Approximation von Standardfunktionen

Markus Triska, Mat-Nr.: 0225855, Kennzahl: 534
Betreuer: Christoph Überhuber

10. November 2004

1 Allgemeines

Folgendes ANSI-C Programm berechnet durch Bisektion die relative Maschinengenauigkeit *eps* des C-Datentyps `double` auf einer konkreten Maschine.

```
(1)   #include <stdio.h>
(2)   int main() {
(3)       double eps0 = 1, epssave, epsplus1;
(4)       int iterations = 0;
(5)       while (1) {
(6)            iterations++; epssave = eps0; eps0 = eps0/2;
(7)            epsplus1 = eps0 + 1;
(8)            if (epsplus1 <= 1) break;
(9)       }
(10)      printf("\n\n%e %d\n", epssave, iterations);
(11)  }
```

Auf meinem Intel Pentium 4 führt das Programm zur Ausgabe:

```
 2.220446e-16  53
```

d.h., nach 53 Iterationen wurde $eps \approx 10^{-16}$ ermittelt.

Die in Haskell verfassten Programme führe ich mit dem Haskell 98 Interpreter `hugs98` in der Version 'November 2003' aus. Haskell unterstützt ebenfalls den Datentyp `Double`, der laut Haskell 98 Report idealerweise zumindest den IEEE Standard umfasst. Tatsächlich scheint `hugs98` genau das zu tun (und kaum mehr):

```
findEps oldeps
     | (1 + oldeps) <= 1 = oldeps
     | otherwise = findEps (oldeps/2)

Main> findEps 1.0
1.11022302462516e-16
Main> 10**308.2547
1.79762872820855e+308
Main> 10**308.2548
inf
```

Für die weitere Betrachtung wollen wir daher eine Architektur mit wenigstens 15 signifikanten Dezimalstellen voraussetzen.

2 Die Exponentialfunktion

Es soll die Exponentialfunktion e^x numerisch approximiert werden. Die Approximation erfolgt durch ein Taylor-Polynom, d.h. die ersten $(n+1)$-Summanden der Taylorreihe, mit Anschlusstelle $x_0 = 0$:

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} + R$$

wobei R das Restglied bezeichnet, in der Darstellung nach Lagrange:

$$R = \frac{x^{n+1}}{(n+1)!} e^{\vartheta x} \quad \text{mit} \quad \vartheta \in [0,1]$$

Folgendes Haskell-Programm liefert die Summe der ersten n Glieder der Taylorreihe von e^x mit Anschlussstelle $x_0 = 0$:

```
fact n = product [2..n]
expApprox x n
    | n <= 0 = 0
    | otherwise = sum [(x^k) / fromInteger (fact k)
              | k <- [0..(n-1)]]
```

Die Approximation gilt dabei prinzipiell für beliebige Werte von $x \in \mathbb{F} \subset \mathbb{R}$, muss aber aus praktischen Gründen eingegrenzt werden: Da in der zugrundeliegenden Arithmetik nur Zahlen bis etwa 10^{308} codiert werden können, kann die Berechnung jedenfalls höchstens (wegen der Monotonie der Exponentialfunktion) für x mit $e^x \leq 10^{308}$ gelten, d.h. $x_{max} \leq 710$. Für bessere Abschätzungen siehe unten.

Da auch keiner der in den Summanden enthaltenen Terme betragsmäßig 10^{308} übersteigen darf, ergibt sich bei obiger naiver Summation aus $x^n \leq 10^{308}$ sofort $n \leq \frac{308}{\log x}$, d.h. auch der Approximationsgenauigkeit sind Grenzen gesetzt.

Beschränken wir die Betrachtung probeweise auf $x \in [0, 100]$, so ergibt sich, dass im Extremfall (nämlich $x = 100$) die Approximation nur mehr höchstens bis etwa $n = \frac{308}{\log 100} = 154$ durchgeführt werden sollte (konkreter Versuch liefert $n = 155$, es wurde mit 10^{308} die tatsächliche Schranke leicht unterschätzt), da sonst ein (positiver) Überlauf zu befürchten ist. Das in diesem Fall ($n = 154$) gelieferte Ergebnis ist

```
Main> expApprox 100 154
2.68811624723737e+43
Main> expApprox 100 156
inf
Main> exp 100
2.68811714181614e+43
```

was wenigstens noch auf 6 Stellen genau ist (verglichen mit dem tatsächlichen, mittels der Haskell Standard-Funktion `exp` berechneten Wert).

