frac{\partial f(x,y)}{\partial x} = \frac{2y\cdot(1-2x^2+y^4-2y^2-2x^2y^2-3x^4)}{(1+x^2-2y+y^2)^2\cdot(1+x^2+2y+y^2)^2}, \tag{B.29}$$

$$\frac{\partial f(x,y)}{\partial y} = \frac{2x\cdot(1+2x^2+x^4+2y^2-2x^2y^2-3y^4)}{(1+x^2-2y+y^2)^2\cdot(1+x^2+2y+y^2)^2}. \tag{B.30}$$

Die Extremalkurven für $\partial f/\partial x$ ergeben sich aus der Forderung $\partial f/\partial x = 0$, siehe Seite 247. Die erste ist wegen des Faktors $y = 0$ die positive x-Achse, und die zweite und dritte Extremalkurve ergibt sich aus $1 - 2x^2 + y^4 - 2y^2 - 2x^2y^2 - 3x^4 = 0$. Für den 1. Quadranten folgt daraus[4]

$$y = h_2(x) := \sqrt{1+x^2+2x\cdot\sqrt{1+x^2}}, \quad 0 \le x \le 16, \tag{B.31}$$

$$:= x\cdot\sqrt{1+u+2\cdot\sqrt{1+u}}, \quad u := (1/x)^2,\ x > 16. \tag{B.32}$$

$$y = h_3(x) := \sqrt{1+x^2-2x\cdot\sqrt{1+x^2}}, \quad 0 \le x \le 1/\sqrt{3}. \tag{B.33}$$

Die Extremalkurven für $\partial f/\partial y$ ergeben sich analog aus der Forderung $\partial f/\partial y = 0$. Die erste ist wegen des Faktors $x = 0$ die positive y-Achse und die zweite ergibt sich aus der Forderung $1 + 2x^2 + x^4 + 2y^2 - 2x^2y^2 - 3y^4 = 0$, aus der für den 1. Quadranten wie oben folgt

$$y = h_1(x) := \sqrt{(1-x^2+2\cdot\sqrt{1+x^2+x^4})/3}, \quad 0 \le x \le 16, \tag{B.34}$$

$$:= x\cdot\sqrt{(u-1+2\cdot\sqrt{1+u+u^2})/3}, \quad u := (1/x)^2,\ x > 16. \tag{B.35}$$

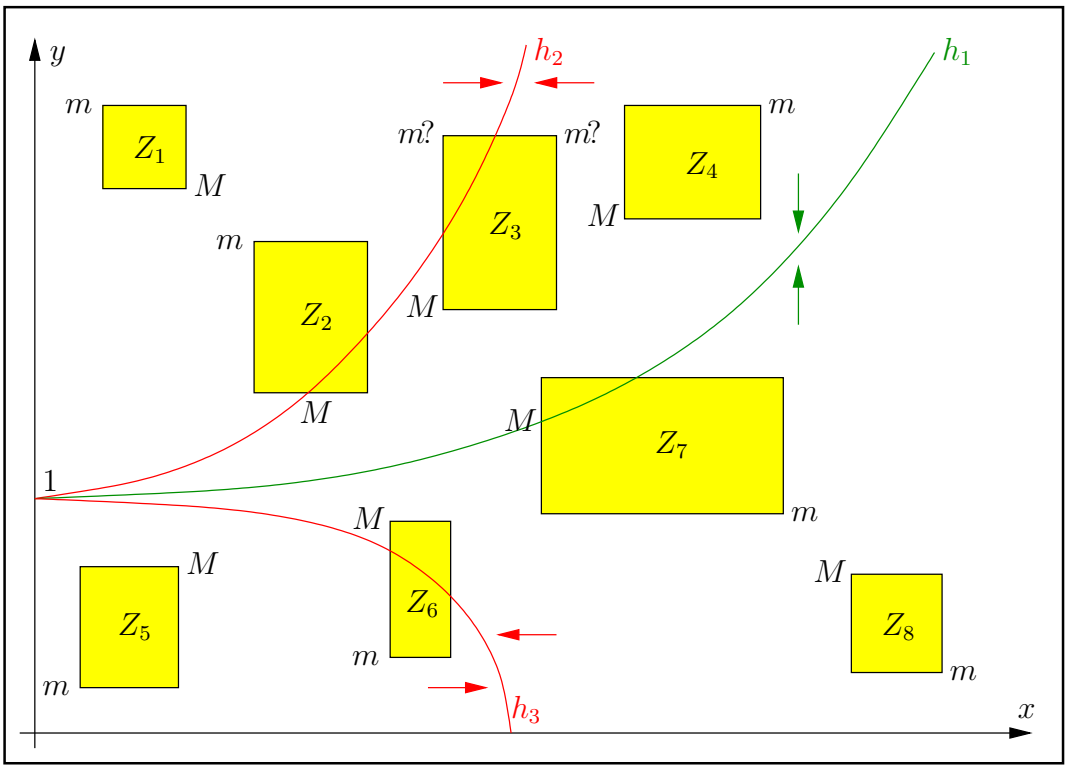


Abbildung B.21.: Rechtecke Z_i und Extremalkurven $h_1(x), h_2(x), h_3(x)$ im 1. Quadranten

In den folgenden Abbildungen sind weitere typische Lagen der Rechtecke Z angegeben. Dabei bedeuten z.B. zwei oder auch mehrere Symbole $m?$, dass jeweils hier ein Minimum vorliegen kann, weshalb entsprechend mehrere Rechnungen und Vergleiche notwendig werden.

[4] $h_1(x)$ wurde berechnet mit *Mathematica* und der `Solve[...]`-Funktion.

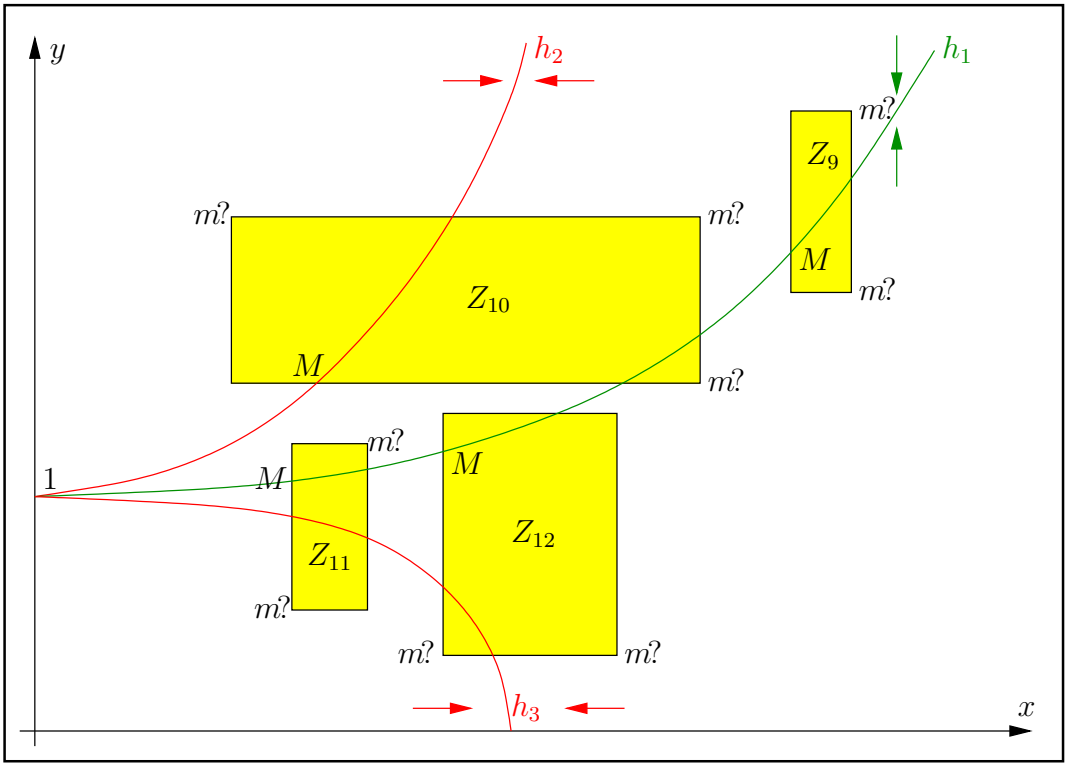


Abbildung B.22.: Rechtecke Z_i und Extremalkurven $h_1(x), h_2(x), h_3(x)$ im 1. Quadranten

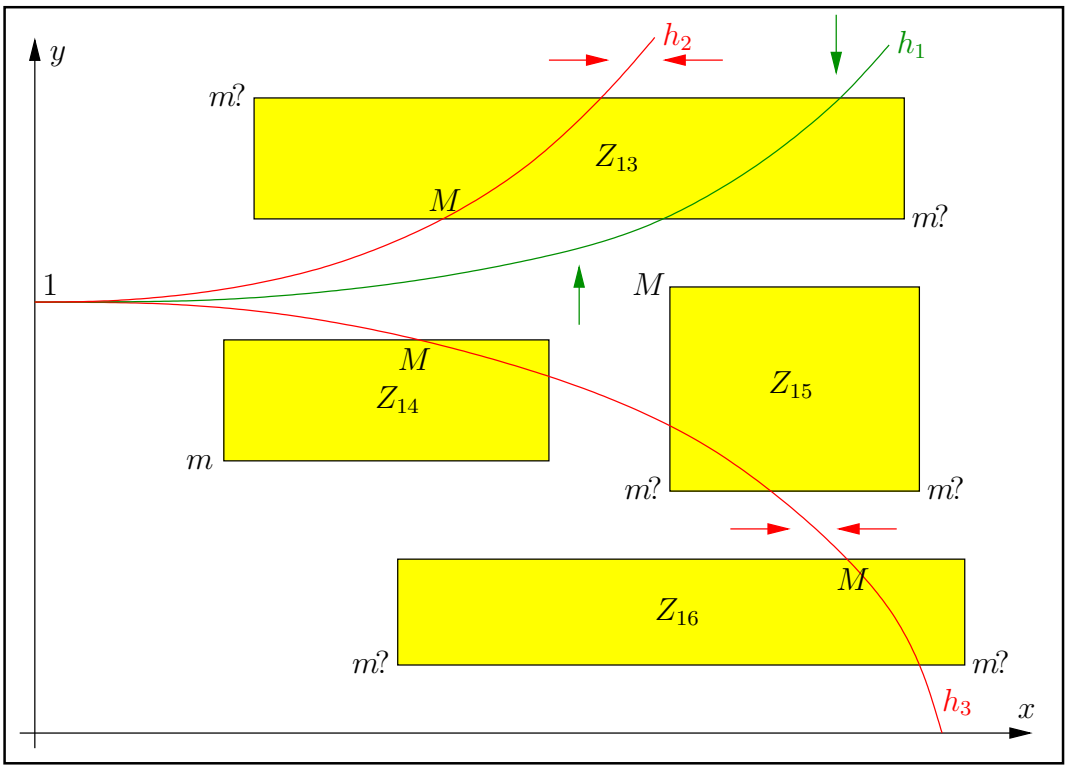


Abbildung B.23.: Rechtecke Z_i und Extremalkurven $h_1(x), h_2(x), h_3(x)$ im 1. Quadranten

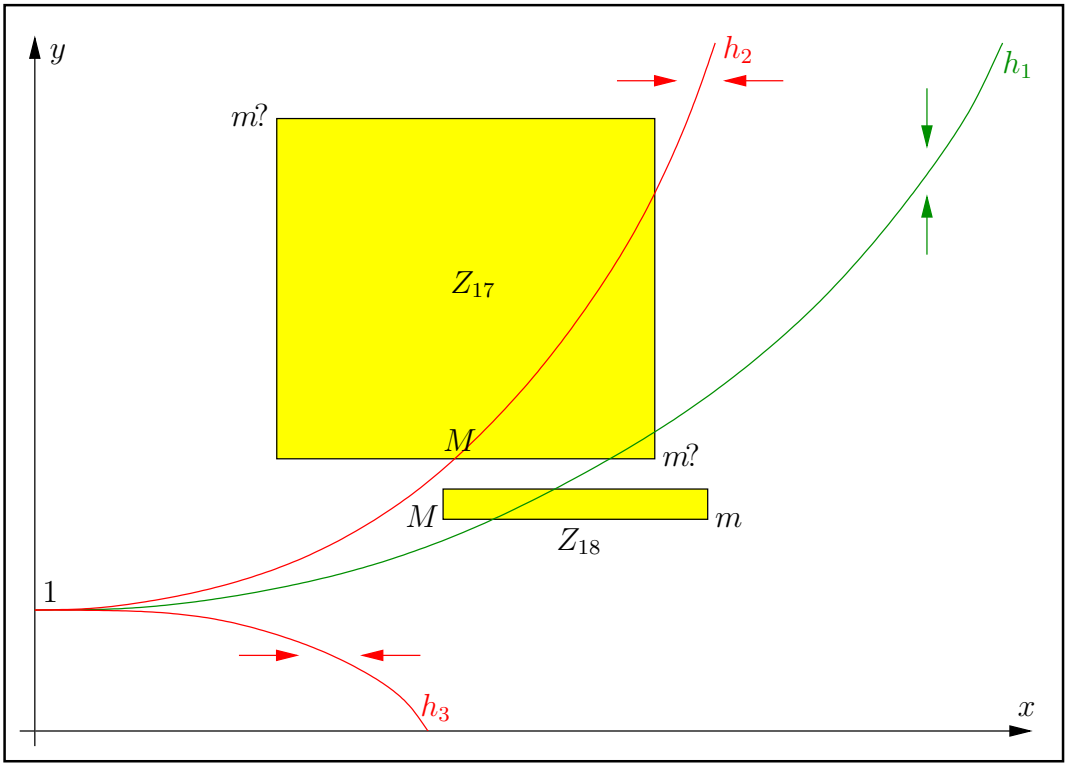


Abbildung B.24.: Rechtecke Z_i und Extremalkurven $h_1(x), h_2(x), h_3(x)$ im 1. Quadranten

B.7.1. Maximumbestimmung

Die Berechnung des Maximums von $f(x, y)$ über einem beliebig vorgegebenen Rechteck kann reduziert werden auf die Berechnung des Maximums über einem Rechteck Z, welches nur im ersten Quadranten liegt. Auf Seite 233 haben wir gezeigt, wie vorzugehen ist, wenn das ursprüngliche Rechteck z.B. nur in der oberen Halbebene liegt. Wir betrachten jetzt also nur Rechtecke Z, die im 1. Quadranten liegen und den Punkt $(0, +1)$ nicht enthalten. Bevor jedoch der maximale Funktionswert von $f(x, y)$ auf dem Rand von Z berechnet werden kann, muss der Punkt M auf dem Rand von Z bestimmt werden, auf dem der maximale Funktionswert angenommen wird, und genau dies ist das eigentliche Problem, das für jede neue Funktion $f(x, y)$ immer wieder neu gelöst werden muss. In unserem Fall benutzen wir den folgenden

Satz B.1 (Die linke Parallele von Z schneidet $h_1(x)$).
Schneidet die linke Parallele von Z den Graphen von $h_1(x)$, so ist dieser Schnittpunkt gleich M.

In den obigen Abbildungen wird dies bestätigt durch die Rechtecke Z_7, Z_9, Z_{11}, Z_{12}. Der Algorithmus zur Bestimmung von M kann daher konstruiert werden durch die folgenden drei Fälle, die sich selbst gegenseitig ausschließen:

1. Die linke obere Ecke von Z liegt unterhalb von $h_1(x)$, vgl. $Z_5, Z_6, Z_8, Z_{14}, Z_{15}, Z_{16}$.
2. Die linke untere Ecke von Z liegt oberhalb von $h_1(x)$, vgl. $Z_1, Z_2, Z_3, Z_4, Z_{10}, Z_{13}, Z_{17}, Z_{18}$.
3. Die linke Parallele von Z schneidet den Graphen von $h_1(x)$, vgl. Z_7, Z_9, Z_{11}, Z_{12}.

Auf der folgenden Seite findet man nach diesen Vorüberlegungen einen Grobalgorithmus zur Bestimmung des Maximumpunktes M.

```
if (y2 <= h1(x1)) // Linke obere Ecke von Z liegt unterhalb von h1(x):
{
  if (y2 <= h3(x2))
    M(x2,y2);
  else
  {
    if (y2 <= h3(x1)) // M: Schnittpunkt der oberen Parallelen mit h3(x)
      M(xS,y2) mit y2 = h3(xS); vgl. Z14
    else M(x1,y2);
  }
}
else
{
  if (y1 > h1(x1)) // Linke untere Ecke von Z liegt oberhalb von h1(x)
  {
    if (y1 > h2(x2)) M(x2,y1);
    else
    {
      if (y1 > h2(x1)) // M ist S.P. der unteren Parallelen mit h2(x)
        M(xS,y1) mit y1 = h2(xS);
      else M(x1,y1);
    }
  }
  else // Die linke Parallele von Z schneidet h1(x)
    M ist dieser Schnittpunkt mit h1(x);
}
```

Den Algorithmus zur Bestimmung des maximalen Funktionswertes findet man in der Funktion

```
MPFR::MpfrClass fM_Im_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

die in `mpficlass.cpp` definiert ist. Wir betrachten jetzt noch die Fälle, in denen M der Schnittpunkt vom Rand von Z mit einer der Extremalkurven $h_1(x), h_2(x), h_3(x)$ ist.

Z_{14}: Die obere Parallele von Z_{14} schneidet den Graphen von $h_3(x)$ im Punkt $M(x_S, y_2)$, mit $y_2 = h_3(x_S) = \sqrt{1 + x_S^2 - 2x_S \cdot \sqrt{1 + x_S^2}}$. Um $f(x_S, y_2)$ berechnen zu können, muss zunächst $x_S = x_S(y_2)$ als Funktion von $y_2 < 1$ berechnet werden: $x_S = \sqrt{(-1 - y_2^2 + 2\sqrt{1 - y_2^2 + y_2^4})/3}$. Für $f(x_S, y_2)$ erhält man schließlich, wenn y_2 durch y ersetzt wird:

$$f(x_S(y), y) = h_4(y) := \frac{0.75 \cdot y \cdot \sqrt{-3 - 3y^2 + 6 \cdot \sqrt{1 - y^2 + y^4}}}{1 + y^4 + \sqrt{1 - y^2 + y^4} + y^2 \cdot (-4 + \sqrt{1 - y^2 + y^4})}, \quad 0 \le y \le 1/2.$$

Man erkennt, dass der obige Term für $y \to 1$ die Form 0/0 besitzt und deshalb umgeformt werden muss:

$$f(x_S(y), y) = h_4(y) := \frac{(1 + y^2) \cdot \sqrt{1 - y^2 + y^4} - 1 + 4y^2 - y^4}{4y \cdot (1 - y^2) \cdot \sqrt{1 + y^2 + 2 \cdot \sqrt{1 - y^2 + y^4}}}, \quad 1/2 < y < 1.$$

Die Auswertung von $h_4(x)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfiClass Im_r1pz2_fh4(const MPFR::MpfrClass& x)
```

die in `mpficlass.cpp` definiert ist. Beachten Sie bitte, dass bei der Auswertung von $h_4(y)$ in den beiden Teilbereichen $0 \le y \le 1/2$ und $1/2 < y < 1$ keine Auslöschung auftreten kann.

Z_{17}: Die untere Parallele von Z_{17} schneidet den Graphen von $h_2(x)$ im Punkt $M(x_S, y_1)$, mit $y_1 = h_2(x_S) = \sqrt{1 + x_S^2 + 2x_S \cdot \sqrt{1 + x_S^2}}$. Um $f(x_S, y_1)$ berechnen zu können, muss zunächst $x_S = x_S(y_1)$ als Funktion von $y_1 > 1$ berechnet werden: $x_S = \sqrt{(-1 - y_1^2 + 2\sqrt{1 - y_1^2 + y_1^4})/3}$. Für $f(x_S, y_1)$ erhält man, wenn y_1 durch y ersetzt wird:

$$f(x_S(y), y) = h_6(y) := \frac{(1 + y^2) \cdot \sqrt{1 - y^2 + y^4} - 1 + 4y^2 - y^4}{4y \cdot (y^2 - 1) \cdot \sqrt{1 + y^2 + 2 \cdot \sqrt{1 - y^2 + y^4}}}, \quad 1 < y \le 8.$$

Für $y \to \infty$ liefert der obige Zähler starke Auslöschung, so dass weiter umgeformt wird:

$$h_6(y) := \frac{9 \cdot y \cdot (y^2 - 1)}{4 \cdot \sqrt{1 + y^2 + 2 \cdot \sqrt{1 - y^2 + y^4}} \cdot \left(1 - 4y^2 + y^4 + (1 + y^2) \cdot \sqrt{1 - y^2 + y^4}\right)}, \quad y > 8.$$

Für $y \to \infty$ gilt $h_6(y) \sim (9/(8\sqrt{3}))/y^2$, so dass ein vorzeitiger Überlauf zu vermeiden ist. Mit $u := (1/y)^2$ erhält man für $y > 8$ die Darstellung

$$h_6(y) = g(u) := u \cdot \frac{9 \cdot (1 - u)}{4 \cdot \sqrt{1 + u + 2 \cdot \sqrt{1 - u + u^2}} \cdot \left(1 - 4u + u^2 + (1 + u) \cdot \sqrt{1 - u + u^2}\right)}.$$

Die Auswertung von $h_6(x)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfiClass Im_r1pz2_fh6(const MPFR::MpfrClass& x)
```

die in `mpficlass.cpp` definiert ist. Beachten Sie bitte, dass bei der Auswertung von $h_6(y)$ in den beiden Teilbereichen $1 < y \le 8$ und $8 < y < \infty$ keine Auslöschung auftreten kann.

Z_{12}: Die linke Parallele von Z_{12} schneidet den Graphen von $h_1(x)$ im Punkt $M(x_1, h_1(x_1))$, mit $h_1(x_1) = \sqrt{(1 - x_1^2 + 2 \cdot \sqrt{1 + x_1^2 + x_1^4})/3}$. Für $f(x, h_1(x))$ erhält man den Ausdruck

$$f(x, h_1(x)) = h_5(x) := \frac{3x \cdot \sqrt{3 - 3x^2 + 6 \cdot \sqrt{1 + x^2 + x^4}}}{4 \cdot \left(1 + 4x^2 + x^4 + (x^2 - 1) \cdot \sqrt{1 + x^2 + x^4}\right)}, \quad 2 \le x < \infty.$$

Die Einschränkung $2 \le x < \infty$ ist notwendig, da der Nenner für $x \to 0$ starke Auslöschung verursacht. Für $x \to \infty$ gilt $h_5(x) \sim 3\sqrt{3}/(8x^2)$, so dass ein vorzeitiger Überlauf vermieden werden muss. Mit $u := (1/x)^2$ erhält man nach einfachen Umformungen

$$h_5(x) = g(u) := u \cdot \frac{0.75 \cdot \sqrt{3u - 3 + 6 \cdot \sqrt{1 + u + u^2}}}{1 + 4u + u^2 + (1 - u) \cdot \sqrt{1 + u + u^2}}, \quad u = (1/x)^2,\ 2 \le x < \infty.$$

Im Fall $x \to 0$ muss der obige Ausdruck für $f(x, h_1(x))$ weiter umgeformt werden zu:

$$h_5(x) := \frac{3 \cdot (1 + x^4 + \sqrt{1 + x^2 + x^4}) \cdot \sqrt{3 - 3x^2 + 6 \cdot \sqrt{1 + x^2 + x^4}}}{4x \cdot \left(4 + 2x^2 + 5x^4 + x^6 + (5 + x^4) \cdot \sqrt{1 + x^2 + x^4}\right)}, \quad 0 < x \le 2.$$

Beim Auswerten der Funktionen $h_5(x)$ und $g(u)$ tritt jetzt in beiden Teilbereichen keinerlei Auslöschung auf. Die nahezu optimale Einschließung der Funktionswerte von $h_5(x)$ und $g(u)$ erfolgt auf der Maschine mit Hilfe der Funktion

```
MpfiClass Im_r1pz2_fH1(const MPFR::MpfrClass& x)
```

die in `mpficlass.cpp` definiert ist.

Damit haben wir alle Fälle behandelt, in denen der Maximumpunkt M ein Schnittpunkt des Randes von Z mit einer der Extremalkurven $h_1(x), h_2(x), h_3(x)$ ist.

B.7.2. Minimumbestimmung

In Abbildung B.24 geben die Pfeile die Richtung wachsender Funktionswerte von $f(x,y)$ an, so dass ein Minimumpunkt m nicht auf einer der Extremalkurven $h_1(x), h_2(x), h_3(x)$ liegen kann. Daher kann m nur einer der vier Eckpunkte eines gegebenen Rechtecks Z im 1. Quadranten sein, so dass bei einem echten Rechteck vier Funktionswertberechnungen und drei Vergleiche notwendig sind. Bei einem entarteten Rechteck oder bei einem zweidimensionalen Punktintervall sind entsprechend nur zwei bzw. ein Funktionswert zu berechnen. Die Bestimmung des minimalen Funktionswertes von $f(x,y)$ über einem der Eckpunkte von Z erfolgt mit Hilfe der Funktion

`MpfrClass fm_Im_r1pz2(const MpfiClass& X, const MpfiClass& Y)`,

die in `mpficlass.cpp` definiert ist.

In nachfolgender Tabelle sind alle notwendigen numerischen Funktionen zusammengestellt, um den maximalen oder minimalen Funktionswert $f(x,y)$ über dem Rand eines vorgegebenen Rechtecks Z aus dem 1. Quadranten berechnen zu können.

Nr.	Funktionsterm, $u := (1/x)^2$	Funktionsaufruf
1.	$f(x,y) = \dfrac{2xy}{4x^2y^2 + (1+x^2-y^2)^2}$, mit Skalierung	`Im_r1pz2(x,y,rnd)`
2.	$h_1(x) = \sqrt{(1 - x^2 + 2\sqrt{1+x^2+x^4})/3},\ 0 \le x \le 16$	`Re_r1pz2_H1(x)`
3.	$h_1(x) = x \cdot \sqrt{(u - 1 + 2\sqrt{1+u+u^2})/3},\ x > 16$	`Re_r1pz2_H1(x)`
4.	$h_2(x) = \sqrt{1 + x^2 + 2x \cdot \sqrt{1+x^2}},\ 0 \le x \le 16$	`Re_r1pz2_H2(x)`
5.	$h_2(x) = x \cdot \sqrt{1 + u + 2\sqrt{1+u}},\ x > 16$	`Re_r1pz2_H2(x)`
6.	$h_3(x) = \sqrt{1 + x^2 - 2x \cdot \sqrt{1+x^2}},\ 0 \le x \le 1/\sqrt{3}$	`Re_r1pz2_H3(x)`
7.	$h_4(x) = \dfrac{0.75 \cdot x \cdot \sqrt{-3 - 3x^2 + 6 \cdot \sqrt{1 - x^2 + x^4}}}{1 + x^4 + \sqrt{1 - x^2 + x^4} + x^2(\sqrt{1 - x^2 + x^4} - 4)},\ 0 \le x \le 1/2$	`Im_r1pz2_fH4(x)`
8.	$h_4(x) = \dfrac{(1+x^2) \cdot \sqrt{1 - x^2 + x^4} - 1 + 4x^2 - x^4}{4x \cdot (1-x^2) \cdot \sqrt{1 + x^2 + 2 \cdot \sqrt{1 - x^2 + x^4}}},\ 1/2 < x < 1$	`Im_r1pz2_fH4(x)`
9.	$h_5(x) = u \cdot \dfrac{0.75 \cdot \sqrt{3u - 3 + 6 \cdot \sqrt{1 + u + u^2}}}{1 + 4u + u^2 + (1-u) \cdot \sqrt{1 + u + u^2}},\ x > 2$	`Im_r1pz2_fH1(x)`
10.	$h_5(x) = \dfrac{3(1 + x^4 + \sqrt{1 + x^2 + x^4}) \cdot \sqrt{3 - 3x^2 + 6\sqrt{1 + x^2 + x^4}}}{4x \cdot \left(4 + 2x^2 + 5x^4 + x^6 + (5 + x^4) \cdot \sqrt{1 + x^2 + x^4}\right)},\ x \le 2$	`Im_r1pz2_fH1(x)`
11.	$h_6(x) = \dfrac{(1+x^2) \cdot \sqrt{1 - x^2 + x^4} - 1 + 4x^2 - x^4}{4x \cdot (x^2-1) \cdot \sqrt{1 + x^2 + 2 \cdot \sqrt{1 - x^2 + x^4}}},\ 1 < x \le 8$	`Im_r1pz2_fH6(x)`
12.	$h_6(x) = u \cdot \dfrac{9 \cdot (1-u)/(1 - 4u + u^2 + (1+u) \cdot \sqrt{1 - u + u^2})}{4 \cdot \sqrt{1 + u + 2 \cdot \sqrt{1 - u + u^2}}},\ x > 8$	`Im_r1pz2_fH6(x)`

Obige Tabelle zeigt die große Vielfalt der notwendigen numerischen Funktionen, um den minimalen und maximalen Funktionswert $f(x,y)$ über dem Rand eines beliebigen Rechtecks Z aus dem 1. Quadranten berechnen zu können. Realisiert werden muss zusätzlich ein Algorithmus, um diese Berechnungen für ein beliebig vorgegebenes Rechteck Z, z.B. aus der oberen Halbebene, nur für ein entsprechendes Rechteck aus dem 1. Quadranten durchführen zu können. Außerdem benötigt man einen Algorithmus, der für ein beliebiges solches Rechteck aus dem 1. Quadranten die Punkte m, M ermittelt, über denen der minimale bzw. maximale Funktionswert $f(x,y)$ angenommen wird.

B.7.3. Numerische Beispiele

Im **1. Beispiel** wählen wir die Punktintervalle $X = [2,2]$ und $Y = [3,3]$. Mit dem Funktionsaufruf `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [7.49999999999999...9999 \cdot 10^{-2}, 7.50000000000000...00001 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **2. Beispiel** wählen wir $X = [-2,-2],\ Y = [-3,-3]$. Mit `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [7.49999999999999...9999 \cdot 10^{-2}, 7.50000000000000...00001 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **3. Beispiel** wählen wir $X = [+2,+2],\ Y = [-3,-3]$. Mit `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [-7.50000000000000...00001 \cdot 10^{-2}, -7.49999999999999...9999 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **4. Beispiel** wählen wir $X = [-2,-2],\ Y = [+3,+3]$. Mit `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [-7.50000000000000...00001 \cdot 10^{-2}, -7.49999999999999...9999 \cdot 10^{-2}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **5. Beispiel** wählen wir die extremen Punktintervalle: $X = [2^{-1073741824}, 2^{-1073741824}],\ Y = [1-2^{-300}, 1-2^{-300}] = [\mathtt{pred}(1), \mathtt{pred}(1)]$. Mit dem Aufruf `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die nahezu optimale Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [4.9432450833...6885 \cdot 10^{-323228317}, 4.9432450833...6887 \cdot 10^{-323228317}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen.

Im **6. Beispiel** wählen wir die extremen Punktintervalle: $X = [2^{-1073741824}, 2^{-1073741824}],\ Y = [1,1]$. Mit dem Aufruf `T = Im_r1pz2(X,Y)` erhalten wir mit der Präzision `prec` = 300 für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die notwendigerweise grobe Einschließung $W_f \subseteq \mathtt{T}$.

$$\mathtt{T} = [2.0985787164673876924043581168838390706380 9...55303 \cdot 10^{+323228496}, \mathtt{@Inf@}] \supset W_f$$

mit 90 ausgegeben Dezimalstellen. Beachten Sie dabei im Vergleich zum 5. Beispiel die großen Unterschiede bei den Funktionswerten in der Nähe der Singularität (0,1).

Im **7. Beispiel** wählen wir $X = [0,0], Y \not\ni \pm 1$ und erhalten mit dem Aufruf `T = Im_r1pz2(X,Y)` für den Wertebereich $W_f := \{f(x,y) \,|\, x \in X,\ y \in Y\} = [f_m, f_M]$ die exakte Einschließung $[0,0]$.

B.8. $x/\sqrt{1+x^2}$)

Für das vorgegebene reelle Maschinenintervall `X` $\ni x$ liefert die Intervallfunktion

```
MpfiClass xdsqrt1px2 (MpfiClass& X)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x) = \frac{x}{\sqrt{1+x^2}}, \quad \text{mit:} \quad x \in \texttt{X}.$$

Da $f(x)$ im ganzen Definitionsbereich $D_f = \mathbb{R}$ monoton wächst, ist die Implementierung recht einfach, da $f(x)$ nur auf den Intervallrändern `Inf(X)` und `Sup(X)` mit entsprechender Rundung auszuwerten ist.

B.9. $x/\sqrt{1-x^2}$)

Für das vorgegebene reelle Maschinenintervall `X` $\ni x$ liefert die Intervallfunktion

```
MpfiClass xdsqrt1mx2 (MpfiClass& X)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x) = \frac{x}{\sqrt{1-x^2}}, \quad \text{mit:} \quad x \in \texttt{X}.$$

Da $f(x)$ im ganzen Definitionsbereich $D_f = (-1, +1)$ monoton wächst, ist die Implementierung recht einfach, da $f(x)$ nur auf den beiden Intervallrändern `Inf(X)` und `Sup(X)` mit entsprechender Rundung auszuwerten ist.

B.10. $x/\sqrt{x^2-1}$)

Für das vorgegebene reelle Maschinenintervall `X` $\ni x$ liefert die Intervallfunktion

```
MpfiClass xdsqrtx2m1 (MpfiClass& X)
```

eine fast optimale Einschließung aller Funktionswerte

$$f(x) = \frac{x}{\sqrt{x^2-1}}, \quad \text{mit:} \quad x \in \texttt{X} \subseteq D_f := \{x \in \mathbb{R} \,|\, x < -1 \ \texttt{or}\ x > +1\}.$$

Da $f(x)$ im ganzen Definitionsbereich D_f monoton fällt, ist die Implementierung recht einfach, da $f(x)$ nur auf den beiden Intervallrändern `Inf(X)` und `Sup(X)` mit entsprechender Rundung auszuwerten ist.

C. Elementarfunktionen für komplexe Punkt- und Intervallargumente

Die Aufgabenstellung wird wie folgt beschrieben:

Zu einem vorgegebenen Intervallargument

$$Z = X + i \cdot Y; \quad X = [x_1, x_2], \quad Y = [y_1, y_2]; \tag{C.1}$$

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

$$W_{f,Z} := \{w \,|\, w = f(z) \wedge z \in Z\}, \; z = x + i \cdot y. \tag{C.2}$$

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 i.a. erhebliche Überschätzungen liefert, wenn das Rechteck F die Menge W **optimal** einschließt. Für $Z = [-1, 3] + i \cdot [0.2, 1]$ werden dies Überschätzungen in Abb. C.1 am Beispiel der komplexen sin-Funktion, d.h. für $f(z) = \sin(z)$, $z \in Z$, überzeugend dargestellt. So gehört z.B. der ganze von $F(Z)$ eingeschlossene Teil des 3. Quadranten **nicht** zur Wertemenge $W_{f,Z}$, [38, S.16].

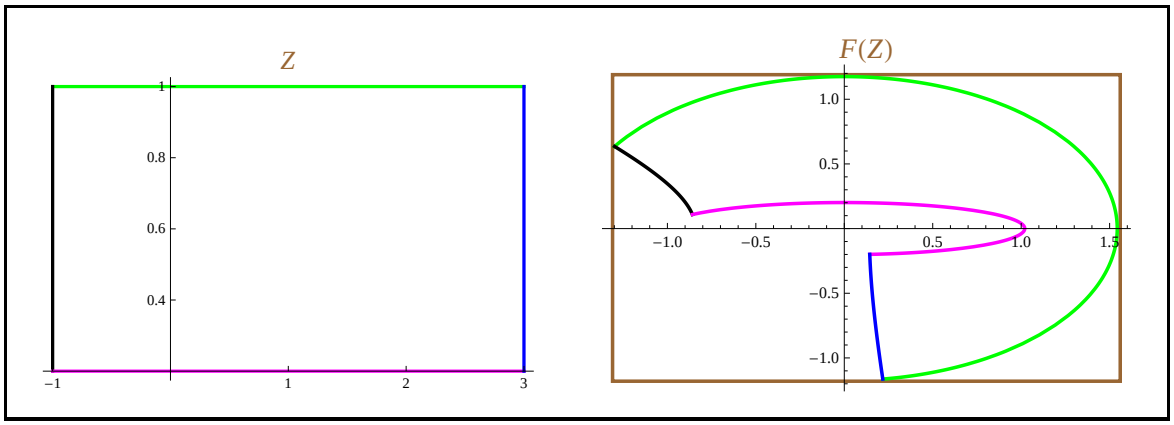


Abbildung C.1.: Intervall-Funktion

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, [5]. 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:

$$u(x,y) = e^x \cdot \cos(y), \qquad v(x,y) = e^x \cdot \sin(y), \quad x \in X,\ y \in Y. \tag{C.3}$$

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 die Tabelle auf Seite 68.

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, [7].

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 angenommen 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.

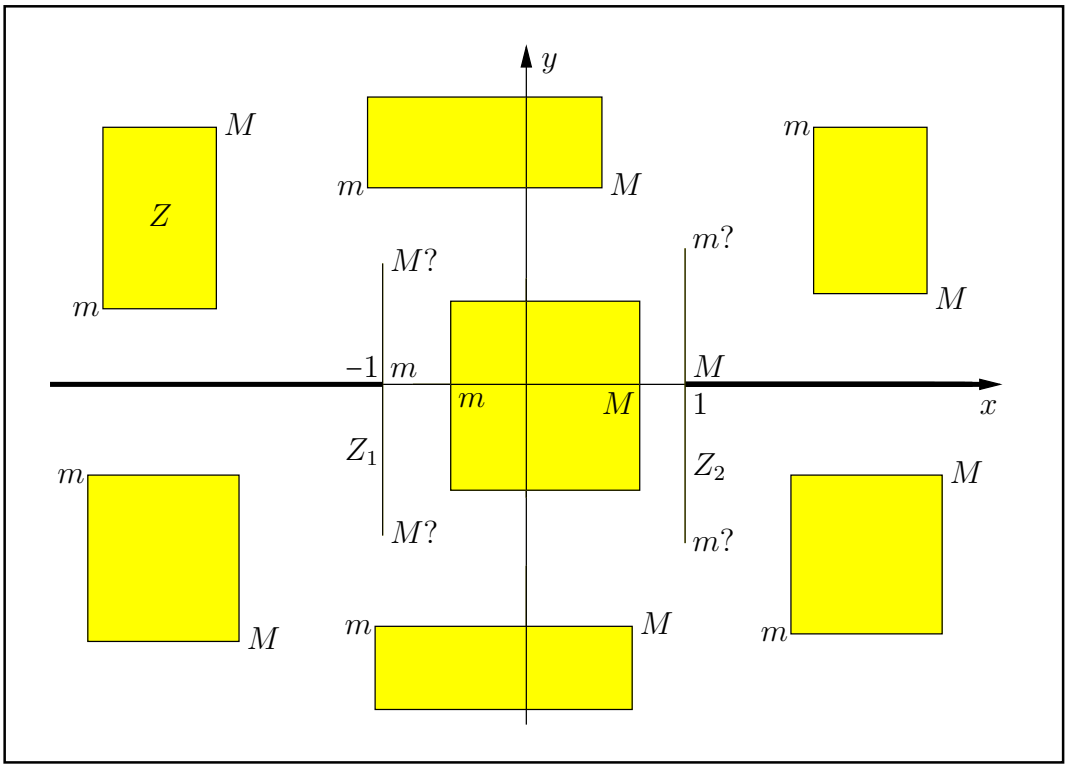


Abbildung C.2.: Die Lage der Punkte m, M auf Z beim Realteil von $\arcsin(z)$.

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. C.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 [17],[18],[38],[39] 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, [54],[55]

$$U(Z) = [\texttt{Inf}(\boldsymbol{u}(m)), \texttt{Sup}(\boldsymbol{u}(M))].$$

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. aufgerundete 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, [23],[24],[52],[60]. 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. C.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, [54],[55], 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. C.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, [38].

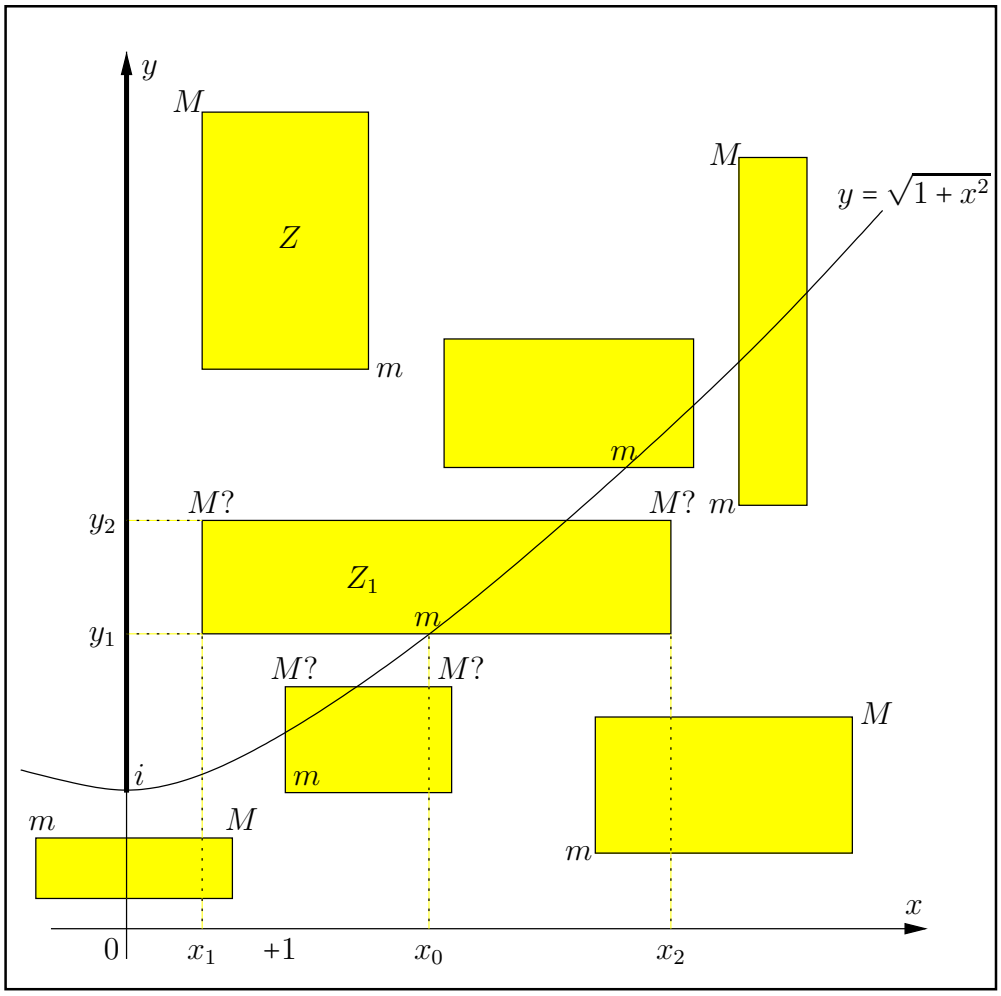


Abbildung C.3.: Die Lage der Punkte m, M auf Z beim Realteil von $\arctan(z)$.

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

$$u(x, y) := \frac{1}{2} \arctan \frac{2x}{1 - x^2 - y^2} + \frac{\pi}{2}, \tag{C.4}$$

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 (C.4) $x = \sqrt{y_1^2 - 1}$ und $y = y_1$

und erhält

$$u(x_0, y_1) := -\frac{1}{2}\arctan\frac{1}{\sqrt{y_1^2 - 1}} + \frac{\pi}{2} =: u(y_1), \tag{C.5}$$

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 245 mit drei verschiedenen Methoden berechnet werden.

Methode 1. Nach [17],[18],[38],[39] 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, [54],[55]

$$\texttt{Inf}(\boldsymbol{u}(y_1)) \le u(y_1) \le u(x, y), \quad z = x + i \cdot y \in Z..$$

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, [23],[24],[52],[60]. 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 [38, Seite 145] erfolgt dies durch eine allgemeine Monotoniebetrachtung der Funktion $u(x, y)$.

2. Nach Abb. C.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 (C.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.

C.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 179. Die Elementarfunktionen für komplexe Punktargumente sind in der Tabelle auf Seite 92 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.

C.1.1. Exponentialfunktion, Realteil

Nach (C.3) von Seite 244 gilt mit $z = x + i \cdot y \in \mathbb{C}$ für den Realteil $u(x, y)$ der Exponentialfunktion

$$\Re(e^z) := u(x, y) = e^x \cdot \cos(y). \tag{C.6}$$

Zunächst sollen nur **aufgerundete** Funktionswerte von $u(x, y)$ berechnet werden. Nach (A.15) und (A.16) von Seite 181 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 $\cos(y)$ berechnet 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`.

C.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 183. Weitere Einzelheiten findet man in der Funktion `MpfcClass sin(const MpfcClass& z, RoundingMode rnd);` die in der Datei `mpfcclass.cpp` definiert ist.

C.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.

C.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.

C.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.

C.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.

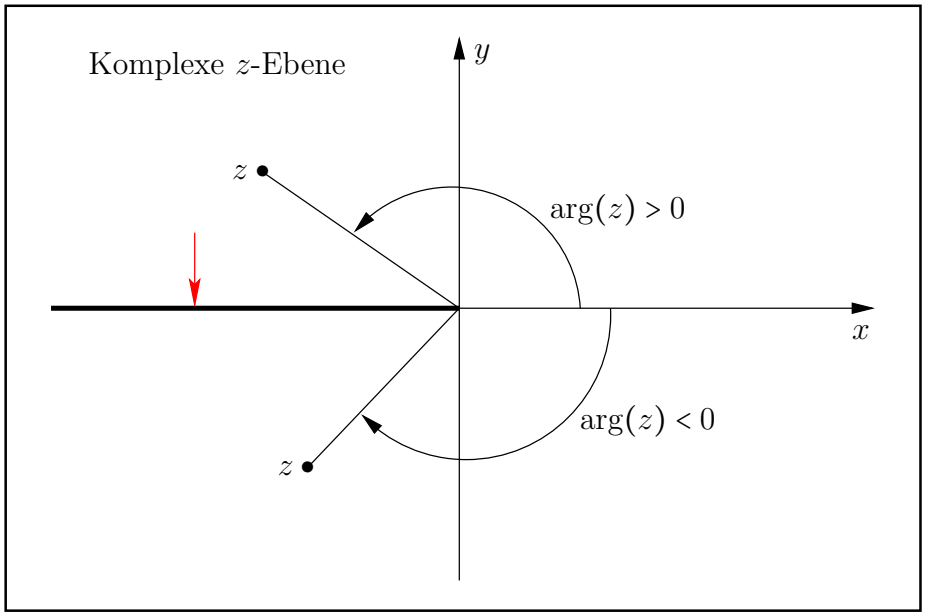


Abbildung C.4.: Verzweigungsschnitt von $\arg(z)$, $z \in \mathbb{C}$.

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. Dabei bedeuten die angegebenen Werte wie $\pi/2$ oder $-3\pi/4$ die bezüglich des voreingestellten Current-Rundungsmodus entsprechend gerundeten Werte.

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 C.5.: Wertetabelle für $\arg(z)$, $z \in \mathbb{C}$.

C.1.7. argp1(z)

Für $z = x + i \cdot y \in \mathbb{C}$ ist das Argument von $1 + z$, d.h. argp1(z), definiert durch das Bogenmaß des Winkels, den der Fahrstrahl vom Ursprung nach $1 + z$ mit der positiven reellen Achse einschließt. Die komplexe Ebene ist dabei längs der negativen reellen Achse von -1 bis $-\infty$ aufgeschnitten.

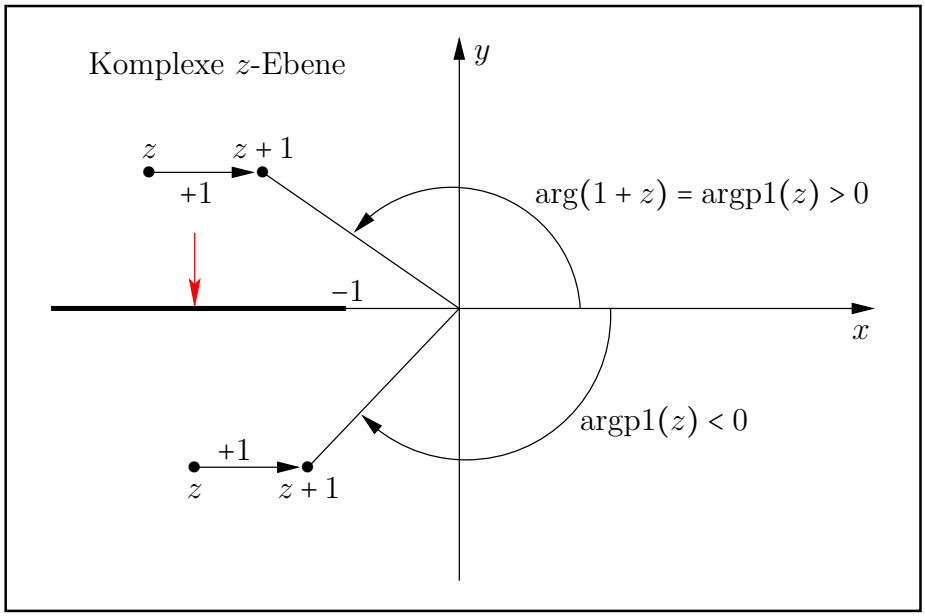


Abbildung C.6.: Verzweigungsschnitt von argp1(z), $z \in \mathbb{C}$.

Der Pfeil zeigt die Richtung an, aus der argp1(z) auf den Verzweigungsschnitt stetig ergänzt wird, d.h. es gilt z.B. argp1$(-2 + 0 \cdot i) = +\pi$. Der zu rundende exakte Funktionswert $\arg(1 + z)$ wird bestimmt mit der Funktion

```
MpfrClass argp1(const MpfcClass& z, RoundingMode rnd, PrecisionType prec);
```

Wird `prec` gesetzt, so wird $\arg(1 + z)$ 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 $\arg(1 + z)$ 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 $\arg(1 + z)$ zusammengestellt. Dabei bedeuten die angegebenen Werte wie $\pi/2$ oder $-3\pi/4$ die bezüglich des voreingestellten Current-Rundungsmodus (`RoundNearest`) entsprechend gerundeten Werte.

x	y	$\arg(1+z)$	x	y	$\arg(1+z)$	x	y	$\arg(1+z)$
-1	-1	0	-1	$+\infty$	$\pi/2$	$-\infty$	$+1$	π
$+1$	0	0	-1	-1	$-\pi/2$	$+\infty$	$+\infty$	$+\pi/4$
$+\infty$	0	0	-1	$-\infty$	$-\pi/2$	$+\infty$	$-\infty$	$-\pi/4$
$+\infty$	$+1$	0	-2	0	π	$-\infty$	$+\infty$	$+3\pi/4$
$+\infty$	-1	0	$-\infty$	0	π	$-\infty$	$-\infty$	$-3\pi/4$
-1	$+1$	$\pi/2$	$-\infty$	-1	$-\pi$	x oder y = NaN		NaN

Abbildung C.7.: Wertetabelle für $\arg(1 + z)$, $z \in \mathbb{C}$.

C.1.8. $|z|$

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.

C.1.9. $\log(z)$

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.

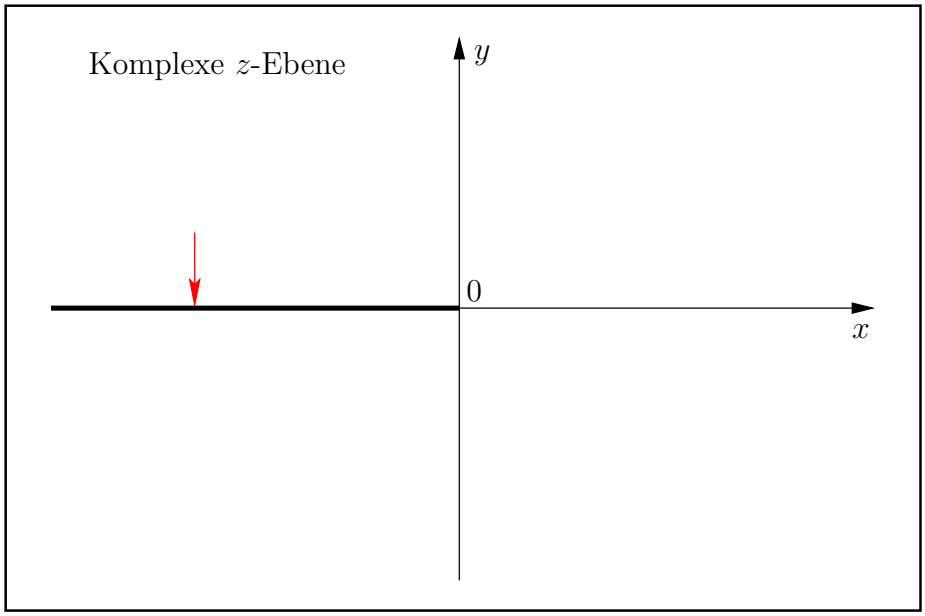


Abbildung C.8.: Verzweigungsschnitt von $\log(z)$, $z \in \mathbb{C}$.

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.

C.1.10. $1/z$

Für $z = x + i \cdot y \in \mathbb{C}$ ist $1/z$ definiert durch

$$f(z) := \frac{1}{z} = u(x,y) + i \cdot v(x,y) = \frac{x}{x^2+y^2} + i \cdot \frac{-y}{x^2+y^2}, \quad x \neq 0 \wedge y \neq 0. \tag{C.7}$$

Die Funktionswerte $f(z)$ werden nahezu optimal berechnet mit

```
MpfcClass reci(const MpfcClass& z, RoundingMode rnd);
```

Für `rnd` sind die folgenden Rundungsmodi zugelassen: `RoundUp, RoundDown, RoundNearest`. Wird `rnd` nicht gesetzt, so wird mit dem Current-Rundungsmodus gerundet.

C.1.10.1. Realteil

Für die Realteilfunktion $u(x,y)$ gilt mit $x \neq 0 \wedge y \neq 0$:

$$u(x,y) = \begin{cases} 0, & x = 0 \\ 1/x, & y = 0 \\ \dfrac{x}{x^2+y^2}, & \text{sonst.} \end{cases} \tag{C.8}$$

Die Auswertung von $u(x,y)$ erfolgt mit Hilfe der Funktion

```
MpfrClass x_div_x2py2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfrclass.cpp` definiert ist. Eine ausführliche Beschreibung der Funktion `x_div_x2py2` findet man auf Seite 185.

C.1.10.2. Imaginärteil

Für die Imaginärteilfunktion $v(x,y)$ gilt mit $x \neq 0 \wedge y \neq 0$:

$$v(x,y) = \begin{cases} 0, & y = 0 \\ 1/y, & x = 0 \\ \dfrac{-y}{x^2+y^2}, & \text{sonst.} \end{cases} \tag{C.9}$$

Die Auswertung von $v(x,y)$ erfolgt mit Hilfe der gleichen Funktion `x_div_x2py2(...)` durch den Aufruf

```
x_div_x2py2(-Im(z), Re(z), rnd);
```

Dabei ist zu beachten, dass der obige Aufruf nicht durch `-x_div_x2py2(Im(z), Re(z),rnd)` ersetzt werden darf, da das Vorzeichen `-` erst zum Schluss zur Anwendung kommt und daher die Rundungen bez. `rnd` vorher falsch berechnet wird.

Numerisches Beispiel:
Mit $x = 2$ und $y = -3$, d.h. mit $z = 2 - 3 \cdot i$, und `prec` = 400 liefern `reci(z,` RoundUp) und `reci(z,` RoundDown) den nahezu optimal auf- bzw. abgerundeten Wert von $1/z$:

$$\begin{aligned} &(0.153846153846153846153846153846153846153846153846153846153846\ldots 153846^{169}_{072}, \\ &\ 0.230769230769230769230769230769230769230769230769230769230769\ldots 230769^{254}_{157}) \end{aligned}$$

mit gemeinsamen 120 Dezimalstellen bei Real- und Imaginärteil.

C.1.11. $1/z^2$

Für $z = x + i \cdot y \in \mathbb{C}$ ist $1/z^2$ definiert durch

$$f(z) := \frac{1}{z^2} = u(x,y) + i \cdot v(x,y) = \frac{x^2 - y^2}{(x^2+y^2)^2} + i \cdot \frac{-2xy}{(x^2+y^2)^2}, \quad x \neq 0 \wedge y \neq 0. \tag{C.10}$$

Die Funktionswerte $f(z)$ werden nahezu optimal berechnet mit

```
MpfcClass reci_z2(const MpfcClass& z, RoundingMode rnd);
```

Für `rnd` sind die folgenden Rundungsmodi zugelassen: `RoundUp, RoundDown, RoundNearest`. Wird `rnd` nicht gesetzt, so wird mit dem Current-Rundungsmodus gerundet.

C.1.11.1. Realteil

Für die Realteilfunktion $u(x,y)$ gilt mit $x \neq 0 \wedge y \neq 0$:

$$u(x,y) = \frac{x^2 - y^2}{(x^2+y^2)^2}. \tag{C.11}$$

Die Auswertung von $u(x,y)$ erfolgt mit Hilfe der Funktion

```
MpfrClass Re_rz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfrclass.cpp` definiert ist. Eine ausführliche Beschreibung der Funktion `Re_rz2(...)` findet man auf Seite 186.

C.1.11.2. Imaginärteil

Für die Imaginärteilfunktion $v(x,y)$ gilt mit $x \neq 0 \wedge y \neq 0$:

$$v(x,y) = \frac{-2xy}{(x^2+y^2)^2} \tag{C.12}$$

Die Auswertung von $v(x,y)$ erfolgt mit Hilfe der Funktion

```
MpfrClass Im_rz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfrclass.cpp` definiert ist. Eine ausführliche Beschreibung der Funktion `mIm_rz2(...)` findet man auf Seite 187, wobei zu beachten ist, dass `mIm_rz2(...)` den Term $+2xy/(x^2+y^2)^2$ auswertet.

C.1.11.3. Numerische Ergebnisse

Im **1. Beispiel** wählen wir $z = 1 + 2 \cdot i$ und erhalten für das exakte Ergebnis $1/z^2 = -0.12 - 0.16 \cdot i$ mit `rnd` = `RoundDown` und der Präzision `prec` = 300 den nahezu optimal abgerundeten Wert mit 93 ausgegebenen Dezimalstellen

$$(-1.200000000000\ldots00000000002 \cdot 10^{-1},\ -1.600000000000\ldots00000000044 \cdot 10^{-1}).$$

Im **2. Beispiel** wählen wir $z = 1 + 2 \cdot i$ und erhalten für das exakte Ergebnis $1/z^2 = -0.12 - 0.16 \cdot i$ mit `rnd` = `RoundUp` und der Präzision `prec` = 300 den nahezu optimal aufgerundeten Wert mit 93 ausgegebenen Dezimalstellen

$$(-1.19999999999999\ldots99999999941 \cdot 10^{-1},\ -1.59999999999999\ldots99999999921 \cdot 10^{-1}).$$

C.1.12. $1/(1+z^2)$

Für $z = x + i \cdot y \in \mathbb{C}$ ist $1/(1+z^2)$ definiert durch

$$f(z) := \frac{1}{1+z^2} = u + i \cdot v = \frac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2} + i \cdot \frac{-2xy}{4x^2y^2+(1+x^2-y^2)^2}, \quad x \neq 0 \wedge y \neq \pm 1. \tag{C.13}$$

Die Funktionswerte $f(z)$ werden nahezu optimal berechnet mit

```
MpfcClass reci_1pz2(const MpfcClass& z, RoundingMode rnd);
```

Für `rnd` sind die folgenden Rundungsmodi zugelassen: `RoundUp, RoundDown, RoundNearest`. Wird `rnd` nicht gesetzt, so wird mit dem Current-Rundungsmodus gerundet.

