Sprawozdanie z listy 1

Michał Gancarczyk 244923

Streszczenie

Celem doświadczeń przeprowadzonych w ramach listy zadań jest przeanalizowanie właściwości specyficznych dla arytmetyki float.

Sprawozdanie poświęca jedną sekcję każdemu z siedmiu zadań. W skład każdej sekcji wchodzą kolejno: krótki opis przedstawionego w zadaniu problemu, schemat działania użytego algorymu, wyniki uzyskane przy użyciu tego algorytmu (często porównane z danymi kontolnymi), oraz wypływające z doświadczenia wnioski.

Do rozwiązania zadań użyty został język Julia, wersja 1.2.0.

1 Zadanie 1

1.1 Epsilony maszynowe

Epsilon maszynowy stanowi najmniejsza liczba $macheps > 0$ taka, że $fl(1.0 + macheps) > 1.0$.

Algorytm, którego wynikiem powinien być epsilon maszynowy dla danego typu float polega na ustawieniu zmiennej x jako jeden, a następnie dzieleniu zmiennej x przez dwa do momentu, gdy wartość wyrażenia $1+x$ zacznie wynosić jeden. Najmniejsza wyznaczona w ten sposób zmienna x zwracana jest jako wynik algorytmu.

Poniższa tabela prezentuje wyniki opisanego powyżej algorytmu, porównane z wartościami zwróconymi przez funkcję $eps()$ języka Julia, oraz wartości pobrane z pliku float.h.

Epsilon			
Typ	**Program**	**eps()**	**float.h**
Float16	0.000977	0.000977	N/A
Float32	1.1920929e-7	1.1920929e-7	1.1920929e-7
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460492503131e-16

Identyczne wartości zaprezentowe w poszczególnych wierszach dowodzą poprawności podanego algorymu. Liczba $macheps$ ma ścisły związek z precyzją arytmetyki. Analizując wzory opisujące $macheps$ (2^{-t}), oraz precyzję arytmetyki (2^{-t-1}), gdzie t stanowi długość mantysy, zauważyć można, że epsilon maszynowy jest dokładnie dwa razy dłuższy od przecyzji arymetyki.

Ponadto, opisane w tabeli wartości pokazują, że im większa jest precyzja danej arytmetyki float, tym mniejszy będzie jej epsilon maszynowy.

1.2 Liczba eta

Liczba eta określona jest jako najmniejsza dodatnia liczba, możliwa do zapisania w zmiennej typu float.

Zgodnie z podaną w poleceniu wskazówką wartości liczby eta dla kolejnych typów float wyznaczone zostały w pętli, gdzie zmienna x o początkowej wartości jeden była dzielona przez dwa dopóki jej wartość nie wyniosła zero. Najmniejsza niezerowa wartość stanowi wynik algorytmu.

Wyniki działania narzuconego zadaniem algorytmu porównane z wartości zwróconymi przez wyrażenie $nextfloat(TP(0.0))$, gdzie TP określa typ zmiennej, zostały zaprezentowane w poniższej tabelce.

Eta		
Typ	**Program**	**nextfloat()**
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

O poprawności zawartej w zadaniu metody świadczy równość liczb w poszczególnych wierszach powyższej tabeli.

Poprawność algorytmu potwierdza również bitowy zapis liczby eta 00000000000000000000000000000001, którego mantysa, przy cesze równej zero, przyjmuje najmniejszą możliwą niezerową wartość. Widniejąca w zapisie zerowa cecha świadczy również o tym, że eta w arytmetyce float jest liczbą zdenormalizowaną. Najmniejszą znormalizowaną liczbę w arytmetyce float otrzymać można używając funkcji $floatmin()$.

1.3 Liczba max

Liczba MAX dla konkretnego typu float uznana jest jako największa, niebędąca nieskończonością, liczba, którą można poprawnie zapisać w zmiennej danego typu.

Zastosowany w rozwiązaniu algorytm, pozwalający ustalić liczbę MAX dla typu TP, opiera się na dwóch istotnych krokach. Krok pierwszy polega na określeniu największej wielokrotności licby dwa, która zostanie poprawnie zapisana w TP. Realizowane jest to pętlą, w której zmienna x mnożona jest przez dwa do momentu osiągnięcia nieskończoności. Drugi krok algorytmu operuje na dodatkowym akumulatorze s, o początkowej wartości zero. Do akumulatora dodawane są coraz mniejsze wielokrotności liczby dwa (wyznaczane przez dzielenie x na pół) począwszy od wartości osiągnietej w kroku pierwszym, kończąc bespośrednio przed uzyskaniem nieskończoności w akumulatorze. Uzyskana w ten sposób wartość zwracana jest jako wynik algorytmu.

