RS 04 (HA) zum 09.11.2012

Paul Bienkowski

15. November 2012

1. a) Dieses Verfahren ist ungenau bei besonders kleinen Werten. Da Fließkommazahlen (floats) aufgrund der flexiblen Genauigkeit auch besonders kleine Werte darstellen können, kann bei Werten nahe der Konstante die Differenz stark abweichen. Werte, die kleiner sind als die Epsilon-Konstante können auf diese Weise erst recht nicht miteinander verglichen werden.

Außerdem ist die Berechnung der Differenz, das Bilden des Betrags und der Vergleich mit einer anderen Zahl insgesamt weniger effizient, und sollte daher in zeitintensiven Berechnung vermieden werden.

b) Um das Problem des absoluten Epsilon-Wertes zu umgehen, kann man diesen als Anteil einer der Zahlen definieren. Zum Beispiel kann man vergleichen, ob der Betrag der Differenz zweier Zahlen größer ist als ein festgelegter Anteil der größeren beider Zahlen:

```
Math.abs(a - b) <= Math.max(Math.abs(a), Math.abs(b)) * eps;
```

c) Problematisch ist es hier, einen geeigneten Wert für eps zu finden.

Eine von verschiedenen Informatikern vorgeschlagene Methode ist es, die genaue Anzahl der möglichen floats zwischen den Werten zu ermitteln. Dieses Vielfache des ULP[1]-Wertes (Unit in the Last Place) kann durch die Differenz der Integer-Repräsentation der Float-Werte gefunden werden. Mehr Informationen zu diesem Verfahren und einen Algorithmus in C findet man zum Beispiel hier[2].

Hat man die Anzahl der ULPs ermittelt, kann man aufgrund der vorher durchgeführten Berechnungen den maximal möglichen Rundungsfehler abschätzen, und eine maximale Abweichung festlegen. Ein ULP bedeutet, dass die Float-Werte direkt aneinander angrenzen, also eine minimale Differenz haben.

2. a) Die Zeichensequenz mit Steuerzeichen ist folgende:

```
D i e (CR) (LF) (SP) L ö s u n g (CR) (LF) (SP) (SP) d e r
(CR) (LF) (SP) (SP) (SP) Ü b u n g s a u f g a b e (CR) (LF)
(SP) (SP) (SP) (SP) l i e g t (CR) (LF) (SP) (SP) (SP) (SP)
(SP) v o r (CR) (LF) (SP) (SP) (SP) (SP) (SP) (SP) I h n e n !
```

Die Steuerzeichen interpretiert ergibt folgenden Text:

```
Die
 Lösung
  der
   Übungsaufgabe
    liegt
     vor
      Ihnen!
```

[1] http://en.wikipedia.org/wiki/Unit_in_the_last_place
[2] http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

b) Da für Zeilenumbrüche anscheinend die Kombination aus Carriage-Return (CR) und Line-Feed (LF) verwendet wurde, ist anzunehmen, dass der Text von einem der folgenden Systeme stammt: Windows, DOS, OS/2, CP/M, TOS (Atari). Das heute naheliegendste ist also, ein Windows-System zu vermuten.

3. Die Anzahl der Sonderzeichen lässt sich berechnen als:

$$(0,56\% + 0,29\% + 0,62\% + 0,31\%) \cdot 800\,000 = 1.78\% \cdot 800\,000 = 14\,240$$

Damit ergibt sich eine Anzahl übriger Zeichen von $800\,000 - 14\,240 = 785\,760$.

a) In ASCII (ISO-8895-1) sind sowohl normale Buchstaben (a-z) als auch deutsche Umlaute im 8 Bit Zeichensatz enthalten, damit ergibt sich:

$$S_{\text{ASCII}} = 800\,000 \cdot 8 \text{ bit} = 800\,000 \text{ Byte} = 800 \text{ KB}$$

In direkter Unicode-Codierung wird jedes enthaltene Zeichen mit 16 bit codiert, daher ergibt sich:

$$S_{\text{UTF-16}} = 800\,000 \cdot 16 \text{ bit} = 1\,600\,000 \text{ Byte} = 1.6 \text{ MB}$$