C.1.12.1. Realteil

Für die Realteilfunktion $u(x,y)$ gilt mit $x \neq 0 \wedge y \neq \pm 1$:

$$u(x,y) = \frac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2}. \tag{C.14}$$

Die Auswertung von $u(x,y)$ erfolgt mit Hilfe der Funktion

```
MpfrClass Re_r1pz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfrclass.cpp` definiert ist. Eine ausführliche Beschreibung der Funktion `Re_r1pz2(...)` findet man auf Seite 189.

C.1.12.2. Imaginärteil

Für die Imaginärteilfunktion $v(x,y)$ gilt mit $x \neq 0 \wedge y \neq \pm 1$:

$$v(x,y) = \frac{-2xy}{4x^2y^2+(1+x^2-y^2)^2} \tag{C.15}$$

Die Auswertung von $v(x,y)$ erfolgt mit Hilfe der Funktion

```
MpfrClass mIm_r1pz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfrclass.cpp` definiert ist. Eine ausführliche Beschreibung der Funktion

```
MpfrClass Im_r1pz2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

welche $-v(x,y)$ auswertet, findet man auf Seite 188.

C.1.12.3. Numerische Ergebnisse

Im **1. Beispiel** wählen wir $z = 2 + 3 \cdot i$, und für den exakten Funktionswert $1/(1+z^2) = -1/40 - 3/40 \cdot i$ erhalten wir mit `rnd` = `RoundDown` und der Präzision `prec` = 300 den nahezu optimal abgerundeten Wert mit 93 ausgegebenen Dezimalstellen

$$(-2.500000000000\ldots000000000030 \cdot 10^{-2},\ -7.500000000000\ldots000000000245 \cdot 10^{-2}).$$

Im **2. Beispiel** wählen wir $z = 2 + 3 \cdot i$, und für das exakte Ergebnis $1/(1+z^2) = -1/40 - 3/40 \cdot i$ erhalten wir mit `rnd` = `RoundUp` und `prec` = 300 den nahezu optimal aufgerundeten Wert mit 93 ausgegebenen Dezimalstellen

$$(-2.499999999999\ldots99999999877 \cdot 10^{-2},\ -7.499999999999\ldots999999999631 \cdot 10^{-2}).$$

C.1.13. 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 Minimum 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.

C.1.14. $\sqrt{z}$

Mit $z = x + i \cdot y \in \mathbb{C}$ ist der Hauptwert der komplexen Quadratwurzel definiert durch

$$\sqrt{z} := \begin{cases} \sqrt{x} + i \cdot 0, & \text{falls } \ y = 0 \wedge x \geq 0 \\ 0 + i \cdot \sqrt{|x|}, & \text{falls } \ y = 0 \wedge x < 0 \\ \dfrac{\sqrt{2 \cdot (|z| + x)}}{2} + i \cdot \dfrac{y}{\sqrt{2 \cdot (|z| + x)}}, & \text{falls } \ y \neq 0. \end{cases} \tag{C.16}$$

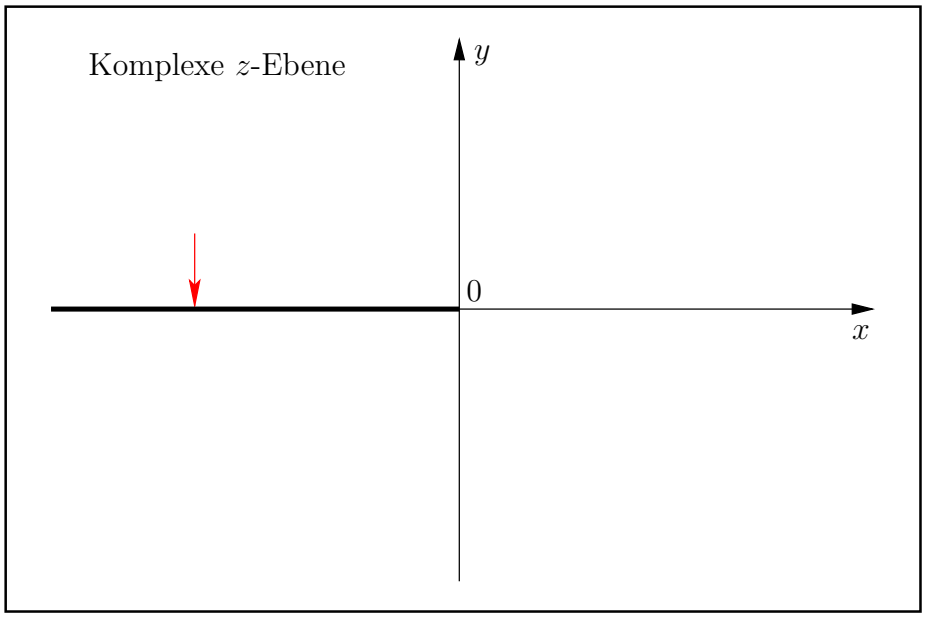


Abbildung C.9.: Verzweigungsschnitt von $\sqrt{z}$, $z \in \mathbb{C}$.

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 (C.16) 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

$$\Re(\sqrt{z}) = \begin{cases} \dfrac{\sqrt{2 \cdot (|z| + x)}}{2}, & \text{falls } \; y \neq 0 \wedge x \geq 0 \\ \dfrac{|y|}{\sqrt{2 \cdot (|z| + |x|)}}, & \text{falls } \; y \neq 0 \wedge x < 0. \end{cases} \tag{C.17}$$

$$\Im(\sqrt{z}) = \begin{cases} \dfrac{y}{\sqrt{2 \cdot (|z| + x)}}, & \text{falls } \; y \neq 0 \wedge x \geq 0 \\ \dfrac{\operatorname{sign}(y)}{2} \cdot \sqrt{2 \cdot (|z| + |x|)}, & \text{falls } \; y \neq 0 \wedge x < 0. \end{cases} \tag{C.18}$$

Die reellen Wurzeln in (C.17) und (C.18) 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.

C.1.15. $z^2 + a \cdot z + b$

Mit den Maschinenzahlen $z, a, b \in \mathbb{C}$ berechnet die Funktion

```
MpfcClass poly2(const MpfcClass& z, const MpfcClass& a,
                const MpfcClass& b, RoundingMode rnd);
```

nahezu optimal gerundete Funktionswerte des Polynoms $z^2 + a \cdot z + b$, wobei für den Rundungsparameter `rnd` nur die Rundungsmodi `RoundDown, RoundUp, RoundNearest` erlaubt sind. Ohne `rnd` wird nach dem Current-Rundungsmodus gerundet; ist dieser jedoch nicht gesetzt, so wird mit `RoundNearest` gerundet. Die Implementierung erfolgt in `mpfcclass.cpp` mit Hilfe der gleichnamigen Intervallfunktion, wobei die komplexen Zahlen z, a, b als Punktintervalle übernommen werden. Weitere Einzelheiten und numerische Ergebnisse findet man auf Seite 291.

C.1.16. $1/\sqrt{z}$

Mit $z = x + i \cdot y \in \mathbb{C}\backslash\{0\}$ ist der Hauptwert der Funktion $f(z) = 1/\sqrt{z}$ definiert durch

$$1/\sqrt{z} := \begin{cases} \dfrac{1}{\sqrt{x}} + i \cdot 0, & \text{falls} \quad y = 0 \wedge x > 0 \\ 0 - \dfrac{i}{\sqrt{-x}}, & \text{falls} \quad y = 0 \wedge x < 0 \\ \dfrac{\sqrt{2 \cdot (|z| + x)}}{2 \cdot |z|} - i \cdot \dfrac{y}{|z| \cdot \sqrt{2 \cdot (|z| + x)}}, & \text{falls} \quad y \neq 0. \end{cases} \tag{C.19}$$

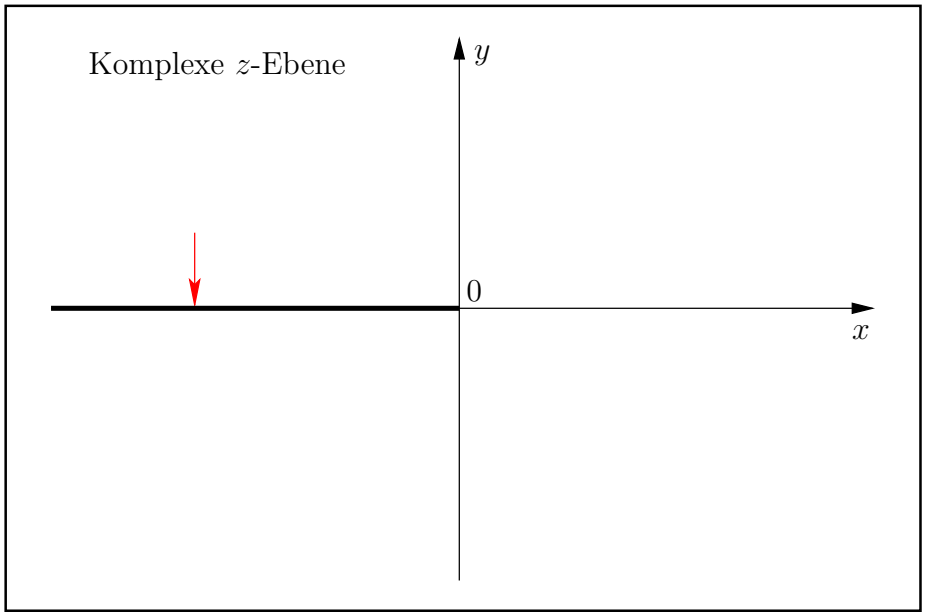


Abbildung C.10.: Verzweigungsschnitt von $1/\sqrt{z}$, $z \in \mathbb{C}\backslash\{0\}$.

Der obige Pfeil gibt die Richtung an, aus der $1/\sqrt{z}$ auf den Verzweigungsschnitt analytisch fortgesetzt wird, d.h. z.B. $1/\sqrt{-1 + 0 \cdot i} = -i$.

Nach (C.19) 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

$$\Re(1/\sqrt{z}) = \begin{cases} \dfrac{\sqrt{2 \cdot (|z| + x)}}{2 \cdot |z|}, & \text{falls} \quad y \neq 0 \wedge x \geq 0 \\ \dfrac{|y|/|z|}{\sqrt{2 \cdot (|z| + |x|)}}, & \text{falls} \quad y \neq 0 \wedge x < 0. \end{cases} \tag{C.20}$$

$$\Im(1/\sqrt{z}) = \begin{cases} -\dfrac{y/|z|}{\sqrt{2 \cdot (|z| + x)}}, & \text{falls} \quad y \neq 0 \wedge x \geq 0 \\ -\dfrac{\operatorname{sign}(y)}{2 \cdot |z|} \cdot \sqrt{2 \cdot (|z| + |x|)}, & \text{falls} \quad y \neq 0 \wedge x < 0. \end{cases} \tag{C.21}$$

Die reellen Wurzeln in (C.20) und (C.21) können jetzt ohne Auslöschung und ohne vorzeitigen Über- oder Unterlauf 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. Ein weiterer möglicher Überlauf bei der Auswertung von C.20 bzw. C.21 wird vermieden, wenn man $|x|, |y| < 2^{1073741821}$ verlangt, und dies ist wegen `expo(MaxFloat())` = 1073741823 keine wirkliche Einschränkung.

Die Auswertung der dann nahezu optimal gerundeten Werte von $1/\sqrt{z}$ erfolgt mit Hilfe der Funktion

```
MpfcClass sqrt_r(const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` implementiert ist.

C.1.17. $\sinh(z)$

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.

C.1.18. $\cosh(z)$

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.

C.1.19. $\tanh(z)$

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.

C.1.20. $\coth(z)$

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.

C.1.21. 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 $+\infty$.

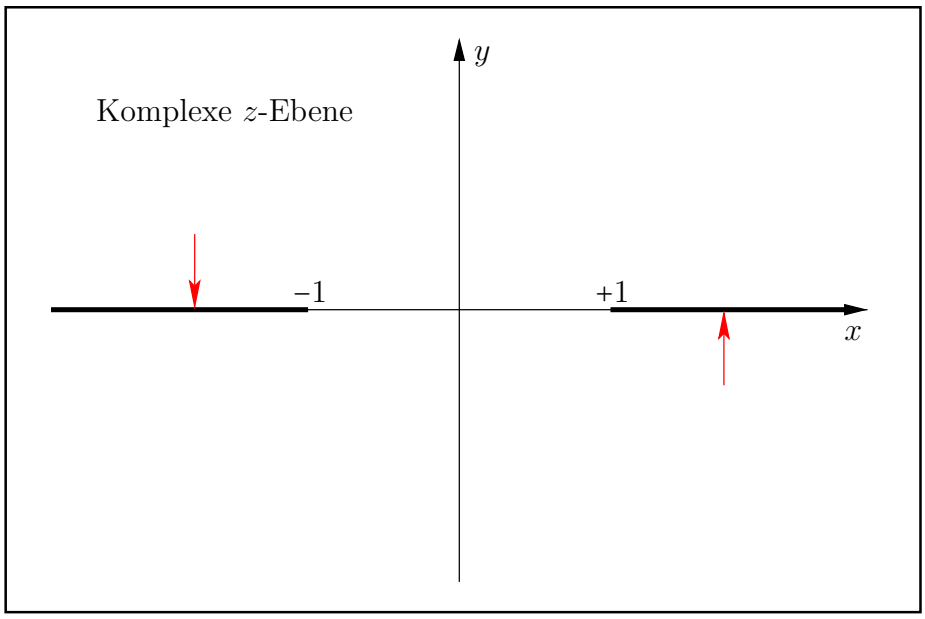


Abbildung C.11.: Verzweigungsschnitte von $\arcsin(z)$, $z \in \mathbb{C}$.

Die Pfeile geben die jeweilige Richtung an, aus denen die Funktionswerte von $\arcsin(z)$ auf den jeweiligen Verzweigungsschnitt analytisch fortgesetzt werden.

C.1.21.1. Realteil

Mit $z = x + i \cdot y \in \mathbb{C}$, $\arcsin(z) = u(x,y) + i \cdot v(x,y)$ und

$$T(x,y) := \frac{1}{2}\left(\sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2}\right) \tag{C.22}$$

gilt z.B. nach [38] für die Realteilfunktion

$$\Re(\arcsin(z)) = u(x,y) = \arcsin\frac{x}{T}. \tag{C.23}$$

Da die reelle arcsin-Funktion rechts in (C.23) 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

$$T(x,y) \equiv T(|x|,|y|) = \frac{1}{2}\left(\sqrt{(|x|+1)^2+|y|^2} + \sqrt{(|x|-1)^2+|y|^2}\right) \geq T(|x|,0), \tag{C.24}$$

$$T(|x|,0) = \frac{1}{2}\{(|x|+1) + ||x|-1|\} = \begin{cases} |x|, & \text{falls } |x| \geq 1, \\ 1, & \text{falls } |x| \leq 1, \end{cases} \tag{C.25}$$

[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 (C.24) 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 (C.24) 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 obige 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 (C.24) zu erhalten. In (C.24) 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 (C.23) und (C.25)

$$u(x,0) = \begin{cases} \arcsin(x), & |x| \leq 1, \\ \arcsin(+1) = +\pi/2, & x \geq +1, \\ \arcsin(-1) = -\pi/2, & x \leq -1. \end{cases}$$

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 (C.23) 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 (C.24) und (C.25) 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 (C.23) 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`.

C.1.21.2. Imaginärteil

Mit der in (C.22) bereits definierten Funktion $T(x,y)$ gilt nach [38] für die Imaginärteilfunktion

$$v(x,y) := \begin{cases} +\log(T+\sqrt{T^2-1}), & \text{falls } y>0 \\ +\log(T+\sqrt{T^2-1}), & \text{falls } y=0 \text{ und } x\le -1 \\ 0 \quad , & \text{falls } y=0 \text{ und } |x|\le +1 \\ -\log(T+\sqrt{T^2-1}), & \text{falls } y=0 \text{ und } x\ge +1 \\ -\log(T+\sqrt{T^2-1}), & \text{falls } y<0. \end{cases} \tag{C.26}$$

Wegen $T(x,y)\ge 1$ gilt zusätzlich

$$\operatorname{arcosh}(T) \equiv \log(T+\sqrt{T^2-1}), \tag{C.27}$$

so dass in (C.26) 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 (C.27) 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 (C.27) 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. die Tabelle auf Seite 37.

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 implementierten, monoton wachsenden Funktion `sqrtp1m1(...)` direkt ausgewertet werden kann, vgl. die Tabelle auf Seite 37. 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!

C.1.22. $\arccos(z)$

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. C.11 auf Seite 261. Für den Hauptwert gelten mit $T(x,y)$ von Seite 261 die Beziehungen

$$\Re(\arccos(z)) = \arccos(x/T), \tag{C.28}$$

$$\Im(\arccos(z)) = -\Im(\arcsin(z)). \tag{C.29}$$

Der Realteil von $\arccos(z)$ wird im Vergleich zu (C.23) 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 (C.29) 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 180 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. Die Funktionswerte $f(z) = \arccos(z)$ werden berechnet mit Hilfe der Funktion

```
MpfcClass acos (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist. Die Funktionswerte $f(z)$, mit $z \in \mathbb{C}\backslash\{\pm 1\}$, werden mit Hilfe des Parameters `rnd` $\in$ `{RoundDown, RoundUp, RoundNearest}` entsprechend nahezu optimal gerundet. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus.

C.1.23. $1/(1-z^2)$

Setzt man $\hat{z} := i \cdot z$, so gilt $f(z) := 1/(1-z^2) = 1/(1+\hat{z}^2)$, d.h. $f(z)$ kann mit Hilfe der bereits definierten Funktion `reci_r1pz2`$(\hat{z})$ ausgewertet werden. Dies wird realisiert in der Funktion

```
MpfcClass reci_1mz2 (const MpfcClass& z, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist. Die Funktionswerte $f(z)$, mit $z \in \mathbb{C}\backslash\{\pm 1\}$, werden mit Hilfe des Parameters `rnd` $\in$ `{RoundDown, RoundUp, RoundNearest}` entsprechend nahezu optimal gerundet. Wird `rnd` nicht gesetzt, so erfolgt die Rundung nach dem Current-Rundungsmodus.

C.1.24. $\log(1+z)$

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.

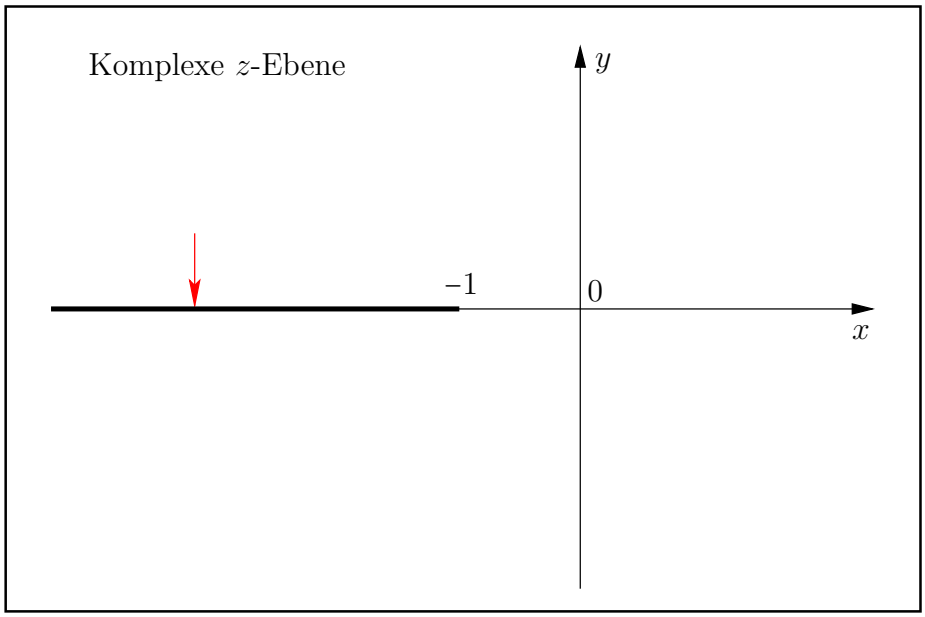


Abbildung C.12.: Verzweigungsschnitt von $\log(1+z)$, $z \in \mathbb{C}$.

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 Funktionswertes 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.

C.1.24.1. Realteil

Die auszuwertende Realteilfunktion lautet

$$u(x,y) := \frac{1}{2}\cdot\ln\big((1+x)^2+y^2\big) = \ln\big(\sqrt{(1+x)^2+y^2}\big); \tag{C.30}$$

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

$$2^{\mathtt{k1}} < \mathtt{pred}(\sqrt{M/2}) \le \sqrt{M/2}-1, \quad k1 = 536870909 \tag{C.31}$$

wird damit ein vorzeitiger Überlauf vermieden, wenn gilt

$$\mathtt{expo}(x) < \mathtt{k1} \ \wedge\ \mathtt{expo}(y) < \mathtt{k1}, \quad \mathtt{k1} = 536870909. \tag{C.32}$$

Zu beachten ist, dass in (C.31) M eine von `prec` abhängige Größe ist und dass $2^{\mathtt{k1}}$ eine Unterschranke für alle `prec` ≥ 2 ist. Nach (C.32) kann für alle `prec` ≥ 2 ein Überlauf nur auftreten, wenn gilt:

$$\mathtt{expo}(x) \ge \mathtt{k1} \ \text{ oder } \ \mathtt{expo}(y) \ge \mathtt{k1}, \quad \mathtt{k1} = 536870909. \tag{C.33}$$

Wir betrachten jetzt den Fall, dass mit (C.33) 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 17 für x,y die für alle `prec` ≥ 2 gültige Oberschranke 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 17 verlangt

$$2\cdot(1+2^{1073741824-k})^2 < 2^{1073741823} \quad\Longleftrightarrow\quad 1+2^{1073741824-k} < 2^{536870911}, \tag{C.34}$$

und wegen $1+2^{1073741824-k} < 2^{1073741825-k}$ ist die letzte Ungleichung in (C.34) erfüllt, wenn gilt

$$2^{1073741825-k} < 2^{536870911} \quad\Longleftrightarrow\quad k > 536870914.$$

Zusammenfassung:
Im Fall (C.33) wird ein Überlauf bei der Auswertung von β vermieden, wenn $u(x,y)$ mithilfe der Konstanten `Ln2(rnd)` wie folgt berechnet wird, vgl. Seite 36:

$$u(x,y) = k\cdot\ln(2)+\frac{1}{2}\cdot\ln\big[(2^{-k}+2^{-k}\cdot x)^2+(2^{-k}\cdot y)^2\big], \quad k = 536870915. \tag{C.35}$$

Wir betrachten jetzt den Fall, dass bei der Berechnung 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. C.13 noch folgende Punkte zu beachten:

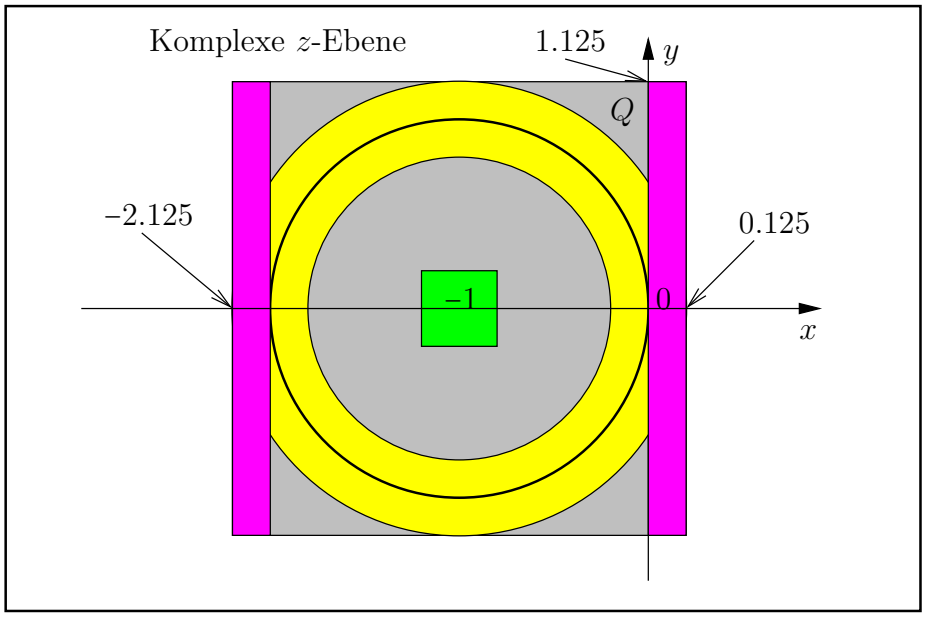


Abbildung C.13.: Verschiedene Bereiche zur Auswertung von $u(x,y)$.

1. Im grünen Quadrat mit dem Mittelpunkt $M = (-1,0)$ und der Kantenlänge $2 \cdot 0.125 = 0.25$ gilt $\ln((1+x)^2 + y^2)/2 = \ln(|1+x|) + \ln(1 + (y/(1+x))^2)/2$, wobei in $-1.25 \le x \le -0.75$ die Summe $(1 \oplus x)$ für alle `prec` ≥ 2 **exakt** berechnet wird. Mit dem exakten Argument $|1 \oplus x| \ll 1$ kann daher die Logarithmusfunktion in der Nähe ihrer Singularität 0 optimal ausgewertet werden. Würde man das Argument $(1+x)^2+y^2$ direkt auswerten, so müsste die Logarithmusfunktion in der Nähe ihrer Singularität 0 mit einem fehlerbehafteten Argument berechnet werden, was starke Überschätzungen zur Folge hätte. Die obige Summe der Logarithmusfunktionen führt nicht zu Auslöschungseffekten, da das grüne Quadrat vom gelben Kreisring weit genug entfernt ist.

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. C.13 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. Der Fall $x = 0$, d.h. $\delta = y^2$, wird gesondert behandelt.

4. Im Fall $|x| \ll 1 \wedge |y| \ll 1$ wird δ intervallmäßig berechnet durch: $\delta = x \cdot (2 + x + (y/x) \cdot y)$, um optimal gerundete δ-Werte berechnen zu können.

5. 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 durch schrittweise Verdoppelung der Current-Präzision bis zur exakten Berechnung von δ vermieden wird.

6. 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: $x \cdot (2+x) > 0$.

7. 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`.

Numerische Ergebnisse:

Bezeichnungen: $t = \ln(\sqrt{(1+x)^2+y^2})$, `t = ln_sqrtxp1_2y2(x,y,rnd);`

1. Mit `prec` = 30000, `rnd = RoundUp` und x = `succ`$(-2) = -2 + 2^{-29\,999}$, y = `minfloat(prec)` erhält man für t die nahezu optimale Oberschranke mit 9030 Dezimalstellen:

   $t <$ `t` $= -2.5186050871681829145830615704304081184503391605...47876326915e-9031;$

2. Mit `prec` = 30000, `rnd = RoundDown`, x = `succ`$(-2) = -2 + 2^{-29\,999}$, y = `minfloat(prec)` erhält man für t die nahezu optimale Unterschranke mit 9030 Dezimalstellen:

   $t >$ `t` $= -2.5186050871681829145830615704304081184503391605...47876326925e-9031;$

3. Mit `prec` = 30000, `rnd = RoundUp` und x = `pred`$(-2) = -2 - 2^{-29\,998}$, y = `minfloat(prec)` erhält man für t die nahezu optimale Oberschranke mit 9030 Dezimalstellen:

   $t <$ `t` $= 5.0372101743363658291661231408608162369006783210 0...95752653837e-9031;$

4. Mit `prec` = 30000, `rnd = RoundDown`, x = `pred`$(-2) = -2 - 2^{-29\,998}$, y = `minfloat(prec)` erhält man für t die nahezu optimale Unterschranke mit 9030 Dezimalstellen:

   $t >$ `t` $= 5.0372101743363658291661231408608162369006783210 0...95752653824e-9031;$

5. Mit `prec` = 300 000, `rnd = RoundUp` und $x = -1$, y = `minfloat(prec)` erhält man für t die nahezu optimale Oberschranke mit 90308 Dezimalstellen:

   $t <$ `t` $= -7.4426111795489301787390319512589204791944768398...268012457556477e8;$

6. Mit `prec` = 300 000, `rnd = RoundDown` und $x = -1$, y = `minfloat(prec)` erhält man für t die nahezu optimale Unterschranke mit 90308 Dezimalstellen:

   $t >$ `t` $= -7.4426111795489301787390319512589204791944768398...268012457556488e8;$

7. Mit `prec` = 300 000, `rnd = RoundUp` und $x = -2^{-300\,001}$, $y = 2^{-150\,000}$ erhält man für t die nahezu optimale Oberschranke mit 90308 Dezimalstellen:

   $t <$ `t` $= 1.2575105123417387304499560989212836168940085103 11...06698175e-180619;$

8. Mit `prec` = 300 000, `rnd = RoundDown` und $x = -2^{-300\,001}$, $y = 2^{-150\,000}$ erhält man für t die nahezu optimale Unterschranke mit 90308 Dezimalstellen:

   $t >$ `t` $= 1.2575105123417387304499560989212836168940085103 11...06698174e-180619;$

Beachten Sie, dass bei den beiden letzten Beispielen wegen $y^2 \approx -2x$ starke Auslöschung bei der Auswertung von $x\cdot(2+x)+y^2$ auftritt, die Ober- und Unterschranken jedoch fast optimal berechnet werden. Bei zu kleinen absoluten Werten von z.B. $x = -2^{-3\,000\,001}$ und $y = 2^{-1\,500\,000}$ ist die optimale Einschließung bei zu kleiner Präzision jedoch nicht mehr möglich, da dann bei der Abrundung von $2+x$ der zu kleine Wert 2 und damit die Unterschranke von t als 0 viel zu grob berechnet wird. Abhilfe schafft in diesem Fall nur eine ausreichend vergrößerte Präzision, mit der t dann wieder fast optimal eingeschlossen werden kann, vgl. Seite 298

In Ausnahmefällen kann also die optimale Einschließung eines Funktionswertes erst durch eine hinreichend große Präzision realisiert werden.

C.1.24.2. Imaginärteil

Die Imaginärteilfunktion $v(x,y) := \arg(1+z)$ wird ausführlich beschrieben ab Seite 251.

C.1.24.3. Numerische Ergebnisse

Bezeichnungen: $t = \ln(1+z),\ z = x + i\cdot y \in \mathbb{C},$ `t = lnp1(z,rnd);`

1. Mit `prec` = 30 000, `rnd = RoundDown` und $x = -2,\ y =$ `minfloat()` erhält man für t die komplexe Unterschranke[3] `t`:

   `t` = $(-0, 3.14159265358979323846264338327950288419716...53653494060340216654 4)$.

2. Mit `prec` = 30 000, `rnd = RoundUp` und $x = -2,\ y =$ `minfloat()` erhält man für t die komplexe Oberschranke `t`:

   `t` = $(2.3825649048879...475741163e-323228497, 3.14159265358979323...3402166545)$.

   Die groben Unter- und Oberschranken des Realteils von t ergeben sich aus dem sehr kleinen Realteil: $\Re(t) = \ln(1+y^2)/2 \ll$ `minfloat()`. Mit $x = 0$ erhält man für für die Ober- und Unterschranken von $\Re(t)$ die gleichen Ergebnisse.

3. Mit `prec` = 30 000, `rnd = RoundDown` und $x =$ `succ`$(-1) = -1 + 2^{-\texttt{prec}},\ y =$ `succ`$(+1) = 1 + 2^{-\texttt{prec}+1}$ erhält man für t die fast optimale komplexe Unterschranke

   `t` = $(2.5186050871681...74787632691e-9031, 1.5707963267948966...4703017010832719)$.

4. Mit `prec` = 30 000, `rnd = RoundUp` und $x =$ `succ`$(-1) = -1 + 2^{-\texttt{prec}},\ y =$ `succ`$(+1) = 1 + 2^{-\texttt{prec}+1}$ erhält man für t die fast optimale komplexe Oberschranke

   `t` = $(2.5186050871681...74787632692e-9031, 1.5707963267948966...4703017010832721)$.

5. Mit `prec` = 30 000, `rnd = RoundDown` und $x = -1,\ y = -$`minfloat()` $= -2^{-1073741824}$ erhält man für t die fast optimale komplexe Unterschranke

   `t` = $(-7.4426111795489301787...8304256133647e8, -1.5707963267948966...7010832724)$.

6. Mit `prec` = 30 000, `rnd = RoundUp` und $x = -1,\ y = -$`minfloat()` $= -2^{-1073741824}$ erhält man für t die fast optimale komplexe Oberschranke

   `t` = $(-7.4426111795489301787...8304256133634e8, -1.5707963267948966...7010832721)$.

7. Mit `prec` = 30 000, `rnd = RoundDown` und $x = y = 2\cdot$`minfloat()` $= 2^{-1073741823}$ erhält man für t die fast optimale komplexe Unterschranke

   `t` = $(4.765129809775902...32546e-323228497, 4.765129809775902...32486e-323228497)$.

8. Mit `prec` = 30 000, `rnd = RoundUp` und $x = y = 2\cdot$`minfloat()` $= 2^{-1073741823}$ erhält man für t die fast optimale komplexe Oberschranke

   `t` = $(4.765129809775902...32726e-323228497, 4.765129809775902...32606e-323228497)$.

[3] Die komplexe Unterschranke bedeutet die jeweilige Unterschranke für den Real- bzw. Imaginärteil.

C.1.25. $e^z - 1$

Wir betrachten die Aufgabe, den Funktionsterm $f(z) = e^z - 1,\ z \in \mathbb{C}$, für $|z| \to 0$ so auszuwerten, dass sich die auf- und abgerundeten Funktionswerte bei einer vorgegebenen Präzision `prec` nur wenig unterscheiden. Wegen der Taylorentwicklung $e^z - 1 = z + z^2/2! + \ldots$ gilt $f(z) \approx z$ für $z \to 0$, so dass mit $e^z \approx 1$ bei naiver Auswertung von $f(z)$ beim Realteil starke Auslöschung zu erwarten ist, die beim verlangten Auf- und Abrunden zu stark unterschiedlichen Funktionswerten führen wird. Mit $z = x + i \cdot y$ betrachten wir dazu das Beispiel $x = y = 2^{-400} = 3.8725\ldots \cdot 10^{-121}$, wobei die verlangten Rundungen durch die naive Intervallauswertung von $f(z)$ mit `prec` = 53 realisiert werden. Man erhält für den Realteil die erwartet grobe Einschließung:

$$f(z) \in ([-1.11...16e-16, 2.22...32e-16], [3.87259191484931e-121, 3.87259191484932e-121]).$$

Um für den Realteil eine Einschließung mit einer Genauigkeit von 53 Bits berechnen zu können, muss e^z mit einer Präzision von mindestens `prec` = 400 + 53 = 453 Bits ausgewertet werden, um nach der Subtraktion von 1 für die Differenz $e^z - 1$ noch 53 korrekte Bits zu erhalten. Bei einer Ausgabe von 16 Dezimalstellen erhält man jetzt für den Realteil die viel bessere Einschließung:

$$f(z) \in ([3.87259...931e-121, 3.87259...932e-121], [3.87259...931e-121, 3.87259...932e-121]).$$

Mögliche, bei der Auswertung arithmetischer Ausdrücke auftretende, Auslöschungseffekte lassen sich mit einer hinreichend großen Präzision vermeiden, wobei jedoch deutlich höhere Laufzeiten in Kauf zu nehmen sind.

Wählen wir daher im oberen Beispiel jetzt $x = y = 2^{-400\,000} = 1.0040016\ldots \cdot 10^{-120412}$, so müsste $f(z)$ mit einer Präzision von `prec` = 400 053 Bits ausgewertet werden, was etwa 120 428 Dezimalstellen entspricht und eine sehr hohe Laufzeit erfordert.

Um diese hohen Laufzeiten zu vermeiden, wird der Realteil $\Re(f(z)) = e^x \cdot \cos(y) - 1$ im Fall[4] $\cos(y) > 0$ wie folgt berechnet:

$$\Re(f(z)) = e^x \cdot \cos(y) - 1 = e^{x+\ln(\cos(y))} - 1, \quad \cos(y) > 0, \tag{C.36}$$

wobei die obere rechte Seite mit der bereits implementierten Funktion `expm1`$(x + \ln(\cos(y)))$ auszuwerten ist. $\ln(\cos(y))$ wird mit der Funktion `ln_cos`(y) berechnet, und bei der Auswertung des Exponenten $x + \ln(\cos(y))$ ist wieder mögliche Auslöschung zu beachten, die jedoch in fast allen Fällen mit einer **nur** doppelten Präzision vermieden werden kann!

Für Maschinenzahlen $z = x + i \cdot y \in \mathbb{C}$ erhält man auf- bzw. abgerundete Funktionswerte $e^z - 1$ mit Hilfe der Funktion

```
MpfcClass expm1(const MpfcClass& z, RoundingMode rnd);
```

wobei für `rnd` die Rundungen `RoundDown,RoundUp,RoundNearest` zur Verfügung stehen. Wird der Rundungsparameter `rnd` nicht gesetzt, so wird der voreingestellte Current-Rundungs-Modus benutzt.

Numerische Beispiele findet man ab Seite 272.

[4] Für $\cos(y) \le 0$ kann bei der Auswertung von $e^x \cdot \cos(y) - 1$ keine Auslöschung auftreten!

C.1.25.1. Realteil

Mit $z = x + i \cdot y \in \mathbb{C}$ lautet die auszuwertende Realteilfunktion

$$\Re(e^z - 1) = \begin{cases} \cos(y) - 1 = -2 \cdot \sin^2(y/2), & \text{falls } x = 0, \\ e^x \cdot \cos(y) - 1, & \text{falls } \cos(y) \le 0, \\ e^{x+\ln(\cos(y))} - 1, & \text{falls } \cos(y) > 0. \end{cases} \tag{C.37}$$

Im Fall $\cos(y) > 0$ wird der Exponent $x + \ln(\cos(y))$ intervallmäßig ausgewertet durch

$$x + \ln(\cos(y)) \in \texttt{u} := [x] \oplus \texttt{ln\_cos}([y]); \tag{C.38}$$

Zur Berechnung eines abgerundeten bzw. aufgerundeten Funktionswertes wird die reelle Intervallfunktion `expm1()` wegen der Monotonie der Exponentialfunktion wie folgt aufgerufen:

$$(e^{x+\ln(\cos(y))} - 1)_d = \texttt{expm1(Inf(u),RoundDown)};$$
$$(e^{x+\ln(\cos(y))} - 1)_u = \texttt{expm1(Sup(u),RoundUp)};$$

Um eine mögliche Auslöschung bei der Auswertung der rechten Seite von (C.38) zu vermeiden, wird in der Hilfsfunktion `MpfiClass expm1_u(const MpfcClass& z)` die Current-Präzision schrittweise verdoppelt, bis der relative Durchmesser von `u` den Wert 2^{-prec} unterschritten hat, wobei `prec` die ursprüngliche Current-Präzision ist. Weitere Einzelheiten findet man in der Datei `mpfcclass.cpp` bei der Funktion `expm1_u`(...).

C.1.25.2. Imaginärteil

Wegen $\Im(e^z - 1) = \Im(e^z) = e^x \cdot \sin(y)$ wird der Imaginärteil von $e^z - 1$ wie bei der Exponentialfunktion berechnet.

Es soll noch auf ein mögliches Problem aufmerksam gemacht werden. Für $x \to +\infty$ und $y \to 0$ kann bei e^x ein vorzeitiger Overflow auftreten, während das Produkt $e^x \cdot \sin(y)$ selbst noch im MPFR-Zahlenformat darstellbar ist. Man könnte das Problem im Fall $\sin(y) > 0$ mit Hilfe der Darstellung

$$e^x \cdot \sin(y) = e^{x+\ln(\sin(y))}, \quad \sin(y) > 0 \tag{C.39}$$

lösen, indem man $x + \ln(\sin(y))$ intervallmäßig in höherer Präzision auswertet, um mögliche Auslöschung zu vermeiden. Damit hätte man dann zwar einen Überlauf beim Imaginärteil vermieden, aber ein Überlauf bei der Auswertung des Realteils $e^x \cdot \cos(y) \approx e^x$ ist dann jedoch unvermeidbar, so dass die Auswertung nach (C.39) nicht zur Anwendung kommt.

C.1.25.3. Numerische Ergebnisse

Nach (C.38) betrachten wir zunächst den Fall $\cos(y) > 0$ und wählen mit $y = 1$ für die Maschinenzahl x die 54-stellige Näherung

$$x \approx -\ln(\cos(1)) \approx \texttt{str} := 0.615626470386014262147037516408891863350935423946372834;$$

Mit der Präzision `prec` = 180 wird dann `str` zur nächsten Maschinenzahl x gerundet. Für den abgerundeten und aufgerundeten Funktionswert $f(z) = e^z - 1$ erhalten wir

$$f(z)_d = (1.24671462282980382312158 8...205e-55, 1.55740772465490223050697 4...003);$$
$$f(z)_u = (1.24671462282980382312158 8...206e-55, 1.55740772465490223050697 4...005).$$

Man erkennt, dass die obigen 54-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 53 Dezimalstellen übereinstimmen.

Im nächsten Beispiel wählen wir mit $x = y = 2^{-400\,000}$ extrem kleine, positive Werte, so dass bei der *naiven* Auswertung des Realteils von $f(z) = e^z - 1$ starke Auslöschung auftreten muss. Mit der Präzision `prec` = 180 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = e^z - 1$ die Ergebnisse

$$f(z)_d = (1.00400160603542656243...179e - 120412, 1.00400160603542656243...179e - 120412);$$
$$f(z)_u = (1.00400160603542656243...180e - 120412, 1.00400160603542656243...180e - 120412).$$

Beachten Sie, dass wir jetzt positive und extrem kleine Real- und Imaginärteilwerte erhalten, die jedoch nahezu optimal gerundet werden konnten; vergleichen Sie dazu auch die Bemerkungen von Seite 271.

Im folgenden Beispiel wählen wir $x = 40$ und $y = 2$, so dass $\cos(y)$ negativ wird und damit $\Re(e^z - 1) = e^x \cdot \cos(y) - 1$ ohne Auslöschung direkt ausgewertet werden kann. Mit der Präzision `prec` = 180 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = e^z - 1$ die Ergebnisse

$$f(z)_d = (-9.79548341640308523575303 6...643e16, 2.14035217447578740435196 14...556e17);$$
$$f(z)_u = (-9.79548341640308523575303 6...640e16, 2.14035217447578740435196 14...557e17).$$

Im letzten Beispiel gilt $x = 0$ und $y = 2^{400\,000}$, so dass der Realteil von $e^z - 1$ ausgewertet wird mit: $\Re(e^z - 1) = -2 \cdot \sin^2(y/2)$. Mit der Präzision `prec` = 180 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = e^z - 1$ die Ergebnisse

$$f(z)_d = (-1.47572347273820436676170379 0...877, -8.79594893963069186853881 4...020e - 1);$$
$$f(z)_u = (-1.47572347273820436676170379 0...875, -8.79594893963069186853881 4...018e - 1).$$

Auch hier erkennt man, dass die berechneten Werte fast optimal gerundet sind. Beachten Sie, dass die Auswertung von $\sin(2^{399\,999})$ mit sehr hohen Laufzeiten (ca. 13 Minuten!) verbunden ist, da die notwendige Argumentreduktion mit Hilfe von π in sehr hoher Präzision erfolgen muss.

C.1.26. $\sqrt{z+1}-1$

Wir betrachten die Aufgabe, $f(z) = \sqrt{z+1}-1$, $z = x+i\cdot y \in \mathbb{C}$, insbesondere für $|z| \to 0$ so auszuwerten, dass sich die auf- und abgerundeten Funktionswerte bei einer vorgegebenen Präzision `prec` möglichst wenig unterscheiden. Wenn man $f(z)$ z.B. für $z = 2^{-30\,000} + i\cdot 2^{-30\,000}$ bei einer Präzision `prec` = 2000 naiv auswertet und dabei zur nächsten Rasterzahl rundet, so erhält man für den Maschinenwert $\widetilde{f}(z)$

$$\widetilde{f}(z) = (0, 6.296512717920457...7933\cdot 10^{-9032}).$$

Vergleicht man mit der Taylorreihe $f(z) = z/2 - z^2/8 \pm ...$, so erwartet man die Näherung

$$f(z) \approx (6.296512717920457...\cdot 10^{-9032}, 6.296512717920457...\cdot 10^{-9032}),$$

so dass bei der naiven Auswertung von $f(z)$ der Realteil 0 viel zu grob ausfällt. Wird bei der naiven Berechnung jedoch aufgerundet, so erhält man für den Realteil den viel zu großen Wert $1.7419619...\cdot 10^{-602}$. Um nahezu optimale Ergebnisse zu erhalten, muss man daher einen anderen Algorithmus benutzen. Ausgehend von der Aufspaltung in Real- und Imaginärteil von $\sqrt{z}$ in [17, 18] erhält man mit $z = x+i\cdot y$ zunächst für den Hauptwert von $\sqrt{z+1}-1$ die Aufspaltung

$$\sqrt{z+1}-1 = \begin{cases} \sqrt{x+1}-1+0\cdot i, & \text{falls } \ y=0 \wedge x \ge -1 \\ -1+i\cdot\sqrt{|x|-1}, & \text{falls } \ y=0 \wedge x < -1 \\ \sqrt{\dfrac{|z+1|+x+1}{2}} - 1 + i\cdot \operatorname{sign}(y)\cdot\sqrt{\dfrac{|z+1|-x-1}{2}}, & \text{falls } \ y \ne 0. \end{cases} \tag{C.40}$$

Der Fall $y = 0 \wedge x \ge -1$ ist trivial, da $\sqrt{x+1}-1$ mithilfe der bereits implementierten Funktion `sqrtp1m1()` direkt ausgewertet werden kann. Im obigen zweiten Fall $y = 0 \wedge x < -1$ kann $\sqrt{|x|-1}$ bez. der Rundungen ebenfalls einfach implementiert werden, da die Wurzelfunktion monoton wachsend ist und $|x|-1$ natürlich ohne Auslöschung berechnet werden kann, da x als exakte Maschinenzahl anzusehen ist.

Im letzten Fall $y \ne 0$ wird Auslöschung möglich, wenn im Real- und Imaginärteil $x+1$ negativ bzw. positiv gewählt werden. Die obige Darstellung in (C.40) ist jedoch vorteilhaft, wenn das Monotonieverhalten mithilfe der partiellen Ableitungen auf den Rändern der Rechteckintervalle für Real- und Imaginärteil zu untersuchen ist. Um im Fall $y \ne 0$ die auftretenden Wurzelausdrücke ohne Auslöschung auswerten zu können, erweitert man analog zu Seite 257 und erhält die Ergebnisse

$$\Re(\sqrt{z+1}-1) = \begin{cases} \dfrac{\sqrt{2\cdot(|z+1|+x+1)}}{2} - 1, & \text{falls } \ y \ne 0 \wedge x \ge -1 \\ \dfrac{|y|}{\sqrt{2\cdot(|z+1|+|x+1|)}} - 1, & \text{falls } \ y \ne 0 \wedge x < -1. \end{cases} \tag{C.41}$$

$$\Im(\sqrt{z+1}-1) = \begin{cases} \dfrac{y}{\sqrt{2\cdot(|z+1|+x+1)}}, & \text{falls } \ y \ne 0 \wedge x \ge -1 \\ \dfrac{\operatorname{sign}(y)}{2}\cdot\sqrt{2\cdot(|z+1|+|x+1|)}, & \text{falls } \ y \ne 0 \wedge x < -1, \end{cases} \tag{C.42}$$

wobei die auftretenden Wurzeln jetzt ohne Auslöschung berechnet werden können. Beachten Sie aber, dass beim Realteil wegen der Subtraktion der Eins immer noch Auslöschung entstehen kann, die gegebenenfalls durch schrittweise Verdoppelung der internen Präzision vermieden werden kann.

Man könnte auch versuchen, den Realteil von $\sqrt{z+1}-1$ mit Hilfe des äquivalenten Terms

$$\sqrt{z+1}-1 = \frac{z}{\sqrt{z+1}+1} \tag{C.43}$$

intervallmäßig auszuwerten. Dabei übersieht man jedoch, dass bei der auftretenden komplexen Intervalldivision mögliche Auslöschungen bei der Realteilberechnung ebenfalls auftreten, so dass die Berechnung des Realteils nach (C.41) aus Laufzeitgründen viel vorteilhafter ist, weil, wie in (C.43), der Imaginärteil hier nicht automatisch mitberechnet werden muss.

Im nächsten Schritt wird untersucht, wo in der komplexen z-Ebene bei der Berechnung des Realteils nach (C.41) mit Auslöschungseffekten zu rechnen ist. Dazu setzt man in den Fällen $y \neq 0$ mit $x \geq -1$ und $x < -1$ die entsprechenden Terme in (C.41) gleich Null und erhält nach einfachen Rechnungen für den geometrischen Ort, an dem Auslöschung bei der Berechnung des Realteils von $\sqrt{z+1}-1$ eintritt, das Ergebnis

$$y^2 = -4x, \quad x \leq 0. \tag{C.44}$$

Natürlich wird Auslöschung nicht nur auf auf der Menge $\alpha := \{(x,y) \,|\, y^2 = -4x \wedge x \leq 0\}$ selbst, sondern auch in einer ganzen Umgebung von α auftreten. In der folgender Abbildung ist diese Umgebung angegeben, in der bei der Berechnung des Realteils nach (C.41) Auslöschung durch schrittweise Verdoppelung der Präzision zu vermeiden ist. Der Pfeil gibt an, aus welcher Richtung die Funktionswerte auf den Verzweigungsschnitt analytisch fortgesetzt werden.

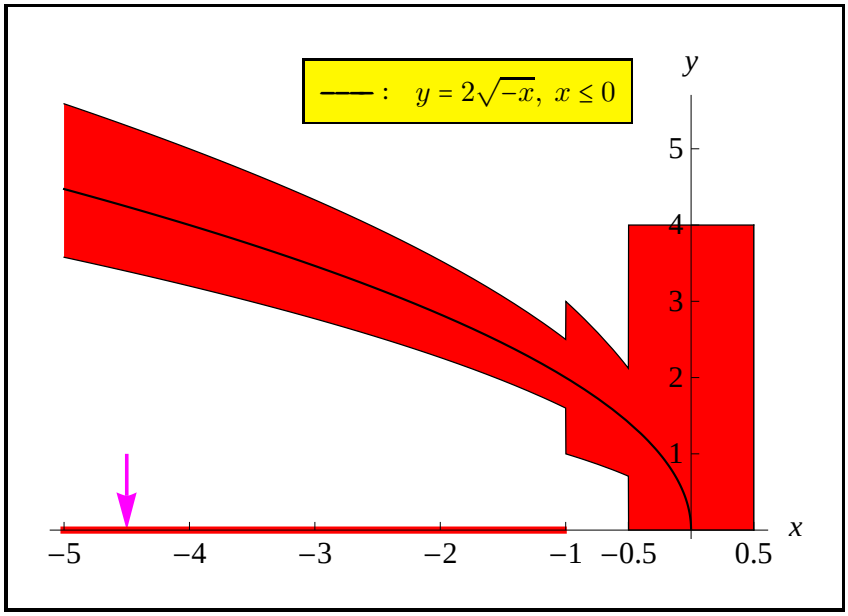


Abbildung C.14.: Verzweigungsschnitt und Auslöschungsbereich für den Realteil

Die obere und untere Umrandung des roten Bereichs ist mit $p(x) := 2\sqrt{-x}$ gegeben durch:

$$g(x) := \begin{cases} 1.25 \cdot p(x), & -\infty \leq x < -1, \\ 1.5 \cdot p(x), & -1 \leq x < -0.5, \\ 4, & -0.5 \leq x \leq +0.5; \end{cases} \qquad u(x) := \begin{cases} 0.8 \cdot p(x), & -\infty \leq x < -1, \\ 0.5 \cdot p(x), & -1 \leq x < -0.5, \\ 0, & -0.5 \leq x \leq +0.5; \end{cases}$$

Beide Funktionen sind definiert in der booleschen Funktion `bool Cancellation(x,y)` , die zu einem gegebenen Real- und Imaginärteil `x,y` entscheidet, ob bei der Realteilberechnung die Präzision entsprechend zu vergrößern ist, d.h. ob $z = x + i \cdot y$ im roten Bereich liegt. In diesem Fall erfolgt für $x < -1 \wedge y \neq 0$ die Berechnung des Realteils nach (C.41) mithilfe der Funktion

```
MpfrClass sqrtp1m1H2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd)
```

und für $-1 \leq x \leq +0.5 \wedge y \neq 0$ mit

```
MpfrClass sqrtp1m1H1(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd).
```

Es ist zu beachten, dass man sich nach (C.41) bei der Berechnung des Realteils auf die obere Halbebene, d.h. auf $y \geq 0$ beschränken kann.

Für $|x| < 0.5 \wedge y < 0.5$, d.h. im Rechteck der Breite 1 und der Höhe 0.5, das ganz im roten Bereich liegt, kann der Realteil trotz hinreichender Verdoppelung der Präzision beim Abrunden nicht optimal berechnet werden, wenn die Ausgangspräzision `pm` nicht groß genug gewählt wird. Wir betrachten dazu das Beispiel mit $x = 0,\ y = 2^{-1000}$ und wählen als Ausgangspräzision `pm` = 200. Nach (C.41) ist auszuwerten

$$s := |z+1| + x + 1 = \sqrt{1+y^2} + 1 = \sqrt{1+2^{-2000}} + 1.$$

Beim Abrunden erhält man daher auch bei doppelter Präzision $2 \cdot$`pm` = 400 das Ergebnis $s_d = 2$, so dass $\Re(\sqrt{z+1}-1)$ den viel zu stark abgerundeten Wert 0 erhält. Um einen positiven, abgerundeten Wert zu erhalten, müsste in diesem Beispiel die Ausgangspräzision mindestens den Wert `pm` = +1000 erhalten, denn bei der ersten Präzisionsverdoppelung wäre die Current-Präzision dann `prec` = 2000. Der folgende Algorithmus liefert mit `ex_x=expo(x)` und `ex_y=expo(y)` den geeigneten Wert `pm`.

```
if ( (ex_x <= -1) && (ex_y <= -1) )
{
   if ((ex_x-1)/2 >= ex_y-1)
      pm = -(ex_x-1)/2 + 1;
   else  pm = -(ex_y-1);
   if (pm < 2) pm = 2;
}
else pm = 2;
```

Ist die intern zu verwendende Präzision `prec` also kleiner als `pm`, so ist `prec = pm` zu setzen. Für das obige Beispiel erhält man dann mit `ex_y` = −999 genau den gewünschten Wert `pm` = +1000.

Bei der Auswertung des Wurzelausdrucks in (C.41) besteht noch ein weiteres Problem, denn mit der Abkürzung $M :=$ `MaxFloat()` gilt die Abschätzung

$$A := \sqrt{2 \cdot (|z+1| + x + 1)} < \sqrt{2 \cdot (\sqrt{2(M+1)^2} + M + 1)} = \sqrt{2(\sqrt{2}+1) \cdot (M+1)} < 3 \cdot \sqrt{M+1} < M,$$

so dass A im Zahlenformat darstellbar ist, während $2 \cdot (|z+1| + x + 1)$ durchaus einen vorzeitigen Überlauf verursachen kann. Um diesen Überlauf zu vermeiden, wird bei zu großen Werten von $|x|$ oder $|y|$ skaliert, d.h. $x, y, 1$ 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 für A kein Überlauf eintreten kann. Wir zeigen noch, dass die genannte Skalierung den vorzeitigen Überlauf tatsächlich verhindert:

$$\begin{aligned} 2 \cdot \left(\sqrt{2^{-8} \cdot (x+1)^2 + 2^{-8} \cdot y^2} + 2^{-4} \cdot (x+1)\right) &= 2^{-3} \cdot \left(\sqrt{(x+1)^2 + y^2} + x + 1\right) \\ &\leq 2^{-3} \cdot \left(\sqrt{(x+1)^2 + (y+1)^2} + x + 1\right) \\ &\leq 2^{-3} \cdot (1+\sqrt{2}) \cdot (M+1) \leq 0.3 \cdot (M+1) < M \ \blacksquare \end{aligned}$$

Die Berechnung von $\sqrt{2 \cdot (|z+1| + x + 1)}$ erfolgt mithilfe der Funktion

```
MpfrClass Sqrt_zpx1 (const MpfrClass& x, const MpfrClass& y, const RoundingMode rnd)
```

die in der Datei `mpfcclass.cpp` definiert ist.

Der vergleichsweise einfache komplexwertige Term $\sqrt{z+1}-1$ zeigt, wie relativ kompliziert die Auswertung werden kann, wenn in der komplexen Ebene bezüglich der verlangten drei Rundungen in allen Bereichen eine fast optimale Genauigkeit erreicht werden soll.

C.1.26.1. Numerische Ergebnisse

Im **1. Beispiel** wählen wir mit $z = x+i\cdot y \in \mathbb{C}$, $x = y =$ `MaxFloat()` $= 2.0985\ldots\cdot 10^{+323228496}$, `prec` $=$ 800 und erhalten für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1}-1$ das Ergebnis:

$$f(z)_d = (1.59160566708640011463...630e161614248, 6.592646532570641472 75...455e161614247);$$
$$f(z)_u = (1.59160566708640011463...631e161614248, 6.592646532570641472 75...457e161614247).$$

Man erkennt, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen. Beachten Sie bitte, dass bei den internen Berechnungen kein vorzeitiger Überlauf auftritt.

Im **2. Beispiel** wählen wir $x = 0.5,\ y = 4.0$ und liegen damit nach Abbildung C.14 im oberen rechten Eckpunkt des roten Bereiches. Mit `prec` $= 800$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1}-1$ das Ergebnis:

$$f(z)_d = (6.98823397628306387333427 79...5497e-1, 1.17728540988554801335506 77...6233088);$$
$$f(z)_u = (6.98823397628306387333427 79...5498e-1, 1.17728540988554801335506 77...6233089).$$

Man erkennt, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen.

Im **3. Beispiel** wählen wir $x = 0.5,\ y = 4.125$ und liegen damit nach Abbildung C.14 außerhalb des roten Bereiches. Mit `prec` $= 800$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1}-1$ das Ergebnis:

$$f(z)_d = (7.15992783568721686352569 89...0391e-1, 1.20192813148704099543179 52...1811408);$$
$$f(z)_u = (7.15992783568721686352569 89...0392e-1, 1.20192813148704099543179 52...1811409).$$

Man erkennt, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen. Beachten Sie bitte, dass wir jetzt außerhalb des roten Bereichs praktisch die gleiche Genauigkeit erhalten wie innerhalb dieses Bereiches, in dem Auslöschung eintreten kann.

