libF
eine lange Gleitpunktarithmetik

Diplomarbeit von
Thomas Papanikolaou

angefertigt nach einem Thema von
Prof. Dr. Johannes Buchmann
am Fachbereich 14 Informatik
der Universität des Saarlandes.

12. Juli 1993

Hiermit versichere ich an Eides statt, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen benutzt habe.

Saarbrücken, den 12. Juli 1993

Thomas Papanikolaou

Mein Dank gilt zuerst Herrn Professor Buchmann. Er gab mir die Anregung, an diesem interessanten Thema zu arbeiten.

Ebenso möchte ich mich bei meiner Familie, der Familie Chalkidis und der Familie Becker bedanken, weil sie immer für mich da waren wenn ich sie brauchte.

Ferner möchte ich Jörg Zayer, Jutta Bartholomes und Damian Weber für Ihre Diskussionsbereitschaft und das Korrekturlesen danken.

Schließlich danke ich allen Mitgliedern der Lehrstuhlfamilie für das gute Arbeitsklima.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschreibt die theoretischen Grundlagen und die Implementierung der mathematischen Umgebung libF. Der Name steht für Floating point library. libF ermöglicht Operationen mit **langen**[1] reellen Zahlen. Dabei können die Grundrechenarten mit einer Präzision bis 300000 Dezimalstellen durchgeführt werden.

libF kann auf drei verschiedene Arten benutzt werden:

- als C-Bibliothek, falls das C-Include File eingebunden wird,
- als C++ Bibliothek, falls das C++ Include File eingebunden wird[2] und
- als interaktiver Interpreter.

libF wird begleitet von einer Komplexen- und einer Polynomarithmetik über den komplexen Zahlen (libC, libP). Beide sind in C++ implementiert und können nur in C++ benutzt werden. Ein Verfahren zur Nullstellenbestimmung über Polynomen mit komplexen (und daher auch reellen und ganzzahligen) Koeffizienten schließt das Paket ab.

Im folgenden wird, nach einer kurzen theoretischen Einführung im Kapitel 2, das Paket beschrieben:

- Kapitel 3 beschreibt die Zahldarstellung; zuerst wird die lange Integer Arithmetik präsentiert, worauf libF aufgebaut wird, und dann der neue Typ, der lange reelle Zahlen darstellt.
- Kapitel 4 beschreibt die Algorithmen, die libF für die Grundoperationen und die transzendente Funktionen benutzt.
- Kapitel 5 beschreibt den C++ Teil von libF. libC und libP sowie das Verfahren der Polynomnullstellenbestimmung werden dann kurz erklärt.

[1] Als **lang** werden die Zahlen bezeichnet, die nicht in einem Maschinentyp (int, double) gespeichert werden können.

[2] Diese Variabilität der Nutzung folgt dem in Dagstuhl (1991) vorgeschlagenen C++ Bibliothek Standard.

- Anhang A enthält Laufzeittests aller nichttrivialen Funktionen.
- Anhang B enthält die vollständige Beschreibung eines Beispiels in C und C++. Thema: die Implementierung der Cosinus Funktion mit libF.
- Anhang C ist ein Manual, das dem schnellen Überblick dient.

Am Ende der Arbeit befindet sich zusätzlich ein Stichwort- und ein Literaturverzeichnis.

Vereinbarungen

In dieser Arbeit werden Programmstücke in `typewriter Schrift` geschrieben. Wichtige Begriffe und Definitionen werden in **bold Schrift**, mathematische Formeln und Konstanten in *italic Schrift* widergegeben.

Literaturhinweise werden in eckigen Klammern gegeben (z.B [1]).

Kapitel 2

Einführung

In diesem Kapitel werden grundlegende Definitionen angegeben, die den Begriff der **floating point** (= Gleitkomma) Zahlen spezifizieren sollen: diese Definitionen sind zu einem großen Teil aus [12] entnommen.

2.1 Definitionen

Sei $x \in \mathbb{R}$, $b \in \mathbb{N}_{>1}$. Dann ist die **b-adische Darstellung** von x gegeben durch

$$x = \pm \sum_{i=-\infty}^{\infty} m_i \cdot b^i, \textit{wobei } 0 \leq m_i < b.$$

Betrachtet man nun diese Formel, so können viele interessante Eigenschaften entdeckt werden, von denen folgende für uns von Bedeutung sind:

1. Falls $\exists j \in \mathbb{N}_{\geq 0} : \forall i < 0$ und $\forall i > j,\ m_i = 0$ ist, beschreibt die obige Formel die b-adische Darstellung einer ganzen Zahl.
2. Falls $m_i = 0,\ \forall i \geq 0$ ist, dann liegt x im Intervall $]-1, 1[$.

In dieser Form ist obige Formel nicht für eine endliche Darstellung geeignet. Deshalb wollen wir nun den Weg zeigen, wie aus diesen Eigenschaften eine präzise Vorstellung über das Aussehen der Gleitkommazahlen gewonnen werden kann:

2.1.1. Definition *Eine* **floating point Zahl** *ist ein Tripel (m, e, b), wobei* $m \in]-1,1[$, $e \in \mathbb{Z}$ *und* $b \in \mathbb{N}_{>1}$.

Diese Schreibweise ist die abgekürzte Form der Zahl $m \cdot b^e$, wobei m die **Mantisse**, e der **Exponent** und b die **Basis** der Darstellung sind. Da $|m| < 1$, steht das **Gleitkomma** (auch **Radix** genannt) ganz links.

2.1.1. Beispiel Das Tripel *(.31415, 1, 10)* repräsentiert die Zahl *3.1415* oder (in der obigen Form geschrieben) die Zahl $0.31415 \cdot 10^1$. In diesem Beispiel wurde die Mantisse mit 5 Ziffern dargestellt.

2.1.2. Definition *Sei x=(m, e, b) eine floating point Zahl, dann heißt die Anzahl der Ziffern von m* **Präzision** *von x.*

Dadurch wird das endliche Aussehen der Darstellung festgelegt. Sei nun m die Mantisse einer floating point Zahl und p ihre Präzision. Aus den obigen Definitionen folgt dann, daß $m \cdot b^p$ eine ganze Zahl ist. Weiter gilt : $-b^p < m \cdot b^p < b^p$.

2.1.1. Bemerkung Es ist üblich, daß man den Exponenten e so **balanciert**, daß am Ende die Zahl die Form $m \cdot b^E$ besitzt mit $E = e - q$. Dieser Vorgang ermöglicht das Arbeiten mit *kleinen* Exponenten. Die positive ganze Zahl q heißt **Bias**.

2.1.2. Beispiel Sei $x = m \cdot b^e$, $0 \leq e \leq M$, M positive ganze Zahl. Wähle $q = \frac{M}{2}$. Balanciert man den Exponenten der Zahl x mit q, so hat man $x = m \cdot b^E$ mit $|E| \leq \frac{M}{2}$.

2.2 Normalisierte Darstellung

2.2.1. Definition *Eine floating point Zahl (m, e, b) heißt* **normalisiert**, *falls*

1. *die signifikanteste Ziffer (d.h. die erste Nachkommastelle) ihrer Mantisse ungleich Null ist, d.h.* $\frac{1}{b} \leq |m| < 1$.
2. *oder* $m = 0$ *und e seinen minimalen Wert hat.*

2.2.1. Beispiel Sei $x = 0.062435 \cdot 10^2$; x ist nicht normalisiert. Dagegen ist $x = 0.624350 \cdot 10^1$ normalisiert.

Diese weitere Einschränkung des Aussehens der Mantisse (erste Ziffer $\neq 0$) liefert aus verschiedenen Gründen Vorteile:

- Das Vergleichen zweier floating point Zahlen kann auf das Vergleichen der Exponenten (und der Mantissen, falls die Exponenten gleich sind) zurückgeführt werden.
- Die Zahldarstellung ist einheitlich.
- Die relativen Fehler, die bei Rechenoperationen entstehen, können im voraus berechnet werden.

2.3 Fehleranalyse

Eine unserer wichtigsten Aufgaben ist es, die Genauigkeit eines Rechenresultats zu beurteilen. Wir unterscheiden dabei drei Arten von Fehlern:

- Fehler in den Eingabedaten,
- Rundungsfehler,

- Darstellungsfehler[1]

Fehler der zweiten Art können minimiert werden, indem die Genauigkeit der Zwischenrechnungen erhöht wird. Eingabe- und Darstellungsfehler sind dagegen unvermeidbar.

2.3.1. Beispiel Die Zahl *1.1* hat im Binärsystem eine unendliche periodische Darstellung. Sie kann also bei der Eingabe nie exakt dargestellt werden, da notwendigerweise nach endlich vielen Schritten die Rechnung abbrechen muß.

Um den Begriff des relativen Fehlers zu erfassen, definieren wir eine Rundungsfunktion, die eine beliebige (möglicherweise periodische) reelle Zahl x auf eine endliche normalisierte Zahl x' abbildet.

2.3.1. Definition *Sei* $x = sign(x) \cdot |m| \cdot b^e = sign(x) \cdot |0.m_1 m_2 ... m_t m_{t+1} ...| \cdot b^e$. *Bilde*

$$m' = \begin{cases} 0.m_1 m_2 ... m_t \quad \ldots\ldots & m_{t+1} < \frac{b}{2} \\ 0.m_1 m_2 ... m_t + b^{-t} & m_{t+1} \geq \frac{b}{2} \end{cases}$$

Dann heißt x' = **round***(x)* $= sign(x) \cdot m' \cdot b^e$, *die* **t-stellige Rundung** *von x.*

Somit gilt für den **relativen Fehler** von round(x) folgende Abschätzung

$$\left| \frac{round(x) - x}{x} \right| \leq \frac{\frac{b}{2} \cdot b^{-(t+1)}}{|m|} \leq \frac{b}{2} \cdot b^{-t},$$

da $|m| \geq \frac{1}{b}$. Setzen wir nun $eps = \frac{b}{2} \cdot b^{-t}$, so folgt aus der obigen Ungleichung

$$round(x) = x \cdot (1 + \epsilon), \ |\epsilon| \leq eps.$$

Die Zahl eps heißt auch **Maschinengenauigkeit** und kann minimiert werden, indem mehr Stellen bei der Eingabedarstellung benutzt werden. Je nach Genauigkeitseinstellung kann in libF eps zwischen 10^{-28} und $10^{-300000}$ variieren.

Wie sich diese fehlerhafte (auch wenn der Fehler winzig ist) Darstellung auf die Rechenresultate auswirkt (**Fehlerfortpflanzung**), ist nicht Thema dieser Arbeit. Für eine genaue Beschreibung der numerischen Problematik, möge der Leser in [17] nachschlagen.

Im weiteren berücksichtigen wir vor allem folgende Grundregel: die Subtraktion zweier nah beieinander liegender Zahlen ist fehlerverstärkend.

[1] Als Darstellungsfehler bezeichnen wir die Fehler, die schon vor der Rundung aufgrund der endlichen Darstellung entstehen.

Kapitel 3

Zahldarstellung

3.1 libI: eine lange ganzzahlige Arithmetik

Grundstein jeder mathematischen Bibliothek ist eine ganzzahlige Arithmetik. In libF ist sie dafür zuständig die Mantisse der Gleitkommazahlen darzustellen und die erforderlichen Mantisseoperationen durchzuführen. libF ist auf der Arithmetik libI aufgebaut, ein Paket für lange ganzzahlige Operationen, implementiert von R. Dentzer [9].

3.2 Warum libI?

Wie das nächste Kapitel verdeutlichen wird, hängt die Qualität einer reellen Arithmetik zu einem großen Teil von den Eigenschaften ihrer ganzzahligen Umgebung ab. Folgende Aspekte haben lange vor der Implementierung von libF zur Wahl dieser Arithmetik geführt:

1. Operationen werden schneller durchgeführt als in anderen, im Moment existierenden, Integerarithmetiken.
2. Durch ein effizientes Konzept wird minimaler Speicher verbraucht.
3. libI ist portabel zwischen verschiedenen Rechnerarchitekturen.

Die beiden ersten Aspekte sind entscheidend: libI ist genauso für rechen- als auch für speicherintensive Operationen geeignet. Die Portabilität ist ein Faktor, der diese Entscheidung weiterstützt (Ausführliche Tests und Vergleiche mit anderen arithmetischen Systeme wie *maple* oder *pari*, befinden sich im Anhang A).

3.3 Der Datentyp Integer

Der Typ, der lange ganze Zahlen speichert heißt **Integer** und ist wie folgt definiert:

```
typedef struct {
              PLACE *vec;
              int maxlength, length, sign;
} Integer, *pInteger;
```

Der Typ *PLACE* ist dynamisch und je nach Architektur 16 oder 32 Bit lang. *vec* ist ein Zeiger auf PLACE-Worte, in denen der Betrag der ganzen Zahl gespeichert wird. *maxlength* enthält die Anzahl der reservierten PLACE-Worte. Diese Variable wird benutzt, um zu überprüfen, ob für eine Zahl genügend Speicher allociert ist, um ein Ergebnis aufzunehmen. Falls dies nicht der Fall ist, wird neuer Speicherplatz reserviert. *length* ist die aktuelle Länge der Zahl. Falls *length* kleiner wird als *maxlength*, wird Speicherplatz freigegeben. Dadurch werden im Sinne von Platzbedarf Zahlen klein gehalten. *sign* speichert das Vorzeichen der langen Zahl.

3.3.1. Bemerkung *Für jede Integer Zahl können höchstens* 2^{16} *PLACE Worte allociert werden. Der Inhalt von sign ist 1, falls die Zahl kleiner als Null ist, 0 sonst.*

3.4 Namensgebung

Es werden für die Namen der Funktionen folgende Abkürzungen verwendet:

I	Integer	pl	plus	sl	shift left	eq	equal
P	PLACE	mi	minus	sr	shift right	ge	greater equal
int	int	mu	multiply	as	assign	gt	greater than
long	long	di	quotient	c	construct	le	less equal
u	unsigned	re	remainder	d	delete	lt	less than

Somit heißt die Addition zweier langer Zahlen *IasIplI*, was *Integer gleich Integer plus Integer* bedeutet. Man beachte, daß die Ergebnisvariablen **vorne** stehen.

3.5 Operationen mit langen Zahlen

Die Funktionen von libl können in 5 Kategorien unterteilt werden:

1. Speicherverwaltung
2. Eingabe — Ausgabe
3. Typüberprüfung — Zuweisung
4. Vergleiche
5. Arithmetik

3.5.1 Speicherverwaltung

Zu dieser Kategorie gehören die *Konstruktoren* und *Destruktoren* sowie *Imalloc* und *Ifree*.

cI(&x)	x = *ncI()	kreiert ein Integer x
cIasint(&x,i)	x = *ncIasint(i)	kreiert ein Integer x = int
cIasuint(&x,i)		kreiert ein Integer x = unsigned int
cIaslong(&x,i)		kreiert ein Integer x = long
cIasulong(&x,i)		kreiert ein Integer x = unsigned long
cIasI(&x,&y)	x = *ncIasI(&y)	kreiert ein Integer x = Integer y
cImaxlength(&x,i)	x = *ncImaxlength(i)	kreiert ein Integer x mit maxlength = i
dI(&x)	ddI(&x)	gibt c- bzw. n-konstruierten Integer frei

Imalloc bzw. Ifree sind andere Namen für die C-Funktionen malloc bzw. free.

3.5.2 Eingabe — Ausgabe

Die Ein- und Ausgabe langer Zahlen erfolgt in einer C-ähnlichen Weise:

fscanI(f,&x)	liest einen Integer von der Datei f
fprintI(f,&x)	schreibt einen Integer in die Datei f
Itoa(&x,str)	wandelt einen Integer in einen String um
atoI(str,&x)	wandelt einen String in einen Integer um

3.5.3 Typüberprüfung — Zuweisung

Um Zahlen zu erkennen, die in Maschinentypen gespeichert werden können, hat libI folgende 4 Funktionen:

Iisint(&x)	Iisuint(&x)	Iislong(&x)	Iisulong(&x)

Alle Funktionen liefern 1, falls x *int*, ..., *ulong ist*, 0 sonst. Die Zuweisung erfolgt über

IasI(&x,&y)	x = y	i = intasI(&x)	i = x
Iasint(&x,i)	x = i	ui = uintasI(&x)	ui = x
Ias0(&x)	x = 0	l = longasI(&x)	l = x
Ias1(&x)	x = 1	ul = ulongasI(&x)	ul = x

3.5.4 Vergleiche

Vergleiche sind nur zwischen langen Zahlen und zwischen langen Zahlen und Null implementiert. Zwei weitere Funktionen testen Gleichheit mit 1 und Teilbarkeit durch 2.