In UTF-8 wird der Standard-Zeichensatz von ASCII (00 bis 7F) mit 8 bit codiert, Umlaute und Sonderzeichen werden durch Kombination mehrerer 8-bit-Blöcke dargestellt. Deutsche Umlaute werden aus 2 solcher Blöcke zusammengesetzt, damit ergibt sich:

$$\begin{aligned} S_{\text{UTF-8}} &= 785\,760 \cdot 8 \text{ bit} + 14\,240 \cdot 16 \text{ bit} = (6\,286\,080 + 227\,840) \text{ bit} \\ &= 6\,513\,920 \text{ bit} = 814\,240 \text{ Byte} \approx 814 \text{ KB} \end{aligned}$$

b)

$$\begin{aligned} n &= (4\text{DBF}_{16} - 3400_{16} + 1) + (9\text{FCF}_{16} - 4\text{E}00_{16} + 1) \\ &= (19903 - 13312 + 1) + (40911 - 19968 + 1) \\ &= 6592 + 20944 \\ &= 27536 \end{aligned}$$

c) Bei direkter UTF-16-Kodierung werden genau wie im deutschsprachigen Text 1.6 MB benötigt, da jedes Zeichen mit 16 bit kodiert wird. In UTF-8 werden jedoch 24 bit je Zeichen benötigt, daher belegt der Text in UTF-8 2.4 MB.

4. a) $y = (x \ll 3) + x$

b) $y = (x \ll 5) - (x \ll 1)$

c) $y = -(x \ll 5) - (x \ll 4)$

d) $y = (x \ll 6) - (x \ll 2) + 360$

5. a) Wahrheitstafel:

x	y	NOR	$\sim x$	$\sim y$	$(\sim x)\&(\sim y)$
0	0	1	1	1	1
0	1	0	1	0	0
! 1	0	0	0	1	0
1	1	0	0	0	0

```
int bitNor(int x, int y) {
    return (~x) & (~y);
}
```

b) Wahrheitstafel:

x	y	XOR	AND	!AND	NOR	(!AND) & NOR
0	0	0	0	1	0	0
0	1	1	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	0	0	0

```
int bitXor(int x, int y) {
    return ~(x & y) & ~((~x) & (~y));
}
```

c) Zunächst wird x um n Stellen nach rechts verschoben. Von links wird dabei mit Nullen aufgefüllt.

Danach schiebt man x um $32 - n$ Stellen nach links. Somit hat man die rechts „herausgeschobenen“ bits nun an den Stellen liegen, die im ersten Schritt mit Nullen befüllt wurden.

Beide Ergebnisse kombiniert man nun mit einem bitweisen Oder. Damit werden die Einsen der beiden Fragmente übernommen.

```
int rotateRight(int x, int n) {
    return (x >>> n) | (x << (32 + (~n + 1)));
}
```

Anmerkung: Ein Umweg musste gewählt werden, da $(32 - n)$ aufgrund des Subtraktionsoperators nicht zulässig ist. Daher wurde mit dem 2er-Komeplement von n addiert:

$$32 - n = 32 + (\sim\ n + 1)$$

d) Ziel ist es, für negative Werte von x das 2er-Komplement zu bilden, für positive Werte nichts zu verändern.

Zunächst wird der Wert für $x \gg 31$ ermittelt. Dieser Ausdruck kann zwei verschiedene Werte annehmen. Für positive x ist das MSB 0, daher wird er nach 31-maliger Rechtsverschiebung mit Erhalt des Vorzeichen-Bits (MSB) 00000...000 sein, für negative x entsprechend 11111....111.

Als nächstes werden mit Hilfe dieses Wertes negative x invertiert (x ^ 11111..111 = ~x). Positive x werden hiervon nicht beeinflusst (x ^ 00000...000 = x). Die bitweise Negation der negativen x-Werte ist der 1. Schritt in der Bildung des 2er-Komplementes.

Danach wird der Hilfswert subtrahiert. Dies wird mit Hilfe einer Addition des 2er-Komplementes des Hilfswertes erreicht, welches sich durch die Summe aus 1 und der bitweisen Negation errechnen lässt. Für positive x gilt hierbei, dass 0 subtrahiert wird, was das x weiterhin bestehen lässt. Für negative x ist der Hilfswert die Repräsentation von -1, daher ist eine Subtraktion hiervon gleich einer Addition von 1, was der 2. Schritt in der Bildung des 2er-Komplementes ist.

Dieser Algorithmus lässt sich in Java wie folgt umsetzen:

```
int abs(int x) {
    return  (x ^ (x >> 31)) + ~(x >> 31) + 1;
}
```

Anmerkung: Da der Zahlenbereich für 32-bit Integers in 2er-Komplement-Darstellung von -2147483648 bis $+2147483647$ geht, gibt es für -2147483648 keinen darstellbaren Betrag. Daher ist es unmöglich hier das mathematisch korrekte Ergebnis 2147483648 zu erhalten. Stattdessen gibt der Algorithmus den Eingabewert zurück.

Castet man den double-Wert von Math.abs daher nach Integer, erhält man ebenfalls -2147483648 als Ergebnis. Damit ist die oben beschriebene Implementation für Integer-Zahlen äquivalent zur Standard-Implementation.