Krzysztof Dobrucki
Numer albumu 268507
Informatyka Algorytmiczna
Semestr Zimowy 2023/2024
29.10.2023r.

Lista 1

Obliczenia Naukowe

Zadanie 1

Zadanie polegało na iteracyjnym wyznaczeniu kilku wartości, które charakteryzują arytmetykę.

Epsilon maszynowy

Pierwszą z nich był macheps, czyli najmniejsza taka liczba epsilon, że ϵ, że $1 + \epsilon > 1$. Epsilon jest zatem odległością od 1 do najmniejszej liczby większej od 1 możliwej do zapisania w danej arytmetyce. Liczba 1 ma w zapisie dwójkowym mantysę wypełnioną zerami:

In [ ]:
```
bitstring(Float16(1.0))
```

```
"0011110000000000"
```

z tego powodu można przewidywać, że $1 + \epsilon$ różni się od 1 w swoim najmniej znaczącym bicie. Rozpoczniemy proces sprawdzania epsilonów od wartości 1 i będziemy tworzyć kolejne poprzez dzielenie przez 2. W reprezentacji dwójkowej oznacza to przesuwanie jedynki w mantysie coraz bardziej w prawo. Będziemy postępować zgodnie z następującym algorytmem (iterative_epsilon), gdzie argumentem jest typ arytmetyki. Następnie porównamy wynik z wbudowaną funkcją języka Julia.

In [ ]:
```
function iterative_epsilon(type)
    epsilon = type(1.0)
    while type(1.0 + 0.5*epsilon) > type(1.0)
        epsilon = type(0.5 * epsilon)
    end
    return epsilon
end

for i in [Float16, Float32, Float64]
    println(i)
    println(rpad("iterative_epsilon: ", 20), iterative_epsilon(i))
```

```
    println(rpad("in-build function: ", 20), eps(i), '\n')
end
```

```
Float16
iterative_epsilon:  0.000977
in-build function:  0.000977

Float32
iterative_epsilon:  1.1920929e-7
in-build function:  1.1920929e-7

Float64
iterative_epsilon:  2.220446049250313e-16
in-build function:  2.220446049250313e-16
```

Iteracyjnie wyniki są takie same jak te zwracane przez wbudowaną funkcję. Wartości do porównania z plikiem nagłówkowym **float.h** języka C zostały wzięte ze znalezionego artykułu w internecie.

FLT_EPSILON = 1.19209e-07 (Float32)

DBL_EPSILON = 2.22045e-16 (Float64)

LDBL_EPSILON = 1.0842e-19

Float16 nie ma odpowiednika, jednak pozostałe wartości są zbliżone.

Epsilon a precyzja

Teraz zbadamy związek epsilonu maszynowego z precyzją. Precyzją arytmetyki, oznaczaną również jako ϵ, definiuje się jako liczbę wyrażoną względem wzoru:

$$\epsilon = \frac{1}{2}\beta^{1-t}$$

gdzie β to baza reprezentacji (w naszym przypadku 2), a t oznacza liczbę bitów w mantysie znormalizowanej do przedziału $[1/\beta, 1)$. Dla typów Float16, Float32 i Float64 precyzje arytmetyki wynoszą odpowiednio:

Float16:
$2^{-1} \cdot 2^{1-10} = 2^{-10}$
Float32:
$2^{-1} \cdot 2^{1-23} = 2^{-23}$
Float64:
$2^{-1} \cdot 2^{1-52} = 2^{-52}$

Teraz zweryfikujemy obliczone wartości precyzji dla odpowiadających im typów arytmetycznych.

In [ ]:
```
println("Float16: ", Float16(2^-10))
println("Float32: ", Float32(2^-23))
println("Float64: ", Float64(2^-52))
```

```
Float16: 0.000977
Float32: 1.1920929e-7
Float64: 2.220446049250313e-16
```

Wartości precyzji, które otrzymaliśmy, są zgodne z wcześniejszymi obliczonymi epsilonami maszynowymi. Możemy zatem wnioskować, że w przypadku danej arytmetyki, wartość macheps jest równa jej precyzji (choć nie jest to wystarczający dowód).

Liczba eta

Następną wartością do ustalenia jest liczba eta, co oznacza najmniejszą liczbę η taka, że $\eta > 0$. Podobnie jak w przypadku epsilonu, intuicja opiera się na fakcie, że mantysa liczby 0 jest wypełniona zerami. W związku z tym, zastosowana metoda jest zupełnie analogiczna do powyższej.

