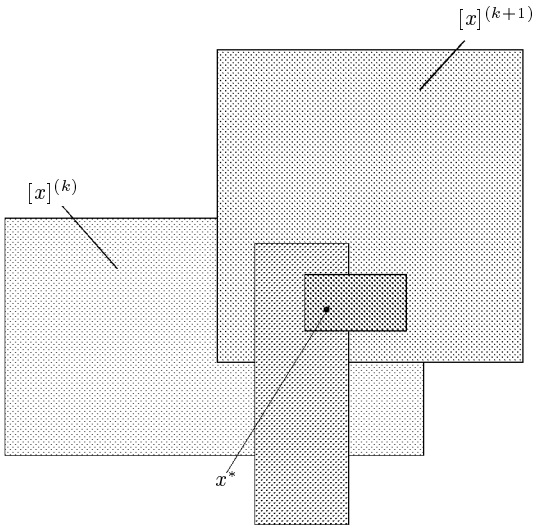
Institut für
Angewandte
Mathematik

Universität Karlsruhe (TH)
D-76128 Karlsruhe

# Implementierung, Handhabung und Beispielanwendungen eines verlässlichen Vorwärtsfehlerkalküls

Armin Bantle und Walter Krämer

**Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation**




**Bericht 2/1999**

# Impressum


| | |
|---|---|
| Herausgeber: | Institut für Angewandte Mathematik<br>Lehrstuhl Prof. Dr. Ulrich Kulisch<br>Universität Karlsruhe (TH)<br>D-76128 Karlsruhe |
| Redaktion: | Dr. Gerd Bohlender |


# Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über

`ftp://iamk4515.mathematik.uni-karlsruhe.de`

im Verzeichnis: `/pub/documents/reports`

oder über die World Wide Web Seiten des Instituts

`http://www.uni-karlsruhe.de/~iam`

# Autoren-Kontaktadresse

Rückfragen zum Inhalt dieses Berichts bitte an


Armin Bantle
Walter Krämer
Institut für Angewandte Mathematik
Universität Karlsruhe (TH)
D-76128 Karlsruhe

E-Mail: armin.bantle@math.uni-karlsruhe.de
walter.kraemer@math.uni-karlsruhe.de

# Implementierung, Handhabung und Beispielanwendungen eines verlässlichen Vorwärtsfehlerkalküls

Armin Bantle und Walter Krämer

## Inhaltsverzeichnis

**Zusammenfassung**

**Implementierung eines verlässlichen Vorwärtsfehlerkalküls:**

In dem vorliegenden Preprint wird eine einfach handhabbare Umsetzung des in den Arbeiten [1], [6], [13], [14] und [2] hergeleiteten und näher beschriebenen Fehlerkalküls in die Softwarebibliothek `Bound` vorgestellt. Diese ermöglicht eine rigorose automatische Fehleranalyse numerischer Algorithmen, die das im IEEE-Standard 754-1985 [3], [9] festgelegte Datenformat doppelt genauer Gleitkommazahlen (64 Bit) verwenden. Auch für bereits existierende Programme können mit nur minimalen Quellcodeanpassungen – im wesentlichen durch Abändern von Datentypen – verlässliche worst-case Vorwärtsfehleranalysen durchgeführt werden.

Die Routinen der Softwarebibliothek `Bound` sind in der Programmiersprache C++ unter Verwendung der Klassenbibliothek C-XSC [10] und der Toolbox [5] realisiert. An einigen Beispielen wird die einfache Handhabung der Software demonstriert. Die Software ist frei verfügbar. Sie kann unter ... abgerufen werden.

**Abstract**

**Implementation, Handling and Application of a Tool for Automatic Forward Error Analyses:** We describe the implementation of a tool that allows the computation of a priori error bounds for floating point algorithms (using the IEEE double data format) automatically. We show the handling of the tool and we discuss some applications. We also give numerical results. The software can be found at ....

# 1 Einleitung

Eine wichtige Eigenschaft numerischer Algorithmen ist ihr Verhalten bzgl. der Verstärkung auftretender Rundungsfehler und möglicher Störungen in den Eingangsdaten. Man ist an einer a priori Oberschranke für die maximale Abweichung von berechnetem Ergebnis zu exaktem Ergebnis interessiert, wobei die Daten aus gewissen (fest) vorgegebenen Bereichen stammen dürfen. Automatische Fehlerkalküle (siehe z. B. [1], [6], [13], [14] [2], [4] ) unterstützen den i. a. sehr aufwendigen Prozess der sicheren Fehlerkontrolle ganz wesentlich.

In der vorliegenden Arbeit wird die Umsetzung eines derartigen Fehlerkalküls in eine Softwarebibliothek mit Namen `Bound` vorgestellt. Diese ermöglicht eine rigorose automatische Fehleranalyse numerischer Algorithmen, die das im IEEE-Standard 754-1985 [3], [9] festgelegte Datenformat doppelt genauer Gleitkommazahlen (64 Bit) verwenden. Auch für bereits existierende Programme können mit nur minimalen Quellcodeanpassungen – im wesentlichen durch Abändern von Datentypen – verlässliche Vorwärtsfehleranalysen durchgeführt werden.

Die Bibliothek `Bound` zeichnet sich besonders aus durch ihre einfache Handhabbarkeit, ihre Kompaktheit (ca. 2300 Zeilen Quellcode), die automatische Erfassung aller Konvertierungs- und Rundungsfehler, die Berücksichtigung des Unterlaufbereichs, sichere worst case Abschätzungen für ganze Daten- bzw. Parameterbereiche und die automatische Erkennung einiger Klassen von Operationen, die rundungsfehlerfrei durchgeführt werden können. Außerdem ermöglicht sie wahlweise die Berechnung absoluter bzw. relativer Fehlerschranken. Falls im relativen Modus die Berechnung einer relativen

Schranke nicht möglich ist, liefert die Bibliothek stattdessen eine absolute Schranke, kehrt aber so bald wie möglich in den relativen Modus zurück. Es hat sich bei konkreten Tests gezeigt, dass die Bibliothek besonders bei der Berechnung relativer Fehlerschranken über einem größeren Datenbereich gute Resultate liefert.

Die Bibliothek `Bound` lässt sich in verschiedenen Bereichen einsetzen wie z. B. der Vorwärtsfehleranalyse, der Stabilitäts- bzw. Sensitivitätsanalyse und der a priori Fehlerbestimmung von Programmstücken. Im letztgenannten Bereich ist die Bibliothek bei der Realisierung schneller Intervallstandardfunktionen bereits erfolgreich eingesetzt worden [8].

Neben der Beschreibung der einzelnen Bibliothekskomponenten werden auch einige Anwendungsbeispiele gegeben, die dem Anwender den Einstieg vereinfachen und ihm den Umgang mit der Bibliothek erläutern sollen. Weitere Beispiele finden sich u. a. in [1], [6], [7], [8] und [13].

Die Routinen der Softwarebibliothek bauen auf Intervalloperationen auf und sind in der Programmiersprache C++ unter Verwendung der Klassenbibliothek C-XSC [10] und der Toolbox [5] realisiert.

# 2 Beschreibung der Bibliothek `Bound`

## 2.1 Typen

Die Bibliothek enthält zwei neue Datentypen:

1. Die Klasse `BoundType`: Sie wird in Abschnitt 2.4 beschrieben.

2. Der Aufzählungstyp `errorflag`: Er umfasst die Elemente `ABS` und `REL` und dient zur Unterscheidung von absoluten und relativen Fehlerschranken.

## 2.2 Globale Größen und Hilfsfunktionen

### Konstanten

- `const real Eps52`

  Maschinenepsilon $\varepsilon = 2^{-52} = 2.220446\ldots \cdot 10^{-16}$ bei Verwendung einer 64 Bit IEEE-Arithmetik mit Rundung nach oben/unten.

- `const real Eps53`

  Maschinenepsilon $\varepsilon = 2^{-53} = 1.110223\ldots \cdot 10^{-16}$ bei Verwendung einer 64 Bit IEEE-Arithmetik mit Rundung zur nächstgelegenen Maschinenzahl.

- `const real MinReal`

  Kleinste positive normalisierte Gleitpunktzahl ($2.225073\ldots \cdot 10^{-308}$).

- `const real dMinReal`

  Kleinste positive denormalisierte Gleitpunktzahl ($4.940656\ldots \cdot 10^{-324}$).

- `const real MaxReal`

  Größte positive Gleitpunktzahl ($1.797693\ldots\cdot 10^{+308}$).

- `const interval UnflowRange`

  Unterlaufbereich ([`-MinReal`,`MinReal`]).

**Hilfsfunktionen**

- `real MaxAbs(interval& X)`

  `X`: *Intervall, dessen absolutes Maximum berechnet werden soll.*

  Das absolute Maximum $|\mathtt{X}| := \max_{a\in\mathtt{X}} |a|$ des Intervalls `X` wird berechnet und zurückgegeben.

- `real MinAbs(interval& X)`

  `X`: *Intervall, dessen absolutes Minimum berechnet werden soll.*

  Das absolute Minimum $\langle\mathtt{X}\rangle := \min_{a\in\mathtt{X}} |a|$ des Intervalls `X` wird berechnet und zurückgegeben.

- `real Max(real& x, real& y)`

  `x`: *1. zu vergleichender Wert*
  `y`: *2. zu vergleichender Wert*

  Das Maximum der beiden Werte `x` und `y` wird ermittelt.

- `real Min(real& x, real& y)`

  `x`: *1. zu vergleichender Wert*
  `y`: *2. zu vergleichender Wert*

  Das Minimum der beiden Werte `x` und `y` wird ermittelt.

- `real rel2abs(interval& X, real& eps)`

  `X`: *Bezugsintervall*
  `eps`: *relativer Fehler der Werte in* `X`

  Aus dem relativen Fehler `eps` wird der maximale absolute Fehler $\mathtt{eps}\cdot|\mathtt{X}|$ über dem Intervall `X` berechnet.

- `real abs2rel(interval& X, real& eps)`

  `X`: *Bezugsintervall*
  `eps`: *absoluter Fehler der Werte in* `X`

  Aus dem absoluten Fehler `eps` wird der maximale relative Fehler $\mathtt{eps}/\langle\mathtt{X}\rangle$ über dem Intervall `X` berechnet.

