Obliczenia Naukowe- Lista 1

Szymon Skoczylas

25 października 2020

Zadanie 1

Cel zadania

Celem zadania było eksperymentalne znalezienie:

1. wartości epsilonów maszynowych (najmniejsza liczba $macheps > 0$ taka, że $macheps + 1.0 > 1.0$)
2. liczby maszynowe eta (najmniejsza liczba taka, że $eta > 0.0$)
3. liczby Max (czyli największa liczba mniejszą od ∞)

dla dostępnych typów zmiennoprzecinkowych $Float16$, $Float32$, $Float64$. Następnie wartości te zostały porównane z funkcjami bibliotecznymi języka $Julia$ ($eps()$, $nextfloat()$, $floatmax()$), oraz z wartością $macheps$ i MAX które znajdują się w pliku nagłówkowym $float.h$ języka C.

Epsilon Maszynowy

W celu znalezienia wartości $macheps$ w języku $Julia$ został użyty następujący algorytm:

```
function find_macheps(Type)
    value = Type(1.0)

    while Type(1.0) + value / Type(2.0) > Type(1.0)
        value = value / Type(2.0)

    end

    return value
end
```

Działa on następująco: w pętli dzieli $value$ przez 2, aż szukana liczba będzie spełniałą warunek $macheps + 1.0 > 1.0$.

Wyniki otzymane tą metodą, dla danych arytmetyk (kolejno: wyliczona, zwrócona przez funkcję $eps()$ i z pliku $float.h$):

1. $Float64$- wyliczona: $2.220446049250313e-16$, $eps()$: $2.220446049250313e-16$, $float.h$: $2.2204460493e-16$

2. $Float32$- wyliczona: $1.1920929e-7$, $eps()$: $1.1920929e-7$, $float.h$: $1.1920928955e-07$

3. $Float16$- wyliczona: 0.000977, $eps()$: 0.000977, $float.h$: brak

Liczba Eta

Aby znaleść liczbę eta został użyty następujący algorytm:

```
function find_eta(Type)
    value = Type(1.0)

    while value / Type(2.0) != Type(0.0)
        value = value / Type(2.0)

    end

    return value
end
```

Działa w następujący sposób: liczba 1.0 jest dzielona przez 2.0 do momentu, aż otrzymamy ostatnią liczbę która spełnia warunek $eta > 0.0$.

Wyniki otrzymane w ten sposób (kolejno wyliczona i zwrócona przez $nextfloat()$):

1. $Float64$- wyliczona: $5.0e-324$, $nextfloat()$: $5.0e-324$

2. $Float32$- wyliczona: $1.0e-45$, $nextfloat()$: $1.0e-45$

3. $Float16$- wyliczona: $6.0e-8$, $nextfloat()$: $6.0e-8$

Liczba MAX

Pokazany niżej algorytm został użyty w celu znaleziania liczby MAX:

```
function find_max(Type)
    max = prevfloat(Type(1.0))

    while !isinf(max*Type(2.0))
        max = max * Type(2.0)

    end

    return max
end
```

Algorytm monoży liczbę $prevfloat(1.0)$ (czyli największą liczbę mniejszą od 1, która posiada mantysę składającą się z samych jedynek) przez 2.0, dopóki nie osiągnie wartości maksymalnej dla danego typu.

Funkcja $!isinf$ użyta jako warunek sprawdzający czy zostanie osiągnięta wartość maksymalna dla danego typu (nieskończoność).

Wartości liczby MAX (kolejno: uzyskane powyższą metodą, zwrócona przez $floatmax()$ i z pliku $float.h$):

1. $Float64$- wyliczona: $1.7976931348623157e308$, $floatmax()$: $1.7976931348623157e308$, $float.h$: $1.7976931348623157e308$
2. $Float32$- wyliczona: $3.4028235e38$, $floatmax()$: $3.4028235e38$, $float.h$: $3.4028234664e38$
3. $Float16$- wyliczona: $6.55e4$, $floatmax()$: $6.55e4$, $float.h$: brak

Wnioski

Porwónując wyniki zastosowanych algorytmów z wartościami zwracanymi przez funkcje biblioteczne można stwierdzić, że algorytmy są poprawne.