IeqI(&x,&y)	x == y	Ieq0(&x)	x == 0
IgtI(&x,&y)	x > y	Igt0(&x)	x > 0
IgeI(&x,&y)	$x \geq y$	Ige0(&x)	$x \geq 0$
IltI(&x,&y)	x < y	Ilt0(&x)	x < 0
IleI(&x,&y)	$x \leq y$	Ile0(&x)	$x \leq 0$
Ieven(&x)	!(x%2)	Ieq1(&x)	x == 1

3.5.5 Arithmetik

Es stehen folgende Funktionen zur Verfügung:

IasIplI(&res,&x,&y)	res = x + y
IplasI(&res,&x)	res += x
Iinc(&x)	x++
IasImiI(&res,&x,&y)	res = x − y
ImiasI(&res,&x)	res −= x
Idec(&x)	x−−
Ineg(&x)	x = −x
IasImuI(&res,&x,&y)	res = x * y
ImuasI(&res,&x)	res ∗ = x
IasImuP(&res,&x,P)	res = x ∗ P
ImuasP(&res,P)	res ∗ = P
IasIdiI(&res,&x,&y)	res = x / y
IdiasI(&res,&x)	res /= x
P = IasIdiP(&res,&y,Q)	P = y % Q , res = y / Q, signed
P = IdiasP(&x,Q)	P = x % Q , x /= Q, signed
IasIreI(&res,&x,&y)	res = x % y
IreasI(&res,&x)	res %= x
Idiv(&q,&r,&x,&y)	x = y∗q + r, (x/y=q , x%y=r), signed
uIdiv(&q,&r,&x,&y)	x = y∗q + r, (x/y=q , x%y=r), unsigned
P = uIasIdiP(&res,&y,Q)	P = y % Q res = y / Q, unsigned
P = uIdiasP(&x,Q)	P = x % Q, x /= Q, unsigned
IasIslint(&res,&x,i)	res = x << i
Islasint(&res,i)	res <<= i
IasIsrint(&res,&x,i)	res = x >> i
Israsint(&res,i)	res >>= i
Isr1(&res)	res >>= 1
Idgcd(&res,&x,&y)	res = gcd(x,y) (Euclid)
Ibgcd(&res,&x,&y)	res = gcd(x,y) (binary Euclid)
Ielba(&res,&a,&b,&x,&y)	res = gcd(x,y) = a∗x + b∗y
Ibelba(&res,&a,&b,&x,&y)	res = gcd(x,y) = a∗x + b∗y, binary
Ireduce(&a,&b)	a = a / gcd(a,b), b = b / gcd(a,b)

3.6 Der Datentyp Float

Wir können nun zum neuen Typ übergehen. **Float** ist eine Erweiterung des Typs Integer, die durch

```
typedef struct {
               Integer m;
               long e;
               int type;
} Float, *pFloat;
```

definiert wird. m repräsentiert die Mantisse[1]: sie wird zur Basis $b = 2^{32}$ dargestellt. Da der Exponent e immer im int-Bereich liegt, also kleiner als b ist, brauchen wir nur ein Wort für seine Darstellung.

type ist eine Variable, die den Typ der Gleitkommazahl enthält. Ist *type* gleich 0, so haben wir es mit einer ganzen Zahl zu tun; ist *type* gleich 1, so liegt eine reelle Zahl vor.

3.7 Vorteile — Nachteile

Float ist ein **struct**[2], der in sich einen weiteren **struct** enthält; dies bedeutet kleine Verzögerungen beim Zugriff auf die Inhalte der Mantisse. Dieser Nachteil wird in libF dadurch minimiert, daß Funktionen mit Pointer-Parametern aufgerufen werden. Andererseits ist der klare Aufbau eines **struct**s wichtig für das übersichtliche Programmieren mit den Bibliotheksfunktionen.

type ist mit 32-Bit eigentlich zu groß, um nur die 0 und die 1 zu verwalten. Allerdings gibt es so die Möglichkeit, neue Typen zu schaffen[3]. Außerdem kann man mittels *type* im Binärsystem exakt darstellbare Zahlen effizient speichern. So wird z.B. die Zahl 2.0 bei Präzision 300000 Dezimalstellen, immer noch mit m = 2, e = 1 und *type* = 1 dargestellt! Man kann weiter durch *type* bestimmen, ob eine Operation ganzzahlig oder reell durchgeführt werden soll, indem man die type-Komponente der Operanden auf 0 oder 1 setzt (eigentlich reicht es im zweiten Fall aus, wenn einer der Operanden den Typ 1 hat).

[1] Da der Gleitpunkt immer links steht, simulieren wir $0.m$ mit dem Integer m.

[2] Siehe [11], Seite 129.

[3] Man könnte z.B die int-Koeffizienten eines Polynoms in der Mantisse und den Grad im Exponent speichern und 2 als Polynomtyp vereinbaren (d.h $type = 2$ setzen).

Kapitel 4

Algorithmen

In diesem Kapitel werden die Algorithmen aller nichttrivialen Funktionen der libF beschrieben.

4.1 Vereinbarungen

Wir benutzen **t**, um die gewünschte Genauigkeit zur Basis 2 zu bezeichnen. Weiter benutzen wir e_x (bzw. m_x) für den Exponenten (bzw. die Mantisse) von x. Alle Variablen, die in den Algorithmen zum ersten Mal auftreten, sind *per default* mit 0 initialisiert. Die benutzten Formeln für die transzendenten Funktionen sind in [1] zu finden, es sei denn ihre Quelle wird explizit angegeben. Weiter bedeutet "="" *Zuweisung* und "==" *Test auf Gleichheit.* Temporäre Variablen werden mit tmp, tmp_1, ... bezeichnet.

Für eine Gleitkommazahl x, liefert $type(x)$ 1, falls x eine reelle Zahl ist, 0 sonst. $sign(x)$ liefert das Vorzeichen 1, 0, -1 und $trunc(x)$ den ganzzahligen Anteil von x. $log(x)$ bezeichnet immer den natürlichen Logarithmus von x. Schließlich bezeichnen wir mit SPA eine $single-precision\ Approximation$.

4.2 Grundoperationen

Die Algorithmen für die Grundoperationen sind modifizierte Versionen aus [12]; ihre Korrektheit wird dort nachgewiesen.

4.2.1 Addition — Subtraktion

Dieser Algorithmus berechnet die Summe $z = x + y$ (bzw. die Differenz, falls y durch $-y$ ersetzt wird) für zwei normalisierte Float Zahlen x, y. Dazu werden Integeraddition und Shifting benötigt.

1. **[Trivialfall]** Falls $x == 0$ (bzw. $y == 0$), setze $z = y$ (bzw. $z = x$) und gehe zu Schritt 8.
2. **[Integeraddition]** Falls $type(x) == type(y) == 0$, dann setze $m_z = m_x + m_y$ (Integer Addition), $e_z = 0$, $type_z = 0$ und gehe zu Schritt 8.
3. **[Exponent Test]** Falls $e_x < e_y$, vertausche x und y.
4. **[Setze e_z]** Setze $e_z = e_x$.
5. **[Shift nach Rechts]** Dividiere m_y durch $2^{e_x - e_y}$.
6. **[Addiere]** Setze $m_z = m_x + m_y$ (Integeraddition), $type_z = 1$.
7. **[Normalisiere]** Normalisiere.
8. **[Rückgabe]** Liefere z zurück.

4.2.2 Multiplikation — Division

Gegeben seien zwei normalisierte Float Zahlen x und y. Folgender Algorithmus berechnet ihr Produkt $z = x \cdot y$ (bzw. ihren Quotienten, falls y durch $\frac{1}{y}$ ersetzt wird); dabei werden die Integermultiplikation und -division benötigt.

1. **[Integer Operation]** Falls $type(x) == type(y) == 0$, dann setze $m_z = m_x \cdot m_y$ (bzw. $m_z = \lfloor \frac{m_x}{m_y} \rfloor$; die Division wird ganzzahlig durchgeführt), $e_z = 0$, $type_z = 0$ und gehe zu Schritt 4.
2. **[Operation]** Setze $m_z = m_x \cdot m_y$ (bzw. $m_z = \frac{m_x}{m_y}$; in diesem Fall wird die Division reell durchgeführt), $e_z = e_x + e_y$ (bzw. $e_z = e_x - e_y$) und $type_z = 1$.
3. **[Normalisiere]** Normalisiere das Ergebnis.
4. **[Rückgabe]** Liefere z zurück.

Die reelle Division wird nach der folgenden Methode durchgeführt:

$$m_z = \frac{2^t \cdot m_x}{m_y}, \; e_z = e_x - e_y - t$$

4.2.3 Normalisierung

Nach einer Operation zwischen zwei Float Zahlen kann es sein, daß die Mantisse mehr Ziffern als die momentane Präzision hat (z.B bei der Multiplikation), oder daß sie größer als 1 (bzw. kleiner als $\frac{1}{2}$) ist. In diesem Fall ist das Ergebnis der Operation nicht normalisiert (vgl. Kapitel 2). Der folgende Algorithmus normalisiert eine (möglicherweise) nichtnormalisierte Zahl x. Dazu sei t die momentane Präzision.

1. **[Mantisse Test]** Falls $m_x = 0$, setze $e_x = -t$ und gehe zu Schritt 7. Falls $m_x \geq 1$, gehe zu Schritt 4.

2. **[Ist m_x normalisiert?]** Falls $m_x \geq \frac{1}{2}$, gehe zu Schritt 5.

3. **[Shift nach links]** Setze $m_x = 2 \cdot m_x$, $e_x = e_x - 1$. Gehe zu Schritt 2.

4. **[Shift nach rechts]** Setze $m_x = \frac{m_x}{2}$, $e_x = e_x + 1$.

5. **[Rundung]** Schneide m_x nach der (t+1)ten-Stelle ab; Falls diese Stelle gleich 1 ist, setze $m_x = m_x + 2^{-t}$. Solange $m_x \geq 1$, gehe zu Schritt 4.

6. **[Exponent Test]** Falls e_x zu groß (bzw. zu klein) ist, so liegt ein **Exponentenüberlauf** vor; x kann also bei Präzision t nicht als normalisierte Zahl dargestellt werden. In diesem Fall ist spezielle Behandlung nötig (Erhöhung der Präzision oder Fehlermeldung).

7. **[Rückgabe]** Liefere x zurück.

4.3 Konstanten

libF verfügt über 3 Konstanten:

1. Die Kreiskonstante $\pi = 3.141592653589793...$

2. Die eulersche Konstante $\gamma = \lim_{n\to\infty} \left(\sum_{i=1}^{n} \frac{1}{i} - \log(n) \right) = .5772156649015328...$

3. Die Zahl $e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.718281828459045...$

Diese sind global definiert und werden nur beim ersten Aufruf berechnet, soweit die Präzision t nicht erhöht wird. Das geschieht mit Hilfe des folgenden Schemas:

```
if (Konstante_ist_noch_nicht_konstruiert)
        Konstruiere(Konstante);
if (die_Pr"azision_der_Konstante >= die_momentane_Pr"azision) {
        Weise_die_Konstante_einfach_zu;
        Normalisiere;
} else
        Berechne_die_Konstante_neu;
```

Da in libF die Konstanten eigene Namen besitzen, ist sogar eine direkte Zuweisung möglich. In diesem Fall ist aber der Benutzer allein für die richtige Behandlung der Zuweisung verantwortlich.

4.3.1 Die Zahl π

Wir benutzen folgende Formel, die auf S. Ramanujan [15] zurückgeht:

$$\frac{1}{2\,\pi\,\sqrt{2}} = \frac{1103}{99^2} + \frac{27493}{99^6} \cdot \frac{1}{2} \cdot \frac{1 \cdot 3}{4^2} + \frac{53883}{99^{10}} \cdot \frac{1 \cdot 3}{2 \cdot 4} \cdot \frac{1 \cdot 3 \cdot 5 \cdot 7}{4^2 \cdot 8^2} + \ldots \quad (1)$$

$$\Longleftrightarrow \frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{m=0}^{\infty} \frac{(4\,m)!\,(1103 + 26390\,m)}{(m!)^4\,396^{4\,m}} \quad (2)$$

Gleichheit (1) veranschaulicht die Art und Weise, auf die wir π berechnen. Gleichheit (2) werden wir für die Abschätzung des Restgliedes R_n benutzen.

Reihe (1) läßt sich mit einem Hornerschema-ähnlichen Algorithmus sehr einfach berechnen. Dazu ist es aber notwendig, am Anfang die Anzahl der benötigten Schritte zu berechnen. Wir werden beweisen, daß nach der Aufsummierung von $n = \frac{t}{4\,log_2(99)}$ Reihenglieder, $R_n < 2^{-t}$ gilt.

1. **[Initialisiere]** Setze $k_0 = 99^4,\ k_1 = 1103,\ k_2 = 26390,\ c = log_2(k_0)$.
2. **[Berechne die Anzahl der Schritte]** Setze $n = \frac{t}{c},\ m = 4\,n - 1$.
3. **[Berechnung]** Setze $term = p = k_1 + n\,k_2$. Solange $n > 0$ tue
   (a) $term = \frac{term}{k_0} \cdot \frac{\frac{m}{2}\,(m-2)\,m}{(\frac{m}{2}+1)\,(m+1)^2}$
   (b) $p = p - k_2,\ term = term + p$
   (c) $m = m - 4,\ n = n - 1$
4. **[Letzter Term]** Setze $term = \frac{term}{99^2},\ \pi = \frac{1}{\sqrt{8} \cdot term}$
5. **[Rückgabe]** Liefere π zurück.

Beweis der Korrektheit

Wir beweisen nun, daß für $n, t \in \mathbb{N}_{>0}$ gilt: $n > \frac{t}{4\,log_2(99)} \Longrightarrow R_n < 2^{-t}$. Der Beweis besteht aus 3 Teilen. Zunächst untersuchen wir Eigenschaften der Reihenglieder. Danach bestimmen wir das Restglied der Reihe (2) und schließlich beweisen wir durch Einsetzen des von uns gewählten n die Richtigkeit der Behauptung.

1. Wir definieren die Funktion $f:\ \mathbb{N} \to \mathbb{Q}$

$$f(m) := \frac{(4\,m)!\ (1103 + 26390\,m)}{(m!)^4\,396^{4\,m}}$$

und betrachten den Quotienten $g: \mathbb{Q} \to \mathbb{Q}$

$$
\begin{aligned}
g(m) := \frac{f(m)}{f(m+1)} &= \frac{(4\,m)!\,(1103 + 26390\,m)((m+1)!)^4 \cdot 396^{4\,m+4}}{(m!)^4 \cdot 396^{4\,m} \cdot (4\,m+4)!\,(27493 + 26390\,m)} \\
&= \frac{(4\,m)!\,(1103 + 26390\,m)(m!)^4(m+1)^4 \cdot 396^{4\,m} \cdot 396^4}{(m!)^4 \cdot 396^{4\,m} \cdot (4\,m)! \cdot (4\,m+1)\ldots(4\,m+4)\,(27493 + 26390\,m)} \\
&= 396^4 \cdot \frac{(1103 + 26390\,m)\,(m+1)^4}{(27493 + 26390\,m)(4\,m+1)\ldots(4\,m+4)}
\end{aligned}
$$

Da

$$\frac{m+1}{4\,m+1}, \ldots, \frac{m+1}{4\,m+4} \geq \frac{1}{4} \quad \textit{und} \quad \frac{1103 + 26390\,m}{27493 + 26390\,m} > \frac{1}{25}$$

folgt, daß

$$\frac{f(m)}{f(m+1)} > 1$$

$f(m)$ ist also monoton fallend. Da

$$\frac{g(m)}{g(m+1)} = \frac{(m+1)^3\,(1103 + 26390\,m)\,(4\,m+7)\,(2\,m+3)\,(4\,m+5)\,(53883 + 26390\,m)}{(4\,m+3)\,(2\,m+1)\,(4\,m+1)\,(27493 + 26390\,m)^2\,(m+2)^3},$$

sieht man leicht durch Ausmultiplizieren, daß $\frac{g(m)}{g(m+1)} > 1$, d.h. $g(m)$ ist monoton fallend. Weiter gilt wegen

$$\lim_{m\to\infty} \frac{1103 + 26390\,m}{27493 + 26390\,m} = 1, \quad \lim_{m\to\infty} \left(\frac{m+1}{4\,m+1} \cdot \ldots \cdot \frac{m+1}{4\,m+4} \right) = \frac{1}{256},$$

daß

$$\lim_{m\to\infty} g(m) = \frac{396^4}{256} = 99^4$$

Aus $g(0) > 99^4$ und $g(m)$ monoton fallend, können wir herleiten, daß $f(m+1) < \frac{f(m)}{99^4}$ (3).

2. Wir können nur die Restgliedabschätzung durchführen

$$
\begin{aligned}
R_n &= \frac{\sqrt{8}}{9801} \cdot \sum_{m=n+1}^{\infty} f(m) \\
&< \frac{\sqrt{8}}{9801} \cdot \left(\frac{f(0)}{99^{4\,(n+1)}} + \frac{f(0)}{99^{4\,(n+2)}} + \ldots \right) \\
&= \frac{\sqrt{8}}{9801} \cdot \frac{f(0)}{99^{4\,(n+1)}} \left(1 + \frac{1}{99^4} + \left(\frac{1}{99^4} \right)^2 + \ldots \right) \quad (4)
\end{aligned}
$$

Da $c_1 = 99^{-4} < 1$ ist die Summe der geometrischen Reihe $1 + c_1 + c_1^2 + \ldots = \frac{1}{1-c_1} \approx 1.000000011$. Wegen $f(0) = 1103$ folgt dann durch Ausmultiplizieren in (4)

$$
\begin{aligned}
R_n &< 0.3183098819 \frac{1}{99^{4\,(n+1)}} \\
&< \frac{1}{99^{4\,(n+1)}}
\end{aligned}
$$

3. Aus $n = \dfrac{t}{4\,log_2(99)}$ folgt

$$\begin{aligned} n+1 &> \frac{t}{4\,log_2(99)} \\ \Longleftrightarrow 4\,(n+1)\,log_2(99) &> t \\ \Longleftrightarrow 99^{4\,(n+1)} &> 2^t \\ \Longleftrightarrow R_n < \frac{1}{99^{4\,(n+1)}} &< 2^{-t} \end{aligned}$$

Aus (3) folgt, daß Ramanujan's Reihe mit fast 8 Dezimalstellen pro Term konvergiert.

