Obliczenia naukowe

Sprawozdanie

Mateusz Laskowski

21.10.2018

1. **Zadanie 1**
   1. **Opis problemu**

Wyznaczyć iteracyjnie epsilon maszynowy, liczbę eta oraz liczbę MAX dla wszystkich dostępnych typów zmiennopozycyjnych **Float16, Float32, Float64** oraz porównać zwracane dane do odpowiednich funkcji w języku Julia (**eps(), nextfloat(), realmax()**).

* 1. **Rozwiązanie problemu**

**Wzory algorytmów**

* + 1. **Epsilon maszynowy**

macheps = 1.0

while (1.0 + macheps / 2.0 > 1.0)

macheps = macheps / 2.0

end

* + 1. **Liczba eta**

eta = 1.0

while (eta > 0.0)

if (eta / 2.0 == 0.0)

break

else

eta = eta / 2.0

end

end

* + 1. **Liczba MAX**

max = 2.0

while (isinf(max) = false)

if (isinf(max \* 2.0) == true)

break

else

max = max \* 2.0

end

end

* 1. **Wyniki**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Typ liczb** | **Epsilon maszynowy**  **iteracyjnie** | **Wynik funkcji eps()** | **Liczba eta iteracyjnie** | **Wynik funkcji nextfloat()** | **Liczba MAX iteracyjnie** | **Wynik funkcji realmax()** |
| Float16 | 0.00097656 | 0.000977 | 6.0e-8 | 6.0e-8 | 6.55e4 | 6.55e4 |
| Float32 | 1.1920929e-7 | 1.1920929e-7 | 1.0e-45 | 1.0e-45 | 3.4028235e38 | 3.4028235e38 |
| Float64 | 2.220446049250313e-16 | 2.220446049250313e-16 | 5.0e-324 | 5.0e-324 | 1.7976931348623157e308 | 1.7976931348623157e308 |

W pliku nagłówkowym float.h języka C zawarte są takie dane na temat liczb zmiennoprzecinkowych:

**Epsilon maszynowy**

**Float32:** 1.19209290e-07F

**Float64:** 2.2204460492503131e-16

**Liczba MAX**

**Float32:** 3.402823e+38

**Float64:** 1.797693e+308

1. **Zadanie 2**
   1. **Opis problemu**

Sprawdzić eksperymentalnie w języku Julia słuszność twierdzenia Kahan’a, które stwierdza, że epsilon maszynowy można wyznaczyć z pomocą poniżej podanego wzoru:

macheps =

* 1. **Rozwiązanie problemu**

Użycie powyższego wzoru dla typów zmiennopozycyjnych standardu **IEEE 754**

* 1. **Wyniki**

|  |  |
| --- | --- |
| **Typ zmiennopozycyjny** | **Epsilon maszynowy wg wzoru** |
| Float16 | -0.000977 |
| Float32 | 1.1920929e-7 |
| Float64 | -2.220446049250313e-16 |

* 1. **Wnioski**

Aktualne komputery nie są w stanie dokładnie przechowywać liczb rzeczywistych, gdzie w tym wypadku taką liczbą jest 4/3, ponieważ przechowuje te liczby

w systemie dwójkowym i dlatego musi zaokrąglić otrzymany wynik musi zaokrąglić

z pewną dokładnością co prowadzi do niedokładnego obliczenia.

1. **Zadanie 3**
   1. **Opis problemu**

Sprawdzić eksperymentalnie w języku Julia, czy liczby zmiennopozycyjne w arytmetyce **Float64** są równomiernie rozmieszczone w danych zakresach.

Zakresy: [1, 2], [1/2, 1], [2, 4].

* 1. **Rozwiązanie problemu**

**Wzór algorytmu**

p1 = 1.0 # początek zakresu

p2 = 2.0 # koniec zakresu

delta = nextfloat(p1) – p1 # odstęp między p1, a następnikiem p1

k = 1.0

x = 1.0

while (x <= b)

x = x + k \* delta

if (x – prevfloat(x) != delta)

print(x – prevfloat(x))

end

k = k + 1.0

end

Dany algorytm nie pokazywał gęstości danych przedziałów lecz różnicę pomiędzy deltami. Gęstość można było wyznaczyć za pomocą funkcji **bits(x)**, gdzie argumentem owej funkcji były właśnie początkowa i końcowa liczba z naszych zakresów.