Wir verwenden nun das Restglied um zu einer Abschätzung der Zahl der benötigten Summanden zu gelangen, wobei eine Genauigkeit von $100eps$ angestrebt wird, d.h. in unserem Fall etwa 14 signifikante Dezimalstellen. Das Restglied in Lagranger Darstellung enthält eine Unbekannte (ϑ), die sich aus der Herleitung der Darstellung unter Verwendung des Mittelwertsatzes der Integralrechnung ergibt. Wir beschränken die Betrachtung auf die pessimistischen Abschätzungen $\vartheta = 0$ (für negative x) und $\vartheta = 1$ (für positive x). Der Trivialfall $e^0 = 1$ erfordert keine Abschätzung des Restgliedes.

Aufgrund der oben (willkürlich) festgelegten Genauigkeitsforderung darf der relative Fehler betragsmäßig $0.5 \cdot 10^{-14}$ (0.5 ist "Sicherheitsfaktor") nicht übersteigen, es muss also gelten (das Restglied R entspricht dem absoluten Fehler)

$$\left| \frac{x^{n+1}}{(n+1)!} e^{x(\vartheta-1)} \right| \leq 5 \cdot 10^{-15}$$

Es könnte nun für jedes gegebene x das mindestens benötigte n durch Umformungen und Approximationen (wie etwa der Stirling'schen Approximation $\ln n! \approx n \cdot \ln n - n$) abgeschätzt werden. Da die Verwendung derartiger Approximationen i.a. größere n voraussetzt, die geforderte Genauigkeit und Durchführbarkeit der Rechnung aber eher zu kleinen n führt, verwende ich stattdessen folgendes `bc`-Programm (`bc` ist eine Programmiersprache mit beliebiger Genauigkeit, ich verwende Version 1.06), um n direkt (iterativ) zu berechnen.

```
define fact(x) {
    if (x <= 1) return 1
    return fact(x-1)*x
}

define abs(x) {
    if (x < 0) return -x
    return x
}

define findn(x) {
     n = 0
     theta = 1
     if (x < 0) theta = 0

     while (abs((x^(n + 1)*e(x*(theta - 1)) / fact(n+1))) > \
       5*10^(-15)) {
         n = n + 1
     }
     return n
}
```

Es ergibt sich nun für $x = 100$:

```
findn(100)
299
```

d.h., in obigem Beispiel wären – um der gewünschten Genauigkeitsforderung zu genügen – mindestens 300 Summanden nötig gewesen, wobei wie gesagt eine (für die Laufzeiteffizienz) optimistische Schätzung des Restgliedes vorgenommen wurde. Ich verwende `bc` auch, um (wieder iterativ) bessere obere Schranken für den sich ergebenden (numerischen) Definitionsbereich zu errechnen.

Beschränkt man die Betrachtung auf Potenzen von x, so ergibt sich $x_{max} \leq 55$. Das Ergebnis ist sinnvoll, da `findn(55)` $= 176$, d.h. es müssten 177 Summanden berücksichtigt werden, wobei einer den Term $55^{176} < 10^{308}$ enthält, aber $55^{177} > 10^{308}$, d.h. dies ist tatsächlich eine obere Schranke in der zugrundeliegenden Architektur (`findn(56)` $= 179$). Aus der Betrachtung der $k!$-Terme in den Nennern ergibt sich sogar $x_{max} \leq 52$, da `findn(53)` $= 171$ und $171! > 10^{308}$. Da `findn(-42)` $= 172$ ergibt sich $x_{min} > -42$.

Abbildung 1 veranschaulicht den Zusammenhang von Werten des Arguments x mit der Zahl der mindestens benötigten Summanden des Taylor-Polynoms.