- `bool test_point(interval& X)`

`X`: *zu testendes Intervall*

Es wird geprüft, ob das Intervall `X` ein Punktintervall ist, d. h. ob $\inf(\texttt{X}) = \sup(\texttt{X})$ gilt.

- `bool test_power_of_2(real& x)`

  `x`: *zu testender Wert*

  Es wird geprüft, ob `x` eine Potenz von 2 ist. `x` darf dabei auch im Unterlaufbereich sein.

- `int leading_zeroes(real& x)`

  `x`: *zu untersuchender Wert*

  Diese Funktion gibt die Anzahl der führenden Nullen in der Mantisse von `x` zurück. Für $\texttt{x} = 0$ und normalisierte Zahlen wird 0 zurückgegeben.

- `int trailing_zeroes(real& x)`

  `x`: *zu untersuchender Wert*

  Hier wird die Anzahl der abschließenden Nullen in der Mantisse von `x` ermittelt. Für $\texttt{x} = 0$ ist das Ergebnis 53.

- `int test_underflow(interval& X)`

  `X`: *zu testendes Intervall*

  Die Lage des Intervalls `X` zum Unterlaufbereich `UnflowRange` wird bestimmt und entsprechend einer der folgenden Werte zurückgegeben:

  - 0, falls $\texttt{X} \cap \texttt{UnflowRange} = \emptyset$
  - 1, falls $\texttt{X} \subseteq \texttt{UnflowRange}$
  - 2, falls $\texttt{X} \cap \texttt{UnflowRange} \neq \emptyset$ und $\texttt{X} \cap (\texttt{UnflowRange})^c \neq \emptyset$.

## 2.3 Fehlerschrankenarithmetik

Die Fehlerschrankenarithmetik ist Teil der Klasse `BoundType` (siehe Abschnitt 2.4). Da sie den funktionellen Kern der Klasse bildet, sind ihre Funktionen hier in einem eigenen Abschnitt zusammengefasst.

### Steuerungsfunktionen

Das Verhalten der Fehlerschrankenarithmetik hängt von verschiedenen Parametern und Flags ab. Mit den folgenden Funktionen können diese manipuliert und abgefragt werden. Außerdem führt die Arithmetik verschiedene Zähler mit sich, die zurückgesetzt und abgefragt werden können.

- ```
  void do_not_optimize(bool quiet= false)
  void do_optimize    (bool quiet= false)
  ```
  ```

`quiet`: *Flag für die Ausgabe einer Warnung (optionaler Parameter)*

Diese beiden Funktionen schalten die Fehlerschrankenoptimierung aus bzw. ein. Das Flag `quiet` gibt an, ob eine Warnung ausgegeben werden soll (`false` oder `WARNING`) oder nicht (`true` oder `QUIET`). Das Ausschalten der Optimierung beschleunigt Analysen, kann aber zu schlechteren Fehlerschranken führen, was bei Intervallunterteilungen den Geschwindigkeitsvorteil zunichte machen kann.

- `void SetEpsQuer(real& EpsAkt)`

  `EpsAkt`: *neue Oberschranke für das Maschinenepsilon*

  Mit dieser Funktion wird die Oberschranke für das Maschinenepsilon neu gesetzt. Alle weiteren Berechnungen werden mit dem neuen $\overline{\varepsilon}$-Wert abgeschätzt. Der voreingestellte Wert $\overline{\varepsilon} =$ `Eps52` $= 2^{-52}$ entspricht einer IEEE-Arithmetik, die „faithful“ ist. Falls die verwendete Arithmetik jedoch Zwischenergebnisse immer zur nächstgelegen Maschinenzahl rundet, kann der Wert `Eps53` $= 2^{-53}$ eingestellt werden, wodurch sich bessere Fehlerschranken ergeben.

- `real& GetEpsQuer()`

  Der aktuelle Wert der Oberschranke für das Maschinenepsilon wird zurückgegeben.

- `void SetSwitchTest()`

  Ab dem Zeitpunkt des Aufrufs wird bei jeder Operation geprüft, ob ein erzwungener Wechsel (Null im Zwischenergebnis) vom relativen in den absoluten Fehlermodus stattgefunden hat; wenn ja, wird eine Warnung ausgegeben.

- `void ResetSwitchTest()`

  Schaltet die Warnungen wieder aus.

- `void SetUnderflowTest()`

  Ab dem Zeitpunkt des Aufrufs wird bei jeder Operation mit Ergebnis im Unterlaufbereich eine Warnung ausgegeben.

- `void ResetUnderflowTest()`

  Schaltet die Unterlaufwarnungen wieder aus.

- `void SetErrMode(errorflag fl)`

  `fl`: *neuer Fehlerschranken-Modus*

  Der Fehlerschranken-Modus wird gemäß `fl` gesetzt:

  – für `fl=ABS` werden für weitere Berechnungen absolute Fehlerschranken ermittelt,

  – für `fl=REL` werden für weitere Berechnungen relative Fehlerschranken ermittelt, wenn möglich.

- `errorflag GetErrMode()`

  Der aktuelle Fehlerschranken-Modus wird zurückgegeben.

- `void SetExactConst()`

  Nach dem Aufruf dieser Funktion werden alle weiteren Literalkonstanten als exakt angesehen. Dies gilt allerdings nicht für bereits initialisierte symbolische Konstanten.

- `void ResetExactConst()`

  Die Wirkung von `SetExactConst()` wird aufgehoben; alle folgenden Literalkonstanten werden wieder normal behandelt.

- `void ResetCounters()`

  Während einem Programmdurchlauf werden die Operationen, für die eine Fehlerabschätzung vorgenommen wurde, automatisch mitgezählt. Absolute und relative Schranken werden dabei getrennt gezählt. Außerdem werden Operationen mit Ergebnis im Unterlauf sowie rundungsfehlerfreie Operationen gezählt. Mit diesem Befehl werden alle Zähler auf Null gesetzt.

- `long int GetCounter(errorflag fl)`

  `fl`: *gibt an, welcher Zähler gemeint ist:*
  `fl==ABS` → *Zähler für Operationen mit absoluter Fehlerschranke,*
  `fl==REL` → *Zähler für Operationen mit relativer Fehlerschranke.*

  Liefert den Stand des entsprechenden Zählers.

- `long int GetAbsCounter()`

  Liefert den Stand des Zählers für Operationen mit absoluter Fehlerschranke.

- `long int GetRelCounter()`

  Liefert den Stand des Zählers für Operationen mit relativer Fehlerschranke.

- `long int GetUflCounter()`

  Liefert den Stand des Zählers für Operationen mit Ergebnis im Unterlauf.

- `long int GetExactCounter()`

  Liefert den Stand des Zählers für Operationen ohne Rundungsfehler.

### Funktionen zur Abschätzung der Grundoperationen

- ```
  BoundType ErrAdd(BoundType& x, BoundType& y)
  BoundType ErrSub(BoundType& x, BoundType& y)
  BoundType ErrMul(BoundType& x, BoundType& y)
  ```
  ```

`BoundType ErrDiv(BoundType& x, BoundType& y)`

`x`: *1. Operand*
`y`: *2. Operand*

Diese Funktionen berechnen in Abhängigkeit vom aktuellen Fehlerschranken-Modus[1] eine Schranke für den Fehler, der bei einer gerundeten Operation mit gestörten Operanden entsteht.

Ist der absolute Fehler-Modus eingestellt, so hat man mit dieser Schranke — sie sei `ErrMax` genannt — die Abschätzung

$$|(a \circ b) - (\tilde{a} \boxdot \tilde{b})| \leq \texttt{ErrMax}$$

für alle $a \in$ `x.value` und $b \in$ `y.value` und für alle $\tilde{a}, \tilde{b}$ mit

$$|a - \tilde{a}| \leq \begin{cases} \texttt{x.err} & \text{für } \texttt{x.select=ABS} \\ \texttt{x.err} \cdot |a| & \text{für } \texttt{x.select=REL} \end{cases}$$

und

$$|b - \tilde{b}| \leq \begin{cases} \texttt{y.err} & \text{für } \texttt{y.select=ABS} \\ \texttt{y.err} \cdot |b| & \text{für } \texttt{y.select=REL}. \end{cases}$$

Im relativen Fehler-Modus gilt Entsprechendes.

Als Ergebnis wird ein `BoundType` Objekt, das eine Einschließung der exakten Wertemenge der jeweiligen Operation, die berechnete Fehlerschranke und ein Flag zur Kennzeichnung der Art der Schranke enthält, zurückgegeben.

- `real DeltaDatAdd(BoundType& x, BoundType& y)`
  `real DeltaDatSub(BoundType& x, BoundType& y)`
  `real DeltaDatMul(BoundType& x, BoundType& y)`
  `real DeltaDatDiv(BoundType& x, BoundType& y)`

  `x`: *1. Operand*
  `y`: *2. Operand*

  Diese Funktionen werden von `ErrAdd`, `ErrSub`, `ErrMul` bzw. `ErrDiv` verwendet, um die Fortpflanzung des absoluten Datenfehlers bei exakter Rechnung abzuschätzen. Mit

  $$\texttt{AbsDatErr} = \texttt{DeltaDat}\,Op(\texttt{x}, \texttt{y});$$

  gilt dann für $\circ = +, -, \cdot$ bzw. $/$

  $$|(a \circ b) - (\tilde{a} \circ \tilde{b})| \leq \texttt{AbsDatErr}$$

  für alle $a \in$ `x.value` und $b \in$ `y.value` und für alle $\tilde{a}, \tilde{b}$ mit

  $$|a - \tilde{a}| \leq \begin{cases} \texttt{x.err} & \text{für } \texttt{x.select=ABS} \\ \texttt{x.err} \cdot |a| & \text{für } \texttt{x.select=REL} \end{cases}$$

[1] Falls `REL` eingestellt ist, werden generell relative Fehlerschranken berechnet. Sollte dies aufgrund der Tatsache, dass die Einschließung des Ergebnisses die Null enthält, einmal nicht möglich sein, wird auf den absoluten Fehler ausgewichen. In diesem Fall wird eine Warnung ausgegeben, wenn dies vom Benutzer verlangt wird (siehe `SetSwitchTest()`).

