Obliczenia naukowe

Lista 1

Arkadiusz Ziobrowski
229728

1 Zadanie pierwsze

1.1 Opis problemu

Celem zadania było wyznaczenie epsilonów maszynowych (*macheps*), liczb *eta* oraz liczb (MAX) dla dostępnych w języku `Julia` typów zmiennopozycyjnych `Float16`, `Float32` i `Float64`.

1.2 Rozwiązanie

Wyznaczanie *macheps* Epsilon maszynowy to najmniejsza taka liczba, spełniająca dla danej arytmetyki następującą zależność:

$$fl(1.0 + macheps) > 1.0$$

Epsilon maszynowy wyznaczam wychodząc od wartości `1.0` i dzieląc ją w pętli przez `2.0` do momentu aż przedstawiona powyżej nierówność przestaje obowiązywać. Otrzymana w zmiennej liczba to właśnie *macheps*.

Wyznaczanie *eta* Liczba *eta* to najmniejsza taka liczba spełniająca dla danej arytmetyki następującą nierówność:

$$eta > 0.0$$

Wyznaczam ją dzieląc iteracyjnie zmienną o początkowej wartości `1.0` przez `2.0` dopóki powyższy warunek przestaje być spełniany. Otrzymana w ten sposób liczba to *eta*.

Wyznaczanie *MAX* Liczba MAX jest największą skończoną wartością, która da się przedstawić w danej arytmetyce. Wyznaczam ją iteracyjnie poprzez mnożenie zmiennej przez `2.0` do momentu aż warunek `isinf(zmienna)` nie zostanie spełniony. W tym przypadku pierwotną wartością zmiennej nie jest jeden, lecz liczba `prevfloat(1.0)`. Dobór takiej wartości wynika ze sposobu reprezentacji `1.0` w standardzie IEEE754. Mantysa jedynki składa się bowiem z samych zer, a wzięcie najmniejszej liczby mniejszej od jedynki spowoduje zapełnienie tej mantysy wyłącznie jedynkami. Mnożenie o dwa będzie przesuwać wówczas bity w cesze.

1.3 Wyniki i interpretacja

Wyniki działania zaimplementowanych przez mnie funkcji pokrywają się z wynikami działania funkcji z języka `Julia`.

1.3.1 *Macheps*

Wyniki *macheps*			
Nazwa funkcji	`Float16`	`Float32`	`Float64`
`eps(::Float)`	0.000977	1.1920929e-7	2.220446049250313e-16
`macheps`	0.000977	1.1920929e-7	2.220446049250313e-16

Dla porównania wartości z pliku nagłówkowego`float.h` wynoszą dla arytmetyki `single`: 1.1920928955078125e-7, zaś dla `double`: 2.220446049250313080847263336181640625e-16.

Macheps uzyskujemy przez pomnożenie precyzji arytmetyki ϵ przez dwa. Przykładowo dla arymetyki `double` epsilon maszynowy wynosi 2^{-52}, a precyzja arytmetyki to 2^{-53}. Poniższy eksperyment pokazuje zależność między epsilonem maszynowym, a precyzją arytmetyki `Float32`.

```
   bits(prevfloat(Float32(1.0)):       00111111011111111111111111111111    bits(Float32(1.0) -
eps(Float32)): 00111111011111111111111111111110 //odejmowanie epsilona maszynowego
bits(Float32(1.0) - Float32(2.0^-24)): 00111111011111111111111111111111 //odejmowanie precyzji
arytmetyki
bits(Float32(1.0) - Float32(2.0^-25)):   00111111100000000000000000000000  //odejmowanie liczby
mniejszej od precyzji arytmetyki
```

1.3.2 *Eta*

Wyniki *eta*			
Nazwa funkcji	`Float16`	`Float32`	`Float64`
`nextfloat(::Float(1.0))`	6.0e-8	1.0e-45	5.0e-324
`eta`	6.0e-8	1.0e-45	5.0e-324

W zapisie bitowym *eta* dla arytmetyki `single` wynosi:

`bits(eta32()))`: 00000000000000000000000000000001

Jak widać jest to najmniejsza możliwa do zapisania liczba dodatnia. Jest w postaci nieznormalizowanej, co pokazują bity cechy, a wynosi ona dokładnie tyle ile MIN_{sub}.

