Dlaczego boję się, gdy komputer liczy

Piotr Fulmański*

16 grudnia 2011

Spis treści

*E-mail: fulmanp@math.uni.lodz.pl

1 Dlaczego 2+2=5?

I am a HAL Nine Thousand computer
[...]
The quick brown fox jumps over the lazy
dog [...]
Dave – are you still there?
Did you know that the square root of 10
is 3.162277660168379?
Log 10 to the base e is
0.434294481903252... correction, that is
log e to the base 10 [...]
2 times 2 is... 2 times 2
is... approximately 4.10101010... I seem
to be having difficulty
HAL, 2001: A Space Odyssey

Ograniczenia jakie niesie ze sobą format IEEE 754 zapisu liczby zmiennoprzecinkowej są widoczne i oczywiste po uważniejszym jemu przyjrzeniu się. Zasadniczym motywem zbadania jego był wręcz katastrofalne wyniki jakie uzyskałem w przypadku kilku wręcz elementarnych operacjach arytmetycznych. I nie jest ważne jakiego to dotyczyło języka programowania, bo w każdym będzie tak samo źle.

1.1 Porażka 1

```
critical=0;
for(i=0;;i++,critical++)
{
    System.out.println("dla i="+i+": "+i+"*0.01="+(i*0.01)+",
       (0.8-"+i+"*0.01)="+(0.8-i*0.01));
    if(0.8-i*0.01==0.72)
        break;
    if(critical==20)
        break;
}
System.out.println("critical="+critical);
```

Działanie programu można opisać w następujący sposób. W pętli która jest pętlą nieskończoną odejmuj od wartości 0.8 kolejne wielokrotności 0.01. W momencie, gdy wynik odejmowania będzie równy 0.72 (co naszym zdaniem, powinno niewątpliwie nastąpić w 9-tej iteracji[1] przerywamy pętlę nieskończoną. Zmienna `critical` powinna zatem mieć wartość 8. Zmienną tą dodaliśmy przewidując, że rozwój wypadków nie musi być zgodny z naszymi

[1] W 9-tej, bo zaczynamy od 0.

oczekiwaniami. Ma ona zabezpieczyć nas przed nieskończonym wykonywaniem programu, gdyż osiagnięcie przez nią wartości równej 20 (co nastąpi w 21 iteracji) powoduje przerwanie pętli nieskończonej.

Wynik jaki otrzymujemy wygląda następująco

```
dla i=0: 0*0.01=0.0, (0.8-0*0.01)=0.8
dla i=1: 1*0.01=0.01, (0.8-1*0.01)=0.79
dla i=2: 2*0.01=0.02, (0.8-2*0.01)=0.78
dla i=3: 3*0.01=0.03, (0.8-3*0.01)=0.77
dla i=4: 4*0.01=0.04, (0.8-4*0.01)=0.76
dla i=5: 5*0.01=0.05, (0.8-5*0.01)=0.75
dla i=6: 6*0.01=0.06, (0.8-6*0.01)=0.74
dla i=7: 7*0.01=0.07, (0.8-7*0.01)=0.73
dla i=8: 8*0.01=0.08, (0.8-8*0.01)=0.7200000000000001
dla i=9: 9*0.01=0.09, (0.8-9*0.01)=0.7100000000000001
dla i=10: 10*0.01=0.1, (0.8-10*0.01)=0.7000000000000001
dla i=11: 11*0.01=0.11, (0.8-11*0.01)=0.6900000000000001
dla i=12: 12*0.01=0.12, (0.8-12*0.01)=0.68
dla i=13: 13*0.01=0.13, (0.8-13*0.01)=0.67
dla i=14: 14*0.01=0.14, (0.8-14*0.01)=0.66
dla i=15: 15*0.01=0.15, (0.8-15*0.01)=0.65
dla i=16: 16*0.01=0.16, (0.8-16*0.01)=0.64
dla i=17: 17*0.01=0.17, (0.8-17*0.01)=0.63
dla i=18: 18*0.01=0.18, (0.8-18*0.01)=0.6200000000000001
dla i=19: 19*0.01=0.19, (0.8-19*0.01)=0.6100000000000001
dla i=20: 20*0.01=0.2, (0.8-20*0.01)=0.6000000000000001
critical=20
```

Mówiąc krótko, według komputera wynik poniższego działania

$$0.8 - 8 \cdot 0.01$$

nie jest równy 0.72.

