SPRAWOZDANIE NR 1

OBLICZENIA NAUKOWE

AUTOR: MARCIN ADAMCZYK
NR INDEKSU: 221 429

ZAD 1

1. Wstęp

   Celem zadanie było iteracyjne wyznaczenie kolejno epsilonów maszynowych, liczb eta, oraz liczb realmax dla typów zmiennopozycyjnych (Float16 Float32 i Float64), a następnie porównanie wyników z odpowiednimi wartościami zwracanymi przez wbudowane funkcje języka Julia.

2. Rozwiązania

   2.1 Macheps (machine epsilon), to najmniejsza liczba (większa od 0) taka, że:

   $$1.0 \oplus \text{macheps} > 1.0$$

   Uzyskuję ją dzieląc swoją zmienną o początkowej wartości 1.0 w pętli przez liczbę 2 do momentu, w którym nasza liczba staje się na tyle mała, że nie zachodzi powyższy warunek.

   2.2. Eta jest najmniejszą liczbą większą od 0.0. Uzyskuję ja podobnie jak macheps, inny jest jedynie warunek końca pętli (dopóki zmienna jest większa od 0.0).

   2.3. Aby uzyskać realmax rozpoczynam od ustawienia x na wartość prevfloat(1.0) co daje mi zapełnioną „jedynkami" mantysę. Następnie mnożę x przez 2 dopóki jego wartość jest różna od „nieskończoności".

3. Wyniki i interpretacja

   Na początku zaznaczę, że wszystkie wyniki uzyskane przeze mnie pokrywają się z wartościami zwracanymi przez wbudowane funkcje. Analizując zapisy bitowe należy pamiętać o specjalnych wartościach typu NaN itp., których zapis bitowy jest podobny do wartości granicznych przedstawianych przeze mnie poniżej. Omawiane przykłady zostały uzyskane dla precyzji Float64.

   3.1. Macheps:

Float64 eps:	2.220446049250313e-16
Float32 eps:	1.1920929f-7
Float16 eps:	0.000977

   Dodatkowo dla porównania wartości z pliku <float.h>:

```
C eps 64 = "2.220446049250313080847263336181640625e-16"
C eps 32 = "1.1920928955078125e-7"
```

   Aby pokazać dlaczego akurat ta liczba spełnia warunek bycia macheps-em najlepiej dodać ją do 1 i zobaczyć zapis bitowy:

```
0 01111111111 0000000000000000000000000000000000000000000000000001
```

   Liczba ta będzie wyglądała w następujący sposób:

   $$2^0 * 1.[mantysa]$$

   Mantysa natomiast, właśnie w tym wypadku jest najmniejszą możliwą co jednocześnie bezpośrednio nawiązuje do precyzji arytmetyki.

3.2. Eta

eta 64: 5.0e-324 eta 32: 1.0f-45 eta 16: 6.0e-8

eta 64: 0 00000000000 0000000000000000000000000000000000000000000000000001

Jak widać po zapisie bitowym eta 64 jest po prostu najmniejszą (dodatnią) liczbą możliwą do zapisania.

Zauważyć można również, że wszystkie bity cechy są „zerowe” co oznacza, że jest to liczba nieznormalizowana (subnormal).

3.3. Max

realmax 64: 1.7976931348623157e308 realmax 32: 3.4028235f38 realmax 16: 6.55e4

max 64: 0 11111111110 1111111111111111111111111111111111111111111111111111

Odwrotnie niż w przypadku eta. Realmax jest największą liczą jaką można zapisać na 64 bitach (stosując reguły zapisu przyjęte dla formatu Float64)

Ponownie dla porównania wyniki z <float.h>

C max 64 = "1.797693134862315708145274237317043567980 70e+308" C max 32 = "3.402823466385288598117041834845169 2544e+38"

ZAD 2

1. Wstęp

   Zadanie miało na celu sprawdzenie, że za pomocą wyrażenia 3(4/3 - 1) - 1 jesteśmy w stanie wyznaczyć macheps.