Im **4. Beispiel** wählen wir $x = y = 2^{-5\,000\,000}$ und liegen damit in der rechten Halbebene sehr dicht am Ursprung innerhalb des roten Bereichs. Mit `prec` $= 800$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1}-1$ nach ca. zwei Minuten das Ergebnis:

$$f(z)_d = (5.25593636371215653698...757e-1505151, 5.25593636371215653698...757e-1505151);$$
$$f(z)_u = (5.25593636371215653698...758e-1505151, 5.25593636371215653698...758e-1505151).$$

Man erkennt wieder, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen.

Im **5. Beispiel** wählen wir $x = y = 0$ und liegen damit im Ursprung der komplexen Ebene. Mit `prec` $= 800$ erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1}-1$ das Ergebnis:

$$f(z)_d = (0,0);$$
$$f(z)_u = (0,0).$$

Der exakte Funktionswert $f(z) = 0$ wird also in beiden Rundungsmodi genau berechnet.

Im **6. Beispiel** wählen wir $x = -2^{-5\,000\,000}$, $y = 2 \cdot \sqrt{-x}$ und liegen damit nach Abbildung C.14 in der linken Halbebene genau auf der Parabel, auf der der Realteil von $f(z) = \sqrt{z+1} - 1$ verschwindet. Mit `prec` = 800 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z)$ das Ergebnis:

$$f(z)_d = (0, 1.025274242699206458221471676528028114765173819181718870 3...012e - 752575);$$
$$f(z)_u = (0, 1.025274242699206458221471676528028114765173819181718870 3...013e - 752575).$$

Man erkennt, dass der Realteil sogar exakt berechnet wird und dass der 241-stellige Imaginärteil fast optimal gerundet wurde, da die beiden gerundeten Werte auf den ersten 240 Dezimalstellen übereinstimmen.

Im **7. Beispiel** wählen wir $x = -2^{-5\,000\,000}$, $y = 2 \cdot \sqrt{-x}$ und mit $y = \texttt{succ}(y)$ liegen wir dann nur ganz knapp oberhalb der Parabel $y = 2 \cdot \sqrt{-x}$, so dass der Realteil von $f(z) = \sqrt{z+1} - 1$ nicht verschwinden wird. Mit `prec` = 800 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z)$ nach etwa zwei Minuten das Ergebnis:

$$f(z)_d = (3.15292440747892372470...656e - 1505391, 1.02527424269920645822...012e - 752575);$$
$$f(z)_u = (3.15292440747892372470...657e - 1505391, 1.02527424269920645822...013e - 752575).$$

Man erkennt wieder, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen. Wählt man x betragsmäßig deutlich kleiner, z.B. $x = -2^{-50\,000\,000}$, so wird bei gleicher Präzision die Laufzeit erheblich vergrößert.

Im **8. Beispiel** wählen wir $x = 0$, $y = 2^{-1000}$. Mit `prec` = 200 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1} - 1$ das Ergebnis:

$$f(z)_d = (1.08872622702715208444...708043e - 603, 4.66631809251609439495...310856e - 302);$$
$$f(z)_u = (1.08872622702715208444...708044e - 603, 4.66631809251609439495...310858e - 302).$$

Man erkennt wieder, dass die obigen 60-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 59 Dezimalstellen übereinstimmen. Beachten Sie zu diesem Beispiel auch die entsprechenden Anmerkungen auf Seite 276.

Im **9. Beispiel** wählen wir $x = -1.0625$, $y = 2.5625$ und liegen damit nach Abbildung C.14 innerhalb des roten Bereiches. Mit `prec` = 800 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1} - 1$ das Ergebnis:

$$f(z)_d = (1.18204382436689159650347 79...8317e - 1, 1.14581021155364878689757 26...8878406);$$
$$f(z)_u = (1.18204382436689159650347 79...8318e - 1, 1.14581021155364878689757 26...8878407).$$

Man erkennt, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen.

Im **10. Beispiel** wählen wir $x = -1.062500$, $y = 2.57812500$ und liegen damit nach Abbildung C.14 außerhalb des roten Bereiches. Mit `prec` = 800 erhalten wir für den abgerundeten und aufgerundeten Funktionswert $f(z) = \sqrt{z+1} - 1$ das Ergebnis:

$$f(z)_d = (1.21691237481236252053902 67...3628e - 1, 1.14921331015707747194386 68...4033545);$$
$$f(z)_u = (1.21691237481236252053902 67...3630e - 1, 1.14921331015707747194386 68...4033546).$$

Man erkennt, dass die obigen 241-stelligen Funktionswerte fast optimal gerundet wurden, da sie auf den ersten 240 Dezimalstellen übereinstimmen.

C.1.27. $\sqrt{1+z^2}$

Wir betrachten die Aufgabe, für eine vorgegebene Präzision `prec` ≥ 2 nahezu optimal gerundete Näherungen der Funktion $f(z) = \sqrt{1+z^2},\ z = x+i\cdot y \in \mathbb{C}$ zu berechnen. Die Verzweigungsschnitte von $f(z)$ liegen auf der imaginären Achse von $+i$ bis $+i\infty$ und von $-i$ bis $-i\infty$.

C.1.27.1. Realteil

Analog zu (C.16) auf Seite 256 erhält man für den Realteil von $f(z) = \sqrt{1+z^2}$ den Ausdruck

$$\Re\{f(z)\} = u(x,y) := \sqrt{\left(\sqrt{(x^2-y^2+1)^2+4x^2y^2}+(x^2-y^2+1)\right)/2}\,. \tag{C.45}$$

Aus (C.45) ergibt sich direkt, dass man sich auf $x, y \geq 0$ beschränken kann. Speziell gilt:

$$u(0,y) = \begin{cases} 0, & x = 0 \wedge |y| \geq 1, \\ \sqrt{1-y^2}, & x = 0 \wedge 0 \leq |y| < 1; \end{cases} \tag{C.46}$$

$$u(x,\pm 1) = \sqrt{|x|}\cdot\sqrt{\sqrt{1+\left(\frac{x}{2}\right)^2}+\frac{|x|}{2}}, \quad \text{falls } |y| = 1, \tag{C.47}$$

wobei $\sqrt{1-y^2}$ und $\sqrt{1+(x/2)^2}$ mit den bereits in `mpfrclass.cpp` vordefinierten Funktionen `sqrt1mx2(..)` und `sqrt1px2(..)` optimal berechnet werden können.

Bei der Auswertung von $u(x,y)$ spielt die Summe $s := x^2-y^2+1$ eine zentrale Rolle. Im Fall $x \geq y \geq 0$ kann x^2-y^2 mithilfe von `x2my2(...)`, definiert in `mpfrclass.cpp`, ohne Auslöschung optimal berechnet werden, sonst kann in der Umgebung von $y = \sqrt{1+x^2}$ bei der Auswertung von $s = x^2+(1+y)(1-y)$ Auslöschung auftreten, die jedoch bei einer Rechnung mit doppelter Präzision vermieden werden kann. Außerhalb dieser Umgebung wird $s = x^2+(1+y)(1-y)$ in einfacher Präzision ausgewertet.

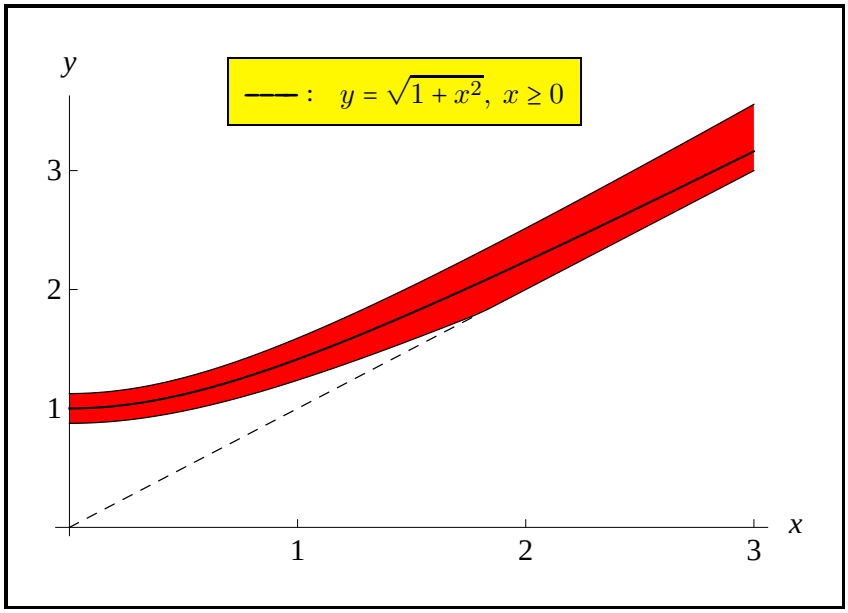


Abbildung C.15.: Auslöschungsbereich für $s = x^2+(1+y)(1-y)$

Der Auslöschungsbereich wird von oben begrenzt durch $g_1(x) := 1.125\cdot\sqrt{1+x^2}$ und von unten für $0 \leq x \leq 1.8$ durch $g_2(x) := 0.875\cdot\sqrt{1+x^2}$ bzw. für $x > 1.8$ durch $g_3(x) = x$. Die Berechnung der Funktionen $g_1(x), g_2(x)$ ist nicht kritisch und erfolgt für $x < 100$ mit Hilfe der schnellen C-XSC-Funktion `sqrt1px2(...)`. Für $x \geq 100$ wird $\sqrt{1+x^2} \approx x$ benutzt. Innerhalb des roten Bereiches ist der Graph von $y = \sqrt{1+x^2}$ gezeichnet, auf dem $s(x,y) = x^2-y^2+1$ verschwindet.

Der Algorithmus zur Berechnung von $s = x^2 - y^2 + 1$ ist gegeben durch:

```
if (x>=y)
  s = (x^2-y^2) + 1;
else
  if (y<=1.125*sqrt{1+x^2} && y>=0.875*sqrt{1+x^2})
    s = x^2 + (y-1)*(y+1);  (Doppelte Praezision)
  else
    s = x^2 + (y-1)*(y+1);  (Einfache Praezision)
```

Beachten Sie, dass es beim obigen Algorithmus für $x \to \infty$ oder für $y \to \infty$ bei der Auswertung von $s = x^2 + (y-1) * (y+1)$ zu einem vorzeitigen Überlauf kommen kann. Dies ist jedoch kein wirklicher Nachteil, da im Fall $|z| \to +\infty$ zur Berechnung von Real- und Imaginärteil die sehr einfache Näherung $\sqrt{1+z^2} \approx z + 1/(2z)$ benutzt wird. Bis auf die genannten Einschränkungen $x, y \to \infty$ erfolgt die nahezu optimale Berechnung von s mit Hilfe der Funktion

```
MpfrClass x2my2p1(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in `mpfcclass.cpp` definiert ist, aber nicht zur allgemeinen Anwendung zur Verfügung steht.

Wir kommen jetzt in (C.45) zur Berechnung des Ausdrucks

$$\alpha := \sqrt{(x^2 - y^2 + 1)^2 + 4x^2y^2} + (x^2 - y^2 + 1) = \sqrt{s^2 + (2xy)^2} + s, \tag{C.48}$$

der auch bei der Auswertung des Imaginärteils in (C.57) auf Seite 282 eine zentrale Rolle spielt. Die Wurzel $\sqrt{s^2 + (2xy)^2}$ wird mit der in `pmfrclass.cpp` definierten Funktion `sqrtx2y2(...)` berechnet, und Auslöschung kann in (C.48) nur für $s \le 0$, d.h. für $y \ge \sqrt{1+x^2} \ge 1$ auftreten, und diese Auslöschung wird vermieden, wenn α wie folgt umgeformt wird:

$$\sqrt{s^2 + (2xy)^2} + s = \frac{(\sqrt{s^2 + (2xy)^2} + s)(\sqrt{s^2 + (2xy)^2} - s)}{\sqrt{s^2 + (2xy)^2} - s} = \frac{(2xy)^2}{\sqrt{s^2 + (2xy)^2} - s}, \tag{C.49}$$

wobei jetzt der letzte Nenner als Summe positiver Summanden problemlos ausgewertet werden kann. Wir fassen zusammen:

$$\alpha = \begin{cases} \sqrt{s^2 + (2xy)^2} + s, & s > 0 \\ \dfrac{((2x) \cdot y)^2}{\sqrt{s^2 + ((2x) \cdot y)^2} - s}, & s \le 0, \quad \text{wobei } y \ge 1 \text{ automatisch erfüllt ist.} \end{cases} \tag{C.50}$$

Wir betrachten den Fall $\boldsymbol{s \le 0}$:
Wenn α aufzurunden ist, so ist im Zähler $(2x) \cdot y$ aufzurunden und unter der Wurzel ist $(2x) \cdot y$ abzurunden, so dass es sinnvoll ist, $(2x) \cdot y$ **intervallmäßig** auszuwerten; dies gilt auch dann, wenn α abzurunden ist. Wenn α aufzurunden ist, so muss neben der Wurzel im Nenner selbst wegen $s \le 0$ auch die nicht-negative Summe $-s \ge 0$ abgerundet werden, so dass $s \le 0$ selbst aufzurunden ist. Wenn andererseits α abzurunden ist, so muss neben der Wurzel im Nenner selbst wegen $s \le 0$ auch die nicht-negative Summe $-s \ge 0$ aufgerundet werden, so dass $s \le 0$ selbst abzurunden ist.
Wir betrachten den Fall $\boldsymbol{s > 0}$:
Um $\alpha = \sqrt{s^2 + (2xy)^2} + s$ z.B. aufzurunden, müssen neben den Grundoperationen auch s und $2xy$ selbst aufgerundet werden und für das Abrunden gelten ganz analoge Aussagen.
Die Auswertung von α nach (C.50) erfolgt mithilfe der Funktion

```
MpfrClass sqrt1px2_alpha(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd)
```

die in der Datei `mpfcclass.cpp` definiert und nur für den internen Gebrauch bestimmt ist.

Im folgenden Abschnitt muss noch geklärt werden, wie bei der Auswertung von α nach (C.50) ein vorzeitiger Überlauf verhindert werden kann. Gesucht ist dabei z.B. eine Schranke $c > 0$, so dass für $0 \le x < c$ und $0 \le y < c$ bei der Auswertung des Zählers $(2xy)^2$ kein Überlauf entsteht, d.h. wir verlangen $(2xy)^2 < M_0 := 1.573934 \cdot 10^{+323228496}$, wobei M_0 eine Unterschranke von `MaxFloat(2)` $\le$ `MaxFloat(prec)` ist. Um α nach (C.50) ohne vorzeitigen Überlauf berechnen zu können, müssen folgende Forderungen erfüllt sein:

1. $4x^2y^2 < M_0 \qquad (\Longrightarrow \quad 2xy < M_0)$,
2. $|s| = |x^2 - y^2 + 1| < M_0$,
3. $\sqrt{s^2 + (2xy)^2} + |s| < M_0$,
4. $\dfrac{4x^2y^2}{\sqrt{s^2 + (2xy)^2} + |s|} < M_0$.

Wegen $\sqrt{s^2 + (2xy)^2} + |s| \ge 2xy$ ist die letzte Forderung erfüllt, wenn gilt $2xy < M_0$, und diese Ungleichung ist erfüllt, wenn die Forderung 1 erfüllt ist. Danach sind nur noch die **drei ersten** Forderungen zu beachten.

Um die erste Forderung zu erfüllen, gilt mit $0 \le x < c_1$ und $0 \le y < c_1$ die Abschätzung $4x^2y^2 < 4 \cdot c_1^4$, und die erste Forderung ist erfüllt für $c_1 = \sqrt[4]{M_0/4} = 7.9201176703... \cdot 10^{+80807123}$.

Mit $0 \le x < c_2$ und $0 \le y < c_2$ ist wegen $|x^2 - y^2 + 1| \le x^2 + y^2 + 1 < 2c_2^2 + 1$ die zweite Forderung erfüllt, wenn gilt: $c_2 = \sqrt{(M_0 - 1)/2}$, und wegen $c := c_1 < c_2$ sind die beiden ersten Forderungen für $0 \le x, y < c = \sqrt[4]{M_0/4}$ erfüllt, so dass jetzt unter der Voraussetzung $0 \le x, y < c$ nur noch die 3. Bedingung zu untersuchen ist. Mit der recht groben Abschätzung

$$\begin{aligned}\sqrt{s^2 + (2xy)^2} + |s| &\le \sqrt{(2c^2+1)^2 + 4c^4} + (2c^2+1) < \sqrt{(3c^2)^2 + 4c^4} + 3c^2 \\ &= \sqrt{9c^4 + 4c^4} + 3c^2 < 7c^2\end{aligned}$$

ist dann die 3. Bedingung erfüllt, wenn gilt: $7c^2 < M_0 \iff 7 \cdot \sqrt{M_0/4} < M_0$, wobei die letzte Ungleichung wegen $M_0 \gg 1$ offensichtlich richtig ist ∎

Anmerkung:
Bei der obigen Berechnung von c wurde nicht berücksichtigt, dass der Maschinenwert von $4x^2y^2$ bei minimaler Präzision `prec` = 2 den exakten Wert etwa um den Faktor 2 übertreffen kann. Wenn man diesen Effekt bei der Berechnung von c aus Sicherheitsgründen mit dem Faktor 8 berücksichtigt, so erhält man für c den etwas kleineren Wert: $c = 4.7093... \cdot 10^{80807123}$. Im Programm benutzen wir für die positiven Werte $x, y \ge 0$ die noch etwas kleinere Oberschranke $c_0 := 2^{268435454} < c = \sqrt[4]{M_0/(4 \cdot 8)} = 4.70933014... \cdot 10^{+80807123}$.

Zusammenfassung:
Unter der Voraussetzung $0 \le x, y < c_0 = 0.5 \cdot 2^{268435455}$ wird α nach (C.50) auf der Maschine für alle `prec` ≥ 2 **ohne** einen vorzeitigen Überlauf berechnet.

Es muss jetzt noch geklärt werden, wie $\Re(f(z)) = u(x,y)$ im Fall $x > c_0 \vee y > c_0$ möglichst optimal berechnet werden kann. Mit $r := 1/z$ und

(C.51) $$R := \sum_{k=3}^{\infty} \frac{1 \cdot 3 \cdot 5 \cdots (2k-3)(-1)^{k+1}}{2 \cdot 4 \cdot 6 \cdots (2k-2) \cdot (2k)} \cdot r^{2k-1}, \quad |r| = \frac{1}{|z|} < 1 \quad \text{erhält man}$$

(C.52) $$f(z) = \sqrt{1+z^2} = \begin{cases} z + \dfrac{1}{2z} - \dfrac{1}{8z^3} + R, & x > 0, \\ -\left(z + \dfrac{1}{2z} - \dfrac{1}{8z^3} + R\right), & x < 0. \end{cases}$$

(C.53) $$\Re(f(z)) = 0, \quad \text{falls} \quad x = 0 \wedge |y| \ge 1.$$

Um $f(z)$ für $|z| \gg 1$ optimal einschließen zu können, benötigen wir zunächst eine Abschätzung für $|R|$. Mithilfe der geometrischen Reihe findet man wegen $|r| < c_0^{-1}$ unmittelbar:

$$|R| \le \frac{|r|^5}{16} \cdot \left(1 + |r| + |r|^2 + \ldots\right) = \frac{|r|^5}{16} \cdot \frac{1}{1-|r|} < \frac{|r|^5}{8} < \frac{c_0^{-5}}{8} < \Delta := \texttt{minfloat()}. \tag{C.54}$$

Um für $x > c_0 \vee y > c_0$, d.h. für $|z| > c_0$ möglichst optimal auf- bzw. abgerundete Werte von $f(z)$ berechnen zu können, wird zunächst der Ausdruck $t := 1/(2z) - 1/(8z^3) \in \mathbb{C}$ intervallmäßig möglichst optimal durch $T \ni t$ eingeschlossen und eine Einschließung für $f(z)$ erhält man dann durch

$$f(z) \in z \oplus (T \oplus ([-\Delta, +\Delta], [-\Delta, +\Delta])), \quad \Delta = \texttt{minfloat()}, \tag{C.55}$$

wobei $([-\Delta, +\Delta], [-\Delta, +\Delta])$ als komplexes Intervall mit jeweils gleichem Real- und Imaginärteilintervall $[-\Delta, +\Delta]$ aufzufassen ist. $t = 1/(2z) - 1/(8z^3)$ wird optimal eingeschlossen durch

$$t \in T = \{(2 \ominus (1 \oslash z)) \oslash (2 \oplus (1 \oslash z))\} \oslash z \oslash 8, \tag{C.56}$$

wobei $(1 \oslash z)$ aus Laufzeitgründen durch `reci`(z) realisiert wird. Die Berechnung der Realteilfunktion $u(x,y) = \Re\{\sqrt{1+z^2}\}$ erfolgt mit Hilfe der Funktion

```
MpfrClass Re_sqrt1px2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in der Datei `mpfcclass.cpp` definiert ist.

C.1.27.2. Imaginärteil

Analog zu (C.16) auf Seite 256 erhält man für den Imaginärteil von $f(z) = \sqrt{1+z^2}$

$$\Im\{f(z)\} = v(x,y) := \begin{cases} \dfrac{\operatorname{sign}(x) \cdot \operatorname{sign}(y) \cdot |x| \cdot |y|}{\sqrt{\left(\sqrt{(x^2-y^2+1)^2 + 4x^2y^2} + (x^2-y^2+1)\right)/2}}, & x \neq 0, \\ \sqrt{y^2-1}, & x = 0 \wedge |y| \ge 1, \\ 0, & x = 0 \wedge |y| < 1 \end{cases} \tag{C.57}$$

Speziell gilt für $y = \pm 1$

$$v(x, \pm 1) = \frac{\operatorname{sign}(x) \cdot \operatorname{sign}(y) \cdot \sqrt{|x|}}{\sqrt{\sqrt{1 + \left(\frac{x}{2}\right)^2} + \frac{|x|}{2}}}, \quad \text{falls } |y| = 1,$$

wobei die Wurzel im Nenner mithilfe der Funktion `sqrt1px2(...)` ausgewertet wird. Die Berechnung der Imaginärteilfunktion $v(x,y) = \Im\{\sqrt{1+z^2}\}$ erfolgt mit Hilfe der Funktion

```
MpfrClass Im_sqrt1px2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd);
```

die in der Datei `mpfcclass.cpp` definiert ist.

C.1.28. $\sqrt{1-z^2}$

Wir betrachten die Aufgabe, für eine vorgegebene Präzision `prec` ≥ 2 nahezu optimal gerundete Näherungen der Funktion $f(z) = \sqrt{1-z^2},\ z = x + i \cdot y \in \mathbb{C}$ zu berechnen. Die Verzweigungsschnitte von $f(z)$ liegen auf der reellen Achse von $+1$ bis $+\infty$ und von -1 bis $-\infty$. Auf dem rechten Verzweigungsschnitt werden die Funktionswerte von $f(z)$ von unten und auf dem linken Verzweigungsschnitt von obenher analytisch fortgesetzt. In der folgenden Abbildung wird dies durch die beiden Pfeile gekennzeichnet.

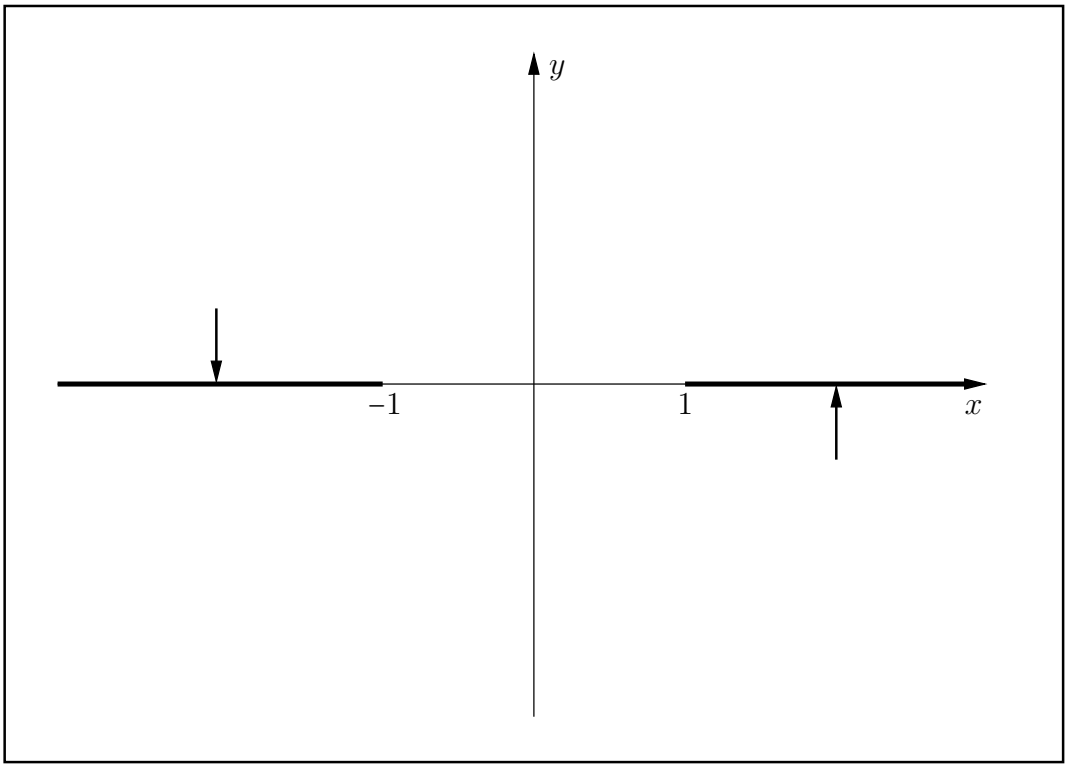


Abbildung C.16.: Verzweigungsschnitte der Funktion $f(z) = \sqrt{1-z^2}$

Die gerundeten Funktionswerte werden berechnet mithilfe der Funktion

```
MpfcClass sqrt1mx2(const MpfcClass& z, RoundingMode rnd);
```

die in der Datei `mpfcclass.cpp` definiert ist. Mit dem Rundungsparameter `rnd` können dabei die Werte `RoundDown, RoundUp, RoundNearest` gewählt werden.

Mit $z = x + i \cdot y \in \mathbb{C}$ und der Transformation

$$z = i \cdot z_T \quad \text{bzw.} \quad z_T = y + i \cdot (-x) \quad \text{erhält man} \qquad \sqrt{1-z^2} = \sqrt{1+z_T^2},$$

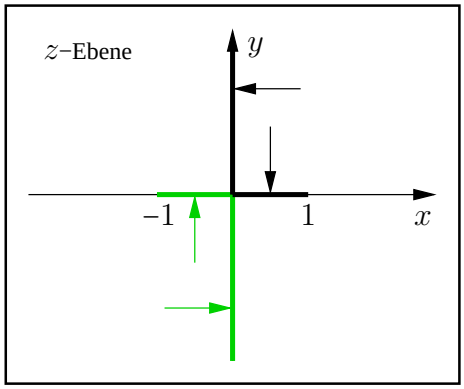
so dass die Auswertung von $f(z) = \sqrt{1-z^2}$ zurückgeführt werden kann auf die Auswertung von $\sqrt{1+z_T^2}$, wobei diese Funktion bereits durch `sqrt1px2(..)` in der Datei `mpfcclass.cpp` definiert wurde.

C.1.29. $\sqrt{z^2-1}$

Wir betrachten die Aufgabe, für eine vorgegebene Präzision `prec` ≥ 2 nahezu optimal gerundete Näherungen der Funktion $f(z) = \sqrt{z^2-1},\ z = x+i\cdot y \in \mathbb{C}$ zu berechnen. Die Verzweigungspunkte ergeben sich aus der Bedingung $z^2-1=0$ zu $z_{1,2} = \pm 1$, und die Verzweigungsschnitte ergeben sich aus der Bedingung $z^2-1 = -\alpha$, mit $\alpha \geq 0$, bzw. aus dem Gleichungssystem

$$\begin{aligned} x^2-y^2-1 &= -\alpha, \quad \alpha \geq 0, \\ x\cdot y &= 0, \qquad \text{mit den zwei } (\pm) \text{ Lösungen} \end{aligned}$$

$$\begin{aligned} x=0,\ y &= \pm\sqrt{\alpha-1}, \quad \text{falls } \alpha \geq 1, \\ y=0,\ x &= \pm\sqrt{1-\alpha}, \quad \text{falls } 0 \leq \alpha \leq 1. \end{aligned}$$



Im linken Bild sind die zwei Verzweigungsschnitte durch die Farben schwarz (+) und grün (−) gekennzeichnet. Die gleichfarbigen Pfeile geben die Richtungen an, aus denen die Funktionswerte von $f(z) = \sqrt{z^2-1}$ auf den jeweiligen Verzweigungsschnitt analytisch fortgesetzt werden.
Zu beachten ist, dass die Funktionswerte der Realteilfunktion $\hat{u}(x,y)$ auf beiden Verzweigungsschnitten nach (C.59) durch den Wert 0 stetig ergänzt werden können.

Bildet man jedoch auf beiden Verzweigungsschnitten die Grenzwerte der Imaginärteilfunktion $\hat{v}(x,y)$, und zwar jeweils von beiden Seiten her, so erhält man mit Ausnahme des Ursprungs stets **verschiedene** Grenzwerte.

Zur Berechnung der Real- und Imaginärteilfunktionen von $f(z) = \hat{u}(x,y)+i\cdot\hat{v}(x,y)$ betrachten wir die formale Umformung

$$\sqrt{z^2-1} = \sqrt{(-1)\cdot(1-z^2)} = i\cdot\sqrt{1-z^2}.$$

Mit der Transformation $z = i\cdot z_T$ und $z^2 = -z_t^2$ bzw. $z_T = y - i\cdot x$ erhält man

$$\sqrt{z^2-1} = i\cdot\sqrt{1+z_T^2} = i\cdot(u(y,-x)+i\cdot v(y,-x)) = -v(y,-x)+i\cdot u(y,-x), \quad \text{d.h.}$$

$$\hat{u}(x,y) = -v(y,-x), \qquad \hat{v}(x,y) = u(y,-x). \tag{C.58}$$

Mit den Ergebnissen von Seite 279 bzw. Seite 282 für $u(x,y)$ bzw. $v(x,y)$ erhält man schließlich mit

$$h(x,y) := \sqrt{\left(\sqrt{(y^2-x^2+1)^2+4x^2y^2}+(y^2-x^2+1)\right)/2} \geq 0$$

für $f(z) = \hat{u}(x,y)+i\cdot\hat{v}(x,y)$ die Beziehungen

$$\hat{u}(x,y) = \begin{cases} \dfrac{|x|\cdot|y|}{h(x,y)}, & y \neq 0, \\ \sqrt{x^2-1}, & y=0 \wedge |x| \geq 1, \\ 0, & y=0 \wedge |x| < 1. \end{cases} \tag{C.59}$$

$$\hat{v}(x,y) = \begin{cases} \operatorname{sign}(x)\cdot\operatorname{sign}(y)\cdot h(x,y), & x\cdot y \neq 0, \\ 0, & y=0 \wedge |x| \geq 1, \\ \sqrt{1-x^2}, & y=0 \wedge |x| < 1, \\ \sqrt{1+y^2}, & x=0. \end{cases} \tag{C.60}$$

In C.59, C.60 sind die Vorzeichen so gewählt, dass $f(0) = \sqrt{-1} = +i$ und $f(x + i \cdot 0) = \sqrt{x^2 - 1}$ für $|x| \geq 1$ gewährleistet sind. Nach C.59, C.60 können damit die Real- und Imaginärteilfunktionen $\hat{u}(x, y)$ und $\hat{v}(x, y)$ mithilfe der entsprechenden Funktionen $v(x, y)$ und $u(x, y)$ von $\sqrt{1 + z^2} = u(x, y) + i \cdot v(x, y)$ berechnet werden, die bereits in der Datei `mpfcclass.cpp` definiert worden sind.
Die Funktionswerte $f(z) = \sqrt{z^2 - 1}$ werden berechnet mithilfe der Funktion

```
MpfcClass sqrtx2m1(const MpfcClass& z, RoundingMode rnd)
```

wobei für `rnd` nur die Rundungsmodi `RoundDown, RoundUp, RoundNearest` zugelassen sind. Wird `rnd` nicht gesetzt, so kommt der aktuelle Rundungsmodus zur Anwendung.

C.1.29.1. Realteil

Die Realteilfunktion $\hat{u}(x, y)$ wird nach (C.59) berechnet mithilfe der Funktion

```
MpfrClass Re_sqrtx2m1(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd)
```

wobei für `rnd` nur die Rundungsmodi `RoundDown, RoundUp, RoundNearest` zugelassen sind. Zu beachten ist, dass nach (C.58) auf die bereits implementierte Imaginärteilfunktion $v(x, y)$ von $\sqrt{1 + z^2}$ zurückgegriffen werden kann.

C.1.29.2. Imaginärteil

Die Imaginärteilfunktion $\hat{v}(x, y)$ wird nach (C.60) berechnet mithilfe der Funktion

```
MpfrClass Im_sqrtx2m1(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd)
```

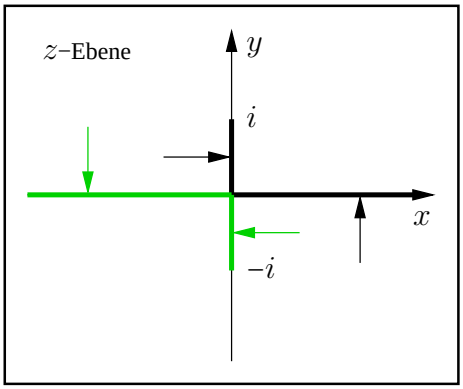
wobei für `rnd` nur die Rundungsmodi `RoundDown, RoundUp, RoundNearest` zugelassen sind. Zu beachten ist dabei, dass nach (C.58) auf die bereits implementierte Realteilfunktion $u(x, y)$ von $\sqrt{1 + z^2}$ zurückgegriffen werden kann.

C.1.30. $\sqrt{-z^2-1}$

Wir betrachten die Aufgabe, für eine vorgegebene Präzision `prec` ≥ 2 nahezu optimal gerundete Näherungen der Funktion $f(z) = \sqrt{-z^2-1}$, $z = x+i\cdot y \in \mathbb{C}$ zu berechnen. Die Verzweigungspunkte ergeben sich aus der Bedingung $z^2+1 = 0$ zu $z_{1,2} = \pm i$, und die Verzweigungsschnitte ergeben sich aus der Bedingung $-z^2-1 = -\alpha$, mit $\alpha \geq 0$, bzw. aus dem Gleichungssystem

$$\begin{aligned} x^2 - y^2 + 1 &= \alpha, \quad \alpha \geq 0, \\ x \cdot y &= 0, \qquad \text{mit den zwei } (\pm) \text{ Lösungen} \end{aligned}$$

$$\begin{aligned} x = 0,\ y &= \pm\sqrt{1-\alpha}, \quad \text{falls } 0 \leq \alpha \leq 1, \\ y = 0,\ x &= \pm\sqrt{\alpha-1}, \quad \text{falls } \alpha \geq 1. \end{aligned}$$



Im linken Bild sind die zwei Verzweigungsschnitte durch die Farben schwarz (+) und grün (−) gekennzeichnet. Die gleichfarbigen Pfeile geben die Richtungen an, aus denen die Funktionswerte von $f(z) = \sqrt{-z^2-1}$ auf den jeweiligen Verzweigungsschnitt analytisch fortgesetzt werden.
Zur Berechnung von $f(z) = \sqrt{-z^2-1}$ betrachten wir die Transformation $z = x + i\cdot y = i\cdot z_T$ und erhalten mit $z_T = y - i\cdot x$ die einfache Beziehung

$$f(z) = \sqrt{-z^2-1} = \sqrt{z_t^2-1},$$

so dass $f(z)$ mit Hilfe von z_T und der bereits implementierten Funktion `sqrtx2m1`(z_T) direkt berechnet werden kann, vgl. Seite 284.

Die Auswertung von $f(z) = \sqrt{-z^2-1}$ erfolgt mithilfe der Funktion

```
MpfcClass sqrtmx2m1(const MpfcClass& z, RoundingMode rnd)
```

die in `mpfcclass.cpp` definiert ist. Für `rnd` stehen die drei Rundungsmodi

```
RoundDown, RoundUp, RoundNearest
```

zur Verfügung.

C.2. Elementarfunktionen für komplexe Intervallargumente

C.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`.

C.2.2. $1/z$

Ist $z = x + i \cdot y \in \mathbb{C}$ ein Element aus einem komplexen Intervall $Z = X + iY$ mit reellen Intervallen X, Y, so berechnet die Funktion

```
MpfciClass reci(const MpfciClass& Z);
```

aus `mpfciclass.cpp` mit dem Aufruf `res = reci(Z);` eine optimale Einschließung der komplexen Menge

$$\{1/z \,\big|\, x \in X \wedge y \in Y\} \subseteq \texttt{res}, \quad 0 \notin Z.$$

Anmerkungen:

1. `1 ⊘ Z` und `reci(Z)` liefern bei gleicher Präzision `prec` ≥ 2 **identische** komplexe Ergebnisintervalle.

2. Im Vergleich zu `1 ⊘ Z` liefert `reci(Z)` eine um den Faktor 13 geringere Laufzeit, so dass `reci(Z)` bevorzugt eingesetzt werden sollte!

3. Ist `W` ein weiteres komplexes Intervall, so gilt i.a. `W ⊘ Z` $\subset$ `W ⊙ reci(Z)`.
   Sind `W` und `Z` voneinander unabhängige komplexe Intervalle, so liefert die komplexe Intervalldivision `W ⊘ Z` eine optimale Einschließung des Wertebereichs $W := \{w \cdot z \,|\, w \in \texttt{W} \wedge z \in \texttt{Z}\}$, und dies gilt in der komplexen Intervallarithmetik für alle vier Grundoperationen.

C.2.3. $1/z^2$

Ist $z = x + i \cdot y \in \mathbb{C}$ ein Element aus einem komplexen Intervall $Z = X + iY$ mit reellen Intervallen X, Y, so berechnet die Funktion

```
MpfciClass reci_z2(const MpfciClass& Z);
```

aus `mpfciclass.cpp` mit dem Aufruf `res = reci_z2(Z);` eine optimale Einschließung der komplexen Menge

$$W_Z := \{1/z^2 \,|\, x \in X \wedge y \in Y\} \subseteq \texttt{res}, \quad 0 \notin Z.$$

$$1/z^2 = u(x,y) + i \cdot v(x,y); \qquad u(x,y) := \frac{x^2 - y^2}{(x^2+y^2)^2}, \quad v(x,y) := \frac{-2xy}{(x^2+y^2)^2}.$$

Die optimale Einschließung der Menge $\{u(x,y) \in \mathbb{R} \,|\, x \in X \wedge y \in Y\}$ mit Hilfe der Funktion

```
MpfiClass Re_rz2(const MpfiClass& X, const MpfiClass& Y)
```

wird ausführlich beschrieben ab Seite 210, und für den negativen Imaginärteil wird die optimale Einschließung der Menge $\{-v(x,y) \in \mathbb{R} \,|\, x \in X \wedge y \in Y\}$ mit Hilfe der Funktion

```
MpfiClass mIm_rz2(const MpfiClass& X, const MpfiClass& Y)
```

ausführlich beschrieben ab Seite 216. Beachten Sie dabei, dass bei der Auswertung der Funktionen $u(x,y), v(x,y)$ **kein** vorzeitiger Überlauf eintreten kann.

C.2.3.1. Numerische Ergebnisse

Im **1. Beispiel** wählen wir $X = [2,5]$ und $Y = [1,3]$ und berechnen mit dem Funktionsaufruf `res = reci_z2(Z)` und `prec` = 53 eine optimale Einschließung der Menge W_Z durch

$$([-2.95857988165681e-2, 1.20000000000001e-1], [-1.62379763209583e-1, -1.47928994082840e-2])$$

Im **2. Beispiel** und **3. Beispiel** wählen wir für Z die gleichen reellen Intervalle $X = [2,5]$ und $Y = [1,3]$ und berechnen $1/Z^2$ im **2. Beispiel** durch $\texttt{sqr}(1 \diamond Z)$ und im **3. Beispiel** durch $1 \diamond (\texttt{sqr}(Z))$. Im folgenden werden die Einschließungen verglichen mit dem Ergebnis aus dem **1. Beispiel**:

$$([-2.95857988165681e-2, 1.20000000000001e-1], [-1.62379763209583e-1, -1.47928994082840e-2])$$
$$([-4.08737024221454e-2, 1.58520710059172e-1], [-2.00000000000001e-1, -1.13122171945701e-2])$$
$$([-1.25000000000000e-1, 1.25000000000000e-1], [-2.50000000000000e-1, -6.75675675675675e-3]).$$

Man erkennt, dass die Einschließungen des **2.** und **3. Beispiels** im Vergleich zum **1. Beispiel** deutlich schlechter ausfallen.

Zusammenfassung

Wertet man beliebige, reelle, äquivalente, arithmetische Intervallausdrücke, wie z.B. $1 \diamond (\texttt{sqr}(X))$ oder $\texttt{sqr}(1 \diamond X)$, welche die Variable X nur einmal enthalten, naiv aus, so erhält man stets die gleiche, optimale Einschließung der Intervallausdrücke. Im Gegensatz dazu liefern entsprechende äquivalente, komplex Intervallausdrücke mit nur einer Variablen Z i.a jeweils verschiedene und meist nicht-optimale Einschließungen. Man sollte daher auf eine möglichst große Anzahl von komplexen Funktionen zurückgreifen können, die optimale Einschließungen liefern, vgl. dazu die Tabelle auf Seite 115 und Abschnitt 6.12.4 auf Seite 119.

C.2.4. $1/(1+z^2)$

Ist $z = x + i \cdot y \in \mathbb{C}$ ein Element aus einem komplexen Intervall $Z = X + iY$ mit reellen Intervallen X, Y, so berechnet die Funktion

```
MpfciClass reci_1pz2(const MpfciClass& Z);
```

aus `mpfciclass.cpp` mit dem Aufruf `res = reci_1pz2(Z);` eine optimale Einschließung der komplexen Menge

$$W_Z := \{1/(1+z^2) \,|\, x \in X \wedge y \in Y\} \subseteq \texttt{res}, \quad (0 \pm i) \notin Z.$$

$$1/(1+z^2) = u + i \cdot v; \quad u(x,y) := \frac{1+x^2-y^2}{4x^2y^2 + (1+x^2-y^2)^2}, \quad v(x,y) := \frac{-2xy}{4x^2y^2 + (1+x^2-y^2)^2}.$$

Die optimale Einschließung der Menge $\{u(x,y) \in \mathbb{R} \,|\, x \in X \wedge y \in Y\}$ mit Hilfe der Funktion

```
MpfiClass Re_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

wird ausführlich beschrieben ab Seite 227. Für den Imaginärteil wird die optimale Einschließung der Menge $\{v(x,y) \in \mathbb{R} \,|\, x \in X \wedge y \in Y\}$ mit Hilfe der mit (-1) zu multiplizierenden Funktion

```
MpfiClass Im_r1pz2(const MpfiClass& X, const MpfiClass& Y)
```

ausführlich beschrieben ab Seite 233. Beachten Sie dabei, dass bei der Auswertung der beiden Funktionen $u(x,y), v(x,y)$ **kein** vorzeitiger Überlauf eintreten kann.

C.2.4.1. Numerische Ergebnisse

Im **1. Beispiel** wählen wir $X = [1/8, 1/4]$ und $Y = [1/8, 2]$ und berechnen mit dem Funktionsaufruf `res = reci_1pz2(Z)` und `prec` = 53 eine optimale Einschließung der Menge W_Z durch

$$([-1.76556443707464, 2.26556443707464], [-3.98443603238417, -3.12195121951219e-2])$$

Im **2. Beispiel** wählen wir für Z die gleichen reellen Intervalle $X = [1/8, 1/4]$ und $Y = [1/8, 2]$ und berechnen eine Einschließung von W_Z durch die naive Intervallauswertung $1 \oslash (1 \oplus \texttt{sqr}(Z))$. Im folgenden wird diese Einschließungen verglichen mit dem Ergebnis aus dem **1. Beispiel**:

$([-1.76556443707464, 2.26556443707464], [-3.98443603238417, -3.12195121951219e-2])$
$([-1.60000000000000e1, 1.60000000000000e1], [-3.20000000000000e1, -3.50829107852542e-3])$

Man erkennt, dass die Einschließung von W_Z im **2. Beispiels** im Vergleich zum **1. Beispiel** deutlich schlechter ausfällt, d.h. dass die naive Intervallauswertung Überschätzungen bis etwa zum Faktor 10 verursachen kann.

Im **3. Beispiel** wählen wir für Z die gleichen reellen Intervalle $X = [1/8, 1/4]$ und $Y = [1/8, 2]$ und berechnen eine Einschließung von W_Z durch `reci_z2(sqrt1px2(Z))`. Im folgenden wird diese Einschließungen verglichen mit dem Ergebnis aus dem **1. Beispiel**:

$([-1.76556443707464, 2.26556443707464], [-3.98443603238417, -3.12195121951219e-2])$,
$([-6.01034495038369, 4.64224263704784e1], [-3.12306684752388e1, -2.91187528315533e-2])$.

Man erkennt, dass auch diese Einschließung keine optimale Einschließung des Wertebereichs W_Z darstellt. Es genügt daher nicht, zuerst mit `T = sqrt1px2(Z)` eine optimale Einschließung von $\{\sqrt{1+z^2} \,|\, x \in X \wedge y \in Y\} \subseteq$ `T` und anschließend mit `V = reci_z2(T)` eine Einschließung von $W_Z \subseteq$ `V` zu berechnen, um mit `V` eine optimale Einschließung von W_Z zu erhalten. Eine solche optimale Einschließung berechnet man nur mit dem Aufruf `res = reci_1pz2(Z)` aus dem 1. Beispiel.

C.2.5. $1/(1-z^2)$

Ist $z = x + i\cdot y \in \mathbb{C}$ ein Element aus einem komplexen Intervall $Z = X + iY$ mit reellen Intervallen X, Y, so berechnet die Funktion

```
MpfciClass reci_1mz2(const MpfciClass& Z);
```

aus `mpfciclass.cpp` mit dem Aufruf `res = reci_1mz2(Z);` eine optimale Einschließung der komplexen Menge

$$W_Z := \{1/(1-z^2) \,\big|\, x \in X \wedge y \in Y\} \subseteq \texttt{res}, \quad \pm 1 \notin Z.$$

Mit $f(z) := 1/(1-z^2)$, $z \in Z$, und $z = i\cdot\hat{z}$ erhält man $f(z) = 1/(1+\hat{z}^2)$, mit $\hat{z} = y - i\cdot x$ und

$$W_Z = \{1/(1+\hat{z}^2) \,\big|\, \Re(\hat{z}) \in Y \wedge \Im(\hat{z}) \in -X\}, \quad \hat{z} \neq \pm i.$$

Damit kann man die nahezu optimale Einschließung von W_Z zurückführen auf die Berechnung von `reci_1pz2`$(\hat{Z})$, mit $\hat{Z} := Y - i\cdot X$, und genau dies wird bei der Implementierung der obigen Funktion `reci_1mz2(...)` realisiert. Beachten Sie, dass $1/(1-z^2)$ die 1. Ableitung der `acoth`-Funktion ist.

C.2.5.1. Numerische Beispiele

Im **1. Beispiel** wählen wir $h(z) := 1/(2+z-3z^2)$ und zeigen, dass wir mit $X = [1/2, 7/4]$, $Y = [2,4]$ und der Funktion `reci_1mz2(...)` ebenfalls eine fast optimale Einschließung der Wertemenge

$$W_{Z,h} = \{1/(2+z-3z^2) \,\big|\, x \in X \wedge y \in Y\}$$

berechnen können. Mit Hilfe der Umformung

$$\frac{1}{2+z-3z^2} = 0.48 \cdot \frac{1}{1-w^2}, \quad \text{und} \quad w := \frac{6z-1}{5}$$

wird zunächst in der komplexen Ebene das Rechteck Z mit nur minimalen Überschätzungen in ein achsenparalleles Rechteck $W := (6 \diamond Z \diamond 1) \diamond 5$ abgebildet, so dass dann mit mit der Anweisung `T` $= 0.48 \diamond$ `reci_1mz2(W)` eine nahezu optimale Einschließung $W_{Z,h} \subset$ `T` berechnet werden kann. Mit der Präzision `prec` $= 53$ erhalten wir

`T` $= ([1.30737618861925e-2, 6.70527278268821e-2], [3.15076923076922e-3, 4.81179721970281e-2])$.

Im **2. Beispiel** wählen wir die gleiche Funktion $h(z)$ und die gleichen Intervalle X, Y wie im 1. Beispiel und zeigen, dass die naive Intervallauswertung von $h(z)$ im Vergleich zum 1. Beispiel zu starken Überschätzungen von $W_{Z,h}$ führt:

$([1.30737618861925e-2, 6.70527278268821e-2], [3.15076923076922e-3, 4.81179721970281e-2])$.
$([3.26276015114256e-3, 1.64868468905322e-1], [7.67754318618042e-4, 9.41176470588236e-2])$.

Im **3. Beispiel** wählen wir $h(z)$ und X, Y wie im 1. Beispiel und zeigen mit der Umformung

$$h(z) = \frac{1}{2+z-3z^2} = -\frac{1}{3}\cdot\frac{1}{z^2 - z/3 - 2/3}, \quad \text{und} \quad \texttt{A} = -1 \diamond 3,\ \texttt{B} = -2 \diamond 3,$$

dass auch die optimale Einschließung von $z^2 - z/3 - 2/3$ mit Hilfe der Funktion `poly2(z,A,B)` und anschließender Division ebenfalls zu starken Überschätzungen von $W_{Z,h}$ führt:

$([1.30737618861925e-2, 6.70527278268821e-2], [3.15076923076922e-3, 4.81179721970281e-2])$,
$([4.41305107318572e-3, 1.11103762978640e-1], [1.60574052236746e-3, 7.61904761904763e-2])$.

Die im 3. Beispiel berechnete Einschließung ist jedoch erwartungsgemäß besser als die aus dem 2. Beispiel, da `poly2(z,A,B)` schon eine optimale Einschließung des Nenners berechnet.

C.2.6. $z^2 + a \cdot z + b$

Mit den gegebenen komplexen Intervallen $Z = X + i \cdot Y,\ A = A_1 + i \cdot A_2,\ B = B_1 + i \cdot B_2$ ist für ein komplexes Intervallpolynom 2. Grades

$$W = Z^2 + A \cdot Z + B := \{w = z^2 + a \cdot z + b \in \mathbb{C} \,|\, z \in Z,\ a \in A,\ b \in B\} \tag{C.61}$$

eine optimale Einschließung des komplexen Intervalls W zu berechnen. Mit den reellen Intervallen X, Y, A_1, A_2, B_1, B_2 erhält man eine Einschließung von W durch

$$\begin{aligned} W \subset P := &\left(X + \frac{A_1}{2}\right)^2 - \left(\frac{A_1}{2}\right)^2 + \left(Y + \frac{A_2}{2}\right)^2 - \left(\frac{A_2}{2}\right)^2 + B_1 \\ &+ i \cdot \left\{2 \cdot \left(X + \frac{A_1}{2}\right)\left(Y + \frac{A_2}{2}\right) - 2 \cdot \frac{A_1}{2} \cdot \frac{A_2}{2} + B_2\right\}. \end{aligned} \tag{C.62}$$

Die obige Einschließung ist sogar **optimal**, d.h. es gilt $W = P$, wenn der komplexe Intervallausdruck P zur Vermeidung von Auslöschung in z.B. doppelter Präzision ausgewertet wird und wenn nur A_1, A_2 als Punktintervalle gewählt werden, d.h. die Intervalle X, Y, B_1, B_2 können beliebige echte Intervalle sein, da sie im Real- und Imaginärteil jeweils nur einmal vorkommen, vgl. Seite 74. Die Einschließung $W \subseteq P$ bleibt auch dann noch nahezu optimal, wenn die Intervalle A_1, A_2 z.B. eine nicht darstellbare komplexe Zahl $a = \sqrt{2} - i \cdot \sqrt{5} \in A_1 + i \cdot A_2$ in der Current-Precision selbst optimal einschließen und damit einen nur minimalen Durchmesser besitzen.

Bei der Auswertung der Intervallfunktionen in C.62 besteht noch ein weiteres Problem. Beispielsweise kann die Berechnung von $(A_1/2)^2$ zu einem vorzeitigen Überlauf führen, während der ganze Realteil in C.62 ein noch darstellbares Intervall ist. Dies kann z.B. eintreten, wenn man im Fall $B = 0$ für Z die Nullstelle $Z_1 = -A$ des Polynoms $Z^2 + A \cdot Z + 0$ wählt und dabei das komplexe Punktintervall $A :=$ `MaxFloat()` $+ i \cdot$ `MaxFloat()`$/2$ definiert. Wegen $A_1 =$ `MaxFloat()` würde dann die Auswertung von C.62 mit Sicherheit zu einem vorzeitigen Überlauf führen. Um diesen Überlauf zu vermeiden, wird das Intervall A_1 wie folgt skaliert[5] $A_1 \subseteq 2^{k_1} \cdot \hat{A}_1$, wobei $\hat{A}_1$ möglichst groß ausfällt, aber $2 \cdot \hat{A}_1 \cdot \hat{A}_1$ noch keinen Überlauf erzeugen darf. In diesem Sinne wird in C.62 der ganze Realteil berechnet, und erst am Schluss wird mit dem entsprechenden Faktor 2^k multipliziert, so dass erst dann ein möglicher und nicht zu vermeidender Überlauf eintreten kann. Ganz analog wird der Imaginärteil berechnet. Die Berechnung der Polynomeinschließung P erfolgt mit Hilfe der Funktion

```
MpfciClass poly2(const MpfciClass& Z, const MpfciClass& A, const MpfciClass& B);
```

die in `mpfciclass.cpp` definiert ist.

C.2.6.1. Numerische Beispiele

Im **1. Beispiel** wählen wir wie oben

$$\begin{aligned} Z &= [-\texttt{MaxFloat()}, -\texttt{MaxFloat()}] + i \cdot [-\texttt{MaxFloat()/2}, -\texttt{MaxFloat()/2}] = -A, \\ B &= [0,0] + i \cdot [0,0], \end{aligned}$$

so dass das Polynom $Z^2 + A \cdot Z + 0$ an seiner Nullstelle $Z_1 = -A$ auszuwerten ist, und tatsächlich erhält man mit dem Funktionsaufruf `P = poly2(Z,A,B);` **ohne** einen vorzeitigen Überlauf die **optimale** Einschließung

$$\texttt{P} = [0,0] + i \cdot [0,0] = W.$$

Wertet man das Polynom naiv aus, d.h. durch $\texttt{sqr}(Z) \oplus A \odot Z \oplus B$, so erhält man die völlig unbrauchbare Einschließung