In [ ]:

```
function iterative_eta(type)
    eta = type(1.0)
    while type(0.5*eta) > 0
        eta = type(0.5*eta)
    end
    return eta
end

for i in [Float16, Float32, Float64]
    println(i)
    println(rpad("iterative_eta: ", 20), iterative_eta(i))
    println(rpad("in-build function: ", 20), nextfloat(i(0.0)), '\n')
end
```

```
Float16
iterative_eta:      6.0e-8
in-build function:  6.0e-8

Float32
iterative_eta:      1.0e-45
in-build function:  1.0e-45

Float64
iterative_eta:      5.0e-324
in-build function:  5.0e-324
```

Ponownie iteracyjnie wyliczona wartość jest ideantyczna z uzyskaną za pomocą wbudowanej funkcji.

$\mathrm{MIN}_{\mathrm{sub}}$

$\mathrm{MIN}_{\mathrm{sub}}$ to najmniejsza liczba w postaci nieznormalizowanej w danej arytmetyce, co oznacza, że jej znormalizowane reprezentowanie wymagałoby użycia wykładnika niższego, niż jest to dopuszczalne. Obliczamy ją zgodnie z następującym wzorem:

$$\mathrm{MIN}_{sub} = 2^{1-t} \cdot 2^{c_{min}}$$

gdzie t oznacza liczbę cyfr mantysy (z zakresu $[1, 2)$), a c_{min} to minimalna możliwa cecha, wyznaczana ze wzoru:

$$c_{min} = -2^{d-1} + 2$$

gdzie d to liczba bitów przeznaczonych na zapis cechy. Dla typów w języku Julia obliczenia wyglądają następująco:

Dla Float16:
$c_{min} = -2^{5-1} + 2 = -14$
$\text{MIN}_{\text{sub}} = 2^{-10} \cdot 2^{-14} = 2^{-24}$

Dla Float32:
$c_{min} = -2^{8-1} + 2 = -126$
$\text{MIN}_{\text{sub}} = 2^{-23} \cdot 2^{-126} = 2^{-149}$

Dla Float64:
$c_{min} = -2^{11-1} + 2 = -1022$
$\text{MIN}_{\text{sub}} = 2^{-52} \cdot 2^{-1022} = 2^{-1074}$

In [ ]:
```
println("Float16: ", Float16(2^-24))
println("Float32: ", Float32(2^-149))
println("Float64: ", Float64(2^-1074))
```

```
Float16: 6.0e-8
Float32: 1.0e-45
Float64: 5.0e-324
```

Uzyskane wartości są zgodne z tymi z punktu **Liczba eta**

MIN_{nor}

Przeprowadźmy taki sam test dla funkcji floatmin() i porównajmy wyniki.

In [ ]:
```
println("Float16: ", floatmin(Float16))
println("Float32: ", floatmin(Float32))
println("Float64: ", floatmin(Float64))
```

```
Float16: 6.104e-5
Float32: 1.1754944e-38
Float64: 2.2250738585072014e-308
```

Wyniki są większe od poprzednich.

In [ ]:
```
bitstring(floatmin(Float16))
```

```
"0000010000000000"
```

Wynik wskazuje, że mamy do czynienia z formą znormalizowaną. Sprawdźmy związek wartości floatmin() z MIN_{nor} (najmniejszą znormalizowaną wartością możliwą do zapisania), która wyrażana jest wzorem:

$$MIN_{nor} = 2^{c_{min}}$$

gdzie c_{min} jest takie samo jak dla MIN_{sub}.

Float16:
$\text{MIN}_{\text{nor}} = 2^{-14}$

Float32:
$\text{MIN}_{\text{nor}} = 2^{-126}$

Float64:
$\text{MIN}_{\text{nor}} = 2^{-1022}$

In [ ]:
```
println("Float16: ", Float16(2^-14))
println("Float32: ", Float32(2^-126))
println("Float64: ", Float64(2^-1022))
```

```
Float16: 6.104e-5
Float32: 1.1754944e-38
Float64: 2.2250738585072014e-308
```

Wartości MIN_{nor} pokrywają się z floatmin().