Przedstawiona poniżej tabela prezentuje wyniki użytego algorytmu, wartości zwracane przez funkcję $floatmax()$ języka Julia, oraz wartości odczytane z pliku float.h.

Max			
Typ	**Program**	**floatmax()**	**float.h**
Float16	6.55e4	6.55e4	N/A
Float32	3.4028235e38	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e308

Wartości w kolejnych wierszach tabeli świadczą o poprawności algorytmu.

2 Zadanie 2

Celem zadania jest (podobnie jak w podpunkcie pierwszym pierwszego zadania) ustalenie epsilonu maszynowego, dla wszystkich dostępnych typów float.

W odróżnieniu od pierwszego zadania opisany w poleceniu algorytm wyprowadzenia epsilona polega na obliczeniu wyrażenia $3 \cdot (\frac{4}{3} - 1) - 1$ w zmiennopozycyjnej arytmetyce każdego z typów.

Wyniki algorymu porównane z wynikami funkcji *eps*() języka Julia zostały przedstawione w poniższej tabelce.

Macheps		
Typ	**Program**	**eps()**
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Chociaż między liczbami w pierwszym i trzecim wierszu tabeli widoczne są różnice, wartości bezwzględne tych liczb są sobie równe. W arytmetyce float epsilon maszynowy służy określeniu przedziału liczb zaokrąglanych do zera, rozpoczętego ujemnym, a zakończonego dodatnim epsilonem. Ze względu na ten fakt wartości uzyskane przy użyciu algorytmu są akceptowalne. Powodem otrzymania w wyniku liczb ujemnych jest specyfika działania algorytmu - do ustalenia epsilona używa on liczby 4/3, której w arytmetyce float nie można bezstratnie zapisać, wymuszając w ten sposób błąd zaokrąglenia. Zgodnie z zasadą “round to even” liczba ta zaokrąglona zostaje z niedomiarem, co podczas kolejnych wykonanych operacji odejmowania skutkować może zmianą znaku wyniku.

3 Zadanie 3

Rozmieszczenie liczb w arytmetyce **Float64**, przedziale $[1, 2]$, i kroku $\delta = 2^{-52}$ można określić równomiernym, jeśli każda liczba zmiennopozycyjna x pomiędzy 1 i 2 może być przedstawiona za pośrednictwem wzoru $x = 1 + k\delta$, gdzie $k = 1, 2, ..., 2^{52} - 1$, a δ jest wcześniej wymienionym krokiem.

Aby zweryfikować, czy rozmieszczenie liczb dla opisanych wyżej parametrów jest równomierne zastosowany został następujący algorytm: Iterator pętli początkowo ustawiony został na wartość min odpowiadającą początkowi weryfikowanego przedziału. Następnie sprawdzane jest, czy wynik wyrażenia $nextfloat(x)$ jest równy wynikowi wyrażenia $min + i * delta$, a niezgodność wyników powoduje zakończenie algorytmu z niepowodzeniem. Ostatnim krokiem pętli jest przypisanie do zmiennej x wartości $nextfloat(x)$ i zakończenie algorymu sukcesem w przypadku osiągnięcia przez x wartości terminalnej max (stanowiącej koniec przedziału).

Zaprezentowane poniżej tabele prezentują jedynie dziesięć pierwszych liczb w przedziale.

Liczby w przedziale $[1, 2]$**,** $\delta = 2^{-52}$	
substr(n)	**n**
0 01111111111 0000000000000000000000000000000000000000000000000001	1.0000000000000002
0 01111111111 0000000000000000000000000000000000000000000000000010	1.0000000000000004
0 01111111111 0000000000000000000000000000000000000000000000000011	1.0000000000000007
0 01111111111 0000000000000000000000000000000000000000000000000100	1.0000000000000009
0 01111111111 0000000000000000000000000000000000000000000000000101	1.000000000000001
0 01111111111 0000000000000000000000000000000000000000000000000110	1.0000000000000013
0 01111111111 0000000000000000000000000000000000000000000000000111	1.0000000000000016
0 01111111111 0000000000000000000000000000000000000000000000001000	1.0000000000000018
0 01111111111 0000000000000000000000000000000000000000000000001001	1.000000000000002
0 01111111111 0000000000000000000000000000000000000000000000001010	1.0000000000000022

Algorytm potwierdził równomierność rozłożenia liczb dla zadanego przedziału i kroku. Jego poprawność można dodatkowo udowodnić obserwując zmiany zapisu binarnego liczb w kolejnych wierszach tabeli: bity znaku i cechy nie zmieniają się, a wartość mantysy zwiększa się dokładnie o 1 co każdą iterację.