und

$$|b - \tilde{b}| \leq \begin{cases} \texttt{y.err} & \text{für } \texttt{y.select=ABS} \\ \texttt{y.err} \cdot |b| & \text{für } \texttt{y.select=REL}. \end{cases}$$

- ```
  real EpsDatAdd(BoundType& x, BoundType& y)
  real EpsDatSub(BoundType& x, BoundType& y)
  real EpsDatMul(BoundType& x, BoundType& y)
  real EpsDatDiv(BoundType& x, BoundType& y)
  ```

  x: *1. Operand*
  y: *2. Operand*

  Entsprechend `DeltaDatAdd`, `DeltaDatSub`, `DeltaDatMul` und `DeltaDatDiv` für die Fortpflanzung des relativen Datenfehlers.

- ```
  bool RndFreeAdd_U(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeSub_U(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeMul_U(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeDiv_U(BoundType& x, BoundType& y, interval& ResultSet)
  ```

  x: *1. Operand*
  y: *2. Operand*
  ResultSet: *Einschließung für* $\texttt{x.}\widetilde{\texttt{value}}\diamondsuit\texttt{y.}\widetilde{\texttt{value}}$

  Diese Funktionen werden von `ErrAdd`, `ErrSub`, `ErrMul` bzw. `ErrDiv` verwendet, um zu prüfen, ob eine Operation mit Ergebnis im Unterlaufbereich rundungsfehlerfrei durchführbar ist.

- ```
  bool RndFreeAdd_N(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeSub_N(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeMul_N(BoundType& x, BoundType& y, interval& ResultSet)
  bool RndFreeDiv_N(BoundType& x, BoundType& y, interval& ResultSet)
  ```

  x: *1. Operand*
  y: *2. Operand*
  ResultSet: *Einschließung für* $\texttt{x.}\widetilde{\texttt{value}}\diamondsuit\texttt{y.}\widetilde{\texttt{value}}$

  Entsprechend `RndFreeAdd_U`, `RndFreeSub_U`, `RndFreeMul_U` und `RndFree- Div_U` für Operationen mit Ergebnis im normalisierten Bereich.

### Funktionen zur Abschätzung der mathematischen Standardfunktionen

- ```
  BoundType ErrSqrt(BoundType& x)
  BoundType ErrExp (BoundType& x)
  BoundType ErrLn  (BoundType& x)
  ...
  ```

  x: *Argument der Standardfunktion*

  Mit Hilfe der Funktion `Delta`*Fkt* bzw. `Eps`*Fkt* wird in Abhängigkeit vom aktuellen Fehlerschranken-Modus eine Schranke `ErrMax` für den Fehler berechnet,

der bei Anwendung der Maschinenapproximation einer mathematischen Standardfunktion auf ein gestörtes Argument entsteht.

Im absoluten Fehler-Modus z. B. würde mit dieser Schranke die Ungleichung

$$|f(a) - \tilde{f}(\tilde{a})| \leq \texttt{ErrMax}$$

für alle $a \in$ `x.value` und für alle $\tilde{a}$ mit

$$|a - \tilde{a}| \leq \begin{cases} \texttt{x.err} & \text{für } \texttt{x.select=ABS} \\ \texttt{x.err} \cdot |a| & \text{für } \texttt{x.select=REL} \end{cases}$$

gelten.

Als Ergebnis wird ein `BoundType` Objekt, das eine Einschließung der exakten Wertemenge, die Fehlerschranke und ein Flag zur Kennzeichnung der Art der Schranke enthält, zurückgegeben.

- ```
  real DeltaSqrt(BoundType& x)
  real DeltaExp (BoundType& x)
  real DeltaLn  (BoundType& x)
  ...
  ```

  `x:` *Argument der Standardfunktion*

  Diese Funktionen berechnen die eigentliche absolute Fehlerabschätzung für `ErrSqrt`, `ErrExp`, `ErrLn` usw.

- ```
  real EpsSqrt(BoundType& x)
  real EpsExp (BoundType& x)
  real EpsLn  (BoundType& x)
  ...
  ```

  `x:` *Argument der Standardfunktion*

  Entsprechend `DeltaSqrt`, `DeltaExp`, `DeltaLn`,... für die relativen Fehlerabschätzungen.

## 2.4 Die Klasse `BoundType`

### Datenelemente

- `interval value`

  Intervall zur Einschließung der exakten Werte.

- `real err`

  Zugehöriger maximaler absoluter oder relativer Fehler.

- `errorflag select`

  Flag zur Kennzeichnung von `err` als absolut bzw. relativ.

### Standardkonstruktor und -destruktor

- `BoundType()`
- `~BoundType()`

### Konstruktoren mit C++ und C-XSC Datentypen

Falls `ExactConst` mit `SetExactConst()` gesetzt worden ist, wird bei all diesen Konstruktoren `err` auf 0 und `select` gemäß dem aktuellen Fehler-Modus gesetzt — im absoluten Modus (nach `SetErrMode(ABS)`) also auf `ABS` und im relativen Modus (nach `SetErrMode(REL)`) auf `REL`.

- `BoundType(int n)`
  `BoundType(long int n)`

  `n`: *Wert, mit dem das Objekt initialisiert werden soll*

  `value` wird mit dem Punktintervall $[\mathtt{n},\mathtt{n}]$ initialisiert. `err` wird auf 0 gesetzt, da ganzzahlige Werte nicht mit Konvertierungsfehlern behaftet sind. `select` wird nach dem aktuellen Fehler-Modus-Flag gesetzt.

- `BoundType(char* s)`

  `s`: *zeigt auf eine Zeichenkette, die eine Zahl im* `double`*-Format darstellt*

  Das Objekt wird mittels `s >> *this` initialisiert; siehe `char* operator>> (char* s, BoundType& x)`.

- `BoundType(double x)`
  `BoundType(real& x)`

  `x`: *Wert, mit dem das Objekt initialisiert werden soll*

  Gleitpunkt-Zahlen sind in der Regel mit Konvertierungsfehlern behaftet. Deshalb wird `x` hier durch den Vorgänger und den Nachfolger[2] im Gleitpunktraster eingeschlossen (vgl. Abbildung 1). Für den Fehler werden folgende Fälle der Reihe nach betrachtet:

  1. $\mathtt{value} \cap \mathtt{UnflowRange} \neq \emptyset$
     (a) Arithmetik mit „gradual underflow“
     $\leadsto$ `err` := `dMinReal`, `select` := `ABS`
     (b) Arithmetik ohne „gradual underflow“
     $\leadsto$ `err` := `MinReal`, `select` := `ABS`
  2. absoluter Fehler-Modus $\leadsto$ `err` $:= \overline{\varepsilon} \cdot |$`value`$|$, `select` := `ABS`

[2]Eine Einschließung durch Vorgänger und Nachfolger ist notwendig, da i. allg. nicht bekannt ist, in welche Richtung der exakte Wert in das Gleitpunktraster gerundet worden ist.

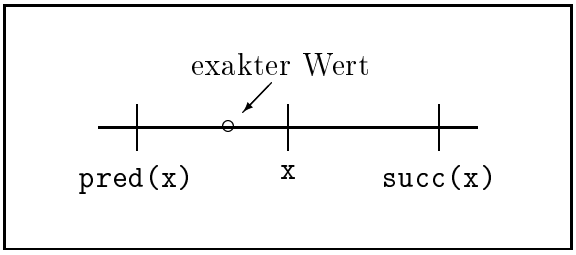



Abbildung 1: Einschließung einer gerundeten Zahl durch Vorgänger und Nachfolger

3. relativer Fehler-Modus $\leadsto$ `err` $:= \overline{\varepsilon}$, `select` := `REL`

- `BoundType(interval& X)`

  `X`: *Intervall, mit dem das Objekt initialisiert werden soll*

  `X` wird selbst als Einschließung verwendet. Der Benutzer muss selbst dafür sorgen, dass der gewünschte Wert bzw. Bereich tatsächlich von `X` eingeschlossen wird. Für den Fehler werden folgende Fälle der Reihe nach betrachtet:

  1. `value` $\subseteq$ `UnflowRange`
     (a) Arithmetik mit „gradual underflow“
     $\leadsto$ `err` := `dMinReal`, `select` := `ABS`
     (b) Arithmetik ohne „gradual underflow“
     $\leadsto$ `err` := `MinReal`, `select` := `ABS`
  2. `value` $\cap$ `UnflowRange` $\neq \emptyset$
     (a) Arithmetik mit „gradual underflow“
     $\leadsto$ `err` := $\max($`dMinReal`$, \overline{\varepsilon} \cdot |$`value`$|)$, `select` := `ABS`
     (b) Arithmetik ohne „gradual underflow“
     $\leadsto$ `err` := $\max($`MinReal`$, \overline{\varepsilon} \cdot |$`value`$|)$, `select` := `ABS`
  3. absoluter Fehler-Modus $\leadsto$ `err` $:= \overline{\varepsilon} \cdot |$`value`$|$, `select` := `ABS`
  4. relativer Fehler-Modus $\leadsto$ `err` $:= \overline{\varepsilon}$, `select` := `REL`

### Konstruktoren zum expliziten Setzen der Datenelemente

- `BoundType(char* s, char* e, errorflag fl)`

  `s`: *zeigt auf eine Zeichenkette, die eine Zahl im* `double`*-Format darstellt*
  `e`: *zeigt auf eine Zeichenkette, die den Fehler* `r` *im* `double`*-Format darstellt*
  `fl`: *Flag zur Kennzeichnung des Fehlers*

Die durch `s` dargestellte Zahl wird maximal genau in ein Gleitpunktintervall eingeschlossen; `err` wird mit der evtl. nach oben gerundeten Zahl `r` und `select` mit `fl` initialisiert.

- `BoundType(real& x, real& e, errorflag fl)`

  `x`: *Wert, der zugewiesen werden soll*
  `e`: *zugehöriger Fehler*
  `fl`: *Flag zur Kennzeichnung des Fehlers*

  `value` wird mit dem Punktintervall $[\mathtt{x}, \mathtt{x}]$, `err` mit `e` und `select` mit `fl` initialisiert. Der Benutzer muss dafür Sorge tragen, dass keine Konvertierungsfehler entstehen, d. h. `x` sollte eine Maschinenzahl sein, und `e` sollte auch bei einer Rundung nach unten groß genug sein, um den exakten Fehler von `x` zu beschränken.

- `BoundType(interval& X, char* e, errorflag fl)`

  `X`: *Intervall, das zugewiesen werden soll*
  `e`: *zeigt auf eine Zeichenkette, die den zugehörigen Fehler* `r` *im* `double`*-Format darstellt*
  `fl`: *Flag zur Kennzeichnung des Fehlers*

  Das Intervall `X` wird als Einschließung verwendet, `err` mit der evtl. nach oben gerundeten Zahl `r` und `select` mit `fl` initialisiert.