```
P = ([-@Inf@,@Inf@], [-@Inf@,@Inf@]).
```

[5] Die Skalierung erfolgt mit Hilfe der Funktion `Interval_Scaling(...)`, vgl. Seite 72.

Im **2. Beispiel** wählen wir $Z = [1,2] + i \cdot [-2,-1]$, $A = [-2,-2] + i \cdot [2,2]$, $B = [2,2] + i \cdot [-1,-1]$ und erhalten mit dem Funktionsaufruf `P = poly2(Z,A,B);` die **optimale** Einschließung

$$\texttt{P} = [1,3] + i \cdot [-1,+1] = W.$$

Wertet man das Polynom jedoch naiv aus, d.h. durch $\texttt{sqr}(Z) \oplus A \odot Z \oplus B$, so erhält man die viel zu grobe Einschließung

$$\texttt{P} = [-3,+7] + i \cdot [-5,+5].$$

Das Beispiel zeigt, wie wichtig es ist, wenigstens für einfache komplexe Intervallausdrücke, wie hier beim Polynom 2. Grades, optimale Einschließungen zu realisieren.

C.2.7. $\log(z)$

Der Hauptwert der komplexen Logarithmusfunktion zur Basis e ist für $z = x + i \cdot y \in \mathbb{C}$ definiert 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[6] 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 243.

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 liefert, der nicht mehr zum Hauptwert des Logarithmus gehört, da das maximale Argument π auf der Maschine durch einen etwas größeren 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 295.

[6] Die Null darf auch nicht auf dem Rand von Z liegen!

C.2.7.1. Analytische Funktion

Bedeutet $\log(z)$ den auf Seite 293 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 C.17 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z angegeben.

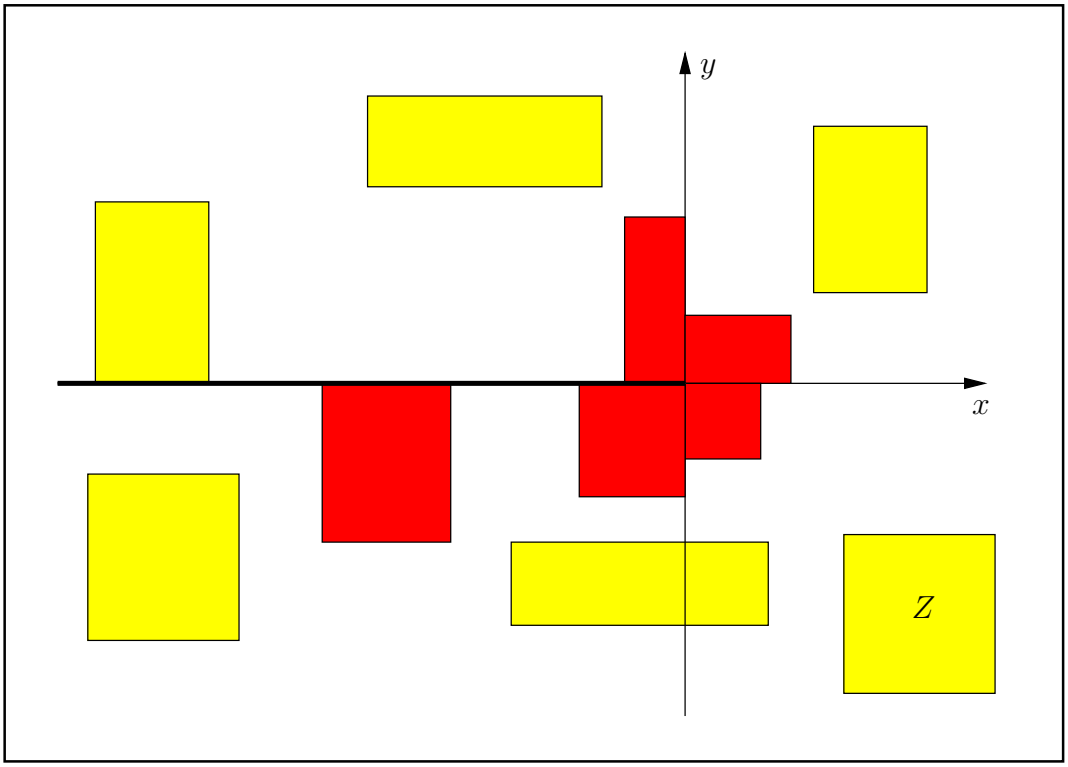


Abbildung C.17.: Erlaubte und nicht erlaubte Intervalle Z von Ln(Z).

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`.

C.2.7.2. Nicht-analytische Funktion

Ausgangspunkt ist zunächst wieder der auf Seite 293 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

$$\log(Z) := \{y \in \mathbb{C} \,|\, y = \log(z) \wedge z \in Z\}$$

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 C.18 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $\ln(Z)$ angegeben.

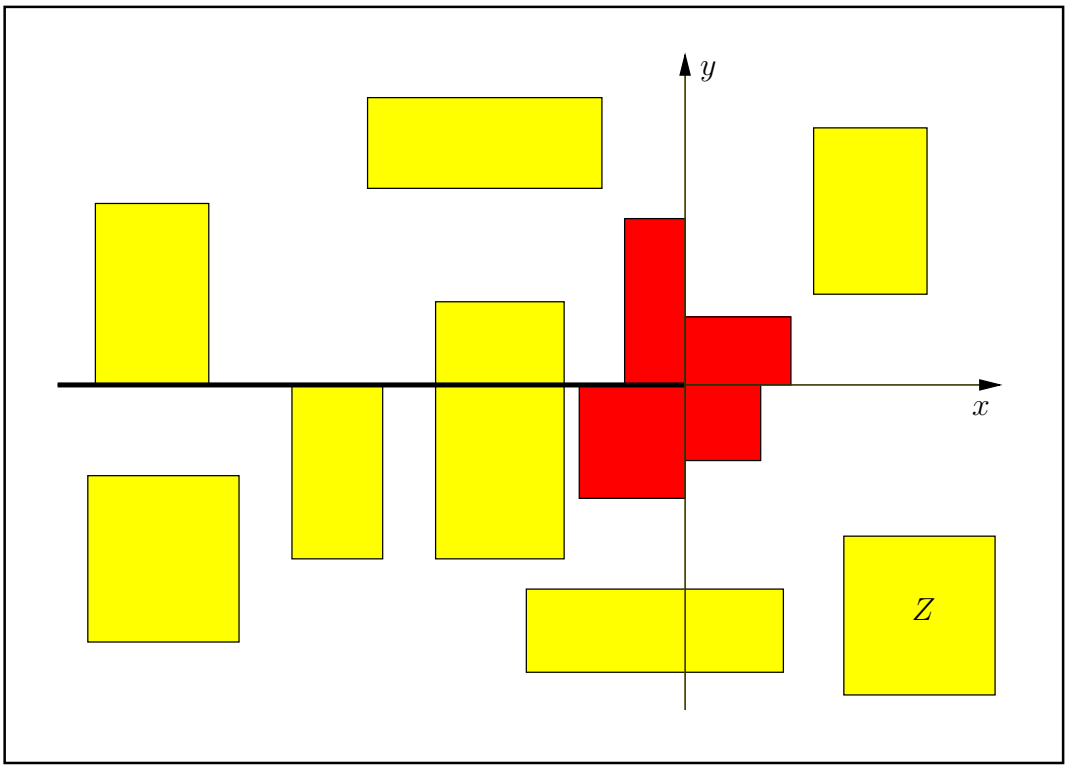


Abbildung C.18.: Erlaubte und nicht erlaubte Intervalle Z von ln(Z).

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:

$\texttt{ln}([+2,+4] + i \cdot [+0,+0]) = ([6.93147180e-1, 1.38629437], [0.00000000, 0.00000000])$,
$\texttt{ln}([-4,-2] + i \cdot [+0,+0]) = ([6.93147180e-1, 1.38629437], [3.14159265, 3.14159266])$,
$\texttt{ln}([-4,-2] + i \cdot [+0,+1]) = ([6.93147180e-1, 1.41660668], [2.67794504, 3.14159266])$,
$\texttt{ln}([-4,-2] + i \cdot [-1,+0]) = ([6.93147180e-1, 1.41660668], [-3.14159266, -2.67794504])$,
$\texttt{ln}([-4,-2] + i \cdot [-1,+1]) = ([6.93147180e-1, 1.41660668], [2.67794504, 3.60524027])$,
$\texttt{ln}([-2,+2] + i \cdot [-3,-2]) = ([6.93147180e-1, 1.28247468], [-2.35619450, -7.85398163e-1])$.

C.2.8. $\log(1+z)$

Der Hauptwert der komplexen Logarithmusfunktion zur Basis e mit dem Argument $1+z$ ist für $z = x + i \cdot y \in \mathbb{C}$ definiert durch

$$\log(1+z) := \ln(|1+z|) + i \cdot \arg(1+z), \quad |1+z| := \sqrt{(1+x)^2 + y^2}, \quad -\pi < \arg(1+z) \leq +\pi.$$

Der Verzweigungsschnitt ist die negative reelle Achse von $-\infty$ bis -1, wobei der Punkt $(-1,0)$ selbst der Verzweigungspunkt ist. $\ln(|1+z|)$ bedeutet dabei den reellen Logarithmus zur Basis e mit dem Argument $|1+z|$. Ist nun Z ein komplexes Rechteckintervall, das $(-1,0)$ nicht enthält[7] und den Verzweigungsschnitt nur von oben berühren darf, so enthält die komplexe Zahlenmenge

$$\log(1+Z) := \{y \in \mathbb{C} \,\big|\, y = \log(1+z) \wedge z \in Z\}$$

nur Funktionswerte des Hauptwertes der Logarithmusfunktion $\log(1+z)$. Beachten Sie, dass $\log(1+Z)$ i.a. kein Rechteckintervall ist. Beispielsweise ist das Bild $\log(1+Z)$ 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 243.

Bezeichnen wir mit $\mathrm{Lnp1}(Z)$ die Einschließung von $\log(1+Z)$ durch ein achsenparalleles Rechteck, so enthält auch dieses Rechteck nur Funktionswerte des Hauptwertes $\log(1+z)$ und es gilt

$$\log(1+Z) := \{y \in \mathbb{C} \,\big|\, y = \log(1+z) \wedge z \in Z\} \subseteq \mathrm{Lnp1}(Z).$$

Wir bezeichnen daher $\mathrm{Lnp1}(Z)$ als **Analytische Logarithmusfunktion**, wobei das Rechteckintervall Z die Singularität $z_S = -1 + 0 \cdot i$ nicht enthalten darf und den Verzweigungsschnitt nur von oben berühren darf. Man sollte noch beachten, dass z.B. im Fall $Z = [-3,-2] + i \cdot [0,1]$ die Einschließung

$$\mathrm{Lnp1}(Z) = \texttt{([0,8.04719e-1], [2.35619,3.14160])}$$

mit dem Supremum ihres Imaginärteils einen Wert liefert, der nicht mehr zum Hauptwert des Logarithmus gehört, da das maximale Argument π auf der Maschine durch einen etwas größeren 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 Singularität $z_S = -1 + 0 \cdot i$ weder in Z noch auf seinem Rand liegen darf. Durch analytische Fortsetzung ist es jetzt auch möglich, dass Z den Verzweigungsschnitt von unten berührt, wobei dann aber die Funktionswerte auf dem Verzweigungsschnitt den Imaginärteil $-\pi$ erhalten. Bezeichnen wir mit $\log(1+Z)$ 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 $\mathrm{lnp1}(Z)$ bezeichnet, so gilt

$$\log(1+Z) \subseteq \mathrm{lnp1}(Z), \; z \in Z.$$

Da $\mathrm{lnp1}(Z)$, je nach Lage von Z, Funktionswerte einschließen kann, die nicht zum Hauptwert der Logarithmusfunktion gehören, wird $\mathrm{lnp1}(Z)$ auch als **Nicht-analytische Logarithmusfunktion** bezeichnet. Weitere Einzelheiten zu den erlaubten Intervallen Z findet man auf Seite 299.

[7] $z_S = -1 + 0 \cdot i$ darf auch nicht auf dem Rand von Z liegen!

C.2.8.1. Analytische Funktion

Bedeutet $\log(1+z)$ den auf Seite 296 definierten Hauptwert der komplexen Logarithmusfunktion und ist Z ein achsenparalleles Rechteckintervall, wobei die Singularität $z_S = -1 + 0 \cdot i$ nicht in seinem Innern oder auf dem Rand von Z liegen darf und Z den Verzweigungsschnitt auch nur von oben berühren darf, so liefert die analytische Logarithmusfunktion $\mathrm{Lnp1}(Z)$ eine nahezu optimale achsenparallele Rechteck-Einschließung der komplexen Zahlenmenge

$$\log(1+Z) := \{y \in \mathbb{C} \,|\, y = \log(1+z) \wedge z \in Z\} \subseteq \mathrm{Lnp1}(Z).$$

In Abbildung C.19 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z angegeben.

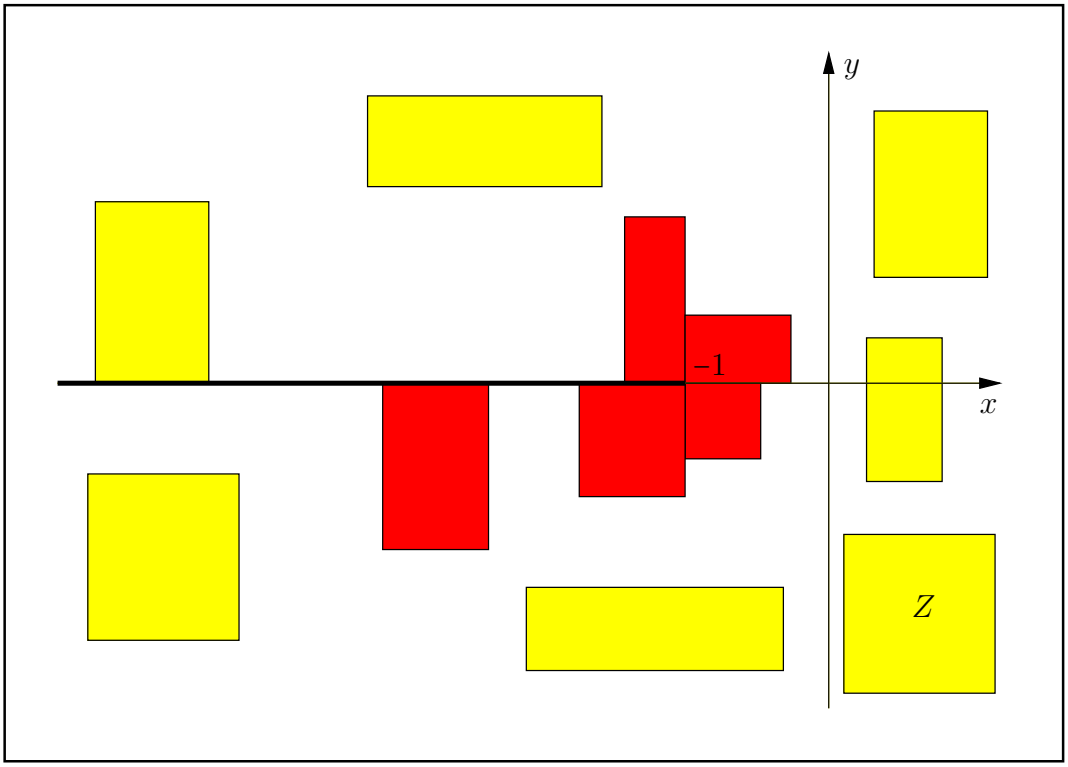


Abbildung C.19.: Erlaubte und nicht erlaubte Intervalle Z von Lnp1(Z).

Die Einschließung Lnp1(Z) wird berechnet mit der C-XSC-Funktion

```
MpfciClass Lnp1(const MpfciClass& Z);
```

die in der Datei `mpfciclass.cpp` definiert ist. Mit $Z = X + i \cdot Y$ wird der Realteil realisiert durch die C-XSC-Funktion

```
MpfiClass ln_sqrtxp1_2y2(const MpfiClass& X, const MpfiClass& Y);
```

die in der Datei `mpficlass.cpp` definiert ist, vgl. Seite 205. Der Imaginärteil wird berechnet mit der C-XSC-Funktion

```
MpfiClass argp1(const MpfciClass& Z, PrecisionType prec);
```

die in der Datei `mpfciclass.cpp` definiert ist, vgl. Seite 116.

Numerische Ergebnisse:

Bezeichnungen: $Z = X + i \cdot Y,\ X = [x_1, x_2],\ Y = [y_1, y_2], \qquad t = \log(1 + Z)$;

1. Mit $x_1 = x_2 =$ `minfloat()` $= 2^{-1073741824}$; $y_1 = y_2 =$ `minfloat()` und `prec=30 000` erhält man für t die notwendigerweise grobe Einschließung:

   $$t \subset ([0, 2.38256490488...41164e-323228497], [0, 2.38256490488...41164e-323228497]).$$

2. Mit $x_1 = x_2 = 2 \cdot$ `minfloat()` $= 2^{-1073741823}$; $y_1 = y_2 = 2 \cdot$ `minfloat()` und `prec=30 000` erhält man für t die fast optimale Einschließung, wobei Real- und Imaginärteil mit jeweils 9030 gemeinsamen Dezimalziffern berechnet werden:

   $$t \subset ([4.765129809775...82325e-323228497, 4.765129809775...82328e-323228497],$$
   $$[4.765129809775...82324e-323228497, 4.765129809775...82327e-323228497]).$$

3. Mit $x_1 = y_1 = 2 \cdot$ `minfloat()` $= 2^{-1073741823}$, $x_2 = y_2 = 4 \cdot$ `minfloat()`, und `prec=30 000` erhält man für t die fast optimale Einschließung, wobei Infimum und Supremum von Real- und Imaginärteil mit jeweils 9030 korrekten Dezimalziffern berechnet werden:

   $$t \subset ([4.765129809775...82325e-323228497, 9.530259619551...64655e-323228497],$$
   $$[4.765129809775...82324e-323228497, 9.530259619551...64653e-323228497]).$$

4. Mit $x_1 = x_2 = -1$, $y_1 = y_2 =$ `minfloat()` $= 2^{-1073741824}$ und `prec=30 000` erhält man für t die fast optimale Einschließung, wobei Real- und Imaginärteil mit jeweils 9030 gemeinsamen Dezimalziffern berechnet werden:

   $$t \subset ([-7.44261117954...03830425613365e8, -7.44261117954...03830425613363e8],$$
   $$[1.570796326794...67470301701083272, 1.570796326794...7470301701083273]).$$

   Beachten Sie, dass das Argument $Z = [-1, -1] + i \cdot$ `[minfloat(),minfloat()]` jetzt in der unmittelbaren Nähe der Singularität $z_S = -1 + i \cdot 0$ liegt und damit einen betragsmäßig sehr großen Realteil verursacht, der nahezu optimal eingeschlossen wird.

5. Mit $x_1 = x_2 = -2^{3\,000\,001}$, $y_1 = y_2 = 2^{-1\,500\,000}$ und `prec=800 000` erhält man für t die fast optimale Einschließung, wobei Real- und Imaginärteil mit jeweils 240823 gemeinsamen Dezimalziffern berechnet werden:

   $$t \subset ([1.327168685081681...30691e-1806181, 1.327168685081681...30692e-1806181],$$
   $$[1.0150887817716533...57311e-451545, 1.0150887817716533...57314e-451545]).$$

   Die sehr große Präzision `prec=800 000` ist hier notwendig, um den Realteil von t nahezu optimal einschließen zu können, vgl. dazu die Bemerkungen auf Seite 269.

6. Mit $x_1 = -3$, $x_2 = -2$, $y_1 = y_2 = 0$ und `prec=30 000` erhält man für t die fast optimale Einschließung, wobei jetzt nur der Imaginärteil mit 9030 gemeinsamen Dezimalziffern eingeschlossen wird:

   $$t \subset ([0, 6.9314718055994530941723212145817656807550013436025...6258516803e-1],$$
   $$[3.141592653589793...6034021665442, 3.141592653589793...6034021665448]).$$

   Beachten Sie, dass $Z = [-3, -2] + i \cdot [0, 0]]$ den Verzweigungsschnitt von oben berührt.

7. $Z = [-3, -2] + i \cdot [-1, 0]$ und $Z = [-1, 0] + i \cdot [0, 1]$ sind nicht erlaubte komplexe Intervalle und erzeugen entsprechende Fehlermeldungen.

C.2.8.2. Nicht-analytische Funktion

Ausgangspunkt ist zunächst wieder der schon auf Seite 296 definierte Hauptwert $\log(1+z)$ der komplexen Logarithmusfunktion. Ist Z dann ein achsenparalleles komplexes Rechteckintervall, das die Singularität $z_S = -1 + i \cdot 0$ nicht enthält und den Verzweigungsschnitt nur von oben berührt, so enthält die komplexe Zahlenmenge

$$\log(1+Z) := \{y \in \mathbb{C} \,|\, y = \log(1+z) \wedge z \in Z\}$$

nur Funktionswerte des Hauptwertes $\log(1+z)$. 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 $\mathrm{lnp1}(Z)$ eine Rechteck-Einschließung dieser Zahlenmenge $\log(1+Z)$, so gilt

$$\log(1+Z) \subseteq \mathrm{lnp1}(Z), \quad z \in Z.$$

$\mathrm{lnp1}(Z)$ wird dabei als nicht-analytische Logarithmusfunktion bezeichnet, da jetzt auch Funktionswerte eingeschlossen werden können, die nicht zum Hauptwert der Logarithmusfunktion gehören. In Abbildung C.20 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $\mathrm{lnp1}(Z)$ angegeben.

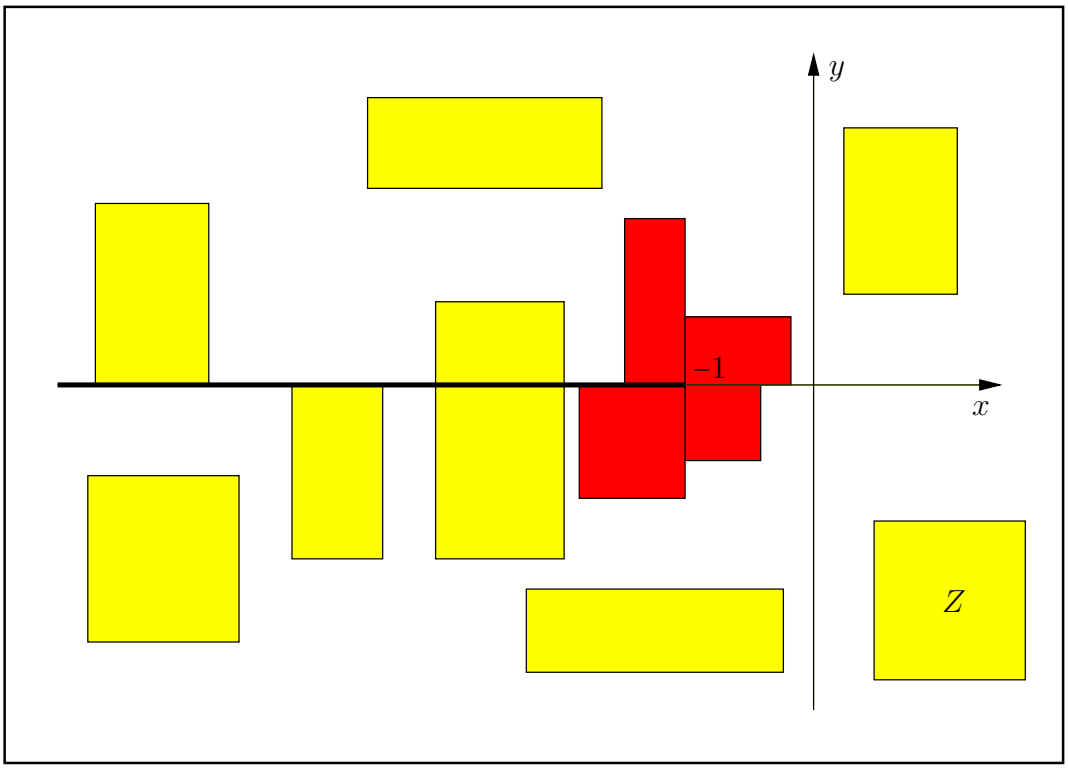


Abbildung C.20.: Erlaubte und nicht erlaubte Intervalle Z von lnp1(Z).

Numerische Ergebnisse:

Bezeichnungen: $Z = X + i \cdot Y,\ X = [x_1, x_2],\ Y = [y_1, y_2], \qquad t = \log(1+Z);$

1. Mit Ausnahme des letzten Beispiels gelten alle numerischen Ergebnisse von Seite 298 auch für die nicht-analytische Funktion `lnp1`(Z).

2. Mit $Z = [-3,-2] + i \cdot [-1,0]$ und `prec` = 30 000 erhält man die Einschließung

$$t \subset ([0, 8.0471895621705018730037966661309381976280067713425886 0...765881e-1],$$
$$[2.3561944901923449288469 8...249077, 3.9269908169872415480783 0...081817]).$$

C.2.9. $\sqrt{z}$

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 C.21 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $\sqrt{Z}$ angegeben.

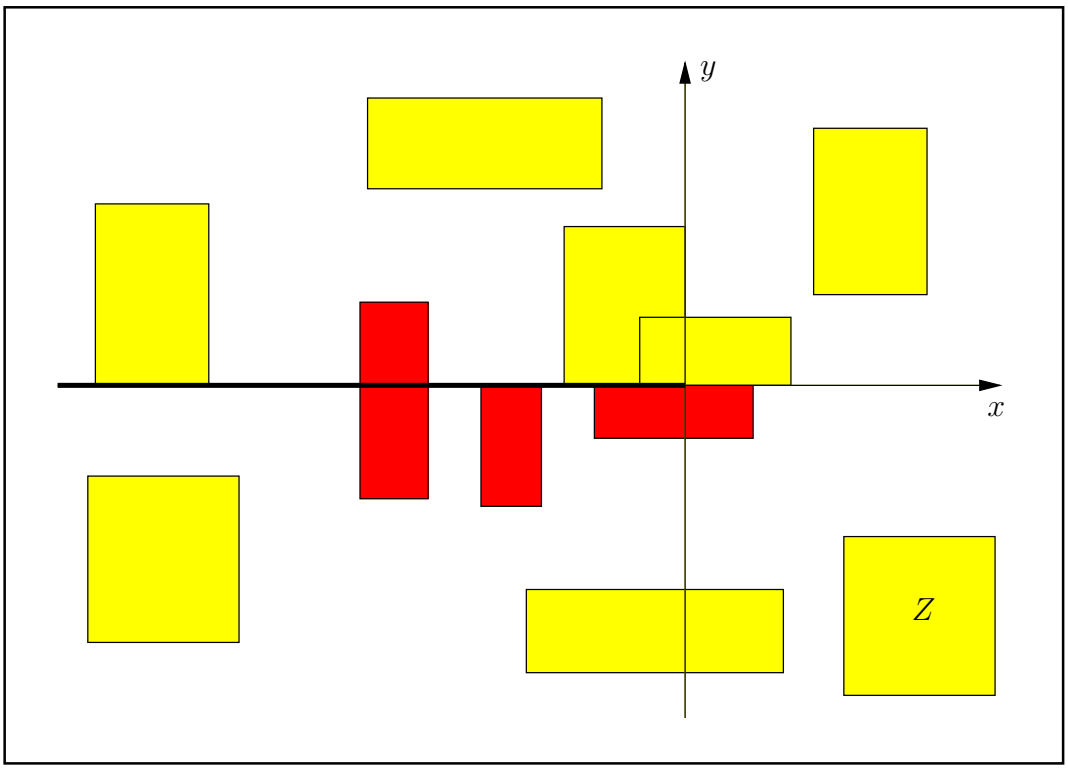


Abbildung C.21.: Erlaubte und nicht erlaubte Intervalle Z von $\sqrt{Z}$.

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.
```

C.2.10. $\sqrt{z}$, Beide Quadratwurzeln

Während mit `sqrt(`Z`)` nach Seite 300 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`$([-4,-2]+i\cdot[-4,+0]) \rightsquigarrow$
$([0.00000000, 1.11178595], [-2.19736823, -1.41421356])$
$([-1.11178595, -0.00000000], [1.41421356, 2.19736823])$.

C.2.11. $\sqrt[n]{z}$

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 komplexen Zahlen

$$w_k := \sqrt[n]{r} \cdot \left\{ \cos\left(\frac{\varphi + 2\pi k}{n}\right) + i \cdot \sin\left(\frac{\varphi + 2\pi k}{n}\right) \right\} = \sqrt[n]{r} \cdot e^{i \frac{\varphi + 2\pi k}{n}}, \quad k = 0, 1, 2, \ldots, n-1, \tag{C.63}$$

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 (C.63) für die dritten Wurzeln aus z, [62]

$$\begin{aligned} \sqrt[3]{-1+i} \;:=\; w_0 &= 2^{1/6} \cdot \{\cos(\pi/4) + i \cdot \sin(\pi/4)\}, \\ w_1 &= 2^{1/6} \cdot \{\cos(11\pi/12) + i \cdot \sin(11\pi/12)\}, \\ w_2 &= 2^{1/6} \cdot \{\cos(19\pi/12) + i \cdot \sin(19\pi/12)\}. \end{aligned}$$

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 (C.63) 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 achsenparalleles Rechteckintervall $Z \subset \mathbb{C}$ sollen jetzt für alle komplexen Zahlen $z \in Z$ die Funktionswerte des Hauptzweiges der n-ten Wurzel durch ein achsenparalleles 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 C.22 sind für $n = 2, 3, 4, \ldots$ einige erlaubte und nicht erlaubte Rechteckintervalle Z angegeben:

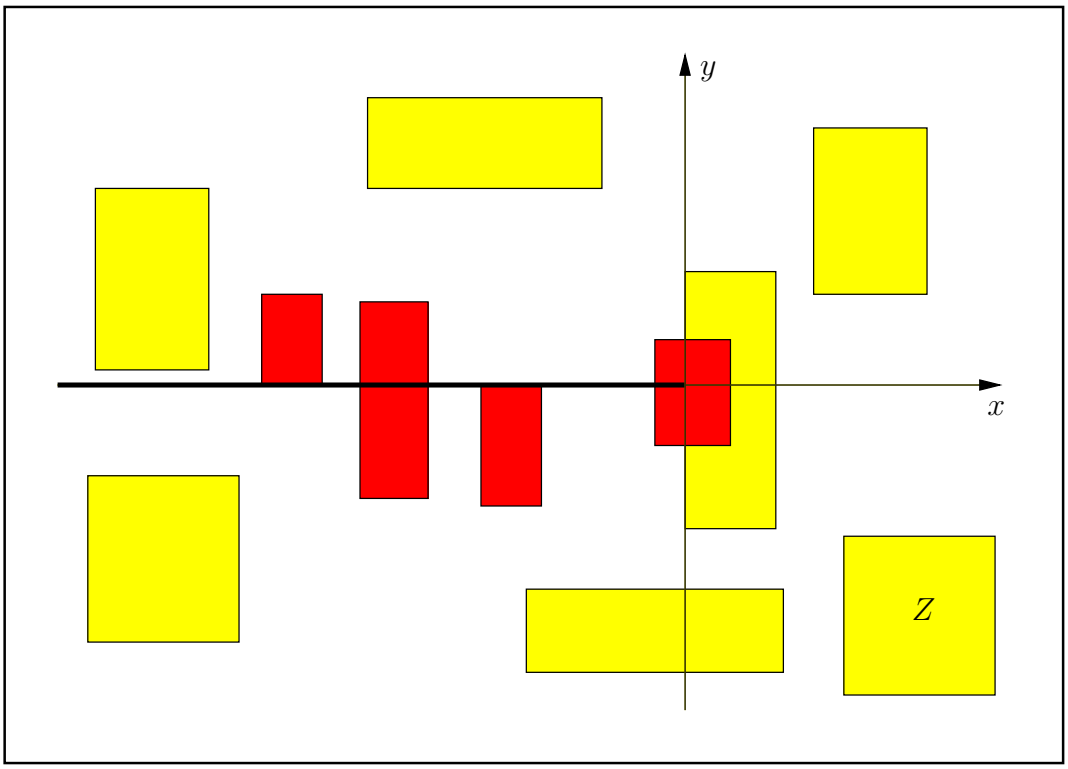


Abbildung C.22.: Erlaubte und nicht erlaubte Intervalle Z von `sqrt(Z,n)`.

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!

C.2.12. $\sqrt[n]{z}$ Alle Wurzeln

Für ein vorgegebenes achsenparalleles Rechteckintervall $Z \subset \mathbb{C}$ berechnet die Funktion, [13], [55]

```
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 302.

```
// 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 302. 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 $w^5 = 1$.

$$
\begin{aligned}
w_0 &\in ([1.00000000, 1.00000000], [0.00000000, 0.00000000]),\\
w_1 &\in ([3.09016994e-1, 3.09016995e-1], [9.51056516e-1, 9.51056517e-1]),\\
w_2 &\in ([-8.09016995e-1, -8.09016994e-1], [5.87785252e-1, 5.87785253e-1]),\\
w_3 &\in ([-8.09016995e-1, -8.09016994e-1], [-5.87785253e-1, -5.87785252e-1]),\\
w_4 &\in ([3.09016994e-1, 3.09016995e-1], [-9.51056517e-1, -9.51056516e-1]).
\end{aligned}
$$

C.2.13. $\sqrt{z+1}-1$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = \sqrt{z+1} - 1,\ \ z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision `prec` ≥ 2 zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind. Für $f(z) = \sqrt{z+1} - 1$ ist $z_V = -1 + 0 \cdot i$ der Verzweigungspunkt, und der Verzweigungsschnitt ist die Menge $\{r \in \mathbb{R} \,|\, -\infty < r < -1\}$. Für den Hauptwert von $f(z)$ darf Z den Verzweigungsschnitt nicht in seinem Inneren enthalten und auch nicht von unten berühren. In Abbildung C.21 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $F(Z) = \sqrt{Z+1} - 1 \supseteq W$ angegeben.

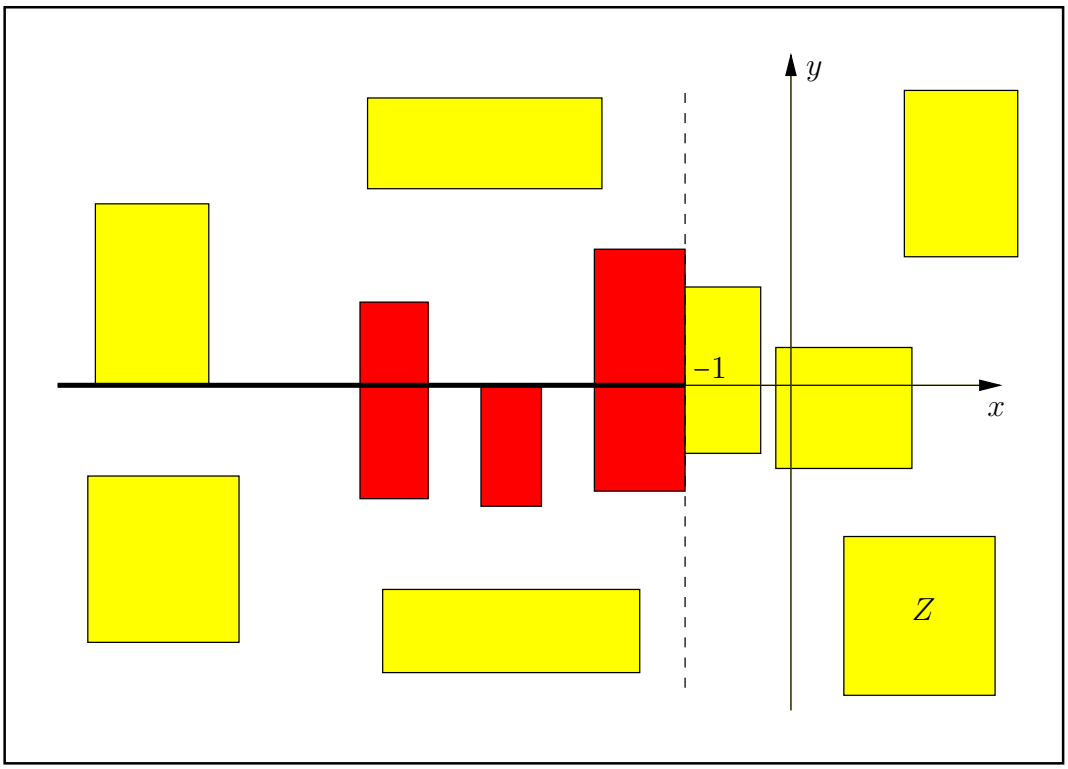


Abbildung C.23.: Erlaubte und nicht erlaubte Intervalle Z von $\sqrt{Z+1} - 1$.

Zum vorgegebenen, erlaubten Eingangsintervall `Z` wird die nahezu optimale Einschließung F berechnet mithilfe der Funktion

```
MpfciClass sqrtp1m1 (const MpfciClass& Z);
```

die in der Datei `mpfciclass.cpp` definiert ist.

Um für $F(Z) := \sqrt{Z+1} - 1 = U + i \cdot V$ die reellen Intervalle U, V für die Einschließungen des Real- und Imaginärteils optimal berechnen zu können, benötigen wir jetzt für die Real- und Imaginärteilfunktionen $u(x,y), v(x,y)$ von $f(z) = \sqrt{z+1} - 1 = u + i \cdot v$ jeweils die Minimum- und Maximumpunkte m und M auf dem **Rand** des erlaubten Eingangsintervall Z, vgl. Seite 244. Um die Lage der Punkte m und M auf dem Rand von Z bestimmen zu können, müssen für $u(x,y)$ und $v(x,y)$ jeweils die Partiellen Ableitungen nach x und y berechnet und auf Minima und Maxima der Funktionen $u(x,y)$ und $v(x,y)$ jeweils auf dem Rand von Z untersucht werden.

C.2.13.1. Realteil

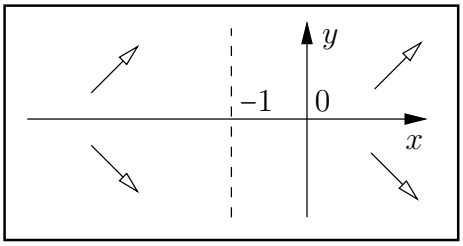
Nach (C.40) gilt für die Realteilfunktion $u(x,y)$ von $f(z) = \sqrt{z+1} - 1 = u(x,y) + i \cdot v(x,y)$ mit einem positiven Faktor $\delta(x,y) > 0$

$$u(x,y) = \sqrt{\frac{|z+1| + x + 1}{2}} - 1, \quad z = x + i \cdot y \in \mathbb{C},$$
$$\frac{\partial u(x,y)}{\partial x} = \delta \cdot \frac{(x+1) + \sqrt{(x+1)^2 + y^2}}{\sqrt{(x+1)^2 + y^2}}, \quad x \neq -1,$$
$$\frac{\partial u(x,y)}{\partial y} = \delta \cdot \frac{y}{\sqrt{(x+1)^2 + y^2}}, \quad x \neq -1.$$

Im Fall $z \neq -1 + 0 \cdot i$ gilt also $\partial u/\partial x \geq 0$, und $\partial u/\partial x = 0$ ist nur möglich auf dem Verzweigungsschnitt, d.h für $y = 0 \wedge -\infty < x < -1$. Außerhalb des Verzweigungsschnitts gilt damit:

(C.64) $$\operatorname{sign}(\frac{\partial u}{\partial x}) = 1, \quad x \neq -1,$$

(C.65) $$\operatorname{sign}(\frac{\partial u}{\partial y}) = \operatorname{sign}(y), \quad x \neq -1.$$



Nach (C.64) und (C.65) geben im linken Bild die vier Pfeile die Richtung wachsender Funktionswerte von $u(x,y)$ an.
In der unteren Abbildung sind zu einigen Eingangsintervallen Z die Randpunkte m, M angegeben, in denen das Minimum bzw. Maximum von $u(x,y)$ angenommen wird.

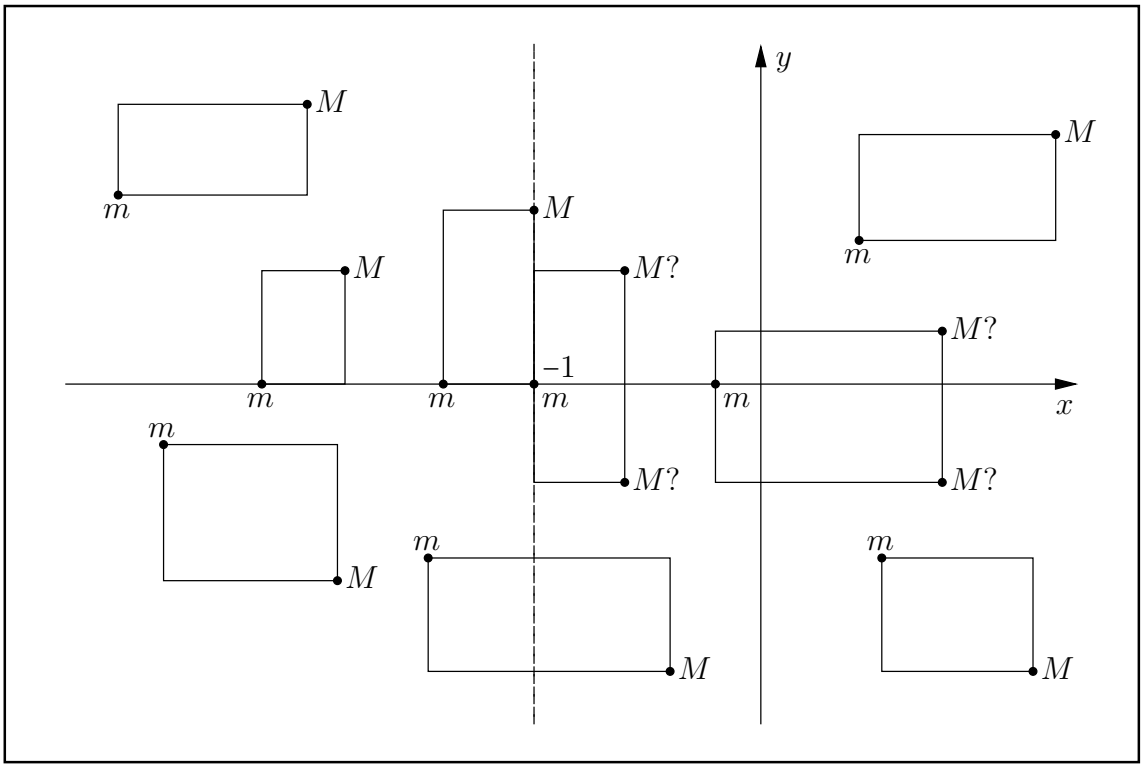


Abbildung C.24.: Minimum- und Maximumpunkte m, M der Realteilfunktion $u(x,y)$.

$Z = X + i \cdot Y, \quad X = [x_1, x_2], \; Y = [y_1, y_2]; \qquad M? = (x_2, y_1)$, falls $|y_1| > |y_2|$, sonst $M? = (x_2, y_2)$.

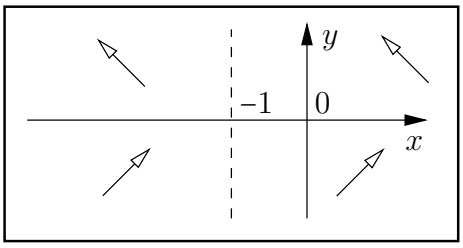
C.2.13.2. Imaginärteil

Nach (C.42) auf Seite 274 gilt für die Imaginärteilfunktion $v(x,y)$ von $f(z) = u(x,y) + i \cdot v(x,y)$ die allgemeine Darstellung

$$v(x,y) = \frac{y}{\sqrt{2 \cdot (|z+1| + x + 1)}}, \quad y \neq 0, \quad \text{und einfache Rechnungen ergeben:}$$

$$\text{(C.66)} \qquad \operatorname{sign}(\frac{\partial v}{\partial x}) = -\operatorname{sign}(y), \quad y \neq 0,$$

$$\text{(C.67)} \qquad \operatorname{sign}(\frac{\partial v}{\partial y}) = +1, \quad y \neq 0.$$



Nach (C.66) und (C.67) geben im linken Bild die vier Pfeile die Richtung wachsender Funktionswerte von $v(x,y)$ an.
In der unteren Abbildung sind zu einigen Eingangsintervallen Z die Randpunkte m, M angegeben, in denen das Minimum bzw. Maximum des Imaginärteils $v(x,y)$ angenommen wird.

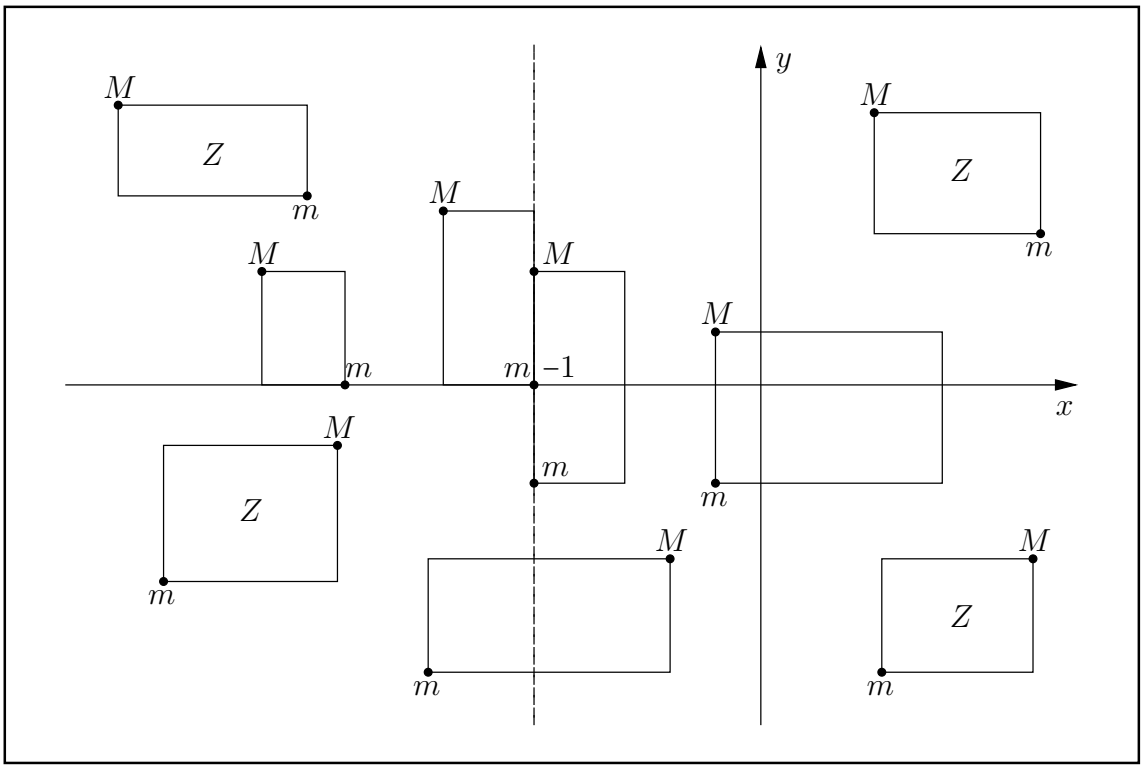


Abbildung C.25.: Minimum- und Maximumpunkte m, M der Imaginärteilfunktion $v(x,y)$.

C.2.13.3. Numerische Ergebnisse

Im **1. Beispiel** wählen wir mit $Z = X + i \cdot Y = [x_1, x_2] + i \cdot [y_1, y_2]$ und mit der Current-Präzision `prec` = 800 für Z das Punktintervall $x_1 = x_2 = y_1 = y_2 =$ `MaxFloat()` $= 2.0985\ldots \cdot 10^{+323228496}$. Für den Wertebereich W von $f(z)$, mit $z \in Z$, erhalten wir die nahezu optimale Einschließung

$$W \subseteq F(Z) = ([1.5916056670864...8630 \cdot 10^{+161614248}, 1.5916056670864...8631 \cdot 10^{+161614248}],$$
$$[6.5926465325706...2455 \cdot 10^{+161614247}, 6.5926465325706...2457 \cdot 10^{+161614247}]).$$

Man erkennt die fast optimale Einschließung an den jeweils 240 gemeinsamen Dezimalstellen.

Im **2. Beispiel** wählen wir mit $Z = X + i \cdot Y = [x_1, x_2] + i \cdot [y_1, y_2]$ und mit der Current-Präzision `prec` = 800 für Z das Punktintervall $x_1 = x_2 = 0.5,\ y_1 = y_2 = 4$. Nach Abbildung C.14 auf Seite 275 liegt dann Z im oberen rechten Eckpunkt des roten Bereiches. Für den Wertebereich W von $f(z) = \sqrt{z+1} - 1$, mit $z \in Z$, erhalten wir die nahezu optimale Einschließung

$$W \subseteq F(Z) = ([6.98823397628306387333...5497 \cdot 10^{-1}, 6.98823397628306387333...5498 \cdot 10^{-1}],$$
$$[1.17728540988554801335506 77...3088, 1.17728540988554801335506 77...3089]).$$

Man erkennt die fast optimale Einschließung an den jeweils 240 gemeinsamen Dezimalstellen.

Im **3. Beispiel** wählen wir mit $Z = X + i \cdot Y = [x_1, x_2] + i \cdot [y_1, y_2]$ und mit der Current-Präzision `prec` = 800 für Z das Punktintervall $x_1 = x_2 = 1,\ y_1 = y_2 = +6$. Für den Wertebereich W von $f(z) = \sqrt{z+1} - 1$, mit $z \in Z$, erhalten wir die nahezu optimale Einschließung

$$W \subseteq F(Z) = ([1.04016608641756892919563 25...8935, 1.04016608641756892919563 25...8936],$$
$$[1.47046851723128684330254 17...4107, 1.47046851723128684330254 17...4108]).$$

Man erkennt die fast optimale Einschließung an den jeweils 240 gemeinsamen Dezimalstellen.

Im **4. Beispiel** wählen wir mit $Z = X + i \cdot Y = [x_1, x_2] + i \cdot [y_1, y_2]$ und mit der Current-Präzision `prec` = 800 für Z das echte Intervall $Z = [0.5, 1] + i \cdot [4, 6]$. Für den Wertebereich W von $f(z) = \sqrt{z+1} - 1$, mit $z \in Z$, erhalten wir die nahezu optimale Einschließung

$$W \subseteq F(Z) = ([6.98823397628306387333...5497 \cdot 10^{-1}, 1.04016608641756892919563 2...8936],$$
$$[1.11178594050284234398409 60...8368, 1.53046699383333349802179 00...2684]).$$

Vergleichen Sie bitte die Realteileinschließung dieses Beispiels mit den Realteileinschließungen der Beispiele 2 und 3. Beachten Sie dabei die Punkte m, M des Eingangsintervalls oben rechts in Abb. C.24 auf Seite 307 und wählen Sie $m(0.5|4)$ und $M(1|6)$. Die entsprechenden Intervallgrenzen werden dabei im 4. Beispiel genau bestätigt.

Im **5. Beispiel** soll für $\alpha := \sqrt{2^{-200\,000}(1+i) + (1+2i)} - \sqrt{1+2i}$ mit der Current-Präzision `prec` = 10 000 eine möglichst optimale Einschließung berechnet werden. Mit der Umformung

$$\alpha = \sqrt{(1+2i) \cdot \left(\frac{2^{-200\,000}(1+i)}{1+2i} + 1 \right)} - \sqrt{1+2i}$$
$$= \sqrt{1+2i} \cdot \left(\sqrt{\frac{2^{-200\,000}(1+i)}{1+2i} + 1} - 1 \right)$$

kann die Funktion $f(z) = \sqrt{z+1} - 1$ direkt benutzt werden, wenn für $z := 2^{-200\,000}(1+i)/(1+2i)$ vorher eine optimale Einschließung berechnet wird. Damit erhält man für α die nahezu optimale Einschließung:

$$\alpha \in ([+3.005996416218562137...5710 \cdot 10^{-60207}, +3.005996416218562137...5712 \cdot 10^{-60207}],$$
$$[-1.001998805406187379...1904 \cdot 10^{-60207}, -1.001998805406187379...1903 \cdot 10^{-60207}])$$

mit 3009 von 3010 möglichen Dezimalstellen. Noch eine Anmerkung zur obigen Umformung: Mit $u = |u| \cdot e^{i\varphi_u}$, $v = |v| \cdot e^{i\varphi_v}$ und $-\pi < \varphi_u, \varphi_v \leq +\pi$ gilt $\sqrt{u \cdot v} = \sqrt{u} \cdot \sqrt{v}$ nur, falls $-\pi < \varphi_u + \varphi_v \leq +\pi$, wobei diese Voraussetzung bei der obigen Umformung von α erfüllt ist.

Als Gegenbeispiel betrachten wir $u = v = -1 = e^{i\varphi_u}$ mit $\varphi_u = \varphi_v = \pi$ und erhalten wegen $\varphi_u + \varphi_v = 2\pi > \pi$ die Ungleichung $1 = \sqrt{(-1) \cdot (-1)} \neq \sqrt{-1} \cdot \sqrt{-1} = i \cdot i = -1$.

C.2.14. $\sqrt{1+z^2}$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = \sqrt{1+z^2},\ \ z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision `prec` ≥ 2 zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind. Für $f(z) = \sqrt{1+z^2}$ sind $z_{\pm 1} = 0 \pm i$ die Verzweigungspunkte, und die Verzweigungsschnitte sind die Mengen $\{\pm r \cdot i \,|\, 1 \leq r < +\infty\}$. Für den Hauptwert von $f(z)$ darf Z einen Verzweigungsschnitt nicht in seinem Inneren enthalten. In der oberen Halbebene darf Z den Verzweigungsschnitt nur von rechts und in der unteren Halbebene nur von links berühren. In Abbildung C.26 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $F(Z) = \sqrt{1+Z^2} \supseteq W$ angegeben.

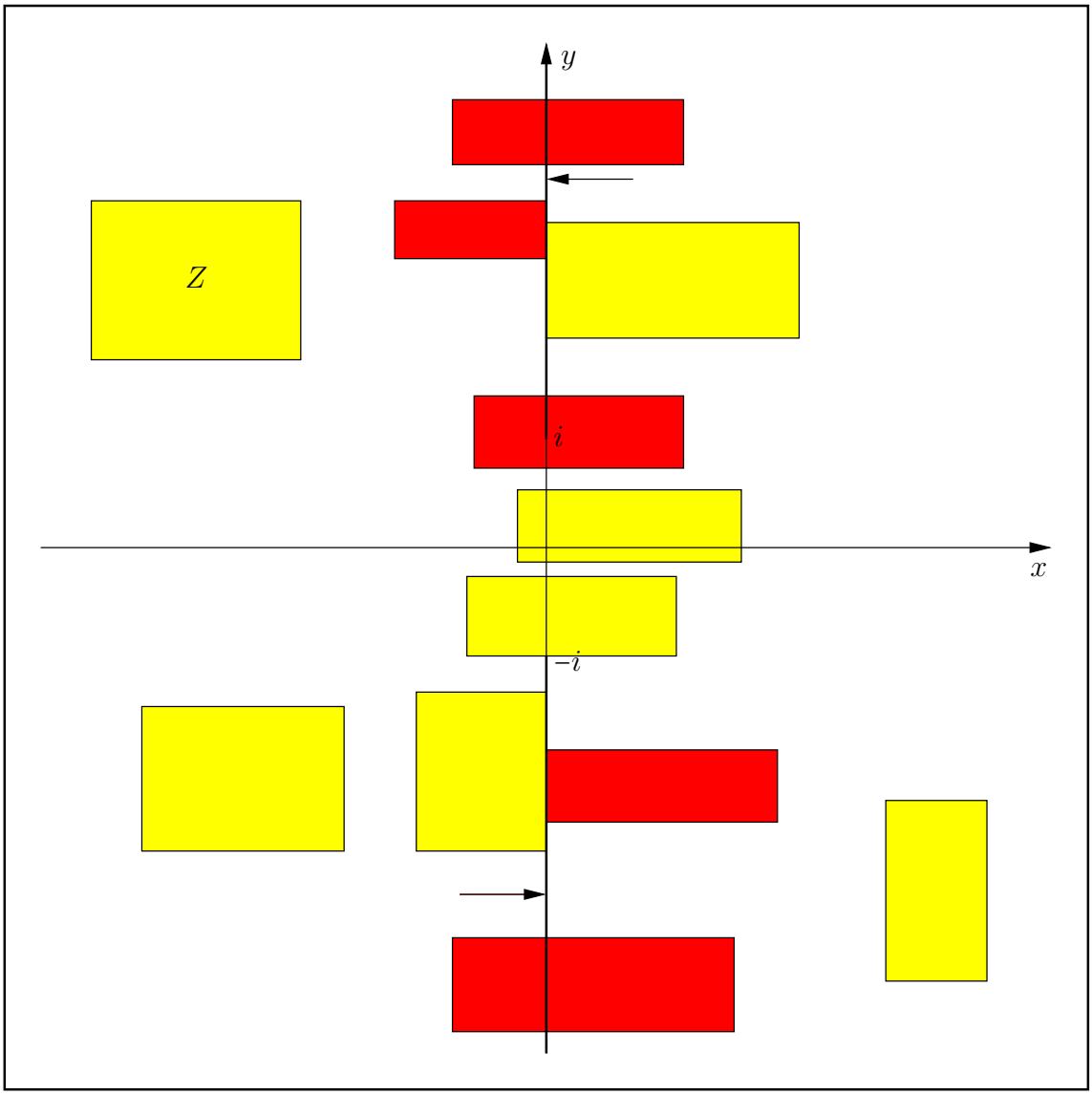


Abbildung C.26.: Erlaubte und nicht erlaubte Intervalle Z für den Hauptzweig von $f(z)$

Zu einem vorgegebenen Intervall Z wird die optimale Einschließung F berechnet mit Hilfe der Funktion

```
MpfciClass sqrt1px2(const MpfciClass& Z);
```

die in `mpfciclass.cpp` definiert ist. Da die Funktion $f(z) = \sqrt{1+z^2} = u(x,y)+i\cdot v(x,y)$ für $z \in Z$ holomorph ist, nehmen die Real- und Imaginärteilfunktionen $u(x,y)$, $v(x,y)$ als harmonische Funktionen ihre Extrema jeweils auf dem Rand von $Z = U + i \cdot V$ an, vgl. Seite 244.

C.2.14.1. Realteil

Nach (C.45) auf Seite 279 gilt für die Realteilfunktion $u(\pm x, \pm y) = u(x,y)$, so dass man für die Berechnung der Extrema m, M nur die Intervalle $X^* := \texttt{abs}(X)$ und $Y^* := \texttt{abs}(Y)$ der jeweiligen Absolutbeträge zu betrachten hat. Das komplexe Intervall $Z^* = X^* + i \cdot Y^*$ liegt damit nur im ersten Quadranten und besitzt für $u(|x|,|y|)$ die gleichen Extrema m, M wie das ursprüngliche Intervall Z für $u(x,y)$. Im ersten Quadranten erhält man für die beiden partiellen Ableitungen der Realteilfunktion $u(x,y)$ die Ergebnisse:

$$\frac{\partial u(x,y)}{\partial x} = \begin{cases} 0, & x = 0 \wedge 0 \le y < 1, \\ > 0 & \text{sonst.} \end{cases} \qquad \frac{\partial u(x,y)}{\partial y} = \begin{cases} 0, & y = 0 \vee (x = 0 \wedge y \ge +1), \\ < 0 & \text{sonst.} \end{cases}$$

Damit ergibt sich in Abb. C.27 für drei typische komplexe Intervalle Z^* aus dem 1. Quadranten die Lage der Punkte mit den gesuchte Extrema m, M.

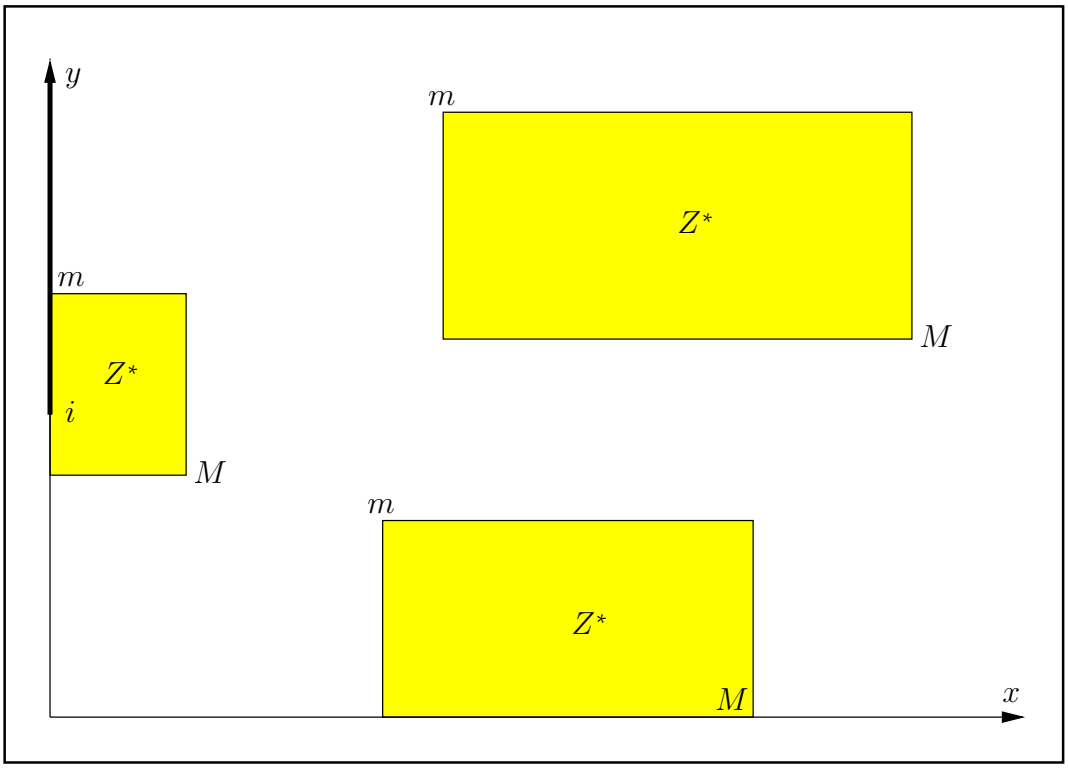


Abbildung C.27.: Lage der Extrema m, M der Realteilfunktion $u(x,y)$ auf Intervallen Z^*

Mit Hilfe der Koordinaten x_m, y_m bzw. x_M, y_M lassen sich dann das Minimum m bzw. das Maximum M von $u(x,y)$ für ein Intervall Z^* aus dem 1. Quadranten berechnen mit der Funktion

```
MpfrClass Re_sqrt1px2(const MpfrClass& x, const MpfrClass& y, RoundingMode rnd)
```

Eine Unterschranke `m` des Minimums $m \ge$ `m` und eine Oberschranke `M` des Maximums $M \le$ `M` erhält man also mit den Funktionsaufrufen

```
m = Re_sqrt1px2(xm, ym, RoundDown);      M = Re_sqrt1px2(xM, yM, RoundUp);
```

C.2.14.2. Imaginärteil

Nach (C.57) von Seite 282 gilt für die Imaginärteilfunktion

(C.68) $$v(-x,y) = -v(x,y) \qquad \text{und} \qquad v(x,-y) = -v(x,y)$$

so dass über einem gegebenen Rechteck $Z = X + i \cdot Y$, mit $\mathtt{Sup}(X) \le 0$, die Imaginärteilfunktion $v(x,y)$ die gleichen Funktionswerte liefert wie über dem Rechteck $Z^* := -X + i \cdot (-Y) = -Z$, das dann jedoch nur noch in der rechten Halbebene liegt. Um die gesuchten Extrema m, M über Z zu bestimmen, müssen neben den Intervallen Z^* aus der rechten Halbebene wegen der auf der imaginären Achse liegenden Verzweigungsschnitte nur noch komplexe Intervalle $Z = X + i \cdot Y$ betrachtet werden, für die mit $X = [x_1, x_2]$ gilt: $0 \in (x_1, x_2)$ und $|y| < 1 \forall y \in Y$.