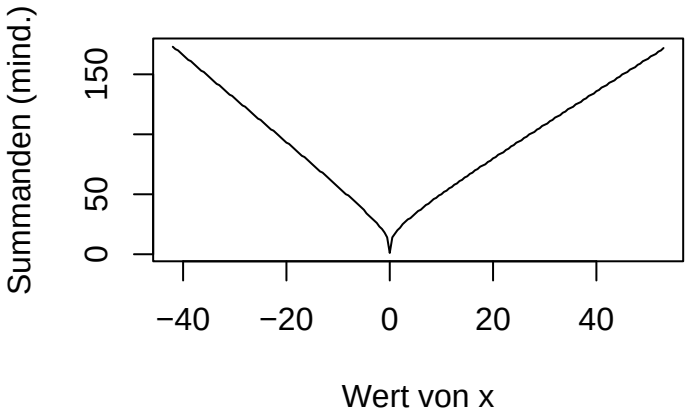

Abbildung 1: Anzahl mind. notwendiger Summanden e^x, naive Summation

2.1 Einige konkrete Werte

Wir wollen nun durch Einsetzen und nähere Betrachtung konkreter Werte die Plausibilität der soeben durchgeführten Überlegungen überprüfen. Tabelle 1 listet die bei gegebenen x-Werten entstehenden relativen Fehler auf, wobei wieder die interne Haskell-Funktion als Referenz dient. Der relative Fehler wurde durch das Haskell-Programm

```
rel x n = abs(((exp x) - expApprox x (n+1)) / (exp x))
```

berechnet. Dabei ist $n =$ `findn(x)`.

x	n	relativer Fehler
1	16	$8.16856 \cdot 10^{-16}$
−1	17	$1.50895 \cdot 10^{-16}$
10	49	$3.30328 \cdot 10^{-16}$
−10	55	$7.23419 \cdot 10^{-16}$
20	79	$2.45709 \cdot 10^{-16}$
−20	92	1.32023

Tabelle 1: Relative Fehler für konkrete Werte (`expApprox`)

Der sehr große relative Fehler bei $x = -20$ erfordert genauere Betrachtung. Es werden hierbei 93 Terme addiert, die (sofern man in "natürlicher" Reihenfolge summiert) betragmäßig zuerst ansteigen und dann monoton fallen. Zur Veranschaulichung in logarithmischer Skala siehe Abbildung 2. Der entstehende Fehler resultiert *nicht* (in erster Linie) aus Rechenfehlern, da bei umgekehrter Summation zur Minderung von Rechenfehlern immer noch ein relativer Fehler von 0.88934 auftritt. Die Fehlerquelle ist *Auslöschung*, da wegen der in diesem Fall alternierenden Vorzeichen in der Reihenentwicklung annähernd gleich große Zahlen voneinander abgezogen werden.

2.2 Verbesserung der Approximation

Um die Approximationsgenauigkeit zu verbessern, verwenden wir die Methode der *Argumentreduktion*, die im wesentlichen die für die Exponentialfunktion geltende Identität $e^x = e^{k \ln 2 + r} = 2^k e^r$, sofern $x = k \ln 2 + r$, ausnützt. Es reicht somit, die