1.2 Porażka 2

```
suma=0;
for(i=1;i<=10;i++)
{
   suma+=0.1;
   System.out.println("suma dla i="+i+" wynosi "+suma);
}

if(suma==1.0)
   System.out.println("Równe");
```

```
if(suma>1.0)
   System.out.println("Suma > 1.0");
if(suma<1.0)
   System.out.println("Suma < 1.0");
```

Działanie programu można opisać w następujący sposób. W pętli która jest pętlą skończoną, gdyż wykona się dokładnie 10 razy powiększam wartość zmiennej `suma` o 0.1. Jako, że operacja ta zostanie wykonana 10 razy, więc mam pewne podstawy aby sądzić, że po zakończeniu pętli wartość zmiennej `suma` będzie równa 1. Jak łatwo się domyślać, tak oczywiście nie będzie co pokazuje wynik działania programu

```
suma dla i=1 wynosi 0.1
suma dla i=2 wynosi 0.2
suma dla i=3 wynosi 0.30000000000000004
suma dla i=4 wynosi 0.4
suma dla i=5 wynosi 0.5
suma dla i=6 wynosi 0.6
suma dla i=7 wynosi 0.7
suma dla i=8 wynosi 0.7999999999999999
suma dla i=9 wynosi 0.8999999999999999
suma dla i=10 wynosi 0.9999999999999999
Suma < 1.0
```

1.3 Porażka 3

```
if(0.1+0.1+0.1==0.3)
    System.out.println("Równe");
else
    System.out.println("Błąd");
```

Działanie tego prościutkiego fragmentu można bez problemu przewidzieć na podstawie dwóch wcześniejszych porażek. Także i w tym przypadku sprzeczne jest ono z naszym, potocznym, pojmowaniem rzeczywistosci.

```
Błąd
```

1.4 Porażka 4

Powiedzą Państwo, że się uwziąłem. Wybrałem dwie liczby, które nie posiadają skończonego rozwinięcia dwójkowego

```
0.1 | 2 * 0.1 = 0.2 + 0      0.3 | 2 * 0.3 = 0.6 + 0
0.2 | 2 * 0.2 = 0.4 + 0      0.6 | 2 * 0.6 = 0.2 + 1
0.4 | 2 * 0.4 = 0.8 + 0      0.2 | 2 * 0.2 = 0.4 + 0
0.8 | 2 * 0.8 = 0.6 + 1      0.4 | 2 * 0.4 = 0.8 + 0
```

```
0.6 | 2 * 0.6 = 0.2 + 1      0.8 | 2 * 0.8 = 0.6 + 1
0.2 | 2 * 0.2 = 0.4 + 0      0.6 | 2 * 0.6 = 0.2 + 1
...                      ...

0.1 = 0.000110...           0.3 = 0.010011...
```

więc i wyniki nie mogą być dobre. Po części jest to prawda, ale zauważmy, że dla wartości 0.2, pomimo iż także nie posiada ona skończonego rozwinięcia dwójkowego

```
0.2 | 2 * 0.2 = 0.4 + 0
0.4 | 2 * 0.4 = 0.8 + 0
0.8 | 2 * 0.8 = 0.6 + 1
0.6 | 2 * 0.6 = 0.2 + 1
0.2 | 2 * 0.2 = 0.4 + 0
...

0.2 = 0.00110...
```

wyniki otrzymujemy poprawne. Oto kod programu

```
if(0.1+0.1==0.2)
    System.out.println("Równe");
else
    System.out.println("Błąd");
```

...i wynik jego działania

```
Równe
```

A więc znowu porażka, bo nastawiliśmy się na wynik błędny a tym razem mamy poprawny.

1.5 Porażka 5

Kolejny przykład, podobnie jak poprzedni, pokazuje, że nie zamy dnia ani godziny gdy wynik będzie dobry lub błędny.

```
double d;
d = 28.0 * 0.01;
System.out.println(d);
System.out.println((int) (d * 100));

d = 29.0 * 0.01;
System.out.println(d);
System.out.println((int) (d * 100));
```

Teoretycznie w obu przypadkach powinniśmy otrzymać liczbę od której zaczynamy (a więc odpowiednio 28 i 29), ale jak łatwo zgadnąc z kontekstu niniejszego dokumentu, tak nie jest.

```
0.28
28

0.29
28
```

A więc raz prawdą jest, że

$$(x \cdot 0.01) \cdot 100 = x$$

(ma to miejsce dla $x = 28$) innym razem już nie (ma to miejsce dla $x = 29$).

