MOwNiT2 Lab1

Zadanie 1 Porównać w języku Julia/C++ reprezentację bitową liczby 1/3 dla Float32, Float64 oraz liczby, która jest inicjalizowana jako Float32, a potem rzutowana na Float64. W przypadku C++ należy odpowiednio zmodyfikować nazwy typów zmiennych oraz wykonać stosowne rzutowania.

Kod w języku Julia:

```julia
decode(x::Float32) = (b=bitstring(x); (b[1], b[2:9], b[10:32]))
decode(x::Float64) = (b=bitstring(x); (b[1], b[2:12], b[13:64]))

a = Float32(1/3)
b = Float64(1/3)
c = Float64(a)

println("liczba\t\t\tznak\tcecha\t\tmantysa")
println(a,"\t\t",decode(a)[1],"\t",decode(a)[2],"\t",decode(a)[3])
println(b,"\t",decode(b)[1],"\t",decode(b)[2],"\t",decode(b)[3])
println(c,"\t",decode(c)[1],"\t",decode(c)[2],"\t",decode(c)[3])
```

Output:

```
liczba                  znak    cecha           mantysa
0.33333334              0       01111101        01010101010101010101011
0.3333333333333333      0       01111111101     0101010101010101010101010101010101010101010101010101
0.3333333432674408      0       01111111101     0101010101010101010101100000000000000000000000000000
```

Wnioski:

- Liczby a(Float32) i b(Float64) są zakodowane odpowiednio dla swoich typów. Liczba c(Float64), będąc inicjalizowaną jako Float32, posiada cechę odpowiednią dla typu Float64, lecz bity mantysy dodane po rzutowaniu zostały ustawione na 0.
- Przekłada się to na różnicę między wartością b i c.

Zadanie 2 Napisać program w języku C++, który oblicza kolejne wyrazy dowolnego nietrywialnego ciągu. Wykonać to zadanie dla reprezentacji float oraz double. Wykonać program na różnych maszynach. Objaśnić wyniki oraz fakt, że są różne.

Kod w języku Julia:

```
a = Float32(1.01) # float
b = Float64(1.01) # double
println("a\t\t\tb\n",a,"\t\t\t",b)
for i=1:25
    a = a*3 + 0.01
    b = b*3 + 0.01
    println(a,"\t",b)
end
```

Output dla pierwszych 10 przejść pętli:

```
a                       b
1.01                    1.01
3.0399999713897703      3.04
9.12999991416931        9.13
27.399999742507934      27.400000000000002
82.20999922752381       82.21000000000001
246.63999768257142      246.64000000000001
739.9299930477142       739.9300000000001
2219.799979143143       2219.8
6659.409937429429       6659.410000000001
19978.239812288284      19978.24
```

Wnioski:

- W arytmetyce komputerowej niemożliwe jest zapisanie liczby rzeczywistej w dokładnej postaci. Ze względu na ograniczoną dokładność liczb zmiennoprzecinkowych wartości podczas obliczeń są zaokrąglane i pozornie proste obliczenia mogą powodować niedokładność wyników.

Zadanie 3 Jedną z bibliotek numerycznych, jaką będziemy używać na zajęciach jest GSL (język C/C++). Korzystając ze wsparcia dla wyświetlania reprezentacji liczb zmiennoprzecinkowych zobaczyć jak zmienia się cecha i mantysa dla coraz mniejszych liczb. Zaobserwować, kiedy mantysa przestaje być znormalizowana i dlaczego?

Kod w języku C:

```c
#include <stdio.h>
#include <gsl/gsl_ieee_utils.h>

int main(int argc, char **argv){

   float f = 1.0;
   double d = 1.0;

   for(int i=0;i<400;i++){

       printf("Przejscie %d:\nf = ",i+1);
       gsl_ieee_printf_float(&f);
       printf("\nd = ");
       gsl_ieee_printf_double(&d);
       printf("\n");
       f = f / 10.0;
       d = d / 10.0;
   }
   return 0;
}
```

(Output na kolejnej stronie)

Wnioski:

- Wraz z dzieleniem liczby zmniejsza się jej mantysa. Ze względu na dzielenie przez 10 zmienia się również cecha (dla potęg 2 nie pozostawałaby taka sama).
- Po przekroczeniu najmniejszej możliwej do zakodowania liczby znormalizowanej liczby zostają zapisane w postaci zdenormalizowanej, odpowiednio po 38 i 308 dzieleniach przez 10 dla typów float i double.
- Gdy przekroczymy wartość najmniejszej zdenormalizowanej liczby danego typu zostaje ona zaokrąglona do zera, odpowiednio po 46 i 324 dzieleniach przez 10 dla typów float i double.