Największa liczba możliwa do wyrażenia

Ostatnią poszukiwaną wartością jest MAX, czyli największa możliwa liczba, jaką można wyrazić w danej arytmetyce. Intuicja podpowiada, że będzie to liczba, której wykładnik ma maksymalną dopuszczalną wartość, a mantysa składa się wyłącznie z jedynek.

In [ ]:
```
function iterative_max(type)
    max = type(1.0)
    while !isinf(2*max)
        max *= 2
    end
    max *= (type(2.0) - eps(type))
    return max
end

for i in [Float16, Float32, Float64]
    println(i)
    println(rpad("iterative_max: ", 25), iterative_max(i))
    println(rpad("in-build function: ", 25), floatmax(i), '\n')
end
```

```
Float16
iterative_max:          6.55e4
in-build function:      6.55e4

Float32
iterative_max:          3.4028235e38
in-build function:      3.4028235e38

Float64
iterative_max:          1.7976931348623157e308
in-build function:      1.7976931348623157e308
```

Wyniki ponownie są takie same, czyli nasze iteracyjne podejście jest poprawne. Sprawdźmy jeszcze wartości dla języka C z pliku **float.h**:

FLT_MAX = 3.40282e+38 (Float32)

DBL_MAX = 1.79769e+308 (Float64)

LDBL_MAX = 1.18973e+4932

Float16 nie ma odpowiednika, jednak pozostałe wartości są zbliżone.

Wnioski

Standard IEEE-754 narzuca pewne ograniczenia na dokładność reprezentacji liczb zmiennoprzecinkowych (jak każdy system oparty na maszynach liczących). Wzrost liczby bitów reprezentujących liczbę przekłada się na wzrost precyzji. W miarę jak liczba bitów rośnie, zmniejsza się najmniejsza możliwa do wyrażenia dodatnia liczba (η) oraz najmniejsza liczba większa od 1 ($1 + \epsilon$), a jednocześnie znacząco rośnie maksymalna liczba możliwa do wyrażenia.

Zadanie 2

Cel to eksperymentalne sprawdzenie prawidłowości wzoru Kahana na epsilon maszynowy:

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

Możemy porównać wyniki z tymi zwracanymi w podpunkcie **Epsilon maszynowy**.

In [ ]:
```
function kahan_epsilon(type)
    type(3.0) * (type(4.0/3.0) - type(1.0)) - type(1.0)
end

for i in [Float16, Float32, Float64]
    println(i)
    println(rpad("kahan_epsilon: ", 20), kahan_epsilon(i))
    println(rpad("in-build functio: ", 20), eps(i), '\n')
end
```

```
Float16
kahan_epsilon:      -0.000977
in-build functio:   0.000977

Float32
kahan_epsilon:      1.1920929e-7
in-build functio:   1.1920929e-7

Float64
kahan_epsilon:      -2.220446049250313e-16
in-build functio:   2.220446049250313e-16
```

Z dokładnością do znaku wzór okazał się poprawny dla sprawdzanych arytmetyk.

Zadanie 3

Eksperymentalnie zweryfikujemy, że w arytmetyce Float64 liczby z przedziału $[1, 2]$ są równomiernie rozmieszczone z krokiem $\delta = 2^{-52}$. Ze względu na wielkość obliczeń sprawdzania wszystkich dostępnych liczb, skupimy się na wartościach wokół końców tego przedziału:

In [ ]:
```julia
function check_values(start, delta, func)
    real = start
    for i in 1:1000
        start += delta
        real = func(real)
        if start != real
            return false
        end
    end
    return true
end

delta = 2.0^-52
if (check_values(1.0, delta, nextfloat) && check_values(2.0, -delta, prevfloat))
    println("For a given interval, the numbers are evenly distributed.")
else
    println("For a given interval, the numbers are NOT evenly distributed.")
end
```

```
For a given interval, the numbers are evenly distributed.
```

Sprawdźmy czy poza tym przedziałem wynik będzie taki sam:

In [ ]:
```julia
if (1.0 - delta == prevfloat(1.0))
    println("True")
else
    println("False")
end
```

```
False
```

In [ ]:
```julia
println(bitstring(prevfloat(1.0)))
println(bitstring(1.0))
println(bitstring(nextfloat(1.0)))
```

```
0011111111101111111111111111111111111111111111111111111111111111
0011111111110000000000000000000000000000000000000000000000000000
0011111111110000000000000000000000000000000000000000000000000001
```

Jeśli spróbujemy zrozumieć jak działa reprezentacja liczb w komputerach, czyli maszynach binarnych, możemy dojśc do wniosku, że taki wynik jest spodziewany. Przy przekroczeniu kolejnej potęgi dwójki potrzebujemy więcej bitów, aby zapisać części całkowite, co zmniejsza "miejsce" dla części ułamkowej.