Zadanie 2

Cel zadania

Sprawdzić eksperymentalnie w języku $Julia$ poprawność stwierdzenia Kahana. Stwierdzenie to przedstawai się następująco: epsilon paszynowy można otrzymać obliczając wyrażenie:

$$3\left(\frac{4}{3}-1\right)-1$$

Wykonanie i wyniki

Kod użyty do sprawdzenia wyżej sformułowanego stwierdzenia:

```
type = Float64
println("eps() for Float64: ", eps(type))
println("Kahan eps for Float64: ", type(3) * (type(4)/type(3) - type(1)) - type(1)  , "\n")

type = Float32
println("eps() for Float32: ", eps(type))
println("Kahan eps for Float32: ", type(3) * (type(4)/type(3) - type(1)) - type(1)  , "\n")

type = Float16
println("eps() for Float16: ", eps(type))
println("Kahan eps for Float16: ", type(3) * (type(4)/type(3) - type(1)) - type(1)  , "\n")
```

Kod zaprezentowany wyżej daje następujące wyniki:

```
[szymo@szymoHost lista_1]$ julia zad2.jl
eps() for Float64: 2.220446049250313e-16
Kahan eps for Float64: -2.220446049250313e-16

eps() for Float32: 1.1920929e-7
Kahan eps for Float32: 1.1920929e-7

eps() for Float16: 0.000977
Kahan eps for Float16: -0.000977
```

Wnioski

Jak można zauważyć w wyżej zaprezentowancyh wynikach, wyliczenie wartości wyrażenia $3\left(\frac{4}{3}-1\right)-1$ daje poprawne wyniki z dokładnością do znaku (arytmetyki $Float32$ i $Float64$). Błąd ten spowodowany jest reprezentacją liczby $\frac{4}{3}$, która jest okresowa, przez co zastosowane jest zaokrąglenie.

Zadanie 3

Cel zadania

Eksperymentalnie sprawdzić, że w arytmetyce $Float64$ liczby są równomiernie rozmieszczone w przedziale $[1, 2]$, z krokiem $\delta = 2^{-52}$. Sprawdzić jak rozmieszczone są liczby w przedziałach $\left[\frac{1}{2}, 1\right]$ i $[2, 4]$, oraz jak mogą być przedstawione dla danego przedziału.

Wykonanie i wyniki

W celu sprawdzenia rozmieszczenia liczb w zadanych przedziałach użyta została funkcja *bitstring*(), która zwraca reprezentację binarną danej liczby. Jeśli liczby liczby w danym przedziale są rozmieszczone z krokiem δ to dodanie do liczby z danego zakresu δ powinno zwiększać mantysę o 1. Poniższy kod został zastosowany do sprawdzenia tego stwierdzenia:

```
function check_density(start, delta)
    for i in 1:8
        println(bitstring(start+i*delta))
    end
end

println("[1,2]")
check_density(1, 2.0^(-52))

println("[0.5, 1]")
check_density(2, 2.0^(-51))

println("[2,4]")
check_density(0.5, 2.0^(-53))
```

Wyniki uzyskane po wywołaniu powyższego algorytmu:

```
[szymo@szymoHost lista_1]$ julia zad3.jl
[1,2]
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000000111
0011111111110000000000000000000000000000000000000000000000001000
```

```
[0.5, 1]
0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
0100000000000000000000000000000000000000000000000000000000000101
0100000000000000000000000000000000000000000000000000000000000110
0100000000000000000000000000000000000000000000000000000000000111
0100000000000000000000000000000000000000000000000000000000001000
[2,4]
0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011
0011111111100000000000000000000000000000000000000000000000000100
0011111111100000000000000000000000000000000000000000000000000101
0011111111100000000000000000000000000000000000000000000000000110
0011111111100000000000000000000000000000000000000000000000000111
0011111111100000000000000000000000000000000000000000000000001000
```