1.3.3 *MAX*

Wyniki *MAX*			
Nazwa funkcji	`Float16`	`Float32`	`Float64`
`realmax(::Float)`	6.55e4	3.4028235e38	1.7976931348623157e308
`max`	6.55e4	3.4028235e38	1.7976931348623157e308

W zapisie bitowym *MAX* dla arytmetyki `single` wynosi:

`bits(max32()))`: 01111111011111111111111111111111

Jest to największa liczba, która da się przedstawić w danej arytmetyce. W implementacji języka C wartości dla arytmetyk `single` i `double` wynoszą:

`single`: 3.40282346638528859811704183484516925440e38
`double`: 1.79769313486231570814527423731704356798070e308.

2 Zadanie drugie

2.1 Opis problemu

Celem zadania drugiego było sprawdzenie eksperymentalnie słuszności stwierdzenia, iż epsilon maszynowy można otrzymać obliczając $3 * (4/3 - 1) - 1$ w arytmetyce zmiennopozycyjnej.

2.2 Wyniki i interpretacja

Liczba otrzymana w wyniku zgadza się z wartością *macheps* dla zadanych arytmetyk zmiennopozycyjnych.

$4.0/3.0$	0 01111111 01010101010101010101011
$4.0/3.0 - 1.0$	0 01111101 01010101010101010101100
$3.0 * (4.0/3.0 - 1.0)$	0 01111111 00000000000000000000001
$3.0 * (4.0/3.0 - 1.0) - 1.0$	0 01101000 00000000000000000000000

W powyższej tabeli przedstawione zostało krokowe wykonanie wyrażenia zaproponowanego przez Kahana dla arytmetyki `single`. Liczba w ostatnim wierszu tabeli odpowiada epsilonowymi maszynowemu.

3 Zadanie trzecie

3.1 Opis problemu

Celem zadania było sprawdzenie równomierności rozmieszczenia liczb zmiennopozycyjnych w przedziale $[1, 2]$ z krokiem $\delta = 2^{-52}$, a następnie zbadanie rozmieszczenia liczb w przedziałach $[1/2, 1]$ oraz $[2, 4]$.

3.2 Rozwiązanie

Pierwsza liczba z zadanego przedziału jest zapamiętywana w zmiennej. W zmiennej również zapamiętywany jest stały krok dla arytmetyki `double`, czyli $\delta = 2^{-52}$. W pętli liczba z zadanego przedziału jest zwiększana o `k * δ`, co jest implementacją wzoru: $x = 1 + k * \delta$. Mnożnik k jest zwiększany w każdym przejściu pętli. Oczekiwaną wartością mnożnika k jest 2^{52} po zakończeniu iterowania, czyli w momencie przekroczenia przez `x` liczby z końca przedziału. Gdy taka wartość jest osiągana to znaczy, iż liczby są równomiernie rozmieszczone w zadanym przedziale.

3.3 Wyniki i interpretacja

Poniższa tabela przedstawia pierwsze przejścia pętli oraz przedostatni krok.

$1.0 + 0 * \delta$	0011111111110000000000000000000000000000000000000000000000000000
$1.0 + 1 * \delta$	0011111111110000000000000000000000000000000000000000000000000001
$1.0 + 2 * \delta$	0011111111110000000000000000000000000000000000000000000000000010
$1.0 + 3 * \delta$	0011111111110000000000000000000000000000000000000000000000000011
$1.0 + 4 * \delta$	0011111111110000000000000000000000000000000000000000000000000100
$2.0 - \delta$	0011111111111111111111111111111111111111111111111111111111111111

Jak widać bity zwiększają się o 1. W przedostatnim kroku cała mantysa jest zapełniona bitami o wartości 1 i kolejne dodanie δ spowoduje wyzerowanie mantysy i dodanie 1 do cechy. Żeby zmieniła się cecha do mantysy δ musi być dodana 2^{52} razy. Im bliżej 0.0 tym gęściej rozmieszczone są liczby.

3.4 Wnioski

Dla przedziałów $[1/2, 1]$ oraz $[2, 4]$ otrzymane rozłożenie liczb jest także równomierne, jednakże dla $[1/2, 1]$ z powodu, iż przedział jest dwa razy mniejsze wynika, iż liczby są dwa razy gęściej rozłożone. Analogicznie dla przedziału $[2, 4]$ z uwagi, iż jest on dwa razy większy niż przedział $[1, 2]$ wynika, że liczby są tam dwa razy rzadziej rozmieszczone.