2 Doświadczenie 1

Tak więc na tapetę wzięty został format IEEE 754 a dokładniej mówiac jego modyfikacja wykorzystująca 8 bitów. Format taki przedstawiony został w [1]. Przypominijmy, że przyjmujemy następujące znaczenie bitów: pierwszy bit od lewej oznacza znak liczby, kolejne 3 mantysę zaś ostatnie 4 bity będą cechą. Jako wartość stałej K_C (odejmowanej od cechy) przyjmijmy 7. Tak więc liczby dające się zapisać w tym formacie są postaci

$$z_m M \cdot 2^{C-K_C},$$

gdzie z_m to znak mantysy, M – mantysa, C – cecha. Tak przyjęty format jest dużo protszy do analizy (ze względu na mniejszą ilość bitów) a przy tym raczej nie fałszuje w sposób istotnych względnych wyników (tj. stosunku poprawnych i błędnych obliczeń). Dla ułatwienia wzięte zostały pod obserwację tylko liczby dodatnie.

Najpierw stworzony został program generujący wszystkie liczby jakie można zapisać w tym formacie.

```
#define K_C 7

[...]

//przyjmuje nastepujaca kolejnosc bitow
//b2 b1 b0 c3 c2 c1 c0
char b2,b1,b0,c3,c2,c1,c0;
int cecha,i=0,j,k,blad=0,zakres=0;
float mantysa;
double t[128];
```

```
//przebieg po cesze
for(c3=0;c3<=1;c3++)
for(c2=0;c2<=1;c2++)
for(c1=0;c1<=1;c1++)
for(c0=0;c0<=1;c0++)
{
    cecha=c0*1+c1*2+c2*4+c3*8-K_C;
   //przebieg po mantysie
   for(b2=0;b2<=1;b2++)
   for(b1=0;b1<=1;b1++)
   for(b0=0;b0<=1;b0++)
   {
      mantysa=1.0+0.5*b2+0.25*b1+0.125*b0;
      t[i]=wypisz(b2,b1,b0,c3,c2,c1,c0,mantysa,cecha);
      i++;
   }
}
```

Funkcja `Wypisz` zdefiniowana została jak poniżej

```
double Wypisz(char b2,char b1,char b0,char c3,char c2,char c1,
              char c0,float mantysa,int cecha)
{
   double r;
   fprintf(stdout,"%c",b2==0?'0':'1');
   fprintf(stdout,"%c",b1==0?'0':'1');
   fprintf(stdout,"%c",b0==0?'0':'1');
   fprintf(stdout,"%c",c3==0?'0':'1');
   fprintf(stdout,"%c",c2==0?'0':'1');
   fprintf(stdout,"%c",c1==0?'0':'1');
   fprintf(stdout,"%c",c0==0?'0':'1');
   r=mantysa*pow(2,cecha);
   fprintf(stdout," : %f*2^%d=%.10f",mantysa,cecha,r);
   fprintf(stdout,"\n");
   return r;
}
```

I tak, zgodnie z oczekiwaniami najmniejszą możliwą do reprezentowania liczbą jest 0.0078125 zapisane jako `0000000` ($1.000000 \cdot 2^{-7}$), zaś największa to 480.0 zapisane jako `1111111` ($1.875000 \cdot 2^{8}$); łącznie mamy oczywiście 128 różnych liczb.

Teraz przyszedł czas na najważniejszą część tego doświadczenia. Biorąc oto wszystkie możliwe kombinacje dwóch spośród otrzymanych 128 liczb, sprawdzamy czy ich suma jest jedną z tych 128 liczb. Mówiąc inaczej sprawdzamy czy suma dwóch spośród owych 128 liczb daje się wyrazić w tym formacie.

```
//szukam zlych sum
for(i=0;i<128;i++)
for(j=0;j<128;j++)
{
   for(k=0;k<128;k++)
   {
      if(t[i]+t[j]==t[k])
      {
        k=255;
        break;
      }
   }
   if(k!=255)
   {
      if(t[i]+t[j]<=t[127])
      {
         blad++;
         fprintf(stdout,"%.10f+%.10f=%.10f",t[i],t[j],
                                            t[i]+t[j]);
         fprintf(stdout," ERROR\n");
      }
      else
      {
         zakres++;
         fprintf(stdout,"%.10f+%.10f=%.10f",t[i],t[j],
                                            t[i]+t[j]);
         fprintf(stdout," ZAKRES\n");
      }
   }
}
fprintf(stdout,"Lacznie bledow %d, poza zakresem %d (%d)\n",
        blad,zakres,128*128);
```

Zauważmy, że łacznie mamy $128 \cdot 128 = 16384$ możliwości. Wyniki nie są optymistyczne. Otóż 434 sumy są poza zakresem obejmowanym przez tak przyjęty format (np. $480+480$ daje 960). Trudno o to mieć jednak pretensje – ograniczona ilość bitów skutkuje ograniczonym zakresem. Pozostaje więc 15950 sum. Z tego 14232 nie dają się wyrazić w tym formacie. Oznacza to, że jedynie 1718 sum jest prawidłowych. Reszta to będą przybliżenia. Zatem jedynie 9.28% wyników jest poprawna!!!