- `BoundType(interval& X, real& e, errorflag fl)`

  `X`: *Intervall, das zugewiesen werden soll*
  `e`: *zugehöriger Fehler*
  `fl`: *Flag zur Kennzeichnung des Fehlers*

  `value` wird mit dem Intervall `X`, `err` mit `e` und `select` mit `fl` initialisiert. Der Benutzer muss darauf achten, dass `X` den gewünschten Wert bzw. Bereich tatsächlich einschließt und `e` auch bei einer Rundung nach unten noch groß genug ist, um den Fehler von `X` zu beschränken.

### Zuweisungsoperatoren

Neben dem Standardzuweisungsoperator

- `BoundType& operator= (BoundType& x)`,

der die Datenelemente von `x` in das aufrufende Objekt kopiert, werden wegen der impliziten Typumwandlung durch die verschiedenen Konstruktoren keine weiteren Zuweisungsoperatoren benötigt.

### Ein- und Ausgabeoperatoren

- `char* operator>> (char* s, BoundType& x)`

  `s`: *zeigt auf eine Zeichenkette, die eine Zahl* `r` *im* `double`*-Format darstellt*
  `x`: *Variable, der die Zahl* `r` *zugewiesen werden soll*

Falls `ExactConst` mit `SetExactConst()` gesetzt worden ist, wird `x.value` mit der `r` am nächsten gelegenen Maschinenzahl initialisiert. `err` wird auf 0 und `select` nach dem aktuellen Fehler-Modus-Flag gesetzt.

Sonst wird `r` maximal genau in ein Intervall eingeschlossen, welches `x.value` zugewiesen wird. Für den Fehler werden dann folgende Fälle der Reihe nach betrachtet, wobei $a := \inf($`x.value`$)$ und $b := \sup($`x.value`$)$:

1. Arithmetik mit „gradual underflow“:
   (a) $a = b \quad \leadsto$ `x.err` $:= 0.0$, `x.select` entsprechend aktuellem Fehler-Modus-Flag
   (b) $[a,b] \cap$ `UnflowRange` $\neq \emptyset \quad \leadsto$ `x.err` $:=$ `dMinReal`, `x.select` $:=$ `ABS`
   (c) absoluter Fehler-Modus $\leadsto$ `x.err` $:= \overline{\varepsilon} \cdot |[a,b]|$, `x.select` $:=$ `ABS`
   (d) relativer Fehler-Modus $\leadsto$ `x.err` $:= \overline{\varepsilon}$, `x.select` $:=$ `REL`
2. Arithmetik ohne „gradual underflow“:
   (a) $a = b,\ 0 \neq a \in$ `UnflowRange` $\leadsto$ `x.err` $:=$ `MinReal`, `x.select` $:=$ `ABS`
   (b) $a = b \quad \leadsto$ `x.err` $:= 0.0$, `x.select` entsprechend aktuellem Fehler-Modus-Flag
   (c) $[a,b] \cap$ `UnflowRange` $\neq \emptyset \quad \leadsto$ `x.err` $:=$ `MinReal`, `x.select` $:=$ `ABS`
   (d) absoluter Fehler-Modus $\leadsto$ `x.err` $:= \overline{\varepsilon} \cdot |[a,b]|$, `x.select` $:=$ `ABS`
   (e) relativer Fehler-Modus $\leadsto$ `x.err` $:= \overline{\varepsilon}$, `x.select` $:=$ `REL`

- `istream& operator>> (istream& s, BoundType& x)`

  `s`: *Eingabestream*
  `x`: *Variable, der die aus* `s` *ausgelesene Zahl zugewiesen werden soll*

  Siehe `char* operator>> (char* s, BoundType& x)`.

- `ostream& operator<< (ostream& s, BoundType& x)`

  `s`: *Ausgabestream*
  `x`: *Variable, die auf* `s` *ausgegeben werden soll*

  `x` wird im Format `value AbsErr= err` für `select=ABS` bzw. `value RelErr= err` für `select=REL` auf `s` ausgegeben.

### Arithmetische Grundoperationen

- `BoundType operator- (BoundType& x)`

  `x`: *Operand*

  Negation: Es wird `BoundType(-x.value, x.err, x.select)` zurückgegeben. Der Fehler bleibt von der Negation unberührt.

- `BoundType operator+ (BoundType& x, BoundType& y)`
  `BoundType operator- (BoundType& x, BoundType& y)`

```
BoundType operator* (BoundType& x, BoundType& y)
BoundType operator/ (BoundType& x, BoundType& y)
```

`x`: *1. Operand*
`y`: *2. Operand*

Der Rückgabewert wird von `ErrAdd(x,y)`, `ErrSub(x,y)`, `ErrMul(x,y)` bzw. `ErrDiv(x,y)` geliefert (siehe Unterabschnitt 2.3).

- ```
  BoundType& operator+= (BoundType& x, BoundType& y)
  BoundType& operator-= (BoundType& x, BoundType& y)
  BoundType& operator*= (BoundType& x, BoundType& y)
  BoundType& operator/= (BoundType& x, BoundType& y)
  ```

  `x`: *1. Operand*
  `y`: *2. Operand*

  Kurzschreibweise: `x ○= y` statt `x = x ○ y`.

### Vergleichsoperatoren

Die Vergleichsoperatoren verhalten sich wie im Punktfall: Ein Vergleich `x ◇ y`, $\diamond \in \{=, \neq, <, \leq, >, \geq\}$, ist

- wahr, wenn $a \diamond b \quad \forall a \in \mathtt{x}.\widetilde{\mathtt{value}}\ \forall b \in \mathtt{y}.\widetilde{\mathtt{value}}$,
- falsch, wenn $\neg(a \diamond b) \quad \forall a \in \mathtt{x}.\widetilde{\mathtt{value}}\ \forall b \in \mathtt{y}.\widetilde{\mathtt{value}}$,
- unentscheidbar sonst (in diesem Fall wird mit einer Fehlermeldung abgebrochen).

Die logische Negation `!x` wird wie `x==0` behandelt.

- ```
  int operator! (BoundType& x)
  ```

  `x`: *Operand*

  Negation

- ```
  int operator== (BoundType& x, BoundType& y)
  int operator!= (BoundType& x, BoundType& y)
  int operator<  (BoundType& x, BoundType& y)
  int operator<= (BoundType& x, BoundType& y)
  int operator>  (BoundType& x, BoundType& y)
  int operator>= (BoundType& x, BoundType& y)
  ```