Poniższy kod udowadnia to stwierdzenie:

In [ ]:
```
exponent = parse(Int, bitstring(0.5)[2:12], base=2)
real_exp = exponent - 1023 - 52
delta = 2.0^real_exp

if (check_values(0.5, delta, nextfloat) && check_values(1.0, -delta, prevfloat))
    println("[0.5; 1.0] = $delta = 2^$real_exp")
end
```

```
[0.5; 1.0] = 1.1102230246251565e-16 = 2^-53
```

In [ ]:
```
exponent = parse(Int, bitstring(2.0)[2:12], base=2)
real_exp = exponent - 1023 - 52
delta = 2.0^real_exp

if (check_values(2.0, delta, nextfloat) && check_values(4.0, -delta, prevfloat))
    println("[2.0; 4.0] = $delta = 2^$real_exp")
end
```

```
[2.0; 4.0] = 4.440892098500626e-16 = 2^-51
```

Wnioski

Liczby w przedziałach między kolejnymi potęgami liczby 2 są równomiernie rozmieszczone. Co istotne, ze względu na ograniczoną liczbę możliwych wartości mantysy, każdy taki przedział zawiera dokładnie taką samą liczbę liczb. W miarę jak odległości między kolejnymi potęgami dwójki rosną, krok δ na przedziałach między nimi musi również rosnąć.

Zadanie 4

Należy znaleźć najmniejszą liczbę $x \in (1, 2)$ typu Float64, dla której $x \cdot \frac{1}{x} \neq 1$. Zastosujemy więc najprostszą metodę. Będziemy przeglądać kolejne liczby od 1 w górę, aż do momentu, w którym warunek ten zostanie spełniony:

In [ ]:
```
current = 1.0
while nextfloat(current) * (1.0/nextfloat(current)) == 1.0
    current = nextfloat(current)
end
println("The smallest number found is: $current")
```

```
The smallest number found is: 1.0000000572289969
```

Wnioski

Ograniczenia sprzętowe w liczbie bitów reprezentujących daną liczbę powoduje czasami błędy.

Zadanie 5

Celem zadania jest zaimplementowanie czterech różnych strategii obliczania iloczynu skalarnego dwóch wektorów x i y, a następnie porównanie uzyskanych wyników.

1. "w przód" - $\sum_{i=1}^{n} x[i] \cdot y[i]$:

In [ ]:
```
function forward(x, y, type)
    sum = type(0.0)
    for i in 1:length(x)
        sum += x[i] * y[i]
    end
    return sum
end
```

```
forward (generic function with 1 method)
```

2. "w tył" - $\sum_{i=n}^{1} x[i] \cdot y[i]$:

In [ ]:
```
function backward(x, y, type)
    sum = type(0.0)
    for i in length(x):-1:1
        sum += x[i] * y[i]
    end
    return sum
end
```

```
backward (generic function with 1 method)
```

3. "od największego do najmniejszego" - dodaj dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe

In [ ]:
```
function descending_order(x, y)
    p = x .* y
    sum_pos = sum(sort(filter(a -> a>0, p), rev=true))
    sum_neg = sum(sort(filter(a -> a<0, p)))
    return sum_pos+sum_neg
end
```

```
descending_order (generic function with 1 method)
```

4. "od najmniejszego do największego" - przeciwnie do punktu 3.

In [ ]:
```
function ascending_order(x, y)
    p = x .* y
    sum_pos = sum(sort(filter(a -> a>0, p)))
    sum_neg = sum(sort(filter(a -> a<0, p), rev=true))
    return sum_pos+sum_neg
end
```

```
ascending_order (generic function with 1 method)
```

Porównamy teraz powyższe sposoby obliczania iloczynu skalarnego dla danych z zadania z faktyczną wartością czyli $-1.00657107000000 \cdot 10^{-11}$.