Output:

```
Przejscie 1:
f =  1.00000000000000000000000*2^0
d =  1.0000000000000000000000000000000000000000000000000000*2^0
Przejscie 2:
f =  1.10011001100110011001101*2^-4
d =  1.1001100110011001100110011001100110011001100110011010*2^-4
Przejscie 3:
f =  1.01000111101011100001010*2^-7
d =  1.0100011110101110000101000111101011100001010001111011*2^-7
Przejscie 4:
f =  1.00000110001001001101110*2^-10
d =  1.0000011000100100110111010010111100011010100111111100*2^-10
[...]
Przejscie 38:
f =  1.00010000001110011101010*2^-123
d =  1.0001000000111001110101000010100010101000101110010001*2^-123
Przejscie 39:
f =  0.11011001110001111101110*2^-126
d =  1.1011001110001111101110011101101010100111100011101000*2^-127
Przejscie 40:
f =  0.00010101110001110011000*2^-126
d =  1.0101110001110010111110110001010101010010110110000110*2^-130
[...]
Przejscie 46:
f =  0.00000000000000000000001*2^-126
d =  1.0110110101100000000110101101001101110110101010111011*2^-150
Przejscie 47:
f =  0
d =  1.0010010001001100111000100100001011000101010101100010*2^-153
[...]
Przejscie 308:
f =  0
d =  1.0001111110100001100000101100010000000110001100011000*2^-1020
Przejscie 309:
f =  0
d =  0.0111001100001101011001111000000110011110100011010110*2^-1022
Przejscie 310:
f =  0
d =  0.0000101110000001010101110010011010001111110110101111*2^-1022
[...]
Przejscie 324:
f =  0
d =  0.0000000000000000000000000000000000000000000000000010*2^-1022
Przejscie 325:
f =  0
d =  0
```

Zadanie 4 Wymyślić własny przykład algorytmu niestabilnego numerycznie.

1. Zademonstrować wersję niestabilną, pokazać, że działa źle.
2. Wyjaśnić, dlaczego działa źle.
3. Zademonstrować wersję stabilną.

Algorytmy obliczają rozwiązania równania kwadratowego.

Jego niestabilna wersja liczy w sposób “szkolny” za pomocą delty: $\Delta = b^2 - 4ac$, zaś

wersja stabilna korzysta z jednego ze wzorów Viete’a: $x_1 x_2 = c/a$.

Algorytm niestabilny:

```c
struct ans unstable_solve(double a,
double b, double c){
   double delta = b*b - 4*a*c;
   struct ans answer;
   if(delta < 0){
       answer.x1 = HUGE_VAL;
       answer.x2 = HUGE_VAL;
       return answer;     // inf on
error
   }
   delta = sqrt(delta);
   answer.x1 = (-b+delta)/(2*a);
   answer.x2 = (-b-delta)/(2*a);
   return answer;
   // same values if only 1 answer
}
```

Algorytm stabilny:

```c
struct ans stable_solve(double a,
double b, double c){
   double delta = b*b - 4*a*c;
   struct ans answer;
   if(delta < 0){
       answer.x1 = HUGE_VAL;
       answer.x2 = HUGE_VAL;
       return answer; // inf on error
   }
   delta = sqrt(delta);
   if(b<0){
       answer.x1 = (-b+delta)/(2*a);
       answer.x2 = c/a/answer.x1;
   }
   else{
       answer.x2 = (-b-delta)/(2*a);
       answer.x1 = c/a/answer.x2;
   }
   return answer;
}
```

Kod wywołania:

```c
int main(int argc, char **argv){
   double a = 0.00001;
   double b = -2000;
   double c = 0.00001;

   printf("a=%f, b=%f, c=%f\n\n",a,b,c);
   struct ans unstable_sol = unstable_solve(a,b,c);
   printf("UNSTABLE:\n%0.18f, %0.18f\n",unstable_sol.x1,unstable_sol.x2);
   struct ans stable_sol = stable_solve(a,b,c);
   printf("STABLE:\n%0.18f, %0.18f\n",stable_sol.x1,stable_sol.x2);
   return 0;
}
```

Output:

```
a=0.000010, b=-2000.000000, c=0.000010

UNSTABLE:
199999999.999999970197677612, 0.000000011368683772
STABLE:
199999999.999999970197677612, 0.000000005000000000
```

Oczekiwane rozwiązania dla danych wartości (Wolfram Alpha):

$x_1 = 1.9999999999999999500 * 10^8$

$x_2 = 5.0000000000000001250 * 10^{-9}$

Wnioski:

- Pierwszy algorytm działa źle, ponieważ dla dużej wartości wyrażenia b^2 i małej wartości $4ac$ zachodzi $b \approx \sqrt{\Delta}$. Powoduje to w dalszym kroku odejmowanie bliskich sobie liczb, gdzie tracimy na dokładności.
- Drugi algorytm unika tego odejmowania, pierwsze rozwiązanie jest liczone poprzez dodawanie, a drugie z podanego wzoru Viete'a.
- Dla danych wartości błąd względny osiągnął 127% !