4.3.2 Die Eulersche Konstante γ

$$\gamma = \lim_{n\to\infty}(H_n - \log(n)) = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \log(n)\right) = .5772156649015328... \text{ nach [6]}$$

Die direkte Berechnung von γ (Auswertung der Reihe, Logarithmus) erweist sich als sehr langwierig. Der Grund dafür ist die langsame Konvergenzgeschwindigkeit von H_n. Daher wurden andere Methoden entwickelt (siehe [4], [18]). Wir benutzen einen Algorithmus, der von R.P Brent 1980 präsentiert wurde:

Seien die modifizierte Besselfunktionen $I_n(z)$, $K_0(z)$ definiert durch

$$I_n(z) = \sum_{k=0}^{\infty}\frac{\left(\frac{z}{2}\right)^{n+2\,k}}{k!\,\Gamma(n+k+1)}, \quad K_0(z) = -\frac{\partial I_n(z)}{\partial n}|_{n=0},$$

wobei $\Gamma(z) = \int_0^\infty t^{z-1}\,e^{-t}\,dt$, $z > 0$. Durch Differentiation folgt

$$\gamma + log\left(\frac{z}{2}\right) = \frac{S_0(z) - K_0(z)}{I_0 z},$$

wobei $S_0(z) = \sum_{k=0}^{\infty}\frac{\left(\frac{z}{2}\right)^{2k}}{(k!)^2}H_k$. Durch Betrachtung der oberen und unteren Grenzen der asymptotischen Darstellungen von $I_0(z)$ und $K_0(z)$ folgt aus $z = 2\,n \geq 1$ [19]

$$0 < \frac{U(n)}{V(n)} - \gamma = \frac{K_0(2n)}{I_0(2n)} < \pi\,e^{-4n},$$

wobei $U(n) = S_0(2n) - I_0(2n)\,log(n) = \sum_{k=0}^{\infty}\left(\frac{n^k}{k!}\right)^2(H_k - log(n))$ und $V(n) = I_0(2n) = \sum_{k=0}^{\infty}\left(\frac{n^k}{k!}\right)^2$ ist. Wir müssen nun n so wählen, daß $\pi\,e^{-4n} < 2^{-t}$. Nach Auswertung der Ungleichung folgt $n > \frac{t}{4}\cdot log(2) + \frac{log(\pi)}{4} \approx \frac{t}{4}\cdot log(2) + 0.286$.

1. **[Wähle n]** Setze $n = 1 + \frac{t}{4} \cdot log(2)$
2. **[Initialisiere]** Setze $A_0 = -log(n)$, $B_0 = 1$, $U_0 = A_0$, $V_0 = 1$.
3. **[Bereche U_n, V_n]** Für $k = 1 \ldots (1 + 3.591 \cdot n)$ [1] tue
$$B_k = B_{k-1}\frac{n^2}{k^2}, \quad A_k = \frac{\left(A_{k-1}\frac{n^2}{k^2} + B_k\right)}{k}$$
$$U_k = U_{k-1} + A_k, \quad V_k = V_{k-1} + B_k$$
4. **[Bereche γ]** Setze $\gamma = \frac{U_k}{V_k}$.
5. **[Rückgabe]** Liefere γ zurück.

Beweis der Korrektheit

Siehe [4], Seite 305-308.

4.3.3 Die Zahl e

Die Zahl e wird mit Hilfe des Kettenbruchs
$$e = 1 + \cfrac{1}{1 - \cfrac{1}{3 - \cfrac{2}{4 - \cfrac{3}{5 \ldots}}}}$$
berechnet. Dieser Kettenbruch ist eine Umformung [14] der Entwicklung der Exponentialreihe für $x = 1$
$$e^1 = \sum_{n=0}^{\infty} \frac{1^n}{n!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$
und hat somit die gleiche Konvergenzgeschwindigkeit.

1. **[Berechne die Anzahl n der Schritte]** Setze $l = 0$, $n = 2$. Solange $l < t$ tue

   $l = l + log_2(n)$, $n = n + 1$.[2]

2. **[Initialisiere die Schleife]** Setze $num = n$, $den = 0$, $tmp = 1$. [3]

3. **[Schleife]** Tue

   (a) $den = num$

[1] Der Beweis dafür steht in [6], Seite 307.
[2] $log_2(n!) = log_2(1) + log_2(2) + ... + log_2(n)$.
[3] Wir berechnen den Kettenbruch durch 'Einrichten von hinten her': wir konstruieren also bei jedem Schritt einen neuen Bruch.

(b) $num = num \cdot (n-1)$

(c) $tmp = -tmp \cdot (n-2)$

(d) $num = num + tmp$

(e) $tmp = den$

(f) $n = n - 1$

solange $n > 3$.

4. **[Letzter Term]** Setze $den = num - den$, $num = num + den$.

5. **[Berechne** e**]** Setze $e = \dfrac{num}{den}$ (floating-point Division).

6. **[Rückgabe]** Liefere e zurück.

Beweis der Korrektheit

Für das Restglied der Exponentialfunktion für $x = 1$ gilt [10]

$$r_n \leq \frac{2}{n!}$$

Wir müssen also ein $n_0 \in \mathbb{N}$ finden, so daß für $n \geq n_0$ gilt:

$$\begin{aligned} \frac{2}{n!} &< 2^{-t} \\ \Longleftrightarrow \frac{1}{n!} &< 2^{-t-1} \\ \Longleftrightarrow n! &> 2^{t+1} \\ \Longleftrightarrow log(n!) &> (t+1)\, log(2) \end{aligned}$$

Um die Existenz eines solchen n_0 zu zeigen, reicht es zu beweisen, daß $\lim_{n\to\infty} log(n!) = \infty$. Dann existiert nämlich zu jedem $c \in \mathbb{N}$ ein $n_1 \in \mathbb{N}$, so daß

$$n \geq n_1 \Longleftrightarrow log(n!) > c$$

Der Behauptung folgt aus der strengen Monotonie der Logarithmus- und der Fakultätfunktion, und aus $\lim_{n\to\infty} n! = \infty$. Die Existenz von n_0 garantiert, daß die Berechnung im Schritt 1 endlich ist. Wähle $n_0 = n_1$.

4.4 Transzendente Funktionen

4.4.1 sqrt

Dieser Algorithmus berechnet y=sqrt(x) für eine gegebene Float Zahl x mit Hilfe der Newton'schen Iterationsvorschrift

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right),$$

wobei y_0 ein geeigneter Startwert ist [17]. Der Algorithmus setzt $x \geq 0$ voraus.

Zuerst untersuchen wir die Trivialfälle $x = 0$ und $x = 1$. Falls $x \neq 0$ und $x \neq 1$ berechnen wir eine 32-bit Approximation (Startwert) der Wurzel von x mittels

$$sqrt(x) \approx sqrt(first\_32bit(x)) \cdot 2^{\frac{e_x}{2}}$$

Eine kleine Modifikation ist notwendig falls der Exponent von x (e_x) ungerade ist. Da die Newton'sche Iteration quadratisch konvergiert, verdoppelt sich die Anzahl der richtig berechneten Stellen von $sqrt(x)$ in jedem Schritt. Es genügen also $1 + \frac{log\left(\frac{t}{32}\right)}{log(2)}$ Schritte [4] um $sqrt(x)$ mit einer Genauigkeit von t Stellen zu berechnen, da unsere Approximation auf mindestens 31 Bit richtig ist.

1. **[Trivialfälle]** Falls $x == 0$ (bzw. $x == 1$), setze $y == 0$ (bzw. $y == 1$) und gehe zu Schritt 5.
2. **[Startwert (SPA)]** Sei $odd = e_x\%2$ $(= sign(e_x) \cdot |e_x|\%2)$.
   (a) Setze $e_y = \frac{e_x}{2}$.
   (b) Falls $odd < 0$, setze $odd = odd + 2$, $e_y = e_y - 1$.
   (c) Setze $y = sqrt((odd + 1) \cdot first\_32bit(m_x))$.
   (d) Setze $e_y = e_y + e_x$
3. **[Berechne die Anzahl n der Schritte]** Setze $n = 1 + \frac{log\left(\frac{t}{32}\right)}{log(2)}$.
4. **[Newton Iteration]** Berechne $y = \frac{1}{2}\left(y + \frac{x}{y}\right)$ n-mal.
5. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Siehe [4], Seite 244, Lemma 2.3. □

4.4.2 exp

Dieser Algorithmus berechnet y=exp(x) für eine gegebene Float Zahl x mittels

$$exp(x) = \begin{cases} 1 & x = 0 \\ \dfrac{1}{exp(-x)} & \infty < x < 0 \\ exp\left(\dfrac{x}{\lambda}\right)^{\lambda} & 2^{-c} \leq x < \infty,\ \lambda = 2^q,\quad c, q \in \mathbb{N}_{>1} \\ \displaystyle\sum_{i=0}^{\infty} \frac{x^i}{i!} & 0 < x < 2^{-c} \end{cases}$$

Wir untersuchen zuerst den Trivialfall $x = 0$. Falls $x < 0$ ist nutzen wir die zweite Identität um ein positives x zu erhalten. Um die Auswertung der Reihe zu beschleunigen, dividieren wir x durch eine geeignete Potenz von 2. Das Ergebnis erhalten wir dann durch Quadrieren.

1. [**Test auf 0**] Falls $x == 0$, setze $y = 1$ und gehe zu Schritt 8.
2. [**Ist $x < 0$?**] Falls $x < 0$, setze $sign = 1$ und $x = |x|$.
3. [**Berechne die Anzahl n der Schritte**] Setze $c = \sqrt{\frac{t}{2}}$. Falls $log_2(x) > -c$, setze $n = 1 + \frac{t}{c}$, $f = 1 + log_2(x) + c$. Sonst setze $n = 1 + \frac{t}{|log_2(x)|}$, $f = 0$.
4. [**Reduziere x**] Setze $x = \frac{x}{2^f}$.
5. [**Berechnung**] Setze $y = 1$ und $i = n$. Solange $i \geq 0$ tue
   (a) $y = 1 + \frac{y}{i} \cdot x$.
   (b) $i = i - 1$
6. [**Rekonstruiere**] Solange $f \neq 0$ tue $y = y^2$, $f = f - 1$.
7. [**Teste sign**] Falls $sign == 1$, setze $y = \frac{1}{y}$.
8. [**Rückgabe**] Liefere y zurück.

Beweis der Korrektheit

Für das Restglied der Exponentialfunktion gilt folgende Abschätzung [10]:

$$R_n \leq 2\,\frac{x^n}{n!}$$

Wir beweisen folgende Lemmata:

4.1. Lemma *Sei* $t \in \mathbb{N}_{>0}$, $x \in \mathbb{R}_{>0}$, $x < 2^{-c}$, *wobei* $c \in \mathbb{N}_{>1}$. *Dann gilt:*

$$n \in \mathbb{N}_{>1},\ n \geq \frac{t}{c} \Longleftrightarrow R_n < 2^{-t}$$

Beweis:

$$
\begin{aligned}
2\,\frac{x^n}{n!} &\le x^n,\ da\ n > 1\\
&< 2^{-c\,n},\ da\ x < 2^{-c}\\
&< 2^{-c\,\frac{t}{c}},\ da\ n \ge \frac{t}{c}\\
&< 2^{-t}
\end{aligned}
$$

■

4.2. Lemma *Sei* $t \in \mathbb{N}_{>0}$*,* $x < 2^{-\sqrt{\frac{t}{2}}}$*. Dann gilt:*

$$n \in \mathbb{N}_{>1},\ n \ge \sqrt{2\,t} \Longleftrightarrow R_n < 2^{-t}$$

Beweis: Das Lemma ist ein Spezialfall des obigen Lemmas mit $c = \sqrt{\frac{t}{2}}$. ■

4.4.3 log

Dieser Algorithmus berechnet y=log(x) für eine gegebene Float Zahl x mit Hilfe der folgenden Identitäten:

$$
log(x) = \begin{cases}
-log\left(\frac{1}{x}\right) & 0 < x < 1\\
2\,log(\sqrt{x}) & 1 < x < \infty\\
2\sum_{i=0}^{\infty} \frac{y^{2i+1}}{2i+1},\ y = \frac{x-1}{x+1} & 1 < x < 1 + 2^{-c},\ \ c \in \mathbb{N}_{>1}
\end{cases}
$$

1. [**Teste x**]
   (a) Falls $x \le 0$, gib eine Fehlermeldung aus und terminiere.
   (b) Falls $x == 1$, setze $y = 0$ und gehe zu Schritt 9.
   (c) Falls $x < 1$, setze $sign = 1$ und $x = \frac{1}{x}$.
2. [**Reduktion 1**] Setze $f_1 = 0$. Solange $x \ge 2$ tue $f_1 = f_1 + 1$, $x = \sqrt{x}$.
3. [**Berechne die Anzahl n der benötigten Schritte**] Setze $c = \frac{32}{10}\sqrt{\frac{t}{32 \cdot 3}}$. Weiter setze $lt = \frac{t}{2 \cdot 32}\,log(2)$. Falls $log_2(x-1) \ge -c$, setze $n = 1 + \frac{t}{c}$, $f = 1 + log_2(x-1) + c$. Sonst setze $n = 1 + \frac{lt}{a}$, $f = 0$.
4. [**Reduktion 2**] Für i gleich 1 bis f, tue $x = \sqrt{x}$.
5. [**Initialisierung**] Setze $n = 2\,n + 1$, $tmp = \frac{x-1}{x+1}$, $x = \left(\frac{x-1}{x+1}\right)^2$ und $y = \frac{1}{n}$.
6. [**Schleife**] Setze $n = n - 2$. Solange $n \ge 1$ tue $y = y \cdot x$, $y = y + \frac{1}{n}$, $n = n - 2$.

7. **[Rekonstruiere]** Setze $y = 2^{1+f+f_1} \cdot y \cdot tmp$.
8. **[Teste** *sign***]** Falls $sign == 1$, setze $y = -y$.
9. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Es genügt zu beweisen, daß für $1 < x \leq 1 + 2^{-c+1}$ die Anzahl der benötigten Schritte, um log mit einem relativen Fehler kleiner 2^{-t} zu berechnen, gleich $\frac{t}{2\,c}$ ist. Wir beweisen dazu ein Lemma und einen Satz.

4.3. Lemma *Sei $c > 0$, $1 < x < 1 + 2^{-c+1}$. Dann gilt*

$$\frac{x-1}{x+1} < 2^{-c}$$

Beweis: Aus der Voraussetzung folgt:

$$\begin{aligned} \frac{x-1}{x+1} &< \frac{2^{-c+1}}{2} \\ &= 2^{-c} \end{aligned}$$

■

4.1. Satz *Sei f(x)=log(x) berechnet nach der Formel*

$$log(x) = 2\sum_{i=0}^{\infty} \frac{y^{2\,i+1}}{2\,i+1}, \quad wobei\ y = \frac{x-1}{x+1}.$$

Für gegebene t, $c \in \mathbb{N}_{>0}$ und $1 < x < 1 + 2^{-c+1}$ gilt dann:

$$f\ddot{u}r\ n \in \mathbb{N}\ \ mit\ \ n \geq \frac{t}{2\,c} \quad \Longrightarrow \quad \left| f(x) - 2\sum_{i=0}^{n} \frac{y^{2\,i+1}}{2\,i+1} \right| < 2^{-t}$$

Beweis:

$$\begin{aligned} \left| f(x) - 2\sum_{i=0}^{n} \frac{y^{2\,i+1}}{2\,i+1} \right| &= \left| 2\sum_{i=0}^{\infty} \frac{y^{2\,i+1}}{2\,i+1} - 2\sum_{i=0}^{n} \frac{y^{2\,i+1}}{2\,i+1} \right| \\ &= 2\left| \sum_{i=n+1}^{\infty} \frac{y^{2\,i+1}}{2\,i+1} \right| \\ &= 2\left| \frac{x^{2\,n+3}}{2\,n+3} + \frac{x^{2\,n+5}}{2\,n+5} + \frac{x^{2\,n+7}}{2\,n+7} + \ldots \right| \end{aligned}$$

Substituiere $z = 2\,n + 3$:

$$\begin{aligned} 2\left| \frac{y^z}{z} + \frac{y^{z+2}}{z+2} + \frac{y^{z+4}}{z+4} + \ldots \right| &\leq 2\left| \frac{y^z}{z} + \frac{y^{z+2}}{z} + \frac{y^{z+4}}{z} + \ldots \right| \\ &= 2\,\frac{|y|^z}{z} \left| 1 + y^2 + y^4 + \ldots \right| \quad (1) \end{aligned}$$

Da $|y| < 1$, ist die Summe der geometrischen Reihe $1 + y^2 + y^4 + \ldots$ gleich $\frac{1}{1-y^2}$. Daraus folgt:

$$(1) \quad = \quad 2\,\frac{|y|^z}{z}\,\frac{1}{1-y^2} \quad (2)$$

Wir schätzen nun y durch 2^{-c} ab (siehe Lemma 4.3) und führen die Rücksubstitution $z = 2\,n + 3$ aus:

$$\begin{aligned}(2) \quad &< \quad 2\,\frac{2^{-c\,z}}{z}\,\frac{1}{1-2^{-2\,c}} \\ &= \quad 2\,\frac{2^{-2\,c\,n-3\,c}}{2\,n+3}\,\frac{1}{1-2^{-2\,c}} \quad (3)\end{aligned}$$

Wir benutzen nun die Tatsache, daß $n \geq \frac{t}{2\,c}$ ist:

$$\begin{aligned}(3) \quad &\leq \quad 2\,\frac{2^{-2\,c\,\frac{t}{2\,c}-3\,c}}{2\,\frac{t}{2\,c}+3}\,\frac{1}{1-2^{-2\,c}} \\ &= \quad 2^{-t}\,\frac{2^{-3\,c}}{\left(\frac{t}{2\,c}+\frac{3}{2}\right)(1-2^{-2\,c})} \quad (4)\end{aligned}$$

Da $\left(\frac{t}{2\,c}+\frac{3}{2}\right) > 1$, folgt weiter

$$(4) \quad < \quad 2^{-t}\,\frac{2^{-3\,c}}{1-2^{-2\,c}}$$

Es bleibt zu beweisen, daß $\frac{2^{-3\,c}}{1-2^{-2\,c}} \leq 1$ ist. Dazu schreiben wir

$$\begin{aligned}\frac{2^{-3\,c}}{1-2^{-2\,c}} \quad &= \quad \frac{1}{2^{3\,c}\,(1-2^{-2\,c})} \\ &= \quad \frac{1}{2^{c}\,(2^{2\,c}-1)} \\ &< \quad 1, \; da\; c > 0\end{aligned}$$

Damit ist Satz 4.1 bewiesen. ■

4.4.4 Die Besselfunktion J_n

Für eine Float Zahl x und eine ganze Zahl n berechnet dieser Algorithmus y=J_n(x) nach [1]. Wir benutzen dabei

$$J_n(x) = \begin{cases} (-1)^n J_{-n}(x) & n < 0 \\ \dfrac{x^n}{2^n} \displaystyle\sum_{i=0}^{\infty} \frac{\left(-\frac{x^2}{4}\right)^i}{i!\,(n+i)!} & n \geq 0 \end{cases}$$

1. **[Teste n]** Falls $n < 0$, setze $n = -n$. Falls $n\%2 == 1$, setze $sign = 1$.
2. **[Berechne den Multiplikator]** Setze $tmp_1 = \dfrac{x^n}{2^n \cdot n!}$
3. **[Initialisiere]** Setze $y = tmp_2 = 1$.
4. **[Schleife]** Setze $i = 1$. Solange $log_2(tmp_2) > -(t + log_2(tmp_1))$, tue

$$tmp_2 = tmp_2 \cdot \frac{\frac{-x^2}{4}}{i(n+i)},\ y = y + tmp_2,\ i = i + 1$$

5. **[Multipliziere mit tmp_1]** Setze $y = y \cdot tmp_1$
6. **[Teste sign]** Falls $sign == 1$, setze $y = -y$.