  `x`: *1. Operand*
  `y`: *2. Operand*

  Vergleiche
  ```

### Mathematische Standardfunktionen

- `BoundType sqrt(BoundType& x)`

  `x:` *Argument*

  Wurzelfunktion: Rückgabewert ist `ErrSqrt(x)`.

- `BoundType sqr(BoundType& x)`

  `x:` *Argument*

  Quadratfunktion: Die Einschließung wird mit `sqr(x.value)` und der Fehler über `ErrMul(x,x)` berechnet.

- `BoundType exp(BoundType& x)`

  `x:` *Argument*

  Exponentialfunktion: Rückgabewert ist `ErrExp(x)`.

- `BoundType log(BoundType& x)`

  `x:` *Argument*

  Natürlicher Logarithmus: Rückgabewert ist `ErrLn(x)`.

usw.

### Funktionen zum Lesen der Datenelemente

- `interval GetValue (BoundType& x)`

  `x:` *bezogenes Objekt*

  Gibt `x.value` zurück.

- `interval GetRange (BoundType& x)`

  `x:` *bezogenes Objekt*

  Ermittelt den Wertebereich

$$\mathtt{x}.\widetilde{\mathtt{value}} = \begin{cases} \mathtt{x.value} + [-\mathtt{x.err}, \mathtt{x.err}], & \text{falls } \mathtt{x.select=ABS}, \\ \mathtt{x.value} \cdot [1 - \mathtt{x.err}, 1 + \mathtt{x.err}], & \text{falls } \mathtt{x.select=REL}. \end{cases}$$

- `real GetErr (BoundType& x)`

  `x:` *bezogenes Objekt*

  Liefert den Fehler `x.err`.

- `real GetErr (BoundType& x, errorflag fl)`

  `x:` *bezogenes Objekt*
  `fl:` *gibt an, ob der absolute oder der relative Fehler zurückgegeben werden soll*

  Bestimmt in Abhängigkeit von `fl` den absoluten bzw. relativen Fehler von `x`.

- `real GetAbsErr (BoundType& x)`

  `x`: *bezogenes Objekt*

  Berechnet den absoluten Fehler von `x`.

- `real GetRelErr (BoundType& x)`

  `x`: *bezogenes Objekt*

  Berechnet den relativen Fehler von `x`. Falls `x` mit dem absoluten Fehler `x.err` $\neq 0$ behaftet ist und $0 \in$ `x.value` gilt, wird mit einer Fehlermeldung abgebrochen.

- `errorflag GetErrFlag (BoundType& x)`

  `x`: *bezogenes Objekt*

  Liefert das Fehlerflag `x.select`.

**Funktionen zum expliziten Setzen der Datenelemente**

- `void SetValue (BoundType& x, char* s)`

  `x`: *bezogenes Objekt*
  `s`: *zeigt auf eine Zeichenkette, die eine Zahl im* `double`*-Format darstellt*

  Die durch `s` gegebene Gleitpunktzahl wird maximal genau in ein Intervall eingeschlossen; mit diesem Intervall wird `x.value` belegt.

- `void SetValue (BoundType& x, real& y)`

  `x`: *bezogenes Objekt*
  `y`: *neuer Wert*

  `x.value` wird mit dem Punktintervall $[$`y`$,$`y`$]$ belegt.

- `void SetValue (BoundType& x, interval& Y)`

  `x`: *bezogenes Objekt*
  `Y`: *neue Einschließung*

  `Y` wird `x.value` zugewiesen.

- `void SetAbsErr (BoundType& x, char* e)`

  `x`: *bezogenes Objekt*
  `e`: *zeigt auf eine Zeichenkette, die den Fehler im* `double`*-Format darstellt*

  Die durch `e` gegebene Gleitpunktzahl wird evtl. nach oben gerundet und `x.err` zugewiesen; `x.select` wird auf `ABS` gesetzt.

- `void SetRelErr (BoundType& x, char* e)`

  `x`: *bezogenes Objekt*
  `e`: *zeigt auf eine Zeichenkette, die den Fehler im* `double`*-Format darstellt*

Die durch `e` gegebene Gleitpunktzahl wird evtl. nach oben gerundet und `x.err` zugewiesen; `x.select` wird auf `REL` gesetzt.

- `void SetAbsErr (BoundType& x, real& e)`

  `x`: *bezogenes Objekt*
  `e`: *neuer absoluter Fehler von* `x`

  `e` wird `x.err` zugewiesen und `x.select` auf `ABS` gesetzt.

- `void SetRelErr (BoundType& x, real& e)`

  `x`: *bezogenes Objekt*
  `e`: *neuer relativer Fehler von* `x`

  `e` wird `x.err` zugewiesen und `x.select` auf `REL` gesetzt.

- `void SetErr (BoundType& x, char* e, errorflag fl)`

  `x`: *bezogenes Objekt*
  `e`: *zeigt auf eine Zeichenkette, die den Fehler im* `double`*-Format darstellt*
  `fl`: *Flag zur Kennzeichnung des Fehlers*

  Die durch `e` gegebene Gleitpunktzahl wird evtl. nach oben gerundet und `x.err` zugewiesen; `x.select` wird gemäß `fl` gesetzt.

- `void SetErr (BoundType& x, real& e, errorflag fl)`

  `x`: *bezogenes Objekt*
  `e`: *neuer Fehler von* `x`
  `fl`: *Flag zur Kennzeichnung des Fehlers*

  `e` wird `x.err` zugewiesen und `x.select` gemäß `fl` gesetzt.

### Verschiedene Funktionen

- `BoundType sign (BoundType& x)`

  `x`: *Argument*

  Diese Funktion ermittelt das Vorzeichen von `x` nach folgender Vorschrift:

  - $sgn = 0$, falls $\texttt{x.}\widetilde{\texttt{value}} = 0$,
  - $sgn = 1$, falls $\inf(\texttt{x.}\widetilde{\texttt{value}}) \geq 0$,
  - $sgn = -1$, falls $\sup(\texttt{x.}\widetilde{\texttt{value}}) \leq 0$,
  - sonst wird mit einer Fehlermeldung abgebrochen.

  Es wird der Wert `BoundType(`$sgn$`, 0.0,` $flag$`)` zurückgegeben, wobei $flag$ dem aktuellen Fehlerschranken-Modus entspricht.

- `BoundType fabs(BoundType& x)`
  `BoundType abs (BoundType& x)`

`x`: *Argument*

Diese Funktionen berechnen den Absolutbetrag von `x`; der Fehler bleibt unberührt. Es wird `BoundType (abs(x.value), x.err, x.select)` zurückgegeben.

- `BoundType min (BoundType& x, BoundType& y)`

  `x`: *1. Argument*
  `y`: *2. Argument*

  Hier wird das Minimum von `x` und `y` ermittelt:

  - $\min(\texttt{x},\texttt{y}) := \texttt{x}$, falls $\sup(\texttt{x}.\widetilde{\texttt{value}}) \leq \inf(\texttt{y}.\widetilde{\texttt{value}})$,
  - $\min(\texttt{x},\texttt{y}) := \texttt{y}$, falls $\sup(\texttt{y}.\widetilde{\texttt{value}}) \leq \inf(\texttt{x}.\widetilde{\texttt{value}})$,
  - sonst wird mit einer Fehlermeldung abgebrochen.

- `BoundType max (BoundType& x, BoundType& y)`

  `x`: *1. Argument*
  `y`: *2. Argument*

  Hier wird das Maximum von `x` und `y` ermittelt:

  - $\max(\texttt{x},\texttt{y}) := \texttt{x}$, falls $\inf(\texttt{x}.\widetilde{\texttt{value}}) \geq \sup(\texttt{y}.\widetilde{\texttt{value}})$,
  - $\max(\texttt{x},\texttt{y}) := \texttt{y}$, falls $\inf(\texttt{y}.\widetilde{\texttt{value}}) \geq \sup(\texttt{x}.\widetilde{\texttt{value}})$,
  - sonst wird mit einer Fehlermeldung abgebrochen.

- `BoundType rnd_to_n (real& x)`
  `BoundType rnd_to_n (interval& X)`

  `x, X`: *Wert, der zugewiesen werden soll, bzw. eine Intervalleinschließung dafür*

  Erzeugt ein `BoundType` Objekt wie der entsprechende Konstruktor, allerdings wird `Eps53` $= 2^{-53}$ statt $\overline{\varepsilon}$ als Maschinenepsilon verwendet.

- `BoundType exact (real& x)`
  `BoundType exact (interval& X)`

  `x, X`: *Wert, der zugewiesen werden soll, bzw. eine Intervalleinschließung dafür*

  Erzeugt ein `BoundType` Objekt mit `err` $= 0$ und `select` ensprechend dem aktuellen Fehlerschranken-Modus.

# 3 Anwendungsbeispiele

## 3.1 Eine einfache Anwendung

Gegeben sei die Funktion $f(x) := 4.875 \cdot (x+1)^2 - 5.6 \cdot \sqrt{x+1}$ bzw. in Programmform

```
double f(double x)
{
    return 4.875*sqr(x+1)-5.6*sqrt(x+1);
}
```

Gesucht ist eine Abschätzung des absoluten und relativen Fehlers für Argumente im Intervall $[1, 2]$. Dazu muss zunächst die Funktion `double f(double)` an die Bibliothek `Bound` angepasst werden:

```
BoundType f(BoundType x)
{
    return "4.875"*sqr(x+exact(1))-"5.6"*sqrt(x+exact(1));
}
```

Der Datentyp `double` wird durch `BoundType` ersetzt. Dies würde für eine Analyse schon ausreichen. Durch weitere Änderungen kann die Abschätzung evtl. noch verbessert werden:

- Die Konstanten 4.875 und 5.6 können in Stringkonstanten umgewandelt werden, damit sie maximal genau in Intervalle eingeschlossen werden.
- Die Konstante 1 ist im Datentyp `double` exakt darstellbar, weshalb sie mit `exact(1)` gekennzeichnet wird.

Das komplette Programm könnte etwa so aussehen:

```
// Einbinden der Headerdateien:

#include "BoundType.hpp"


// Definition der Funktion f:

BoundType f(BoundType x)
{
    return "4.875"*sqr(x+exact(1))-"5.6"*sqrt(x+exact(1));
}


// Hauptprogramm:

int main()
{
    BoundType x, res;
    SetErrMode(REL);        // relative Fehlerschranken

    // Berechnung
    x= _interval(1.0,2.0);
```

```
    res= f(x);

    // Ausgabe der Ergebnisse:
    cout << "Einschliessung des..." << endl;
    cout << "...exakten Wertebereichs:                    "
         << GetValue(res) << endl;
    cout << "...Wertebereichs bei Gleitkommaauswertung: "
         << GetRange(res) << endl;
    cout << "maximaler abs. Fehler: "
         << RndUp << GetAbsErr(res) << endl;
    if (!(GetValue(res)>=0.0))
        // Wertebereich enthaelt nicht die Null
        cout << "maximaler rel. Fehler: "
             << RndUp << GetRelErr(res) << endl;

    return 0;
}
```

Falls nur Maschinenzahlen betrachtet werden sollen, kann `x= _interval(1.0,2.0)` durch `x= exact(_interval(1.0,2.0))` ersetzt werden. Außerdem können Einschliessung und Fehlerschranken durch verschiedene Intervallunterteilungs-Strategien evtl. verbessert werden.

Programmausgabe:

```
Einschliessung des...
...exakten Wertebereichs:                  [  9.800515, 35.955405]
...Wertebereichs bei Gleitkommaauswertung: [  9.800515, 35.955405]
maximaler abs. Fehler: 1.229930E-013
maximaler rel. Fehler: 3.420709E-015
```

Das Resultat einer gleitkommamäßigen Funktionsauswertung für beliebige Argumente aus dem Intervall $[1, 2]$ stimmt also mit dem exakten Ergebnis mindestens in den ersten 49 Binärstellen bzw. 14 Dezimalstellen überein.

## 3.2 Sensitivitätsanalyse und Stabilisierung eines Programms

Man betrachte folgendes Programm, das zu $p \in [-10^{100}, -10]$ die kleinere der beiden Nullstellen des Polynoms $f(x) = x^2 + px + 1$ berechnet. Es wird dazu die bekannte Formel

$$\hat{x} = -\frac{p}{2} - \sqrt{\left(\frac{p}{2}\right)^2 - 1} \tag{1}$$

verwendet.

```
#include <math.h>
#include <iostream.h>
```

```
int main()
{
    double p, x;

    do {
        cout << "Wert fuer p (-1e100 <= p <= -10)? "; cin >> p;
    } while (p<-1.0e100 || p>-10.0);

    x= -p/2.0-sqrt(p*p/4.0-1.0);
    cout << "x^2 + (" << p << ")*x + 1 = 0  ";
    cout << "fuer  x = " << x << endl;
    cout << 1.0/(-p/2.0-fabs(p)/p*sqrt(p*p/4.0-1.0)) << endl;

    return 0;
}
```

Ausgabebeispiele:

```
Wert fuer p (-1e100 <= p <= -10)? -10
x^2 + (-10)*x + 1 = 0  fuer  x = 0.101021

Wert fuer p (-1e100 <= p <= -10)? -87.5
x^2 + (-87.5)*x + 1 = 0  fuer  x = 0.0114301

Wert fuer p (-1e100 <= p <= -10)? -1000
x^2 + (-1000)*x + 1 = 0  fuer  x = 0.001

Wert fuer p (-1e100 <= p <= -10)? -1e8
x^2 + (-1e+08)*x + 1 = 0  fuer  x = 1.00008e-08

Wert fuer p (-1e100 <= p <= -10)? -1e13
x^2 + (-1e+13)*x + 1 = 0  fuer  x = -0.000119209

Wert fuer p (-1e100 <= p <= -10)? -1e100
x^2 + (-1e+100)*x + 1 = 0  fuer  x = 1.55091e+83
```

Die letzten beiden Lösungen müssen falsch sein, da $\hat{x} \in [0,1]$ für jedes $p \in [-10^{100}, -10]$ gilt! Um dieses Fehlverhalten genauer zu untersuchen soll nun mit Hilfe des folgenden Programms eine Sensitivitätsanalyse durchgeführt werden.