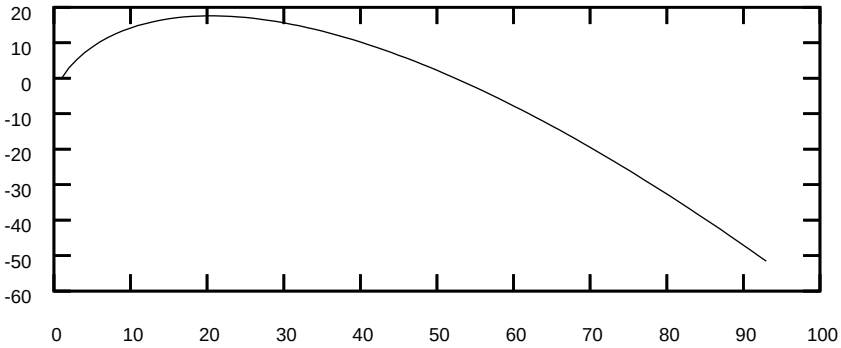

Abbildung 2: Beträge der Summanden für e^{-20}, naive Summation

Exponentialfunktion lediglich auf dem Intervall $I = [0, \ln 2]$ durch die Taylorreihe zu approximieren, da außderdem $e^{-\alpha^2} = 1/e^{\alpha^2}$. Wir verwenden zur Approximation für alle $x \in I$ die ersten 15 Summanden der Taylorreihe, die zur Verminderung von Rechenfehlern in umgekehrter Reihenfolge summiert werden. Die Methode wird von folgendem Haskell-Programm implementiert:

```
findkr x = findkr_helper x 0
    where findkr_helper x k
        | x <= log(2) = (k,x)
        | otherwise = findkr_helper (x-log(2)) (k+1)

expApproxRed x
    | x < 0 = 1 / (expApproxRed (abs x))
    | x <= log(2) = sum (reverse [(x^k) / fromInteger (fact k) |
          k<-[0..14]])
    | otherwise = 2^k * (expApproxRed r) where (k,r) = findkr x
```

Für einige konkrete Werte ergeben sich die in Tabelle 2 angeführten relativen Fehler. Man beachte die signifikante Verbesserung bei großen negativen Exponenten. Außerdem konnte mit dieser Methode der numerische Definitionsbereich beträchtlich vergrößert werden.

x	relativer Fehler
10	$9.90984 \cdot 10^{-16}$
-10	$8.95543 \cdot 10^{-16}$
20	$6.38843 \cdot 10^{-15}$
-20	$6.42111 \cdot 10^{-15}$
100	$1.45710 \cdot 10^{-13}$
-100	$1.45602 \cdot 10^{-13}$
709	$1.43608 \cdot 10^{-11}$
-709	$1.43605 \cdot 10^{-11}$

Tabelle 2: Relative Fehler für konkrete Werte (`expApproxRed`)

3 Die Cosinusfunktion

Es soll eine Funktionsprozedur für die Cosinusfunktion $\cos x$ entwickelt werden. Zur Approximation der Funktion verwenden wir wieder die ersten $(n+1)$ Summanden der Taylorreihe des Cosinus' mit Anschlusstelle $x_0 = 0$:

$$\cos x = \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!} + R$$

Für das Restglied R gilt

$$R = \cos x - \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!} = (-1)^{n+1} \frac{x^{2n+2}}{(2n+2)!} \cos(\vartheta x), \quad mit\ \vartheta \in [0,1].$$

Wegen $|\cos x| \leq 1$ gilt

$$|R| \leq \left| (-1)^{n+1} \frac{x^{2n+2}}{(2n+2)!} \right| = \left| \frac{x^{2n+2}}{(2n+2)!} \right|$$

Es werden wieder etwa 14 richtige Stellen angestrebt (wobei zusätzlich ein "Sicherheitsfaktor" von 0.5 einbezogen wird), das heißt wir können aus der Ungleichung

$$\left| \frac{x^{2n+2}}{(2n+2)! \cdot \cos x} \right| \leq 5 \cdot 10^{-15}$$

wieder iterativ die mindestens benötigte Anzahl an Summanden für jedes x bestimmen. Die Ungleichung ist nur für $x_k = \frac{(1+2k)\pi}{2}$ nicht definiert, da für diese Stellen trivialerweise $\cos x_k = 0$. Aufgrund der endlichen relativen Maschinengenauigkeit $eps \approx 10^{-16}$ gibt es um jede (reelle) Nullstelle von $\cos x$ eine Umgebung, in der höchstens eine Zahl $z \in \mathbb{F}$ liegt. Aus diesem Grund kann n in der Umgebung nicht beliebig groß werden. Gleichzeitig ist damit auch klar, dass beim Bestimmen von Schranken für die naive Summation nur die Umgebungen (aus $\mathbb{F}$) von $\frac{(1+2k)\pi}{2}$ betrachtet werden müssen. Es ergibt sich $x_{max} < \frac{27}{2}\pi \approx 42.41$, da in einer eps-Umgebung relativ zu dieser Stelle wenigstens 87 Summanden auszuwerten wären, wobei einer den Term $(2 * 87)! = 174! > 10^{308}$ enthält. Aus Symmetriegründen ergibt sich $x_{min} > -42.41$. Diese Werte wurden mit etwas höherer Genauigkeit ermittelt (10^{-20}), was den Sachverhalt aber nicht übermäßig verzerrt: Berücksichtigt man nämlich lediglich den *absoluten* Fehler, so sind bereits bei $x = 54$ mindestens 88 Summanden erforderlich, d.h. diese Schranke beschränkt die naive Summation mindestens.