Wnioski

Z powyższego przykładu można wywnioskować, że δ dla zadanych przedziałów wynosi:

1. $[1, 2]$, $\delta = 2^{-52}$
2. $\left[\frac{1}{2}, 1\right]$, $\delta = 2^{-51}$
3. $[2, 4]$, $\delta = 2^{-53}$

W każdym z przedziałów jest tyle samo liczb, lecz ich gęstość jest różna. Jako, że liczby w zakresach $[2, 4]$, $[0.5, 1]$ są kolejno dwa razy większe bądź mniejsze od $[1, 2]$ to krok δ też musi ulec zmianie. Z uzyskanych danych wynika, że oddalając się od 0 krok pomiędzy poszczególnymi liczbami również się zwiększa. Co spowoduje mniejszą dokładność obliczeń wraz ze wzrostem cechy.

Zadanie 4

Cel zadania

Eksperymentalne znalezienie liczby x w arytmetyce $Float64$ taką, że:

1. $1 < x < 2$
2. $x\left(\frac{1}{x}\right) \neq 1$

Oraz znalezienie najmniejszego x spełniającego powyższe warunki.

Wykonanie i wyniki

Liczba x została wyznaczona zaczynając od 1 i 0 (najmniejsze x) za pomocą poniższego kodu:

```
let num = Float64(1.0)

    while nextfloat(num) * (Float64(1.0) / nextfloat(num)) == Float64(1.0)
        num = nextfloat(num)
    end

    println(nextfloat(num))

    num = Float64(0.0)

    while nextfloat(num) * (Float64(1.0) / nextfloat(num)) == Float64(1.0)
        num = nextfloat(num)
    end

    println(nextfloat(num))

end
```

Wyniki uzyskane za pomocą powyższego algorytmu:

1. Zaczynając od 1: 1.000000057228997
2. Zaczynając od 0: $5.0e-324$

Wnioski

W tym zadaniu widać jak niedokładność reprezentacji liczb zmiennoprzecinkowych w pamięci komputera może prowadzić do błędów obliczeniowych i fałszywych wyników.

Zadanie 5

Cel zadania

Obliczenie iloczynu skalarnego dwóch wektorów:

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

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

używając trzech różnych sposobów:

1. sumowanie kolejnych iloczynów współrzędnych wektorów od początku do końca
2. sumowanie kolejnych iloczynów współrzędnych wektorów od końca do początku
3. sumowenie iloczynów współrzędnych od największego do najmniejszego (najpierw w określonej kolejności dodajemy ujemne do ujemnych, dodatnie do dodatnich, a następnie te dwie sumy częściowe do siebie)
4. podobnie jak w poprzednim punkcie, ale od najmniejszego do największego

Wykonanie i wyniki

Algorytmy wyspecyfikowane w poleceniu zostały w następujący sposób:

```
function alg_a(Type)
    sum = 0

    for i in 1:length(x)
        sum += Type(x[i]) * Type(y[i])
    end

    return sum
end

function alg_b(Type)
    sum = 0

    for i in length(x):-1:1
        sum += Type(x[i]) * Type(y[i])
    end

    return sum
end
```

```
function alg_c(Type)
  greater_than_0 = Type[]
  less_eq_0 = Type[]

  for i in 1:length(x)
    product = Type(x[i]) * Type(y[i])

    if product > 0
      append!(greater_than_0, product)

    else
      append!(less_eq_0, product)

    end

  end

  sort(less_eq_0)
  sort(greater_than_0, rev=true)

  return sum(greater_than_0) + sum(less_eq_0)
end

function alg_d(Type)
  greater_than_0 = Type[]
  less_eg_0 = Type[]

  for i in 1:length(x)
    product = Type(x[i]) * Type(y[i])

    if product > 0
        append!(greater_than_0, product)

    else

        append!(less_eg_0, product)
    end
  end
```

