Wojciech Myszka

Laboratorium 4 wer. 39 z drobnymi modyfikacjami!

2018-04-07 10:42:17 +0200

1. Laboratorium 4: Arytmetyka zmiennoprzecinkowa komputerów

1.1. Wstęp

Te zajęcia nawiązują do sposobu zapisu liczb opisanego w „części teoretycznej". Ich celem jest odświeżenie tej wiedzy, zaprojektowanie prostego eksperymentu pozwalającego sprawdzić jak obliczenia są wykonywane (zwłaszcza w arkuszu kalkulacyjnym).

Druga część ma pokazać w jaki sposób można te ograniczenia omijać korzystając ze specjalnych bibliotek.

1.2. Proste obliczenia

Jest taki program [1][1]

```
#include <stdio.h>
int main()
{
    float s;
    double d;
    long double e;
    int i;
    s = d = e = 0.5;
    for(i=1;i<=100;i++)
    {
        s = 3.8F * s * (1.F - s);
        d = 3.8 * d * (1. - d);
        e = 3.8L * e * (1.L - e);
        if (i%10==0) printf("%10i %16.5f %16.5lf %16.5Lf\n", i, s, d, e);
    }
```

[1] Problem pojawił się po raz pierwszy podczas prób symulacji izolowanej populacji insektów.

```
//    printf("%ld\n", sizeof(long double));
    return 0;
}
```

nie prowadzi on żadnych skomplikowanych obliczeń przeprowadzając jedynie proste działanie:

$$x_{i+1} = \alpha x_i(1 - x_i)$$

gdzie $\alpha = 3.8$, a $x_0 = 0.5$.

Wykonuje on obliczenia na liczbach typu **float** (32 bity), **double** (64 bity) i **long double** (128 bitów). Program wykonuje 100 iteracji drukując co dziesiątą z nich. Wyniki programu (można go łatwo zapisać na dysku jako `caos.c` i skompilować poleceniem `gcc -o caos caos.c`.

Wyniki wyglądają jakoś tak:

10	0.18510	0.18510	0.18510
20	0.23951	0.23963	0.23963
30	0.88445	0.90200	0.90200
40	0.23023	0.82493	0.82493
50	0.76124	0.53714	0.53714
60	0.72004	0.66878	0.66879
70	0.89952	0.53189	0.53203
80	0.61599	0.93573	0.93275
90	0.84367	0.68312	0.79884
100	0.94858	0.65620	0.23138

Jak widać rezultaty różnią się w sposób znaczący. Jedynym wytłumaczeniem jest różna liczba bitów uwzględnianych w obliczeniach. Zatem przypuszczać można że wersja 128-bitowa jest najbliższa rzeczywistości.

Jeżeli wierzyć rozważaniom z [1] „poprawne wyniki” (uzyskane w arytmetyce o 1000 cyfr) są następujące:

10	0.18509
20	0.23963
30	0.90200
40	0.82492
50	0.53713
60	0.66878
70	0.53202
80	0.93275
90	0.79885
100	0.23161

i jak widać odrobinę różnią się od najlepszego uzyskanego wyniku.

Okazuje się, że dla $0 \leq \alpha < 3$ zachowanie obliczeń jest deterministyczne, a dla $3 \leq \alpha < 4$ — chaotyczne. Takie systemy (chaotyczne) są bardzo czułe na dokładność obliczeń.

1.3. Mathematica

Mathematica dla każdego obliczenia potrafi podać precyzję wyniku. Służy do tego funkcja `Precision[]`. Zazwyczaj obliczenia wykonywane są z „maszynową precyzją“ (co oznacza obliczenia na liczbach podwójnej precyzji (64 bity).

Dodatkowo można zadeklarować w jakiej precyzji mają być wykonywane obliczenia.

```
Block[{$MinPrecision = 5,$MaxPrecision = 5},x = a~* x * (1 - x)]
```

Znalazłem też instrukcję (której do końca nie rozumiem) powodującą, że wszystkie obliczenia w notatniku będą odbywać się z zadaną precyzją. Wygląda ona jakoś tak:

```
$PreRead=(#/.s_String/; StringMatchQ[s, NumberString] &&
   Precision @ ToExpression @ s==MachinePrecision:> s<>"‘1000."&);
```

i powinna być umieszczona na początku notatnika[2]. Precyzja określona jest stałą tekstową na samym końcu polecenia (`"‘1000.`"). Demonstruje to prosty przykład (notatnik Mathematici).