Beweis der Korrektheit

Wir benutzen die Leibnizsche Regel, um das Restglied der Reihe zu bestimmen.

4.4. Lemma *(Leibnizsche Regel) Falls a_i eine monoton fallende Nullfolge ist, dann ist die alternierende Reihe $\sum_{i=1}^{\infty}(-1)^i\, a_i$ konvergent. Sei S die Summe der Reihe und s_i die Teilsummen i-ter Ordnung, so gilt weiter*

$$0 < (-1)^i\,(S - s_i) < a_{i+1}\ \forall n \geq 1$$

Beweis: Siehe [2]. ■

Dieses Lemma besagt, daß der Fehler nach der Aufsummierung von i Gliedern einer solchen Reihe, kleiner als der Wert des (i+1)-ten Gliedes ist (im Algorithmus testen wir ob der Wert des zuletzt aufsummierten Reihenglieds kleiner als die erwünschte Präzision ist). In unserem Fall müssen wir also beweisen, daß:

$$\begin{aligned}
a_i = \frac{x^n}{2^n} \cdot \frac{\left(\frac{x^2}{4}\right)^i}{i!\,\Gamma(n+i+1)} &\longrightarrow 0\ f\ddot{u}r\ i \longrightarrow \infty \\
\Longleftrightarrow \frac{x^n}{2^n} \cdot \frac{1}{4^i} \cdot \frac{(x^2)^i}{i!\,(n+i)!} &\longrightarrow 0\ f\ddot{u}r\ i \longrightarrow \infty \\
\Longleftrightarrow \frac{x^n}{2^n} \cdot \frac{1}{4^i} \cdot \frac{(x^2)^i}{i!} \cdot \frac{1}{(n+i)!} &\longrightarrow 0\ f\ddot{u}r\ i \longrightarrow \infty
\end{aligned}$$

Setze $C = \frac{x^n}{2^n}$, $a_i = \frac{1}{4^i}$, $b_i = \frac{(x^2)^i}{i!}$ und $c_i = \frac{1}{(n+i)!}$. Es ist klar, daß $C\,a_i\,c_i \longrightarrow 0$ für $i \longrightarrow \infty$. Wir müssen noch beweisen, daß $b_i \longrightarrow 0$. Dazu schreibe:

$$b_i = \prod_{j<i} \frac{x^2}{j},\ i > x^2$$

$$\begin{aligned}
&= \prod_{j=1}^{k} \frac{x^2}{j} \prod_{j=k+1}^{i} \frac{x^2}{j},\ mit\ k \leq x^2 < k+1 \\
&= C_1 \prod_{j=k+1}^{i} \frac{x^2}{j} \\
&\longrightarrow 0,\ da\ x^2 < j,\ \forall j \geq k+1
\end{aligned}$$

Da $J_{2n+1}(-x) = -J_{2n+1}(x)$ und $J_{2n}(-x) = J_{2n}(x)$, $n \in \mathbb{N}$ [1], können wir o.B.d.A. annehmen, daß $x > 0$ und somit $a_i > 0$. Mit Hilfe des Quotientenkriteriums [2] folgt trivialerweise, daß a_i monoton fallend ist (da $\frac{a_i}{a_{i+1}} > 1$). Somit sind die Voraussetzungen der Leibnizschen Regel erfüllt und es gilt

$$S - s_i = R_i < a_{i+1}.$$

Es existiert also ein n_0, so daß

$$n \geq n_0\ gilt\ a_n < 2^{-t}$$

Damit haben wir bewiesen, daß die Rechnung im Schritt 4 endlich ist. Die Korrektheit des Resultats folgt nun aus der Bedingung der Schleife: falls $log_2(tmp_2) < -(t + log_2(tmp_1))$, liefern die nachfolgenden Terme keine Änderung im t-stelligen Resultat. □

4.5 Trigonometrische Funktionen

4.5.1 cos

Für eine Float Zahl x, berechnet dieser Algorithmus y=cos(x) mittels

$$cos(x) = \begin{cases} cos(-x) & x < 0 \\ -cos(x + k\,\pi) & x > \pi,\ k \in \mathbb{Z},\ ungerade \\ cos(x + k\,\pi) & x > \pi,\ k \in \mathbb{Z},\ gerade \\ \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!} & x < 2^{-c} \\ 2\,cos^2\left(\frac{x}{2}\right) - 1 & x \in \mathbb{R} \end{cases}$$

1. **[Test auf 0]** Falls $x == 0$, setze $y = 1$ und gehe zu Schritt 7. Sonst setze $x = |x|$.

2. **[Ist $x > \pi$?]** Falls $x > \pi$, setze $k = trunc\left(\frac{x}{\pi}\right)$, $x = x - k\,\pi$. Falls k ungerade, setze $sign = 1$.

3. **[Reduziere]** Setze $c = \sqrt{\frac{t}{4}}$. Falls $log_2(x) > -c$, setze $n = 1 + \frac{t}{c}$, $f = 1 + log_2(x) + c$. Sonst setze $n = 1 + \frac{t}{|log_2(x)|}$, $f = 0$. Setze weiter $x = \frac{x}{2^f}$. Falls n ungerade tue $n = n + 1$.

4. **[Berechnung]** Setze $x = -x^2$, $y = 1$. Solange $n > 0$ tue
   (a) $y = \dfrac{y}{n \cdot (n-1)} \cdot x + 1$
   (b) $n = n - 2$
5. **[Rekonstruktion]** Solange $f > 0$, tue $y = 2 \cdot y^2 - 1$.
6. **[Teste sign]** Falls $sign == 1$, setze $y = -y$.
7. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Für das Restglied der Cosinusfunktion gilt folgende Abschätzung [10]:

$$\begin{aligned} R_{2n+2} &\leq \frac{x^{2n+2}}{(2n+2)!} \; f\ddot{u}r \; |x| \leq 2n+3 \\ \Longleftrightarrow R_N &\leq \frac{x^N}{N!} \; f\ddot{u}r \; |x| \leq N+1, \; N = 2n+2 \end{aligned}$$

Wir beweisen folgende Lemmata:

4.5. Lemma *Sei* $t \in \mathbb{N}_{>0}$, $x \in \mathbb{R}_{>0}$, $x < 2^{-c}$, *wobei* $c \in \mathbb{N}_{>1}$. *Dann gilt:*

$$n \in \mathbb{N}_{>1}, \; n \geq \frac{t}{c} \Longleftrightarrow R_n < 2^{-t}$$

Beweis:

$$\begin{aligned} \frac{x^n}{n!} &\leq x^n, \; da \; n > 1 \\ &< 2^{-cn}, \; da \; x < 2^{-c} \\ &< 2^{-c\frac{t}{c}}, \; da \; n \geq \frac{t}{c} \\ &< 2^{-t} \end{aligned}$$

■

4.6. Lemma *Sei* $t \in \mathbb{N}_{>0}$, $x < 2^{-\sqrt{\frac{t}{4}}}$. *Dann gilt:*

$$n \in \mathbb{N}_{>1}, \; n \geq \sqrt{4t} \Longleftrightarrow R_n < 2^{-t}$$

Beweis: Das Lemma ist ein Spezialfall des obigen Lemmas mit $c = \sqrt{\frac{t}{4}}$. ■

4.5.2 sin

Für eine Float Zahl x, berechnet dieser Algorithmus y=sin(x) mittels

$$sin(x) = -cos\left(x + \frac{\pi}{2}\right)$$

1. **[Trivialfall]** Falls $x == 0$, setze $y = 0$ und gehe zu Schritt 3.
2. **[Berechnung]** Setze $y = -cos\left(x + \frac{\pi}{2}\right)$.
3. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit folgt aus der obigen Identität und der Korrektheit der Berechnung der Cosinusfunktion.

4.5.3 tan — cot

Für eine Float Zahl x berechnet dieser Algorithmus y=tan(x) (bzw. cot(x)) mittels,

$$(1)\ \ tan(x) = \begin{cases} -tan(-x) & x < 0 \\ tan(x + k\,\pi) & x \in \mathbb{R},\ k \in \mathbb{Z} \\ \dfrac{\sqrt{1 - cos^2(x)}}{cos(x)} & 0 \le |x| \le \pi \end{cases}$$

$$(2)\ \ cot(x) = \begin{cases} -cot(-x) & x < 0 \\ cot(x + k\,\pi) & x \in \mathbb{R},\ k \in \mathbb{Z} \\ \dfrac{cos(x)}{\sqrt{1 - cos^2(x)}} & 0 < |x| < \pi \end{cases}$$

$$\begin{aligned} (3)\ \ sin^2(x) + cos^2(x) &= 1 \\ \Longleftrightarrow sin^2(x) &= 1 - cos^2(x) \\ \Longleftrightarrow sin(x) &= \sqrt{1 - cos^2(x)} \end{aligned}$$

1. **[x< 0?]** Falls $x < 0$, setze $x = |x|$ und $sign = 1$. Sonst setze $sign = 0$
2. **[Ist $x > \pi$?]** Falls $x > \pi$, setze $k = trunc\left(\frac{x}{\pi}\right)$, $x = x - k\,\pi$.

3a.**[Berechnung von tan]** Setze $y = \dfrac{\sqrt{1 - cos^2(x)}}{cos(x)}$

3b.**[Berechnung von cot]** Setze $y = \dfrac{cos(x)}{\sqrt{1 - cos^2(x)}}$

4. **[Teste sign]** Falls $sign == 1$, setze $y = -y$.

5. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des obigen Algorithmus folgt aus der Korrektheit der Formeln (1), (2) und der Korrektheit der Berechnung der Cosinusfunktion. □

4.6 Hyperbolische trigonometrische Funktionen

4.6.1 sinh — cosh — tanh — coth

Für eine Float Zahl x berechnet dieser Algorithmus y=sinh(x), cosh(x), tanh(x), coth(x) mittels

$$\begin{array}{llll} (1) & sinh(x) = \dfrac{e^x - e^{-x}}{2} & (2) & cosh(x) = \dfrac{e^x + e^{-x}}{2} \\ (3) & tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}} & (4) & coth(x) = \dfrac{e^x + e^{-x}}{e^x - e^{-x}} \end{array}$$

1. **[Berechnung]** Setze $y = \begin{cases} (1) & für\ sinh \\ (2) & für\ cosh \\ (3) & für\ tanh \\ (4) & für\ coth \end{cases}$

2. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des obigen Algorithmus folgt aus der Korrektheit der Formeln (1)-(4) und der Korrektheit der Berechnung der Exponentialfunktion. □

4.7 Inverse trigonometrische Funktionen

4.7.1 arctan

Für eine Float Zahl x berechnet dieser Algorithmus y=arctan(x) mittels

$$arctan(x) = \begin{cases} \pm\frac{\pi}{4} & x = \pm 1 \\ -\arctan(-x) & x < 0 \\ \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) & x > 1 \\ 2\arctan\frac{x}{1+\sqrt{x^2+1}} & x > 2^{-c},\ c \in \mathbb{N}_{>0} \\ \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{2i+1} & x \leq 2^{-c} \end{cases}$$

1. **[Trivialfall 1]** Falls $x == 0$, setze $y = 0$ und gehe zu Schritt 10.
2. **[Trivialfall 2]** Falls $x == \pm 1$, setze $y = \pm\frac{\pi}{4}$ und gehe zu Schritt 10.
3. **[Initialisiere]** Setze $sign = sign(x)$, $x = |x|$, $f = 0$
4. **[$|x| >$1?]** Falls $x > 1$, setze $x = \frac{1}{x}$ und $inv = 1$.
5. **[Reduziere]** Setze $c = 10$. Solange $log_2(x) > -c$, tue $x = \frac{x}{1+\sqrt{x^2+1}}$; $f = f + 1$.
6. **[Berechne die Anzahl n der Schritte]** Setze $n = \frac{t}{2\,c}$. Falls n ungerade, setze $n = n + 1$.
7. **[Summiere mit Hornerschema]** Setze $y = \frac{1}{2\,n+1}$. Solange $n \geq 1$ tue
   (a) $y = \frac{1}{2\,n-1} - y \cdot x^2$
   (b) $n = n - 1$
8. **[Letzter Term]** Setze $y = y \cdot x \cdot 2^f$.
9. **[Teste inv, $sign$]** Falls $inv == 1$, setze $y = \frac{\pi}{2} - y$. Falls $sign == -1$, setze $y = -y$.
10. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Es genügt zu beweisen, daß für $|x| < 2^{-c}$ die Anzahl der benötigten Schritte, um $arctan$ mit einem relativen Fehler kleiner 2^{-t} zu berechnen, gleich $\frac{t}{2\,c}$ ist. Wir beweisen dazu folgenden Satz.

4.2. Satz *Sei f(x)=arctan(x). Für gegebene* $t,\, c \in \mathbb{N}_{>0}$ *und* $0 < x < 2^{-c}$ *gilt dann:*

$$\textit{für } n \in \mathbb{N} \quad \textit{mit} \quad n \geq \frac{t}{2\,c} \quad \Longrightarrow \quad \left| f(x) - \sum_{i=0}^{n} \frac{y^{2\,i+1}}{2\,i+1} \right| < 2^{-t}$$

Beweis:

$$\begin{aligned}
\left| f(x) - \sum_{i=0}^{n} \frac{x^{2\,i+1}}{2\,i+1} \right| &= \left| \sum_{i=0}^{\infty} \frac{x^{2\,i+1}}{2\,i+1} - \sum_{i=0}^{n} \frac{x^{2\,i+1}}{2\,i+1} \right| \\
&= \left| \sum_{i=n+1}^{\infty} \frac{x^{2\,i+1}}{2\,i+1} \right| \\
&= \left| \frac{x^{2\,n+3}}{2\,n+3} + \frac{x^{2\,n+5}}{2\,n+5} + \frac{x^{2\,n+7}}{2\,n+7} + \ldots \right|
\end{aligned}$$

Substituiere $y = 2\,n + 3$:

$$\begin{aligned}
\left| \frac{x^y}{y} + \frac{x^{y+2}}{y+2} + \frac{x^{y+4}}{y+4} \ldots \right| &\leq \left| \frac{x^y}{y} + \frac{x^{y+2}}{y} + \frac{x^{y+4}}{y} + \ldots \right| \\
&= \frac{x^y}{y} \left| 1 + x^2 + x^4 + \ldots \right| \quad (1)
\end{aligned}$$

Die Summe $1 + x^2 + x^4 + \ldots$ ist gleich der Summe der geometrischen Reihe

$$\sum_{i=0}^{\infty} z^i, \quad \textit{wobei } z = x^2 < 1,$$

also gleich $\dfrac{1}{1-z} = \dfrac{1}{1-x^2}$. Daraus folgt:

$$(1) \quad = \quad \frac{x^y}{y} \, \frac{1}{1-x^2} \quad (2)$$

Wir schätzen nun x durch 2^{-c} ab und führen die Rücksubstitution $y = 2\,n + 3$ aus:

$$\begin{aligned}
(2) \quad &< \quad \frac{2^{-c\,y}}{y} \, \frac{1}{1 - 2^{-2\,c}} \\
&= \quad \frac{2^{-2\,c\,n - 3\,c}}{2\,n+3} \, \frac{1}{1 - 2^{-2\,c}} \quad (3)
\end{aligned}$$

Wir benutzen nun die Tatsache, daß $n \geq \dfrac{t}{2\,c}$ ist:

$$\begin{aligned}
(3) \quad &\leq \quad \frac{2^{-2\,c \frac{t}{2\,c} - 3\,c}}{2\,\frac{t}{2\,c} + 3} \, \frac{1}{1 - 2^{-2\,c}} \\
&= \quad 2^{-t} \, \frac{2^{-3\,c}}{\left(\frac{t}{c} + 3 \right) 1 - 2^{-2\,c}} \quad (4)
\end{aligned}$$

Da $\left(\frac{t}{c}+3\right) > 1$, folgt weiter

$$(4) \quad < \quad 2^{-t}\,\frac{2^{-3\,c}}{1-2^{-2\,c}}$$

Es bleibt nur zu beweisen, daß $\frac{2^{-3\,c}}{1-2^{-2\,c}} \leq 1$ ist. Dazu schreiben wir

$$\begin{aligned} \frac{2^{-3\,c}}{1-2^{-2\,c}} &= \frac{1}{2^{3\,c}\,(1-2^{-2\,c})} \\ &= \frac{1}{2^{c}\,(2^{2\,c}-1)} \\ &< 1,\; da\; c > 0 \end{aligned}$$

■

Im obigen Algorithmus haben wir $c = 10$ gewählt.