Za pomocą powyższego kodu uzyskano następujące wyniki (po kolei dla algorytmów podanych w zadaniu):

```
[szymo@szymoHost lista_1]$ julia zad5.jl
Float64:
1.0251881368296672e-10
-1.5643308870494366e-10
0.0
0.0

Float62
-0.4999443
-0.4543457
-0.5
-0.5
```

Wnioski

Na podstawie tego zadania możemy zaobserwować, że zarówno sposób przeprowadzania obliczeń (np. kolejność działań) jak i sposób reprezentacji liczb w komputerze ma wpływ na dokładność obliczeń.

Zadanie 6

Cele zadania

Policzenie wartości dwóch funkcji:

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$.

Wykonanie i wyniki

Funkcje będą realizowane za pomocą następującego kodu:

```
function f(x)
    sqrt(x ^ 2 + 1) - 1
end

function g(x)
    x ^ 2 / (sqrt(x ^ 2 + 1) + 1)
end
```

Dziesięć pierwszych wartości zwróconych przez każdą z tych funkcji:

```
[szymo@szymoHost lista_1]$ julia zad6.jl
1: f(x)=0.0077822185373186414 g(x)=0.0077822185373187065
2: f(x)=0.00012206286282867573 g(x)=0.00012206286282875901
3: f(x)=1.9073468138230965e-6 g(x)=1.907346813826566e-6
4: f(x)=2.9802321943606103e-8 g(x)=2.9802321943606116e-8
5: f(x)=4.656612873077393e-10 g(x)=4.6566128719931904e-10
6: f(x)=7.275957614183426e-12 g(x)=7.275957614156956e-12
7: f(x)=1.1368683772161603e-13 g(x)=1.1368683772160957e-13
8: f(x)=1.7763568394002505e-15 g(x)=1.7763568394002489e-15
9: f(x)=0.0 g(x)=2.7755575615628914e-17
10: f(x)=0.0 g(x)=4.336808689942018e-19
```

Wnioski

Pomimo tożsamości obu funkcji możemy zauważyć różnicę w ich wartościach zwracanych przez program. Gdy $x \geqslant 10$ funkcja $f(x)$ zwraca 0, podczas gdy funkcja $g(x)$ cały czas zwraca, bardzo małe, ale niezerowe zwrtości. Wywnioskować więc można, że funkcja ta w postaci $g(x)$ jest bardziej dokładna. Wynika to z tego, że w $f(x)$ mamy odejmowanie dwóch bliskich sobie liczb, którego nie uświadczymy w $g(x)$.

Zadanie 7

Cele zadania

Obliczyć przybliżoną wartość pochodnej funkcji

$$f(x) = sinx + cos3x$$

za pomocą:

$$f'(x_0) \approx \tilde{f}'(x) = \frac{f(x_0 + h) - f(x_0)}{h}$$

w punkcie $x_0 = 1$. A następnie policzyć błąd $|f'(x) - \tilde{f}'(x)|$, dla $h = 2^{-n}$, gdzie $n = 0, 1, 2, \ldots, 54$

Wykonanie i wyniki

Aby zrealizować wymienione wyżej funkcje zostanie użyty poniższy kod:

```
function derivative_at_point(func, point, h)
    return (func(point + h) - func(point)) / h
end

function real_derivative(x)
    return cos(x) - 3*sin(3*x)
end
```