```
#include <math.h>
#include <iostream.h>
#include "BoundType.hpp"    // Bibliothek einbinden

int main()
{
    BoundType p, x;         // neuer Datentyp BoundType
```

```
    //Initialisierungen
    SetSwitchTest();        // Warnung bei Moduswechsel ein
    SetUnderflowTest();     // Warnung bei Unterlauf ein
    SetErrMode(REL);        // relative Fehlerschranken
    ResetCounters();        // alle internen Zaehler zuruecksetzen
    p= _interval(-1.0e100,-10.0);

    // Berechnung
    SetExactConst();        // alle Konstanten sind exakt
    x= -p/2.0-sqrt(p*p/4.0-1.0);
    ResetExactConst();

    // Ausgabe
    cout << "x= " << x << endl;
    cout << GetAbsCounter()
         << " Operationen im absoluten Modus" << endl;
    cout << GetRelCounter()
         << " Operationen im relativen Modus" << endl;
    cout << GetUflCounter()
         << " Operationen mit Ergebnis im Unterlauf" << endl;
    cout << GetExactCounter()
         << " Operationen ohne Rundungsfehler" << endl;

    return 0;
}
```

Das Analyseprogramm liefert folgende Ausgabe.

```
*** ErrSub: Null im Ergebnisintervall - es wird die
*** absolute Fehlerschranke berechnet! Weiter mit <Return>
*** ErrSub: Ergebnis im Unterlauf!  Weiter mit <Return>
x= [-5.000001E+099,5.000001E+099]  AbsErr= 5.620505E+084
1 Operationen im absoluten Modus
5 Operationen im relativen Modus
1 Operationen mit Ergebnis im Unterlauf
2 Operationen ohne Rundungsfehler
```

Bis auf eine Operation konnten alle Operationen im relativen Modus abgeschätzt werden. Schuld daran ist, wie man auch an der Warnung erkennen kann, dass das Ergebnisintervall dieser einen Operation die Null enthält und daher keine relative Fehlerschranke berechnet werden kann. Aus demselben Grund wird auch die Unterlaufwarnung ausgegeben. Zwei der Operationen wurden als rundungsfehlerfrei erkannt; das sind die Divisionen durch 2 bzw. 4.

Wie man an der Einschließung für die Nullstelle und an dem extrem großen absoluten Fehler erkennen kann, hat der verwendete Algorithmus eine sehr schlechte Kondition: Es kann vorkommen, dass das berechnete Ergebnis mit der tatsächlichen Nullstelle

überhaupt nichts mehr zu tun hat; selbst das Vorzeichen muss nicht stimmen. Die Beispiele weiter oben bestätigen dies.

Der Algorithmus kann verbessert werden, wenn statt (1) die dazu äquivalente Formel

$$\hat{x} = \frac{1}{-\frac{p}{2} + \sqrt{\left(\frac{p}{2}\right)^2 - 1}} \tag{2}$$

benutzt wird. Zur Verifikation wird in dem Analyseprogramm die Zeile

```
x= -p/2.0-sqrt(p*p/4.0-1.0);
```

durch die Zeile

```
x= 1.0/(-p/2.0+sqrt(p*p/4.0-1.0));
```

ersetzt und das Programm erneut gestartet. Die Ausgabe ist jetzt:

```
x= [9.999999E-101,  0.101021]  RelErr= 1.124101E-015
0 Operationen im absoluten Modus
7 Operationen im relativen Modus
0 Operationen mit Ergebnis im Unterlauf
2 Operationen ohne Rundungsfehler
```

Man sieht, dass der Algorithmus jetzt sehr gut konditioniert ist. Bei gleitkommamäßiger Auswertung stimmen im Ergebnis mindestens die ersten 47 Binär- bzw. 14 Dezimalziffern mit dem exakten Wert der Nullstelle überein. Diese Aussage gilt nun nachgewiesenermaßen für jedes beliebige $p \in [-10^{100}, -10]$!

**Bemerkung:** Die Fehlerschranke wurde hier mit nur einer einzigen Auswertung gefunden. Über den absoluten Modus, d. h. wenn der absolute Fehler abgeschätzt wird und die absolute Fehlerschranke anschließend durch das Betragsminimum der Einschließung des Wertebereichs geteilt wird, bekommt man hier nur dann eine brauchbare Fehlerschranke, wenn das zu untersuchende Intervall $[-10^{100}, -10]$ relativ fein unterteilt wird. Ganz ohne Unterteilung bricht das Programm sogar ab, da die Wurzel aus einem Intervall gezogen werden müsste, welches negative Zahlen enthält.

## 3.3 Anwendung bei der Realisierung schneller Intervallfunktionen

Als Beispiel dient hier eine Realisierung der Standardfunktion $\arctan(x)$ als Tabellenverfahren (s. [8]). Tabellenverfahren werden in der Numerik zur Realisierung mathematischer Funktionen auf Computern verwendet. Sie zeichnen sich durch ihre Schnelligkeit aus, da gewisse Werte vorab berechnet und dem Computer in Form einer Tabelle zur Verfügung gestellt werden.

Im folgenden soll gezeigt werden, wie a priori Fehlerschranken für den relativen Fehler, der bei gleitpunktmäßiger Auswertung für exakt darstellbare Argumente entsteht, berechnet werden können. Dabei werden nicht nur die Rundungs- und Konstantenfehler sondern auch die Approximationsfehler berücksichtigt.

Die so gewonnenen Fehlerschranken können dann bei der Programmierung einer schnellen Intervallfunktion $\arctan(X)$ mit dem reellen Intervallargument $X$ verwendet werden. Die Grundidee besteht darin, dass die Funktionsauswertung rein gleitkommamäßig und ohne zeitraubendes Setzen des Rundungsmodus vorgenommen wird. Eine sichere Einschließung erhält man dann, indem das berechnete Intervall um den Wert der vorab bestimmten (für alle zulässigen Argumente gültigen) relativen Fehlerschranke aufgebläht wird.

Die zugrundeliegende Theorie wird in [8] folgendermaßen beschrieben:

> „Das in [11] beschriebene Verfahren wird hier modifiziert und auf das IEEE-Datenformat angewandt.
>
> Positive Eingabeargumente $x \in I_1 := [0, \frac{1}{\gamma}]$ werden mit Hilfe der bekannten Summenformel
>
> $$\arctan x = \arctan \frac{x - c_i}{1 + xc_i} + \arctan c_i, \quad xc_i > -1$$
>
> und geeignet gewählter Reduktionskonstanten $c_i$ in ein festgelegtes Approximationsintervall $I_\gamma := [-\gamma, \gamma]$ reduziert. Die benötigten Konstanten $a_i := \arctan c_i$ zur Ergebnisanpassung können im voraus berechnet und tabelliert werden.
>
> Für Argumente $r \in I_\gamma$ wird zur Approximation des arctan im Unterschied zu [11] nicht die abgebrochene Potenzreihe
>
> $$\arctan(r) \approx r \cdot \sum_{k=0}^{N} \frac{(-1)^k}{2k+1} r^{2k}$$
>
> mit Polynomgrad $N$ verwendet, sondern die polynomiale Approximation
>
> $$\arctan(r) \approx r \cdot (1 + r^2 \cdot p_N(r^2)) \tag{3}$$
>
> mit einer Bestapproximation $p_N(r^2) = \sum_{k=0}^{N} d_k \cdot r^{2k}$.
>
> Für sehr kleine Argumente $|r| <$ `q_atnt` kann die Approximation
>
> $$\arctan(r) \approx r$$
>
> verwendet werden. Die Berechnung für Eingabeargumente $x \in I_2 := [\frac{1}{\gamma}, \texttt{maxreal}]$ kann mit
>
> $$\arctan(x) = \arctan(-\frac{1}{x}) + \frac{\pi}{2}$$
>
> ebenfalls auf die Approximation (3) zurückgeführt werden, es gilt $r := \frac{-1}{x} \in I_\gamma$. Für negative Argumente kann die Formel
>
> $$\arctan(-x) = -\arctan(x)$$
>
> angewandt werden.“

In der vorliegenden Implementierung wurde für die Approximation $N := 5$ und für die Argumentreduktion der Wert $\gamma := \frac{1}{8}$ verwendet. Als Reduktionsbereiche sind die Intervalle $B_i := [b_{i-1}, b_i)$ $(i = 1, \ldots, 7)$ und $B_8 := [b_7, \texttt{MaxReal}]$ mit

$$\begin{aligned}
b_0 &:= \texttt{q_atnt} = 1.807032\ldots E-008 \\
b_1 &:= 4503599627370496 \,/\, 36028797018963968 = 0.12500\ldots \\
b_2 &:= 7050717449407908 \,/\, 18014398509481984 = 0.39139\ldots \\
b_3 &:= 6454487157372003 \,/\, 9007199254740992 = 0.71659\ldots \\
b_4 &:= 5343484307627230 \,/\, 4503599627370496 = 1.18649\ldots \\
b_5 &:= 4642583338069965 \,/\, 2251799813685248 = 2.06172\ldots \\
b_6 &:= 5472919125137495 \,/\, 1125899906842624 = 4.86093\ldots \\
b_7 &:= 4503599627370496 \,/\, 562949953421312 = 8.00000\ldots
\end{aligned}$$

angegeben. Über den Fehler im Bereich $[-b_0, b_0]$ wird in [8] folgende Aussage getroffen:

> „Für $|x| \leq 1.807032\ldots E-008$ wird die Approximation $\arctan(x) \approx x$ verwendet. Mit einfacher Handrechnung erhält man den für Ergebnisse im Bereich der normalisierten Zahlen gültigen relativen Approximationsfehler (siehe [11], Seite 50):
>
> $$\left|\frac{\arctan x - x}{\arctan x}\right| \leq \frac{x^2}{3} \cdot \frac{1}{1-x^2} \cdot \frac{1}{1-(1/3)x^2} \leq 1.0885e-16 := \varepsilon(\mathrm{app}_N)$$
>
> Im Bereich der denormalisierten Zahlen gilt der absolute Fehler:
>
> $$|\arctan x - x| \leq \frac{x^2}{3} \cdot \frac{1}{1-x^2} \leq 1.66e-616 \leq \texttt{dMinReal} := \Delta(\mathrm{app}_D)\text{“}$$

Es folgt der Originalquellcode der Implementierung:

```
/*****************************************************************/
/*       fi_lib  --- A fast interval library (Version 1.0)       */
/*       (For copyright and info's look at file "fi_lib.h")      */
/*****************************************************************/

#include "fi_lib.h"


#ifdef LINT_ARGS
local double q_atan(double x)
#else
local double q_atan(x)

double x;
#endif
{
  double res;
  double absx,ym,y,ysq;
  int    ind,sgn;
```

```
  if NANTEST(x)
    res=q_abortnan(INV_ARG,&x,16);
  else {

    if (x<0) absx=-x; else absx=x;
    if (absx<=q_atnt) res=x;
    else {
            if (absx<8) {sgn=1; ym=0;}
            else        {sgn=-1; ym=q_piha; absx=1/absx;}

            ind=0;
            while (absx>=q_atnb[ind+1]) ind+=1;
            y=(absx-q_atnc[ind])/(1+absx*q_atnc[ind]);
            ysq=y*y;
            res = (y+y*(ysq*(((((q_atnd[5]*ysq+q_atnd[4])
                  *ysq+q_atnd[3])*ysq+q_atnd[2])
                  *ysq+q_atnd[1])*ysq+q_atnd[0])))+q_atna[ind];
            if (x<0) res=-(res*sgn+ym);
            else     res= (res*sgn+ym);
          }
  }

  return(res);
}
```

Um eine Gesamtfehlerabschätzung mit dem Fehlerkalkül durchführen zu können, muss der Originalquellcode an die Klasse `BoundType` angepasst und an entsprechender Stelle der Approximationsfehler, der in [8] für die Bestapproximation $p_5$ zu $4.7169 \cdot 10^{-16}$ angegeben ist, berücksichtigt werden:

```
/*       q_piha   = pi/2                                                 */
   BoundType q_piha   = rnd_to_n(7074237752028440.0 / 4503599627370496.0);
// q_piha ist maximal genau!

   BoundType q_atnt=exact(1.807032e-8);

   BoundType q_atna[7]={exact(0),
     rnd_to_n(8960721713639278.0 / 36028797018963968.0),   /* =  0.24871... */
     rnd_to_n(8960721713639278.0 / 18014398509481984.0),   /* =  0.49742... */
     rnd_to_n(6720541285229458.0 /  9007199254740992.0),   /* =  0.74613... */
     rnd_to_n(8960721713639278.0 /  9007199254740992.0),   /* =  0.99484... */
     rnd_to_n(5600451071024549.0 /  4503599627370496.0),   /* =  1.24355... */
     rnd_to_n(6720541285229458.0 /  4503599627370496.0)};  /* =  1.49226... */
// Die Konstanten fuer die Ergebnisanpassung sind maximal genau!

   BoundType q_atnb[8]={exact(0),
     exact(4503599627370496.0 / 36028797018963968.0),   /* =  0.12500... */
     exact(7050717449407908.0 / 18014398509481984.0),   /* =  0.39139... */
     exact(6454487157372003.0 /  9007199254740992.0),   /* =  0.71659... */
     exact(5343484307627230.0 /  4503599627370496.0),   /* =  1.18649... */
     exact(4642583338069965.0 /  2251799813685248.0),   /* =  2.06172... */
     exact(5472919125137495.0 /  1125899906842624.0),   /* =  4.86093... */
     exact(4503599627370496.0 /   562949953421312.0)};  /* =  8.00000... */
   BoundType q_atnc[7]={exact(0),
```

```
     exact(4575085335741456.0 / 18014398509481984.0),   /* =  0.25397... */
     exact(4890523484394786.0 /  9007199254740992.0),   /* =  0.54296... */
     exact(8326197945442628.0 /  9007199254740992.0),   /* =  0.92439... */
     exact(6934969934934130.0 /  4503599627370496.0),   /* =  1.53987... */
     exact(6633650527568543.0 /  2251799813685248.0),   /* =  2.94593... */
     exact(7153270512133541.0 /   562949953421312.0)};  /* = 12.70676... */
   BoundType q_atnd[6]={
    exact(-6004799503160653.0 / 18014398509481984.0),   /* = -0.33333... */
    exact( 7205759403717662.0 / 36028797018963968.0),   /* =  0.20000... */
    exact(-5146970946471129.0 / 36028797018963968.0),   /* = -0.14286... */
    exact( 8006368526669049.0 / 72057594037927936.0),   /* =  0.11111... */
    exact(-6546869189017288.0 / 72057594037927936.0),   /* = -0.09086... */
    exact( 5323534481176937.0 / 72057594037927936.0)};  /* =  0.07388... */
// Konstantenfehler der Argumentreduktion und der Polynomkoeffizienten
// sind bereits im Approximationsfehler enthalten!


BoundType q_atan(BoundType x)
{
  BoundType res;
  BoundType absx,ym,y,ysq;
  int    ind,sgn;

  if (x<0) absx=-x; else absx=x;
  if (absx<=q_atnt) res=x;
  else {
          if (absx<8) {sgn=1; ym=0;}  // Integerkonstanten werden automatisch
          else        {sgn=-1; ym=q_piha; absx=1/absx;}  // als exakt erkannt

          ind=0;
          while (absx>=q_atnb[ind+1]) ind+=1;
          y=(absx-q_atnc[ind])/(1+absx*q_atnc[ind]);
          ysq=y*y;
          res = ((((q_atnd[5]*ysq+q_atnd[4])
                *ysq+q_atnd[3])*ysq+q_atnd[2])
                *ysq+q_atnd[1])*ysq+q_atnd[0];

          // absoluten Approximationsfehler aufaddieren:
          if (GetErrFlag(res)==ABS)
            SetAbsErr(res, addu(GetAbsErr(res), 4.7169E-016));
          else
            SetRelErr(res,
              addu(GetRelErr(res), abs2rel(GetValue(res), 4.7169E-016)));

          res = (y+y*(ysq*res))+q_atna[ind];
          if (x<0) res=-(res*sgn+ym);
          else     res= (res*sgn+ym);
        }

  return(res);
}
```

Die folgende Prozedur berechnet mit Hilfe des Bisektionsverfahrens eine obere und eine untere Schranke für den relativen Fehler einer gegebenen Funktion über einem

gegebenen Intervall. Sie benötigt einen Datentyp `istack`, der Stacks für Intervalle repräsentiert. Die Implementierung von `istack` wird hier nicht aufgeführt.

```
void maxerr(BoundType (*pf)(BoundType),
            interval bereich,
            real acc,
            real& MinErr,
            real& MaxErr,
            int& opcount)
  //------------------------------------------------------------
  // Berechnung einer Schranke fuer den maximalen Fehler einer
  // Funktion ueber einem Intervall
  //------------------------------------------------------------
  // INPUT:
  //   (*pf)(BoundType): zu untersuchende Funktion
  //   bereich:          zu untersuchendes Intervall
  //   acc:              relative Genauigkeit der zu berechnenden Schranke
  // OUTPUT:
  //   MinErr:           Unterschranke fuer den maximalen Fehler
  //   MaxErr:           Oberschranke fuer den maximalen Fehler
  //   opcount:          Zaehler fuer die Anzahl der untersuchten
  //                     Teilintervalle
  //------------------------------------------------------------
{
  real rd, lblow, ublow, blow, Unterschranke, Oberschranke, old_MaxErr;
  interval x;
  BoundType erg;
  int i;
  bool fail;
  istack stack, ok;

  //------------------------------------------------------------
  // Initialisierungen
  //------------------------------------------------------------
  create(stack); create(ok);
  opcount= 0;
  MaxErr= MaxReal;
  ublow= 10.0; lblow= 0.0;

  // Berechnung einer unteren Schranke fuer den maximalen Fehler durch
  // Auswertung an 100 aequidistanten Stellen im Intervall "bereich"
  Unterschranke= Max (
          GetRelErr((*pf)(exact(succ(Inf(bereich))))),
          GetRelErr((*pf)(exact(pred(Sup(bereich))))));
  for ( i=1 ; i<100 ; i++ ) {
    rd= i/100.0*diam(bereich)+Inf(bereich);
    Unterschranke= Max(Unterschranke, GetRelErr((*pf)(exact(rd))));
  }

  MinErr= Unterschranke;
  if (Unterschranke<Eps52) Unterschranke= Eps52;
  push(stack, bereich);

  //------------------------------------------------------------
```

```
  // Iteration zur Berechnung einer Oberschranke, die der
  // Genauigkeit 'acc' genuegt
  //-------------------------------------------------------------
  while (ublow-lblow > acc) {
    blow= (lblow+ublow)/2;
    clear(ok);         // Stack loeschen
    fail= false;
    Oberschranke= muld(Unterschranke,addd(1.0,blow));
    old_MaxErr= MaxErr;
    MaxErr= 0.0;

    // Bisektionsverfahren ---------------------------------------
    while (!empty(stack)) {
      x= pop(stack);
      erg= (*pf)(exact(x));
      opcount++;
      if ((MinAbs(GetValue(erg))!=0.0 && GetRelErr(erg)<=Oberschranke)
          || GetErr(erg)==0.0) {
      // Fehler ueber Teilintervall ist kleiner als vorgegebene Schranke
        push(ok,x);
        MaxErr= Max(MaxErr, GetRelErr(erg));
      }
      else if ((diam(x)/MaxAbs(x)<Eps52)||(diam(x)<=dMinReal)) {
      // Intervall nicht weiter teilbar -> Abbruch
        push(stack,x);
        fail= true;
        break;
      }
      else {
      // Fehler ueber Teilintervall ist groesser als vorgegebene Schranke
      // -> Intervallunterteilung
        push(stack,_interval(Inf(x),mid(x)));
        push(stack,_interval(mid(x),Sup(x)));
      }
    } // endwhile - Bisektionsverfahren

    if (fail) {         // vorgegebene Schranke nicht eingehalten
      MaxErr= old_MaxErr;
      lblow= blow;
    }
    else {              // vorgegebene Schranke eingehalten
      stack= ok;
      ublow= Min(blow, subu(divu(MaxErr, Unterschranke), 1.0));
    }
  } // endwhile - Iteration
}
```

Das komplette Analyseprogramm könnte in etwa so aussehen:

```
#include <time.h>
#include "BoundType.hpp"


// Bereiche B1 bis B8
interval bereich[8]= {
```

```
   _interval(1.807032e-8,4503599627370496.0/36028797018963968.0),
   _interval(4503599627370496.0/36028797018963968.0,
             7050717449407908.0/18014398509481984.0),
   _interval(7050717449407908.0/18014398509481984.0,
             6454487157372003.0/ 9007199254740992.0),
   _interval(6454487157372003.0/ 9007199254740992.0,
             5343484307627230.0/ 4503599627370496.0),
   _interval(5343484307627230.0/ 4503599627370496.0,
             4642583338069965.0/ 2251799813685248.0),
   _interval(4642583338069965.0/ 2251799813685248.0,
             5472919125137495.0/ 1125899906842624.0),
   _interval(5472919125137495.0/ 1125899906842624.0,
             4503599627370496.0/  562949953421312.0),
   _interval(4503599627370496.0/  562949953421312.0,MaxReal)
};

// weitere globale Konstanten
...