2. Wynik

   Liczba uzyskana w wyniku obliczenia powyższego wyrażenia, faktycznie odpowiada wartości eps dla wszystkich trzech typów zmiennopozycyjnych, dla których przeprowadzany był eksperyment.

4/3	0 01111111111 0101010101010101010101010101010101010101010101010101
4/3 - 1	0 01111111101 0101010101010101010101010101010101010101010101010100
3(4/3 - 1)	0 01111111101 0101010101010101010101010101010101010101010101010100
3(4/3 - 1) - 1	1 01111001011 0000000000000000000000000000000000000000000000000000

Powyższa tabela przedstawia zapis bitowy wyników pojedynczych operacji dla typu Float64.

Liczba w ostatnim wierszu to właśnie macheps.

ZAD 3

1. Wstęp

   W zadaniu tym należało sprawdzić równomierność rozmieszczenia liczb z zakresu [1,2] w arytmetyce Float64 (double).

2. Rozwiązanie

   Zapamiętuję stałą wartość delta jako d = 2 ^ (-52);

   Zapisuję w zmiennej x wartość 0;

   W pętli obliczam wartość wyrażenia 1 + k*d oraz porównuję ją z x;

   Po każdym przejściu pętli inkrementuje k (zaczynam od 0), a za x podstawiam nextfloat(x);

   Pętla kończy działanie w przypadku gdy porównanie zwróci false, lub x osiągnie wartość graniczną – 2;

   Po zakończeniu pętli, x mający wartość >=2, oznacza, że przeszliśmy wszystkie liczby w zakresie i każdą z nich mogliśmy przedstawić za pomocą formuły 1 + k*d.

   Niestety, pomimo, że asymptotyczna złożoność algorytmu to tylko O(n), program działa bardzo wolno ponieważ wciąż musi przejść przez wszystkie 2 ^ 52 liczb znajdujących się w zakresie…

3. Wynik i interpretacja

   Po zakończeniu działania program oznajmia, iż faktycznie liczby double w zakresie [1,2] są rozłożone równomiernie.

   Aby wyjaśnić dlaczego tak jest poniżej przedstawiam zapis bitowy dwóch pierwszych i dwóch ostatnich liczb tego zakresu.

1 =	0 01111111111 0000000000000000000000000000000000000000000000000000
nextfloat(1) =	0 01111111111 0000000000000000000000000000000000000000000000000001
prevfloat(2) =	0 01111111111 1111111111111111111111111111111111111111111111111111
2 =	0 10000000000 0000000000000000000000000000000000000000000000000000

   Zauważyć można, że liczby z zakresu [1, 2) różnią się jedynie mantysą, co więcej wartość cechy sprawia, ze liczby te wyglądają w następujący sposób:

   2^0 (== 1) * 1. [mantysa]

   W tym momencie widać, że różnica pomiędzy kolejnymi wartościami liczb w tym zakresie jest taka jak różnica mantys tych liczb, a zwiększanie wartości mantysy „bit po bicie” powoduje wzrost na poziomie 2 ^ (-52).

4. Wyniki w innych przedziałach

   a. [1/2, 1]

      Dwie pierwsze i przedostatnia liczba:

      0 01111111110 0000000000000000000000000000000000000000000000000000
      0 01111111110 0000000000000000000000000000000000000000000000000001
      0 01111111110 1111111111111111111111111111111111111111111111111111

      Jak widać te liczby również różnią się jedynie mantysą, a więc będzie ich tyle samo co w poprzednim przypadku (2^52), ale w porównaniu z poprzednim zakresem, ten jest dwa razy mniejszy, a więc same liczby musza być rozmieszczone dwa razy gęściej.

   b. [2, 4]

      Podobnie jak powyżej: dwa pierwsze i przedostatni wynik:

      0 10000000000 0000000000000000000000000000000000000000000000000000
      0 10000000000 0000000000000000000000000000000000000000000000000001
      0 10000000000 1111111111111111111111111111111111111111111111111111

      I znów ta sama sytuacja, ale w tym wypadku zakres jest dwa razy dłuższy niż w pierwszym, a zatem gęstość liczb dwa razy mniejsza

5. Wnioski

   Czym bliżej 0.0 znajdują się liczby, tym gęściej są rozłożone. Sama gęstość oraz jej przedziały oparte są na potęgach dwójki, a to pozwala przedstawiać liczby z odpowiednich zakresów za pomocą odpowiednich formuł, tak jak w przypadku początku tego zadania.

ZAD 4

1. Wstęp

   Celem zadania jest znalezienie w przedziale (1, 2) takiej liczby x, że:

   $$x * (1/x) != 1$$

2. Rozwiązanie

   Zaczynając od nextfloat(1.0) zaczynamy przeszukiwanie w pętli całego zakresu do momentu znalezienia „anomalii". W ten sposób znajdujemy od razu najmniejszą taką liczbę.

   Na szczęście następuje to dużo szybciej niż w przypadku zadania nr 2.

3. Wynik i interpretacja

   Wynik przedstawia się następująco:

   ```
   x == 1.000000057228997
   x == 0 01111111111 0000000000000000000000001111010111001011111100101010
   ```

   Powyższy przykład ilustruje niepożądany przypadek, kiedy to operacja dzielenia nie będzie odwracalna.

   Jeśli wykonamy działanie przedstawione we wstępie dla tej liczby otrzymamy:

   ```
   0.9999999999999999
   ```

   Jest to spowodowane możliwością zapisu jedynie ograniczonej ilości liczb. Wyniki działań, które nie mogą być przedstawione dokładnie są zaokrąglane, co przy wykonaniu kolejnych operacji może potęgować błąd obliczeń.

4. Wniosek

   Podczas obliczeń na liczbach zmiennopozycyjnych zawsze trzeba brać po uwagę możliwość wystąpienia błędu spowodowanego przez zaokrąglenie i dalszych konsekwencji mogących wynikać z wielokrotnego powielania się takiego błędu.

   Przykładem mógłby być warunek pętli

   while (x(1/x) == 1){}*

   Bardzo szybko moglibyśmy w tym wypadku niespodziewanie zakończyć działanie pętli.

ZAD 5

1. Wstęp

   Zadanie polegało na obliczeniu iloczynu skalarnego dwóch wektorów przy użyciu czterech różnych metod, różniących się jedynie kolejnością sumowania iloczynów składowych.

2. Rozwiązanie

   Sposób implementacji jest trywialny więc nie będę się rozpisywał na jego temat. Dodam jedynie, że dla dwóch ostatnich sposobów (dodawanie najpierw największych składowych i najpierw najmniejszych) użyłem wbudowanego sortowania, oraz zapamiętywałem w zmiennej indeks w którym składowe w posortowanej tablicy zmieniają znak wartości.

3. Wyniki i interpretacja

```
forward 64:     1.0251881368296672e-10
backward 64: -1.5643308870494366e-10
maxFirst 64:    0.0
minFirst 64:    0.0

forward 32:    -0.4999443
backward 32: -0.4543457
maxFirst 32:  -0.5
minFirst 32 :  -0.5
```

correct answer: -1.00657107000000_{10} **- 11**

Ciężko jest wyjaśnić dokładnie w którym miejscu w obliczeniach powstały takie, a nie inne błędy, ponieważ analizując zapis pojedynczych działań okazuje się, że niepozorne na pierwszy rzut oka wyniki cząstkowe dając odbiegające od oczekiwanych rezultaty.