Gdyby opuścić wszystkie sumy, które się powtarzają, czyli nie rozróżniać czy dodajemy x_1 do x_2 czy na odwrót: x_2 do x_1, wówczas mamy 8256 różnych sum, z których 221 jest poza zakresem naszego formatu a 7121 nie daje się wyrazić w tym formacie. Oznacza to, że jedynie 914 sum jest

prawidłowych, czyli 11.37%.

Już samo to co napisano powyżej wystarcza do tego aby przenieść się z prowadzeniem obliczeń na liczydła. Zróbmy jednak jeszcze jeden test. Bierzmy więc dwie kolejne liczby i liczmy ich różnicę. Otrzymamy w ten sposób wartość mówiącą nam o dokładności reprezentacji. Wyniki zebrano w tabeli 1. Dla czytelności pominięte zostały różnice dające takie same wyniki. I tak patrząc na wiersz o liczbie porządkowej 127 mamy tam różnicę (dwóch kolejnych) liczb 480 i 448 wynoszącą 32. Oznacza to, że **żadna** liczba całkowita (nie mówiąc już o rzeczywistych) większa od 448 a mniejsza od 480 nie daje się wyrazić w przyjętym formacie. Każda z takich liczb będzie zaokrąglona „w górę" do 480 lub „w dół" do 448 w zależności od przyjętych algorytmów.

3 Doświadczenie 2

Z pewnością wszystkim doskonale znany jest przypadek Lorenza i jego motyla opisany w „*Can the flap of a butterfly wing stir up a tornado in Texas*". Problem dotyczy powstawania i akumulacji błędów zaokrągleń i związany jest z niemożnością przewidywania w deterministycznych układach ze sprzężeniem zwrotnym w tym także w matematycznych modelach, które wykorzystywano do długoterminowych analiz pogody. Jak to zwykle bywa na problem ten Lorentz natrafił przez przypadek.

Zaczęło się to wszystko gdzieś około roku 1956, kiedy pewne metody przewidywania [pogody] zostały zaproponowane [. . .] jako najlepsze z dostępnych, z czym się jednak nie zgadzałem. Zadecydowałem sam wysmażyć niewielki układ równań symulujących zachowanie atmosfery, rozwiązać go za pomocą komputerów, które wtedy właśnie zaczęły być dostępne, a następnie potraktować wyniki tak, jakby to były rzeczywiste dane z obserwacji atmosferycznych, i sprawdzić, czy proponowane metody stosują się do nich. Prawdziwym problemem było uzyskanie takiego układu równań, który doprowadziłby do wyników nadających się do przetestowania, ponieważ szybko stało się jasne, że jeśli rozwiązanie tych równań będzie cykliczne, to proponowane metody będą trywialne, a tym samym będą stosowały się idealnie.

Musieliśmy więc otrzymać układ równań, mający rozwiązania niecykliczne, takie które nie powtarzają się, ale przebiegają nieregularnie i w sposób niezdefiniowany. Znalazłem w końcu układ dwunastu równań, które to spełniały i sprawdziłem, że proponowane metody nie były odpowiednie. Gdy to robiłem chciałem sprawdzić niektóre z wyników w sposób bardziej szczegółowy. W swoim biurze miałem wtedy mały komputer, wpisałem więc kilka pośrednich wartości, które komputer wydrukował, jako nowe warunki początkowe dla następnych obliczeń, i na chwile wyszedłem. Kiedy wróciłem, zobaczyłem, że rozwiązania były inne niż przedtem; komputer zachowywał się inaczej. W pierwszej chwili podejrzewałem, że mam jakieś kłopoty z komputerem,