4.7.2 arcsin

Für eine Float Zahl x$\in [-1,1]$ berechnet dieser Algorithmus y=arcsin(x) mittels

$$arcsin(x) = \begin{cases} \pm\frac{\pi}{2} & x = \pm 1 \\ arctan\left(\frac{x}{\sqrt{1-x^2}}\right) & x \in]-1,1[\end{cases}$$

1. **[Trivialfall]** Falls $|x| == 1$, setze $y =$sign(x)$\frac{\pi}{2}$ und gehe zu Schritt 3.
2. **[Berechnung]** Setze $y = \arctan\frac{x}{\sqrt{1-x^2}}$.
3. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des Algorithmus folgt aus der Korrektheit der Berechnung der Arcustangensfunktion. □

4.7.3 arccos

Für eine Float Zahl x$\in [-1,1]$ berechnet dieser Algorithmus y=arccos(x) mittels

$$arccos(x) = \begin{cases} 0 & x = 1 \\ \pi & x = -1 \\ \frac{\pi}{2} - arctan\left(\frac{x}{\sqrt{1-x^2}}\right) & x \in]-1,1[\end{cases}$$

1. **[Trivialfall 1]** Falls $x == 1$, setze $y = 0$ und gehe zu Schritt 4.
2. **[Trivialfall 2]** Falls $x == -1$, setze $y = \pi$ und gehe zu Schritt 4.
3. **[Berechnung]** Setze $y = \frac{\pi}{2} - \arctan \frac{x}{\sqrt{1-x^2}}$.
4. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des Algorithmus folgt aus der Korrektheit der Berechnung der Arcustangensfunktion. □

4.7.4 arccot

Für eine Float Zahl x berechnet dieser Algorithmus y=arccot(x) mittels

$$arccot(x) = \begin{cases} \frac{\pi}{4} & x = 1 \\ -\frac{3\pi}{4} & x = -1 \\ \frac{\pi}{2} - arctan(x) & x \in]-1, 1[\end{cases}$$

1. **[Trivialfall 1]** Falls $x == 1$ setze $y = \frac{\pi}{4}$ und gehe zu Schritt 3.
2. **[Trivialfall 2]** Falls $x == -1$ setze $y = \frac{3\,\pi}{4}$ und gehe zu Schritt 3.
3. **[Berechnung]** Setze $y = \frac{\pi}{2} - \arctan x$.
4. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des Algorithmus folgt aus der Korrektheit der Berechnung der Arcustangensfunktion. □

4.8 Inverse hyperbolische trigonometrische Funktionen

4.8.1 arsinh — arcosh — artanh — arcoth

Für eine Float Zahl x berechnet dieser Algorithmus y=arsinh(x), arcosh(x), artanh(x), arcoth(x) mittels

$$\begin{aligned}
(1)\ arsinh(x) &= \log\left(x+\sqrt{x^2+1}\right), \quad x \in \mathbb{R} \\
(2)\ arcosh(x) &= \log\left(x+\sqrt{x^2-1}\right), \quad x > 1 \\
(3)\ artanh(x) &= \frac{1}{2}\log\frac{1+x}{1-x}, \quad x \in]-1,1[\\
(4)\ arcoth(x) &= \frac{1}{2}\log\frac{x+1}{x-1}, \quad x \in]-\infty,-1[\ \cup\ ]1,\infty[
\end{aligned}$$

1. **[Berechnung]** Setze $y = \begin{cases} (1) & f\ddot{u}r\ arsinh \\ (2) & f\ddot{u}r\ arcosh \\ (3) & f\ddot{u}r\ artanh \\ (4) & f\ddot{u}r\ arcoth \end{cases}$

2. **[Rückgabe]** Liefere y zurück.

Beweis der Korrektheit

Die Korrektheit des Algorithmus folgt aus der Korrektheit der Formeln (1)-(4) und der Korrektheit Berechnung der Logarithmusfunktion. □

Kapitel 5

bigfloat — libC — libP

Neben der C-Implementierung enthält libF auch eine C++—Implementierung der mathematischen Operationen. Diese Alternative wird angeboten, um das Arbeiten mit langen Zahlen zu vereinfachen. Ich möchte hier die Vorteile von C++ nicht auflisten, verweise aber den Leser auf [13]. Der Aufbau des C++ Teils von libF folgt vielen der Vorschläge, die dort auftreten.

5.1 Die Klasse bigfloat

Der C++—Typ, der lange Gleitkommazahlen darstellt, heißt **bigfloat**. Bigfloat ist eine Klasse, die in etwa folgendes Aussehen hat:

```
class           bigfloat {
         private:
                 Float F;
         public:
                 Konstruktoren;
                 Destruktor;
                 Grundoperationen (+, -, * , /, ^);
                 Transzendente Funktionen (sqrt, sin, pi(), ...);
                 Eingabe/Ausgabe;
};
```

Der entscheidende Vorteil dieses Typs ist, daß er wie ein Machinentyp benutzt werden kann (siehe Anhang B). Alle C++—Funktionen rufen C Funktionen auf, wodurch der unangenehme Effekt einer riesigen Bibliothek ausbleibt, außerdem ist die Geschwindigkeit gleich der der C Version. Eine vollständige Beschreibung des Include-Files steht in Anhang C.

5.2 Anwendung: Ein interaktiver Taschenrechner

libF verfügt über einen interaktiven Taschenrechner mit dem Namen **thales**. thales verfügt über alle Bibliotheksfunktionen und stellt sogar geschachtelte **for**, **while** und **if** Anweisungen zur Verfügung. Weiter verfügt er über einen Stack, in dem bis zu 256 bigfloat's gespeichert werden können.

Syntax

Die von thales akzeptierte Syntax lautet:

$< LZahl > \ldots\ldots long(derMaschinentyp)$
$< GanzeZahl > \ldots\ldots \epsilon | < GanzeZahl > 0 | ... | < GanzeZahl > 9$
$< Zahl > \ldots\ldots < GanzeZahl > . < GanzeZahl >$
$| < GanzeZahl > . < GanzeZahl > e < LZahl >$
$| < GanzeZahl > . < GanzeZahl > E < LZahl >$
$< Variablennummer > \ldots\ldots 0|1|2|...|255$
$< Variable > \ldots\ldots v < Variablennummer > |V < Variablennummer >$
$< Zuweisung > \ldots\ldots \epsilon | < Variable > = < Zuweisung > | < Variable > = < Ausdruck >$
$| < Ausdruck >$
$< Ausdruck > \ldots\ldots < EinfacherAusdruck > + < EinfacherAusdruck >$
$| < EinfacherAusdruck > - < EinfacherAusdruck >$
$< EinfacherAusdruck > \ldots\ldots < Term > * < Term > | < Term > / < Term >$
$< Term > \ldots\ldots < EinfacherFaktor > \hat{} < EinfacherFaktor >$
$< EinfacherFaktor > \ldots\ldots < Funktion >$
$< Funktion > \ldots\ldots < Zahl > | < Zuweisung > | < EinfacherFaktor >$
$| < EinfacheFunktion >$
$< EinfacheFunktion > \ldots\ldots for(< LZahl >, < LZahl >, < LZahl >, < Anweisung >)|$
$if(< Bedingung >:< Anweisung1 >:< Anweisung2 >)|$
$abs(< Zahl >)|sqrt((< Zahl >)|...|q$
$< Anweisung > \ldots\ldots < Zuweisung >)$

Beim Aufruf von **thales** erscheint

```
 -----         --          --  --
   |    |__| |__| |     |__ |__
   |    |  | |  | |__  |       |
   |    |               |__     |
   |      Version 1.0        __|
      by Papanikolaou Thomas
Algebraic Software - Copyright 1992

        Type '?' for help


>
```

Jetzt kann der Benutzer Ausdrücke mit beliebiger Genauigkeit berechnen lassen. Hier sind einige Beispiele

```
> precision = 50
```

thales meldet sich mit

```
 precision = 50 significant digits
```

```
> sqrt(2)
%   .14142135623730950488016887242096980785696718753769e1

> pi
%   .31415926535897932384626433832795028841971693993751e1

> quit

Goodbye!
```

5.3 Die Klasse Complex

Dieser C++—Typ stellt lange komplexe Zahlen in der Form (a , b) dar. a, b sind bigfloat's. Es ist nicht schwer zu erraten, daß die Klasse **Complex** wie folgt aussieht:

```
class           Complex {
         private:
                  bigfloat re, im;
         public:
                  Konstruktoren;
                  Destruktor;
                  Grundoperationen (+, -, * , /, ^);
                  Transzendente Funktionen (sqrt, sin, arg, ...);
                  Eingabe/Ausgabe;
};
```

Diese Klasse behält natürlich alle bigfloat Funktionen: es darf ohne Einschränkung zwischen bigfloat's und Complex'es gerechnet werden. Die genaue Definition dieser Klasse befindet sich im Anhang C.

5.4 Die Klasse Polynom

Eine weitere Stufe in C++ ist die **Polynom** Klasse. Sie enthält Polynome mit komplexen (und daher auch mit reellen und ganzzahligen) Koeffizienten.

```
class           Polynom {
         protected:
```

```
                Complex        *coeff;
                int             degree;
        public:
                Konstruktoren;
                Destruktor;
                Grundoperationen (+, -, * , /);
                Eingabe/Ausgabe;
};
```

Diese Klasse enthält natürlich alle bigfloat und Complex Funktionen. Weiter können in dieser Klasse Polynomableitungen und GGT berechnet werden.

5.5 Anwendung: Nullstellenbestimmung von Polynomen

Im folgenden wird ein Verfahren zur Bestimmung der Nullstellen eines Polynoms mit komplexen Koeffizienten präsentiert. Der Algorithmus stammt aus [8], wurde aber so modifiziert, daß er sogar für **nicht-quadratfreie**[1] Polynome (z.B $(x-1)^8$) funktioniert.

Der Algorithmus *Complex Roots*

Die Prozedur, in der dieser Algorithmus implementiert wurde, heißt **Roots**; sie erhält als Eingabe ein Polynom P mit komplexen Koeffizienten und liefert die Nullstellen des Polynoms zurück. Die Arbeitsweise kann leicht durch das folgende Schema erklärt werden:

1. Falls P quadratfrei, benutze den Algorithmus *Cohen* und terminiere.
2. Berechne eine Nullstelle a, dividiere P mit (x-a). Tue das, bis alle Nullstellen berechnet sind.

Ausführlicher:

Eingabe	: ein Polynom P
Rückgabe	: alle Nullstellen des Polynoms P

1. **[Teste die Koeffizienten]** Falls P nur reelle Koeffizienten hat setze $f = 0$; Sonst setze $f = 1$;
2. **[Teste P]** Falls $ggT(P, P') == 1$, rufe Cohen(P, f) und gehe zu Schritt 4. Sonst setze $n = Grad(P)$.
3. **[P ist nicht quadratfrei]** Setze $SQRF = \frac{P}{ggT(P,P')}$. Solange $n > 0$ tue
   (a) $i = Grad(SQRF)$.
   (b) Cohen(SQRF, f);

[1] Ein Polynom heißt **quadratfrei** falls $ggT(P, P') == 1$ ist; P' bezeichnet die erste Ableitung von P.

(c) n = n - i;
(d) Falls $n > 0$, setze $P = \frac{P}{SQRF}$, $SQRF = \frac{P}{gcd(P,P')}$

4. **[Rückgabe]** Liefere die gefundenen Nullstellen zurück (z.B als Vektor).

Der Algorithmus *Cohen*

Der folgende Algorithmus stammt aus [8]. Er liefert die Nullstellen eines quadratfreien Polynomes P zurück. Die Variable f stammt aus dem Algorithmus **Roots**.

Eingabe	: ein quadratfreies Polynom P und eine Zahl f (siehe *Roots*)
Rückgabe	: alle Nullstellen des Polynoms P

1. **[Initialisierung]** Setze $Q = P$, $R = 1$ und $n = Grad(Q)$.
2. **[Schleife]** Solange $n > 0$ tue
   (a) **[Startwert]** Setze $x = (1.3, 0.3141592)$
   (b) **[Finde eine Nullstelle]** Tue $x = Root(Q, x)$, $R = R \cdot (X - x)$, $n = n - 1$ und gebe x aus.
   (c) **[Teste f]** Falls $f == 1$ und $imag(x) == 0$ [2] gebe $conj(x)$ aus und setze $R = R \cdot (X - conj(x))$, $n = n - 1$. [3]
   (d) **[Abdividieren]** Setze $Q = P/R$.
3. **[Rückgabe]** Liefere die Nullstellen zurück.

Der Algorithmus *Root*

Der Algorithmus *Cohen* hat noch einen nicht besprochenen Prozeduraufruf: die Funktion *Root*. Diese Funktion bestimmt eine Nullstelle des Polynoms Q mit Hilfe des Newton'schen Verfahrens.

$$x_{n+1} = x_n - \frac{Q(x_n)}{Q'(x_n)}$$

Eingabe	: ein Polynom Q und ein ein Startwert x
Rückgabe	: eine Nullstelle des Polynoms Q

1. **[Initialisierung]** Setze $u = Q(x)$, $m = norm(u)$, $dx = \frac{u}{Q'(x)}$.
2. **[Schleife]** Solange $|dx| > 2^{-t}$ tue
   (a) **[Initialisierung]** Setze $c = 0$, $dx = \frac{u}{Q'(x)}$. Falls $|dx| < 2^{-t}$ gehe zu Schritt (c).
   (b) **[Schleife]** Tue

[2] *imag* bezeichnet den imaginären Teil von x.
[3] *conj* bezeichnet die konjugierte Zahl von x.

i. $y = x - dx$, $u_1 = Q(y)$, $m_1 = norm(u_1)$

ii. Falls $m_1 < m$, verlasse die Schleife. Sonst setze $c = c + 1$, $dx = \frac{dx}{4}$. Falls $c > 20$ konnte die Prozedur keine Nullstelle finden. In diesem Fall muß der Benutzer die Präzision erhöhen oder einen neuen Startwert initialisieren.

solange $m_1 >= m$

(c) Setze $x = y$, $u = u_1$, $m = m_1$.

3. [**Rückgabe**] Liefere x zurück.

Bemerkungen zu der Korrektheit dieses Algorithmus stehen in [8], Seite 82-84.

Kapitel 6

Anhang A — Laufzeittests

Die aufgelisteten Laufzeiten in diesem Kapitel sollen die Rolle eines Wegweisers für denjenigen spielen, der mit libF effizient programmieren möchte. Wir benutzen hierfür die Unterscheidung zwischen Floats, die nicht reduziert werden können (z.B die $1.1 = (1.0001100110011...)_2$) und Floats, die im Binärsystem exakt darstellbar und damit reduzierbar sind (z.B alle ganze Zahlen oder rationale Zahlen mit einer zweier Potenz im Nenner). Eine Zahl der ersten Kategorie nennen wir **real**, eine Zahl der zweiten Kategorie **sreal**.

6.1 Die Benchmarks

Um die Geschwindigkeit der Grundoperationen zu überprüfen, werden for-Schleifen mit einer variierenden Präzision bei konstanter Durchlaufanzahl benutzt. Die besten Benchmarks einer solchen Bibliothek sind natürlich ihre eigenen transzendenten Funktionen. Es werden nur die Funktionen *sqrt*, *exp*, *log*, *cos* und *atan* getestet, da diese die Grundlage aller anderen Funktionen sind.

Alle Tests wurden auf einer ELC Sun Sparcstation mit 21.0 mips durchgeführt.

6.2 Ergebnisse

6.2.1 Grundoperationen

Jede Funktion wurde 1000 mal ausgeführt. Die durchschnittliche Prozessorzeit wurde in Millisekunden gemessen. Die Operationen wurden bei Präzision 100, 500, 1000 und 5000 Dezimalstellen getestet. In der folgenden Tabelle steht in der ersten Spalte die allgemeine Beschreibung der Operation und in der zweiten die tatsächlich ausgeführte Anweisung.

x, y, z sind vom Typ *Float*, s vom Typ *long*. Die Argumente e, π, und $\sqrt{3}/2$ sind vorberechnet.