Die Anzahl der mindestens notwendigen Summanden in Abhängigkeit vom Wert von x ist in Abbildung 3 grob skizziert. Man erkennt bereits bei der verwendeten Abszissen-Schrittweite von $\Delta x = 0.1$ ein leichtes "Ausfransen" des Graphen. Abbildung 4 stellt den willkürlich herausgegriffenen Abschnitt $[5\pi, 6\pi]$ dar, wobei $\Delta x = 0.01$. Man beachte das Ansteigen der notwendigen Summanden bei der in diesem Intervall liegenden Nullstelle von $\cos x$, nämlich $x = \frac{11}{2}\pi \approx 17.28$. In Abbildung 5 wird ein noch kleineres Intervall mit $\Delta x = 10^{-8}$ betrachtet.

Die naive Summation kann in Haskell wie folgt realisiert werden:

```
cosApprox x n =
    sum [((-1)^k)*x^(2*k) / fromInteger (fact (2*k)) |
         k <- [0..(n-1)]]
```

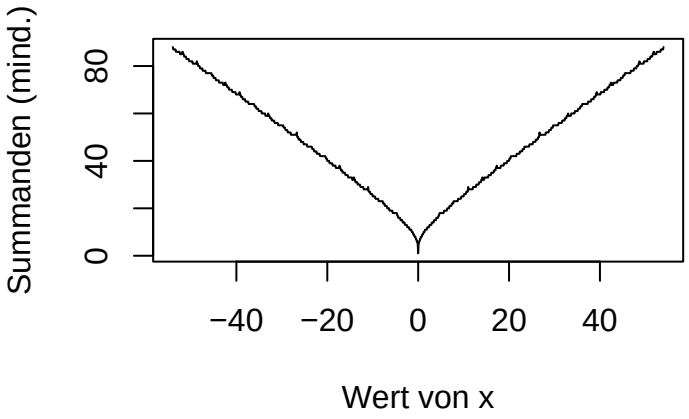

Abbildung 3: Anzahl mind. notwendiger Summanden, $\cos x$

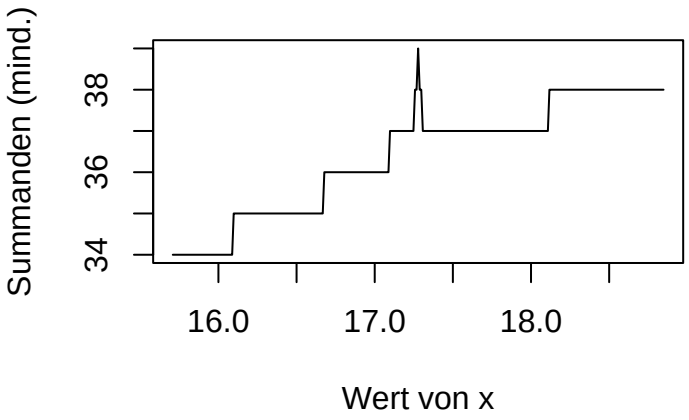

Abbildung 4: Anzahl mind. notwendiger Summanden, $\cos x$

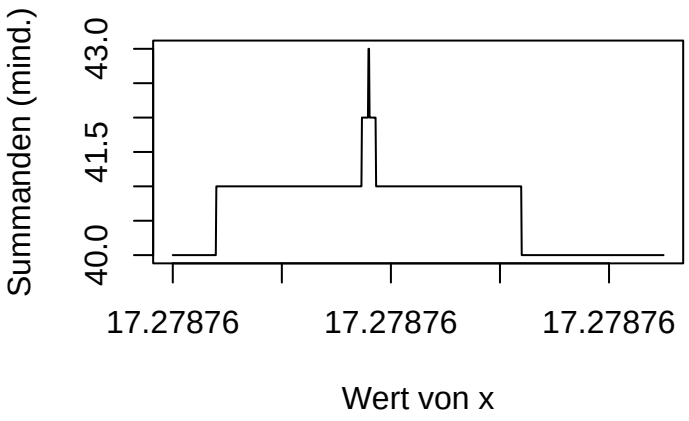

Abbildung 5: Anzahl mind. notwendiger Summanden, $\cos x$

3.1 Einige konkrete Werte

Wir wollen nun wieder durch Einsetzen einiger konkreter Werte die auftretenden relativen Fehler messen und in Tabelle 3 festhalten. Es fällt auf, dass die Approximation für große Argumente nicht gut funktioniert. Der Grund ist wie oben *Auslöschung*, da in der Taylorreihe inhärent alternierende Vorzeichen auftreten, und bei großen Argumenten viele betragsmäßig fast