L.p.	x	y	$y-x$
1	0.0078125000	0.0087890625	0.0009765625
8	0.0146484375	0.0156250000	0.0009765625
9	0.0156250000	0.0175781250	0.0019531250
16	0.0292968750	0.0312500000	0.0019531250
17	0.0312500000	0.0351562500	0.0039062500
24	0.0585937500	0.0625000000	0.0039062500
25	0.0625000000	0.0703125000	0.0078125000
32	0.1171875000	0.1250000000	0.0078125000
33	0.1250000000	0.1406250000	0.0156250000
40	0.2343750000	0.2500000000	0.0156250000
41	0.2500000000	0.2812500000	0.0312500000
48	0.4687500000	0.5000000000	0.0312500000
49	0.5000000000	0.5625000000	0.0625000000
56	0.9375000000	1.0000000000	0.0625000000
57	1.0000000000	1.1250000000	0.1250000000
64	1.8750000000	2.0000000000	0.1250000000
65	2.0000000000	2.2500000000	0.2500000000
72	3.7500000000	4.0000000000	0.2500000000
73	4.0000000000	4.5000000000	0.5000000000
80	7.5000000000	8.0000000000	0.5000000000
81	8.0000000000	9.0000000000	1.0000000000
88	15.0000000000	16.0000000000	1.0000000000
89	16.0000000000	18.0000000000	2.0000000000
96	30.0000000000	32.0000000000	2.0000000000
97	32.0000000000	36.0000000000	4.0000000000
104	60.0000000000	64.0000000000	4.0000000000
105	64.0000000000	72.0000000000	8.0000000000
112	120.0000000000	128.0000000000	8.0000000000
113	128.0000000000	144.0000000000	16.0000000000
120	240.0000000000	256.0000000000	16.0000000000
121	256.0000000000	288.0000000000	32.0000000000
127	448.0000000000	480.0000000000	32.0000000000

Tablica 1. Różnice pomiędzy kolejnymi wartościami

jednak wkrótce odkryłem, że prawdziwą przyczyną jest to, iż liczby wpisane przeze mnie różniły się od liczb wyjściowych, które zaokrągliłem, i ta niewielka różnica pomiędzy czymś rozwiniętym do szóstego miejsca po przecinku i jego zaokrągleniem do trzeciego miejsca w czasie symulacji dwóch miesięcy pogody stała się tak wielka jak sam sygnał. Wynikało z tego, że jeżeli prawdziwa atmosfera zachowuje się w podobny sposób, to nie jesteśmy po prostu w stanie przewidzieć pogody na dwa miesiące naprzód. Te małe błędy będą się powiększać, aż staną się wielkie [2].

Postanowiłem więc sprawdzić jak to faktczynie z tym motylem jest. Jako temat dalszych rozważań wybrałem tak zwany *model logistyczny* a więc rekurencyjne wyrażenie następującej postaci

$$p_{n+1} = p_n + rp_n(1 - p_n) \tag{1}$$

oraz jego „drugą" postać otrzymaną przez zastosowanie elementarnych przekształceń algebraicznych (tak więc z matematycznego punktu widzenia oba te wyrażenia, jeśli nawet nie są takie same, to dają takie same wyniki)

$$p_{n+1} = (1 + r)p_n - rp_n^2. \tag{2}$$

W powyższych równaniach r jest pewną stałą, natomiast n i $n+1$ to indeksy odpowiednio poprzedniego i nowo obliczanego wyrazu. Jako wartość stałej r przyjęto 3.0 natomiast jako wyraz a_1 przyjęto wartość 0.01.

Dla obu wzorów dokonano po 500-set iteracji na dwóch różnych typach liczbowych: float i double. Kilka przykładowych wyników przedstawiono w tabeli 2. Zamieszczone w niej dane należy czytać w następujący sposób. W ramach każdej iteracji mamy trzy wiersze: pierwszy to wynik obliczeń przeprowadzonych na zmiennych typu float, drugi na zmiennych typu double trzeci zaś to różnica pomiędzy tymi wynikami. Tak więc porównanie wiersza pierwszego i drugiego daje nam pojęcie o wpływie przyjętej reprezentacji na prowadzone obliczenia (w ramach tego samego wzoru). Jeśli natomiast chcemy porównać ze sobą wzory, to interesuje nas pierwsza kolumna od prawej, w której to zawarto różnice pomiędzy wartościami otrzymanym za pomocą obu wzorów. I tak na przykład w iteracji 399 mamy, że dla wzoru 1 różnica pomiędzy wartościami obliczonymi przy użyciu zmiennych typu float i double wynosi -0.442180476115175. Ta sama różnica dla wzoru 2 wynosi 0.848301242880773. Zauważmy, że różnice są tego samego rzędu co obliczane wartości! Jeśli natomiast chodzi o różnice pomiędzy wzorami to dla typu double wynosi ona 1.290481718995948. Zastanawiający może być fakt braku różnic dla typu float. tutu W tym momencie nie mam wiarygodnego wyjaśnienia, choć mam kilka hipotez :) Oba wzory dają takie same wyniki do 257 iteracji[3].

[2] W: [4], s. 79.

[3] Mówiąc dokładniej: wyniki są takie same biorąc pod uwagę 15 miejsc po przecinku.