Zur Bestimmung der Koordinaten der Extremstellen benötigen wir die partiellen Ableitungen

$$\mathtt{sign}\left(\frac{\partial v(x,y)}{\partial x}\right) = \mathtt{sign}(y), \qquad \frac{\partial v(x,y)}{\partial y} = \begin{cases} 0, & x = 0 \wedge |y| \le 1, \\ > 0, & x > 0 \vee (x = 0 \wedge |y| > 1), \\ < 0, & x < 0 \vee (x = 0 \wedge |y| > 1). \end{cases}$$

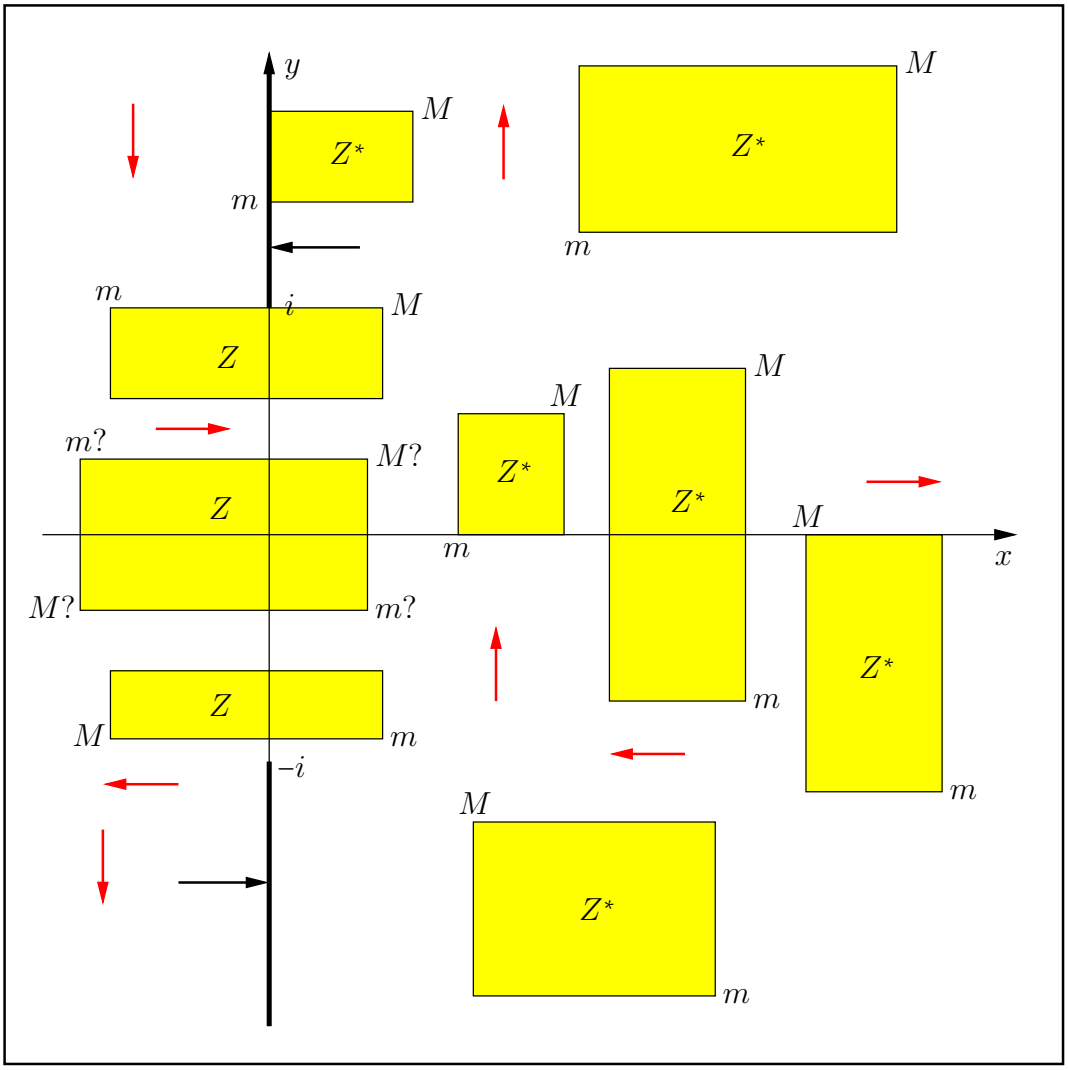


Abbildung C.28.: Extremstellen auf den Intervallrändern von Z bzw. Z^*

Die beiden schwarzen Pfeile geben die Richtungen an, aus denen die Funktionswerte $v(x,y)$ auf den jeweiligen Verzweigungsschnitt stetig ergänzt werden. Die roten Pfeile geben die Richtung wachsender Funktionswerte von $v(x,y)$ an.

Beachten Sie bitte, dass in Abb. C.28 mit Z^* neben den am Ursprung gespiegelten Intervallen aus der linken Halbebene natürlich auch solche Intervalle symbolisiert werden, die ursprünglich in der rechten Halbebene vorgegeben waren.

C.2.14.3. Numerische Ergebnisse

Im **1. Beispiel** wählen wir $Z = [0, \texttt{minfloat()}] + i \cdot [1,1] = [1, 2^{-1073741824}] + i \cdot [1,1]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$, mit $z \in Z$, die nahezu optimale Einschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0, 4.881152430408...027 \cdot 10^{-161614249}], [0, 4.881152430408...027 \cdot 10^{-161614249}]).$$

Im **2. Beispiel** wählen wir $Z = [-2,+1] + i \cdot [-1,+0.5]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$\begin{aligned} W \subseteq F(Z) = ([0, 2.23606797749978969640917366873127623544061835961152572427...8351], \\ [-7.861513777574232860 69...1307 \cdot 10^{-1}, 9.101797211244546826 08...9583 \cdot 10^{-1}]). \end{aligned}$$

Im **3. Beispiel** wählen wir $Z = [-2,-1] + i \cdot [-4,+2]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$\begin{aligned} W \subseteq F(Z) = ([+1.03065889036991751569220...6366, +2.23606797749978969640917 3...8351] \\ [-1.879129818333282376255771...9882, 3.899774854972245632918720...7245]). \end{aligned}$$

Zeigen Sie mithilfe der Symmetrieeigenschaften von $u(x,y)$ und $v(x,y)$, dass mit dem komplexen Eingangsintervall $Z_1 = [1,2] + i \cdot [-2,4]$ die gleiche obige Einschließung $F(Z)$ berechnet wird.

Im **4. Beispiel** wählen wir $Z = [2^{+1073741000}, 2^{+1073741820}] + i \cdot [2^{+1073741000}, 2^{+1073741820}]$. Mit `prec` = 400 und

$$\begin{aligned} Z = [9.379481717334163189...964 \cdot 10^{323228247}, 6.558058488960586538...127 \cdot 10^{323228494}] \\ [9.379481717334163189...964 \cdot 10^{323228247}, 6.558058488960586538...127 \cdot 10^{323228494}] \end{aligned}$$

erhalten wir für den Wertebereich W von $f(z) = \sqrt{1+z^2}$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$\begin{aligned} W \subseteq F(Z) = [9.3794817173341631...963 \cdot 10^{323228247}, 6.5580584889605865...128 \cdot 10^{323228494}] \\ [9.3794817173341631...963 \cdot 10^{323228247}, 6.5580584889605865...127 \cdot 10^{323228494}]. \end{aligned}$$

Beachten Sie, dass bei der naiven Intervallauswertung von Z^2 ein vorzeitiger Überlauf eintreten würde und dass wegen $\texttt{Inf}(\Re(Z)) \gg 1$ und $\texttt{Inf}(\Im(Z)) \gg 1$ die Intervalle Z und $F(Z)$ nahezu identisch sind.

Anmerkung:

Es sei nochmals betont, dass die Funktion `sqrt1px2(Z)` eine **optimale** Einschließung $F(Z)$ der Wertemenge W von $f(z) = \sqrt{1+z^2}$ für alle $z \in Z$ berechnet. Wenn man jedoch, z.B. mit dem Intervall $Z = [-2,+1] + i \cdot [-1,0.5]$ aus dem 2. Beispiel, den Funktionsterm $\sqrt{1+z^2}$ in naiverweise intervallmäßig auswertet, so erhält man mit dem Ergebnis

$$\begin{aligned} \sqrt{1 \oplus Z^2} = ([0, 2.3877944046161981789417931481130409131181119551430533155 7...6078], \\ [-1.000...001, 1.4142135623730950488016887242096980785696718753 7...9703]). \end{aligned}$$

eine deutliche Überschätzung der optimalen Einschließung $F(Z)$ aus dem 2. Beispiel. Der obige linke Ausdruck $\sqrt{1 \oplus Z^2}$ soll die naive intervallmäßige Auswertung symbolisieren.

C.2.15. $\sqrt{1-z^2}$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = \sqrt{1-z^2},\ \ z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision `prec` ≥ 2 zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind. Für $f(z) = \sqrt{1-z^2}$ sind $z_{\pm 1} = \pm 1$ die Verzweigungspunkte, und die Verzweigungsschnitte sind die Mengen $\{\pm r \,|\, 1 \leq r < +\infty\}$. Für den Hauptwert von $f(z)$ darf Z einen Verzweigungsschnitt nicht in seinem Inneren enthalten. In der rechten Halbebene darf Z den Verzweigungsschnitt nur von unten und in der linken Halbebene nur von oben berühren. In Abbildung C.29 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion $F(Z) = \sqrt{1-Z^2} \supseteq W$ angegeben.

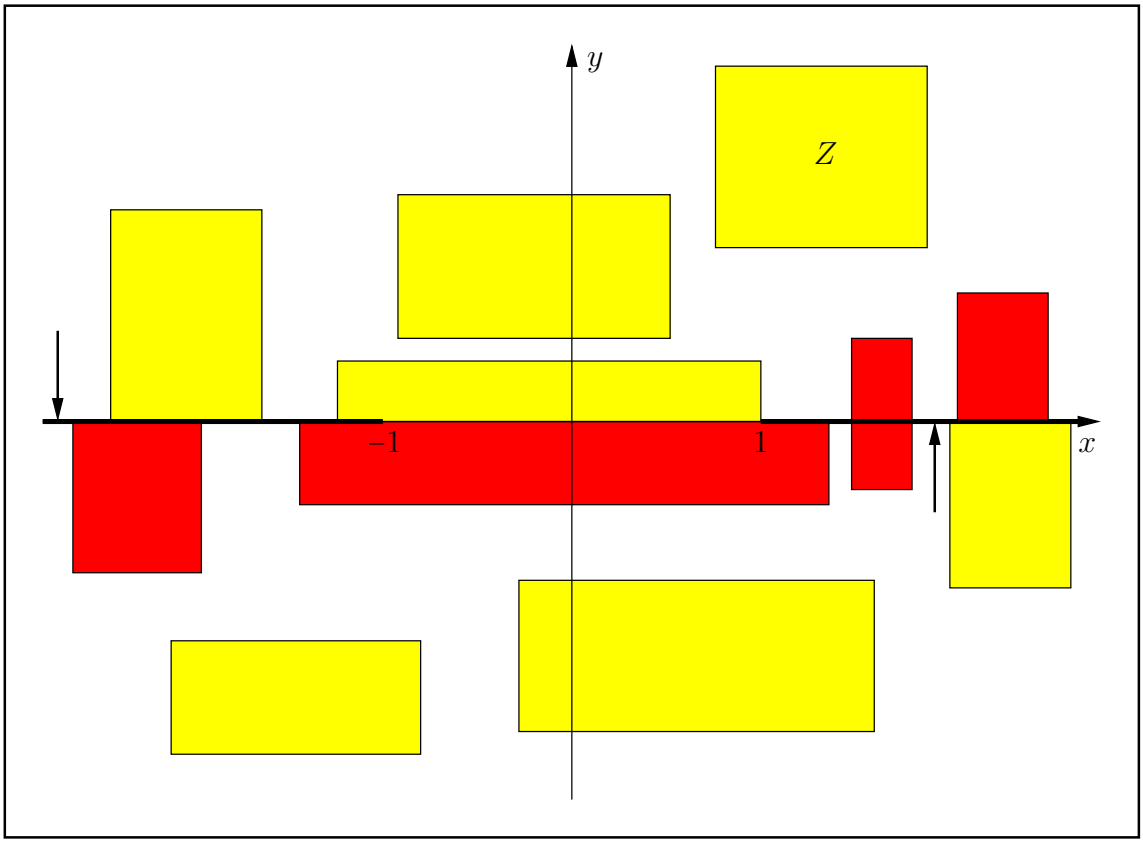


Abbildung C.29.: Erlaubte und nicht erlaubte Intervalle Z für den Hauptzweig von $f(z)$

Ganz analog zu Seite 283 erhält man mit der Transformation $Z = i \cdot Z_T$, bzw. mit $Z_T = Y + i \cdot (-X)$ die Beziehung $\sqrt{1-Z^2} = \sqrt{1+Z_T^2}$, so dass die Auswertung von $\sqrt{1-Z^2}$ zurückgeführt werden kann auf die Auswertung von $\sqrt{1+Z_T^2}$.

Die **optimale** Einschließung des Wertebereichs W durch das komplexe Rechteck $F(Z) \supseteq W$ erfolgt mithilfe der Funktion

```
MpfciClass sqrt1mx2(const MpfciClass& Z);
```

die in der Datei `pmfciclass.cpp` definiert ist.

C.2.16. $\sqrt{z^2-1}$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = \sqrt{z^2-1},\ \ z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision `prec` ≥ 2 zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind. Für $f(z) = \sqrt{z^2-1}$ sind $z_{\pm 1} = \pm 1$ die Verzweigungspunkte. Die beiden Verzweigungsschnitte sind in der folgenden Abbildung C.30 schwarz bzw. grün dargestellt, vgl. Seite 284. Typische erlaubte und nicht erlaubte komplexe Intervalle Z sind entsprechend gelb bzw. rot dargestellt.

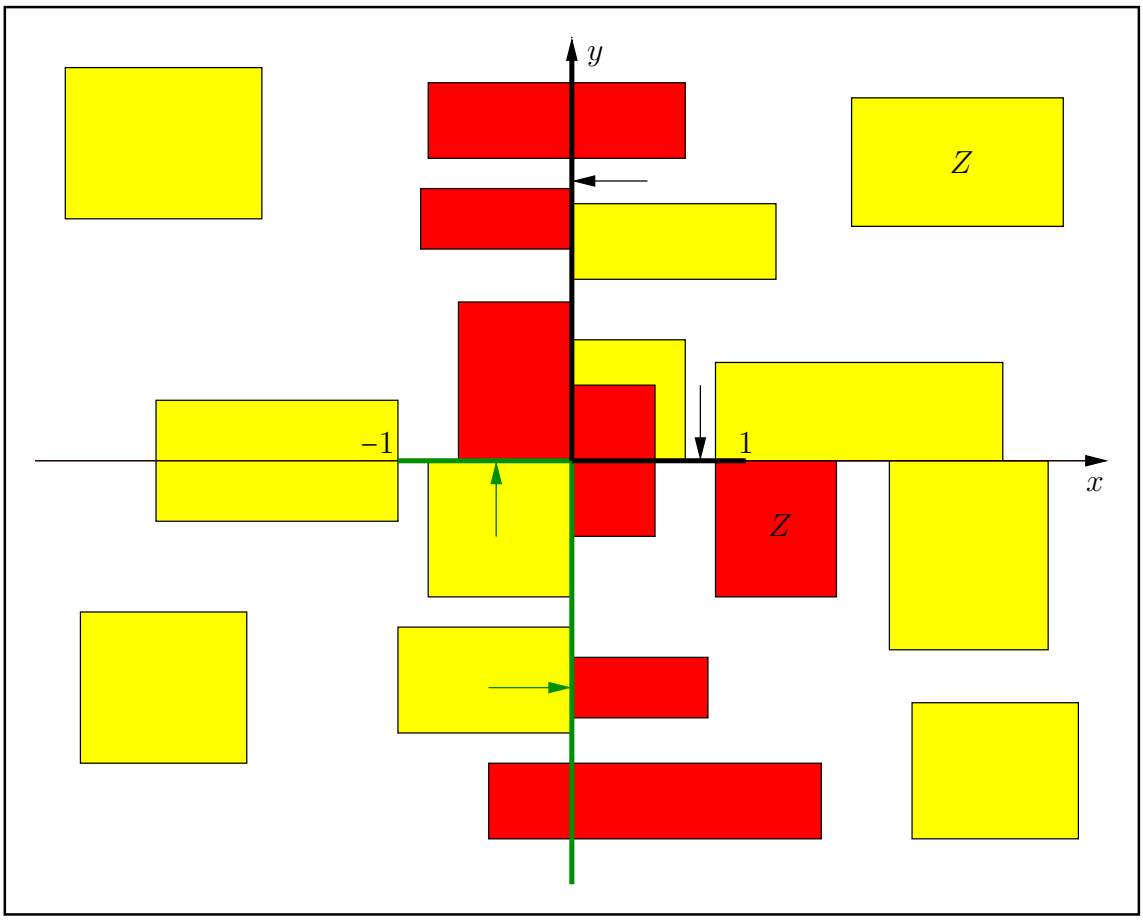


Abbildung C.30.: Erlaubte und nicht erlaubte Intervalle Z für den Hauptzweig von $f(z)$

Beachten Sie, dass auch jedes Intervall der Form $Z = [0,0] + i \cdot Y$ oder $Z = X + i \cdot [0,0]$, d.h. jedes Intervall Z, das auf einer Koordinatenachse liegt, ein erlaubtes Intervall ist. Die schwarzen und die grünen Pfeile geben die Richtungen an, aus denen die Funktionswerte von $f(z) = \sqrt{z^2-1}$, mit $z \in \mathbb{C}$, auf die jeweiligen Verzweigungsschnitte analytisch fortgesetzt werden.

Die **optimale** Einschließung des Wertebereichs W durch das komplexe Rechteck $F(Z) \supseteq W$ erfolgt mithilfe der Funktion

```
MpfciClass sqrtx2m1(const MpfciClass& Z);
```

die in der Datei `pmfciclass.cpp` definiert ist.

C.2.16.1. Realteil

Nach (C.59) auf Seite 284 gilt für die Realteilfunktion $\hat{u}(\pm x, \pm y) = \hat{u}(x, y)$, so dass man für die Berechnung der Extrema m, M nur die Intervalle $X^* := \texttt{abs}(X)$ und $Y^* := \texttt{abs}(Y)$ der jeweiligen Absolutbeträge zu betrachten hat. Das komplexe Intervall $Z^* = X^* + i \cdot Y^*$ liegt damit nur im ersten Quadranten und besitzt für $\hat{u}(|x|, |y|)$ die gleichen Extrema m, M wie das ursprüngliche Intervall Z für $\hat{u}(x, y)$. Für die partiellen Ableitungen erhält man die Ergebnisse

$$\begin{aligned}
\frac{\partial \hat{u}(x,y)}{\partial x} &> 0, \quad \text{falls } y > 0, \\
\frac{\partial \hat{u}(x,y)}{\partial x} &= 0, \quad \text{falls } y = 0 \wedge 0 \le x < 1, \\
\frac{\partial \hat{u}(x,y)}{\partial x} &= \frac{x}{\sqrt{x^2 - 1}} > 0, \quad \text{falls } y = 0 \wedge x > 1;
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \hat{u}(x,y)}{\partial y} &= 0, \quad \text{falls } x = 0, \\
\frac{\partial \hat{u}(x,y)}{\partial y} &> 0, \quad \text{falls } x > 0.
\end{aligned}$$

Damit erhält man in Abb. C.31 für einige typische Intervalle Z^* aus dem 1. Quadranten die Lage der Punkte mit den gesuchte Extrema m, M der Realteilfunktion $\hat{u}(x, y)$.

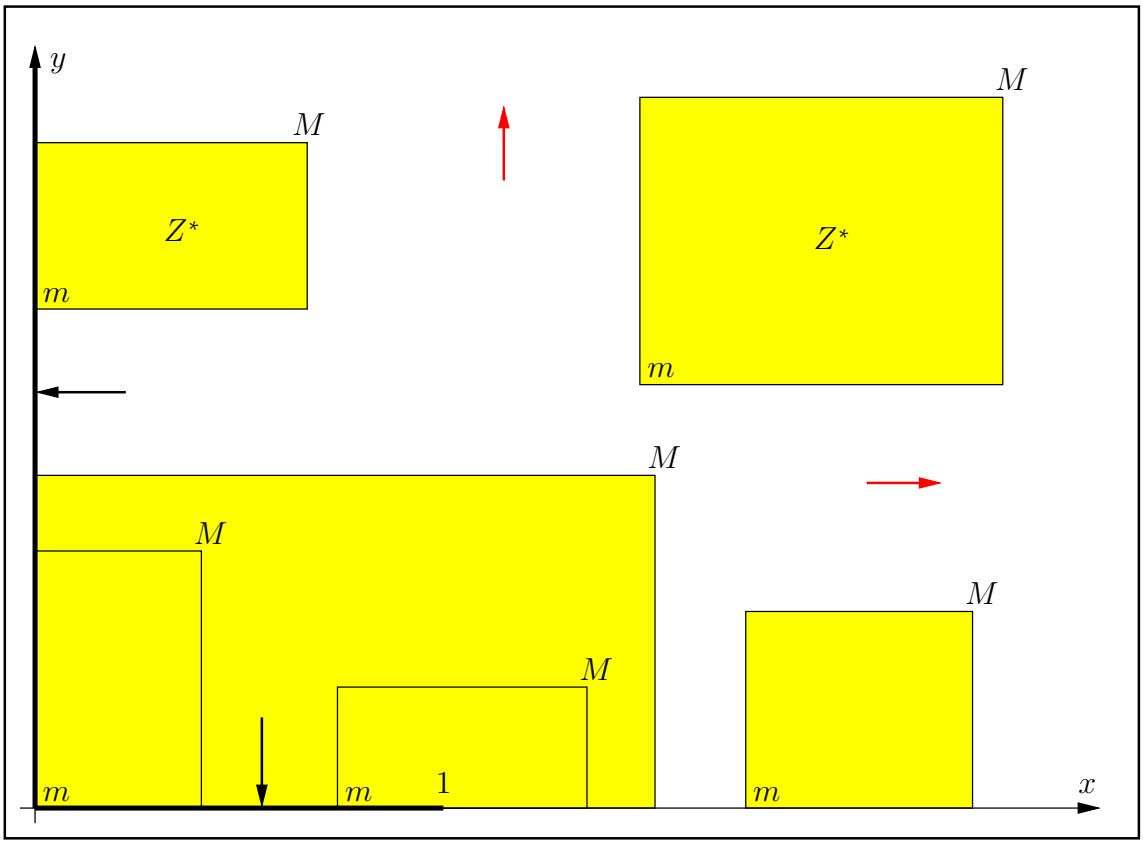


Abbildung C.31.: Extremstellen m, M von $\hat{u}(x, y)$ auf den Intervallrändern von Z bzw. Z^*

Die beiden schwarzen Pfeile geben die Richtungen an, aus der die Funktionswerte $\hat{u}(x, y)$ auf den Verzweigungsschnitt stetig ergänzt werden. Die roten Pfeile geben die Richtung wachsender Funktionswerte von $\hat{u}(x, y)$ an.

C.2.16.2. Imaginärteil

Mit Ausnahme der Intervalle Z, die auf der reellen Achse liegen, befinden sich nach Abb. C.30 alle restlichen erlaubten Intervalle Z entweder in der rechten oder linken Halbebene. Im Falle $y_1 = y_2 = 0$ liegen dann nach der Transformation $Z^* := \texttt{abs}(X) + i \cdot [0,0]$ **alle** erlaubten Intervalle Z bzw. Z^* entweder in der rechten oder linken Halbebene. Im Fall $x_1 = x_2 = 0$ wählen wir die Transformation $Z^* := [0,0] + i \cdot \texttt{abs}(Y)$. Wählt man dann noch im Fall $x_1 < 0$ die Transformation $Z^* := -X - i \cdot Y = -Z$, so befinden sich anschließend wegen $\hat{v}(-x,y) = -\hat{v}(x,y)$ und $\hat{v}(x,-y) = -\hat{v}(x,y)$ alle erlaubten Intervalle Z, Z^* nur noch in der rechten Halbebene, wobei die Wertemengen der Funktion $\hat{v}(x,y)$ über den Intervallen Z^* und deren Urbildern Z jeweils gleich sind, vgl. (C.60) auf Seite 284. Für die partiellen Ableitungen erhält man in der rechten Halbebene

$$\begin{aligned}
\frac{\partial \hat{v}(x,y)}{\partial x} &= 0, \quad y = 0 \wedge x > 1, \\
\frac{\partial \hat{v}(x,y)}{\partial x} &< 0, \quad y = 0 \wedge 0 < x < 1, \\
\texttt{sign}\left(\frac{\partial \hat{v}(x,y)}{\partial x}\right) &= -\texttt{sign}(y), \quad y \neq 0 \\
\\
\frac{\partial \hat{v}(x,y)}{\partial y} &= 0, \quad y = 0 \wedge 0 \leq x < 1, \\
\frac{\partial \hat{v}(x,y)}{\partial y} &> 0, \quad \text{sonst.}
\end{aligned}$$

Damit erhält man in Abb. C.32 für einige typische Intervalle Z^* aus der rechten Halbebene die Lage der Punkte mit den gesuchten Extrema m, M der Imaginärteilfunktion $\hat{v}(x,y)$.

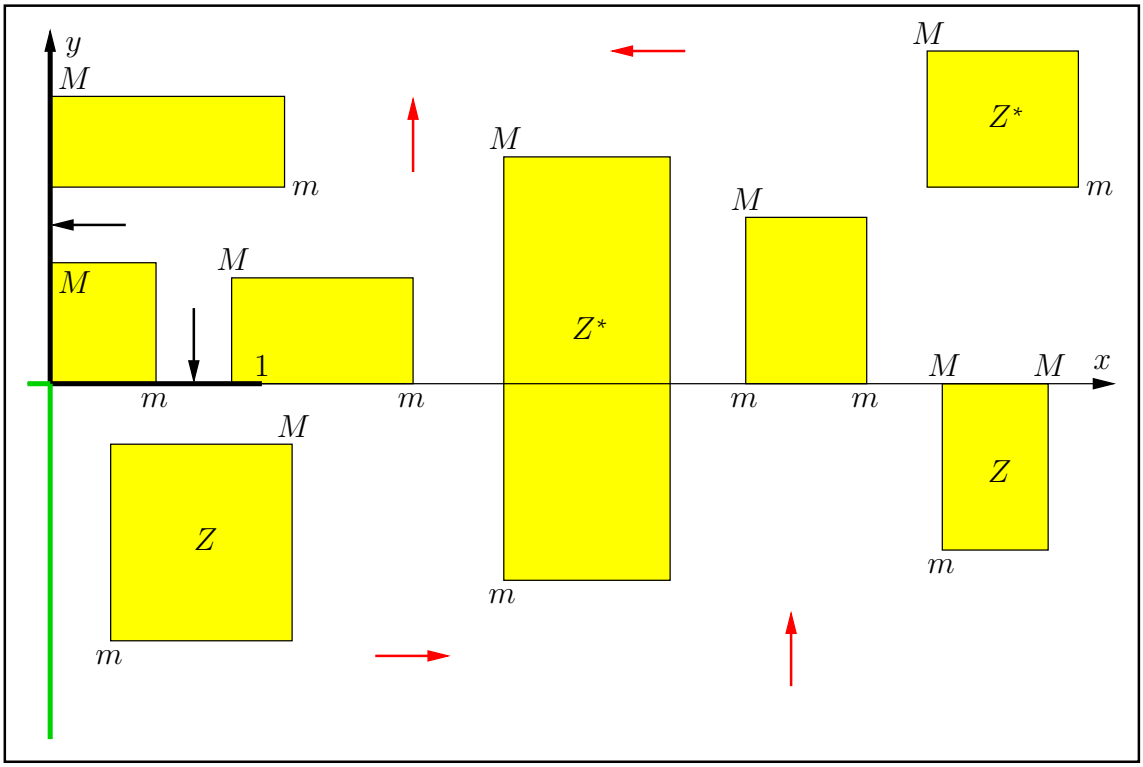


Abbildung C.32.: Extremstellen m, M von $\hat{v}(x,y)$ auf den Intervallrändern von Z bzw. Z^*

Die roten Pfeile geben die Richtung wachsender Funktionswerte von $\hat{v}(x,y)$ an. Auf der reellen Achse ist $\hat{v}(x,0)$ für $x \geq 1$ konstant, und für $0 \leq x < 1$ wächst $\hat{v}(x,0)$ in Richtung zum Ursprung.

C.2.16.3. Numerische Ergebnisse

Im **1. Beispiel** liegt $Z = [-3,2] + i \cdot [0,0]$ auf der reellen Achse. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$, mit $z \in Z$, die nahezu optimale Einschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0, 2.828427124746190097603377448419396157139343750753 8...9406], [0,1]).$$

Beachten Sie, dass man nach (C.59) und (C.60) auf Seite 284 z.B. für $Z_1 = [0,3] + i \cdot [0,0]$ die gleiche obige Einschließung $F(Z) = F(Z_1)$ erhält.

Im **2. Beispiel** liegt $Z = [0,0] + i \cdot [-3,2]$ auf der imaginären Achse. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$, mit $z \in Z$, die nahezu optimale Einschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0,0], [1, 3.162277660168379331998893544432718533719555139325216826 8]).$$

Beachten Sie, dass man nach (C.59) und (C.60) auf Seite 284 z.B. für $Z_1 = [0,0] + i \cdot [0,3]$ die gleiche obige Einschließung $F(Z) = F(Z_1)$ erhält.

Im **3. Beispiel** wählen wir $Z = [-2,-1] + i \cdot [-4,+2]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$\begin{aligned} W \subseteq F(Z) = ([0, 1.95134929319282850287643038811238266618366236389778578 54...1558] \\ [-2.19736822693561993207960 2...4282, 4.11634205454298450064396 2...7563]). \end{aligned}$$

Zeigen Sie mithilfe der Symmetrieeigenschaften von $\hat{u}(x,y)$ und $\hat{v}(x,y)$, dass mit dem komplexen Eingangsintervall $Z_1 = [1,2] + i \cdot [-2,4]$ die gleiche obige Einschließung $F(Z)$ berechnet wird.

Im **4. Beispiel** wählen wir $Z = [-1,0] + i \cdot [-2,0]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0, 9.101797211244546826 08...9583 \cdot 10^{-1}], [0, 2.236067977499789696 40...8351]).$$

Zeigen Sie mithilfe der Symmetrieeigenschaften von $\hat{u}(x,y)$ und $\hat{v}(x,y)$, dass mit dem komplexen Eingangsintervall $Z_1 = [0,1] + i \cdot [0,2]$ die gleiche obige Einschließung $F(Z)$ berechnet wird.

Im **5. Beispiel** wählen wir $Z = [0.5,1.5] + i \cdot [0,1]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0, 1.2767925917505305352 49...3706], [0, 1.372145115699253839874 24...0792]).$$

Zeigen Sie mithilfe der Symmetrieeigenschaften von $\hat{u}(x,y)$ und $\hat{v}(x,y)$, dass mit dem komplexen Eingangsintervall $Z_1 = [-1.5,-0.5] + i \cdot [-1,0]$ die gleiche obige Einschließung $F(Z)$ berechnet wird.

Im **6. Beispiel** wählen wir das Punktintervall $Z = [2,2] + i \cdot [3,3]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$\begin{aligned} W \subseteq F(Z) = ([1.92566973609167184450654 ...2310, 1.92566973609167184450654 ...2311] \\ [3.11579908410336514259287 ...6742, 3.11579908410336514259287 ...6743]). \end{aligned}$$

Zeigen Sie mithilfe der Symmetrieeigenschaften von $\hat{u}(x,y)$ und $\hat{v}(x,y)$, dass mit dem komplexen Eingangsintervall $Z_1 = [-2,-2] + i \cdot [-3,-3]$ die gleiche obige Einschließung $F(Z)$ berechnet wird.

C.2.17. $\sqrt{-z^2-1}$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = \sqrt{-z^2-1},\ \ z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision `prec` ≥ 2 zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind. Für $f(z) = \sqrt{-z^2-1}$ sind $z_{\pm 1} = \pm i$ die Verzweigungspunkte. Die beiden Verzweigungsschnitte sind in der folgenden Abbildung C.33 schwarz bzw. grün dargestellt, vgl. Seite 286. Typische erlaubte und nicht erlaubte komplexe Intervalle Z sind entsprechend gelb bzw. rot dargestellt.

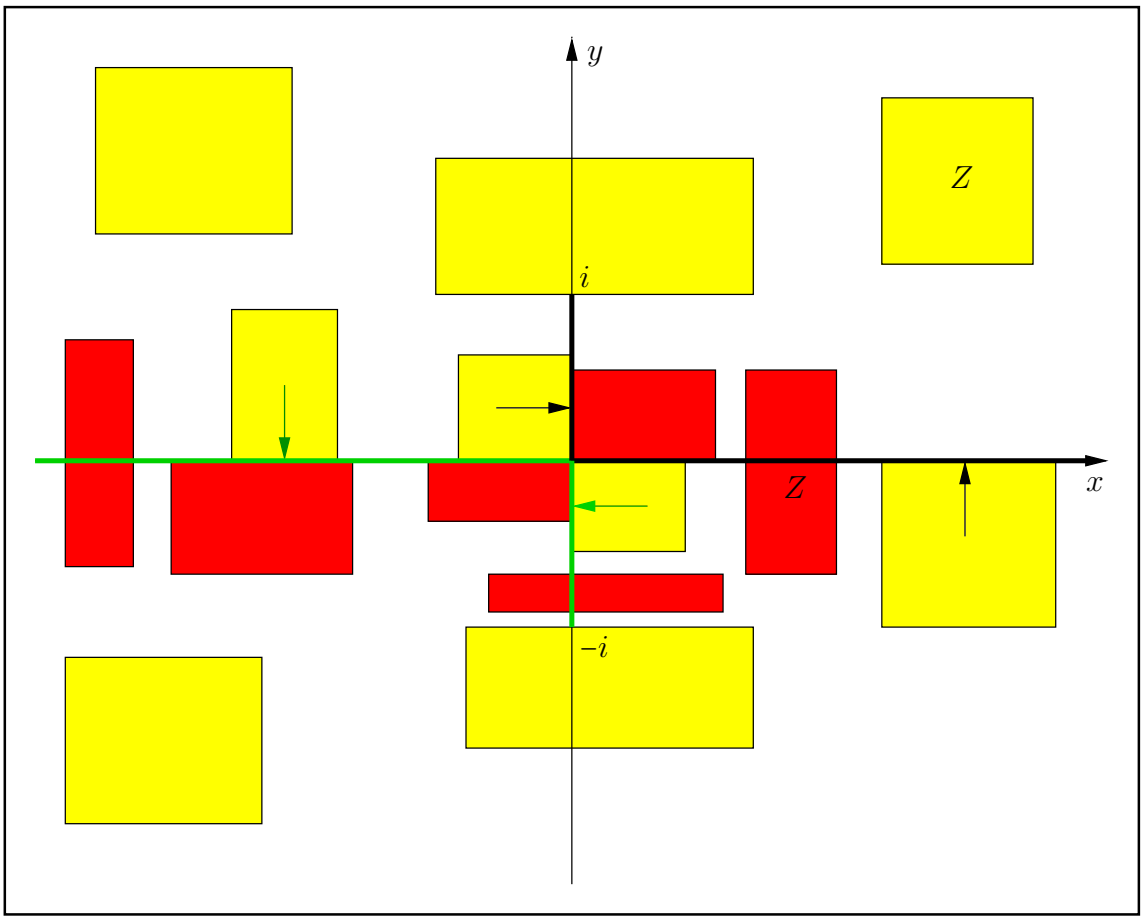


Abbildung C.33.: Erlaubte und nicht erlaubte Intervalle Z für den Hauptzweig von $f(z)$

Beachten Sie, dass auch jedes Intervall der Form $Z = [0,0] + i \cdot Y$ oder $Z = X + i \cdot [0,0]$, d.h. jedes Intervall Z, das auf einer Koordinatenachse liegt, ein erlaubtes Intervall ist. Die schwarzen und die grünen Pfeile geben die Richtungen an, aus denen die Funktionswerte von $f(z) = \sqrt{-z^2-1}$, mit $z \in \mathbb{C}$, auf die jeweiligen Verzweigungsschnitte analytisch fortgesetzt werden.

Die **optimale** Einschließung des Wertebereichs W durch das komplexe Rechteck $F(Z) \supseteq W$ erfolgt mithilfe der Funktion

```
MpfciClass sqrtmx2m1(const MpfciClass& Z);
```

die in der Datei `pmfciclass.cpp` definiert ist.

C.2.17.1. Numerische Ergebnisse

Im **1. Beispiel** wählen wir das Intervall $Z = [2,3] + i \cdot [1,2]$. Mit `prec` = 400 erhalten wir für den Wertebereich W von $f(z)$ für alle $z \in Z$ die nahezu optimale Rechteckeinschließung $F(Z)$ mit 120 korrekten Dezimalstellen:

$$W \subseteq F(Z) = ([0.91017972112445468260871 5...9580, 1.92566973609167184450654...2311] \\ [-3.14774949975311471570686...0773, -2.12864484453120427681 38...1189]).$$

Im **2. Beispiel** wählen wir das gleiche Intervall $Z = [2,3] + i \cdot [1,2]$ und berechnen jetzt aber eine Einschließung $F_1(Z)$ des Wertebereichs W mit dem naiven Ansatz

```
MpfciClass z(interval(2,3),interval(1,2)), res;      res = sqrt(-sqr(z)-1);
```

und erhalten die viel gröbere Einschließung

$$W \subseteq F_1(Z) = ([0.65148208025858421217529 1...2081, 2.34963769321913708102402...5287] \\ [-3.46410161513775458705 48...0607, -1.60048518044024083831781...4802]).$$

Es gilt also:

$$W \subseteq F(Z) \subset F_1(Z).$$

Wird ein mit holomorphen Funktionen verschachtelter, komplexer Intervallausdruck, in dem die Intervallvariable Z nur einmal vorkommt, naiv ausgewertet, so erhält man nach obigem Beispiel mit $F_1(Z)$ i.a. keine optimale Einschließung des Wertebereichs W.

Wertet man im Gegensatz dazu einen **reellen** Intervallausdruck aus, in dem die Intervallvariable x nur einmal vorkommt und der aus nur stetigen Funktionen zusammengesetzt ist, so erhält man stets optimale Einschließungen für den entsprechenden Wertebereich W. So erhält man z.B für den Ausdruck $\sqrt{x^2+1}$ mit `sqrt(sqr(x)` $\oplus$ `1)` eine optimale Einschließung, während man mit `sqrt(x` $\diamond$ `x` $\oplus$ `1)` i.a. eine starke Überschätzung erhält, weil im letzten Ausdruck die Variable `x` zweimal vorkommt und daher gilt: `sqr(x)` $\subseteq$ `x` $\diamond$ `x`, vgl. auch Seite 74.

> Um den Wertebereich W eines aus holomorphen Funktionen zusammengesetzten Intervallausdrucks möglichst optimal einzuschließen, sollte man auf möglichst viele Teilfunktionen zurückgreifen können, deren Wertebereiche selbst jeweils optimal eingeschlossen werden, vgl. dazu die Tabelle auf Seite 115.

Die Einschließung des Wertebereichs W_f von z.B. $f(z) := \sqrt{-z^2-1} \cdot \log(1+z)$ wird man daher realisieren durch

```
MpfciClass z(...), F;      F = sqrtmx2m1(z) ⋄ lnp1(z);
```

Beachten Sie aber, dass die Einschließung von W_f durch `F` noch keinesfalls optimal sein muss, da uns nach der Tabelle auf Seite 115 nur optimale Einschließungen der beiden Teilausdrücke $\sqrt{-z^2-1}$ und $\log(1+z)$ zur Verfügung stehen, nicht aber eine optimale Einschließung das ganzen Produkts $f(z) = \sqrt{-z^2-1} \cdot \log(1+z)$. Da in der Realität nur wenige Intervallausdrücke wirklich optimal eingeschlossen werden können, kommen wir zum Ergebnis:

> Die Einschließung des Wertebereichs W eines komplexen Intervallausdrucks ist i.a. ein eigenständiges Problem, das nur in Ausnahmefällen optimal gelöst werden kann, vgl. dazu auch Seite 120.

C.2.18. $1/\sqrt{z}$

Der Verzweigungsschnitt von $f(z) = 1/\sqrt{z},\ z = x + i \cdot y \in \mathbb{C}\backslash\{0\}$, ist in der komplexen Ebene die negative reelle Achse. Ein achsenparalleles Argumentintervall Z darf diesen Verzweigungsschnitt nicht in seinem Innern enthalten und auch nicht von unten berühren. Die in `mpfciclass.cpp` definierte Funktion

```
MpfciClass sqrt_r(const MpfciClass& Z)
```

liefert mit `W = sqrt_r(Z)` eine nahezu optimale, achsenparallele Rechteck-Einschließung der komplexen Wertemenge

$$\{w \in \mathbb{C} \,|\, w = 1/\sqrt{z},\ z \in Z\} \subseteq \mathtt{W},$$

wobei $\sqrt{z}$ der Hauptwert der komplexen Quadratwurzel ist.

In Abbildung C.34 sind einige erlaubte und nicht erlaubte achsenparallele Argumentintervalle Z der Funktion `sqrt_r(Z)` angegeben.

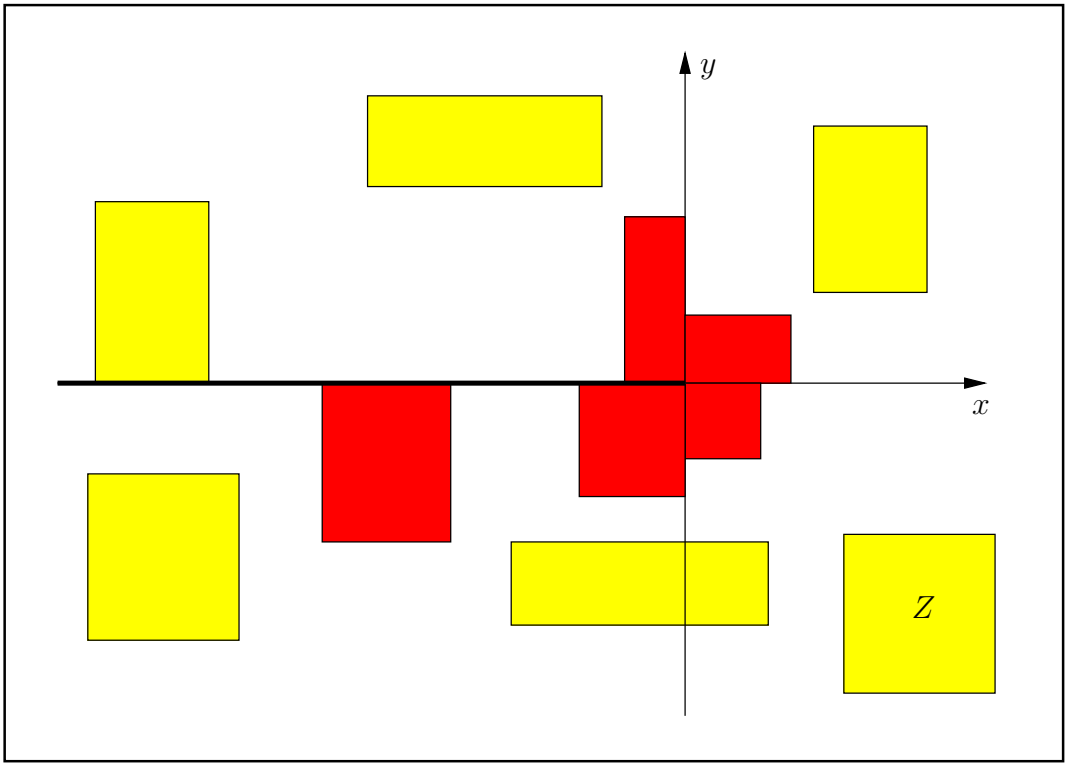


Abbildung C.34.: Erlaubte und nicht erlaubte Intervalle Z von `sqrt_r(Z)`.

C.2.18.1. Realteil

Der Realteil von $f(z) = u(x,y) + i \cdot v(x,y)$ ist nach (C.19) von Seite 258 gegeben durch

$$u(x,y) := \frac{\sqrt{|z| + x}}{\sqrt{2} \cdot |z|}, \quad |z| = \sqrt{x^2 + y^2} \neq 0.$$

Für ein erlaubtes Intervall $Z = X + i \cdot Y$ liegen dann die Maximum- und Minimumpunkte M, m der harmonischen Funktion $u(x,y)$ auf dem Rand von $Z(X,Y)$, und wegen $u(x,-y) \equiv u(x,y)$ kann man sich bei der Berechnung von M, m auf die obere Halbebene beschränken, d.h. $u(x,y)$ nimmt auf den Rechtecken $Z(X,Y)$ und $Z^*(X, \mathtt{abs}(Y))$ die gleichen Funktionswerte an, wobei $\mathtt{abs}(Y)$ das Intervall der Absolutbeträge von Y ist. Zur Vereinfachung werden wir in diesem Teilabschnitt das Rechteck $Z^*(X, \mathtt{abs}(Y))$ aus der oberen Halbebene wieder mit Z bezeichnen.

Zur Berechnung der Extrema m, M benötigen wir von $u(x,y)$ die partiellen Ableitungen, die mit $\alpha(x,y) > 0$ gegeben sind durch

$$\frac{\partial u(x,y)}{\partial x} = (y^2 - x \cdot (x + \sqrt{x^2+y^2})) \cdot \alpha,$$
$$\frac{\partial u(x,y)}{\partial y} = -y \cdot (2x + \sqrt{x^2+y^2})) \cdot \alpha.$$

Die Extremalkurven bez. $\partial u(x,y)/\partial x = 0$ ergeben sich aus: $y^2 - x \cdot (x + \sqrt{x^2+y^2}) = 0$ zu:

1. $y = 0$ für $x < 0$, d.h. auf der negativen reellen Achse gilt $u(x,y) \equiv 0$,
2. $y = \sqrt{3} \cdot x$ für $x > 0$, d.h. die Extremalkurve liegt nur im 1. Quadranten.

Die Extremalkurven bez. $\partial u(x,y)/\partial y = 0$ ergeben sich aus: $y \cdot (2x + \sqrt{x^2+y^2})) = 0$ zu:

1. $y = 0$ für $x > 0$, d.h. die Extremalkurve ist die positiven reellen Achse,
2. $y = -\sqrt{3} \cdot x$ für $x < 0$, d.h. die Extremalkurve liegt nur im 2. Quadranten.

In folgender Abbildung ist in der oberen Halbebene das Monotonieverhalten von $u(x,y)$ angegeben, und für typische Lagen von Z sind die Punkte m, M der gesuchten Extrema von $u(x,y)$ auf dem Rand von Z dargestellt.

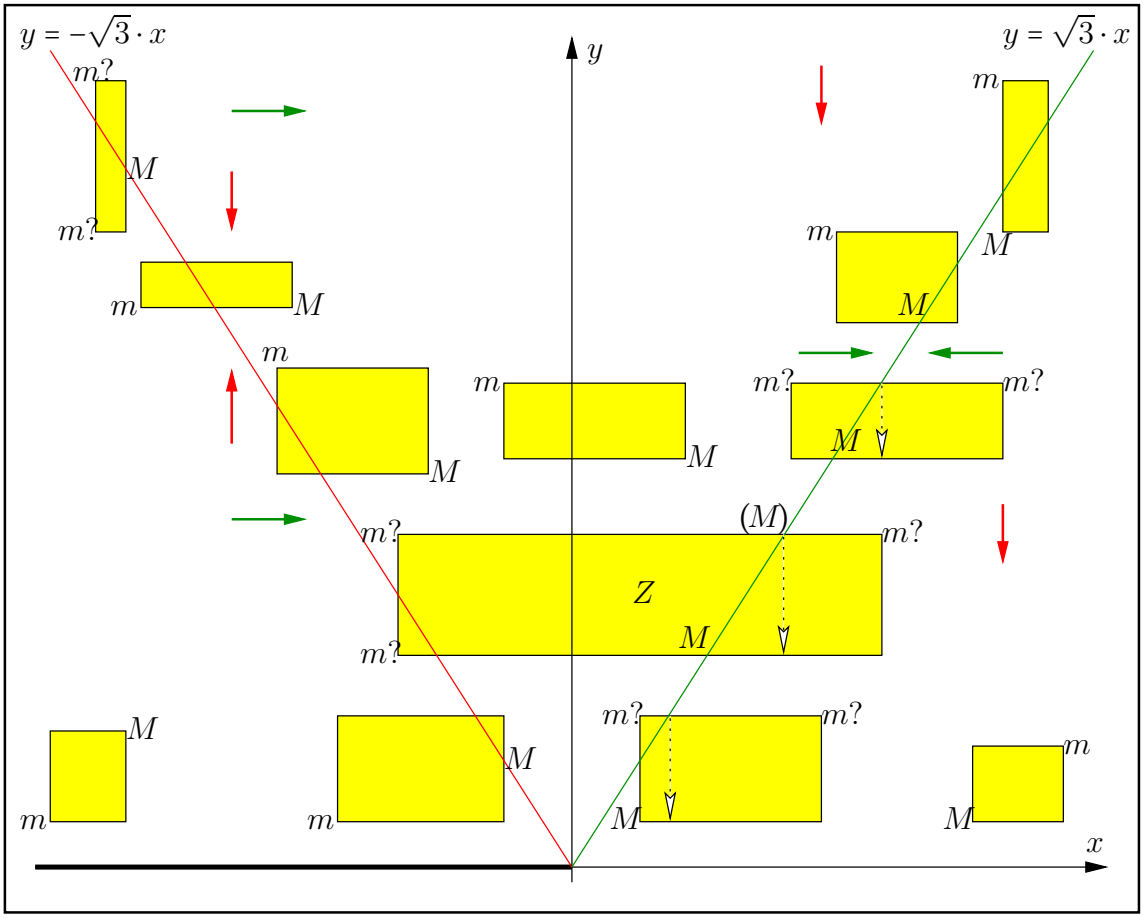


Abbildung C.35.: Typische Lagen von Z mit den Punkten m, M bez. $u(x,y)$

Im Rechteck Z kann (M) nicht der Maximumpunkt sein, denn längs des punktierten Pfeils kann man nach unten und anschließend bis M nur in Richtung wachsender Funktionswerte von $u(x,y)$ laufen. Analog argumentiert man bei den zwei anderen Rechtecken mit den punktierten Pfeilen.

Mit Abb. C.35 und mit den Abkürzungen:

`ULP`: Unterer linker Eckpunkt von Z; `URP`: Unterer rechter Eckpunkt von Z;
`OLP`: Oberer linker Eckpunkt von Z; `ORP`: Oberer rechter Eckpunkt von Z;

ergibt sich der folgende Grobalgorithmus zur Bestimmung der Punkte m, M für das Minimum bzw. Maximum von $u(x, y)$ über dem Rand von Z:

```
if (ULP unterhalb von y=sqrt(3)*x)
  M = ULP;   m = ORP oder m = OLP;
else
  if (Untere Parallele schneidet y=sqrt(3)*x)
    M ist dieser Schnittpunkt;  m = ULP oder m = OLP oder m = ORP;
  else
    if (URP liegt rechts von y=-sqrt(3)*x)
      M = LRP;  m = OLP oder m = ULP;
    else
      if (Rechte Parallele schneidet y=-sqrt(3)*x)
        M ist dieser Schnittpunkt;   m = ULP oder m = OLP;
      else
        M = ORP;  m = ULP;
```

Anmerkungen:

1. Weitere Einzelheiten findet man mit dem komplexen Intervall `z` = `x` + $i \cdot$ `y` in der Funktion

   ```
   MpfiClass sqrt_rRe(const MpfiClass& x, const MpfiClass& y)
   ```

   die in `mpficlass.cpp` definiert ist.

2. Der maximale Funktionswert `r` wird z.B. im unteren linken Eckpunkt von z berechnet mit dem Aufruf: `r = sqrt_rRe(x1, y1, RoundUp);`

3. Schneidet die rechte Parallel vom komplexen Rechteckintervall z die Extremalkurve $y = -\sqrt{3} \cdot x$ im Punkt $S(x_2, y_S)$, so wird mit $y_S = -\sqrt{3} \cdot x_2$ der maximale Funktionswert `r` bei geeigneter Rundung berechnet durch: `r` = $\sqrt{2} \cdot (1/\sqrt{-x_2})/4$.

C.2.18.2. Imaginärteil

Der Imaginärteil von $f(z) = u(x,y) + i \cdot v(x,y)$ ist für $|z| = \sqrt{x^2+y^2} \neq 0$ nach (C.19) von Seite 258 gegeben durch

$$v(x,y) := \begin{cases} \dfrac{-1}{\sqrt{-x}}, & y = 0 \wedge x < 0, \\ \dfrac{-y}{\sqrt{2} \cdot |z| \cdot \sqrt{|z|+x}}, & \text{sonst, d.h. falls } y \neq 0 \vee x \geq 0. \end{cases}$$

Wegen der für $x < 0$ gültigen Grenzwerte

$$\lim_{y \to 0^+} v(x,y) = -1/\sqrt{-x} \qquad \text{und} \qquad \lim_{y \to 0^-} v(x,y) = +1/\sqrt{-x}$$

besitzt $v(x,y)$ auf der negativen reellen Achse eine Sprungstelle und im Ursprung eine Singularität. Für ein erlaubtes Intervall $Z = X + i \cdot Y$ liegen die Maximum- und Minimumpunkte M, m der harmonische Funktion $v(x,y)$ auf dem Rand von $Z(X,Y)$, und wegen $v(x,-y) \equiv -v(x,y)$ kann man sich bei der Berechnung der Funktionswerte $l = v(m)$ und $r = v(M)$ auf die obere Halbebene beschränken. Mit $X = [x_1, x_2]$, $Y = [y_1, y_2]$ erfolgt dies mit folgendem Algorithmus:

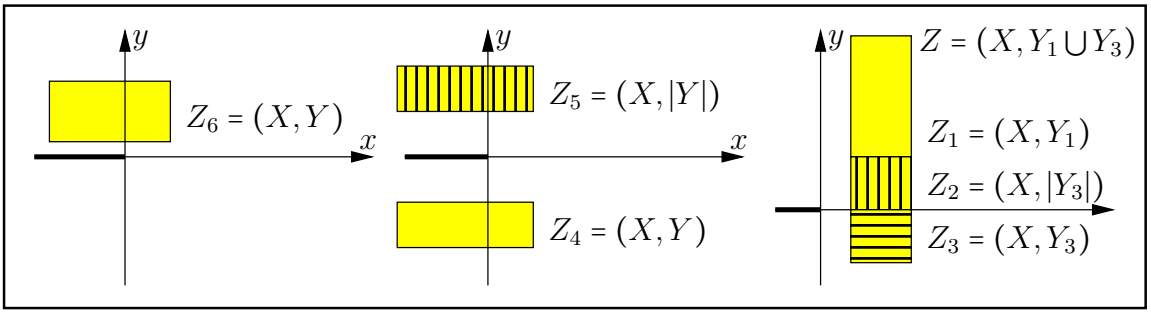


Abbildung C.36.: Zur Bestimmung von l, r bei Auswertung in der oberen Halbebene

Algorithmus zur Berechnung der Extrema l, r nur in der oberen Halbebene:

```
if (y1<0)
  if (x1>0 && y2>0)
    l in Z1 berechnen;  // l ist Minimum in Z;
    l2 in Z2 = (X,|Y3|) berechnen;
    r = -l2;            // r ist Maximum in Z;
  else  // x1<=0 oder y2<=0:
    l in Z5 berechnen;
    r in Z5 berechnen;
    l4 = -r;  // l4: Minimum in Z4
    r4 = -l;  // r4: Maximum in Z4
else  // y1>=0
  l in Z6 berechnen;
  r in Z6 berechnen;
```

Wir benötigen daher die folgenden Funktionen:

1. Zur Berechnung von l, wenn Z_1 oder Z_2 im 1. Quadranten liegen:
   `MpfrClass sqrt_rIm1Q(const MpfiClass& x, const MpfiClass& y)`
2. Zur Berechnung von l, r, wenn Z_5 oder Z_6 in der oberen Halbebene liegen:
   `MpfiClass sqrt_rImOH(const MpfiClass& x, const MpfiClass& y)`

Um in der oberen Halbebene die Extrema $l = v(m)$ und $r = v(M)$ nach obigem Algorithmus berechnen zu können, benötigen wir von $v(x,y)$ noch die partiellen Ableitungen, die für $y \geq 0$ mit den positiven Faktoren $\beta(x,y) > 0$ und $\delta(x,y) > 0$ gegeben sind durch:

$$\frac{\partial v(x,y)}{\partial x} = y \cdot \left(y^2 + 3x \cdot (x + \sqrt{x^2+y^2})\right) \cdot \beta(x,y),$$
$$\frac{\partial v(x,y)}{\partial y} = (-2x + \sqrt{x^2+y^2}) \cdot \sqrt{x + \sqrt{x^2+y^2}} \cdot \delta(x,y).$$

Die Extremalkurven bez. $\partial v(x,y)/\partial x = 0$ ergeben sich aus: $y\left(y^2 + 3x \cdot (x + \sqrt{x^2+y^2})\right) = 0$ zu:

1. $y = 0$ für $x > 0$, d.h. auf der positiven reellen Achse gilt: $v(x,y) \equiv 0$.
2. $y = -\sqrt{3} \cdot x$ für $x < 0$, d.h. die Extremalkurve liegt nur im 2. Quadranten.

Die Extremalkurven bez. $\partial v(x,y)/\partial y = 0$ ergeben sich aus: $(-2x + \sqrt{x^2+y^2}) \cdot \sqrt{x + \sqrt{x^2+y^2}} = 0$ zu:

1. $y = 0$ für $x < 0$, d.h. auf der negativen reellen Achse gilt: $\partial v(x,y)/\partial y = 0$.
2. $y = \sqrt{3} \cdot x$ für $x > 0$, d.h. die Extremalkurve liegt nur im 1. Quadranten.