* 1. **Wyniki**

julia> bits(Float64(1.0))

"0011111111110000000000000000000000000000000000000000000000000000"

julia> bits(Float64(2.0))

"0100000000000000000000000000000000000000000000000000000000000000"

julia> bits(Float64(0.5))

"0011111111100000000000000000000000000000000000000000000000000000"

julia> bits(Float64(1.0))

"0011111111110000000000000000000000000000000000000000000000000000"

julia> bits(Float64(2.0))

"0100000000000000000000000000000000000000000000000000000000000000"

julia> bits(Float64(4.0))

"0100000000010000000000000000000000000000000000000000000000000000"

* 1. **Wnioski**

Po eksperymentowaniu z funkcją **bits()**, doszedłem do takich oto wniosków:

|  |  |
| --- | --- |
| **Zakres** | **Gęstość liczb** |
| [1, 2] |  |
| [1/2, 1] |  |
| [2, 4] |  |

Gęstość liczb zmiennoprzecinkowych w arytmetyce **Float64 zmniejsza się** wraz ze wzrostem liczb, poprzez coraz to mniejszą precyzję.

1. **Zadanie 4**
   1. **Opis problemu**

Znaleźć eksperymentalnie w arytmetyce **Float64** zgodnej ze standardem **IEEE 754** liczbę zmiennopozycyjną ***x*** w przedziale 1 < *x* < 2 taką, że spełnia poniższe warunki:

tj.

Znajdź najmniejszą taką liczbę.

* 1. **Rozwiązanie problemu**

**Wzór algorytmu**

x = nextfloat(1.0)

eta = nextfloat(1.0) – 1.0

while (fl(x \* fl(1 / x) == 1

x = x+ eta

end

Aby znaleźć najmniejszą liczbę wystarczy pod x podstawić najmniejszą liczbę zmiennopozycyjną **Float64**, którą można wyznaczyć za pomocą funkcji **realmin()**.

* 1. **Wyniki**

Liczba zmiennopozycyjna *x* w przedziale 1 < *x* < 2 z wcześniej podanym warunkiem:

*x* = 1.000000057228997

Najmniejsza liczba

*x* = 2.225073985845947e-308

1. **Zadanie 5**
   1. **Opis problemu**

Napisać program w języku Julia realizujący następujący eksperyment obliczania iloczynu skalarnego dwóch wektorów:

*x =* [2.718281828, −3.141592654, 1.414213562, 0.5772156649, 0.3010299957]

*y* = [1486.2497, 878366.9879, −22.37492, 4773714.647, 0.000185049]

1. „w przód”
2. „w tył”
3. Suma wszystkich częściowych iloczynów skalarnych zaczynając od **największego**

do **najmniejszego** wyniku częściowego

1. Suma wszystkich częściowych iloczynów skalarnych zaczynając od **najmniejszego** do **największego** wyniku częściowego
   1. **Rozwiązanie problemu**

Zliczyłem wszystkie iloczyny skalarne podanych wektorów.

W algorytmach **(a)** i **(b)**  dodałem elementy tablicy z wynikami częściowych iloczynów skalarnych odpowiednio w kolejności w tablicy.

W algorytmie **(c) i (d)** posortowałem oraz zsumowałem elementy tablicy z wynikami częściowymi jak w opisie problemu **Ad.5.1**.

* 1. **Wyniki**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Typ liczb** | **„w przód”** | **„w tył”** | **(c)** | **(d)** |
| Float32 | -0.4999443 | -0.4543457 | -0.5 | -0.5 |
| Float64 | 1.0251881368296672e-10 | -1.5643308870494366e-10 | 0.0 | 0.0 |

* 1. **Wnioski**

Najdokładniejszy jest sposób „w tył”, ponieważ był najbliżej wyrażenia

– 11. Przy dodawaniu liczb od największej do najmniejszej i na odwrót występuje **zjawisko pochłonięcia** mniejszej liczby przez większą!

1. **Zadanie 6**
   1. **Opis problemu**

Policzyć w języku Julia w arytmetyce **Float64** dla wartości następujących funkcji:

* 1. **Rozwiązanie problemu**

**Wzór algorytmu**

i = 1

f = 0.0

g = 0.0

while (i < 180)

x = (1 / 8)^i

f = sqrt(x^2 + 1.0) – 1.0 # funkcja f(x)

g = x^2 / sqrt(x^2 + 1.0) + 1.0 # funkcja g(x)

print(f)

print(g)