Iteracja	Według wzoru 1	Według wzoru 2	Różnice pomiędzy wzorami
1	0.009999999776483	0.009999999776483	0.000000000000000
	0.010000000000000	0.010000000000000	0.000000000000000
	-0.000000000223517	-0.000000000223517	0.000000000000000
257	0.070538848638535	0.070538848638535	0.000000000000000
	1.189384903904431	1.189384903904431	0.000000000000000
	-1.118846055265897	-1.118846055265896	-0.000000000000000
258	0.267228215932846	0.267228215932846	0.000000000000000
	0.513630266710465	0.513630266710467	-0.000000000000001
	-0.246402050777619	-0.246402050777621	0.000000000000001
300	0.099408328533173	0.099408328533173	0.000000000000000
	0.967600411062598	0.961800737412491	0.005799673650107
	-0.868192082529425	-0.862392408879318	-0.005799673650107
399	0.856545627117157	0.856545627117157	0.000000000000000
	1.298726103232332	0.008244384236384	1.290481718995948
	-0.442180476115175	0.848301242880773	-1.290481718995948
500	0.260175228118896	0.260175228118896	0.000000000000000
	0.940545426832992	0.257679774120511	0.682865652712481
	-0.680370198714096	0.002495453998385	-0.682865652712481

Tablica 2. Różnice pomiędzy kolejnymi wyrazami otrzymanymi za pomocą wzorów 1 oraz 2

Na zakończenie jeszcze jeden test – test wrażliwościowy. Sprawdzimy jaki wpływ na wyniki obliczeń ma zakłócenie warunków początkowych. To całkiem tak jak przy pogodzie – zrobiliśmy pomiar temperatury powietrza i „trochę" się pomyliliśmy. Tabela 3 prezentuje część wyników. Pierwszy wiersz dla każdej iteracji policzony został dla współczynnika $r = 3.0$, drugi dla $r = 3.000000000001$, trzeci dla $r = 3.0000000000001$, czwarty dla $r = 3.00000000000001$. Zmniejszenie części ułamkowej o następny rząd wielkości, dawało dobre wyniki przez 20000 iteracji. Powodem tego może być niemożność odwzorowania liczby 3 z tak małą częścią ułamkową i zaokrąglenie jej do 3.0 lub też błędy na dalszych, niewyświetlanych, pozycjach co jest raczej wątpliwe bo po tylu iteracjach powinny one stać się widoczne.

4 Doświadczenie 3

Czytając przebieg doświadczenia 2 opisanego w rozdziale 3 z pewnością na myśl nasuwało się pytanie: „Jak to się stało, jak to możliwe?". Szukając odpowiedzi na to pytanie przyjrzyjmy się kolejnemu przykładowi. Zanim to jednak nastąpi, krótkie wprowadzenie teoretyczne.

Reprezentacja zmiennoprzecinkowa do wyrażenia wartości posługuje się mantysą i cechą. W idealnym (nieskończonym) świecie mantysa miała by postać

$$M = 1 + \sum_{1}^{\infty} d_{-i} 2^{-i}.$$

Tak by było, ale nie jest i w praktyce mantysa z którą mamy do czynienia wygląda tak

$$M_t = 1 + \sum_{1}^{t} d_{-i} 2^{-i},$$

gdzie t jest liczbą cyfr mantysy. W konsekwencji

$$|M - M_t| \leq 2^{-t}.$$

Tak więc w komputerze zamiast wielkości z_m, C, K_C oraz M określających daną liczbę $x = z_m M \cdot 2^{C-K_C}$ zapamiętywane są wielkości z_m, C, K_C oraz M_t, które definiuja reprezentację zmiennoprzecinkową $x_F = z_m M_t \cdot 2^{C-K_C}$ liczby x. Z powyższego i założenia iż mantysa spełnia nierówność $1 \leq M < 2$ wynika następujące oszacowanie

$$\left|\frac{x_F - x}{x}\right| = \left|\frac{z_m M_t \cdot 2^{C-K_C} - z_m M \cdot 2^{C-K_C}}{z_m M \cdot 2^{C-K_C}}\right| \leq \left|\frac{M_t - M}{1}\right| \leq 2^{-t}$$