Um den Minimumpunkt m für ein Rechteck $Z = (X,Y)$ aus dem 1. Quadranten mit Hilfe der Funktion `MpfrClass sqrt_rIm1Q(const MpfiClass& x, const MpfiClass& y)` bestimmen zu können, betrachten wir in folgender Abbildung typische Lagen dieser Rechtecke $Z = (X,Y)$:

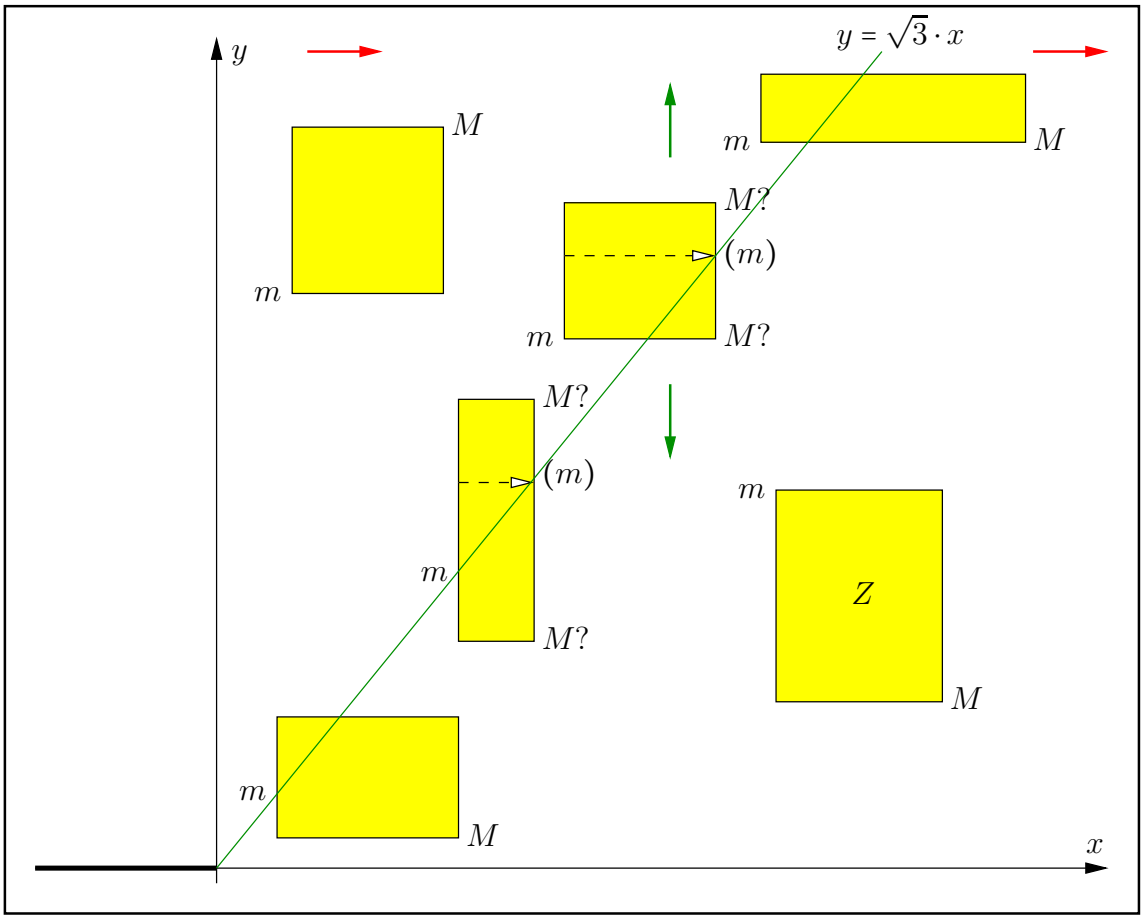


Abbildung C.37.: Zur Bestimmung von m, M bei Auswertung von $v(x,y)$ im 1. Quadranten

Die roten und grünen Pfeile geben die Richtung wachsender Funktionswerte an. An den Punkten (m) kann kein Minimum angenommen werden und bei $M?$ ist jeweils ein Maximum möglich.

Der Minimumpunkt m des Imaginärteils $v(x,y)$ über einem Rechteck Z aus dem 1. Quadranten wird mit folgendem Algorithmus berechnet:

```
if (ULP liegt links von y = sqrt(3)*x)
  m = ULP;
else
  if (Linke Parallele schneidet y = sqrt(3)*x im Punkt S)
    m = S;
  else
    m = OLP;  // m = Oberer linker Eckpunkt von Z;
```

Um die Punkte m, M der Extrema für ein Rechteck $Z = (X,Y)$ aus der oberen Halbebene mit Hilfe der Funktion `MpfiClass sqrt_rImOH( const MpfiClass& x, const MpfiClass& y )` bestimmen zu können, betrachten wir in folgender Abbildung typische Lagen dieser Rechtecke $Z = (X,Y)$:

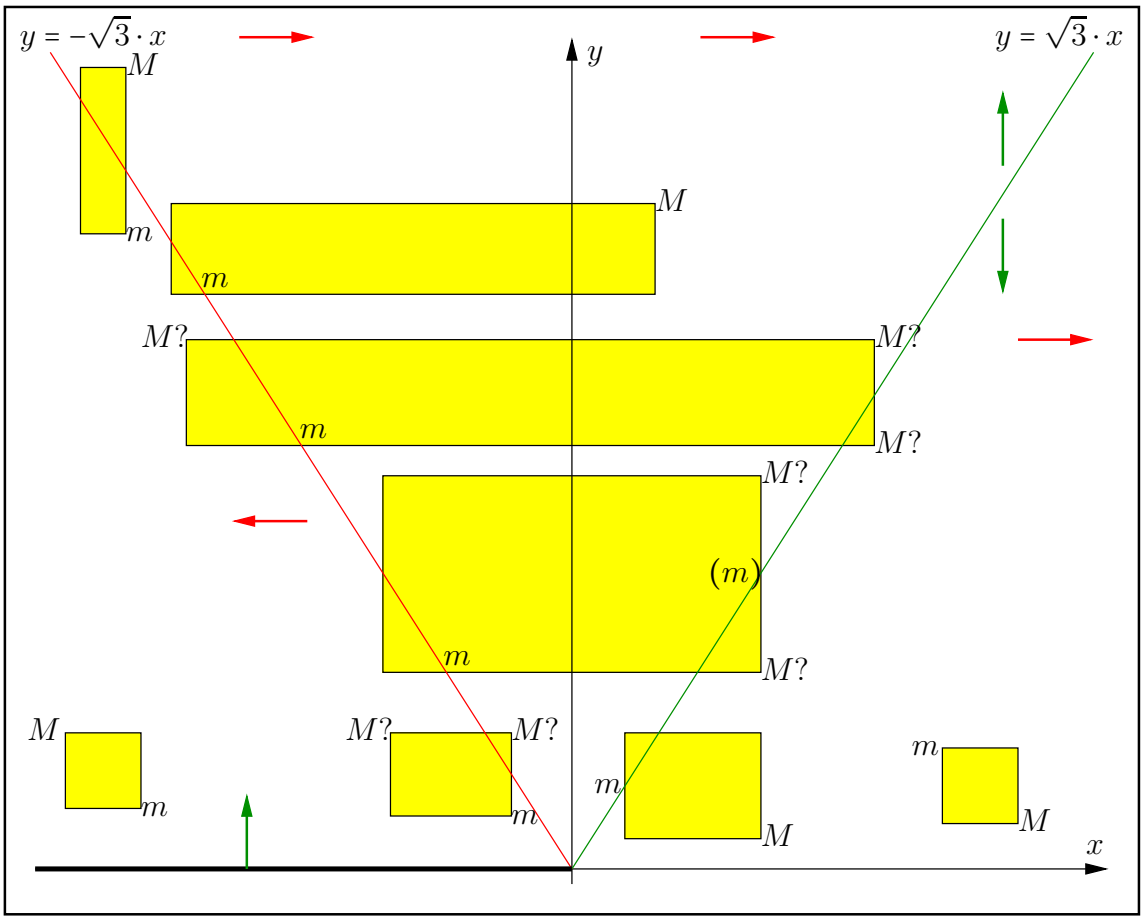


Abbildung C.38.: Lage von m, M bei Auswertung von $v(x,y)$ in der oberen Halbebene

Die Lage der Extremalpunkte m, M für Rechtecke $Z = (X,Y)$, die mit ihren unteren Parallelen die Extremalkurve $y = -\sqrt{3}\cdot x$ nicht schneiden und rechts von dieser Geraden liegen, findet man in Abb. C.37. Mit den Abbildungen C.37, C.38 und mit den Abkürzungen:

`ULP`: Unterer linker Eckpunkt von Z; `URP`: Unterer rechter Eckpunkt von Z;
`OLP`: Oberer linker Eckpunkt von Z; `ORP`: Oberer rechter Eckpunkt von Z;

ergibt sich der folgende Grobalgorithmus zur Bestimmung der Punkte m, M für das Minimum bzw. Maximum von $v(x,y)$ über dem Rand von Z, wenn Z in der oberen Halbebene liegt:

```
if (URP liegt unterhalb von y = -sqrt(3)*x)
  m = URP;  M = ORP oder M = OLP;
else
  if (Untere Parallele schneidet y = -sqrt(3)*x im Punkt S)
    m = S;  M = OLP oder M = ORP oder M = URP;
  else  // Z liegt rechts von y = -sqrt(3)*x;
    if (ULP liegt links von y = +sqrt(3)*x)
      m = ULP;  M = ORP oder M = URP;
    else
      if (Linke Parallele schneidet y = +sqrt(3)*x im Punkt S)
        m = S;  M = ORP oder M = URP;
      else
        m = OLP;  M = URP;
```

Der obige Algorithmus wird realisiert mit der Funktion

```
MpfiClass sqrt_rImOH(const MpfiClass& x, const MpfiClass& y)
```

wobei das Intervall `y` in der oberen Halbebene liegen muss.

C.2.18.3. Numerische Beispiele

Im **1. Beispiel** wählen wir mit $Z = [-1/2, -2^{-5000}] + i \cdot [0,1]$ ein Argumentintervall in der Nähe der Singularität $z_0 = 0$ und erhalten bei einer Präzision von `prec` = 700 mit dem Funktionsaufruf `W = sqrt_r(Z)` für den Wertebereich $\{w \in \mathbb{C} \,|\, w = 1/\sqrt{z},\ z \in Z\} \subseteq$ `W` mit 210 korrekten Dezimalstellen die nahezu optimale Einschließung

$$\texttt{W} = ([0, 1.328752\ldots574 \cdot 10^{752}], [-3.758280\ldots802 \cdot 10^{752}, -7.071067\ldots853 \cdot 10^{-1}]).$$

Zum Vergleich erhalten wir bei der Intervallauswertung von $\texttt{W}_I = 1 \diamond$ `sqrt(Z)` die viel gröbere Einschließung

$$\texttt{W}_I = ([0, 1.879140\ldots901 \cdot 10^{752}], [-3.758280\ldots802 \cdot 10^{752}, -5.321582\ldots764 \cdot 10^{-753}]),$$

wobei die Oberschranke des Imaginärteils um 752 Zehnerpotenzen zu groß ausfällt.

Im **2. Beispiel** wählen wir $Z = [1/4, 1/2] + i \cdot [-2, +1]$ und erhalten mit `prec` = 700 und mit dem Funktionsaufruf `W = sqrt_r(Z)` für den Wertebereich $\{w \in \mathbb{C} \,|\, w = 1/\sqrt{z},\ z \in Z\} \subseteq$ `W` mit 210 korrekten Dezimalstellen die nahezu optimale Einschließung

$$\texttt{W} = ([5.280517\ldots552 \cdot 10^{-1}, 2.000000\ldots000], [-7.071067\ldots861 \cdot 10^{-1}, 7.071067\ldots861 \cdot 10^{-1}]).$$

Zum Vergleich erhalten wir bei der Intervallauswertung von $\texttt{W}_I = 1 \diamond$ `sqrt(Z)` auch jetzt die viel gröbere Einschließung

$$\texttt{W}_I = ([4.4139110926\ldots396 \cdot 10^{-1}, 2.00000000\ldots000], [-1.0000000\ldots000, +1.0000000\ldots000]),$$

wobei aber die Größenordnungen der Intervallgrenzen in diesem Beispiel übereinstimmen.

C.2.19. $\cot(z)$

Mit $z = x + i \cdot y$ gilt nach Abramowitz (4.3.58)

$$\cot(z) = \frac{\sin(2x) - i \cdot \sinh(2y)}{\cosh(2y) - \cos(2x)} = u(x,y) + i \cdot v(x,y) \tag{C.69}$$

$$= \frac{\cos(z)}{\sin(z)}, \quad z \neq k\pi,\ k \in \mathbb{Z} \tag{C.70}$$

$$= \frac{1}{\tan(z)} \tag{C.71}$$

$$= \tan(\pi/2 - z) = \tan(\pi/2 - z + k\pi), \quad k \in \mathbb{Z} \tag{C.72}$$

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 (C.71) ist dabei ungeeignet, denn im Falle $\pi/2 \in Z$ liegt eine Polstelle des Tangens in Z, so dass mit (C.71) Programmabbruch erfolgt, obwohl der Cotangens in der Umgebung von $\pi/2$ existiert und bei $z_{s,0} = \pi/2$ eine Nullstelle besitzt. Die Darstellung (C.70) ist ebenfalls ungeeignet, da zwei Ergebnisintervalle zu dividieren sind, womit eine erhebliche, zusätzliche Überschätzung verbunden ist. Zur Implementierung des Cotangens wählen wir daher (C.72). 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.

C.2.20. $\arcsin(z)$

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.

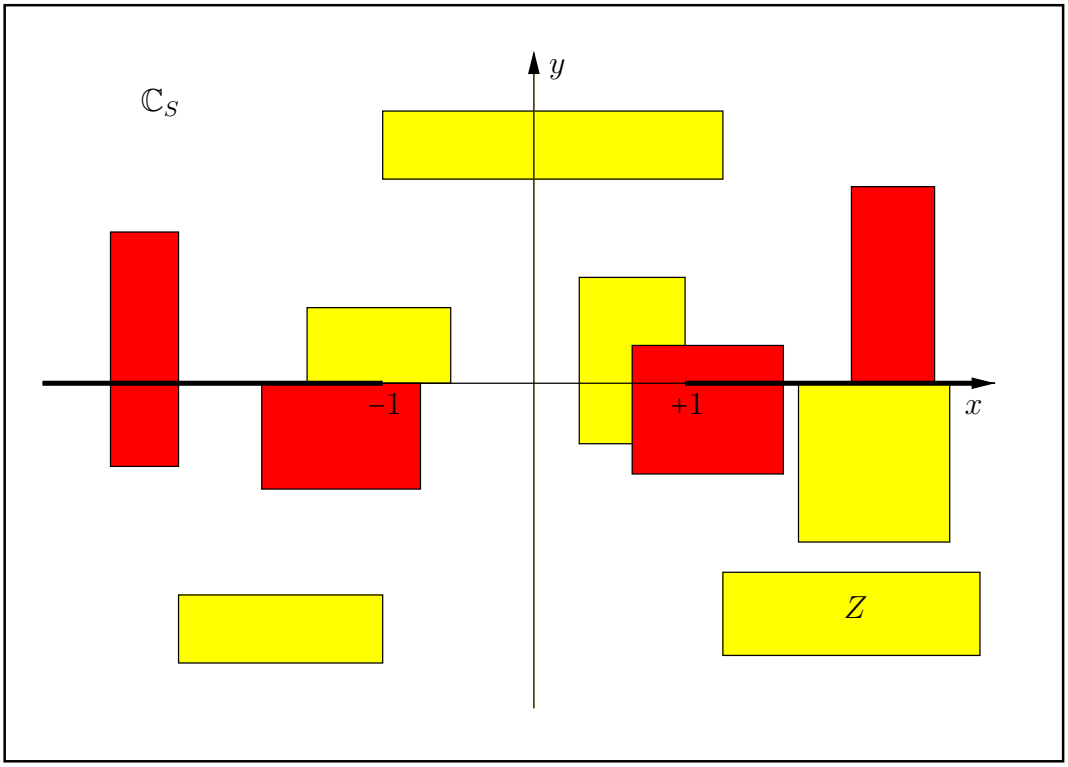


Abbildung C.39.: Erlaubte und nicht erlaubte Intervalle Z von `asin(Z)`.

Mit $Z = [1,1] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[8]

$$\begin{aligned}\texttt{asin(Z)} \;=\; &([1.57079632679489661922, 1.57079632679489661924],\\ &[4.88115243040816240520e-161614249, 4.88115243040816240522e-161614249]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{asin(Z)} \;=\; &([3.49439062857213293627411825e-1, 1.57079632679489661923132170],\\ &[-1.26047518779845407285290814, 0.00000000000000000000000000]).\end{aligned}$$

[8] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

C.2.20.1. Algorithmus

Der nachfolgende Algorithmus zur Berechnung der Rechteck-Einschließung `asin(Z)` für ein vorgegebenes achsenparalleles Rechteckintervall Z basiert auf [55], [54], [12], wobei sich die Verbesserungen in [12] 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, [38]:

$$\alpha := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\} \tag{C.73}$$

$$\beta := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} - \sqrt{(x-1)^2 + y^2} \right\} \tag{C.74}$$

$$\beta = \frac{x}{\frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\}} = \frac{x}{\alpha} \tag{C.75}$$

$$\Re(w) := \arcsin(\beta) \tag{C.76}$$

$$\Im(w) = \begin{cases} +\operatorname{arcosh}(\alpha), & \text{falls } y > 0 \\ +\operatorname{arcosh}(\alpha), & \text{falls } y = 0 \text{ und } x \le -1 \\ 0 \quad , & \text{falls } y = 0 \text{ und } -1 \le x \le +1 \\ -\operatorname{arcosh}(\alpha), & \text{falls } y = 0 \text{ und } x \ge +1 \\ -\operatorname{arcosh}(\alpha), & \text{falls } y < 0 \end{cases} \tag{C.77}$$

Wegen $\alpha(x,y) \ge \alpha(x,0) = (|x+1| + |x-1|)/2$ erhält man mit einfachen Fallunterscheidungen ($|x| \le 1,\ x < -1,\ x > +1$):

$$\alpha(x,y) \ge \max(1,|x|) = \begin{cases} 1, & \text{wenn } |x| \le 1, \\ |x|, & \text{wenn } |x| > 1, \end{cases} \qquad \text{oder gröber: } \alpha(x,y) \ge 1. \tag{C.78}$$

Unter den Voraussetzungen $y = 0$ und $-1 \le x \le +1$ gilt:

$$\alpha(x,0) \equiv 1, \quad \text{d.h. } \Im(w) = \pm\operatorname{arcosh}(1) = 0, \quad \Re(w) = \arcsin(x); \tag{C.79}$$

Es gilt außerdem

$$|\beta(x,y)| \le |\beta(x,0)| \le 1$$

Zum Beweis gilt nach (C.75):

$$|\beta(x,y)| \le |\beta(x,0)| = \frac{2 \cdot |x|}{|x+1| + |x-1|} =: R(x),$$

und wegen $R(-x) \equiv R(x)$ kann man sich auf $x \ge 0$ beschränken:

Sei $0 \le x \le 1$, d.h. $|x-1| = -(x-1) \ \rightsquigarrow\ R(x) = \frac{2x}{(x+1)-(x-1)} = x \le 1$;

Sei $x > 1$, d.h. $|x-1| = +(x-1) \ \rightsquigarrow\ R(x) = \frac{2x}{(x+1)+(x-1)} = 1$ ∎

In [54] 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 (C.73) und (C.75) 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 (C.73) und (C.75) ist folgender Term $T(x,y)$ zu berechnen:

$$T := \sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2} \tag{C.80}$$

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[9]:

$$2\sqrt{2}\cdot(M+1) < \texttt{MaxFloat(2)}$$

$$\Longleftrightarrow \qquad M < \frac{\texttt{MaxFloat(2)}}{2\sqrt{2}} - 1 = 0.530330085889\ldots\cdot 2^{1073741822}$$

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 (C.78) überprüft, die durch Intervallüberschätzungen verletzt sein kann.

Eine nahezu optimale Einschließung des Realteils $\Re(w)$ nach (C.76) durch die Auswertung von $\arcsin(\boldsymbol{\beta})$ scheint damit gewährleistet zu sein, wenn man zusätzlich im entsprechenden Algorithmus aus [54] (`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:

$$\arcsin(\beta) = \arcsin(1-\delta) = \frac{\pi}{2} - \arcsin\left(\sqrt{\delta\cdot(2-\delta)}\right), \quad 0 \leq \delta \leq 2; \tag{C.81}$$

Wertet man jetzt in Gleichung (C.81) 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 (C.147), (C.137) und (C.114). 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

[9] 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 (C.77) $\operatorname{arcosh}(\boldsymbol{\alpha})$ für das Intervallargument $\boldsymbol{\alpha}$ auszuwerten ist. Nach (C.73) 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

$$\operatorname{arcosh}(\alpha) \equiv \operatorname{arcosh}(1+\delta) \tag{C.82}$$

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 $z = x + i \cdot y$:

$$|x| \geq 2 \ \vee\ |y| \geq 2 \quad \Longrightarrow \quad \alpha(x,y) \geq 2; \tag{C.83}$$

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 (C.83) $\operatorname{arcosh}(\alpha)$ 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

$$x = \pm 1 \quad \rightsquigarrow \quad \delta = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2}, \tag{C.84}$$

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:

$$\delta = \frac{V(x,y)}{2} \quad \text{, falls } |x| < 1 \tag{C.85}$$

$$\delta = \frac{V(x,y)}{2} + (|x| - 1) \quad \text{, falls } |x| > 1 \tag{C.86}$$

$$\delta = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2} \quad \text{, falls } |x| = 1, \tag{C.87}$$

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 (C.82) mit Hilfe der Ausdrücke für δ nach (C.85) und (C.87) können im Falle $\delta \ll 1$ noch wesentlich vereinfacht werden. Dabei sind für x, y Punktintervalle einzusetzen, so dass in (C.85) und (C.87) 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[10]:

$$\operatorname{arcosh}(1+t) = \sqrt{2t} \cdot Q(t); \tag{C.88}$$

$$Q(t) := \sum_{k=0}^{\infty} a_k \cdot t^k = 1 - \frac{1}{12}t + \frac{3}{160}t^2 - \frac{5}{896}t^3 + \frac{35}{18432}t^4 - \frac{63}{90112}t^5 \pm \ldots$$

$$a_k = \frac{1}{(2k+1)\cdot 2^k} \cdot \binom{-0.5}{k}, \quad a_0 = 1,$$

$$a_{k+1} = -a_k \cdot \frac{(2k+1)^2}{4 \cdot (2k+3)(k+1)},\ k = 0, 1, 2, \ldots$$

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:

$$\operatorname{arcosh}(\sqrt{1+(t/2)^2} + t/2) = \sqrt{t} \cdot H(t), \quad H(t) = \sum_{k=0}^{\infty} b_k,$$

$$H(t) = 1 + \frac{1}{12}t - \frac{3}{160}t^2 - \frac{5}{896}t^3 + \frac{35}{18432}t^4 + \frac{63}{90112}t^5 - - + + \ldots,$$

[10] 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[11] $|a_k| = |b_k|$. Unter dieser Voraussetzung ist dann auch $H(t)$ absolut konvergent, so dass wie folgt geklammert werden kann:

$$\begin{aligned} H(t) &= (1+\frac{1}{12}t) - (\frac{3}{160}t^2 + \frac{5}{896}t^3) + (\frac{35}{18432}t^4 + \frac{63}{90112}t^5) - + \ldots \\ &= h_1 - h_2 + h_3 - h_4 \pm \ldots, \quad h_j \geq 0; \end{aligned} \tag{C.89}$$

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 (C.89) 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:

$$\sqrt{t} \cdot \left[\left(1 + \frac{t}{12}\right) - t^2 \left(\frac{3}{160} + \frac{5t}{896}\right) \right] \leq \operatorname{arcosh}(\alpha), \quad t = |y| < 1; \tag{C.90}$$

$$\operatorname{arcosh}(\alpha) = \operatorname{arcosh}\left(\sqrt{1+\left(\frac{t}{2}\right)^2} + \frac{t}{2}\right) \leq \sqrt{t} \cdot \left(1 + \frac{t}{12}\right), \; t = |y| < 1; \tag{C.91}$$

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 (C.90) und (C.91) erhalten wir schließlich im Fall $|x| = 1$:

$$\sqrt{t} \cdot \left(1 - \frac{3}{80}t^2\right) \leq \operatorname{arcosh}(\alpha) \leq \sqrt{t} \cdot \left(1 + \frac{t}{12}\right), \qquad t = |y| < 1; \tag{C.92}$$

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 (C.85) 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 $\mathfrak{I}(\boldsymbol{w})$ von $\boldsymbol{w} = \arcsin(\boldsymbol{z})$ ist dann nach (C.82) 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

$$\sqrt{1+t} - 1 = t \cdot \sum_{k=0}^{\infty} (-1)^{k+1} \cdot \left(\prod_{n=0}^{k} \frac{2n-1}{2n+2} \right) \cdot t^k \tag{C.93}$$

$$= \frac{t}{2} \cdot \left(1 - \frac{t}{4} + \frac{t^2}{8} - \frac{5 \cdot t^3}{64} \pm \ldots\right), \quad |t| < 1; \tag{C.94}$$

[11] $|a_k| = |b_k|$ 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 vorzeitigen 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 (C.94) für $0 \le t < 1$ eine alternierende Leibniz-Reihe ist, gilt die Abschätzung[12]:

$$\frac{t}{2} \cdot \left(1 - \frac{t}{4}\right) \le \sqrt{1+t} - 1 \le \frac{t}{2}, \quad \text{falls } \; 0 \le t < 1; \tag{C.95}$$

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:

$$V(x,y) \le \frac{y^2}{2} \cdot \left[\frac{1}{|x+1|} + \frac{1}{|x-1|}\right] = \frac{y^2}{1-x^2}. \tag{C.96}$$

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

$$V(x,y) \le \frac{y^2}{1-x^2}, \qquad \text{falls } |x| < 1 \text{ und } |y| < 2^{-\texttt{prec}}. \tag{C.97}$$

Um $V(x,y)$ einzuschließen, berechnen wir jetzt eine **Unterschranke**. Mit (C.95) 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:

$$1 - \frac{1}{4}\left(\frac{y}{x+1}\right)^2 > \texttt{pred}(1) = 1 - 2^{-\texttt{prec}}, \quad \text{d.h.} \quad \left(\frac{y}{x+1}\right)^2 < 2^{-\texttt{prec}+2}.$$

[12] 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:

$$\frac{1}{2} \cdot \frac{y^2}{|x+1|} \cdot \mathtt{pred}(1) \leq v_1, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}. \tag{C.98}$$

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

$$\frac{1}{2} \cdot \frac{y^2}{|x-1|} \cdot \mathtt{pred}(1) \leq v_2, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}, \quad \text{und damit:} \tag{C.99}$$

$$\frac{y^2}{2} \cdot \left[\frac{1}{|x+1|} + \frac{1}{|x-1|}\right] \cdot \mathtt{pred}(1) = \frac{y^2}{1-x^2} \cdot \mathtt{pred}(1) \leq v_1 + v_2 = V(x,y); \tag{C.100}$$

Für $V(x,y)$ gilt dann zusammen mit (C.96) für $\mathtt{prec} \geq 2$ die garantierte Einschließung

$$\mathtt{pred}(1) \cdot \frac{y^2}{1-x^2} \leq V(x,y) \leq \frac{y^2}{1-x^2}, \quad \text{falls} \ \ |y| < 2^{-((3\mathtt{prec})/2)}. \tag{C.101}$$

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 (C.88) auf Seite 333. 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:

$$\begin{aligned} &\sqrt{2\delta} \cdot \left(1 - \frac{\delta}{12}\right) \leq \operatorname{arcosh}(1+\delta) \leq \sqrt{2\delta}, \quad \delta = V(x,y)/2 < 2 \quad \rightsquigarrow \\ &\sqrt{V(x,y)} \cdot \left(1 - \frac{V(x,y)}{24}\right) \leq \operatorname{arcosh}(1+V(x,y)/2) \leq \sqrt{V(x,y)}, \quad \text{falls } V(x,y) < 4. \end{aligned} \tag{C.102}$$

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:

$$\sqrt{V} \cdot \mathtt{pred}(1) \leq \operatorname{arcosh}(1+V/2) \leq \sqrt{V}, \quad \text{falls} \ \ V(x,y) < 3 \cdot 2^{-\mathtt{prec}+3}. \tag{C.103}$$

Mit (C.101) und (C.103) 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[13]:

$$\mathtt{pred}(1) \cdot \sqrt{\mathtt{pred}(1)} \cdot \frac{|y|}{\sqrt{1-x^2}} \leq \operatorname{arcosh}(1+V(x,y)/2) \leq \frac{|y|}{\sqrt{1-x^2}}. \tag{C.104}$$

[13] Man kann mit (C.101) 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:

$$\texttt{pred}(1) \cdot \texttt{pred}(1) \cdot \frac{|y|}{\sqrt{1-x^2}} \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \frac{|y|}{\sqrt{1-x^2}}. \tag{C.105}$$

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:

$$\texttt{pred}(1) \cdot x \geq \texttt{pred}(x). \tag{C.106}$$

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 17 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 (C.106) sind damit zwei Fälle zu unterscheiden. Für $m = 0.5$ kann (C.106) 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 (C.106) 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 (C.105) 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 Einschließung 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 (C.106)

$$\begin{aligned} \texttt{pred}(1) \cdot \{\texttt{pred}(1) \cdot |y|/\sqrt{x^2-1}\} \quad &\geq \quad \texttt{pred}(1) \cdot \{\texttt{pred}(1) \cdot \texttt{u1}\} \geq \texttt{pred}(1) \cdot \{\texttt{pred(u1)}\} \\ &\geq \quad \texttt{pred}\{\texttt{pred(u1)}\}. \end{aligned}$$

Unter der Voraussetzung $|y| < 2^{-(3\mathtt{prec})/2}$ erhalten wir schließlich mit (C.105) für $\mathtt{prec} \geq 2$ die sehr effektive Einschließung

$$\mathtt{pred}\,\{\mathtt{pred}(\mathtt{u1})\} \leq \operatorname{arcosh}(1 + V(x,y)/2) \leq \mathtt{u2}. \tag{C.107}$$

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 (C.107) 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 (C.107) im Vergleich zu (C.102) 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 Eingangsintervall gegeben durch das Punktintervall $Z = [x_1, x_1] + i \cdot [y_1, y_1]$.

Mit (C.102) 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 (C.107) für den Imaginärteil die fast optimale Einschließung

$$\begin{aligned}\mathtt{asin}(Z) \;=\; &([1.57079632675373758748, 1.57079632675373758749],\\ &[5.78868064589607735286e-323228487, 5.78868064589607735289e-323228487]).\end{aligned}$$

C.2.21. $\arccos(z)$

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.

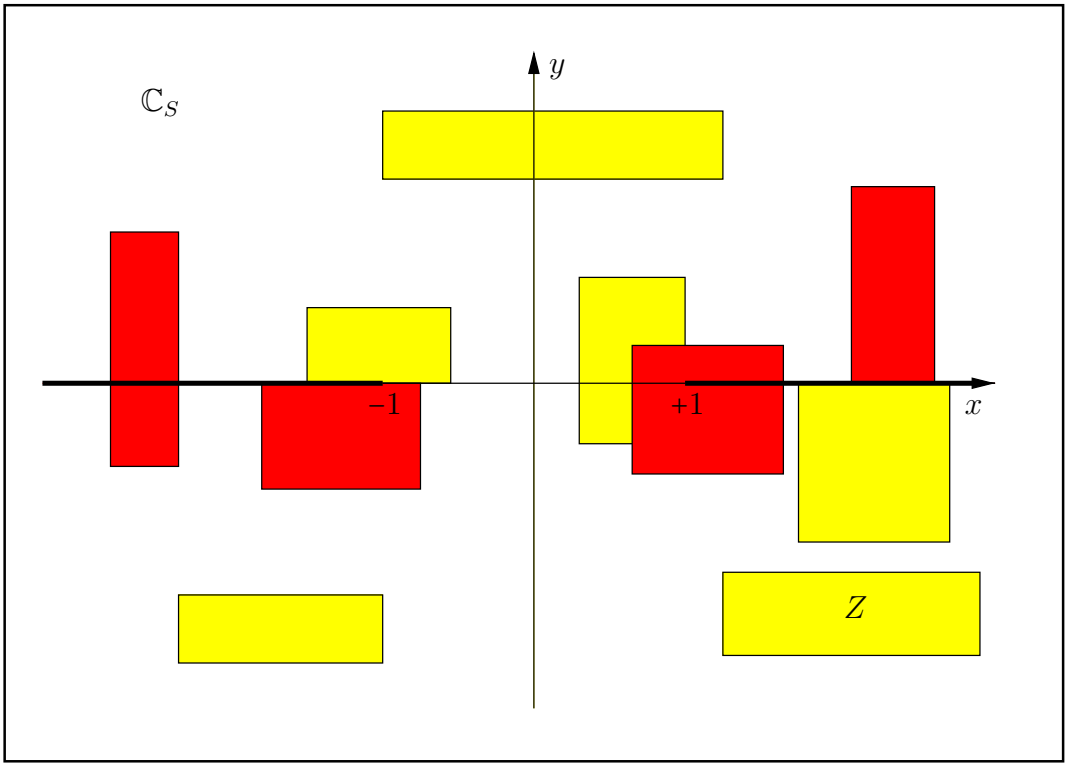


Abbildung C.40.: Erlaubte und nicht erlaubte Intervalle Z von `acos(Z)`.

Mit $Z = [1,1] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung[14]

$$\begin{aligned}\texttt{acos(Z)} \;=\; &([4.88115243040816240520e{-}161614249, 4.88115243040816240522e{-}161614249],\\ &[-4.88115243040816240522e{-}161614249, -4.88115243040816240520e{-}161614249]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{acos(Z)} \;=\; &([0.00000000000000000000000, 1.22135726393768332560392],\\ &[0.00000000000000000000000, 1.26047518779845407285292]).\end{aligned}$$

[14] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

C.2.21.1. Algorithmus

Der in [54] beschriebene Algorithmus verwendet die folgende Identität:

$$\arccos(z) \equiv \frac{\pi}{2} - \arcsin(z), \quad z = x + i \cdot y; \tag{C.108}$$

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 (C.108) 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 (C.108) 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 [38] 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:

$$\arccos(\boldsymbol{\beta}), \quad \boldsymbol{\beta} := \frac{2\boldsymbol{x}}{\sqrt{(\boldsymbol{x}+1)^2 + \boldsymbol{y}^2} + \sqrt{(\boldsymbol{x}-1)^2 + \boldsymbol{y}^2}}, \tag{C.109}$$

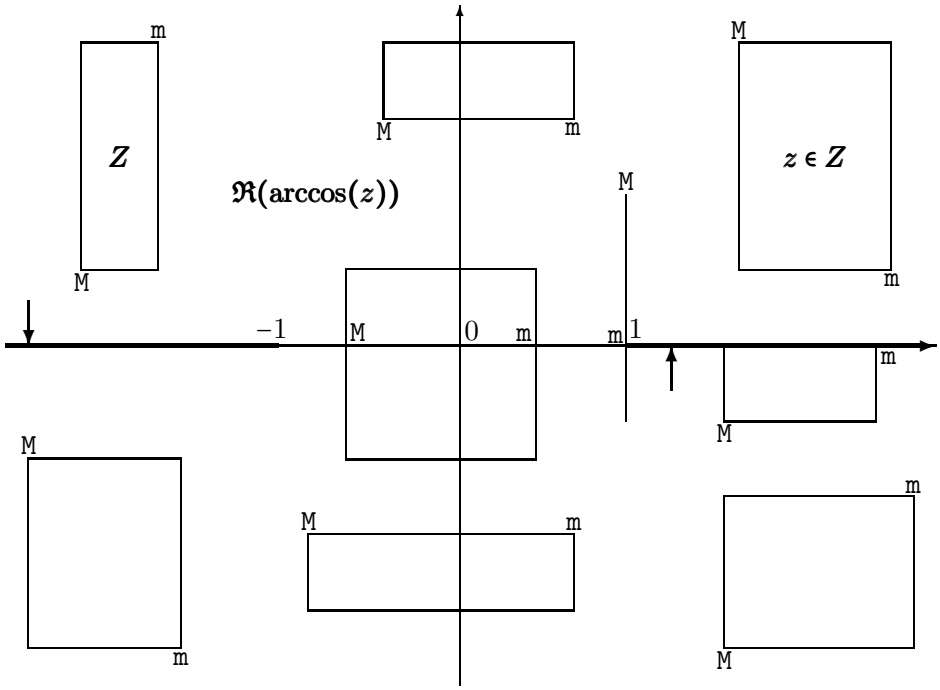
wobei die Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ durch die Extremalpunkte auf dem Rand von Z bestimmt sind, vgl. [38, S. 135].

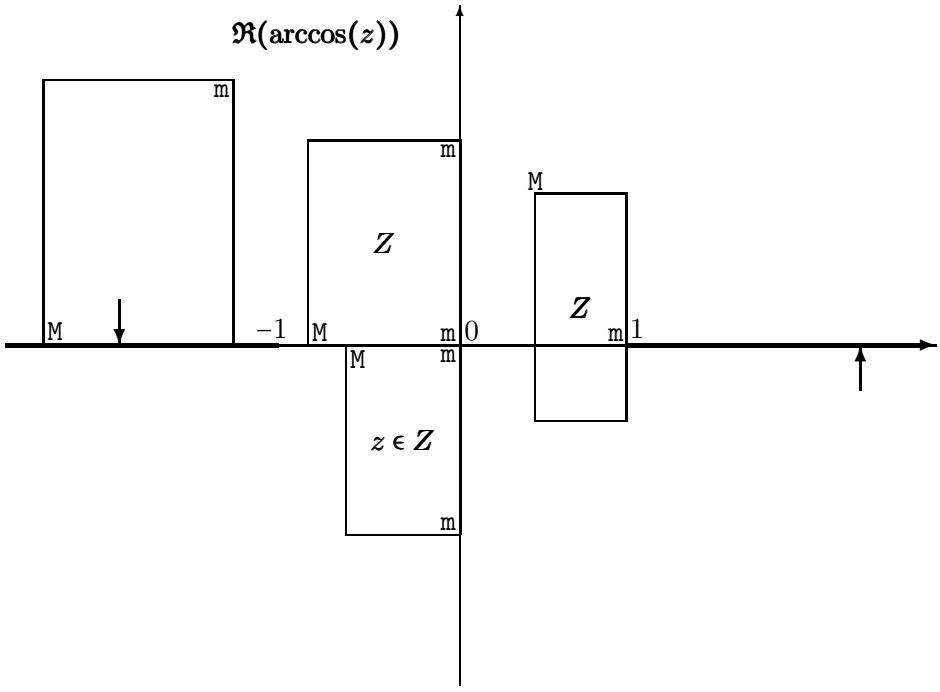
In den folgenden Abbildungen sind zu einigen Intervallen Z, die in der komplexen Ebene als Rechtecke dargestellt werden, die Extremalpunkte angegeben. Dabei sind die Koordinaten von `m` die Koordinaten des Extremalpunktes, an dem das Minimum von $\Re(w)$ angenommen wird. Entsprechend sind die Koordinaten 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 330 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 (C.109) 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

$$\beta := \frac{2x}{\sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2}} = 1 - \delta, \quad \delta \geq 0 \tag{C.110}$$

im Falle $\beta \geq 0.75$ für $\arccos(\beta)$ die folgende Darstellung benutzt:

$$\arccos(\beta) = \arccos(1-\delta) = \arcsin(\sqrt{2\delta - \delta^2}) \tag{C.111}$$

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 (C.110) 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 (C.110) erhält man für diese Maximumwerte direkt:

$$\beta(x,0) = \begin{cases} x & \text{falls } \ 0 \leq x \leq 1 \\ 1 & \text{falls } \ x > 1 \end{cases} \quad \rightsquigarrow \quad \delta = 0; \tag{C.112}$$

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

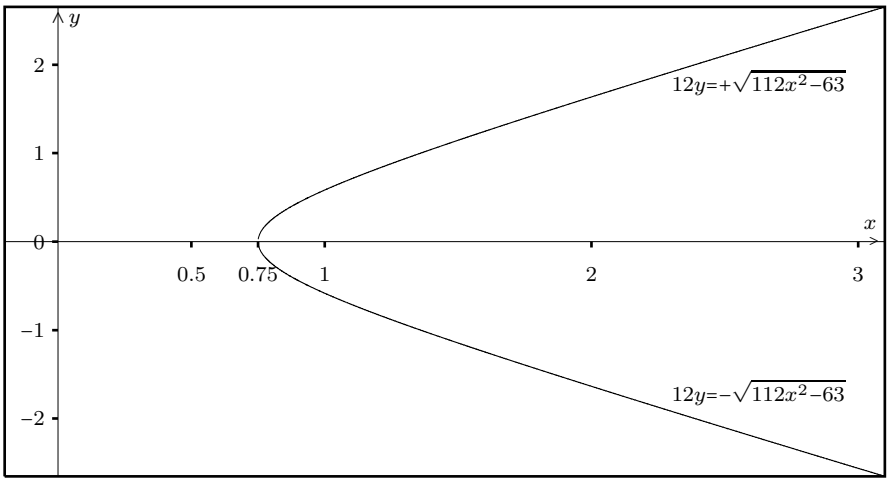
$$\beta(x,y) = \frac{3}{4} \quad \Longleftrightarrow \quad |y| = \frac{\sqrt{112\,x^2 - 63}}{12};$$

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 (C.111) 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

$$2\delta = 2 - \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \geq 0. \tag{C.113}$$

Aber (C.113) 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. (C.112), kann man sich jetzt auf $y \neq 0$ beschränken. Aus (C.113) 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 $x > 1$:

$$2\delta = \frac{\dfrac{y^2}{x^2-1} - \left[\sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right] \cdot \left[\sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right]}{1 + \dfrac{1}{2} \cdot \left(\sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right) + \dfrac{1}{2} \cdot \left(\sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right)}. \tag{C.114}$$

Der Nenner in (C.114) 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 (C.114) 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 (C.114) sehr einfach eingeschlossen werden. Für eine Oberschranke erhält man aus (C.114) direkt:

$$2\cdot\delta(x,y) \le \frac{y^2}{x^2-1}. \tag{C.115}$$

Die Berechnung einer Unterschranke für $2\cdot\delta(x,y)$ ist etwas komplizierter. Nach (C.114) 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 334 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:

$$\frac{1}{N} \ge 1-\frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2}, \quad \text{falls } \frac{|y|}{x-1}<1 \ \wedge\ \frac{y^2}{2}\cdot\frac{x^2+1}{(x^2-1)^2}<1. \tag{C.116}$$

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 (C.116). Es gilt damit:

$$\frac{1}{N} \geq 1 - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2}, \quad \text{falls} \quad \frac{|y|}{x-1} < 1. \tag{C.117}$$

Wir müssen jetzt noch eine Unterschranke des Zählers Z in (C.114) 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 (C.117) ergibt sich für $2 \cdot \delta(x,y)$ die Abschätzung:

$$2\delta \geq \frac{y^2}{x^2-1} \cdot \left\{1 - \frac{y^2}{4(x^2-1)}\right\} \cdot \left\{1 - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2}\right\}, \text{ falls } \frac{|y|}{x-1} < 1. \tag{C.118}$$

$$\begin{aligned} \big\{...\big\} \cdot \big\{...\big\} &\geq 1 - \frac{y^2}{4(x^2-1)} - \frac{y^2}{2} \cdot \frac{x^2+1}{(x^2-1)^2} \\ &= 1 - \frac{y^2}{2(x^2-1)} \left(\frac{1}{2} + \frac{x^2+1}{x^2-1}\right). \end{aligned} \tag{C.119}$$

$$\begin{aligned} \left(\frac{1}{2} + \frac{x^2+1}{x^2-1}\right) &= \frac{3x^2+1}{2(x+1)} \cdot \frac{1}{x-1} \leq \frac{1}{x-1} \cdot \frac{3x^2+1}{2x} \\ &= \frac{1}{x-1} \cdot \left[\frac{3}{2} \cdot x + \frac{1}{2x}\right] \leq \frac{1}{x-1} \cdot \left[\frac{3}{2} \cdot x + \frac{1}{2}\right] \\ &= \frac{3x+1}{2(x-1)} \qquad \rightsquigarrow \end{aligned}$$

$$\big\{...\big\} \cdot \big\{...\big\} \geq 1 - \frac{y^2}{4(x-1)^2} \cdot \frac{3x+1}{x+1}.$$

Wegen $x > 1$ gilt noch

$$\frac{3x+1}{x+1} \leq \frac{3x+1}{x} = 3 + \frac{1}{x} < 4 \quad \text{und damit}$$

$$\big\{...\big\} \cdot \big\{...\big\} \geq 1 - \frac{y^2}{(x-1)^2}.$$

Zusammen mit (C.115) und (C.118) erhalten wir unter der Voraussetzung $x > 1$ die Einschließung:

$$\left[1 - \frac{y^2}{(x-1)^2}\right] \cdot \frac{y^2}{x^2-1} \leq 2 \cdot \delta(x,y) \leq \frac{y^2}{x^2-1}, \quad \text{falls} \quad \frac{|y|}{x-1} < 1. \tag{C.120}$$

Wir verlangen zusätzlich noch

$$1 - \frac{y^2}{(x-1)^2} > \mathtt{pred}(1) = 1 - 2^{-\mathtt{prec}} \quad \Longleftrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}$$

und erhalten damit im Falle $x > 1$ für $2 \cdot \delta(x,y)$ die gesuchte Einschließung:

$$\mathtt{pred}(1) \cdot \frac{y^2}{x^2-1} \le 2 \cdot \delta(x,y) \le \frac{y^2}{x^2-1}, \quad \text{falls} \ \ \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}; \tag{C.121}$$

Mit Hilfe der gefundenen Doppelungleichung (C.121) 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:

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \frac{1}{1-\delta}, \quad 0 \le \delta < 1. \tag{C.122}$$

Wir verlangen noch zusätzlich

$$\frac{1}{1-\delta} < \mathtt{succ}(1) = 1 + 2^{-\mathtt{prec}+1} \quad \Longleftrightarrow \quad \delta < \frac{2^{-\mathtt{prec}+1}}{1 + 2^{-\mathtt{prec}+1}}$$

und die letzte Ungleichung ist erfüllt, wenn $\delta < 2^{-\mathtt{prec}}$, d.h. es gilt

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \mathtt{succ}(1), \quad \text{falls} \ \ \delta < 2^{-\mathtt{prec}}. \tag{C.123}$$

Zusammen mit (C.121) ergibt sich daraus:

$$\sqrt{\mathtt{pred}(1)} \cdot \frac{|y|}{\sqrt{x^2-1}} \le \arccos(1-\delta) \le \frac{|y|}{\sqrt{x^2-1}} \cdot \mathtt{succ}(1). \tag{C.124}$$

Die bisherigen Bedingungen für (C.124) lauten:

$$x > 1 \tag{C.125}$$

$$\frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}} \tag{C.126}$$

$$\delta < 2^{-\mathtt{prec}}. \tag{C.127}$$

Wir zeigen jetzt, dass mit (C.125) und (C.126) die Bedingung (C.127) automatisch erfüllt ist. Nach (C.121) gilt mit (C.126) die Abschätzung:

$$\begin{aligned} \delta &\le \frac{y^2}{2(x^2-1)}, \quad \text{und damit} \\ \delta &\le \frac{y^2}{(x-1)^2} \cdot \frac{x-1}{2(x+1)}, \quad \text{und wegen} \quad \frac{x-1}{2(x+1)} < \frac{1}{2} \quad \text{folgt} \\ \delta &< 2^{-\mathtt{prec}} \cdot \frac{1}{2} = 2^{-\mathtt{prec}-1} < 2^{-\mathtt{prec}} \ \blacksquare \end{aligned}$$

Für $\arccos(1-\delta)$ gilt damit im Falle $x > 1$ und $|y|/(x-1) < \sqrt{2^{-\mathtt{prec}}}$ die Einschließung

$$\sqrt{\mathtt{pred}(1)} \cdot \frac{|y|}{\sqrt{x^2-1}} \le \arccos(1-\delta) \le \frac{|y|}{\sqrt{x^2-1}} \cdot \mathtt{succ}(1). \tag{C.128}$$

Um für $\arccos(1-\delta)$ eine möglichst einfach auszuwertende Einschließung angeben zu können, beweisen wir jetzt noch den folgenden **Satz**:

Für alle Maschinenzahlen $x \ge 0$ gilt:

$$x \cdot \mathtt{succ}(1) \le \mathtt{succ}(\mathtt{succ}((x))). \tag{C.129}$$

Der **Beweis** für $x = 0$ ist trivial. Wir betrachten daher jetzt positive Zahlen $x = m \cdot 2^{ex}$, mit $0.5 \le m < 1$. Nach (3.2) von Seite 17 gilt $\mathtt{succ}(x) = x + 2^{ex-\mathtt{prec}}$ und daraus folgt direkt:

$$\begin{aligned}
x \cdot \mathtt{succ}(1) \le \mathtt{succ}(\mathtt{succ}((x)) \quad &\Longleftrightarrow \quad \mathtt{m} \cdot 2^{\mathtt{ex}} \cdot (1 + 2^{1-\mathtt{prec}}) \le \mathtt{succ}[x + 2^{-\mathtt{prec}+ex}] = \\
&= \quad x + 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]}. \text{ Zu zeigen ist also} \\
&\quad x \cdot (1 + 2^{1-\mathtt{prec}}) \le x + 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad x \cdot 2^{1-\mathtt{prec}} \le 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad \mathtt{m} \cdot 2^{ex-\mathtt{prec}+1} \le 2^{-\mathtt{prec}+ex} + 2^{-\mathtt{prec}+\mathtt{expo}[\ldots]} \\
&\Longleftrightarrow \quad \mathtt{m} \cdot 2^{ex+1} \le 2^{ex} + 2^{\mathtt{expo}[\ldots]}.
\end{aligned}$$

Mit den zusätzlichen Abschätzungen: $\mathtt{m} \cdot 2^{ex+1} < 2^{ex+1}$ und $2^{ex} + 2^{\mathtt{expo}[\ldots]} \ge 2^{ex} + 2^{ex} = 2^{ex} + 1$ bleibt also zu zeigen $2^{ex+1} \le 2^{ex+1}$ ∎

Man beweist sehr einfach $\sqrt{\mathtt{pred}(1)} > \mathtt{pred}(1)$, und mit dem **Satz**

$$\mathtt{pred}(1) \cdot x \ge \mathtt{pred}(x), \quad \text{falls } x \ge 0$$

von Seite 337 folgt zusammen mit (C.128) 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: $\mathtt{pred}(\mathtt{u1}) \le \arccos(1-\delta) \le \mathtt{succ}(\mathtt{succ}(\mathtt{u2}))$.

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[15], 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 `prec` ≥ 2 ungerade ist.

Für `prec` ≥ 2 und $x > 1$ müssen wir also noch beweisen:

$$\mathtt{expo}(\mathtt{Sup}(\mathtt{tm})) <= -\mathtt{prec} \oslash 2 - 1 \quad \Longrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}. \tag{C.130}$$

Zum **Beweis** sei zunächst $y \ge 0$. Dann gilt $y/(x-1) \le \mathtt{Sup}(\mathtt{tm})$ und die Behauptung rechts in (C.130) ist erfüllt, falls

$$\begin{aligned}
&\mathtt{Sup}(\mathtt{tm}) < \sqrt{2^{-\mathtt{prec}}} \\
\Longleftrightarrow \quad &m \cdot 2^{ex} < \sqrt{2^{-\mathtt{prec}}}, \qquad 0.5 \le m < 1, \quad ex := \mathtt{expo}(\mathtt{Sup}(\mathtt{tm})) \\
\Longleftrightarrow \quad &m^2 \cdot 2^{2ex} < 2^{-\mathtt{prec}},
\end{aligned}$$

[15] `x,y` sind Punktintervalle, welche die Maschinenzahlen x, y einschließen.

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:

$$\mathtt{expo(Sup(tm))} <= -\mathtt{prec}/2 \quad \Longrightarrow \quad \frac{|y|}{x-1} < \sqrt{2^{-\mathtt{prec}}}. \tag{C.131}$$

Da auf der Maschine integer-Divisionen zur Null gerundet werden, gilt $-\mathtt{prec}/2 \geq -\mathtt{prec} \oslash 2 - 1$, womit dann (C.130) für $y \geq 0$ bewiesen ist.
Für den Rest des Beweises sei jetzt $y < 0$. Dann gilt $0 \geq \mathtt{Sup(tm)} \geq y/(x-1)$ bzw.

$$-\mathtt{Sup(tm)} \leq \frac{-y}{x-1} = \frac{|y|}{x-1}, \tag{C.132}$$

und die Behauptung rechts in (C.130) ist erfüllt, falls

$$\begin{aligned} & -\mathtt{Sup(tm)} < \sqrt{2^{-\mathtt{prec}}} \\ \Longleftrightarrow \quad & -(m \cdot 2^{ex}) < \sqrt{2^{-\mathtt{prec}}}, \qquad -1 < m \leq -0.5, \quad ex := \mathtt{expo(Sup(tm))} \\ \Longleftrightarrow \quad & m^2 \cdot 2^{2ex} < 2^{-\mathtt{prec}}, \end{aligned}$$

und wegen $m^2 < 1$ ist die letzte Ungleichung erfüllt, falls $2^{2ex} < 2^{-\mathtt{prec}}$, bzw. falls $ex < -\mathtt{prec}/2$. Der Rest des Beweises verläuft dann wie auf Seite 348 ∎

Im Fall $x > 1$ und $|y|/(x-1) \geq \sqrt{2^{-\mathtt{prec}}}$ wird δ nach (C.114) ausgewertet:

$$2\delta = \frac{\dfrac{y^2}{x^2-1} - \left[\sqrt{1+\left(\dfrac{y}{x-1}\right)^2} - 1\right] \cdot \left[\sqrt{1+\left(\dfrac{y}{x+1}\right)^2} - 1\right]}{1 + \dfrac{1}{2} \cdot \left(\sqrt{1+\left(\dfrac{y}{x-1}\right)^2} - 1\right) + \dfrac{1}{2} \cdot \left(\sqrt{1+\left(\dfrac{y}{x+1}\right)^2} - 1\right)} \tag{C.133}$$

Dabei entsteht sofort wieder die Frage, ob der Zähler in (C.133) als Differenz zweier positiver Größen ohne merkliche Auslöschung berechnet werden kann. Zur Beantwortung dieser Frage betrachten wir die für $y \neq 0$ äquivalenten Gleichungen

$$\begin{aligned} \frac{y^2}{x^2-1} - \left[\sqrt{1+\left(\frac{y}{x-1}\right)^2} - 1\right] \cdot \left[\sqrt{1+\left(\frac{y}{x+1}\right)^2} - 1\right] &= r \cdot \frac{y^2}{x^2-1} \\ 1 - r = \frac{1}{y^2} \cdot \left[\sqrt{(x-1)^2+y^2} - (x-1)\right] \cdot \left[\sqrt{(x+1)^2+y^2} - (x+1)\right] & \end{aligned}$$

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 (C.133) 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:

$$1 - r = \left[\sqrt{\left(\frac{x-1}{y}\right)^2 + 1} - \frac{x-1}{y}\right] \cdot \left[\sqrt{\left(\frac{x+1}{y}\right)^2 + 1} - \frac{x+1}{y}\right] \tag{C.134}$$

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:

$$y = \frac{1}{12} \cdot \sqrt{112\,x^2 - 63} \tag{C.135}$$

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 (C.135) in (C.134) ein. Nach einigen Umrechnungen erhält man dann das erstaunlicher Ergebnis:

$$1-r \equiv \frac{1}{7} = 0.142857\ldots, \quad \text{falls } x \geq 1, \tag{C.136}$$

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 (C.133) 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 (C.113) von Seite 343 erhält man:

$$\delta = \left\{1 - \sqrt{1+\left(\frac{y}{2}\right)^2}\right\} + \frac{|y|}{2}, \quad \text{falls } x = 1; \tag{C.137}$$

Wir zeigen zunächst, dass (C.137) im Bereich $\beta \geq 0.75$ nur für $|y| \leq 7/12$ auszuwerten ist. Im Falle $x = 1$ gilt nach (C.113)

$$\begin{aligned}
\beta \quad &= \quad \frac{2}{\sqrt{4+y^2}+|y|} \geq \frac{3}{4} \\
&\iff \sqrt{4+y^2}+|y| \leq \frac{8}{3} \quad \iff \quad \sqrt{4+y^2} = \frac{8}{3}-|y| \\
&\iff |y| \leq \frac{7}{12} = 0.58333\ldots \ \blacksquare
\end{aligned}$$

In (C.137) wird der erste Summand $\{.....\} \leq 0$ mit Hilfe der Funktion `sqrtp1m1()` ausgewertet, die Auslöschung innerhalb von $\{\ldots\}$ vermeidet. Man könnte jedoch einwenden, dass in (C.137) die Addition eines negativen und positiven Summanden zur Auslöschung führen kann. Um dies auszuschließen, zeigen wir mit $t := |y|/2$, dass in

$$t - \left\{\sqrt{1+t^2}-1\right\} = r\cdot t \tag{C.138}$$

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

$$r(t) \geq r(7/24) = \frac{6}{7} = 0.85714\ldots \tag{C.139}$$

und mit dieser Unterschranke für $r(t)$ folgt aus (C.138), dass für $|y| \leq 7/12$ bei der Auswertung von (C.137) keine wesentliche Auslöschung auftreten kann. Damit könnte man den Fall $x = 1$ als erledigt ansehen. Für $|y| \ll 1$ lässt sich die Auswertung nach (C.137) jedoch noch wesentlich

vereinfachen. Mit $t := |y|/2$ folgt nach (C.137) zunächst $\delta = 1 + t - \sqrt{1+t^2}$, und damit ergibt sich ganz elementar die folgende Einschließung für δ:

$$t \cdot \left(1 - \frac{t}{2}\right) \le \delta \le t, \quad t := \frac{|y|}{2}. \tag{C.140}$$

Mit $0 \le \delta < 1$ gilt nach (C.122) für $\arccos(1-\delta)$ die Einschließung

$$\sqrt{2\delta} \le \arccos(1-\delta) \le \sqrt{2\delta} \cdot \frac{1}{1-\delta}, \quad 0 \le \delta < 1, \tag{C.141}$$

und zusammen mit (C.140) folgt direkt:

$$\sqrt{2t} \cdot \sqrt{1 - \frac{t}{2}} \le \arccos(1-\delta) \le \frac{\sqrt{2t}}{1-t}, \quad \text{falls} \quad 0 \le t = \frac{|y|}{2} < 1. \tag{C.142}$$

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 (C.142) erhält man:

$$\mathtt{pred}(1) \cdot \sqrt{|y|} \le \arccos(1-\delta) \le \frac{\sqrt{|y|}}{1-t}, \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 2. \tag{C.143}$$

Wie auf Seite 346 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 (C.143) ergibt sich daraus die Einschließung:

$$\mathtt{pred}(1) \cdot \sqrt{|y|} \le \arccos(1-\delta) \le \sqrt{|y|} \cdot \mathtt{succ}(1), \quad \text{falls} \quad \mathtt{expo}(y) \le -\mathtt{prec} + 1. \tag{C.144}$$

Um die Einschließung in (C.144) 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 (C.144) erhält man daher

$$\mathtt{pred}(1) \cdot \mathtt{u1} \le \arccos(1-\delta) \le \mathtt{u2} \cdot \mathtt{succ}(1), \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 1; \tag{C.145}$$

Bei Anwendung von (C.106) und (C.129) kann man außerdem noch die beiden Multiplikationen in (C.145) vermeiden:

$$\mathtt{pred(u1)} \le \arccos(1-\delta) \le \mathtt{succ(succ(u2))}, \quad \text{falls } \mathtt{expo}(y) \le -\mathtt{prec} + 1. \tag{C.146}$$

(C.146) 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 $\texttt{expo}(y) > -\texttt{prec}+1$ benutzt man (C.137) 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 **$0.75 \leq x < +1$** betrachten:
Ausgehend von (C.113) auf Seite 343 findet man nach einigen Umrechnungen:

$$\begin{aligned} 2\delta &= -(x+1)\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\}+ \\ &\quad +(1-x)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1\right\}+2\cdot(1-x) \end{aligned}$$

(C.147)
$$\begin{aligned} 2\cdot\delta(x,y) &= -(x+1)\cdot\left\{\sqrt{1+\left(\frac{y}{x+1}\right)^2}-1\right\}+ \\ &\quad +(1-x)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}+1\right\}, \quad \text{falls } x<1; \end{aligned}$$

Da in (C.147) 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

$$\begin{aligned} &(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right]-(1+x)\cdot\left[\sqrt{1+\left(\frac{y}{1+x}\right)^2}-1\right] \\ &\qquad = r\cdot(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right] \end{aligned}$$

(C.148)
$$1-r = \frac{(1+x)\cdot\left[\sqrt{1+\left(\frac{y}{1+x}\right)^2}-1\right]}{(1-x)\cdot\left[\sqrt{1+\left(\frac{y}{1-x}\right)^2}+1\right]}$$

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 (C.147) 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:

(C.149)
$$y = \frac{1}{12}\cdot\sqrt{112\,x^2-63}$$

Für $0.75 \leq x < 1$ ist die partielle Ableitung $\partial(1-r)/\partial y$ mit

$A := \sqrt{(1-x)^2+y^2} > 1-x, \quad B := \sqrt{(1+x)^2+y^2} > 1+x$ gegeben durch:

(C.150)
$$\frac{\partial(1-r)}{\partial y} = \frac{y\cdot\{-4x+A\cdot(1-x)+B\cdot(1+x)\}}{A\cdot B\cdot(A+1-x)^2}$$

und für $y>0$ ist $\partial(1-r)/\partial y > 0$, wenn in (C.150) gilt: $\{\ldots\} > 0$, d.h. wenn

(C.151)
$$C := (1-x)\cdot\sqrt{(1-x)^2+y^2}+(1+x)\cdot\sqrt{(1+x)^2+y^2} > 4x.$$

Wegen $C \geq (1-x)^2 + (1+x)^2 = 2 + 2x^2$ ist (C.151) 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

$$y = \frac{1}{12} \cdot \sqrt{112x^2 - 63}. \tag{C.152}$$

Um das Maximum zu berechnen, setzen wir y aus (C.152) ein in (C.148) 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 (C.147) 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 (C.147),(C.137),(C.114) 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 340 die reelle arccos-Funktion für den Intervall-Ausdruck $\boldsymbol{\beta}$ auszuwerten:

$$\arccos(\boldsymbol{\beta}), \quad \boldsymbol{\beta} := \frac{2\boldsymbol{x}}{\sqrt{(\boldsymbol{x}+1)^2 + \boldsymbol{y}^2} + \sqrt{(\boldsymbol{x}-1)^2 + \boldsymbol{y}^2}}, \tag{C.153}$$

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:

$$\Re(w) \in [3.14159262\mathbf{37874627}, 3.14159265\mathbf{35898020}]$$

mit nur 8 korrekten Dezimalziffern. Diese zu groben Einschließungen lassen sich durch Anwendung folgender, für $0 \leq \delta \leq 2$ geltenden Identität vermeiden:

$$\arccos(-1+\delta) \equiv \pi - \arccos(1-\delta) \equiv \pi - \arcsin(\sqrt{\delta \cdot (2-\delta)}). \tag{C.154}$$

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 (C.114),(C.137) und (C.147) intervallmäßig berechnet werden kann. Die intervallmäßige Auswertung der rechten Seite von (C.154) 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.