Fakt ten został wykorzystany, aby określić wartości δ dla przedziałów $[\frac{1}{2}, 1]$ i $[2, 4]$: kolejno 2^{-53}, oraz 2^{-51}.

Liczby w przedziale $[\frac{1}{2}, 1]$**,** $\delta = 2^{-53}$	
substr(n)	**n**
0 0111111111 00000000000000000000000000000000000000000000000000001	0.5000000000000001
0 0111111111 00000000000000000000000000000000000000000000000000010	0.5000000000000002
0 0111111111 00000000000000000000000000000000000000000000000000011	0.5000000000000003
0 0111111111 00000000000000000000000000000000000000000000000000100	0.5000000000000004
0 0111111111 00000000000000000000000000000000000000000000000000101	0.5000000000000006
0 0111111111 00000000000000000000000000000000000000000000000000110	0.5000000000000007
0 0111111111 00000000000000000000000000000000000000000000000000111	0.5000000000000008
0 0111111111 00000000000000000000000000000000000000000000000001000	0.5000000000000009
0 0111111111 00000000000000000000000000000000000000000000000001001	0.500000000000001
0 0111111111 00000000000000000000000000000000000000000000000001010	0.5000000000000011

Liczby w przedziale $[2, 4]$**,** $\delta = 2^{-51}$	
substr(n)	**n**
0 1000000000 0000000000000000000000000000000000000000000000000001	2.0000000000000004
0 1000000000 0000000000000000000000000000000000000000000000000010	2.000000000000001
0 1000000000 0000000000000000000000000000000000000000000000000011	2.0000000000000013
0 1000000000 0000000000000000000000000000000000000000000000000100	2.0000000000000018
0 1000000000 0000000000000000000000000000000000000000000000000101	2.000000000000002
0 1000000000 0000000000000000000000000000000000000000000000000110	2.0000000000000027
0 1000000000 0000000000000000000000000000000000000000000000000111	2.000000000000003
0 1000000000 0000000000000000000000000000000000000000000000001000	2.0000000000000036
0 1000000000 0000000000000000000000000000000000000000000000001001	2.000000000000004
0 1000000000 0000000000000000000000000000000000000000000000001010	2.0000000000000044

We wszystkich przypadkach ilość liczb w danym przedziale wynosiła dokładnie 2^{52}. Zaprezentowane powyżej dane pokazują, że w arytmetyce float ze wzrostem bezwzględnych wartości liczb wiąże się zmniejszenie ich gęstości.

4 Zadanie 4

Celem zadania jest znalezienie w arytmetyce **Float64** liczby x, dla której wartością wyrażenia $x \cdot \frac{1}{x}$ nie będzie liczba jeden.

Algorytm wykorzystany do ustalenia liczb spełniających powyższy warunek wykorzystuje pętlę, w której kolejne wartości zmiennej x w zadanym przedziale (ustalane przy użyciu funkcji $nexfloat()$) sprawdzane są pod kątem spełnienia warunku $x \cdot \frac{1}{x} = 1$.

Najmniejsza uzyskana w ten sposób liczba to 1.000000057228997 (wyrażenie $x \cdot \frac{1}{x}$ wynosi w jej przypadku 0.9999999999999999).

Podczas wykonywania obliczeń w arytmetyce float błędy są nieuniknione. Można jednak starać się o ich zminimalizowanie przez zwiększenie precyzji arytmetyki dla istotnych obliczeń, lub w przypadku wyników obecnego doświadczenia zredukować wyrażenie do postaci 1, eliminując w ten sposób błędy wynikające z zaokrąglania liczby przy dzieleniu i mnożeniu.

5 Zadanie 5

Poleceniem w zadaniu jest obliczenie iloczynu skalarnego wektorów:

$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$

$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$

na cztery różne sposoby:

5.1 Algorytm A

Metoda "w przód" polega na zsumowaniu ze sobą iloczynów $x_i \cdot y_i$, gdzie i inkrementowane jest od 1 do n, a n stanowi wielkość obu wektorów.

5.2 Algorytm B

Metoda "w tył" polega na zsumowaniu ze sobą iloczynów $x_i \cdot y_i$, gdzie i dekrementowane jest od n do 1, a n stanowi wielkość obu wektorów.