Powinno to wystarczyć do zaprogramowania opisywanego przykładu.

Mathematica to pełnowymiarowy język programowania, wyposażony w instrukcje warunkowe (`If` czy służące do tworzenia pętli (`For, Do, While`). Prosty przykład programiku z tymi instrukcjami zawiera notatnik.

Jak ktoś ciągle ma kłopoty może skorzystać z kolejnego przykładowego programu.

1.3.1. Pakiet Computer Arithmetics

Mathematica wyposażona jest w pakiet Computer Aruithmetics służący do symulowania obliczeń korzystając z jakiejś „wymyślonej” arytmetyki komputerowe. Pozwala to badać jaki wpływ na na precyzję obliczeń może mieć liczba dostępnych bitów czy zakres zmian wartości wykładnika.

W szczególności pakiet pozwala na symulowanie komputerów o arytmetyce dziesiętnej (nie binarnej). Wydaje się że do prowadzenia złożonych obliczeń inżynierskich taki rodzaj arytmetyki może być lepszy niż stosowana dziś arytmetyka binarna.

[2] Ustalona precyzja będzie obowiązywała we wszystkich równocześnie otwartych notebookach

Aby z pakietu skorzystać, trzeba go załadować. Służy d tego polecenie iNeeds.

```
Needs["ComputerArithmetic‘"].
```

Następnie definiujemy rodzaj arytmetyki poleceniem `SetArithmetic[d,b]` gdzie `d` to liczba używanych cyfr (niestety z zakresu 1 do 10), a `b` to podstawa systemu liczenia z zakresu od 2 do 16. Nie można zatem poszaleć z liczbą używanych w obliczeniach cyfr (co utrudnia nieco nasze zadanie).

Kolejne polecenie to `ComputerNumber` definiujące obiekt w wybranej arytmetyce:

```
x = ComputerNumber[0.5]
a = ComputerNumber[3.8]
```

(Standardowo nie można wykonywać obliczeń na liczbach mieszanych więc trzeba uważać na zapis:

```
x = x * a * (1 - x)
```

gdyż jedynka jest z innego świata (obiektem innego typu). Czyli powyższe trzeba zapisać jako:

```
x = x * a * (ComputerNumber[1] - x)
```

albo

```
one = ComputerNumber[1]
x = x * a * (one - x)
```

Takie obliczenia również można wykonać aby sprawdzić zachowanie naszego problemu. Od razu podpowiadam, że można wykorzystać arytmetykę szesnastkową o 10 cyfrach (uzyskamy większą precyzję niż w przypadku arytmetyki 10 o 10 cyfrach). Ale i tak to za mało.

1.4. Python

Język programowania python może być wyposażony w bibliotekę mpmath ([2]), pozwalającą na obliczenia w dowolnej precyzji.

Korzystanie z niej jest stosunkowo proste. Trzeba ją najpierw „załadować”

```
from mpmath import *
```

Polecenia `mp.prec` i `mp.dps` definiują precyzję obliczeń. Pierwsze określa ją w bitach, drugie w cyfrach dziesiętnych. Wystarczy korzystać tylko z jednego, gdyż ich wartości zależne są od siebie: prec $\sim$ 3.33 dps

```
>>> mp.dps=100
>>> mp.prec
336
```

Największy kłopot jest z deklaracją stałych, które nie mogą być standardowymi literałami.

```
>>> x=mpf(4)
>>> x
mpf('4.0')
>>> x/mpf('3.')
mpf('1.3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
>>> mp.dps=30
>>> x/mpf('3.')
mpf('1.33333333333333333333333333333327')
```

Wprowadzając dane trzeba pamiętać o błędach konwersji z układu dziesiętnego do dwójkowego. W związku z tym spodziewać można się dziwnych zjawisk

```
>>> mpf(.1)
mpf('0.10000000000000000555111151231257 83')
>>> mpf('.1')
mpf('0.09999999999999999999999999999951')
>>> mpf(1)/mpf(10)
mpf('0.09999999999999999999999999999951')
```

w pierwszym przypadku najpierw robiona jest konwersja z liczby dziesiętnej do binarnej, a później wynik traktowany jest jako liczba wielokrotnej precyzji. W drugim wypadku do funkcji `mpf` przekazujemy tekst, który jest interpretowany i konwertowany od razu na liczbę wielokrotnej precyzji.