równoważne relacji

$$\left|\frac{x_F - x}{x}\right| = |\varepsilon|,$$

Iteracja	Współczynnik	Wyniki
1	3.0	0.039700000000000
	3.0+10E-12	0.039700000000010
	3.0+10E-13	0.039700000000001
	3.0+10E-14	0.039700000000000
	3.0+10E-15	0.039700000000000
2	3.0	0.154071730000000
	3.0+10E-12	0.154071730000075
	3.0+10E-13	0.154071730000008
	3.0+10E-14	0.154071730000001
	3.0+10E-15	0.154071730000000
3	3.0	0.545072626044421
	3.0+10E-12	0.545072626044783
	3.0+10E-13	0.545072626044457
	3.0+10E-14	0.545072626044425
	3.0+10E-15	0.545072626044422
4	3.0	1.288978001188801
	3.0+10E-12	1.288978001189313
	3.0+10E-13	1.288978001188852
	3.0+10E-14	1.288978001188806
	3.0+10E-15	1.288978001188801
5	3.0	0.171519142109176
	3.0+10E-12	0.171519142106890
	3.0+10E-13	0.171519142108948
	3.0+10E-14	0.171519142109153
	3.0+10E-15	0.171519142109174
6	3.0	0.597820120107100
	3.0+10E-12	0.597820120100453
	3.0+10E-13	0.597820120106437
	3.0+10E-14	0.597820120107033
	3.0+10E-15	0.597820120107094
7	3.0	1.319113792413797
	3.0+10E-12	1.319113792411292
	3.0+10E-13	1.319113792413548
	3.0+10E-14	1.319113792413772
	3.0+10E-15	1.319113792413795
51	3.0+10E	0.074892694909774
	3.0+10E-12	0.462398200088556
	3.0+10E-13	0.056394912819080
	3.0+10E-14	1.280547749402646
	3.0+10E-15	0.178875826166109

Tablica 3. Wpływ niedokładności na otrzymywane wyniki

gdzie $|\varepsilon| \leq 2^{-t}$. Z powyższego otrzymujemy, wykorzystywany dalej, związek

$$x_F = x(1+\varepsilon),$$

przy czym ogólnie zakładać będziemy, że $|\varepsilon| \leq \eta$. Dla współczesnych komputerów $\eta \in (10^{-15}, 10^{-6})$. Mogłoby się wydawać, że tak niewielkie błędy reprezentacji liczb są do zaniedbania. Poprzednio pokazaliśmy że tak jednak nie jest a teraz przejdziemy do kolejnego przykładu.

Zastanówmy się, co się stanie gdy spróbujemy obliczyć wartość wyrażenia $x^2 - y^2$ na dwa, algebraicznie równoważne, sposoby

- sposób 1: $w_1 = xx - yy$,
- sposób 2: $w_2 = (x-y)(x+y)$.

Oczywiście w arytmetyce dokładnej oba wzory są sobie równoważne – otrzymujemy identyczne wyniki. W arytmetyce zmiennoprzecinkowej otrzymujemy zaś

$$\begin{aligned} w_{1F} &= ((xx)(1+\varepsilon_1) - (yy)(1+\varepsilon_2))(1+\varepsilon_3) \\ &= (x^2-y^2)\left(1 + \tfrac{x^2\varepsilon_1 - y^2\varepsilon_2}{x^2-y^2}\right)(1+\varepsilon_3), \\ w_{2F} &= ((x-y)(1+\varepsilon_4)(x+y)(1+\varepsilon_5))(1+\varepsilon_6) \\ &= (x^2-y^2)(1+\varepsilon_4)(1+\varepsilon_5)(1+\varepsilon_6). \end{aligned}$$

Zauważmy, że dla x bliskiego y wartość w_{1F} może znacznie odbiegać od wartości dokładnej w_1. Drugi sposób we wszystkich przypadkach daje wynik z dużą dokładnością.

5 Doświadczenie 4

Załóżmy, że chcemy sprawdzić, czy punkty $p_1, p_2 \in R^3$ leżą po tej samej stronie płaszczyzny o równaniu

$$0 = ax + by + cz - d.$$

W arytmetyce dokładnej wystarczy sprawdzić czy znak wielkości

$$w_1 = ax_1 + by_1 + cz_1 - d,$$

jest taki sam jak

$$w_2 = ax_2 + by_2 + cz_2 - d.$$

Dla dowolnego punktu $p = (x, y, z)$ otrzymujemy

$$\begin{aligned} w_F &= ((((ax)(1+\varepsilon_1) + (by)(1+\varepsilon_2))(1+\varepsilon_3) \\ &\quad +(cz)(1+\varepsilon_4))(1+\varepsilon_5) - d)(1+\varepsilon_6) \\ &= ax(1+\varepsilon_1)(1+\varepsilon_3)(1+\varepsilon_5)(1+\varepsilon_6) \\ &\quad +by(1+\varepsilon_2)(1+\varepsilon_3)(1+\varepsilon_5)(1+\varepsilon_6) \\ &\quad +cz(1+\varepsilon_4)(1+\varepsilon_5)(1+\varepsilon_6) - d(1+\varepsilon_6). \end{aligned}$$