In [ ]:
```
x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]
```

```
for i in [Float32, Float64]
    a = Array{i,1}(x)
    b = Array{i,1}(y)
    println(i)
    println(rpad("Precise value: ", 20), "-1.00657107000000e-11")
    println(rpad("Forward: ", 20), forward(a, b, i))
    println(rpad("Backward: ", 20), backward(a, b, i))
    println(rpad("Descending_order: ", 20), descending_order(a, b))
    println(rpad("Ascending_order: ", 20), ascending_order(a, b), '\n')
end
```

```
Float32
Precise value:      -1.00657107000000e-11
Forward:            -0.4999443
Backward:           -0.4543457
Descending_order:   -0.5
Ascending_order:    -0.5

Float64
Precise value:      -1.00657107000000e-11
Forward:            1.0251881368296672e-10
Backward:           -1.5643308870494366e-10
Descending_order:   0.0
Ascending_order:    0.0
```

Wnioski

Jak widać uzyskaliśy różne wyniki. Na błędy w obliczeniach wpływa typ arytmetyki i kolejność wykonywania działań.

Zadanie 6

Dane są dwie funkcje f i g dane wzorami:

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

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

Warto zauważyć, że:

$$f = g$$

sprawdźmy, jakie wyniki daszą obie feunkcje dla:

$$x = 8^{-k}; k = 1, 2, 3, \dots$$

In [ ]:
```
function f(x)
    sqrt(x^2 + 1.) - 1.
end

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

```
end

println(rpad("x", 8), rpad("|", 2), rpad("f(x)", 25), rpad("|", 2), rpad("g(x)",
println("-"^8, "+", "-"^26, "+", "-"^26)
for i in 1:20
    println(rpad("8^-$i", 8), rpad("|", 2), rpad(f(8.0^-i), 25), rpad("|", 2), r
end
```

```
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
8^-11   | 0.0                      | 6.776263578034403e-21
8^-12   | 0.0                      | 1.0587911840678754e-22
8^-13   | 0.0                      | 1.6543612251060553e-24
8^-14   | 0.0                      | 2.5849394142282115e-26
8^-15   | 0.0                      | 4.0389678347315804e-28
8^-16   | 0.0                      | 6.310887241768095e-30
8^-17   | 0.0                      | 9.860761315262648e-32
8^-18   | 0.0                      | 1.5407439555097887e-33
8^-19   | 0.0                      | 2.407412430484045e-35
8^-20   | 0.0                      | 3.76158192263132e-37
```

Wnioski

Matematycznie funkcje są sobie równe, jednak problemem jest odejmowanie małych liczb, więc funkcja g daje bliższe prawdy wyniki. Wartości x spadają do zera ponieważ dla bardzo małych x mamy $\sqrt{x^2+1} \approx 1$, czyli odejmowanie małych liczb. Jeśli chcemy uzyskać wyniki jak najbliższe prawdy powinniśmy przekształcić odpowiednio wzory, zamiast "ślepo" wrzucać do programu.

Zadanie 7

W tym zadaniu skorzystamy ze wzoru na pochodną funkcji:

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

Badana fukcja:

$$f(x) = sin(x) + cos(3x)$$

Pochodna badanej fukcji:

$$f'(x) = cos(x) + 3sin(3x)$$

Cel to porównanie wyników z dokładnymi wartościami.

In [ ]:
```
println("f(1.0)  = ", (sin(1.0) + cos(3*1.0)))
println("f'(1.0) = ", (cos(1.0) - 3 * sin(3*1.0)))
```

```
f(1.0)  = -0.1485215117925489
f'(1.0) = 0.11694228168853815
```

In [ ]:
```
function f(x)
    return sin(x) + cos(3x)
end