i = i + 1

end

* 1. **Wyniki**

x = 1.250000e-01

f(x=8^-1) = 0.0077822185373186414

g(x=8^-1) = 0.0077822185373187065

x = 1.562500e-02

f(x=8^-2) = 0.00012206286282867573

g(x=8^-2) = 0.00012206286282875901

x = 1.953125e-03

f(x=8^-3) = 1.9073468138230965e-6

g(x=8^-3) = 1.907346813826566e-6

x = 2.441406e-04

f(x=8^-4) = 2.9802321943606103e-8

g(x=8^-4) = 2.9802321943606116e-8

x = 3.051758e-05

f(x=8^-5) = 4.656612873077393e-10

g(x=8^-5) = 4.6566128719931904e-10

x = 3.814697e-06

f(x=8^-6) = 7.275957614183426e-12

g(x=8^-6) = 7.275957614156956e-12

x = 4.768372e-07

f(x=8^-7) = 1.1368683772161603e-13

g(x=8^-7) = 1.1368683772160957e-13

x = 5.960464e-08

f(x=8^-8) = 1.7763568394002505e-15

g(x=8^-8) = 1.7763568394002489e-15

x = 7.450581e-09

f(x=8^-9) = 0.0

g(x=8^-9) = 2.7755575615628914e-17

x = 9.313226e-10

f(x=8^-10) = 0.0

g(x=8^-10) = 4.336808689942018e-19

x = 1.164153e-10

f(x=8^-11) = 0.0

g(x=8^-11) = 6.776263578034403e-21

…

x = 1.138052e-159

f(x=8^-176) = 0.0

g(x=8^-176) = 6.4758e-319

x = 1.422566e-160

f(x=8^-177) = 0.0

g(x=8^-177) = 1.012e-320

x = 1.778207e-161

f(x=8^-178) = 0.0

g(x=8^-178) = 1.6e-322

x = 2.222759e-162

f(x=8^-179) = 0.0

g(x=8^-179) = 0.0

* 1. **Wnioski**

Patrząc na wyniki **Ad.6.3**, można zauważyć, że funkcja f(x) jest mniej dokładna, ponieważ podczas obliczeń dochodzi do redukcji **cyfr znaczących**. Dużo lepiej sobie radzi funkcja g(x), gdzie dopiero przy ma wartość 0.0.

1. **Zadanie 7**
   1. **Opis problemu**

W języku Julia w arytmetyce **Float64** użyć wzoru na przybliżoną wartość pochodnej w punkcie oraz błędów

dla

Wzór do obliczania przybliżonej wartości pochodnej

* 1. **Rozwiązanie problemu**

**Funkcja pochodnaP(x, h)**

function pochodnaP(x, h)

return (sin(x + h) + cos(3.0 \* (x +h)) – sin(x) + cos(3.0 \* x)) / h

end

**Funkcja blad(x, w)**

function blad(x, w)

return abs((cos(x) – 3.0 \* cos(3.0 \* x)) – w)

end

**Wzór algorytmu**

n = 0

wynik = 0.0

while (n <= 54)

h = 1.0 / (2.0^n)

wynik = pochodnaP(1.0, h);

print(wynik)

print(blad(x, wynik))

n = n +1

end

* 1. **Wyniki**

h = 2^-0

h = 1.0 || ~f'(x) = 2.0179892252685967

1 + h = 2.0

f'(x) = 0.11694228168853815

Błąd = 1.9010469435800585

h = 2^-1

h = 0.5 || ~f'(x) = 1.0837422107583725

1 + h = 1.5

f'(x) = 0.11694228168853815

Błąd = 0.9667999290698344

h = 2^-2

h = 0.25 || ~f'(x) = 0.7225113091862211

1 + h = 1.25

f'(x) = 0.11694228168853815

…

Błąd = 3.3444071258271275e14

h = 2^-52

h = 2.220446049250313e-16 || ~f'(x) = 6.688814251654259e14

1 + h = 1.0000000000000002

f'(x) = 0.11694228168853815

Błąd = 6.688814251654258e14

h = 2^-53

h = 1.1102230246251565e-16 || ~f'(x) = 1.3377628503308518e15

1 + h = 1.0

f'(x) = 0.11694228168853815

Błąd = 1.3377628503308518e15

h = 2^-54

h = 5.551115123125783e-17 || ~f'(x) = 2.675525700661704e15

1 + h = 1.0

f'(x) = 0.11694228168853815

Błąd = 2.675525700661704e15

* 1. **Wnioski**

Błąd na samym początku przy przybliżeniu niestety nie został zniwelowany pomimo zmniejszającego się h. Jeżeli już raz pojawi się błąd w obliczeniach zmiennopozycyjnych to jest on zawarty do końca obliczeń. Można zauważyć, że przy 1 + h wartości znaczące zostały zaokrąglone, gdzie już przy samym końcu h zniwelowało do 0.