4 Zadanie czwarte

4.1 Opis problemu

Celem zadania było eksperymentalne znalezienie liczby nie spełniającej równania $x * (1/x) = 1$ dla $1 < x < 2$ oraz najmniejszej takiej liczby niespełniającej zadanego równania dla arytmetyki `double`.

4.2 Rozwiązanie

Dla pierwszego przypadku, czyli znalezienia takiej anomalii w przedziale $(1, 2)$ zaczynamy przeszukiwanie od liczby `nextfloat(1.0)` i iterujemy po kolejnych `nextfloat` do momentu napotkania anomalii. W przypadku dojścia do krańca przedziału, czyli liczby `prevfloat(2.0)`, kończymy.

Dla drugiego przypadku rozpoczynamy przeszukiwania od liczby `nextfloat(typemin(Float64))` i kończymy w `prevfloat(typemax(Float64))`.

4.3 Wyniki i interpretacja

Dla pierwszego przypadku znaleziona anomalia to:

```
x = 1.000000057228997
bits(x) = 0011111111110000000000000000000000000111101011100101111100101010
```

Anomalia ta wynika ze sposobu reprezentacji liczb zmiennopozycyjnych. Przez skończoną liczbę bitów przeznaczonych na zapis mantysy wyniki nie mogą być dokładnie reprezentowane i są zaokrąglane, co z kolei rzutuje na dokładność obliczeń.

Dla drugiego przypadku znaleziona anomalia wynosi:

```
x = -1.7976931348623157e308
```

4.4 Wnioski

Przy tworzeniu programów należy mieć na uwadze sposób reprezentacji liczb zmiennopozycyjnych. Podczas obliczeń na takich liczbach mogą pojawiać się błędy zaokrągleń, które mogą całkowicie zaburzyć wynik lub spowodować nieoczekiwane zachowanie programu. Przykładowo poprawne z matematycznego punktu widzenia wyrażenie $x * 1/x = 1$ nie musi być zawsze spełnione. Użycie takiego wyrażenia jako warunek instrukcji warunkowej może powodować trudne do wykrycia błędy.

5 Zadanie piąte

5.1 Opis problemu

Celem zadania była implementacja różnych algorytmów liczenia iloczynu skalarnego dwóch wektorów, a następnie określenie, który z nich zwraca wyniki najbliższe wartości dokładnej iloczynu skalarnego.

5.2 Rozwiązanie

Do wykonania zadania i określenia poprawności wyników zaimplementowałem (wyłączając zadane algorytmy) funkcje do liczenia błędu względnego, rzutowania typów wektorów oraz do rozbijania tablicy z wyliczonymi iloczynami na wartości nieujemne oraz ujemne.

5.3 Wyniki i interpretacja

Wyniki algorytmów				
Nazwa algorytmu	`Float32`	`Float64`	δ(`Float32`)	δ(`Float64`)
Algorytm 1	-0.4999443	1.0251881368296672e-10	4.9668057661282845e10	11.184955313981627
Algorytm 2	-0.4543457	-1.5643308870494366e-10	4.51379655800096e10	14.541186645165915
Algorytm 3	-0.5	0.0	4.967359135306107e10	1.0
Algorytm 4	-0.5	0.0	4.967359135306107e10	1.0

Poprawna wartość iloczynu skalarnego dla zadanych wektorów wynosi: $-1.00657107000000 * 10^{-11}$.

Dla arytmetyki `double` algorytmy 3 i 4 zwracają 0.0, co oznacza że wektory są prostopadłe. W wynikach widoczny jest wpływ kolejności obliczeń. Dodawanie do siebie dwóch oddalonych wartości powoduje powstawanie dużych błędów.

5.4 Wnioski

Poprzez zastosowanie różnych algorytmów, które sumują wartości w różnych kolejnościach można stwierdzić, że kolejność sumowania ma znaczenie. Faktycznie tak jest, gdyż następuje redukcja cyfr znaczących w przypadku dodawania do liczby dużej do małej.

6 Zadanie szóste

6.1 Opis problemu

Celem zadania było określenie, która z funkcji $f(x) = sqrt(x^2 + 1) - 1$, $g(x) = x^2/(sqrt(x^2 + 1) + 1)$ daje poprawne wyniki.

6.2 Rozwiązanie

Rozwiązanie zadania polegało na implementacji powyższych funkcji w języku `Julia`, a następnie porównaniu ich wyników dla kolejnych iteracji, gdzie $x = 8^i, i = 1...15$. Już piętnaście iteracji wystarczyło, aby wygenerować znaczne błędy.