Operation	Argumente	100	500	1000	5000
z = x + y	z = e + π	0.05	0.16	0.28	1.32
z += y	z + = π	0.07	0.21	0.35	1.65
z = x + s	z = e + 1	0.04	0.09	0.14	0.63
z += s	z + = 1	0.06	0.14	0.21	0.9
z + +	z + +	0.05	0.14	0.21	0.9
$\|z\|$	$\|z\|$	0	0	0	0
z = x − y	z = e − π	0.04	0.1	0.19	0.84
z − = y	z − = π	0.08	0.21	0.33	1.54
z = x− s	z = e − 1	0.04	0.1	0.16	0.71
z − = s	z − = 1	0.06	0.14	0.21	0.9
z−−	z − −	0.06	0.13	0.21	0.91
− z	− z	0	0	0	0
z = x ∗ y	z = $\sqrt{3}$ / 2 ∗ π	0.4	3.5	10.7	138.4
z ∗ = y	z ∗ = π	0.3	3.6	10.9	138.3
z = x ∗ s	z = e ∗ 2	0.05	0.14	0.27	1.28
z ∗ = s	z ∗ = 2	0.05	0.14	0.27	1.28
z = x / y	z = e / π	1	10.1	33.3	647.9
z / = y	z / = e	1.1	10.1	33.4	657.3
z = x / s	z = e / 2	0.1	0.4	0.7	3.5
z / = s	z / = 2	0.1	0.3	0.7	3.1
z = s / y	z = 1 / π	0.7	6.5	22.6	510.5
z = s / s	z = 1 / 3	0.1	0.4	0.7	3.2

6.2.2 Transzendente Funktionen

Wir testen nun die Funktionen *sqrt*, *exp*, *log*, *cos* und *atan*. Jede Funktion wurde diesmal nur einmal ausgeführt. Die Prozessorzeit wurde in Millisekunden gemessen. Die Operationen wurden bei Präzision 100, 200, ... und 4000 Dezimalstellen getestet. Um die Aussage der Laufzeiten optisch darzustellen, werden folgende Tabellen durch zwei Kurvenbilder ergänzt: auf der x-Achse steht die Präzision und auf der y-Achse die benötigte Zeit in Millisekunden.

sreal-Argument

Präzision	sqrt(2)	exp(2)	log(2)	cos(2)	atan(2)
100	0	10	20	20	50
200	0	30	70	70	120
300	10	50	170	130	270
400	10	80	270	200	410
500	10	120	360	310	590
600	10	170	490	440	870
700	20	210	850	590	1270
800	30	260	1030	790	1620
900	30	340	1260	970	2160

Fortsetzung der Tabelle

Präzision	sqrt(2)	exp(2)	log(2)	cos(2)	atan(2)
1000	30	400	1480	1210	2650
1100	40	480	1710	1430	3190
1200	30	540	1990	1710	3570
1300	70	630	3220	1970	4980
1400	70	750	3330	2180	6140
1500	70	800	3910	2850	6670
1600	80	920	4310	2780	8150
1700	90	1100	4890	3400	9210
1800	80	1140	5100	3790	10200
1900	90	1320	6040	4180	11580
2000	90	1510	6170	4650	13670
2100	100	1580	7120	5640	14600
2200	110	1780	7230	5450	16810
2300	110	1930	8280	6230	17940
2400	110	1930	8180	5920	19490
2500	210	2300	12780	7230	22690
2600	220	2370	12920	7400	24070
2700	210	2470	13600	7630	25620
2800	210	2920	14520	9130	29040
2900	220	2930	15660	9340	30880
3000	240	2980	16100	9580	32690
3100	240	3560	17840	11130	37330
3200	240	3640	18610	11410	39200
3300	260	3710	19610	11720	41310
3400	270	4310	20570	13500	46270
3500	280	4400	22000	13890	48550
3600	290	4360	22480	14020	50890
3700	290	5050	23650	16190	56830
3800	300	5160	25200	16500	59420
3900	320	5190	25910	17250	62200
4000	330	5930	28020	19440	69340

real-**Argument**

Präzision	sqrt	exp	log	cos	atan
100	0	10	20	20	50
200	0	40	50	70	130
300	10	90	130	130	290
400	10	150	200	190	440
500	20	220	300	300	620
600	20	330	420	430	910
700	40	440	710	550	1330
800	50	550	890	780	1700
900	40	760	1080	930	2220
1000	60	910	1230	1120	2700
1100	60	1080	1490	1340	3260
1200	70	1180	1710	1690	3740
1300	120	1480	2760	1810	5010
1400	120	1830	2980	2230	5980
1500	130	1910	3330	2670	6570
1600	130	2270	3910	2770	7880
1700	150	2740	4220	3230	9150
1800	150	2820	4640	3780	9900
1900	170	3340	5290	3980	11520
2000	180	3860	5690	4590	13240
2100	180	3970	6270	5270	14370
2200	190	4460	6660	5620	16290
2300	200	4700	7300	5720	17600
2400	210	5020	7660	5880	18930
2500	350	5820	11750	7190	23240
2600	360	6090	11800	7540	24510
2700	350	6360	12480	7580	25990
2800	360	7330	13040	9000	29640
2900	380	7680	14240	9330	31530
3000	400	7860	14790	9520	33270
3100	420	9210	16360	11010	37670
3200	440	9580	17240	11350	39930
3300	460	9630	18340	11840	41870
3400	470	11000	19270	13390	47260
3500	480	11420	20020	13850	49650
3600	500	11700	21190	13930	52110
3700	520	13210	21920	16090	57770
3800	540	13470	23580	16400	60670
3900	550	13970	23990	16740	63220
4000	580	15710	26170	19160	70220

Bild 1 — sreal

Bild 2 — real

6.2.3 Zusammenfassung

Sei T(x) die Realzeit die benötigt wird, um eine Operation x in libF auszuführen, so gelten folgende Ungleichungen:

$$\begin{gathered} T(add) \leq T(sub) < T(mul) < T(div) \\ T(\pi) \leq T(e) < T(\gamma) \\ T(sqrt) < T(exp) < T(log) \\ T(cos) \leq T(sin) < T(tan) \leq T(cot) \\ T(arctan) \leq T(arccot) < T(arccos) \leq T(arcsin) \\ T(cosh) \leq T(sinh) < T(tanh) \leq T(coth) \\ T(artanh) \leq T(arcoth) < T(arcosh) \leq T(arsinh) \end{gathered}$$

6.3 Vergleiche mit anderen Systemen

Die führende Computer Algebra Systeme von heute sind MAPLE[1], Mathematica[2] und PARI[3]. Wir benutzen für diesen Test die Sparc Versionen von MapleV Release 2, Mathematica 2.1 und PARI Version 1.37 compiliert mit der -O4 Option.

In diesem Test beschränken wir uns auf die transzendenten Funktionen. Die Ergebnisse sind repräsentativ für die Effizienzunterschiede zwischen den 4 Systemen: wir wollen sie aber nicht als absolut bezeichnen, da jedes System (möglicherweise) auf andere Zahlenbereiche spezialisiert bzw. optimiert ist. Bisher hat es sich aber gezeigt, daß das Verhältnis der Laufzeiten konstant bleibt.

Testkonfiguration

1. Präzision = 1000 Dezimalstellen.
2. Jede Funktion wurde 1000mal ausgeführt. Die durchschnittliche Prozessorzeit wurde in Millisekunden gemessen.
3. Die Argumente e, π, $\pi/3$, $\pi/4$, $log(2)$ und $\sqrt{3}/2$ sind vorberechnet.
4. x, y, z sind vom Typ *Float*, s vom Typ *long*.
5. z wurde am Anfang des Tests mit 0 initialisiert.

Im folgenden steht *Mathe* für Mathematica, "-" für "nicht vorhanden", "nm" für nicht meßbar und *LookUp* für "als Konstante abgespeichert".

[1]MAPLE is ein Warenzeichen von Waterloo Maple Software.
[2]Mathematica ist ein Warenzeichen von Wolfram Research Inc.
[3]PARI: Copyright 1989,1992 by C. Batut, D. Bernardi, H. Cohen and M. Olivier.

Operation	Argumente	libF	Pari	Maple	Mathe
z = π	π	160	170	LookUp	650
z = γ	γ	9520	7840	72466	22316
z = e	e	170	970	667	183
z = sqrt(y)	z = $\sqrt{3}$	27	50	467	166
z = exp(y)	z = exp(log(2))	910	1020	2333	6066
z = log(y)	z = log(exp(2))	1532	1540	1300	4700
z = Jn(x)	z = jn(10, 2)	3160	-	70217	10100
z = pow(x, y)	z = pow(e, π)	2430	2550	3883	10950
z = pow(x, s)	z = pow(π, 10)	55	70	483	150
z = sin(y)	z = sin(π/3)	945	1370	6083	6783
z = cos(y)	z = cos(π/3)	945	1250	6000	6133
z = tan(y)	z = tan(π/4)	1036	1270	6000	6900
z = cot(y)	z = cot(π/4)	1036	-	6000	6900
z = asin(y)	z = asin($\sqrt{3}/2$)	2850	3460	8767	24200
z = acos(y)	z = acos($\sqrt{3}/2$)	2840	3440	8616	23533
z = atan(y)	z = atan($\sqrt{3}/2$)	2700	3310	7817	27683
z = acot(y)	z = acot($\sqrt{3}/2$)	2700	-	8250	27783
z = sinh(y)	z = sinh(log(2))	940	1040	2683	2633
z = cosh(y)	z = cosh(log(2))	930	1050	2683	2650
z = tanh(y)	z = tanh(log(2))	950	1080	3900	2750
z = coth(y)	z = coth(log(2))	950	-	4333	2800
z = asinh(y)	z = asinh(π)	1590	1620	4584	8950
z = acosh(y)	z = acosh(π)	1590	1610	4083	4950
z = atanh(y)	z = atanh(1/π)	1510	1530	3650	7833
z = acoth(y)	z = acoth(π)	1520	-	9433	4600
z = rndtoi(y)	z = rndtoi(π)	nm	nm	nm	nm
z = ceil(y)	z = ceil(π)	nm	nm	nm	nm
z = floor(y)	z = floor(π)	nm	nm	nm	nm
z = trunc(y)	z = trunc(π)	nm	nm	nm	nm

Aus dieser Tabelle geht hervor, daß libF (mit Ausnahme der γ Konstante) bei allen anderen Funktionen schneller ist. Im Vergleich zur Pari bis zu 20%, gegen Maple bis 85% und gegen Mathematica bis 90%.

Kapitel 7

Anhang B - Ein Beispiel

7.1 Implementierung von cos(x) in C

Wir werden den im Kapitel 4 präsentierten Algorithmus benutzen.

7.1.1 Die Funktion

```
void
cosinus(y, x)
        register pFloat y;
        const Float    *x;
{
        Float           tmp;
        long            ex, f = 0, s = 0, t = BinPrec;
        long            i, k, n;

        if (x->type != 1) /* Falls x keine reelle Zahl ist ... */
                Ferror("Fcos needs a real argument!");

        /* Schritt 1 */
        if (Fis0(x)) { /* Falls x gleich 0 ist, bei Praezision t ... */
                Fasdbl(y, 1.0);
                return;
        }
        cFasF(&tmp, x); /* tmp = x */
        p0.m.sign = 0; /* tmp = |x| */


        /* Schritt 2 */
        Fpi(y); /* Berechne pi */
        if (FgeF(&tmp, &F_pi)) { /* Falls tmp > pi tue */
                FasFdiF(y, &tmp, &F_pi); /* y = tmp / pi */
```

```
        Ftrunc(y, y); /* y = trunc(tmp / pi) */
        i = y->m.vec[0] & 1; /* ist y ungerade? */
        FmuasF(y, &F_pi); /* y = trunc(tmp / pi) * pi */
        FmiasF(&tmp, y); /* tmp = tmp - i * pi */
        if (i & 1) /* Falls i ungerade ... */
                s = 1;
}

/* Schritt 3 */
ex = tmp.e + Flength(&tmp); /* Vorkommastellen von tmp */
k = -sqrt(t / 4.0); /* Konvergenzbereich (KB) */
if ( ex > k ) {
        n = 1 + sqrt(4.0 * t); /* Anzahl der Schritte */
        f = 1 + ex - k; /* Anzahl der Reduktionsschritte */
        tmp.e -= f; /* tmp = tmp / 2^n */
} else
        n = 1 + (t * 1.0) / abs(ex); /* Falls x schon im KB */
if (n & 1) /* Falls n ungerade ... */
        n++;

/* Schritt 4 */
FasF(y, &tmp); /* y = tmp */
FmuasF(&tmp, y); /* tmp = tmp^2 */
tmp.m.sign = 1; /* tmp = -tmp^2 */
Fasdbl(y, 1.0); /* y = 1 */
for (i = n; i > 0; i -= 2) {
        Fdiasint(y, i * (i - 1)); /* y = y / (i*(i-1)) */
        FmuasF(y, &tmp); /* y = y * tmp */
        Fplasint(y, 1); /* y = y + 1 */
}

/* Schritt 5 */
if (f) { /* Falls Rekonsruktion noetig ... */
        FasF(&tmp, y); /* tmp = y */
        for (i = 0; i < f; i++) {
                FmuasF(y, &tmp); /* y = y^2 */
                y->e++; /* y = 2 * y^2 */
                FasFplint(&tmp, y, -1); /* y = 2 * y^2 - 1 */
                FasF(y, &tmp); /* y = tmp */
        }
}

/* Schritt 6 */
if (s)
        y->m.sign = !y->m.sign;

dF(&tmp);
```

```
        /* Schritt 7 */
}
```

7.2 Implementierung von cos(x) in C++

Wir werden nun die gleiche Funktion mit Hilfe des C++ Typs bigfloat implementieren. Die Kommentare wurden diesmal weggelassen, da der Programmcode in C++ selbsterklärend ist.

```
bigfloat
cosinus(bigfloat x)
{
        bigfloat        y, tmp = abs(x);
        long            ex, f = 0, s = 0, t = BinPrec;
        long            i, k, n;

        if (type(x) != 1)
                Ferror("Fcos needs a real argument!");
        if (is0(x)) {
                y = 1.0;
                return y;
        }
        if ( tmp > pi() ) {
                y = trunc(tmp / pi);
                i = isodd(y);
                tmp -= y * pi();
                if (i & 1)
                        s = 1;
        }
        ex = expo(tmp) + log2(tmp) + 1;
        k = -sqrt(t / 4.0);
        if ( ex > k ) {
                n = 1 + sqrt(4.0 * t);
                f = 1 + ex - k;
                tmp >>= f;
        } else
                n = 1 + (t * 1.0) / abs(ex);

        if (n & 1)
                n++;

        tmp = -tmp^2;
        y = 1.0;
        for (i = n; i > 0; i -= 2) {
                y /= i * (i - 1);
                y *= tmp;
                y++;
        }
```

```
        if (f)
                for (i = 0; i < f; i++)
                        y = 2 * y^2 - 1;
        if (s)
                y = -y;

        return y;
}
```

Kapitel 8

Anhang C - Manual

Für die Beschreibung der Funktionen benutzen wir folgendes Format

NAME
Der(die) Name(n) der Funktion(en) .... kurze Beschreibung

ÜBERSICHT
Deklaration, Form des Aufrufs mit Parameter- und Rückgabetyp

BESCHREIBUNG
Ausführliche Beschreibung der Funktion(en)

BEMERKUNG
Bemerkungen und Hinweise für die Praxis

SIEHE AUCH
Verweis auf verwandte Funktionen

Die einzige Ausnahme in Bezug auf diesem Format, bilden die Seiten für die include-files von libF (ffloat.h, bigfloat.h). Sie sind in diesem Manual beinhaltet um den Programmierer vor Fehlern bei der Parameterübergabe zu bewahren.

ADDITION

NAME

FasFplF, FplasF, FasFplint, Fplasint, Finc .... Addition

ÜBERSICHT

```
void FasFplF(x,y,z)
Float *x,*y,*z;

void FplasF(x,y)
Float *x,*y;

void FasFplint(x,y,z)
Float *x,*y;
int z;

void Fplasint(x,y)
Float *x;
int y;

void Finc(x)
Float *x;
```

BESCHREIBUNG

FasFplF() und FplasF() entsprechen den Operatoren + bzw. + = für Float Zahlen (d.h. $x = y + z$, $x + = y$). FasFplint() und Fplasint() entsprechen den Operatoren + bzw. + = für eine Float Zahl und einen int (d.h. $x = y + z$, $x + = y$). Finc() entspricht dem ++ Operator.

BEMERKUNG

FasFplF() darf nicht mit y == z aufgerufen werden.

SIEHE AUCH

Subtraktion, Multiplikation, Division, Shift

BIGFLOAT.H

```
/*
 * Copyright(C) 1992 Algebraic Software
 * written by Papanikolaou Thomas
 * (papanik@cs.uni-sb.de).
 */

#ifndef _BIGFLOAT_H
#define _BIGFLOAT_H

#include <iostream.h>
#include "ffloat.h"


class           bigfloat {
         private:

                 FloatF;

         public:

                 bigfloat();
                 bigfloat(Float f);
                 bigfloat(Integer I);
                 bigfloat(double d);
                 bigfloat(int i);
                 bigfloat(unsigned int i);
                 bigfloat(long i);
                 bigfloat(unsigned long i);
                 bigfloat(bigfloat & a);
                 ~bigfloat();

bigfloat & operator = (bigfloat & a);
double          operator = (double d);
int             operator = (int i);

friend BOOLEAN  isint(bigfloat & a);
friend BOOLEAN  isuint(bigfloat & a);
friend BOOLEAN  islong(bigfloat & a);
friend BOOLEAN  isulong(bigfloat & a);
friend BOOLEAN  isdbl(bigfloat & a);

friend bigfloat operator + (bigfloat & a, bigfloat & b);
friend bigfloat operator - (bigfloat & a, bigfloat & b);
friend bigfloat operator - (bigfloat & a);
friend bigfloat operator * (bigfloat & a, bigfloat & b);
```

```
friend bigfloat operator / (bigfloat & a, bigfloat & b);
friend bigfloat operator << (bigfloat & a, unsigned int u);
friend bigfloat operator >> (bigfloat & a, unsigned int u);
friend bigfloat operator ^ (bigfloat & a, bigfloat & b);
friend bigfloat operator ^ (bigfloat & a, int n);

bigfloat & operator += (bigfloat & a);
bigfloat & operator -= (bigfloat & a);
bigfloat & operator *= (bigfloat & a);
bigfloat & operator /= (bigfloat & a);
bigfloat & operator <<= (unsigned int u);
bigfloat & operator >>= (unsigned int u);
bigfloat & operator++ ();
bigfloat & operator-- ();

              BOOLEAN operator ! ();
inline friend BOOLEAN operator == (bigfloat & a, bigfloat & b);
inline friend BOOLEAN operator > (bigfloat & a, bigfloat & b);
inline friend BOOLEAN operator != (bigfloat & a, bigfloat & b);
inline friend BOOLEAN operator >= (bigfloat & a, bigfloat & b);
inline friend BOOLEAN operator < (bigfloat & a, bigfloat & b);
inline friend BOOLEAN operator <= (bigfloat & a, bigfloat & b);

friend int is0(bigfloat & a);
friend int sign(bigfloat & a);
friend int expo(bigfloat & a);
friend int type(bigfloat & a);
friend void settype(bigfloat & a, int type);
friend void setprec(long prec);
friend int log2(bigfloat & a);
friend int fscanF(FILE * f, bigfloat & a);
friend int fprintF(FILE * f, bigfloat & a);
friend int Ftoa(bigfloat & a, char s[]);
friend int atoF(char s[], bigfloat & a);

friend double   dbl(bigfloat & a);
friend bigfloat abs(bigfloat & a);
friend bigfloat neg(bigfloat & a);
friend bigfloat trunc(bigfloat & a);
friend bigfloat rndtoi(bigfloat & a);
friend bigfloat ceil(bigfloat & a);
friend bigfloat floor(bigfloat & a);
friend bigfloat sqrt(bigfloat & a);
friend bigfloat exp(bigfloat & a);
friend bigfloat log(bigfloat & a);
friend bigfloat jn(int n, bigfloat & a);
friend bigfloat euler();
friend bigfloat pi();
```

```
friend bigfloat e();
friend bigfloat sin(bigfloat & a);
friend bigfloat cos(bigfloat & a);
friend bigfloat tan(bigfloat & a);
friend bigfloat cot(bigfloat & a);
friend bigfloat sinh(bigfloat & a);
friend bigfloat cosh(bigfloat & a);
friend bigfloat tanh(bigfloat & a);
friend bigfloat coth(bigfloat & a);
friend bigfloat asin(bigfloat & a);
friend bigfloat acos(bigfloat & a);
friend bigfloat atan(bigfloat & a);
friend bigfloat acot(bigfloat & a);
friend bigfloat asinh(bigfloat & a);
friend bigfloat acosh(bigfloat & a);
friend bigfloat atanh(bigfloat & a);
friend bigfloat acoth(bigfloat & a);

};

ostream & operator << (ostream & out, bigfloat & a);
IN_FLOAT_BUF_LENGTH = 10000;
istream & operator >> (istream & in, bigfloat & a);

#endif
```

CPLUSPLUS

NAME

bigfloat .... C++ Typ

ÜBERSICHT

class bigfloat {...};

BESCHREIBUNG

bigfloat ist der C++ Typ, der auf Float aufgebaut ist. Dieser neue Typ ermöglicht eine viel einfachere Benutzung der Float Arithmetik: bigfloat kann in arithmetischen Ausdrücken wie eine double Zahl benutzt werden.