gleich große Terme mit unterschiedlichem Vorzeichen addiert werden. Abbildung 6 veranschaulicht die Beträge der einzelnen Summanden für $x = 33$. Auch hier ist deswegen durch Umordnung der Summanden keine signifikante Verbesserung der Genauigkeit zu erwarten.

x	n	relativer Fehler
1	8	0
2	11	$1.33393 \cdot 10^{-16}$
4	15	$1.01911 \cdot 10^{-15}$
8	22	$7.64947 \cdot 10^{-14}$
16	33	$1.24237 \cdot 10^{-10}$
32	56	0.00118

Tabelle 3: Relative Fehler für konkrete Werte (`cosApprox`)

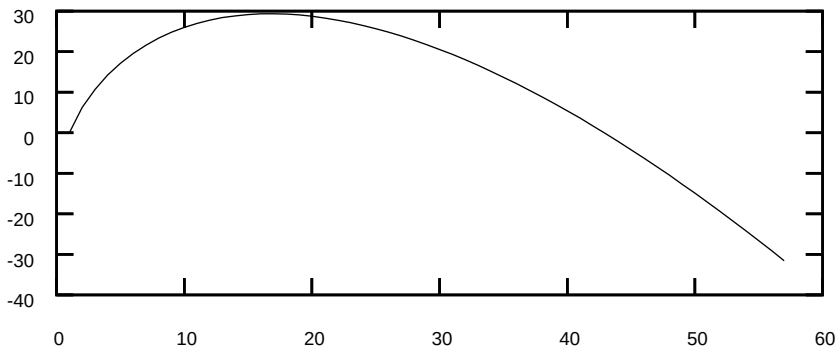

Abbildung 6: Beträge der Summanden für $\cos 32$, naive Summation

3.2 Verbesserung der Approximation

Es wird nun wieder versucht, durch *Argumentreduktion* die Genauigkeit der Approximation zu verbessern. Es reicht, $\cos x$ für $x \in [0, \pi]$ zu approximieren, wobei die Summanden der Taylorreihe wieder in umgekehrter Reihenfolge addiert werden. Für $x \in (\pi, 2\pi]$ ist $\cos x = -\cos(x - \pi)$. Für $x > 2\pi$ stellt man x dar als $x = 2n\pi + r$, $0 \leq r < \pi$ und verwendet die Periodizität von cos: $\cos(2n\pi + r) = \cos r$. Für $x < 0$ verwendet man $\cos(-x) = \cos x$.

```
cosApproxRed1 x
    | (x >= 0) && (x <= pi) = sum [((-1)^k)*x^(2*k) /
        fromInteger (fact (2*k)) | k<-[0..17]]
    | (x > pi) && (x <= 2*pi) = - (cosApproxRed1 (x - pi))
    | x > 2*pi = cosApproxRed1 (findr x)
    | x < 0 = cosApproxRed1 (-x)
    where findr x
            | x < 2 *pi = x
            | otherwise = findr (x - 2*pi)
```