Reszta jest już stosunkowo prosta.

1. Dodawanie: `fadd()`
2. Odejmowanie: `fsub()`
3. Negacja (zmiana znaku): `fneg()`
4. Iloczyn: `fmul()`
5. Iloraz: `fdiv()`

Chociaż jak już zmienne zostały „zadeklarowane” jako dużej precyzji można korzystać ze standardowych operatorów `+-/*`.

```
>>> mp.dps = 100
>>> x = mpf('1.')
>>> y = mpf('3.')
>>> z~= x / y
>>> nprint(z,20)
0.33333333333333333333
```

1.4.1. Idle

Można ułatwić sobie pracę w pythonie uruchamiając proste środowisko graficzne zwane **idle**. Pozwala ono zapisać program korzystając z prostego edytora i bardzo łatwo uruchamiać go.

1.4.2. Jupyter

Znacznie bardziej wygodne może być skorzystanie z notatnika Jupyter. Przygotowałem odpowiedni przykład, który można pobrać (wraz z innymi) stąd.

1.5. Zadania do wykonania

1. Zaproponować eksperyment obliczeniowy wyznaczający liczbę bitów i dokładność prowadzonych obliczeń.
2. Przetestować go na matlabie i Mathematici, w arkuszu kalkulacyjnym LibreOffice calc, Blockly i — jeżeli ktoś konto Google posiada dla arkusza kalkulacyjnego Google (i ewentualnie dla Microsoft Office Online).
3. Pobawić się w pythonie[3] i Mathematici prowadzeniem obliczeń w dużej precyzji programując podany na początku przykład i porównując (jakieś wykresy?) wyniki uzyskane dla różnych precyzji obliczeń.

1.6. Instrukcja w postaci jednego pliku...

...jest również dostępna.

[3] Programowanie w pythonie jest stosunkowo proste ([3], [4], [5], [6], [7]). Można również zerknąć do instrukcji laboratoryjnych do zajęć z Technologii Informacyjnych. Na stronach TI dla informatykji znajduje się również obszerny spis literatury. Natomiast, jeżeli ktoś boi się zaczynać, powinien zerknąc na przygotowaną stronę w Blockly zawierający program i pozwalającą na podejrzenie wersji skonwertowanej do pythona. Skopiować z tego okienka jest trudno, ale się da. Klikamy w nie i w ciemno Ctrl-A, Ctrl-C i zawartość wklejamy do edytora. Później trzeba tylko jeszcze podzielić na linie (i zadbać o właściwe wcięcia, ale jak robić to ostrożne – udaje się).

Bibliografia

[1] Ward A., *Some issues on floating-point precision under linux*, Linux Gazette, , 53, 2000, URL `https://linuxgazette.net/53/ward.html`.

[2] Johansson F., *mpmath's documentation*, URL `http://mpmath.org/doc/current/index.html` 2015.

[3] Pilgrim M., *Zanurkuj w Pythonie*, WikiBooks 2010, URL `http://pl.wikibooks.org/wiki/Python`.

[4] *Python documentation index*, `http://www.python.org/doc/`, URL `http://www.python.org/doc/` 2010.

[5] Shaw Z.A., *Learn Python The Hard Way* 2010, URL `http://learnpythonthehardway.org/book/`.

[6] *Python Programming*, WikiBooks 2010, URL `http://en.wikibooks.org/wiki/Python_Programming`.

[7] Tagliaferri L., *How to code in Python 3*, DigitalOcean 2018, URL `https://www.digitalocean.com/community/tutorials/digitalocean-ebook-how-to-code-in-python`.