function real_df(x)
    return cos(x) - 3sin(3x)
end

function approx_df(x, h)
    return (f(x + h) - f(x)) / h
end

println(rpad("h", 8), rpad("|", 2), rpad("approx_value", 25), rpad("|", 2), rpad
println("-"^8, "+", "-"^26, "+", "-"^26)
for i in 1:54
    println(rpad("2^-$i", 8), rpad("|", 2), rpad(Float64(approx_df(1.0, 2.0^-i))
end
```

h	approx_value	error_value
2^-1	1.8704413979316472	1.753499116243109
2^-2	1.1077870952342974	0.9908448135457593
2^-3	0.6232412792975817	0.5062989976090435
2^-4	0.3704000662035192	0.253457784514981
2^-5	0.24344307439754687	0.1265007927090087
2^-6	0.18009756330732785	0.0631552816187897
2^-7	0.1484913953710958	0.03154911368255764
2^-8	0.1327091142805159	0.015766832591977753
2^-9	0.1248236929407085	0.007881411252170345
2^-10	0.12088247681106168	0.0039401951225235265
2^-11	0.11891225046883847	0.001969968780300313
2^-12	0.11792723373901026	0.0009849520504721099
2^-13	0.11743474961076572	0.0004924679222275685
2^-14	0.11718851362093119	0.0002462319323930373
2^-15	0.11706539714577957	0.00012311545724141837
2^-16	0.11700383928837255	6.155759983439424e-5
2^-17	0.11697306045971345	3.077877117529937e-5
2^-18	0.11695767106721178	1.5389378673624776e-5
2^-19	0.11694997636368498	7.694675146829866e-6
2^-20	0.11694612901192158	3.8473233834324105e-6
2^-21	0.1169442052487284	1.9235601902423127e-6
2^-22	0.11694324295967817	9.612711400208696e-7
2^-23	0.11694276239722967	4.807086915192826e-7
2^-24	0.11694252118468285	2.394961446938737e-7
2^-25	0.116942398250103	1.1656156484463054e-7
2^-26	0.11694233864545822	5.6956920069239914e-8
2^-27	0.11694231629371643	3.460517827846843e-8
2^-28	0.11694228649139404	4.802855890773117e-9
2^-29	0.11694222688674927	5.480178888461751e-8
2^-30	0.11694216728210449	1.1440643366000813e-7
2^-31	0.11694216728210449	1.1440643366000813e-7
2^-32	0.11694192886352539	3.5282501276157063e-7
2^-33	0.11694145202636719	8.296621709646956e-7
2^-34	0.11694145202636719	8.296621709646956e-7
2^-35	0.11693954467773438	2.7370108037771956e-6
2^-36	0.116943359375	1.0776864618478044e-6
2^-37	0.1169281005859375	1.4181102600652196e-5
2^-38	0.116943359375	1.0776864618478044e-6
2^-39	0.11688232421875	5.9957469788152196e-5
2^-40	0.1168212890625	0.0001209926260381522
2^-41	0.116943359375	1.0776864618478044e-6
2^-42	0.11669921875	0.0002430629385381522
2^-43	0.1162109375	0.0007313441885381522
2^-44	0.1171875	0.0002452183114618478
2^-45	0.11328125	0.003661031688538152
2^-46	0.109375	0.007567281688538152
2^-47	0.109375	0.007567281688538152
2^-48	0.09375	0.023192281688538152
2^-49	0.125	0.008057718311461848
2^-50	0.0	0.11694228168853815
2^-51	0.0	0.11694228168853815
2^-52	-0.5	0.6169422816885382
2^-53	0.0	0.11694228168853815
2^-54	0.0	0.11694228168853815

Najdokładniejsze przybliżenie otrzymujemy, gdy h wynosi 2^{-28}. Dalsze zmniejszenie wartości h skutkuje ponownym wzrostem błędu. Przyjrzyjmy się teraz wartościom wyrażenia $1 + h$:

In [ ]:
```
println(rpad("h", 8), rpad("|", 2), rpad("1+h", 22),  rpad("|", 2), rpad("f(1+h)
println("-"^8, "+", "-"^23, "+", "-"^23,  "+", "-"^23)
for i in 1:54
    println(rpad("2^-$i", 8), rpad("|", 2), rpad(1.0+2.0^-i, 22), rpad("|", 2),
end
```

h	1+h	f(1+h)	f(1+h)-f(1)
2^-1	1.5	0.7866991871732747	0.9352206989658236
2^-2	1.25	0.12842526201602544	0.27694677380857435
2^-3	1.125	-0.0706163518803512	0.07790515991219771
2^-4	1.0625	-0.12537150765482896	0.02315000413771995
2^-5	1.03125	-0.14091391571762557	0.0076075960749233396
2^-6	1.015625	-0.1457074873658719	0.0028140244266769976
2^-7	1.0078125	-0.14736142276621222	0.0011600890263366859