BoundType q_atan(BoundType x)
{
...
}

void maxerr(BoundType (*pf)(BoundType),
            interval bereich,
            real acc,
            real& MinErr,
            real& MaxErr,
            int& opcount)
{
...
}


int main ()
{
  int i, opcount, totcount;
  double time, tottime;
  real MinErr, MaxErr, minulp, maxulp;

  SetExactCompare();
  SetErrMode(REL);

  totcount= 0;
  tottime= clock();

  for (i=1 ; i<=8 ; i++)
  {
    cout << SetPrecision(18,15) << Scientific;
    time= clock();
    maxerr(q_atan, bereich[i-1], 1.0e-5, MinErr, MaxErr, opcount);
    time= (clock()-time)/CLOCKS_PER_SEC;
    minulp= divu(MinErr,Eps52);
```

```
    maxulp= divu(MaxErr,Eps52);
    totcount+= opcount;
    cout << "Bereich [" << RndDown << Inf(bereich[i-1]) << ","
         << RndUp << Sup(bereich[i-1]) << "]:\n";
    cout << "MaxRelErr.Inf= " << RndUp << MinErr << " = " << minulp << " ulp\n";
    cout << "MaxRelErr.Sup= " << RndUp << MaxErr << " = " << maxulp << " ulp\n";
    if (opcount==1) cout << "1 Funktionsauswertung\n";
    else cout << opcount << " Funktionsauswertungen\n";
    cout << SetPrecision(5,1) << Fixed << "Zeit: " << time << "sec.\n\n";
  }
  if (totcount==1) cout << "Total: 1 Funktionsauswertung\n";
  else cout << "Total: " << totcount << " Funktionsauswertungen\n";
  cout << SetPrecision(5,1) << Fixed << "       "
       << (clock()-tottime)/CLOCKS_PER_SEC << "sec.\n";

  return 0;
}
```

Wird das Programm übersetzt und gestartet, erhält man die folgende Ausgabe:

```
Bereich [1.807031999999999E-008,1.250000000000000E-001]:
MaxRelErr.Inf= 2.340924987030341E-016 = 1.054258889929213E+000 ulp
MaxRelErr.Sup= 2.340940051571515E-016 = 1.054265674395415E+000 ulp
230 Funktionsauswertungen
Zeit: 1.21sec.

Bereich [1.250000000000000E-001,3.913934426229509E-001]:
MaxRelErr.Inf= 1.104507255702020E-015 = 4.974258465207624E+000 ulp
MaxRelErr.Sup= 1.130005764518850E-015 = 5.089093540013602E+000 ulp
339 Funktionsauswertungen
Zeit: 1.76sec.

Bereich [3.913934426229508E-001,7.165920254261774E-001]:
MaxRelErr.Inf= 6.124632367739199E-016 = 2.758289204913154E+000 ulp
MaxRelErr.Sup= 6.124680777230450E-016 = 2.758311006609830E+000 ulp
111 Funktionsauswertungen
Zeit: 0.82sec.

Bereich [7.165920254261773E-001,1.186491862010193E+000]:
MaxRelErr.Inf= 5.108234135382082E-016 = 2.300544134862800E+000 ulp
MaxRelErr.Sup= 5.108267656702379E-016 = 2.300559231523359E+000 ulp
111 Funktionsauswertungen
Zeit: 0.84sec.

Bereich [1.186491862010192E+000,2.061721166266557E+000]:
MaxRelErr.Inf= 4.680603675917073E-016 = 2.107956497072911E+000 ulp
MaxRelErr.Sup= 4.680631813320814E-016 = 2.107969169033011E+000 ulp
106 Funktionsauswertungen
Zeit: 0.81sec.

Bereich [2.061721166266556E+000,4.860928659711213E+000]:
MaxRelErr.Inf= 4.182243172256942E-016 = 1.883514879214917E+000 ulp
MaxRelErr.Sup= 4.182255810876710E-016 = 1.883520571143245E+000 ulp
96 Funktionsauswertungen
Zeit: 0.76sec.
```

```
Bereich [4.860928659711212E+000,8.000000000000000E+000]:
MaxRelErr.Inf= 4.217238013923566E-016 = 1.899275154803886E+000 ulp
MaxRelErr.Sup= 4.226567046019213E-016 = 1.903476577350855E+000 ulp
472 Funktionsauswertungen
Zeit: 2.33sec.

Bereich [8.000000000000000E+000,1.797693134862316E+308]:
MaxRelErr.Inf= 3.822252756279683E-016 = 1.721389608889704E+000 ulp
MaxRelErr.Sup= 3.822263978588683E-016 = 1.721394662968366E+000 ulp
5156 Funktionsauswertungen
Zeit: 19.89sec.

Total: 6621 Funktionsauswertungen
       28.42sec.
```

Der maximale relative Fehler der Funktion `q_atan` für beliebige zulässige Argumente beträgt also $\varepsilon_{\texttt{q\_atan}} := 1.131 \cdot 10^{-15} = 5.090 \cdot 2^{-52}$. Man kann jetzt also bei jeder gleitkommamäßigen Auswertung von `q_atan(x)` sicher sein, dass das exakte Ergebnis $\arctan(x)$ im Intervall $[\texttt{q\_atan(x)} \cdot (1 - \varepsilon_{\texttt{q\_atan}}), \texttt{q\_atan(x)} \cdot (1 + \varepsilon_{\texttt{q\_atan}})]$ liegt.[3]

# 4 Schlussbemerkungen

Vergleicht man die C++ Implementierung zum Vorwärtsfehlerkalkül mit einer Pascal-XSC Implementierung, so stellt man fest, dass letztere doch aus wesentlich eleganterem (leichter lesbaren, weniger fehleranfälligen) Quellcode besteht. Trotzdem ist die hier besprochene C-XSC Implementierung wichtig, da sie unmittelbar zur Fehlerkontrolle von C bzw C++ Quellen verwendet werden kann. Ein Umschreiben der C Quellen nach Pascal ist nicht mehr nötig. Vielmehr können durch einfache Eingriffe in die C Quellen diese so modifiziert werden, dass automatisch eine a priori worst-case Fehlerschranke für den vorgegebenen Gleitkommaalgorithmus berechnet wird.

Die Beispiele zeigen, dass der Kalkül eine enorme Erleichterung beim bestimmen von verlässlichen Fehlerschranken darstellt. Gerade bei Algorithmen, die heuristisch so getrimmt sind, dass sie nur wenig sensitiv auf Rundungs- und Datenfehler reagieren, kann dies Verhalten mit den besprochenen Werkzeugen mathematisch bewiesen werden. Die Qualität der dabei gewonnenen Fehlerschranken ist in diesen Fällen gut.

Bei zu schlechter Kondition können die Fehlerschranken unrealistisch groß werden. Trotzdem ist das Verfahren auch in solchen Fällen wertvoll, da es in einem Algorithmus gerade die Stellen offenlegt, an denen es zu großer Fehlerverstärkung kommt. Der Algorithmus kann dann gezielt verbessert werden.

# Literatur

[1] Armin Bantle. *Ein Kalkül für verläßliche rechnergestützte Fehlerabschätzungen und dessen Anwendung.* Diplomarbeit am IWRMM, Universität Karlsruhe, 1998.

[3] Im Bereich der denormalisierten Zahlen gilt $\arctan(x) \in [\texttt{q\_atan(x)} - \texttt{dMinReal}, \texttt{q\_atan(x)} + \texttt{dMinReal}]$.

[2] Armin Bantle, Walter Krämer. *Ein Kalkül für verläßliche absolute und relative Fehlerabschätzungen.* Preprint 98/5 des IWRMM, Universität Karlsruhe, 1998.

[3] Gerd Bohlender, Christian Ullrich. *Standards zur Computerarithmetik.* In: Jürgen Herzberger (Hrsg.): *Wissenschaftliches Rechnen*, 9-46, Akademie Verlag, 1995.

[4] Michael Bräuer, Walter Krämer. *Rückwärtsmethode zur automatischen Berechnung von worst-case Fehlerschranken.* Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, Bericht 1/1999, Karlsruhe, 1999.

[5] R. Hammer, M. Hocks, U. Kulisch, D. Ratz. *C++ Toolbox for Verified Computing.* Springer-Verlag, Heidelberg, 1995. ISBN 3-540-59110-9.

[6] Werner Hofschuster, Walter Krämer. *Ein rechnergestützter Fehlerkalkül mit Anwendung auf ein genaues Tabellenverfahren.* Preprint 96/5 des IWRMM, Universität Karlsruhe, 1996.

[7] Werner Hofschuster, Walter Krämer. *A Computer Oriented Approach to Get Sharp Reliable Error Bounds, Reliable Computing 3*, pp. 239-248, 1997.

[8] Werner Hofschuster, Walter Krämer. *Eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-Double-Format.* Preprint 98/7 des IWRMM, Universität Karlsruhe, 1998.

[9] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985.* Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[10] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, A. Wiethoff. *C-XSC: A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, Heidelberg, 1993. xvi+269 S. ISBN 3-540-56328-8.

[11] Walter Krämer. *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate.* Dissertation, Universität Karlsruhe, 1987.

[12] Walter Krämer. *Eine Fehlerfaktorarithmetik für zuverlässige a priori Fehlerabschätzungen.* Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, Bericht 5/1997, 21 Seiten, Karlsruhe, 1997.

[13] Walter Krämer. *Constructive Error Analysis.* Journal of Universal Computer Science (JUCS), Vol. 4, No. 2, pp. 147-163, 1998.

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