5.3 Algorytm C

Opisana w poleceniu metoda polega na pomnożeniu par liczb podobnie jak we wcześniejszych podpunktach i zapisaniu rezultatu do nowego wektora z. Wektor z jest następnie dzielony na część złożoną z liczb ujemnych, oraz część zawierającą pozostałe liczby nieujemne. Otrzymane w ten sposób części sortowane są od największego do najmniejszego, względem wartości bezwzględnych składowych liczb. Jako ostatni krok algorytmu posortowane części są osobno sumowane przy użyciu dwóch akumulatorów, a powstałe w ten sposób sumy częściowe dodawane są do siebie, tworząc w ten sposób wynik całego algorytmu.

5.4 Algorytm D

Ostatni z opisanych w zadaniu algorytmów działa w sposób analogiczny do poprzednika. Jedyną różnicą jest zmiana kolejności sortowania z malejącej na rosnącą.

Poniżej przedstawione są wyniki wszystkich algorytmów.

Wyniki					
Typ	**Prawidłowa wartość**	**A**	**B**	**C**	**D**
Float32	$-1.00657107000000 \cdot 10^{-11}$	-0.4999443	-0.4543457	-0.5	-0.5
Float64	$-1.00657107000000 \cdot 10^{-11}$	1.0251881368296672e-10	-1.5643308870494366e-10	0.0	0.0

Powyższe wyniki prezentują znaczący problem arytmetyki zmiennoprzecinkowej - przy dużej ilości wykonanych na liczbach działań, rozwiązanie zawierać może znaczący błąd. Błąd dla arytmetyki **Float64** jest jednak mniejszy niż ten zaobserwowany przy użyciu **Float32**, co skłania do zwiększenia precyzji arytmetyki w krytycznych sekcjach obliczeniowych programu.

Powodem uzyskania dużych błędów względnych w wynikach są również paramery wejściowe. Podane w zadaniu wektory są prawie ortogonalne, co znacznie redukuje poprawność wyników przy oblicszniu ich iloczynu skalarnego.

6 Zadanie 6

Zgodzie z treścią zadania porównane zostały wyniki funkcjii $f(x) = \sqrt{x^2+1} - 1$, oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla argumentów $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$ Wszystkie operacje przeprowadzono na liczbach typu **Float64**.

Poniża tabela prezentuje wyżej wymienione wyniki.

Wyniki		
x	**f(x)**	**g(x)**
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	1.9073468138230965e-6	1.907346813826566e-6
8^{-4}	2.9802321943606103e-8	2.9802321943606116e-8
8^{-5}	4.656612873077393e-10	4.6566128719931904e-10
8^{-6}	7.275957614183426e-12	7.275957614156956e-12
8^{-7}	1.1368683772161603e-13	1.1368683772160957e-13
8^{-8}	1.7763568394002505e-15	1.7763568394002489e-15
8^{-9}	0.0	2.7755575615628914e-17
8^{-10}	0.0	4.336808689942018e-19

Wstępna analiza wzorów obu funkcji, oraz przekazywanych im parametrów sugeruje, że wynikami powinny być liczby dążące do 0, jednak nigdy go nie osiągające. Pomimo podobieństwa wyników obu funkcji dla początkowych wartości x, funkcja $f(x)$ dla $x = 8^{-9}$, jak i każdej kolejnej wartości x osiąga wartość 0. Przyczyną zaobserwowanej anomalii jest odejmowanie od siebie bardzo bliskich sobie liczb podczas obliczania $f(x)$. Wyniki funkcji $g(x)$ są dużo bardziej zbliżone do prawdziwych. Osiąga ona wartość 0 dopiero dla $x = 8^{-179}$.

7 Zadanie 7

Celem zadania jest użycie wzoru na przybliżoną wartość pochodnej $f'(x_0)$ $f\ '(x_0) = \frac{f(x_0+h)-f(x_0)}{h}$ do określenia w arytmetyce **Float64** wartości pochodnej funkcji $f(x) = sinx + cos3x$ przy $x_0 = 1$ i $h = 2^{-0}, 2^{-1}, ...$

