Politechnika Wrocławska
Wydział Podstawowych Problemów Techniki

Obliczenia naukowe

Sprawozdanie z zajęć laboratoryjnych

Lista 1

Autor:
Jakub Pezda
221426

1.1 Opis problemu

Ćwiczenie składa się z kilku części. W pierwszej należy wyznaczyć iteracyjnie epsilony maszynowe dla wszystkich typów zmiennopozycyjnych i porównać z wartościami funkcji języka Julia oraz z danymi zawartymi w pliku nagłówkowym `float.h` języka C.

W drugiej części trzeba iteracyjnie liczbę `eta`, taką że `eta` jest najmniejszą liczbą > 0.0 dla wszystkich typów zmiennopozycyjnych oraz porównać z wartościami funkcji języka Julia.

W ostatniej części należy wyznaczyć iteracyjnie liczbę `MAX` dla wszystkich typów zmiennopozycyjnych oraz porównać z wartościami funkcji języka Julia.

1.2 Rozwiązanie

Aby obliczyć epsilon maszynowy dla danego typu należy w pętli dzielić liczbę przez dwa dopóki ta liczba plus jeden jest większa od jedynki. Na listingu 1.1 przedstawiono pseudokod takiej funkcji.

```
while 1 + x / 2  > 1
     x /= 2
```

Listing 1.1 pętla obliczająca epsilon maszynowy

Iteracyjne wyznaczenie liczby `eta` jest podobne do powyższej metody. Tym razem jednak nie dodaje się jedynki oraz porównuje się liczbę podzieloną przez dwa z zerem. Poniżej zaprezentowano pseudokod funkcji wyznaczającej iteracyjnie liczbę `eta`.

```
while x / 2 > 0
     x /= 2
```

Listing 1.2 pętla wyznaczająca liczbę `eta`

Pętla wyznaczająca liczbę `MAX` jest analogiczna do powyższych. Zamiast jednak dzielić przez dwa należy mnożyć przez dwa oraz sprawdzać czy liczba jest nieskończonością. Na kolejnym listingu pokazano psudokod funkcji.

```
while x * 2 jest liczbą
     x *= 2
```

Listing 1.3 pętla obliczająca liczbę `MAX`

1.3 Wyniki

Wyniki zwracane przez funkcję przedstawioną na listingu 1.1 oraz wartości zwracane przez funkcję `eps()` języka Julia zostały przedstawione na listingu 1.4. Pierwsze trzy wartości są to epsilony obliczone iteracyjnie. Kolejne trzy są wartościami zwracanymi przez funkcję `eps()`. Ostatnie zaś pochodzą z pliku nagłówkowego `float.h` języka C.

```
macheps Float16: 0.00097656
macheps Float32: 1.1920929e-7
macheps Float64: 2.220446049250313e-16
eps(Float16): 0.00097656
eps(Float32): 1.1920929e-7
eps(Float64): 2.220446049250313e-16
```

```
float macheps: 1.1920929e-07
double macheps: 2.2204460492503131e-16
```

Listing 1.4 epsilony maszynowe wyznaczone iteracyjnie, pochodzące z funkcji `eps()` oraz języka C

Można zauważyć, że wyniki dla wszystkich testów są takie same. Należy jednak uważać, ponieważ domyślnie obliczenia wykonywane są zawsze z maksymalną precyzją.

Wyniki funkcji obliczającej liczbę `eta` oraz wartości zwracane przez funkcję `nextfloat()` przedstawiono na listingu 1.5.

```
eta Float16: 5.9605e-8
eta Float32: 1.0e-45
eta Float64: 5.0e-324
nextfloat(Float16(0.0)): 5.9605e-8
nextfloat(Float32(0.0)): 1.0e-45
nextfloat(Float64(0.0)): 5.0e-324
```

Listing 1.5 wyznaczona iteracyjnie liczba `eta`

Tutaj również wyniki iteracyjnego obliczenia funkcji `eta` oraz wartości zwracane przez funkcję `nextfloat()` są takie same. Wszytkie liczby mniejsze od liczby `eta` oraz większe od zera są traktowane jako zero. Po wykonaniu operacji `bits()` w języku Julia na liczbie `eta` można zauważyć, że liczba ta jest równa najmniejszej liczbie subnormalnej.