Przedstawione funkcje zwracają następujące wartości (kolejno wartość h, $\tilde{f}'(x)$ i błąd przybliżenia):

```
[szymo@szymoHost lista_1]$ julia zad7.jl
2^-1: 1.5 error: 1.753499116243109
2^-2: 1.25 error: 0.9908448135457593
2^-3: 1.125 error: 0.5062989976090435
2^-4: 1.0625 error: 0.253457784514981
2^-5: 1.03125 error: 0.1265007927090087
2^-6: 1.015625 error: 0.0631552816187897
2^-7: 1.0078125 error: 0.03154911368255764
2^-8: 1.00390625 error: 0.015766832591977753
2^-9: 1.001953125 error: 0.007881411252170345
2^-10: 1.0009765625 error: 0.0039401951225235265
2^-11: 1.00048828125 error: 0.001969968780300313
2^-12: 1.000244140625 error: 0.0009849520504721099
2^-13: 1.0001220703125 error: 0.0004924679222275685
2^-14: 1.00006103515625 error: 0.0002462319323930373
2^-15: 1.000030517578125 error: 0.00012311545724141837
2^-16: 1.0000152587890625 error: 6.155759983439424e-5
2^-17: 1.0000076293945312 error: 3.077877117529937e-5
2^-18: 1.0000038146972656 error: 1.5389378673624776e-5
2^-19: 1.0000019073486328 error: 7.694675146829866e-6
2^-20: 1.0000009536743164 error: 3.8473233834324105e-6
2^-21: 1.0000004768371582 error: 1.9235601902423127e-6
2^-22: 1.000000238418579 error: 9.612711400208696e-7
2^-23: 1.0000001192092896 error: 4.807086915192826e-7
2^-24: 1.0000000596046448 error: 2.394961446938737e-7
2^-25: 1.0000000298023224 error: 1.1656156484463054e-7
2^-26: 1.0000000149011612 error: 5.6956920069239914e-8
2^-27: 1.0000000074505806 error: 3.460517827846843e-8
```

```
2^-28: 1.0000000037252903 error: 4.802855890773117e-9
2^-29: 1.0000000018626451 error: 5.480178888461751e-8
2^-30: 1.0000000009313226 error: 1.1440643366000813e-7
2^-31: 1.0000000004656613 error: 1.1440643366000813e-7
2^-32: 1.0000000002328306 error: 3.5282501276157063e-7
2^-33: 1.0000000001164153 error: 8.296621709646956e-7
2^-34: 1.0000000000582077 error: 8.296621709646956e-7
2^-35: 1.0000000000291038 error: 2.7370108037771956e-6
2^-36: 1.000000000014552 error: 1.0776864618478044e-6
2^-37: 1.000000000007276 error: 1.4181102600652196e-5
2^-38: 1.000000000003638 error: 1.0776864618478044e-6
2^-39: 1.000000000001819 error: 5.9957469788152196e-5
2^-40: 1.0000000000009095 error: 0.0001209926260381522
2^-41: 1.0000000000004547 error: 1.0776864618478044e-6
2^-42: 1.0000000000002274 error: 0.0002430629385381522
2^-43: 1.0000000000001137 error: 0.0007313441885381522
2^-44: 1.0000000000000568 error: 0.0002452183114618478
2^-45: 1.0000000000000284 error: 0.003661031688538152
2^-46: 1.0000000000000142 error: 0.007567281688538152
2^-47: 1.000000000000007 error: 0.007567281688538152
2^-48: 1.0000000000000036 error: 0.023192281688538152
2^-49: 1.0000000000000018 error: 0.008057718311461848
2^-50: 1.0000000000000009 error: 0.11694228168853815
2^-51: 1.0000000000000004 error: 0.11694228168853815
2^-52: 1.0000000000000002 error: 0.6169422816885382
2^-53: 1.0 error: 0.11694228168853815
2^-54: 1.0 error: 0.11694228168853815
```

Wnioski

Jak można wywnioskować z powyższych danych, gdy wartość h maleje dokładność przybliżenia rośnie, ale tylko do pewnego momentu. Do $h = 2^{-28}$ błąd przybliżenia jest marginalny. Potem już rośnie. Wzrost błędu obliczeń można wytłumaczyć tym, że dla bardzo małych wartości h przy dodawaniu do niej 1 jest ono pochłaniane (tj. zachodzi $h + 1 = 1$).