BEISPIEL

Um die obige Aussage zu verdeutlichen, betrachten wir die Berechnung der Quadratwurzel einer Zahl auf prec dezimale Stellen:

```
bigfloat
sqrt(bigfloat x, long prec)
{
        bigfloat        y = x / 2, y1;
        long            hp = DecPrec;

        setprec(prec);
        while(1) {
        y1 = y;
        y = (y + x / y ) / 2;
                i = (y1 == y);
        if (i)
                break;
        }
        setprec(hp);

        return y;
}
```

SIEHE AUCH

bigfloat.h

DIVISION

NAME

FasFdiF, FdiasF, FasFdiint, Fdiasint, FasintdiF, Fasintdiint .... Division

ÜBERSICHT

```
void FasFdiF(x,y,z)
Float *x,*y,*z;

void FdiasF(x,y)
Float *x,*y;

void FasFdiint(x,y,z)
Float *x,*y;
int z;

void Fdiasint(x,y)
Float *x;
int y;

void FasintdiF(x,y,z)
Float *x,*z;
int y;

void Fasintdiint(x,y,z)
Float *z;
int x,y;
```

BESCHREIBUNG

FasFdiF() und FdiasF() entsprechen den Operatoren / bzw. / = für Float Zahlen (d.h. x = y / z, x / = y). FasFdiint() und Fdiasint() entsprechen den Operatoren / bzw. / = für eine Float Zahl und ein int (d.h. x = y / z, x / = y). FasintdiF() ist die int/Float Division und Fasintdiint() die int/int Division mit langem Ergebnis.

BEMERKUNG

FasFdiF() darf nicht mit y == z aufgerufen werden.

SIEHE AUCH

Addition, Subtraktion, Multiplikation, Shift

EINGABE/AUSGABE

NAME

atoF, Ftoa, fscanF, fprintF, printF .... Standard und File Eingabe/Ausgabe

ÜBERSICHT

```
int atoF(s,x)
char s[];
Float *x;

int Ftoa(x,s)
Float *x;
char s[];

int fprintF(fp,x)
FILE *fp;
Float *x;

int fscanF(fp,x)
FILE *fp;
Float *x;

int printF(msg,x)
char msg[];
Float *x;
```

BESCHREIBUNG

atoF() wandelt eine Zeichenkette in einem Float um (Syntax: $\pm x.y$[eE]$\pm z$) und liefert die Länge der Zeichenkette zurück. Ftoa() schreibt eine Float Zahl als Zeichenkette der Form [-].xe[-]z und liefert die Länge der Zeichenkette zurück.

fscanF() und fprintF() sind die entsprechenden Funktionen für die File Ein-/Ausgabe. printF() schreibt die Zeichenkette msg und die Float Zahl x auf den Bildschirm.

BEMERKUNG

Bei der Zeichenkette s in atoF() sind keine Leerzeichen erlaubt.

SIEHE AUCH

Hilfsfunktionen(1)

FFLOAT.H

```
/*
 * Copyright(C) 1992 Algebraic Software
 * written by Papanikolaou Thomas
 * (papanik@cs.uni-sb.de)
 */

#ifndef _FFLOAT_H
#define _FFLOAT_H

#include "iint.h"
#include <math.h>

#define MAXDECPREC      300000
#define MAXBINPREC      1993157
#define MAXPREC         62287
#define L2B10           0.30102999566398119521
#define LOG2            0.69314718055994530942
#define MYPI            3.14159265358979323846
#define K0              9.63295986124739824672
#define F31             2147483648
#define F32             4294967296.0

extern long     BinPrec;
extern long     DecPrec;
extern long     Prec;
extern Float    F_e;
extern Float    F_pi;
extern Float    F_euler;

#ifndef C_VARIETIES_H
#ifdef __cplusplus
#define EXTERN_FUNCTION( rtn, args ) extern "C" { rtn args; }
#define _VOID_ /* anachronism */
#else
#ifdef __STDC__
#ifdef __PARC__
#define EXTERN_FUNCTION( rtn, args ) rtn()
#define _VOID_
#define const
#else
#define EXTERN_FUNCTION( rtn, args ) rtn args
#define _VOID_ void
#endif
#else
#define EXTERN_FUNCTION( rtn, args ) rtn()
```

```
#define _VOID_
#define const
#endif
#endif
#endif

typedef struct {
        Integer         m; /* mantisse */
        long            e; /* exponent */
        int             type; /* type */
}               Float, *pFloat;

EXTERN_FUNCTION(void cF, (Float *));
EXTERN_FUNCTION(void cFasF, (Float *, Float *));
EXTERN_FUNCTION(void cFasI, (Float *, Integer *));
EXTERN_FUNCTION(void cFasdbl, (Float *, double));
EXTERN_FUNCTION(void cFasint, (Float *, int));
EXTERN_FUNCTION(void cFasuint, (Float *, unsigned int));
EXTERN_FUNCTION(void cFaslong, (Float *, long));
EXTERN_FUNCTION(void cFasulong, (Float *, unsigned long));
EXTERN_FUNCTION(void cFmaxlength, (Float *, int));
EXTERN_FUNCTION(void dF, (Float *));
EXTERN_FUNCTION(Float * ncF, (void));
EXTERN_FUNCTION(Float * ncFasF, (Float *));
EXTERN_FUNCTION(Float * ncFasI, (Integer *));
EXTERN_FUNCTION(Float * ncFasdbl, (double));
EXTERN_FUNCTION(Float * ncFasint, (int));
EXTERN_FUNCTION(Float * ncFmaxlength, (int));
EXTERN_FUNCTION(void ddF, (Float *));

EXTERN_FUNCTION(void FasF, (Float *, Float *));
EXTERN_FUNCTION(void FasI, (Float *, Integer *));
EXTERN_FUNCTION(void Fasdbl, (Float *, double));
EXTERN_FUNCTION(void Fasint, (Float *, int));
EXTERN_FUNCTION(void Fas0, (Float *));
EXTERN_FUNCTION(void Fas1, (Float *));
EXTERN_FUNCTION(void IasF, (Integer *, Float *));
EXTERN_FUNCTION(void dblasF, (double *, Float *));
EXTERN_FUNCTION(BOOLEAN Fisint, (Float *));
EXTERN_FUNCTION(BOOLEAN Fisuint, (Float *));
EXTERN_FUNCTION(BOOLEAN Fislong, (Float *));
EXTERN_FUNCTION(BOOLEAN Fisulong, (Float *));
EXTERN_FUNCTION(BOOLEAN Fisdbl, (Float *));

EXTERN_FUNCTION(void FasFplF, (Float *, Float *, Float *));
EXTERN_FUNCTION(void FplasF, (Float *, Float *));
EXTERN_FUNCTION(void FasFplint, (Float *, Float *, int));
EXTERN_FUNCTION(void Fplasint, (Float *, int));
```

```
EXTERN_FUNCTION(void Finc, (Float *));
EXTERN_FUNCTION(void Fabs, (Float *));
EXTERN_FUNCTION(void FasFmiF, (Float *, Float *, Float *));
EXTERN_FUNCTION(void FmiasF, (Float *, Float *));
EXTERN_FUNCTION(void FasFmiint, (Float *, Float *, int));
EXTERN_FUNCTION(void Fmiasint, (Float *, int));
EXTERN_FUNCTION(void Fdec, (Float *));
EXTERN_FUNCTION(void Fneg, (Float *));
EXTERN_FUNCTION(void FasFmuF, (Float *, Float *, Float *));
EXTERN_FUNCTION(void FmuasF, (Float *, Float *));
EXTERN_FUNCTION(void FasFmuint, (Float *, Float *, int));
EXTERN_FUNCTION(void Fmuasint, (Float *, int));
EXTERN_FUNCTION(void FasFdiF, (Float *, Float *, Float *));
EXTERN_FUNCTION(void FdiasF, (Float *, Float *));
EXTERN_FUNCTION(void FasFdiint, (Float *, Float *, int));
EXTERN_FUNCTION(void Fdiasint, (Float *, int));
EXTERN_FUNCTION(void FasintdiF, (Float *, int, Float *));
EXTERN_FUNCTION(void FasFslint, (Float *, Float *, unsigned int));
EXTERN_FUNCTION(void Fslasint, (Float *, unsigned int));
EXTERN_FUNCTION(void FasFsrint, (Float *, Float *, unsigned int));
EXTERN_FUNCTION(void Fsrasint, (Float *, unsigned int));

EXTERN_FUNCTION(void Fsqrt, (Float *, Float *));
EXTERN_FUNCTION(void Fexp, (Float *, Float *));
EXTERN_FUNCTION(void Flog, (Float *, Float *));
EXTERN_FUNCTION(void FasFpowF, (Float *, Float *, Float *));
EXTERN_FUNCTION(void FasFpowint, (Float *, Float *, int));
EXTERN_FUNCTION(void Fbesselj, (Float *, int, const Float));
EXTERN_FUNCTION(void Fe, (Float *));
EXTERN_FUNCTION(void Fpi, (Float *));
EXTERN_FUNCTION(void Feuler, (Float *));
EXTERN_FUNCTION(void Fsin, (Float *, Float *));
EXTERN_FUNCTION(void Fcos, (Float *, Float *));
EXTERN_FUNCTION(void Ftan, (Float *, Float *));
EXTERN_FUNCTION(void Fcot, (Float *, Float *));
EXTERN_FUNCTION(void Fasin, (Float *, Float *));
EXTERN_FUNCTION(void Facos, (Float *, Float *));
EXTERN_FUNCTION(void Fatan, (Float *, Float *));
EXTERN_FUNCTION(void Facot, (Float *, Float *));
EXTERN_FUNCTION(void Fsinh, (Float *, Float *));
EXTERN_FUNCTION(void Fcosh, (Float *, Float *));
EXTERN_FUNCTION(void Ftanh, (Float *, Float *));
EXTERN_FUNCTION(void Fcoth, (Float *, Float *));
EXTERN_FUNCTION(void Fasinh, (Float *, Float *));
EXTERN_FUNCTION(void Facosh, (Float *, Float *));
EXTERN_FUNCTION(void Fatanh, (Float *, Float *));
EXTERN_FUNCTION(void Facoth, (Float *, Float *));
EXTERN_FUNCTION(void Frndtoi, (Float *, Float *));
```

```
EXTERN_FUNCTION(void Fceil, (Float *, Float *));
EXTERN_FUNCTION(void Ffloor, (Float *, Float *));
EXTERN_FUNCTION(void Ftrunc, (Float *, Float *));

EXTERN_FUNCTION(BOOLEAN FeqF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN FgtF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN FneF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN FgeF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN FltF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN FleF, (Float *, Float *));
EXTERN_FUNCTION(BOOLEAN Fge0, (Float *));
EXTERN_FUNCTION(BOOLEAN Fgt0, (Float *));
EXTERN_FUNCTION(BOOLEAN Fle0, (Float *));
EXTERN_FUNCTION(BOOLEAN Flt0, (Float *));
EXTERN_FUNCTION(BOOLEAN Feq0, (Float *));
EXTERN_FUNCTION(BOOLEAN Fne0, (Float *));
EXTERN_FUNCTION(BOOLEAN Fis0, (Float *));
EXTERN_FUNCTION(BOOLEAN FcompF, (Float *, Float *));

EXTERN_FUNCTION(int Fsign, (Float *));
EXTERN_FUNCTION(void Fsetsign, (Float *, int));
EXTERN_FUNCTION(long Fexpo, (Float *));
EXTERN_FUNCTION(void Fsetexpo, (Float *, long));
EXTERN_FUNCTION(int Ftype, (Float *));
EXTERN_FUNCTION(void Fsettype, (Float *, int));
EXTERN_FUNCTION(void Fsetprec, (long));
EXTERN_FUNCTION(int Flead0s, (unsigned long));
EXTERN_FUNCTION(int Fleadzeros, (Float *));
EXTERN_FUNCTION(int Flength, (Float *));
EXTERN_FUNCTION(void Fbinul, (unsigned long));
EXTERN_FUNCTION(void Fbinary, (Float *));
EXTERN_FUNCTION(void Fnorm, (Float *));
EXTERN_FUNCTION(void Froundb, (Float *, unsigned int));
EXTERN_FUNCTION(void Fnormalize, (Float *));
EXTERN_FUNCTION(int Flog2, (Float *));
EXTERN_FUNCTION(int fscanF, (FILE *, Float *));
EXTERN_FUNCTION(int fprintF, (FILE *, Float *));
EXTERN_FUNCTION(int Ftoa, (Float * n, char s[]));
EXTERN_FUNCTION(int atoF, (char s[], Float * n));
EXTERN_FUNCTION(void printF, (char s[], Float * n));
EXTERN_FUNCTION(void Ferror, (char *));
EXTERN_FUNCTION(char *Fmalloc, (int));
EXTERN_FUNCTION(void Ffree, (char *));
EXTERN_FUNCTION(void Fprint_statistics, (void));
#endif
```

HILFSFUNKTIONEN

NAME

Fsign, Fsetsign, Fexpo, Fsetexpo, Ftype, Fsettype, Flength .... Hilfsfunktionen

ÜBERSICHT

```
int Fsign(x)
Float *x;

void Fsetsign(x,y)
Float *x;
int y;

long Fexpo(x)
Float *x;

void Fsetexpo(x,y)
Float *x;
long y;

int Ftype(x)
Float *x;

void Fsettype(x,y)
Float *x;
int y;

int Flength(x)
Float *x;
```

BESCHREIBUNG

Fsign() liefert das Vorzeichen einer Float Zahl x: 0 if x.m$\geq$0, 1 sonst. Fsetsign() setzt das Vorzeichen der Mantisse. Fexpo() liefert den Exponenten, Fsetexpo() setzt ihn. Ftype() liefert den Typ: 0, falls type == integer, 1 falls type == real. Fsettype() setzt den Typ. Flength() ist gleich der Länge der Mantisse in Binärstellen.

BEMERKUNG

Der Exponent ist intern als eine Zahl kleiner gleich Null gespeichert. Die Anzahl der binären Vorkommastellen ist gleich Fexpo()+Flength().

HYPERBOLISCH

NAME

Fsinh, Fcosh, Ftanh, Fcoth, Fasinh, Facosh, Fatanh, Facoth .... hyperbolische Winkelfunktionen

ÜBERSICHT

```
void Fsinh(x,y)
Float *x,*y;

void Fcosh(x,y)
Float *x,*y;

void Ftanh(x,y)
Float *x,*y;

void Fcoth(x,y)
Float *x,*y;

void Fasinh(x,y)
Float *x,*y;

void Facosh(x,y)
Float *x,*y;

void Fatanh(x,y)
Float *x,*y;

void Facoth(x,y)
Float *x,*y;
```

BESCHREIBUNG

Fsinh(), Fcosh(), Ftanh(), Fcoth() berechnen sinh(y), cosh(y), tanh(y), coth(y) und speichern das Ergebnis in x.

Fasinh(), Facosh(), Fatanh(), Facoth() berechnen arsinh(y), arcosh(y), artanh(y), arcoth(y) und speichern das Ergebnis in x.

SIEHE AUCH

Winkelfunktionen, Konstanten

KONSTANTEN

NAME

F_e, F_pi, F_euler, Fe, Fpi, Feuler .... Float Konstanten

ÜBERSICHT

Float F_e, F_pi, F_euler;

void Fe(x)
Float *x;

void Fpi(x)
Float *x;

void Feuler(x)
Float *x;

BESCHREIBUNG

F_e, F_pi und F_euler sind Namen für die Konstanten e, π und γ. Fe(), Fpi() und Feuler() sind die Funktionen die diese Konstanten berechnen und in ihrem Aufrufsargument speichern. Einmal berechnet bei einer Präzision, können diese Konstanten direkt benutzt werden; x wird lediglich die entsprechende Konstante zugewiesen. Wurde die Präzision inzwischen erhöht, so wird die entsprechende Konstante neu berechnet und x zugewiesen.