Wyniki			
h	$\widetilde{f}'(1)$	$\lvert f'(1) - \widetilde{f}'(1)\rvert$	$h+1$
2^{-0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
2^{-2}	1.1077870952342974	0.9908448135457593	1.25
2^{-3}	0.6232412792975817	0.5062989976090435	1.125
2^{-4}	0.3704000662035192	0.253457784514981	1.0625
2^{-5}	0.24344307439754687	0.1265007927090087	1.03125
2^{-6}	0.18009756330732785	0.0631552816187897	1.015625
2^{-7}	0.1484913953710958	0.03154911368255764	1.0078125
2^{-8}	0.1327091142805159	0.015766832591977753	1.00390625
2^{-9}	0.1248236929407085	0.007881411252170345	1.001953125
2^{-10}	0.12088247681106168	0.0039401951225235265	1.0009765625
2^{-11}	0.11891225046883847	0.001969968780300313	1.00048828125
2^{-12}	0.11792723373901026	0.0009849520504721099	1.000244140625
2^{-13}	0.11743474961076572	0.0004924679222275685	1.0001220703125
2^{-14}	0.11718851362093119	0.0002462319323930373	1.00006103515625
2^{-15}	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-16}	0.11700383928837255	6.155759983439424e-5	1.0000152587890625
2^{-17}	0.11697306045971345	3.077877117529937e-5	1.0000076293945312
2^{-18}	0.11695767106721178	1.5389378673624776e-5	1.0000038146972656
2^{-19}	0.11694997636368498	7.694675146829866e-6	1.0000019073486328
2^{-20}	0.11694612901192158	3.8473233834324105e-6	1.0000009536743164
2^{-21}	0.1169442052487284	1.9235601902423127e-6	1.0000004768371582
2^{-22}	0.11694324295967817	9.612711400208696e-7	1.000000238418579
2^{-23}	0.11694276239722967	4.807086915192826e-7	1.0000001192092896
2^{-24}	0.11694252118468285	2.394961446938737e-7	1.0000000596046448
2^{-25}	0.116942398250103	1.1656156484463054e-7	1.0000000298023224
2^{-26}	0.11694233864545822	5.6956920069239914e-8	1.0000000149011612
2^{-27}	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
2^{-28}	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
2^{-29}	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
2^{-30}	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
2^{-31}	0.11694216728210449	1.1440643366000813e-7	1.0000000004656613
2^{-32}	0.11694192886352539	3.5282501276157063e-7	1.0000000002328306
2^{-33}	0.11694145202636719	8.296621709646956e-7	1.0000000001164153
2^{-34}	0.11694145202636719	8.296621709646956e-7	1.0000000000582077
2^{-35}	0.11693954467773438	2.7370108037771956e-6	1.0000000000291038
2^{-36}	0.116943359375	1.0776864618478044e-6	1.000000000014552
2^{-37}	0.1169281005859375	1.4181102600652196e-5	1.000000000007276
2^{-38}	0.116943359375	1.0776864618478044e-6	1.000000000003638
2^{-39}	0.11688232421875	5.9957469788152196e-5	1.000000000001819
2^{-40}	0.1168212890625	0.0001209926260381522	1.0000000000009095
2^{-41}	0.116943359375	1.0776864618478044e-6	1.0000000000004547
2^{-42}	0.11669921875	0.0002430629385381522	1.0000000000002274
2^{-43}	0.1162109375	0.0007313441885381522	1.0000000000001137
2^{-44}	0.1171875	0.0002452183114618478	1.0000000000000568
2^{-45}	0.11328125	0.003661031688538152	1.0000000000000284
2^{-46}	0.109375	0.007567281688538152	1.0000000000000142
2^{-47}	0.109375	0.007567281688538152	1.000000000000007
2^{-48}	0.09375	0.023192281688538152	1.0000000000000036
2^{-49}	0.125	0.008057718311461848	1.0000000000000018
2^{-50}	0.0	0.11694228168853815	1.0000000000000009
2^{-51}	0.0	0.11694228168853815	1.0000000000000004
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0
2^{-54}	0.0	0.11694228168853815	1.0

Analiza powyższych wyników pokazuje, że dla $n = 53$ wartość h stała się na tyle mała, że wynikiem wyrażenia $1+h$ jest 1. Efektem tego zaokrąglenia jest całkowite zaburzenie przybliżonych wyników pochodnej. Jest to skutek dodawania do siebie liczb o mocno różniących się wykładnikach.

Jak widać w kolejnych wierszach środkowej kolumny problemy z zaokrągleniami wynikającymi z różnic wykładników między h, a pozostałymi liczbami w wyrażeniach wystąpiły znacznie wcześniej. Obliczane w niej wyrażenie nie miało bowiem prostej postaci $1 + h$, ale wymagające wykonania większej ilości działań przybliżenie pochodnej.