To co jednak widać, to że wynik iloczynu jest liczbą zbliżoną do 0.0, a to oznacza wektory prostopadłe, które jak wiadomo mają tendencję do generowania dużych błędów względnych

Gdyby nie tak specyficznie dobrane wartości wartościową obserwacją byłaby różnica między dodawaniem najpierw wartości o małej wartości bezwzględnej, a operacją odwrotną. Tutaj jednak wyniki są takie same…

ZAD 6

1. Wstęp

   Zadanie polegało na policzeniu wartości tej samej funkcji zapisanej w dwóch różnych postaciach:

$$f(x) = \sqrt{x^2 + 1} - 1$$

$$g(x) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

   Jako argument x miał przyjmować wartości $x = 8^{-1}, 8^{-2}, \ldots$

2. Wynik i interpretacja

$x = 8^{-1}$	f(x) = 0.0077822185373186414
$x = 8^{-1}$	g(x) = 0.0077822185373187065
$x = 8^{-8}$	f(x) = 1.7763568394002505e-15
$x = 8^{-8}$	g(x) = 1.7763568394002489e-15
$x = 8^{-9}$	f(x) = 0.0
$x = 8^{-9}$	g(x) = 2.7755575615628914e-17
$x = 8^{-10}$	f(x) = 0.0
$x = 8^{-10}$	g(x) = 4.336808689942018e-19

   Jak widać do pewnego momentu obie funkcje zwracają bardzo zbliżone wyniki. Jednak od $x = 8^{-9}$ funkcja f zaczyna zwracać 0.0. Funkcja g natomiast dalej podaje wartości prawidłowe (porównywane z WolframAlpha).

Dzieje się tak ponieważ w funkcji f (dla bardzo małego x) dokonywane jest odejmowanie wartości zbliżonych. Zauważmy, że wtedy:

$$\sqrt{x^2+1} \approx 1$$

Tak więc gdy dokonamy odejmowania liczby zbliżonej do 1.0 i samego 1.0 uzyskamy duży błąd, a w tym wypadku dochodzący do 100%.

3. Wniosek

   Odejmowanie wartości zbliżonych generuje duże błędy względne.

ZAD 7

1. Wstęp

   Naszym celem było obliczanie przybliżonej wartości pochodnej zadanej funkcji:

$$f(\mathrm{x}) = \sin(x) + \cos(3x)$$

$$f'(x_{0)} \approx \widetilde{f}'(x_0) = \frac{f(x_0+h) - f(x_0)}{h}$$

   w punkcie $x_0 = 1$ oraz błędów:

$$\left|f'(x_0) - \widetilde{f}'(x_0)\right|$$

   dla $h = 2^{-n}$ $(n = 0, 1, \dots, 54)$.

2. Rozwiązanie

   Jako dokładną wartość pochodnej funkcji f przyjmuję wartość funkcji

$$f'(\mathrm{x}) = \cos(x) - 3sin(3x)$$

   Samo działanie programu nie wymaga raczej specjalnego komentarza.

3. Wyniki i interpretacja

n = 0 difference	= 1.9010469435800585
n = 1 difference	= 1.753499116243109
n = 38 difference	= 1.0776864618478044e-6
n = 39 difference	= -5.9957469788152196e-5
n = 53 difference	= -0.11694228168853815
n = 54 difference	= -0.11694228168853815

   Powyższa tabela przedstawia wartości błędów (bo to one tak naprawdę nas interesują…). To co można zauważyć, to że początkowo wraz ze zmniejszaniem h błąd maleje – ale nie jest to żadna niespodzianka.

   Niespodziewaną natomiast sytuacją jest ponowny wzrost błędu od pewnego momentu. Jest to spowodowane utratą części danych po dodaniu bardzo małego h=(2^{-n}) do stosunkowo dużego 1.

4. Wnioski

   Ustalenie zbyt małego h spowoduje utratę jego części podczas operacji h + 1, co z kolei przeniesie się na dalsze (względnie duże) błędy w obliczeniach.