BEMERKUNG

Eine direkte Zuweisung der Form FasF(&x,&F_pi) (bzw. FasF(&x,&F_euler) oder ...) ist nur möglich, wenn Fpi() (bzw. Feuler() ...) schon mindestens einmal aufgerufen worden ist.

SIEHE AUCH

Winkelfunktionen, Hyperbolisch

KONSTRUKTOREN(1)

NAME

cF, cFasF, cFasI, cFasdbl, cFasint, cFasuint, cFaslong, cFasulong, cFmaxlength, dF .. Konstruktoren und Destruktor

ÜBERSICHT

```
void cF(x)
Float *x;

void cFasF(x,y)
Float *x,*y;

void cFasI(x,y)
Float *x;
Integer *I;

void cFasdbl(x,y)
Float *x;
double y;

void cFasint(x,y)
Float *x;
int y;

void cFasuint(x,y)
Float *x;
unsigned int y;

void cFaslong(x,y)
Float *x;
long y;

void cFasulong(x,y)
Float *x;
unsigned long y;

void cFmaxlength(x,y)
Float *x;
int y;

void dF(x)
Float *x;
```

BESCHREIBUNG

Die obige Funktionen konstruieren eine Float Zahl. Funktionen, deren Name die Silbe 'as' enthält, weisen einen Wert zu (i.a Float, Integer, double, ..., unsigned long). cFmaxlength() konstruiert eine Float Zahl mit maximaler Mantisselänge y. dF() gibt den mit cF(), ..., cFmaxlength() allocierten Speicherplatz frei.

SIEHE AUCH

Konstruktoren(2)

KONSTRUKTOREN(2)

NAME

ncF, ncFasF, ncFasI, ncFasdbl, ncFasint, ncFmaxlength, ddF .... Konstruktoren und Destruktor

ÜBERSICHT

```
Float *ncF(x)
Float *x;

Float *ncFasF(x,y)
Float *x,*y;

Float *ncFasI(x,y)
Float *x;
Integer *I;

Float *ncFasdbl(x,y)
Float *x;
double y;

Float *ncFasint(x,y)
Float *x;
int y;

Float *ncFmaxlength(x,y)
Float *x;
int y;

void ddF(x)
Float *x;
```

BESCHREIBUNG

Die obigen Funktionen konstruieren eine Float Zahl. Funktionen, deren Name die Silbe 'as' enthält, weisen bei der Konstruktion einen Wert zu (i.a Float, Integer, double, int). ncFmaxlength() konstruiert eine Float Zahl mit maximaler Mantisselänge y. ddF() gibt den mit ncF(), ..., ncFmaxlength() allocierten Speicherplatz frei.

BEMERKUNG

Es können höchstens 2^{16} Maschinenworte pro Zahl reserviert werden; eine höhere Anforderung (z.B. via cFmaxlength) führt zu einem Fehler.

SIEHE AUCH

Konstruktoren(1)

MALLOC

NAME

Fmalloc, Ffree, Fprint_statistics .... Speicherverwaltung

ÜBERSICHT

char *Fmalloc(i)
int i;

void Ffree(x)
Float *x;

void Fprint_statistics(void);

BESCHREIBUNG

Fmalloc(i) liefert einen Pointer auf einen Block von mindestens i Bytes. Ffree() gibt einen früher allocierten Block frei. Fprint_statistics() schreibt eine Statistik über die Speichernutzung während der Ausführung eines Programms.

BEMERKUNG

Siehe malloc in 'Sunview Manual Pages'.

SIEHE AUCH

Konstruktoren(1), Konstruktoren(2)

MATHEMATISCHE BIBLIOTHEK(1)

NAME

Fabs, Fneg, Ftrunc, Frndtoi, Fceil, Ffloor .... mathematische Funktionen

ÜBERSICHT

void Fabs(x)
Float *x;

void Fneg(x)
Float *x;

void Ftrunc(x,y)
Float *x,*y;

void Frndtoi(x,y)
Float *x,*y;

void Fceil(x,y)
Float *x,*y;

void Ffloor(x,y)
Float *x,*y;

BESCHREIBUNG

Fabs() setzt x gleich $|x|$. Fneg() negiert sein Argument. Ftrunc() liefert die Vorkommastellen von x zurück. Frndtoi() rundet zur nächsten ganzen Zahl. Fceil() rundet nach oben und Ffloor() rundet nach unten.

SIEHE AUCH

Mathematische Bibliothek(2), Hyperbolisch, Winkelfunktionen.

MATHEMATISCHE BIBLIOTHEK(2)

NAME

Fsqrt, Fexp, Flog, Fbesselj, FasFpowF, FasFpowint ...... mathematische Funktionen

ÜBERSICHT

void Fsqrt(x,y)
Float *x,*y;

void Fexp(x,y)
Float *x,*y;

void Flog(x,y)
Float *x,*y;

void Fbesselj(x,n,y)
Float *x,*y; int n;

void FasFpowF(x,y,z)
Float *x,*y,*z;

void FasFpowint(x,y,z)
Float *x,*y; int z;

BESCHREIBUNG

Fsqrt() liefert in x die Quadratwurzel von y. Fexp() liefert in x e^y. Flog() liefert in x $log\, y$. Fbesselj() liefert in x die Besselfunktion $J_n(y)$. Die Funktionen FasFpowF() und FasFpowint() simulieren den Poweroperator (d.h. x = y^z). In FasFpowF() muß y positiv oder Null sein.

BEMERKUNG

FasFpowF() ruft Fexp() und Flog() auf: ihre Auswertung dauert länger als FasFpowint(), falls z ein int ist.

SIEHE AUCH

Mathematische Bibliothek(1), Hyperbolisch, Winkelfunktionen.

MULTIPLIKATION

NAME

FasFmuF, FmuasF, FasFmuint, Fmuasint .... Multiplikation

ÜBERSICHT

```
void FasFmuF(x,y,z)
Float *x,*y,*z;

void FmuasF(x,y)
Float *x,*y;

void FasFmuint(x,y,z)
Float *x,*y;
int z;

void Fmuasint(x,y)
Float *x;
int y;
```

BESCHREIBUNG

FasFmuF() und FmuasF() entsprechen den Operatoren $*$ bzw. $*=$ für Float Zahlen (d.h. $x = y * z$, $x *= y$). FasFmuint() und Fmuasint() entsprechen den Operatoren $*$ bzw. $*=$ für eine Float Zahl und ein int (d.h. $x = y * z$, $x *= y$).

BEMERKUNG

FasFmuF() darf nicht mit y == z aufgerufen werden.

SIEHE AUCH

Addition, Subtraktion, Division, Shift

PRÄZISION

NAME

Fsetprec, DecPrec, BinPrec, Prec .... Präzision

ÜBERSICHT

void Fsetprec(x)
long x;

long DecPrec,BinPrec,Prec;

BESCHREIBUNG

Fsetprec() setzt die momentane Präzision DecPrec auf x Dezimalstellen. BinPrec ist die benötigte Anzahl von Binärstellen um Präzision DecPrec zu garantieren, und Prec die Anzahl von Maschinenworten, die für BinPrec Stellen benötigt werden. DecPrec, BinPrec und Prec sind globale Variablen.

BEMERKUNG

Die maximale Präzision, mit der gerechnet werden kann, ist MAXDECPREC, wie in ffloat.h definiert.

SIEHE AUCH

ffloat.h

SHIFT

NAME

FasFslint, Fslasint, FasFsrint, Fsrasint .... Shift

ÜBERSICHT

void FasFslint(x,y,z)
Float *x,*y;
unsigned int z;

void Fslasint(x,y)
Float *x;
unsigned int y;

void FasFsrint(x,y,z)
Float *x,*y;
unsigned int z;

void Fsrasint(x,y)
Float *x;
unsigned int y;

BESCHREIBUNG

FasFslint() und Fslasint() entsprechen den Operatoren << bzw. <<= für eine Float Zahl und ein unsigned int (d.h. x = y << z, x <<= y). FasFsrint() und Fsrasint() entsprechen den Operatoren >> bzw. >>= für eine Float Zahl und ein unsigned int (d.h. x = y >> z, x >>= y).

BEMERKUNG

Da die Anzahl der Shiftstellen als unsigned int implementiert ist, führt ein Aufruf mit einer negativen Zahl im dritten Argument zu einem Speicherfehler.

SIEHE AUCH

Addition, Subtraktion, Multiplikation, Division

SUBTRAKTION

NAME

FasFmiF, FmiasF, FasFmiint, Fmiasint, Fdec .... Subtraktion

ÜBERSICHT

```
void FasFmiF(x,y,z)
Float *x,*y,*z;

void FmiasF(x,y)
Float *x,*y;

void FasFmiint(x,y,z)
Float *x,*y;
int z;

void Fmiasint(x,y)
Float *x;
int y;

void Fdec(x)
Float *x;
```

BESCHREIBUNG

FasFmiF() und FmiasF() entsprechen den Operatoren − bzw. − = für Float Zahlen (d.h. x = y − z, x − = y). FasFmiint() und Fmiasint() entsprechen den Operatoren − bzw. − = für eine Float Zahl und einen int (d.h. x = y − z, x − = y). Fdec() entspricht dem −− operator.

BEMERKUNG

FasFmiF() darf nicht mit y == z aufgerufen werden.

SIEHE AUCH

Addition, Multiplikation, Division, Shift

TYPÜBERPRÜFUNG

NAME

Fisdbl, Fisint, Fislong, Fisuint, Fisulong .... Typüberprüfung

ÜBERSICHT

BOOLEAN Fisdbl(x)
Float *x;

BOOLEAN Fisint(x)
Float *x;

BOOLEAN Fislong(x)
Float *x;

BOOLEAN Fisuint(x)
Float *x;

BOOLEAN Fisulong(x)
Float *x;

BESCHREIBUNG

Fisdbl() liefert 1 falls x als double gespeichert werden kann, 0 sonst. Fisint(), Fislong(), Fisuint(), Fisulong() funktionieren entsprechend für die Typen int, long, unsigned int bwz. unsigned long.

BEMERKUNG

Die obigen Funktionen dienen als Vorsichtsmaßnahme vor einer Zuweisung $double = Float$, ...,$unsignedlong = Float$.

SIEHE AUCH

Zuweisung

VERGLEICHE(1)

NAME

Feq0, Fge0, Fgt0, Fis0, Fle0, Flt0 .... Vergleich mit 0

ÜBERSICHT

BOOLEAN Feq0(x)
Float *x;

BOOLEAN Fge0(x)
Float *x;

BOOLEAN Fgt0(x)
Float *x;

BOOLEAN Fis0(x)
Float *x;

BOOLEAN Fle0(x)
Float *x;

BOOLEAN Flt0(x)
Float *x;

BOOLEAN Fne0(x)
Float *x;

BESCHREIBUNG

Feq0() liefert (x.m==0), wobei x.m die Mantisse von x ist. Fge0() liefert ($x \geq 0$). Fgt0() liefert ($x>0$). Fis0() liefert (x==0). Fle0() liefert ($x \leq 0$). Flt0() liefert ($x<0$). Fne0() liefert (x!=0).

Feq0() liefert 1 nur dann, wenn die Mantisse ihres Argumentes 0 ist. Sei zum Beispiel 0.1e-10 bei Präzision von 5 Dezimalstellen. In diesem Fall liefert Feq0(0.1e-10) 0, da die Mantisse 0.1 ungleich 0 ist. Falls der 'echte' Vergleich mit 0 erwünscht ist, benutze Fis0(): diese Funktion liefert 1, falls die Mantisse von x gleich 0 bezüglich der momentanen Präzision ist, 0 sonst. Im obigen Beispiel liefert Fis0(0.1e-10) 1, da x bei Präzision von 5 Dezimalstellen 'gleich' 0 ist.

SIEHE AUCH

Vergleiche(2)

VERGLEICHE(2)

NAME

FeqF, FgeF, FgtF, FleF, FltF, FneF, FcompF .... Vergleich zweier Float Zahlen

ÜBERSICHT

```
BOOLEAN FeqF(x,y)
Float *x,*y;

BOOLEAN FgeF(x,y)
Float *x,*y;

BOOLEAN FgtF(x,y)
Float *x,*y;

BOOLEAN FleF(x,y)
Float *x,*y;

BOOLEAN FltF(x,y)
Float *x,*y;

BOOLEAN FneF(x,y)
Float *x,*y;

BOOLEAN FcompF(x,y)
Float *x,*y;
```

BESCHREIBUNG

FeqF() liefert (x==y). FgeF() liefert ($x \geq y$). FgtF() liefert (x>y). FleF() liefert ($x \leq y$). FltF() liefert (x<y). FneF() liefert (x!=y). FcompF() liefert 1, 0, -1 je nachdem, ob >, == oder < wahr ist.

SIEHE AUCH

Vergleiche(1)

WINKELFUNKTIONEN

NAME

Fsin, Fcos, Ftan, Fcot, Fasin, Facos, Fatan, Facot .... Winkelfunktionen

ÜBERSICHT

```
void Fsin(x,y)
Float *x,*y;

void Fcos(x,y)
Float *x,*y;

void Ftan(x,y)
Float *x,*y;

void Fcot(x,y)
Float *x,*y;

void Fasin(x,y)
Float *x,*y;

void Facos(x,y)
Float *x,*y;

void Fatan(x,y)
Float *x,*y;

void Facot(x,y)
Float *x,*y;
```

BESCHREIBUNG

Fsin(), Fcos(), Ftan(), Fcot() berechnen sin(y), cos(y), tan(y), cot(y) und speichern das Ergebnis in x.

Fasin(), Facos(), Fatan(), Facot() berechnen arcsin(y), arccos(y), arctan(y), arccot(y) und speichern das Ergebnis in x.

SIEHE AUCH

Hyperbolisch, Konstanten

ZUWEISUNG

NAME

FasF, FasI, Fasdbl, Fasint, Fas0, Fas1, IasF, dblasF .... Zuweisung

ÜBERSICHT

```
void FasF(x,y)
Float *x,*y;

void FasI(x,y)
Float *x;
Integer *y;

void Fasdbl(x,y)
Float *x;
double y;

void Fasint(x,y)
Float *x;
int y;

void Fas0(x)
Float *x;

void Fas1(x)
Float *x;

void IasF(x,y)
Integer *x;
Float *y;

void dblasF(x,y)
double *x;
Float *y;
```

BESCHREIBUNG

Die hier aufgelisteten Funktionen simulieren den Zuweisungsoperator: FasF() ist die Zuweisung für Float Zahlen. FasI(), Fasdbl(), Fasint() weisen einer Float Zahl eine Integer, double und int Zahl zu. Fas0(), Fas1() weisen dem Aufrufsargument den Wert 0, bzw. 1 zu. IasF() dient zur Kompatibilität zwischen Float und Integer. dblasF() ist die *double = Float* Zuweisung. Durch sie kann man auch die *int = Float* Zuweisung simulieren, indem man die cast Funktion (int)(...) benutzt.

Literaturverzeichnis

[1] Abramowitz, M. und Stegun, I.A.: *Handbook of mathematical functions with formulas, graphs and mathematical tables.* National Bureau of Standards, Washington, D.C., 1964

[2] Apostol, M. Tom: *CALCULUS.* Blaisdell Publ. Co., New York, 1962

[3] Brent, P. Richard: *The Complexity of Multiple-Precision Arithmetic.* Complexity of Computational Problem Solving (R.S Anderssen and R.P Brent Eds.), University of Queensland Press, Brisbane, 1976

[4] Brent, P. Richard: *Fast Multiple-Precision Evaluation of Elementary Functions.* Journal of ACM, Vol. 23, Seite 242, 1976

[5] Brent, P. Richard: *Some New Algorithms for High-Precision Computation of Euler's Constant.* Mathematics of Computation, Vol. 34, Nr. 149, Seite 305, January 1980

[6] Brent, P. Richard: *Multiple-precision zero finding methods and the complexity of elementary function evaluation.* Proc. Symp. on Analytic Computational Complexity, Seite 151-176, J.F. Traub, Ed., Academic Press, New York, 1976

[7] Carlson, B.C: *An Algorithm for Computing Logarithms and Arctangents.* Mathematics of Computation, Vol. 26, Nr. 118, Seite 543, April 1972

[8] Cohen, Henri: *Algorithms in Computational Algebraic Number Theory.* 1991

[9] Dentzer, Ralf: *libI: eine lange ganzzahlige Arithmetik.* Heidelberg, 1991

[10] Forster, Otto: *Analysis I.* Vieweg Verlag, 1983

[11] Kernigham, W.Brian und Ritchie, M.Dennis: *Programmieren in C.* Carl Hanser Verlag, 1983

[12] Knuth, E. Donald: *The Art of Computer Programming, Vol. 2.* Addison–Wesley Co., Inc., 1980

[13] Merscher, Patrick: *Standards für objektorientiertes Programmieren in der Computeralgebra.* Diplomarbeit, Universität des Saarlandes, 1992

[14] Perron, Oskar: *Die Lehre von den Kettenbrüchen I, II.* B.G Teubner, Stuttgart, 1977

[15] Ramanujan, Srinivasa: *Collected Papers of Srinivasa Ramanujan.* Cambridge University Press, Campridge, 1927

[16] Shanks, D. und Wrench, J.W: *The Calculation of π to 100,000 decimals.* Mathematics of Computation, Vol. 16, Seite 76, 1962

[17] Stoer, Josef: *Einführung in die numerische Mathematik I.* Springer-Verlag, 1983

[18] Sweeny, D.W: *On the Computation of Euler's Constant.* Mathematics of Computation, Vol. 15, 1961

[19] Watson, G.N.: *A Treatise on the Theory of Bessel Functions.* 2nd ed., Cambridge University Press, London, 1944

Stichwortverzeichnis