Mit dieser Implementierung ergeben sich die in Tabelle 4 festgehaltenen relativen Fehler für einige konkrete Werte. Neben der beachtlichen Verbesserung der Approximation konnte durch die Argumentreduktion der numerische Definitionsbereich wieder beträchtlich vergrößert werden.

x	relativer Fehler
1	$2.50548 \cdot 10^{-16}$
2	$1.33393 \cdot 10^{-16}$
4	$3.39703 \cdot 10^{-16}$
8	$1.71683 \cdot 10^{-15}$
16	$3.47793 \cdot 10^{-16}$
32	$1.06468 \cdot 10^{-15}$
887	$3.57576 \cdot 10^{-12}$
887.5	$2.58211 \cdot 10^{-08}$
888	$3.56157 \cdot 10^{-12}$

Tabelle 4: Relative Fehler für konkrete Werte (`cosApproxRed1`)

Wir wollen nun den großen relativen Fehler bei $x = 887.5$ näher untersuchen. Obiges Programm reduziert das Argument unter Verwendung von $x \approx 141 \cdot 2\pi + 1.57087168768025$. Rechnet man (z.B. mittels `bc`) allerdings mit erhöhter Genauigkeit von 25 Stellen, so erhält man $\hat{r} = 1.5708716876783067535346 76$. Rundet man das auf 14 Stellen und berechnet den relativen Fehler von `cosApproxRed` für diese gerundete Zahl, so ergibt sich der weitaus bessere Wert $1.296611 \cdot 10^{-13}$. Der Grund für den großen relativen Fehler an der Stelle ist folglich eine zu wenig genaue Argumentreduktion. Der Fehler kann gemildert werden, indem man die Anzahl der Subtraktionen verringert, z.B.:

```
cosApproxRed2 x
    | (x >= 0) && (x <= pi) = sum [((-1)^k)*x^(2*k) /
         fromInteger (fact (2*k)) | k<-[0..17]]
    | (x > pi) && (x <= 2*pi) = - (cosApproxRed2 (x - pi))
    | x > 2^15*pi = cosApproxRed2 (x - 2^15*pi)
    | x > 2^14*pi = cosApproxRed2 (x - 2^14*pi)
    ...
    | x > 2^2*pi = cosApproxRed2 (x - 2^2*pi)
    | x > 2*pi = cosApproxRed2 (findr x)
    | x < 0 = cosApproxRed2 (-x)
    where findr x
             | x < 2 *pi = x
             | otherwise = findr (x - 2*pi)
```

Der relative Fehler an der kritischen Stelle kann allein mit dieser simplen Maßnahme auf $4.59224 \cdot 10^{-10}$ reduziert werden.

Die Funktion `cosApproxRed3` nützt "noch mehr" Periodizität der Cosinusfunktion aus und verwendet bei der Argumentreduktion Division anstelle der wiederholten Subtraktion. Tabelle 5 zeigt die bei Verwendung dieser Funktion auftretenden Fehler für dieselben Argumente wie oben.

```
cosApproxRed3 x
    | (x >= 0) && (x <= pi/2) = sum [((-1)^k)*x^(2*k) /
        fromInteger (fact (2*k)) | k<-[0..17]]
    | (x > pi/2) && (x <= pi) = - (cosApproxRed3 (pi - x))
```

```
| (x > pi) && (x <= 2*pi) = - (cosApproxRed3 (x - pi))
| x > 2*pi = cosApproxRed3 (x - 2*pi*
    (fromInteger (floor (x/(2*pi)))))
| x < 0 = cosApproxRed3 (-x)
```

x	relativer Fehler
1	$2.50548 \cdot 10^{-16}$
2	$1.33393 \cdot 10^{-16}$
4	$3.39703 \cdot 10^{-16}$
8	$2.47988 \cdot 10^{-15}$
16	$3.47793 \cdot 10^{-16}$
32	$1.06468 \cdot 10^{-15}$
887	$1.54481 \cdot 10^{-13}$
887.5	$1.11961 \cdot 10^{-09}$
888	$1.54554 \cdot 10^{-13}$

Tabelle 5: Relative Fehler für konkrete Werte (`cosApproxRed3`)