C.2.22. $\arctan(z)$

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.

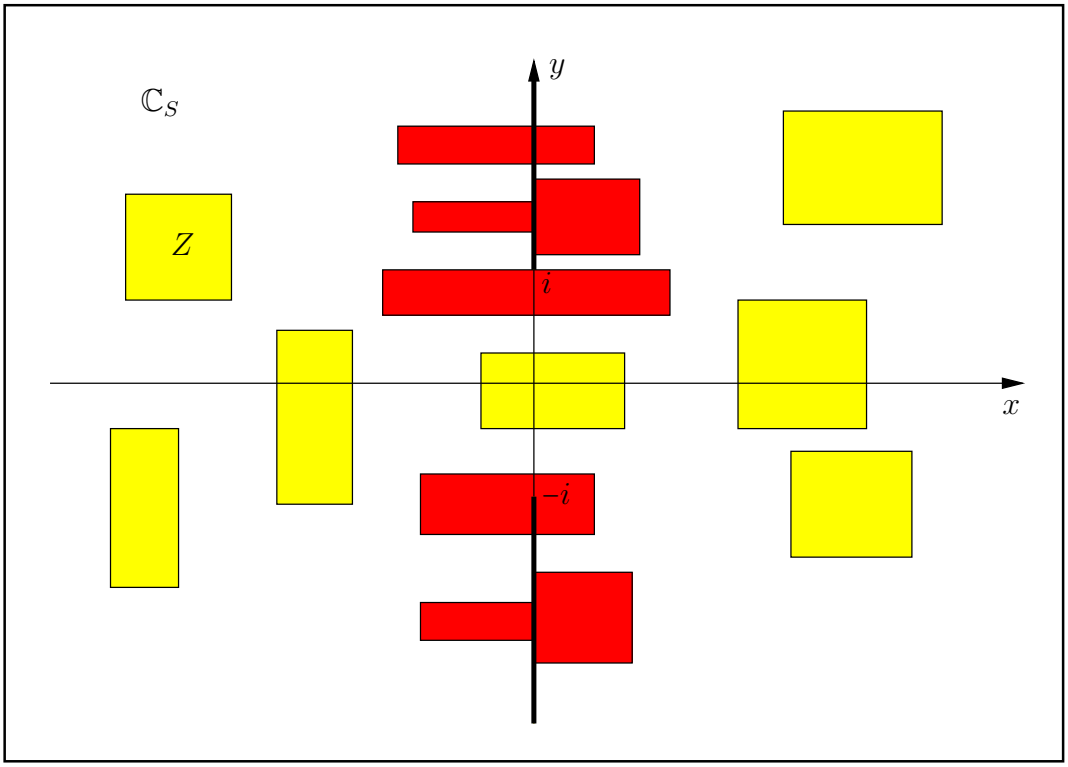


Abbildung C.41.: Erlaubte und nicht erlaubte Intervalle Z von `atan(Z)`.

Nach Abbildung C.41 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[16]

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([7.85398163397448309613e-1, 7.85398163397448309619e-1],\\ &[3.72130559324020099216e8, 3.72130559324020099218e8]).\end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 1]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([4.6364760900080611621425 6e-1, 1.1071487177940905030170 7],\\ &[-7.0830333601405402006238 5e-1, 7.0830333601405402006238 5e-1]).\end{aligned}$$

[16] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

C.2.22.1. Algorithmus

In [54] 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:

$$T(x) := \ln\left[1 + \frac{2}{\sqrt{1+x^2}-1}\right], \quad x = [x]: \text{ Punktintervall} \tag{C.155}$$

$$Q_{1,2}(x,y) := \ln\left[1 \pm \frac{4y}{x^2 + (1\mp y)^2}\right], \quad \text{nur } y = [y] \text{ ist ein Punktintervall.} \tag{C.156}$$

In (C.156) 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:

$$T(x) = \ln\left[2 + x^2 + 2\cdot\sqrt{1+x^2}\right] - 2\cdot\ln(x) \tag{C.157}$$

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:

$$\begin{aligned}\alpha &:= 2 + x^2 + 2\cdot\sqrt{1+x^2} \\ &= 4 + \left[2x^2 + \sum_{k=2}^{\infty}(-1)^{k+1}\frac{1\cdot3\cdot5\cdot\ldots\cdot(2k-3)}{4\cdot6\cdot8\cdot\ldots\cdot2k}\cdot(x^2)^k\right],\end{aligned}$$

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:

$$\ln(4) < \ln(\alpha) < \ln(4+2x^2) = \ln(4) + \ln(1 + \frac{1}{2}\cdot x^2) < \ln(4) + \frac{1}{2}\cdot x^2, \quad \text{d.h.}$$

$$\ln(4) < \ln(\alpha) < \ln(4) + \frac{1}{2}\cdot x^2, \quad x^2 < 1. \tag{C.158}$$

Es gilt also für das folgende Nicht-Maschinenintervall $\mathtt{Nm}(x)$

$$\ln(\alpha) \in \mathtt{Nm}(x) := \left[\ln(4)\ ,\ \ln(4) + \frac{1}{2}\cdot x^2\right], \quad x^2 < 1. \tag{C.159}$$

Das Ziel ist nun, $x \in]0,1[$ in (C.159) so klein zu wählen, dass für jede Current-Präzision `prec`, mit $2 \le \mathtt{prec} \le \mathtt{prec}_0$, ein Maschinenintervall `U` so angegeben werden kann, dass gilt:

$$\mathtt{Nm}(x) \subseteq \mathtt{U} \quad \text{und} \quad \mathtt{sup(U)} = \mathtt{succ(Inf(U))}. \tag{C.160}$$

Es ist klar, dass die zweite Bedingung in (C.160) 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[17] 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 $\mathtt{U}$ 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:

$$0 \le x \le 0.5 \cdot 2^{k_0} \quad \Longrightarrow \quad T(x) \in -2 \diamond \ln([x]) \oplus \hat{\mathtt{U}}, \quad 2 \le \mathtt{prec} \le \mathtt{prec}_0.$$

Mit einer zweiten Schleife (Programmteil `Schleife2`) findet man dazu[18]

$$\text{(C.161)} \quad 0 \le x \le 0.5 \cdot 2^{-(\mathtt{prec} \oslash 2+9)},\ 2 \le \mathtt{prec} \le \mathtt{prec}_0 = 677\,370 \quad \Longrightarrow \quad T(x) \in -2 \diamond \ln([x]) \oplus \hat{\mathtt{U}}.$$

[17] Es zeigt sich, dass die berechneten Einschließungen in doppelter Präzision eng genug sind.

[18] $\oslash$ bedeutet die integer-Division auf der Maschine, wobei zur Null hin gerundet wird, wenn der exakte Quotient keine integer-Zahl ist.

```
// 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 (C.161) 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 (C.157) 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 (C.155) wird der Nenner $\sqrt{1+x^2}-1$ mit Hilfe der Funktion $\mathtt{sqrt1pm1(sqr([}x\mathtt{]))}$ ausgewertet. Um dabei einen Überlauf zu vermeiden, verlangen wir

$$x^2 = \mathtt{m}^2\cdot 2^{2ex} < 2^{2ex} < 2^{+1073741820} \quad\iff\quad \mathtt{expo}(x) = ex < 536870910,$$

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 (C.155) 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:

$$2r+2r^2 < \beta(r) < 2r+2r^2+r^3, \quad 0<r<1. \tag{C.162}$$

Da die Taylorreihe von

$$\ln(1+x) = x-\frac{1}{2}x^2+\frac{1}{3}x^3-+\ldots, \quad -1<x\le 1$$

für $0 < x \le 1$ eine alternierende Leibniz-Reihe ist, folgt die Ungleichung

$$x-\frac{1}{2}x^2 < \ln(1+x) < x, \quad 0<x\le 1; \tag{C.163}$$

Mit den rechten Ungleichungen aus (C.162) und (C.163) 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 (C.162) und (C.163)

$$\ln(1+\beta(r)) > \beta(r) - \frac{1}{2}\beta(r)^2 > 2r + 2r^2 - \frac{1}{2}\beta(r)^2$$

Aus (C.162) 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:

$$2r - 4r^3 - 7r^4 < \ln(1+\beta(r)) < 2r + 2r^2 + r^3, \qquad 2r + 2r^2 + r^3 < 1;$$
$$2r \cdot (1 - 2r^2 - \frac{7}{2}r^3) < \ln(1+\beta(r)) < 2r \cdot (1 + r + \frac{1}{2}r^2),$$

die noch etwas vereinfacht werden kann:

$$2r \cdot (1 - 2r^2 - 4r^3) < \ln(1+\beta(r)) < 2r \cdot (1 + r + \frac{1}{2}r^2), \qquad 2r + 2r^2 + r^3 < 1. \tag{C.164}$$

Für hinreichend kleine r kann diese Einschließung noch wesentlich vereinfacht werden. Dazu verlangen wir für die Unterschranke

$$\begin{aligned} & 1 - 2r^2 - 4r^3 > \mathtt{pred}(1) = 1 - 2^{-\mathtt{prec}} \\ \iff \quad & 2r^2 \cdot (1 + 2r) < 2^{-\mathtt{prec}}. \end{aligned} \tag{C.165}$$

Mit $r = 1/x = (1/m) \cdot 2^{-ex}$, $ex \gg 1$, folgt weiter

$$1 + 2r = 1 + \frac{1}{\mathtt{m}} \cdot 2^{-ex+1} \le 1 + 2^{-ex+2}, \quad \text{d.h. (C.165) ist erfüllt, wenn gilt:} \quad 2r^2 \cdot 2 < 2^{-\mathtt{prec}}.$$

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. (C.165) 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. (C.165) ist erfüllt, wenn gilt

$$ex > \mathtt{prec} \oslash 2 + 3 \quad \Longrightarrow \quad 1 - 2r^2 - 4r^3 > \mathtt{pred}(1). \tag{C.166}$$

Zur Verbesserung der Oberschranke in (C.164) verlangen wir jetzt

$$\begin{aligned} & 1 + r + \frac{r^2}{2} < \mathtt{succ}(1) = 1 + 2^{-\mathtt{prec}+1} \\ \iff \quad & r + \frac{r^2}{2} < 2^{-\mathtt{prec}+1}. \end{aligned} \tag{C.167}$$

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. (C.167) ist erfüllt, wenn gilt: $2^{-ex+2} < 2^{-\mathtt{prec}+1} \iff ex > \mathtt{prec}+1$. Damit erhalten wir

$$ex > \mathtt{prec}+1 \quad \Longrightarrow \quad 1+r+\frac{r^2}{2} < \mathtt{succ}(1). \tag{C.168}$$

Zusammen mit (C.164) 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 (C.106) und (C.129) erhalten wir mit $x = \mathtt{m} \cdot 2^{ex}$, $r = 1/x$, $\mathtt{t} = [2r]$:

$$ex > \mathtt{prec}+2 \quad \Longrightarrow \quad \mathtt{pred(Inf(t))} < \ln(1+\beta(r)) < \mathtt{succ(succ(Sup(t)))}. \tag{C.169}$$

Im Fall $ex \le \mathtt{prec}+2$, der wegen $ex \ge 536870910$ in der Praxis kaum auftreten wird, benutzen wir:

$$2r \cdot [1-2r^2(1+2r)] < \ln(1+\beta(r)) < 2r \cdot [1+r(1+r)]. \tag{C.170}$$

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 (C.156) ist auszuwerten

$$Q_1(x,y) := \ln\left[1+\frac{4y}{x^2+(1-y)^2}\right], \tag{C.171}$$

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 (C.171) 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

$$b := \frac{y}{x^2 + (1-y)^2} \tag{C.172}$$

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 (C.171):

$$2^{2s+2} \cdot bs = \frac{4 \cdot y}{x^2 + (1-y)^2}$$

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

$$Q_1([x],[y]) \subseteq \ln\left[1 + \frac{4[y]}{[x]^2 + (1-[y])^2}\right] \tag{C.173}$$

eine garantierte Einschließung ohne zwischenzeitlichen Overflow. Für $y \to 0$ und $x \to \infty$ muss in (C.173) 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:

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([0.0, 0.0],\\ &[6.96612916462745035964318282065467450915877635032056530391283e1,\\ &\;6.96612916462745035964318282065467450915877635032056530391284e1]).\end{aligned}$$

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:

$$\begin{aligned}\texttt{atan(Z)} \;=\; &([0.0, 0.0],\\ &[6.93147527133535589389886830074237297163784172110322434248307e5,\\ &\;6.93147527133535589389886830074237297163784172110322434248308e5,]),\end{aligned}$$

wobei hier bei der Ausgabe die gleiche Dezimalstellenzahl wie im ersten Beispiel gewählt wurde.

Im Algorithmus von $\texttt{atan}(Z)$ 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

(C.174) $\mathbf{D}_0 \coloneqq 1 - y^2 - x^2$, x, y vom Typ MpfiClass, wobei nur x ein Punktintervall ist.

Prinzipiell ist $\mathbf{D}_0$ in (C.174) schon optimal, da nach [6, 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 $\texttt{prec} = 70$ 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 $\texttt{prec} = 70000$ die Einschließung

$$\begin{aligned} \texttt{atan(Z)} \;=\; & ([-1.4986879...456363e - 323207425, -1.4986879...456362e - 323207425], \\ & [2.4260497893188365802 2e4, 2.4260497893188365802 3e4]). \end{aligned}$$

$\texttt{minfloat}() \coloneqq 2^{-1073741824}$ ist die, von der Current-Präzision unabhängige, kleinste positive Maschinenzahl.

C.2.23. 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.

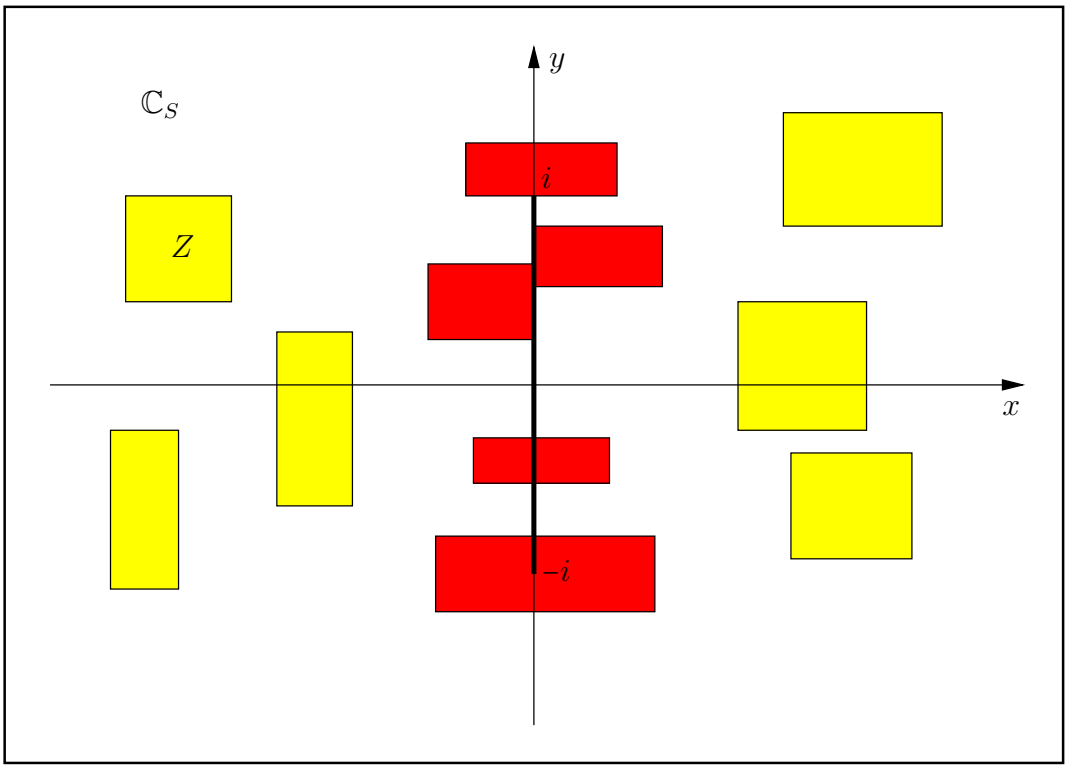


Abbildung C.42.: Erlaubte und nicht erlaubte Intervalle Z von `acot(Z)`.

Nach Abbildung C.42 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[19]

$$\begin{aligned} \texttt{acot(Z)} \;=\; & ([1.57079632679489661922, 1.57079632679489661924], \\ & [-2.46067249098780584844e1, -2.46067249098780584842e1]). \end{aligned}$$

Mit $Z = [0.5, 1.5] + i \cdot [-1, 1]$ erhält man mit der Current-Präzision `prec` = 80 die Einschließung

$$\begin{aligned} \texttt{acot(Z)} \;=\; & ([4.6364760900080611621425 6e-1, 1.1071487177940905030170 7], \\ & [-7.0830333601405402006238 5e-1, 7.0830333601405402006238 5e-1]). \end{aligned}$$

[19] Die Maschinenzahl $2^{-1073741824}$ = `minfloat()` ist präzisionsunabhängig die kleinste positive Maschinenzahl.

C.2.24. 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

$$\texttt{asinh}(Z) = i \cdot \texttt{asin}(-i \cdot Z);$$

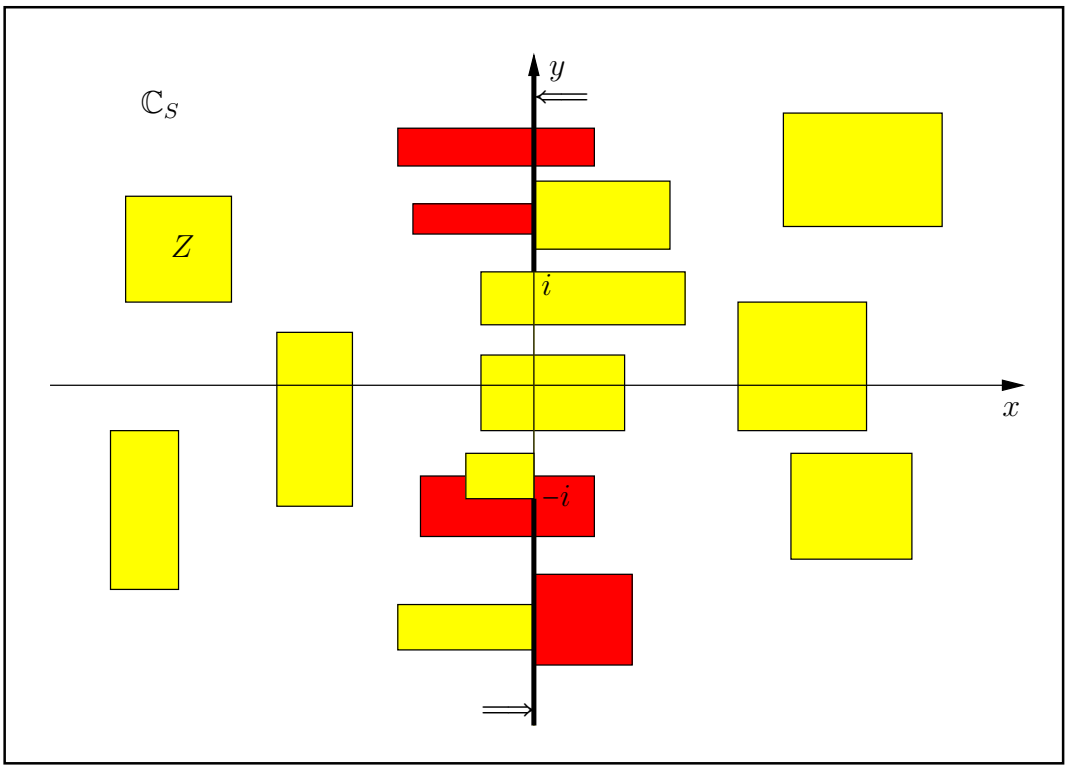


Abbildung C.43.: Erlaubte und nicht erlaubte Intervalle Z von `asinh(Z)`.

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

$$\begin{aligned}\texttt{asinh(Z)} \;=\; &([-7.3285767597364526088867 5e-1, 7.3285767597364526088867 5e-1],\\ &[-1.5707963267948966192313 3, -4.5227844715119068206365 7e-1]).\end{aligned}$$

C.2.25. 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

$$\texttt{acosh}(Z) = i \cdot \texttt{acos}(Z) = \pm i \cdot (\pi/2 - \texttt{asin}(Z));$$

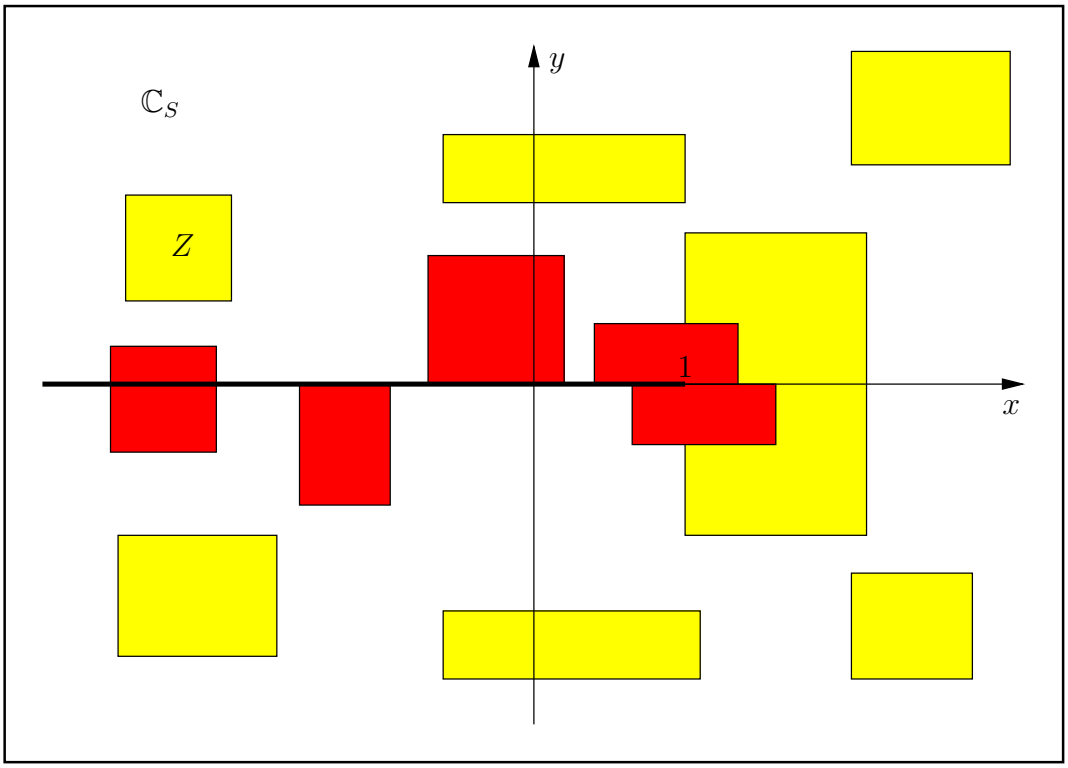


Abbildung C.44.: Erlaubte und nicht erlaubte Intervalle Z von `acosh(Z)`.

Mit $Z = [1 - 2^{-70}, 1 - 2^{-70}] + i \cdot [2^{-1073741824}, 2^{-1073741824}]$ erhält man mit der Current-Präzision `prec` = 70 die Einschließung

$$\begin{aligned}\texttt{acosh(Z)} \;=\; &([5.7886806458960773528 6e - 323228487, 5.7886806458960773528 9e - 323228487],\\ &[4.1159031748919952916 8e - 11, 4.1159031748919952917 1e - 11]).\end{aligned}$$

Mit $Z = [1, 2] + i \cdot [-1, 0]$ erhält man mit der Current-Präzision `prec` = 150 die Einschließung

$$\begin{aligned}\texttt{acosh(Z)} \;=\; &([0, 1.4693517443681852732558443173616476167878033573347881 8],\\ &[-9.0455689430238136412731679566195872143109456096160507 0e - 1, 0]).\end{aligned}$$

C.2.26. 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.

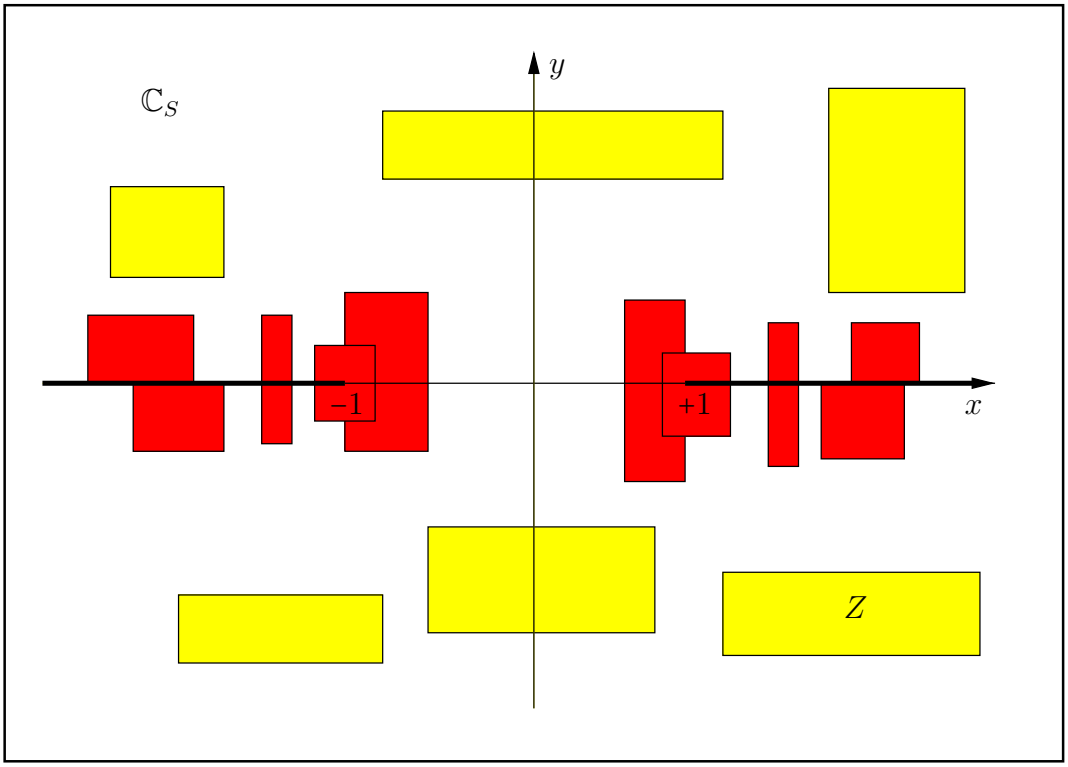


Abbildung C.45.: Erlaubte und nicht erlaubte Intervalle Z von `atanh(Z)`.

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.

C.2.27. 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.

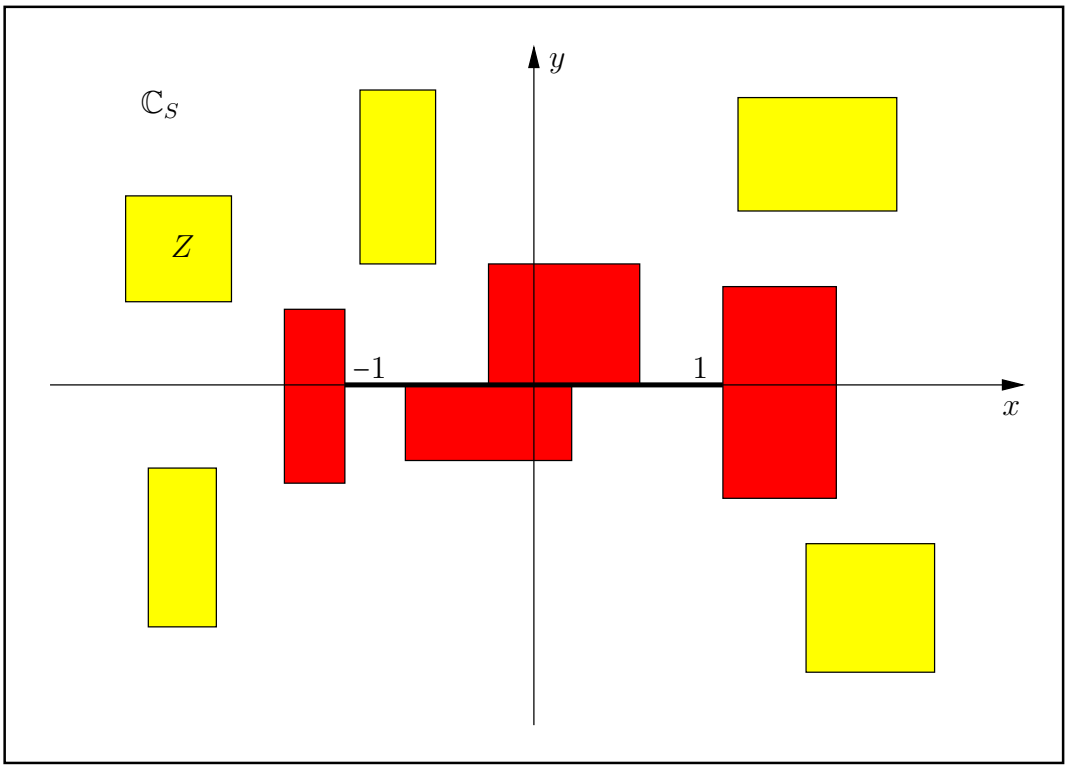


Abbildung C.46.: Erlaubte und nicht erlaubte Intervalle Z von `acoth(Z)`.

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.

C.2.28. 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

$$z^p := e^{p \cdot \ln(z)} = e^{p \cdot \ln|z|} \cdot e^{i \cdot p(\varphi + 2\pi k)}, \quad k \in \mathbb{Z}. \tag{C.175}$$

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

$$T := \{y \in \mathbb{C} \,|\, y = e^{p \cdot \ln|z|} \cdot e^{i \cdot p(\varphi + 2\pi k)}, \quad k \in \mathbb{Z},\ z \in \mathtt{Z},\ p \in \mathtt{P} = [p_1, p_2]\}.$$

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.

$$r_1 = e^{\mathtt{Inf(P \ln |Z|)}}, \quad r_2 = e^{\mathtt{Sup(P \ln |Z|)}}.$$

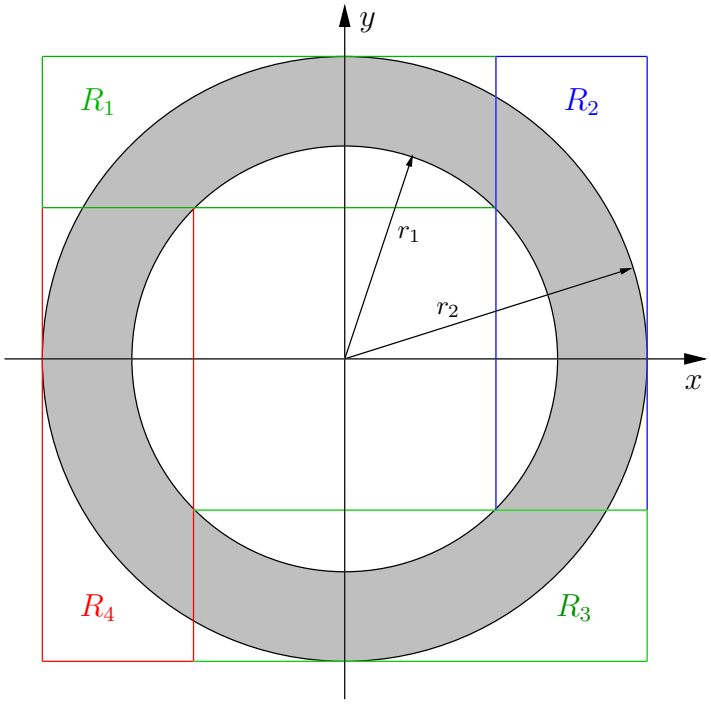


Abbildung C.47.: Den Kreisring einschließende Rechtecke R_ν, $\nu = 1, 2, 3, 4$.

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 Potenzen:
  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 (C.175) 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: $0 \in \mathtt{Z},\ p_1 \leq 0$.
Da 0^p für $p \leq 0$ 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 Imaginärteil zulassen, so wäre in (C.175) 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.2.29. $e^z - 1$

Wir betrachten die Aufgabe, zu einem vorgegebenen komplexen Intervall

$$Z = X + i \cdot Y, \quad X = [x_1, x_2],\ Y = [y_1, y_2]$$

eine möglichst optimale Einschließung der Menge $W := \{w \in \mathbb{C} \,|\, w = e^z - 1, \quad z \in Z\}$ durch ein komplexes Rechteck-Intervall $F \supseteq W$ mit der Präzision $\texttt{prec} \geq 2$ zu berechnen. Beachten Sie dabei jedoch, dass nach Seite 243 mit einem solchen einschließenden Rechteck $F = U + i \cdot V$ oft deutliche Überschätzungen nicht zu vermeiden sind.

C.2.29.1. Realteil

Das gesuchte Einschließungsintervall U für den Realteil von $e^Z - 1$ wird mit dem Maschinenintervall $\cos(Y) = [c_1, c_2]$ berechnet durch:

$$U := e^X \diamond \cos(Y) \ominus 1 = e^{[x_1,x_2]} \diamond [c_1, c_2] \ominus 1. \tag{C.176}$$

Da die Subtraktion der Eins zu Auslöschungsproblemen führen kann, lässt sich in Abhängigkeit der Vorzeichen von c_1, c_2 das Intervall U wie folgt berechnen:

$$U = \begin{cases} [e^{x_1+\ln(c_1)} - 1, e^{x_2+\ln(c_2)} - 1], & 0 \leq c_1 \leq c_2, \\ [c_1 \cdot e^{x_2} - 1, c_2 \cdot e^{x_1} - 1], & c_1 \leq c_2 \leq 0, \\ [c_1 \cdot e^{x_2} - 1, e^{x_2+\ln(c_2)} - 1], & c_1 \leq 0 \leq c_2. \end{cases} \tag{C.177}$$

Dabei ist klar, dass eine Auslöschung nur in den drei Fällen auftreten kann, in denen die Summanden $\ln(c_1)$ oder $\ln(c_2)$ im Exponenten vorkommen. Bei der Berechnung dieser Summanden tritt jedoch noch folgendes Problem auf, das im Fall $\ln(c_1)$ näher zu betrachten ist. Es gilt zunächst nach Definition:

$$\ln(c_1) := \ln(\texttt{Inf}(\cos(Y))),$$

und da die Logarithmusfunktion monoton wachsend ist, folgt zusätzlich

$$\ln(c_1) := \ln(\texttt{Inf}(\cos(Y))) = \texttt{Inf}(\texttt{ln\_cos}(Y)). \tag{C.178}$$

Wertet man jetzt die beiden Ausdrücke rechts in (C.178) auf der Maschine aus mit der Präzision $\texttt{prec} = 180$ und dem Punktintervall[20] $Y = [10^{-300}, 10^{-300}]$, so erhält man die deutlich verschiedenen Ergebnisse

$$\begin{aligned} \widetilde{\texttt{Inf}}(\texttt{ln\_cos}(Y)) &= -5.00000e - 601, & \texttt{prec} &= 180, \\ \widetilde{\ln}(\texttt{Inf}(\cos(Y))) &= -6.52530e - 55, & \texttt{prec} &= 180, \\ \widetilde{\ln}(\texttt{Inf}(\cos(Y))) &= -5.00000e - 601, & \texttt{prec} &= 2160. \end{aligned}$$

Erst bei 12-facher Current-Präzision, d.h. mit $\texttt{prec} = 2160$, erhält man für den optimalen Wert $\ln(c_1)$ mit dem Ausdruck $\widetilde{\ln}(\texttt{Inf}(\cos(Y)))$ das exakt Ergebnis $-5 \cdot 10^{-601}$, so dass aus Laufzeitgründen $\alpha := \texttt{Inf}(\texttt{ln\_cos}(Y))$ auszuwerten ist! Entsprechendes gilt für[21] $\beta := \texttt{Sup}(\texttt{ln\_cos}(Y))$. Damit wird der Realteil viel besser eingeschlossen durch die Maschinenauswertung von

$$U = \begin{cases} [e^{x_1+\alpha} - 1, e^{x_2+\beta} - 1], & 0 \leq c_1 \leq c_2, \\ [c_1 \cdot e^{x_2} - 1, c_2 \cdot e^{x_1} - 1], & c_1 \leq c_2 \leq 0, \\ [c_1 \cdot e^{x_2} - 1, e^{x_2+\beta} - 1], & c_1 \leq 0 \leq c_2. \end{cases} \tag{C.179}$$

[20] Mit 10^{-300} ist hier die zu 10^{-300} nächstgelegene IEEE-Maschinenzahl gemeint.
[21] `ln_cos`(...) ist die bereits in `mpfi.cpp` definierte Intervallfunktion zur Einschließung von $\ln(\cos(Y))$.

Bei der Auswertung der beiden Summen $x_1+\alpha$ und $x_2+\beta$ kann bei jeweils entgegengesetzten Vorzeichen beider Summanden im Exponenten erhebliche Auslöschung auftreten. In diesen Fällen ist die Präzision so lange zu verdoppeln, bis für die jeweilige Summe, also z.B. für $s := x_1 + \alpha$, die durch `prec` vorgegebene Genauigkeit erreicht wird. Um für $e^{x_1+\alpha} - 1$ den optimal abgerundeten Funktionswert zu erhalten, muss vorher natürlich, wie oben beschrieben, die **abgerundete** Summe $x_1 + \alpha$ berechnet worden sein. Anschließend wird dann mit der schon implementierten Punktfunktion `expm1(`s`,RoundDown)` der gesuchte, optimal abgerundete Wert von $e^{x_1+\alpha} - 1$ berechnet. Ganz entsprechend wird der aufzurundende Ausdruck $e^{x_2+\beta} - 1$ ausgewertet. Weitere Einzelheiten findet man in den Funktionen

```
MPFR::MpfrClass expm1_H1(const MPFR::MpfrClass& x, const MpfciClass& z,
                                                bool& Ausl, RoundingMode rnd);
MPFR::MpfrClass expm1_H(const MPFR::MpfrClass& x, const MpfciClass& z,
                                                          RoundingMode rnd);
```

die in `mpfciclass.cpp` definiert sind.

C.2.29.2. Imaginärteil

Da bei der Berechnung des Imaginärteils von $e^Z - 1$ keine Auslöschung auftritt, kann das Einschließungsintervall V intervallmäßig direkt und optimal berechnet werden:

$$V := e^X \diamond \sin(Y), \tag{C.180}$$

wobei jetzt e^X und $\sin(Y)$ die intervallmäßig berechneten Funktionswerte bedeuten.

C.2.29.3. Numerische Ergebnisse

Wir beginnen mit zwei Beispielen, in denen bei der Berechnung der Summen $x_1 + \alpha$ und $x_2 + \beta$ keine Auslöschung auftreten kann.

Im **1. Beispiel** wählen wir $x_1 = x_2 = y_1 = y_2 = 2^{-1\,000\,000\,000}$. Damit haben die Summanden x_1, α bzw. x_2, β zwar unterschiedliche Vorzeichen, ihre Beträge sind jedoch so unterschiedlich, dass eine Auslöschung nicht eintreten kann. Beachten Sie jedoch, dass bei einer naiven Berechnung von z.B $e^{x_1+\alpha} - 1$ wegen Auslöschung bei der Differenzberechnung sehr starke Überschätzungen auftreten würden, die jedoch durch Anwendung der `expm1`-Funktion bei dieser Implementierung vermieden wird. Mit `prec` = 180 und $Z = [x_1, x_2] + i \cdot [y_1, y_2]$ erhalten wir die Einschließung

$$\begin{aligned} e^Z - 1 \subseteq ([&2.1677979676169...8690 \cdot 10^{-301029996}, 2.1677979676169...8692 \cdot 10^{-301029996}], \\ &[2.1677979676169...8690 \cdot 10^{-301029996}, 2.1677979676169...8692 \cdot 10^{-301029996}]). \end{aligned}$$

Beachten Sie, wie dicht diese fast optimale Einschließung von $e^Z - 1$ am Ursprung liegt.

Das **2. Beispiel** zeigt, dass Z kein Punktintervall sein muss. Wir wählen $x_1 = y_1 = 2^{-1\,000\,000\,000}$ und $x_2 = y_2 = 2 \cdot x_1$. Mit `prec` = 180 und $Z = [x_1, x_2] + i \cdot [y_1, y_2]$ erhalten wir die Einschließung

$$\begin{aligned} e^Z - 1 \subseteq ([&2.1677979676169...8690 \cdot 10^{-301029996}, 4.3355959352338...7383 \cdot 10^{-301029996}], \\ &[2.1677979676169...8690 \cdot 10^{-301029996}, 4.3355959352338...7383 \cdot 10^{-301029996}]). \end{aligned}$$

Auch jetzt wird das Einschließungsintervall fast optimal berechnet und liegt ebenfalls sehr dicht am Ursprung.

Im **3. Beispiel** wählen wir zunächst $y_1 = y_2 = 0.75$. Um starke Auslöschung bei der Berechnung von z.B. $x_1 + \alpha$ zu konstruieren, berechnen wir mit $Y = [y_1, y_2]$ und `prec` = 1800 zunächst eine Einschließung T von $-\ln(\cos(Y)) \subseteq T$ und wählen mit $x_1 = x_2 = \texttt{mid}(T)$ das Punktintervall $X = [x_1, x_2]$. Wegen $\texttt{mid}(T) \approx -\alpha := \texttt{Inf}(\texttt{ln\_cos}(Y))$ ist daher bei der Berechnung der Summe $x_1 + \alpha$ erhebliche Auslöschung zu erwarten, vgl. (C.179) auf Seite 370. Für $e^Z - 1$ erhalten wir die Einschließung

$$e^Z - 1 \subseteq ([-2.39983652734278649...3587 \cdot 10^{-544}, -2.39983652734278649...3586 \cdot 10^{-544}], \\ [9.31596459944072461165...0253 \cdot 10^{-1}, 9.31596459944072461165...0255 \cdot 10^{-1}]).$$

Auch diese Einschließung ist mit 541 korrekten Dezimalziffern nahezu optimal.

Im **4. Beispiel** konstruieren wir ganz analog zum 3. Beispiel noch stärkere Auslöschung und wählen $y_1 = y_2 = 2^{-40\,000}$. Mit `prec` = 1800 erhalten wir dann für $e^Z - 1$ nach etwa einer Minute die Einschließung

$$e^Z - 1 \subseteq ([-1.322855833975488484...2062 \cdot 10^{-48166}, -1.322855833975488484...2061 \cdot 10^{-48166}], \\ [6.3120937524672703508...5709 \cdot 10^{-12042}, 6.3120937524672703508...5710 \cdot 10^{-12042}]).$$

Auch diese Einschließung ist mit 541 korrekten Dezimalziffern nahezu optimal.

Im **5. Beispiel** bestätigen wir die bekannte Gleichung

$$e^{i \cdot 2\pi} - 1 = 0 \tag{C.181}$$

und wählen dazu $x_1 = x_2 = 0$. Mit der Anweisung