Zgodnie z rozważaniami z rozdziału 4, wszystkie błędy względne ε_i wytworzone w pojedyńczych działaniach arytmetcznych mają oszacowanie $|\varepsilon_i| \leq \eta$. Przy obecnie spotykanych rzędach wielkości dla η możemy przyjąć, że

$$(1+\varepsilon_1)(1+\varepsilon_3)(1+\varepsilon_5)(1+\varepsilon_6) \approx 1+\varepsilon_1+\varepsilon_3+\varepsilon_5+\varepsilon_6$$

i podobnie dla pozostałych składników wyrażenia w_F. W konsekwencji

$$\begin{aligned} w_F &\approx ax+by+cz-d+[ax(\varepsilon_1+\varepsilon_3+\varepsilon_5+\varepsilon_6) \\ &\quad +by(\varepsilon_2+\varepsilon_3+\varepsilon_5+\varepsilon_6)+cz(\varepsilon_4+\varepsilon_5+\varepsilon_6)-d\varepsilon_6]. \end{aligned}$$

Jak widać, nawias kwadratowy to błąd e będący efektem obliczeń zmiennoprzecinkowych. Możemy go oszacować w następujący sposób

$$|e| \leq |w-w_F| \leq (4|ax|+4|by|+3|cz|+|d|)\eta = \Delta w_F.$$

Dokładna wartość w należy zatem do przedziału $[w_F-\Delta w_F, w_F-\Delta w_F]$. Tylko wtedy, gdy oba krańce tego przedziału mają taki sam znak, to znak w jest taki sam jak znak obliczonej wartości w_F. Dla punktów znajdujących się blisko płaszczyzny, nie musi to już być prawdą.

6 Doświadczenie 5

Rozważmy kolejny przykład geometryczny. Załóżmy, że chcemy obliczyć punkt przecięcia prostych o równaniach

$$y = a_1x+b_1$$

oraz

$$y = a_2x+b_2.$$

Dla uproszczenia rozważań, załóżmy, że tylko b_1 i b_2 są reprezentowane z przybliżeniem. Wówczas zamiast zadania wyjściowego, poszukujemy punktu przecięcia prostych

$$y = a_1x+b_{1F}$$

oraz

$$y = a_2x+b_{2F}.$$

W obliczeniach zmiennoprzecinkowych prosta $y = a_1x+b_1$ będzie reprezentowana przez jakąś prostą z pęku między $b_1(1-\eta)$ a $b_1(1+\eta)$, a prosta $y = a_2x+b_2$ prostą z pęku między $b_2(1-\eta)$ a $b_2(1+\eta)$. Tak więc błędy reprezentacji danych powodują, że poszukiwać będzimy „idealnego" punktu przecięcia prostych gdzieś w czworokącie wyznaczonym przez ograniczenia pęków prostych. Jeśli proste są prawie równoległe, a więc gdy $a_1 \approx a_2$, to wyznaczone rozwiązanie może znacznie odbiegać od prawdziwego.

7 Wnioski

Przedstawione w punkcie 2 rozważania, mimo iż bardzo uproszczone, ukazują jednak cechy zmiennopozycyjnego sposobu reprezentacji liczb rzeczywistych w komputerze. Należy mieć świadomość, że zwiększenie ilości bitów służących do reprezentacji liczby nie wnosi nowej jakości. Owszem, możliwe, że w takm przypadku zamiast 1718, poprawnych będzie 1 milion sum. Ciągle będzie to jednak ok 10% wszystkich możliwych kombinacji. Dla większości i tak będziemy zmuszeni użyć przybliżenia. Zwiększenie ilości bitów skutkuje jedynie przesunięciem pewnych granic, ale nie ich zlikwidowaniem (np. w rozważanym formacie można przedstawić wszystkie liczby całkowite z zakresu [1,15]; powyżej bywa już różnie). W szczególności na przykładzie dwóch ostatnich reprezentowalnych liczb widać, jak duże niedokładności pociąga za sobą używanie „dużych" (dużych dla danego formatu) liczb[4].

W punkcie 3 chcieliśmy natomiast zwrócić uwagę na stabilność obliczeń numerycznych. Oto okazuje się, że nie ma nic bardziej niepewnego niż obliczenia przeprowadzone na komputerze. Ponownie „winę" za taki stan rzeczy ponosi przyjęty format zapisu liczb.

[4]Podobnie sprawa wygląda w przypadku „małych" liczb.

Literatura

[1] P. Fulmański, Ś. Sobieski, *Wstęp do informatyki. Podręcznik*, Wydawnictwo UŁ, 2005.

[2] M. Jankowski, *Elementy grfiki komputerowej*, Wydawnictwa Naukowo-Techniczne, Warszawa, 1990.

[3] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia 2001.

[4] H.-O. Peitgen, H. Jurgens, D. Saupe, *Granice haosu. Fraktale*, tom I, Wydawnictwo Naukowe PWN, Warszawa, 1997.