6.3 Wyniki i interpretacja

Wartości funkcji		
i	wartość $f(8^i)$	wartość $g(8^i)$
1	0.0077822185373186414	0.0077822185373187065
2	0.00012206286282867573	0.00012206286282875901
3	1.9073468138230965e-6	1.907346813826566e-6
4	2.9802321943606103e-8	2.9802321943606116e-8
8	1.7763568394002505e-15	1.7763568394002489e-15
9	0.0	2.7755575615628914e-17
10	0.0	4.336808689942018e-19
15	0.0	4.0389678347315804e-28

W powyższej tabeli zostało przedstawione kilka iteracji wyliczania wartości funkcji. Jak widać od pewnego miejsca wartość funkcji $f(x)$ wynosi 0. Takie zaburzenie wartości funkcji wynika ze zjawiska utraty cyfr znaczących. Spowodowane to jest dla małych x odejmowaniem bliskich liczb, a tym samym zmniejszeniem liczby cyfr znaczących. Dla bardzo małych x

$$sqrt(x^2 + 1) \approx 1$$

co powoduje odjęcie liczby bliskiej 1.0 od 1.0.

6.4 Wnioski

Zjawisko utraty cyfr znaczących mogliśmy wyeliminować odpowiednio przekształcając funkcję $f(x)$. Aby uniknąć zjawiska utraty cyfr znaczących nie należy odejmować od siebie bliskich liczb lub sumować liczb bardzo odległych od siebie.

7 Zadanie siódme

7.1 Opis problemu

Celem zadania było obliczenie przybliżonej wartości pochodnej funkcji $f(x) = sin(x) + cos(3x)$ w punkcie $x_0 = 1.0$ obliczanej przy użyciu ilorazu różnicowego, a następnie porównania otrzymanych wartości dla zmniejszających się $h = 2^{-n}, n = 1...54$ z wartościami dokładnymi. Należało również zbadać zachowanie przybliżonej wartości pochodnej dla zmniejszających się h.

7.2 Rozwiązanie

Do rozwiązania zadania należało zaimplementować funkcję obliczającą przybliżoną wartość pochodnej funkcji w zależności od h. Żeby obliczyć błąd bezwzględny zaimplementowałem również funkcję obliczającą dokładną wartość pochodnej dla x, czyli: $f(x) = cos(x) - 3 * sin(3x)$.

7.3 Wyniki i interpretacja

Dokładną wartością pochodnej funkcji $f(x)$ w punkcie $x_0 = 1.0$ jest 0.11694228168853815.

Wartości funkcji		
n	approx $f'(x)$	błąd bezwzględny
0	2.0179892252685967	1.9010469435800585
1	1.8704413979316472	1.753499116243109
2	1.1077870952342974	0.9908448135457593
3	0.6232412792975817	0.5062989976090435
38	0.116943359375	1.0776864618478044e-6
39	0.11688232421875	5.9957469788152196e-5
40	0.1168212890625	0.0001209926260381522
52	-0.5	0.6169422816885382
53	0.0	0.11694228168853815
54	0.0	0.11694228168853815

Na początku wraz ze zmniejszaniem się h wartość pochodnej funkcji ze wzoru przybliżonego się poprawia względem wartości dokładnej. W okolicach iteracji 39 widać zaburzenie w wartości błędu bezwzględnego. Od tamtego miejsca wynik przybliżonej pochodnej funkcji pogarsza się zamiast poprawiać. Przy końcu iteracji wynik jest natomiast całkowicie błędny. Spowodowane jest to utratą cyfr znaczących. Za redukcję cyfr znaczących odpowiada wartość $1 + h$. W przypadku małych h następuje dodanie wartości bardzo małej do 1.0, co powoduje powstawanie błędów.

7.4 Wnioski

Ustalenie zbyt małego h do wyliczania przybliżonej wartości pochodnej funkcji spowoduje redukcję cyfr znaczących przy obliczaniu $1 + h$. To natomiast wniesie duże błędy do wyniku. Przy obliczaniu wartości pochodnej w punkcie z ilorazu różnicowego należy zatem dobrać odpowiednio małe h, aby uzyskać dobrze przybliżony wynik, jednak na tyle duże, aby nie spowodować zaistnienia zjawiska redukcji cyfr znaczących.