```
MpfiClass Y = 2*MPFI::MpfiClass::Pi();
```

und `prec` = 1800 schließt $Y = [y_1, y_2]$ den Wert 2π optimal ein, und für $e^{i \cdot 2\pi} - 1$ erhalten wir die Einschließung

$$e^{i \cdot 2\pi} - 1 \in ([-4.22415201915475110296024022873 27...0180 \cdot 10^{-1083}, 0], \\ [-9.191465627586006593...6038 \cdot 10^{-542}, 2.005434173965873156...9529 \cdot 10^{-542}]).$$

Mit wachsender Präzision `prec` wird $e^{i \cdot 2\pi} - 1 = 0$ immer enger eingeschlossen, wobei zu beachten ist, dass die optimale Einschließung $([0,0],[0,0])$ natürlich nicht erreicht werden kann, da 2π auf der Maschine **nicht** darstellbar ist. Mit `prec` = 5 000 000 erhält man nach einigen Minuten die sehr enge Einschließung

$$e^{i \cdot 2\pi} - 1 \in ([-2.93976...10^{-3010299}, 0], [-7.41700...10^{-1505151}, 7.66780...10^{-1505150}]).$$

Die größere Laufzeit wird verursacht durch die optimale Einschließung von 2π mit ca. 1 505 149 Dezimalstellen. Beachten Sie, dass das Infimum des Realteils im Vergleich zum Imaginärteil fast den doppelten negativen Zehnerexponenten erhält und dass das Supremum 0 sogar exakt berechnet wird.

D. Einschließung reeller und komplexer Ausdrücke

D.1. Einschließung reeller arithmetischer Ausdrücke

D.1.1. Problemstellung und Definitionen

In einem *reellen arithmetischen Ausdruck* sind per Definition reelle Konstanten und Funktionen aus Tabelle 3.11.3 mit den vier arithmetischen Grundoperationen verknüpft. Ist dann die reelle Funktion $f : D_f \to \mathbb{R}$ ein solcher reeller arithmetischen Ausdruck und ist $X = [x_1, x_2] \subseteq D_f \subseteq \mathbb{R}$ ein vorgegebenes Intervall, so ist

$$W_{f,X} := \{f(x) \,\big|\, x \in X \subset D_f\} \tag{D.1}$$

die Wertemenge $W_{f,X}$ von f über dem reellen Intervall $X \subset D_f$. Ist f über dem abgeschlossenen Intervall $X \subset D_f$ stetig, so gibt es Zahlen $\xi_1, \xi_2 \in X$, so dass gilt $W_{f,X} = f(X) := [f(\xi_1), f(\xi_2)]$, d.h. die Wertemenge ist ein endliches Intervall. Beachten Sie, dass die Stetigkeit von $f(x)$ über $X \subset D_f$ für arithmetische Ausdrücke mit Funktionen aus Tabelle 3.11.3 keine einschränkende Bedingung ist.

Bei der Intervallrechnung verfolgt man u.a. das Ziel, zu einer Funktion $f(x)$ und zu einem gegebenen Maschinenintervall $X \subset D_f$ mit einem möglichst einfachen Algorithmus die Wertemenge $W_{f,X}$ durch ein Maschinenintervall $I(X)$ garantiert einzuschließen, d.h. es soll gelten $W_{f,X} \subseteq I(X)$. Nach [6, Seite 31] erhält man eine solche Maschineneinschließung $I(X)$, wenn der Funktionsterm $f(x)$ intervallmäßig ausgewertet wird, d.h. wenn die auftretenden Variablen x jeweils durch X und die Grundoperationen durch ihre entsprechenden Intervalloperationen ersetzt werden. Wir betrachten dazu das folgende Beispiel:

$f(x) = 3x \cdot e^{-2x+1}$, $X = [0,2]$, mit: $W_{f,X} = [0, 3/2]$, und die intervallmäßige Auswertung liefert

$$I(X) = 3 \diamond X \diamond \texttt{exp}(-2 \diamond X \diamond 1) = [0, 16.309690\ldots] \supset W_{f,X}.$$

Die intervallmäßige Auswertung liefert bei diesem Beispiel mit $I(X)$ eine erhebliche Überschätzung des Wertebereichs $W_{f,X} = [0, 3/2]$, da der Funktionsterm $f(x)$ die Variable x zweimal enthält und weil die Intervallrechnung grundsätzlich annimmt, dass, im Gegensatz zu unserem Beispiel, die auftretenden Intervalle X beide voneinander unabhängig sind. Das Ziel ist also, $I(X) = [I_1, I_2]$ so zu verbessern, dass im Idealfall $I(X) = f(X) := [f(\xi_1), f(\xi_2)]$ erfüllt ist. Beachten Sie aber, dass $I(X) = f(X)$ nur gelten kann, wenn $f(\xi_1)$ und $f(\xi_2)$ beides Maschinenzahlen sind. Nach Definition verlangen wir daher für eine **optimale** Einschließung des Wertebereichs $W_{f,X} = f(X) := [f(\xi_1), f(\xi_2)]$ durch das Maschinenintervall $I(X) = [I_1, I_2]$ die beiden folgenden Doppelungleichungen:

$$I_1 \le f(\xi_1) < \texttt{succ}(I_1), \qquad \texttt{pred}(I_2) < f(\xi_2) \le I_2, \quad \xi_1, \xi_2 \in X. \tag{D.2}$$

Im folgenden Abschnitt wird gezeigt, wie man die durch intervallmäßige Auswertung berechnete Einschließung $I(X)$ eines Wertebereichs weiter so verbessern kann, dass die verbesserte Einschließung dann **optimal** oder nahezu **optimal** ist. Als nahezu optimal bezeichnen wir eine Einschließung, wenn z.B. gilt $I_1 \le f(\xi_1) < \texttt{succ(succ}(I_1))$.

D.1.2. Verbesserung einer Einschließung

Eine erste Methode zur Verbesserung der Einschließung des Wertebereichs $W_{f,X}$ wird realisiert durch Anwendung der *Zentrischen Form* oder der *Mittelwertform*, siehe z.B. [5], [56], [58]. Da diese Methoden jedoch nur bei hinreichend schmalen Intervallen X wirklich hilfreich sind, wird hier nicht näher darauf eingegangen. Eine zweite Methode basiert auf dem folgenden Satz[1], [6, Seite 32]

Satz D.1 (Eine Variable tritt explizit nur einmal auf).
Ist $f(x)$ ein reeller, arithmetischer Ausdruck und tritt in ihm die Variable x explizit nur einmal auf, so liefert die intervallmäßige Auswertung von $f(x)$ über dem Intervall $X \subset D_f$ eine optimale Einschließung des Wertebereichs $W_{f,X}$, d.h. es gilt: $\quad W_{f,X} = [f(\xi_1), f(\xi_2)],\ \xi_1, \xi_2 \in X.$

Eine unmittelbare Anwendung dieses Satzes besteht darin, in einem arithmetischen Ausdruck $f(x)$ die Anzahl der auftretenden Variablen x möglichst zu minimieren. Als Beispiel betrachten wir die beiden äquivalenten Ausdrücke

$$f(x) = \frac{1+x}{x}, \qquad g(x) = 1 + \frac{1}{x}, \quad \text{mit } X = [1,4].$$

Durch intervallmäßige Auswertung erhalten wir die Ergebnisse

$$I_f(X) := (1 \oplus X) \oslash X = [0.5, 5]; \qquad I_g(X) := 1 \oplus 1 \oslash X = [1.25, 2] = W_{g,X} \subset I_f(X),$$

wobei $I_g(X)$ sogar die optimale Einschließung des Wertebereichs von $f(x) \equiv g(x)$ über X ist, da $g(x)$ die Variable x nur einmal enthält.

Eine zweite Anwendung von Satz D.1 ist ein quadratischer Ausdruck der Form

$$f(x) := x^2 + a \cdot x + b = (x + a/2)^2 - a^2/4 + b =: g(x), \quad x, a, b \in \mathbb{R}.$$

Wertet man jetzt $g(x)$ intervallmäßig aus und benutzt dabei z.B. $\texttt{sqr}(X \oplus a \oslash 2)$ für den ersten Summanden $(x+a/2)^2$, so erhält man mit $I_g(X)$ eine optimale Einschließung des Wertebereichs $W_{f,X}$ über einem Intervall X, wenn man zur Vermeidung einer möglichen Auslöschung $g(x)$ insgesamt in hinreichend hoher Präzision auswertet. Beachten Sie dabei, dass man bei der intervallmäßigen Auswertung von $g(x)$ für b ein beliebiges aber endliches Intervall zulassen darf, während man für eine optimale Einschließung a nur durch ein Punktintervall ersetzen darf. Als Beispiel wählen wir

$$f(x) := x^2 - 4 \cdot x - 4 = (x-2)^2 - 8 =: g(x), \quad X = [1,4].$$

Durch intervallmäßige Auswertung erhalten wir die Ergebnisse

$$I_f(X) := \texttt{sqr}(X) \ominus 4 \odot X \ominus 4 = [-19, +8]; \quad I_g(X) := \texttt{sqr}(X \ominus 2) \ominus 8 = [-8, -4] \subset I_f(X),$$

wobei $I_g(X)$ sogar die optimale Einschließung des Wertebereichs von $f(x) \equiv g(x)$ über X ist, da $g(x)$ die Variable x nur einmal enthält. Vergleichen Sie auch den Abschnitt B.5 auf Seite 226.

D.1.3. Optimale Einschließungen

Mit $f(x) = 3x \cdot e^{-2x+1}$ haben wir schon ein Beispiel kennengelernt, bei dem die intervallmäßige Auswertung über $X = [0,2]$ den Wertebereich $W_{f,X} = [0, 3/2]$ erheblich überschätzt. Bei einmal stetig differenzierbaren Funktionen $f(x)$ kann man diese Überschätzungen jedoch vermeiden, wenn man das Intervall $X = \bigcup_i X_i$ in Maschinenintervalle X_i unterteilt, in denen $f(x)$ jeweils monoton ist. Bedeutet dann $I_f(X_i)$ die Einschließung aller Funktionswerte $f(x)$, mit $x \in X_i$, so

[1] Der Satz D.1 gilt sinngemäß auch dann, wenn neben x auch noch weitere Varaiablen, z.B. $y, z, \ldots$, auftreten.

ist $I_{f,X} := \bigcup_i I_f(X_i)$ eine deutlich bessere Einschließung von $W_{f,X} \subseteq I_{f,X}$. Mit dem Teilintervall $X_i = [x_{i,1}, x_{i,2}]$ ist dann $I_f(X_i)$ wie folgt zu berechnen:

$$I_f(X_i) = [\texttt{Inf}(f([x_{i,1}, x_{i,1}])), \texttt{Sup}(f([x_{i,2}, x_{i,2}]))]. \tag{D.3}$$

Die Funktion $f(x)$ ist also jeweils intervallmäßig auf den Intervallrändern von $X_i = [x_{i,1}, x_{i,2}]$ auszuwerten. Um bei $f(x) = 3x \cdot e^{-2x+1}$ die Monotonieintervalle X_i zu bestimmen, benötigen wir die Nullstellen von $f'(x) = 3e^{-2x+1}(1-2x)$ und erhalten damit $X_1 = [0, 0.5]$, $X_2 = [0.5, 2]$ und mit diesen Intervallen nach (D.3) die Einschließungen $I_f(X_1) = [0, 1.5]$, $I_f(X_2) = [6 \cdot e^{-3}, 1.5]$, d.h. $I_{f,X} = I_f(X_1) \bigcup I_f(X_2) = [0, 3/2] = W_{f,X}$. Wesentlich komplizierter wird das Verfahren jedoch, wenn die Nullstellen von $f'(x)$ **keine** Maschinenzahlen sind. Ist dann z.B. $Z_1 = [\zeta_1, \zeta_2]$ eine Maschineneinschließung einer solchen Nullstelle z_1, so muss $f(x)$ wieder intervallmäßig über diesem Intervall Z ausgewertet werden, wodurch die schon bekannten Überschätzungen wieder auftreten können, und diese Überschätzungen machen sich besonders dann negativ bemerkbar, wenn die Nullstelle z_1 von $f'(x)$ entweder ganz in der Nähe einer Nullstelle von $f(x)$ liegt oder sogar mit dieser Nullstelle übereinstimmt, wie es z.B. schon bei $g(x) := (x - \sqrt{33})^2$ realisiert ist. Besonders aufwendig wird das Verfahren auch dann, wenn $f'(x)$ im Intervall X sehr viele Nullstellen besitzt, die natürlich alle möglichst eng eingeschlossen werden müssen. Das beschriebene Verfahren liefert also nur bei verhältnismäßig einfachen Funktionen eine wirklich optimale Einschließung ihres Wertebereichs.

Mit Hilfe der *Globalen Optimierung* kann der Wertebereich einer zweimal stetig differenzierbaren Funktion $f(x)$ ebenfalls garantiert eingeschlossen werden, [27], [28], [29], [31], [59]. Dabei wird eine Unter- bzw. Oberschranke des Wertebereichs $W_{f,X}$ einer Funktion $f(x)$ über einem vorgegebenen Intervall X berechnet. Vergleichen Sie dazu den Abschnitt 7.4 auf Seite 131.

Wir stellen uns jetzt dem Problem, wie man ausgehend von einer Funktion $f : D_f \to \mathbb{R}$, deren Wertebereich durch intervallmäßige Auswertung optimal eingeschlossen wird, neue Funktionen $g : D_g \to \mathbb{R}$ konstruieren kann, deren Wertebereiche $W_{g,X}$ über dem Intervall X dann ebenfalls optimal eingeschlossen werden. Wir formulieren dazu den recht einfachen

Satz D.2 (Multiplikation mit einer reellen Konstanten $a \in \mathbb{R}$).
Liefert $f : D_f \to \mathbb{R}$ bei intervallmäßiger Auswertung über dem Intervall X eine optimale Einschließung ihres Wertebereichs $W_{f,X}$, so liefert auch die Funktion $g(x) := a \cdot f(x)$ eine nahezu optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Beweis. Nach Voraussetzung gibt es nach Seite 373 die optimale Einschließung $I(X)$ des Wertebereichs $W_{f,X}$, d.h. es gilt $W_{f,X} \subseteq I(X)$ mit den Ungleichungen aus (D.2). Mit $g(x) := a \cdot f(x)$ gilt zusätzlich $W_{g,X} = a \cdot W_{f,X}$ und damit $a \cdot W_{f,X} \subseteq a \diamond I(X)$, wobei $a \diamond I(X)$ das Ergebnis der intervallmäßigen Auswertung von $g(x)$ über dem Intervall X ist. Da die Multiplikation von a mit dem Intervall $I(X)$ auf der Maschine optimal implementiert ist, wird $W_{f,X}$ durch $a \diamond I(X)$ optimal oder nahezu optimal eingeschlossen, wobei *nahezu optimal* bedeutet, dass bei der Maschinenmultiplikation $a \diamond I(X)$ nach außen gerundet wird, wobei im ungünstigsten Fall nach rechts und links je ein Bit Genauigkeit verloren geht ∎

Neben der Multiplikation kann man eine neue Funktion $g(x)$ auch durch die Addition einer reellen Zahl b gewinnen, dies beschreibt der

Satz D.3 (Addition einer reellen Konstanten $b \in \mathbb{R}$).
Liefert $f : D_f \to \mathbb{R}$ bei intervallmäßiger Auswertung über dem Intervall X eine optimale Einschließung ihres Wertebereichs $W_{f,X}$, so liefert auch die Funktion $g(x) := f(x) + b$ eine nahezu optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Beweis. Der einfache Beweis bleibt dem Leser überlassen.

Die beiden letzten Sätze lassen sich zusammenfassen zum

Satz D.4 (Multiplikation oder Addition einer reellen Konstanten).
Liefert $f : D_f \to \mathbb{R}$ bei intervallmäßiger Auswertung über dem Intervall X eine optimale Einschließung ihres Wertebereichs $W_{f,X}$, so liefert auch die Funktion $g(x) := a \cdot f(x) + b$ eine nahezu optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Als Beispiel betrachten wir die Exponentialfunktion $f(x) = e^x$, deren Wertebereich mit Hilfe der Intervallfunktion $\texttt{exp}(X)$ über dem Maschinenintervall X optimal eingeschlossen wird. Nach Satz D.4 wird dann auch der Wertebereich $W_{g,X}$ der Funktion $g(x) := \sqrt{2} \cdot e^x + \sqrt{3}$ mit Hilfe des Intervallausdrucks

$$\texttt{sqrt}(2) \diamond \texttt{exp}(X) \oplus \texttt{sqrt}(3) \supseteq W_{g,X}$$

nahezu optimal eingeschlossen, wenn mit $\texttt{sqrt}(2)$ und $\texttt{sqrt}(3)$ die optimalen Einschließungen von $\sqrt{2}$ bzw. $\sqrt{3}$ bezeichnet werden, wobei $\texttt{sqrt}(2)$ und $\texttt{sqrt}(3)$ natürlich keine Punktintervalle sein können. Deshalb werden außer durch die Intervall-Multiplikation und Addition auch noch zusätzliche aber nur minimale Überschätzungen durch die Intervalle $\texttt{sqrt}(2)$ und $\texttt{sqrt}(3)$ verursacht, so dass die Einschließung von $W_{g,X}$ nur *nahezu optimal* sein kann.

Ausgehend von einer Funktion $f : D_f \to \mathbb{R}$, die bei intervallmäßiger Auswertung über einem Maschinenintervall $T \subset D_f$ eine optimale Einschließung ihres Wertebereichs $W_{f,T}$ liefert, kann man mit einer Transformationsfunktion $t : D_t \to \mathbb{R}$ neue Funktionen $g(x) := f(t(x))$ gewinnen, wenn der Wertebereich $W_{t,X}$ bez. des Maschinenintervalls $X \subset D_t$ selbst wieder ein Intervall T ist, so dass gilt $W_{t,X} = T \subset D_f$. Diese neue Funktion $g(x) := f(t(x))$ liefert dann bei intervallmäßiger Auswertung über dem Intervall X eine nahezu optimale Einschließung ihres Wertebereichs $W_{g,X}$, wenn auch die intervallmäßige Auswertung von $t(x)$ über dem Intervall X den Wertebereich $W_{t,X}$ optimal einschließt. Wir formulieren dies im

Satz D.5 (Transformationsfunktion).
Die reelle Funktion $f : D_f \to \mathbb{R}$ liefert nach Voraussetzung bei intervallmäßiger Auswertung über dem Intervall $T \subset D_f$ eine optimale Einschließung ihres Wertebereichs $W_{f,X}$ und $t : D_t \to \mathbb{R}$ besitzt einen Wertebereich $W_{t,X}$ über dem Intervall $X \subset D_t$, der selbst wieder ein Intervall $T := W_{t,X} \subset D_f$ ist. Dann wird bei intervallmäßiger Auswertung der neuen Funktion $g(x) := f(t(x))$ ihr Wertebereich $W_{g,X}$ auch wieder nahezu optimal eingeschlossen, wenn bei intervallmäßiger Auswertung auch der Wertebereich $T := W_{t,X}$ optimal oder nahezu optimal eingeschlossen wird.

Beweis. Nach Voraussetzung ist $W_{t,X}$ selbst ein **Intervall**, das durch das Intervall $T \subset D_f$ optimal oder nahezu optimal eingeschlossen wird und nach Voraussetzung liefert auch die intervallmäßige Auswertung von $f(x)$ über dem Intervall T selbst wieder eine optimale Einschließung ihres Wertebereich $W_{f,T}$, so dass $W_{g,X}$ durch die intervallmäßige Auswertung von $g(x) = f(g(x))$ nahezu optimal ausgewertet wird.

Als Beispiel wählen wir $f(x) = e^x$ und als Transformationsfunktion $t(x) = \sin(x) + 2$ mit dem beliebigen Maschinenintervall X. Damit sind die Voraussetzungen des Satzes D.5 erfüllt, und mit der intervallmäßigen Auswertung von $g(x) = e^{\sin(x)+2}$ erhalten wir über jedem Intervall X eine nahezu optimale Einschließung des Wertebereichs $W_{g,X}$.

Beachten Sie, dass man mit den Intervallfunktionen aus Tabelle 4.14.3 auf Seite 68 eine Vielzahl von Transformationsfunktion $t(x)$ konstruieren kann. Beachten Sie aber auch, dass z.B. $t(x) = \sin(x) + x$ die Voraussetzungen von Satz D.5 nicht erfüllt, da $t(x)$ die Variable x zweimal enthält, so dass $W_{t,X}$ jetzt **keinesfalls** optimal eingeschlossen werden kann.

D.2. Einschließung komplexer arithmetischer Ausdrücke

D.2.1. Problemstellung und Definitionen

In einem *komplexen arithmetischen Ausdruck* sind per Definition konstante komplexe Zahlen und Funktionen aus der Tabelle 6.12.3 mit den vier arithmetischen Grundoperationen verknüpft. Die komplexe Intervall-Variable $Z = X + i \cdot Y$ ist in der komplexen Ebene ein Rechteck, wobei $X = [x_1, x_2]$ und $Y = [y_1, y_2]$ reelle Intervalle sind, vgl. Seite 243.

Ist $f : \mathbb{C} \to \mathbb{C}$ eine komplexwertige Funktion, mit z.B. $f(z) = \sin(z),\ z = x + i \cdot y \in Z \subset \mathbb{C}$, so ist

$$W_{f,Z} := \{ f(z) \,|\, z \in Z \subset \mathbb{C} \} \tag{D.4}$$

die Wertemenge $W_{f,Z}$ von f über dem komplexen Intervall Z.

Bei der komplexen Intervallrechnung ist es nun das Ziel, zu einer Funktion $f(z)$ und zu einem vorgegebenen Intervall Z mit einem geeigneten Algorithmus die Wertemenge $W_{f,Z}$ durch ein Rechteck $F(Z)$ wenigstens **optimal** einzuschließen, d.h. es muss gelten $W_{f,Z} \subseteq F(Z)$, und jede Seite von $F(Z)$ muss mit wenigstens einem Punkt der Menge $W_{f,Z}$ übereinstimmen, vgl. Seite 243. Wir **definieren:**

Ist $f : \mathbb{C} \to \mathbb{C}$, mit $f(z) = u(x,y) + i \cdot v(x,y)$, über dem Rechteck $Z = X + i \cdot Y$ eine holomorphe Funktion, dann ist $F(Z)$ eine optimale Einschließung des Wertebereichs $W_{f,Z}$ $\iff$

$$\begin{array}{ll} \exists x_{u,m} \in X, \exists y_{u,m} \in Y,\ u_m := u(x_{u,m}, y_{u,m}); & \exists x_{u,M} \in X, \exists y_{u,M} \in Y,\ u_M := u(x_{u,M}, y_{u,M}); \\ \exists x_{v,m} \in X, \exists y_{v,m} \in Y,\ v_m := v(x_{v,m}, y_{v,m}); & \exists x_{v,M} \in X, \exists y_{v,M} \in Y,\ v_M := v(x_{v,M}, y_{v,M}); \\ \Longrightarrow \qquad \forall x \in X \ \wedge\ \forall y \in Y \quad \text{gilt:} & u_m \le u(x,y) \le u_M, \quad v_m \le v(x,y) \le v_M. \end{array}$$

Anmerkung: Die Punkte $(x_{u,m}, y_{u,m})$ und $(x_{u,M}, y_{u,M})$ sind die Extremalpunkte m, M von $u(x,y)$ auf dem Rand von Z. Entsprechendes gilt für die Punkte $(x_{v,m}, y_{v,m})$ und $(x_{v,M}, y_{v,M})$.

Mit Ausnahme der Funktionen `power_fast` und `pow_all` liefern alle Funktionen aus Tabelle 6.12.3 eine **optimale** Einschließung $F(Z) \supseteq W_{f,Z}$. Gilt sogar $W_{f,Z} = F(Z)$, so nennt man die Einschließung $F(Z)$ **exakt**, wobei die exakten Einschließungen nur bei einfachen Funktionen, wie z.B. $f(z) = a \cdot z + c,\ a \in \mathbb{R},\ c \in \mathbb{C}$ auftreten.

Wertet man einen arithmetischen Ausdruck, z.B. $f(z) = z/(1+z)$, intervallmäßig aus, indem man die Variable z durch ein vorgegebenes Intervall Z ersetzt, so erhält man mit dem Ergebnis

$$f(Z) := Z \diamondsuit (1 \oplus Z) \tag{D.5}$$

eine Einschließung von $W_{f,Z}$, d.h. es gilt: $W_{f,Z} \subset f(Z)$, aber es ist keinesfalls sicher, dass $f(Z)$ auch eine optimale Einschließung von $W_{f,Z}$ ist. Im folgenden Abschnitt soll untersucht werden, wie z.B der Ausdruck $f(z) = z/(1+z)$ umgeformt werden kann, um die Einschließung weiter zu verbessern.

Fast alle Funktionen aus Tabelle 6.12.3 liefern bei intervallmäßiger Auswertung eine **optimale** Einschließung $F(Z)$ der Wertemenge $W_{f,Z} := \{ f(z) \,|\, z \in Z \subset \mathbb{C} \}$. Für die praktische Anwendung wäre es interessant zu wissen, mit welchen Transformationen, z.B. mit $z \to 1/z$, man ausgehend von $f(z)$ eine neue Funktion $g(z) := f(1/z)$ erhält, die ebenfalls mit $g(Z)$ eine optimale Einschließung des neuen Wertebereichs $W_{g,Z}$ liefert. Als Beispiel betrachten wir $f(z) = z$, und mit $z \to 1/z$ erhält man $g(z) = 1/z$, so dass sowohl $f(Z) = Z$ und $g(Z) = 1 \diamondsuit Z = \texttt{reci}(Z)$ jeweils optimale Einschließungen liefern. Bei $f(Z) = Z$ ist dies wegen der identischen Abbildung klar, und bei $g(Z) = \texttt{reci}(Z)$ liefert $\texttt{reci}(Z)$ nach Tabelle 6.12.3 automatisch eine optimale Einschließung, falls $0 \notin Z$. Beachten Sie, dass auch $g(Z) = 1 \diamondsuit Z$ eine optimale Einschließung liefert, da die direkte Intervalldivision als eine der vier Grundoperationen eine optimale Einschließung berechnet, vergleichen Sie dazu die Anmerkungen auf Seite 287.

Im nächsten Beispiel betrachten wir $f(z) = z^2$, und mit der gleichen Transformation $z \to 1/z$ erhalten wir $g(z) = f(1/z) = 1/z^2$. Mit $f(Z) = \texttt{sqr}(Z)$ erhält man eine optimale Einschließung,

aber $g(Z) = 1 \diamond \texttt{sqr}(Z)$ oder $g(Z) = \texttt{sqr}(1 \diamond Z)$ liefern nach Seite 288 i.a. **keine** optimale Einschließung. Erst mit $G(Z) := \texttt{reci\_z2}(Z)$ wird der Wertebereich $W_{g,Z}$ optimal eingeschlossen. Die Transformation $z \to 1/z$ liefert also i.a. keine optimale Einschließung!

Im Abschnitt D.2.3 werden die Bedingungen für optimale Einschließungen genauer untersucht.

D.2.2. Verbesserung einer Einschließung

In diesem Abschnitt wird untersucht, unter welchen Gesichtspunkten die durch intervallmäßige Auswertung gewonnene garantierte Einschließung $f(Z)$ eines gegebenen arithmetischen Ausdrucks $f(z)$, $z \in Z \subset \mathbb{C}$, noch weiter verbessert werden kann. In nachfolgender Tabelle sind dazu für $Z = [2,5] + i \cdot [1,3]$ die durch intervallmäßige Auswertung berechneten Einschließungen $g(Z) \subset f(Z)$ der jeweils identischen Ausdrücke $g(z) \equiv f(z)$, $z \in Z$, zusammengestellt.

$Z = X + i \cdot Y = [2,5] + i \cdot [1,3]$; Präzision: $\texttt{prec} = 15$.	
$f(z) = z/(1+z)$	$f(Z) = ([3.333282e-1, 1.800049], [-6.666871e-1, 7.000123e-1])$
$g(z) = 1/(1+1/z)$	$g(Z) = ([6.921691e-1, 8.708802e-1], [1.960563e-2, 1.814042e-1])$
$f(z) = (z+1)/z$	$f(Z) = ([5.293884e-1, 3.000000], [-1.270813, 6.213379e-1])$
$g(z) = 1 + 1/z$	$g(Z) = ([1.147033, 1.400025], [-2.500000e-1, -3.845977e-2])$
$f(z) = 1/(1/z)$	$f(Z) = ([1.747924, 6.364991], [2.381515e-1, 3.400147])$
$g(z) = z$	$g(Z) = ([2.000000, 5.000000], [1.000000, 3.000000])$
$f(z) = \frac{z^2-1}{z+1}$	$f(Z) = ([-1.400025, 1.013477e1], [-3.166749, 9.600098])$
$g(z) = z - 1$	$g(Z) = ([1.000000, 4.000000], [1.000000, 3.000000])$

Die oberen vier Beispiele zeigen wegen $g(Z) \subset f(Z)$, dass ein komplexer, arithmetischer Ausdruck so umgeformt werden sollte, dass die Variable z möglichst nur einmal vorkommt und die Zahl der arithmetische Operatoren minimal ist.

$Z = X + i \cdot Y = [2,5] + i \cdot [1,3]$; $a = [-2,-2] + i \cdot [2,2]$, $b = [2,2] - i \cdot [1,1]$;	
$f(z) = \texttt{sqr}(z) \cdot z + (\texttt{sqr}(z) + az + b)$	$f(Z) = ([-2.500e1, 1.100e1], [-2.700e1, 9.000])$
$g(z) = \texttt{sqr}(z) \cdot z + \texttt{poly2}(z,a,b)$	$g(Z) = ([-2.100e1, 7.000], [-2.300e1, 5.000])$

Das letzte Beispiel zeigt, dass ein Teilausdruck, der nach Tabelle 6.12.3 optimal eingeschlossen wird, durch die jeweilige Funktion dieser Tabelle, hier $\texttt{poly2}(z,a,b)$, ersetzt werden sollte. Wir fassen zusammen:

> Zur Verbesserung der Einschließung eines komplexen, arithmetischen Ausdrucks sollte dieser so äquivalent umgeformt werden, dass die Variable z möglichst nur einmal vorkommt und die Zahl der arithmetischen Operatoren minimal ist. Falls möglich sollten Teilausdrücke durch ihre gleichwertigen Funktionen aus Tabelle 6.12.3 mit optimaler Einschließung ersetzt werden. Die verbesserte Einschließung ist dann aber nicht notwendig optimal!

D.2.3. Optimale Einschließungen

In diesem Abschnitt betrachten wir die in einem Gebiet $G \subset \mathbb{C}$ holomorphen Funktionen $f(z)$ der Tabelle 6.12.3, die mit Ausnahme der beiden Funktionen `power_fast` und `pow_all` zu einem gegebenen Intervallargument $Z = X + i \cdot Y \subset G$ bei intervallmäßiger Auswertung mit $f(Z)$ eine

optimale Einschließung des Wertebereichs $W_{f,Z} := \{f(z) \,|\, z \in Z \subset G\}$ liefern und stellen jetzt die Frage, mit welchen Manipulationen man, ausgehend von $f(z)$, eine neue holomorphe Funktion $g(z)$ konstruieren kann, die bei intervallmäßiger Auswertung mit $g(Z)$ ebenfalls eine optimale Einschließung ihres Wertebereichs $W_{g,Z} := \{g(z) \,|\, z \in Z \subset G\}$ liefert.

Bevor wir jedoch zur Beantwortung dieser Frage kommen, soll zunächst noch geklärt werden, unter welchen Voraussetzungen garantiert ist, dass eine gegebene holomorphe Funktion $f(z) = u(x,y) + i \cdot v(x,y)$ mit $f(Z)$ eine optimale Einschließung von $W_{f,Z}$ liefert. Mit $z = x + i \cdot y \in Z \subset G$ gilt:

1. Wenn in den Funktionstermen[2] für $u(x,y)$ und $v(x,y)$ die Variablen x und y jeweils nur einmal vorkommen, so liefert $f(Z)$ eine optimale Einschließung von $W_{f,Z}$, [6, Seite 32]. Ein Beispiel ist die Exponentialfunktion
$$e^z = e^x \cdot \cos(y) + i \cdot e^x \cdot \sin(y), \quad x \in X,\ y \in Y.$$
2. Wenn man in der x,y-Ebene auf dem Rand eines beliebigen Rechtecks $Z \subset G$ für $u(x,y)$ und $v(x,y)$ jeweils die Maximum- und Minimum-Punkte M, m bestimmt hat, so kann man mithilfe der Funktionswerte von $u(x,y)$ und $v(x,y)$ an diesen Punkten für $f(z)$ eine optimale Einschließung von $W_{f,Z}$ berechnen, vgl. für $f(z) = 1/z^2$ die Seiten 210 und 216.

Wir kommen jetzt zur Beantwortung der schon angesprochenen Frage, wie man eine holomorphe Funktion $f(z)$ so manipulieren kann, dass die neue, ebenfalls holomorphe Funktion, $g(z)$ bei intervallmäßiger Auswertung auch eine optimale Einschließung von $W_{g,Z}$ liefert. Wir formulieren dazu den ersten, recht einfachen

Satz D.6 (Addition einer Konstanten $c = c_1 + i \cdot c_2 \in \mathbb{C}$).
Ist $f(z)$ holomorph in $Z = X + i \cdot Y \subset \mathbb{C}$ und liefert die intervallmäßige Auswertung von f eine optimale Einschließung des Wertebereichs $W_{f,Z}$, so liefert auch $g(z) := f(z) + c$ eine optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Beweis. Mit $f(z) = u(x,y) + i \cdot v(x,y)$ gilt: $g(z) = (u(x,y) + c_1) + i \cdot (v(x,y) + c_2)$, und nach Voraussetzung gibt es auf dem Rand von $Z \subset \mathbb{C}$ für $u(x,y)$ und $v(x,y)$ die entsprechenden Extremalpunkte m, M, so dass $\forall x \in X \ \wedge \ \forall y \in Y$ gilt:

$$u_m \le u(x,y) \le u_M, \quad v_m \le v(x,y) \le v_M, \quad \text{und daraus folgt für die gleichen Extremalpunkte}$$
$$u_m + c_1 \le u(x,y) + c_1 \le u_M + c_1, \qquad v_m + c_2 \le v(x,y) + c_2 \le v_M + c_2,$$

wobei zu beachten ist, dass in den obigen Ungleichungen das Gleichheitszeichen gilt, wenn die jeweiligen Extremalpunkte m, M auf dem Rand von Z gewählt werden, und nach Definition sind die beiden letzten Doppelungleichungen gerade die Bedingung für eine optimale Einschließung des Wertebereichs $W_{g,Z}$ ∎

Als **Anwendung** betrachten wir mit $c = 1 + i$ die Funktionen $f(z) = 1/z$ und $g(z) = 1/z + c$, die nach Satz D.6 zu einem beliebigen Rechteck $Z \not\ni 0$ bei intervallmäßiger Auswertung jeweils **optimale** Einschließungen ihrer verschiedenen Wertebereiche $W_{f,Z}$ bzw. $W_{g,Z}$ liefern. Beachten Sie jedoch, dass die äquivalente Funktion $h(z) = (1 + c \cdot z)/z \equiv g(z) = 1/z + c$ **keine optimale** Einschließung von $W_{h,Z} = W_{g,Z}$ mehr liefert!

Satz D.7 (Multiplikation mit einer reellen Konstanten $r \in \mathbb{R}$).
Ist $f(z)$ holomorph in $Z = X + i \cdot Y \subset \mathbb{C}$ und liefert die intervallmäßige Auswertung von f eine optimale Einschließung des Wertebereichs $W_{f,Z}$, so liefert auch $g(z) := r \cdot f(z)$ eine optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

[2] Die Funktionsterme für $u(x,y)$ und $v(x,y)$ können z.B. mit *Mathematica* und `ComplexExpand[]` berechnet werden.

Beweis. Mit $f(z) = u(x,y) + i \cdot v(x,y)$ gilt $g(z) = r \cdot u(x,y) + i \cdot r \cdot v(x,y)$. Nach Voraussetzung gibt es auf dem Rand von $Z \subset \mathbb{C}$ für $u(x,y)$ und $v(x,y)$ die entsprechenden Extremalpunkte m, M, so dass $\forall x \in X \ \wedge \ \forall y \in Y$ gilt:

$$u_m \le u(x,y) \le u_M, \quad v_m \le v(x,y) \le v_M, \quad \text{und daraus folgt für die gleichen Extremalpunkte}$$
$$r \cdot u_m \le r \cdot u(x,y) \le r \cdot u_M, \qquad r \cdot v_m \le r \cdot v(x,y) \le r \cdot v_M, \qquad \text{falls } r \ge 0,$$

wobei zu beachten ist, dass in den obigen Ungleichungen das Gleichheitszeichen gilt, wenn für x, y die Koordinaten der jeweiligen Extremalpunkte m, M auf dem Rand von Z gewählt werden, und nach Definition sind die beiden letzten Doppelungleichungen gerade die Bedingung für eine optimale Einschließung des Wertebereichs $W_{g,Z}$. Im Fall $r < 0$ erhält man ebenfalls mit den gleichen Extremalpunkten die beiden Doppelungleichungen

$$r \cdot u_M \le r \cdot u(x,y) \le r \cdot u_m, \qquad r \cdot v_M \le r \cdot v(x,y) \le r \cdot v_m, \qquad \text{falls } r \ge 0 \ \blacksquare$$

Satz D.8 (Multiplikation mit der imaginären Einheit $i = \sqrt{-1}$).
Ist $f(z)$ holomorph in $Z = X + i \cdot Y \subset \mathbb{C}$ und liefert die intervallmäßige Auswertung von f eine optimale Einschließung des Wertebereichs $W_{f,Z}$, so liefert auch $g(z) := i \cdot f(z)$ eine optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Beweis. Mit $f(z) = u(x,y) + i \cdot v(x,y)$ gilt: $g(z) = -v(x,y) + i \cdot u(x,y)$, und nach Voraussetzung gibt es auf dem Rand von $Z \subset \mathbb{C}$ für $u(x,y)$ und $v(x,y)$ die entsprechenden Extremalpunkte m, M, so dass $\forall x \in X \ \wedge \ \forall y \in Y$ gilt:

$$u_m \le u(x,y) \le u_M, \quad v_m \le v(x,y) \le v_M, \quad \text{und daraus folgt für die gleichen Extremalpunkte}$$
$$u_m \le u(x,y) \le u_M, \qquad -v_M \le -v(x,y) \le -v_m, \qquad \text{falls } r \ge 0,$$

wobei zu beachten ist, dass in den obigen Ungleichungen das Gleichheitszeichen gilt, wenn die jeweiligen Extremalpunkte m, M auf dem Rand von Z gewählt werden, und nach Definition sind die beiden letzten Doppelungleichungen gerade die Bedingung für eine optimale Einschließung des Wertebereichs $W_{g,Z}$ $\blacksquare$.

Man kann die ersten drei Sätze zusammenfassen:

Satz D.9 (Multiplikation mit $r \in \mathbb{R}$ oder mit $i \cdot r$ und anschließender Translation mit $c \in \mathbb{C}$).
Ist $f(z)$ holomorph in $Z = X + i \cdot Y \subset \mathbb{C}$ und liefert die intervallmäßige Auswertung von $f(z)$ eine optimale Einschließung des Wertebereichs $W_{f,Z}$, so liefern sowohl $g(z) := r \cdot f(z) + c$ als auch $h(z) := i \cdot r \cdot f(z) + c$ jeweils eine optimale Einschließung ihrer Wertebereiche $W_{g,Z}$ bzw. $W_{h,Z}$.

Mit $a \in \mathbb{C}$ und $b \in \mathrm{I}\mathbb{C}$ liefern daher nach Satz D.9 $f(z) = z^2 + az + b$, $g(z) = r \cdot (z^2 + az + b)$ und $h(z) = ir \cdot (z^2 + az + b)$ bei intervallmäßiger Auswertung jeweils eine **optimale** Einschließung ihrer Wertebereiche, vgl. die Tabelle auf Seite 115. Beachten Sie, dass die beiden Klammern () jeweils mit der Funktion `poly2(z,a,b)` auszuwerten sind und dass z.B. $r \cdot (z^2 + az + b)$ nicht ausmultipliziert werden darf. Ist beispielsweise der Ausdruck $rz^2 + \alpha z + \beta$ gegeben, so erhält man eine optimale Einschließung über einem beliebigen Rechteck $Z \subset \mathbb{C}$ nur dann, wenn man für $r \neq 0$ den äquivalenten Term $r \cdot (z^2 + \alpha/r + \beta/r)$ intervallmäßig auswertet, $\alpha \in \mathbb{C}$, $\beta \in \mathrm{I}\mathbb{C}$.

Bisher hatten wir durch Addition oder durch geeignete Multiplikation aus einer holomorphen Funktion $f(z)$, die bei intervallmäßiger Auswertung eine optimale Einschließung liefert, eine neue holomorphe Funktion $g(z)$ mit der gleichen Eigenschaft gewonnen.

Jetzt suchen wir zu einer holomorphen Funktion $f : D_f \to \mathbb{C}$ eine holomorphe Transformationsfunktion $t(z)$, mit der man eine neue Funktion $g(z) = f(t(z))$ gewinnt, die ebenfalls bei intervallmäßiger Auswertung eine optimale Einschließung liefert. Eine solche Funktion $t(z)$ müsste die Eigenschaft haben, ein achsenparalleles Rechteck $Z \subset D_f$ wieder in ein achsenparalleles Rechteck $Z_t \subset D_f$ abzubilden. Mit dem neuen Rechteck Z_t liefert dann $f(z)$ nach Voraussetzung bei intervallmäßiger Auswertung eine optimale Einschließung des Wertebereichs W_{f,Z_t}.

Eine geeignetes $t(z)$ ist die lineare Funktion

$$t(z) = \alpha \cdot z + c, \quad \text{mit } \alpha = r \in \mathbb{R} \text{ oder } \alpha = r \cdot i \text{ und } c \in \mathbb{C}, \tag{D.6}$$

und die neue Funktion ist gegeben durch $g(z) = f(t(z))$. Daraus ergibt sich der

Satz D.10.
Die Funktion $f : D_f \to \mathbb{C}$ sei holomorph in D_f und liefere bei intervallmäßige Auswertung über einem beliebigen Rechteck $Z \subset D_f$ eine optimale Einschließung ihres Wertebereichs $W_{f,Z}$. Ist dann $t : D_f \to D_f \subset \mathbb{C}$ definiert durch $t(z) = \alpha \cdot z + c$, mit $\alpha = r \in \mathbb{R}$ oder $\alpha = r \cdot i$ und $c \in \mathbb{C}$, so liefert auch $g(z) := f(t(z))$ bei intervallmäßiger Auswertung eine optimale Einschließung ihres Wertebereichs $W_{g,Z}$.

Anmerkungen:

1. $t(z)$ ist entweder eine Streckung mit anschließender Translation oder eine Drehstreckung um $\pm\pi$ mit anschließender Translation, so dass ein achsenparalleles Rechteck wieder in ein solches abgebildet wird. Die intervallmäßige Auswertung von $t(z)$ liefert daher mit $t(Z)$ nicht nur eine optimale sondern auch eine **exakte** Einschließung von $W_{t,Z}$.
   Mit $t(z) = 2iz + 1 + i$ und $f(z) = \sin(z)$ erhält man mit $g(z) = \sin(2iz + 1 + i)$ eine in ganz $\mathbb{C}$ holomorphe Funktion, die über einem beliebigen Rechteck $Z \subset \mathbb{C}$ bei intervallmäßiger Auswertung mit $g(Z)$ eine optimale Einschließung des Wertebereichs $W_{g,Z}$ liefert.

2. Definiert man $t(z) = \alpha \cdot z + c$ jetzt mit beliebigem $\alpha \in \mathbb{C}$, so ist die Transformation $t(z)$ wieder eine Drehstreckung mit anschließender Translation, aber der Wertebereich $W_{t,Z}$ von $t(z)$ ist jetzt ein beliebig gedrehtes und damit i.a. kein achsenparalleles Rechteck, so dass die intervallmäßige Auswertung $t(Z)$ zwar eine optimale aber **keine exakte** Einschließung von $W_{t,Z}$ liefert. Die intervallmäßige Auswertung von $g(z) := f(t(z))$ ergibt damit i.a. **keine optimale** Einschließung von $W_{g,Z}$.

3. Spiegelt man beispielsweise ein achsenparalleles Rechteck $Z = X + i \cdot Y$ an der 1. Winkelhalbierenden (Vertauschung von X, Y), so erhält man wieder ein achsenparalleles Rechteck Z_S, und wenn dies wieder in D_f liegt, so ist die Einschließung von W_{f,Z_S} auch über diesem Rechteck Z_S wieder optimal. Da aber $t(z)$ durch das Vertauschen von x und y keine holomorphe Funktion mehr ist, so ist auch die neue Funktion $g(z) := f(t(z))$ nicht mehr holomorph, und die Extrema von $g(z)$ müssen nicht mehr nur auf dem Rand von Z liegen, so dass Satz D.10 nicht mehr anwendbar ist.

Eine interessante **Anwendung** zur optimalen Einschließung ist die **Möbiustransformation**:

$$T(z) := \frac{az+b}{cz+d}; \qquad a,b,c,d,z \in \mathbb{C}, \quad ad-bc \neq 0. \tag{D.7}$$

$$T(z) = \begin{cases} \dfrac{a}{d}\cdot z + \dfrac{b}{d}, & c = 0 \\ \dfrac{az+b}{z+d}, & c \neq 0, \quad a,b,d,z \in \mathbb{C}, \quad ad-b \neq 0. \end{cases} \tag{D.8}$$

Den letzten Ausdruck in (D.8) findet man im Fall $c \neq 0$, wenn man in (D.7) Zähler und Nenner durch c dividiert und dann z.B. a/c wieder mit a bezeichnet. Der letzte Ausdruck in (D.8) hat bei intervallmäßiger Auswertung über Z für eine optimale Einschließung des Wertebereichs $W_{T,Z}$ jedoch den Nachtei, dass z im Zähler **und** im Nenner auftaucht. Dies wird jedoch vermieden, wenn man mithilfe der *Polynomdvision* wie folgt umformt

$$T(z) \equiv T_1(z) := \frac{b-ad}{z+d} + a, \quad a,b,d,z \in \mathbb{C}, \quad ad-b \neq 0. \tag{D.9}$$

Ist nun Z in der komplexen Ebene ein achsenparalleles Rechteck, welches den Punkt $z_0 = -d$ nicht enthält, so erhält man bei intervallmäßiger Auswertung des Bruchs in (D.9) eine **optimale** Einschließung seines Wertebereichs, da der Nenner **exakt** eingeschlossen wird und weil die komplexe Intervalldivision eine optimale Einschließung liefert, wenn Zähler und Nenner voneinander unabhängige Intervalle sind. Dies ist in unserem Fall gewährleistet, da a,b,d durch Punktintervalle einzuschließen sind. Die anschließende Addition von $a \in \mathbb{C}$ ändert dann nach **Satz D.6** nichts an der optimalen Einschließung des Wertebereichs $W_{T_1,Z}$.

Mit $a = 1+2i, b = -1+i, d = -2+2i$ und $Z = [1,2]+i\cdot[-1,0]$ vergleichen wir die Einschließungen, die mit den Ausdrücken T_1 und T mit der Präzision `prec` = 40 berechnet werden, man erhält:

$Z = X + i\cdot Y = [1,2] + i\cdot[-1,0]$; Präzision: `prec` = 40.	
$T_1(z) = \dfrac{b-ad}{z+d} + a$	$T_1(Z) = ([0, 4.00000000000], [-3.41547594745, -5.00000000000e-1])$
$T(z) = \dfrac{az+b}{z+d}$	$T(Z) = ([-5.00000000000e-1, 5.00000000000], [-4.41547594744, -0])$

Man erkennt im Vergleich zu $T(Z)$ deutlich die viel bessere Einschließung $T_1(Z) \subset T(Z)$, wobei $T_1(Z)$ sogar die **optimale** Einschließung von $W_{T,Z} = \{w \in \mathbb{C} \,|\, w = (az+b)/(z+d) \wedge z \in Z\}$ ist.

E. Tabellen mathematischer Funktionen

E.1. Reelle Punkt-Funktionen

Tabelle E.1.: Elementarfunktionen mit `x,y,a,b` vom Typ `MpfrClass`, `n`: `unsigned long int`;

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(1+x)$	`lnp1(x)`
x^2	`sqr(x)`	$\log_2(x)$	`log2(x)`
x^2+y^2	`x2py2(x,y)`	$\log_{10}(x)$	`log10(x)`
x^2-y^2	`x2my2(x,y)`	$\ln(\sin(x))$	`ln_sin(x)`
$1/x$	`reci(x)`	$\ln(\cos(x))$	`ln_cos(x)`
$x/(x^2+y^2)$	`x_div_x2py2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
$(x^2-y^2)/(x^2+y^2)^2$	`Re_rz2(x,y)`	$\ln(\sqrt{(1+x)^2+y^2})$	`ln_sqrtxp1_2y2(x,y)`
$2xy/(x^2+y^2)^2$	`mIm_rz2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$\dfrac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2}$	`Re_r1pz2(x,y)`	x^y	`pow(x, y)`
$\dfrac{2xy}{4x^2y^2+(1+x^2-y^2)^2}$	`Im_r1pz2(x,y)`	$\sin(x)$	`sin(x)`
$\sqrt{x}$	`sqrt(x)`	$1/\sin(x)$	`csc(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_n(n)`	$\cos(x)$	`cos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$1/\cos(x)$	`sec(x)`
$\sqrt[3]{x},\quad x\in\mathbb{R}$	`cbrt(x)`	$\tan(x)$	`tan(x)`
$\sqrt[n]{x},\ n=2,3,\ldots$	`sqrt(x,n)`	$\cot(x)$	`cot(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arctan(x)$	`atan(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\arctan(y/x)$	`atan2(y,x)`
$x/\sqrt{1+x^2}$	`xdsqrt1px2(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$x/\sqrt{1-x^2}$	`xdsqrt1mx2(x)`	$\sinh(x)$	`sinh(x)`
$x/\sqrt{x^2-1}$	`xdsqrtx2m1(x)`	$1/\sinh(x)$	`csch(x)`
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$\cosh(x)$	`cosh(x)`
e^x	`exp(x)`	$1/\cosh(x)$	`sech(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)`	AGM(x,y)	`agm(x,y)`

Weitere Anmerkungen findet man auf Seite 38.

E.2. Reelle Punkt-Funktionen der Mathematischen Physik

Tabelle E.2.: Funktionen der Mathematischen Physik mit `x` vom Typ `MpfrClass`

Funktionsterm	Aufruf	Anmerkung
$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	`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;

Weitere Anmerkungen findet man auf Seite 45.

E.3. Reelle Intervall-Funktionen

Tabelle E.3.: Elementarfunktionen mit `x,y` vom Typ `MpfiClass`, `a` vom Typ `MpfrClass`;

Funktion	Aufruf	Funktion	Aufruf
$\lvert x\rvert$	`abs(x)`	$\ln(x)$	`ln(x)`
x^2	`sqr(x)`	$\ln(1+x)$	`lnp1(x)`
x^2+y^2	`x2py2(x,y)`	$\log_2(x)$	`log2(x)`
x^2-y^2	`x2my2(x,y)`	$\log_{10}(x)$	`log10(x)`
$1/x$	`reci(x)`	$\ln(\sin(x))$	`ln_sin(x)`
$x/(x^2+y^2)$	`x_div_x2py2(x,y)`	$\ln(\cos(x))$	`ln_cos(x)`
$(x^2-y^2)/(x^2+y^2)^2$	`Re_rz2(x,y)`	$\ln(\sqrt{x^2+y^2})$	`ln_sqrtx2y2(x,y)`
$2xy/(x^2+y^2)^2$	`mIm_rz2(x,y)`	$\ln(\sqrt{(1+x)^2+y^2})$	`ln_sqrtxp1_2y2(x,y)`
$\dfrac{1+x^2-y^2}{4x^2y^2+(1+x^2-y^2)^2}$	`Re_r1pz2(x,y)`	$x^k,\ k\in\mathbb{Z}$	`power(x,k)`
$\dfrac{2xy}{4x^2y^2+(1+x^2-y^2)^2}$	`Im_r1pz2(x,y)`	x^y	`pow(x, y)`
$\sqrt{x}$	`sqrt(x)`	$\sin(x)$	`sin(x)`
$\sqrt{n},\ n\in\mathbb{N}_0$	`sqrt_N(n)`	$1/\sin(x)$	`csc(x)`
$a/\sqrt{n},\ n\in\mathbb{N}$	`xdivsqrtn(a,n)`	$\cos(x)$	`cos(x)`
$x/\sqrt{n},\ n\in\mathbb{N}$	`xdivsqrtn(x,n)`	$1/\cos(x)$	`sec(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)`	$\arg(x+i\cdot y)$	`atan2(y,x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$x/\sqrt{1+x^2}$	`xdsqrt1px2(x)`	$\sinh(x)$	`sinh(x)`
$x/\sqrt{1-x^2}$	`xdsqrt1mx2(x)`	$1/\sinh(x)$	`csch(x)`
$x/\sqrt{x^2-1}$	`xdsqrtx2m1(x)`	$\cosh(x)$	`cosh(x)`
$\sqrt{x^2+y^2}$	`sqrtx2y2(x,y)`	$1/\cosh(x)$	`sech(x)`
e^x	`exp(x)`	$\tanh(x)$	`tanh(x)`
2^x	`exp2(x)`	$\coth(x)$	`coth(x)`
10^x	`exp10(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
e^x-1	`expm1(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
e^{x^2}	`expx2(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
$e^{x^2}-1$	`expx2m1(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
e^{-x^2}	`expmx2(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$e^{-x^2}-1$	`expmx2m1(x)`	AGM(x,y)	`agm(x,y)`

Weitere Anmerkungen findet man auf Seite 69.

E.4. Reelle Intervall-Funktionen der Mathematischen Physik

Tabelle E.4.: Funktionen der Mathematischen Physik mit `x` vom Typ `MpfiClass`

Funktionsterm	Aufruf	Anmerkung
$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	`erf(x)`	monoton wachsend
$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	`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` ⇝ `-Inf;` `Inf(x)<0` ⇝ `NaN;`

Weitere Anmerkungen findet man auf Seite 71.

E.5. Reelle Funktionen zur Automatischen Differentiation

Tabelle E.5.: Funktionen vom Typ `MPDerivType` zur Automatischen Differentiation

Funktion	Aufruf	Funktion	Aufruf
x^2	`sqr(x)`	$\ln(\cos(x))$	`ln_cos(x)`
$x^n,\ n \in \mathbb{Z}$	`power(x,n)`	$\sin(x)$	`sin(x)`
x^y	`pow(x, y)`	$\cos(x)$	`cos(x)`
$1/x$	`reci(x)`	$\tan(x)$	`tan(x)`
$\sqrt{x}$	`sqrt(x)`	$\cot(x)$	`cot(x)`
$\sqrt[3]{x},\quad x \in \mathbb{R}$	`cbrt(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt[n]{x},\ n = 2, 3, \ldots$	`sqrt(x,n)`	$\arccos(x)$	`acos(x)`
$1/\sqrt{x}$	`sqrt_r(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\operatorname{arccot}(x)$	`acot(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\sinh(x)$	`sinh(x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\cosh(x)$	`cosh(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\tanh(x)$	`tanh(x)`
e^x	`exp(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)`
2^x	`exp2(x)`	$1/\sin(x)$	`csc(x)`
10^x	`exp10(x)`	$1/\cos(x)$	`sec(x)`
$\ln(x)$	`ln(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\cosh(x)$	`sech(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Funktion	Aufruf	Funktion	Aufruf
$\log_2(x)$	`log2(x)`	$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$	`erf(x)`
$\log_{10}(x)$	`log10(x)`	$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,dt$	`erfc(x)`
$\ln(\sin(x))$	`ln_sin(x)`		

Weitere Anmerkungen findet man auf Seite 128.

E.6. Reelle Funktionen zur Taylorarithmetik

Tabelle E.6.: Funktionen vom Typ `MPitaylor` zur Taylor-Arithmetik

Funktion	Aufruf	Funktion	Aufruf
x^2	`sqr(x)`	$\cos(x)$	`cos(x)`
$\sqrt{x}$	`sqrt(x)`	$\tan(x)$	`tan(x)`
$\sqrt[n]{x},\ n = 2, 3, \ldots$	`sqrt(x,n)`	$\cot(x)$	`cot(x)`
$\sqrt{1+x^2}$	`sqrt1px2(x)`	$\arcsin(x)$	`asin(x)`
$\sqrt{1-x^2}$	`sqrt1mx2(x)`	$\arccos(x)$	`acos(x)`
$\sqrt{x^2-1}$	`sqrtx2m1(x)`	$\arctan(x)$	`atan(x)`
$\sqrt{x+1}-1$	`sqrtp1m1(x)`	$\operatorname{arccot}(x)$	`acot(x)`
x^y	`pow(x, y)`	$\sinh(x)$	`sinh(x)`
$x^n,\ n \in \mathbb{Z}$	`power(x,n)`	$\cosh(x)$	`cosh(x)`
e^x	`exp(x)`	$\tanh(x)$	`tanh(x)`
2^x	`exp2(x)`	$\coth(x)$	`coth(x)`
10^x	`exp10(x)`	$\operatorname{arsinh}(x)$	`asinh(x)`
$e^x - 1$	`expm1(x)`	$\operatorname{arcosh}(x)$	`acosh(x)`
$e^{x^2} - 1$	`expx2m1(x)`	$\operatorname{artanh}(x)$	`atanh(x)`
$e^{-x^2} - 1$	`expmx2m1(x)`	$\operatorname{arcoth}(x)$	`acoth(x)`
$\ln(x)$	`ln(x)`	$\operatorname{arcosh}(1+x)$	`acoshp1(x)`
$\log_2(x)$	`log2(x)`	$1/\sin(x)$	`csc(x)`
$\log_{10}(x)$	`log10(x)`	$1/\cos(x)$	`sec(x)`
$\ln(1+x)$	`lnp1(x)`	$1/\sinh(x)$	`csch(x)`
$\ln(\sin(x))$	`ln_sin(x)`	$1/\cosh(x)$	`sech(x)`
$\ln(\cos(x))$	`ln_cos(x)`	$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$	`erf(x)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\sin(x)$	`sin(x)`	$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,dt$	`erfc(x)`

Weitere Anmerkungen findet man auf Seite 139.

E.7. Komplexe Punkt-Funktionen

Tabelle E.7.: Elementarfunktionen mit `z` = $x + i \cdot y$, a, b vom Typ `MpfcClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	$e^z - 1$	`expm1(z)`
$\mathrm{Re}(z) = x$	`Re(z)`	e^{z^2}	`expx2(z)`
$\mathrm{Im}(z) = y$	`Im(z)`	$e^{z^2} - 1$	`expx2m1(z)`
$\lvert z \rvert$	`abs(z)`	e^{-z^2}	`expmx2(z)`
$\arg(z)$	`arg(z)`	$e^{-z^2} - 1$	`expmx2m1(z)`
$\arg(1 + z)$	`argp1(z)`	2^z	`exp2(z)`
z^2	`sqr(z)`	10^z	`exp10(z)`
$z^2 + a \cdot z + b$	`poly2(z,a,b)`	$z^n,\ n \in \mathbb{Z}$	`power(z,n)`
$1/z$	`reci(z)`	$z^r,\ r \in \mathbb{R}$	`pow(z,r)`
$1/z^2$	`reci_z2(z)`	$z^w,\ w \in \mathbb{C}$	`pow(z,w)`
$1/(1 + z^2)$	`reci_1pz2(z)`	$\sin(z)$	`sin(z)`
$1/(1 - z^2)$	`reci_1mz2(z)`	$\cos(z)$	`cos(z)`
$\sqrt{z}$	`sqrt(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\cot(z)$	`cot(z)`
$\sqrt{z+1} - 1$	`sqrtp1m1(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{1 + z^2}$	`sqrt1px2(z)`	$\arccos(z)$	`acos(z)`
$\sqrt{1 - z^2}$	`sqrt1mx2(z)`	$\arctan(z)$	`atan(z)`
$\sqrt{z^2 - 1}$	`sqrtx2m1(z)`	$\mathrm{arccot}(z)$	`acot(z)`
$\sqrt{-z^2 - 1}$	`sqrtmx2m1(z)`	$\sinh(z)$	`sinh(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt(z,n)`	$\cosh(z)$	`cosh(z)`
$\sqrt[n]{z},\ n \in \mathbb{Z}$	`sqrt_all(z,n)`	$\tanh(z)$	`tanh(z)`
$\log(z)$	`ln(z)`	$\coth(z)$	`coth(z)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\log(1+z)$	`lnp1(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
$\log_2(z)$	`log2(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$\log_{10}(z)$	`log10(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
e^z	`exp(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`

Weitere Anmerkungen findet man auf Seite 93.

E.8. Komplexe Intervall-Funktionen

Tabelle E.8.: Funktionen mit `z` = $x + i \cdot y$, a, b vom Typ `MpfciClass`; x, y vom Typ `MpfiClass`

Funktion	Aufruf	Funktion	Aufruf
$\mathrm{conj}(z) = x - i \cdot y$	`conj(z)`	2^z	`exp2(z)`
$\mathrm{Re}(z) = x$	`Re(z)`	10^z	`exp10(z)`
$\mathrm{Im}(z) = y$	`Im(z)`	e^z	`exp(z)`
$\lvert z \rvert$	`abs(z)`	$e^z - 1$	`expm1(z)`
$\mathrm{Arg}(z)$	`Arg(z)`	e^{z^2}	`expx2(z)`
$\arg(z)$	`arg(z)`	$e^{z^2} - 1$	`expx2m1(z)`
$\arg(1 + z)$	`argp1(z)`	e^{-z^2}	`expmx2(z)`
z^2	`sqr(z)`	$e^{-z^2} - 1$	`expmx2m1(z)`
$z^2 + a \cdot z + b$	`poly2(z,a,b)`	z^n, $n \in \mathbb{Z}$	`power_fast(z,n)`
$1/z$	`reci(z)`	z^n, $n \in \mathbb{Z}$	`power(z,n)`
$1/z^2$	`reci_z2(z)`	z^p, p : `MpfiClass`	`pow(z,p)`
$1/(1 + z^2)$	`reci_1pz2(z)`	z^p, p : `MpfiClass`	`pow_all(z,p)`
$1/(1 - z^2)$	`reci_1mz2(z)`	z^w, w : `MpfciClass`	`pow(z,w)`
$\sqrt{z}$	`sqrt(z)`	$\sin(z)$	`sin(z)`
$\sqrt{z}$	`sqrt_all(z)`	$\cos(z)$	`cos(z)`
$1/\sqrt{z}$	`sqrt_r(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z + 1} - 1$	`sqrtp1m1(z)`	$\cot(z)$	`cot(z)`
$\sqrt{1 + z^2}$	`sqrt1px2(z)`	$\arcsin(z)$	`asin(z)`
$\sqrt{1 - z^2}$	`sqrt1mx2(z)`	$\arccos(z)$	`acos(z)`
$\sqrt{z^2 - 1}$	`sqrtx2m1(z)`	$\arctan(z)$	`atan(z)`
$\sqrt{-z^2 - 1}$	`sqrtmx2m1(z)`	$\mathrm{arccot}(z)$	`acot(z)`
$\sqrt[n]{z}$	`sqrt(z,n)`	$\sinh(z)$	`sinh(z)`
$\sqrt[n]{z}$	`sqrt_all(z,n)`	$\cosh(z)$	`cosh(z)`
$\log(z)$	`Ln(z)`	$\tanh(z)$	`tanh(z)`
$\log(1 + z)$	`Lnp1(z)`	$\coth(z)$	`coth(z)`

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite			
Funktion	Aufruf	Funktion	Aufruf
$\log(z)$	`ln(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
$\log(1+z)$	`lnp1(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$\log_2(z)$	`log2(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
$\log_{10}(z)$	`log10(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`

Weitere Anmerkungen findet man auf Seite 116.

E.9. Komplexe Funktionen zur Taylorarithmetik

Tabelle E.9.: Funktionen vom Typ `MPcitaylor` zur Taylor-Arithmetik, z vom Typ `MPcitaylor`, w vom Typ `MpfciClass`;

Funktion	Aufruf	Funktion	Aufruf
z^2	`sqr(z)`	$\log_{10}(z)$	`log10(z)`
$\sqrt{z}$	`sqrt(z)`	$\ln(1+z)$	`lnp1(z)`
$\sqrt[n]{z},\ n = 2,3,\ldots$	`sqrt(z,n)`	$\sin(z)$	`sin(z)`
$\sqrt{1+z^2}$	`sqrt1px2(z)`	$\cos(z)$	`cos(z)`
$\sqrt{1-z^2}$	`sqrt1mx2(z)`	$\tan(z)$	`tan(z)`
$\sqrt{z^2-1}$	`sqrtx2m1(z)`	$\cot(z)$	`cot(z)`
$\sqrt{z+1}-1$	`sqrtp1m1(z)`	$\arcsin(z)$	`asin(z)`
z^w	`pow(z,w)`	$\arccos(z)$	`acos(z)`
$z^n,\ n \in \mathbb{Z}$	`power(z,n)`	$\arctan(z)$	`atan(z)`
e^z	`exp(z)`	$\operatorname{arccot}(z)$	`acot(z)`
2^z	`exp2(z)`	$\sinh(z)$	`sinh(z)`
10^z	`exp10(z)`	$\cosh(z)$	`cosh(z)`
e^z-1	`expm1(z)`	$\tanh(z)$	`tanh(z)`
e^{z^2}	`expx2(z)`	$\coth(z)$	`coth(z)`
$e^{z^2}-1$	`expx2m1(z)`	$\operatorname{arsinh}(z)$	`asinh(z)`
e^{-z^2}	`expmx2(z)`	$\operatorname{arcosh}(z)$	`acosh(z)`
$e^{-z^2}-1$	`expmx2m1(z)`	$\operatorname{artanh}(z)$	`atanh(z)`
$\ln(z)$	`ln(z)`	$\operatorname{arcoth}(z)$	`acoth(z)`
$\log_2(z)$	`log2(z)`		

Weitere Anmerkungen findet man auf Seite 147.

F. 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 zwischen *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 Grigorieff, R. D. *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis).* Computing Supplementum 2, Springer-Verlag, Wien / New York, 1980.

[5] Alefeld G. and Herzberger J. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[6] Alefeld G. and Herzberger J. *Einführung in die Intervallrechnung.* BI, Reihe Informatik/12, 1983.

[7] Behnke H., Sommer F. *Theorie der analytischen Funktionen einer komplexen Veränderlichen.* Springer, Berlin, 1962.

[8] 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.

[9] 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.

[10] 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`

[11] Blomquist, F.; Hofschuster, W.; Krämer, W.: Real and ComplexTaylor Arithmetic in C-XSC. Universität Wuppertal, 57 Seiten, 2005.
`http://www2.math.uni-wuppertal.de/wrswt/preprints/prep_05_4.pdf`

[12] Blomquist F.: Verbesserungen der komplexen Standardfunktionen von Markus Neher (see [54]). Interne Mitteilung, Bergische Universität Wuppertal, 2005.

[13] 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.

[14] 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.

[15] Blomquist, F.: Staggered Correction Computations with Enhanced Accuracy and Extremely Wide Exponent Range. Reliable Computing, Vol. 15, pp. 26-35, May, 2011.

[16] Brand, Hans-Stephan: Integration und Test einer Langzahlintervallbibliothek in C-XSC. Bachelor-Arbeit, Universität Wuppertal, 2010.

[17] 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.

[18] Braune, K.: Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:159-184, 1988.

[19] Corliss, G. F.: *Automatic Differentiation Bibliography.* In [26], pp 331-353, 1991.

[20] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer-Verlag, Berlin / Heidelberg / New York, 1995.

[21] 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.

[22] Fritzsche, A.: Grundkurs Funktionentheorie. Spektrum Akademischer Verlag Heidelberg, 2009.

[23] GNU MP LIBRARY, `http://gmplib.org/`

[24] GNU MPFR LIBRARY, `http://www.mpfr.org/mpfr-current/mpfr.html`

[25] Griewank, A.: *On Automatic Differentiation.* In Iri, M., Tanabe, K.: *Mathematical Programmimg: Recent Developments and Applications,* pp. 83-108, Kluwer Academic Publishers, 1989.

[26] Griewank, A., Corliss, G.: *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia, 1991.

[27] Hansen, E.: *Global Optimization Using Interval Analysis -The One-Dimensional Case.* Journal of Optimization Theory and Applications 29, pp 331-344, 1979.

[28] Hansen, E.: *Global Optimization Using Interval Analysis -The Multi-Dimensional Case.* Numerische Mathematik 34, pp 247-270, 1980.

[29] Hansen, E.: Global Optimization Using Interval Analysis. Marcel Dekker, New York, 1992.

[30] Ratscheck, H.; Rokne, J.: *Computer Methods for the Range of Functions.* Ellis Horwood Limited, Chichester, 1984.

[31] Ratscheck, H.; Rokne, J.: New Computer Methods for Global Optimization. Ellis Horwood Limited, Chichester, 1988.

[32] Herzberger, J. (Ed): Topics in Validated Computations. Proceedings of IMACS-GAMM International Workshop on Validated Numerics, Oldenburg, 1993. North Holland, 1994.

[33] IBM: High-Accuracy Arithmetic Subroutine Library(ACRITH). IBM Deutschland GmbH, third edition, 1986.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Krämer, W.: Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate, Dissertation, Universität Karlsruhe, 1987.

[39] Krämer W.: Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:185-212, 1988.

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

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

[42] Krämer, W.: Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, Bericht des Instituts für Angewandte Mathematik, Universität Karlsruhe, S. 1-80, 1988.

[43] Krämer, W.: Die Berechnung von Funktionen und Konstanten in Rechenanlagen. Habilitationsschrift, Universität Karlsruhe 1993.

[44] 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.

[45] Krämer, W.; Kulisch, U.; Lohner, R.: Numerical Toolbox for Verified Computing II. Springer Verlag. Draft version, Karlsruhe, 1994.

[46] Kulisch, U.: Computer Arithmetic and Validity – Theory, Implementation and Applications. de Gruyter, Berlin, 2008.

[47] 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.

[48] 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.

[49] 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.

[50] Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs, New Jersey, 1966

[51] MPFR manual. April 2011.
http://www.mpfr.org/mpfr-current/mpfr.pdf

[52] MPFI library for arbitrary precision interval arithmetic.
http://cadadr.org/fm/package/mpfi.html

[53] Neher, M.: The mean value form for complex analytic functions. *Computing*, 67:255-268, 2001.

[54] Neher M.: Complex Standard Functions and their Implementation in the CoStLy Library. Preprint Nr. 04/18, Universität Karlsruhe, 2004.

[55] Neher, M.: Complex Standard Functions and Their Implementation in the CoStLy Library, ACM Transactions on Mathematical Software, Vol. 33, Number 1, 27 pages, 2007.

[56] Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Campridge, 1990.

[57] Rall, L. B.: *Applications of Software for Automatic Differentiation in Numerical Computation.* In [4], pp. 141-156, 1980.

[58] Ratscheck, H.: Die Subdistributivität in der Intervallarithmetik. ZAMM 51, pp 189-192, 1971.

[59] Ratz, D.: Automatische Ergebnisverifikation bei globalen Optimierungsproblemen. Dissertation, Universität Karlsruhe, 1992.

[60] 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.

[61] Rotmaier, B.: Die Berechnung der elementaren Funktionen mit beliebiger Genauigkeit. Dissertation, Universität Karlsruhe, 1971.

[62] M. R. Spiegel: *KOMPLEXE VARIABLEN, Theorie und Anwendung*, Schaum's Überblicke, Aufgaben, 1991.

[63] 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.

[64] Törn, A., Zilinskas, A.: *Global Optimization.* Lecture Notes in Computer Science, No. 350, Springer-Verlag, Berlin, 1989.

[65] XSC website on programming languages for scientific computing with validation.
http://www.xsc.de

[66] http://en.wikipedia.org/wiki/Wilkinson's_polynomial

Stichwortverzeichnis