2^-8	1.00390625	-0.14800311681489065	0.0005183949776582653
2^-9	1.001953125	-0.1482777155172741	0.00024379627527482128
2^-10	1.0009765625	-0.1484034624987881	0.00011804929376080242
2^-11	1.00048828125	-0.14846344917024967	5.806262229923753e-5
2^-12	1.000244140625	-0.14849272096399935	2.8790828549563052e-5
2^-13	1.0001220703125	-0.14850717649596556	1.4335296583345425e-5
2^-14	1.00006103515625	-0.14851435917330935	7.152619239558788e-6
2^-15	1.000030517578125	-0.14851793924014578	3.57255240313048e-6
2^-16	1.0000152587890625	-0.1485197264556457	1.7853369032039268e-6
2^-17	1.0000076293945312	-0.14852061935892114	8.924336277749134e-7
2^-18	1.0000038146972656	-0.1485210656344409	4.4615810801396094e-7
2^-19	1.0000019073486328	-0.14852128872817139	2.2306437752472874e-7
2^-20	1.0000009536743164	-0.14852140026402927	1.1152851964180144e-7
2^-21	1.0000004768371582	-0.1485214560292064	5.576334249912662e-8
2^-22	1.000000238418579	-0.1485214839111071	2.7881441821975272e-8
2^-23	1.0000001192092896	-0.1485214978518853	1.3940663623479566e-8
2^-24	1.0000000596046448	-0.14852150482223148	6.970317434351614e-9
2^-25	1.0000000298023224	-0.14852150830739386	3.4851550534398257e-9
2^-26	1.0000000149011612	-0.14852151004997227	1.7425766385414931e-9
2^-27	1.0000000074505806	-0.14852151092126076	8.712881527372929e-10
2^-28	1.0000000037252903	-0.14852151135690494	4.35643965346344e-10
2^-29	1.0000000018626451	-0.14852151157472704	2.1782187165086953e-10
2^-30	1.0000000009313226	-0.14852151168363803	1.0891088031428353e-10
2^-31	1.0000000004656613	-0.14852151173809347	5.4455440157141766e-11
2^-32	1.0000000002328306	-0.14852151176532125	2.722766456741965e-11
2^-33	1.0000000001164153	-0.14852151177893513	1.3613776772558595e-11
2^-34	1.0000000000582077	-0.14852151178574202	6.806888386279297e-12
2^-35	1.0000000000291038	-0.14852151178914552	3.4033886819884174e-12
2^-36	1.000000000014552	-0.14852151179084716	1.70174985214544e-12
2^-37	1.000000000007276	-0.14852151179169815	8.507639037702575e-13
2^-38	1.000000000003638	-0.14852151179212347	4.2543746303636e-13
2^-39	1.000000000001819	-0.1485215117923363	2.1260770921571748e-13
2^-40	1.0000000000009095	-0.14852151179244266	1.0624834345662748e-13
2^-41	1.0000000000004547	-0.14852151179249573	5.3179682879545e-14
2^-42	1.0000000000002274	-0.14852151179252238	2.653433028854124e-14
2^-43	1.0000000000001137	-0.1485215117925357	1.3211653993039363e-14
2^-44	1.0000000000000568	-0.14852151179254225	6.661338147750939e-15
2^-45	1.0000000000000284	-0.1485215117925457	3.219646771412954e-15
2^-46	1.0000000000000142	-0.14852151179254736	1.5543122344752192e-15
2^-47	1.000000000000007	-0.14852151179254813	7.771561172376096e-16
2^-48	1.0000000000000036	-0.14852151179254858	3.3306690738754696e-16
2^-49	1.0000000000000018	-0.1485215117925487	2.220446049250313e-16
2^-50	1.0000000000000009	-0.1485215117925489	0.0
2^-51	1.0000000000000004	-0.1485215117925489	0.0
2^-52	1.0000000000000002	-0.14852151179254902	-1.1102230246251565e-16
2^-53	1.0	-0.1485215117925489	0.0
2^-54	1.0	-0.1485215117925489	0.0

Widzimy, że dla $h \leq 2^{-53}$ w arytmetyce Float64 mamy $1 + h = 1$, a więc $f(1+h) - f(1) = 0$, stąd dalsze zmniejszanie h nie spowoduje już zmiany w jakości przybliżenia.

Wnioski

Niekiedy nie jest możliwe proste przekształcenie wzoru w taki sposób, aby uniknąć utraty cyfr znaczących podczas operacji odejmowania. Błędy obliczeniowe, które występują, często są sprzeczne z intuicją matematyczną, która sugeruje, że używanie coraz mniejszych wartości h prowadziłoby do poprawy dokładności przybliżenia.