Wyniki zwracane przez funkcję obliczającą liczbę `MAX` znajdują się na listingu 1.6.

```
MAX float16: 65504.0
MAX float32: 3.4026535e38
MAX float64: 1.7976841463966415e308
realmax(Float16): 65504.0
realmax(Float32): 3.4028235e38
realmax(Float64): 1.7976931348623157e308
MAX float: 3.4028235e+38
MAX double: 1.7976931348623157e+308
```

Listing 1.6 liczba `MAX`

Pierwsze trzy wartości są liczbami `MAX` wyznaczonymi iteracyjnie. Kolejne trzy są wartościami zwracanymi przez funkcję `realmax()`. Ostatnie wartości pochodzą z pliku nagłówkowego `float.h` języka C.

1.4 Wnioski

Po wykonaniu powyższego ćwiczenia można wyciągnąć kilka wniosków. Im mniejsza wartość epsilonu maszynowego, tym większa jest względna precyzja obliczeń. Precyzja obliczeń jest ograniczona przez typ danych użyty do reprezentacji liczby zmiennopozycyjnej. Sam epsilon maszynowy jest zależny od implementacji.

2.1 Opis problemu

Ćwiczenie polega na sprawdzeniu słuszności stwierdzenia Kahan'a, w którym stwierdził, że epsilon maszynowy można uzyskać za pomocą wyrażenia 3(4/3 – 1) -1.

2.2 Rozwiązanie

Aby wykonać ćwiczenie wystarczy obliczyć wartość wyrażenia podanego przez Kahan'a dla każdego typu zmiennopozycyjnego.

2.3 Wyniki

Listing 2.1 prezentuje obliczone epsilony maszynowe za pomocą wzoru zaproponowanego przez Kahan'a.

```
macheps Float16: -0.00097656
macheps Float32: 1.1920929e-7
macheps Float64: -2.220446049250313e-16
```

Listing 2.1 obliczone epsilony maszynowe

Po porównaniu z wartościami przedstawionymi na listingu 1.4 można zauważyć, że wyniki różnią się znakiem w przypadku float16 oraz float64. Aby otrzymać poprawne wyniki należy dodatkowo we wzorze Kahan'a użyć funkcji `abs()`. W języku C wyniki były identyczne jak na listingu 2.1. Błąd pochodzi z zaokrąglenia liczb, które mają nieskończone rozwinięcie dwójkowe.

2.4 Wnioski

Komputery nie potrafią operować na liczbach rzeczywistych mających dowolną liczbę cyfr. Dokładność z jaką można te liczby przedstawiać, zależy od długości słowa w komputerze. Za pomocą wyrażenia zaproponowanego przez Kahan'a jesteśmy w stanie policzyć poprawny epsilon maszynowy co do wartości lecz nie co do znaku.

3.1 Opis problemu

Celem ćwiczenia jest pokazanie, że liczby w arytmetyce double w standardzie IEEE 754 są równomiernie rozmieszczone. Należy również sprawdzić jak rozmieszczone są liczby w przedziale [½, 1] oraz [2, 4].

3.2 Rozwiązanie

Aby wykonać poprawnie ćwiczenie należy dla danego przedziału w pierwszej kolejności obliczyć krok z jakim rozmieszczone są liczby. Eksperymentalnie można sprawdzić, że krok dla przedziału jest równy $2^{-52+\log_2 min}$. Działa on jednak tylko dla liczb `min`, które są równe potędze liczby dwa. Aby wzór działał dla pozostałych wartości należy wynik logarytmu zaokrąglić w dół. Po obliczeniu kroku można przejść do wyświetlania liczb z podanego przedziału. Listing 3.1 prezentuje pseudokod pętli wyświetlającej liczby.

```
while ilośćWyświetlonych  < ilośćDoWyświetlenia
     wyświelt(min + ilośćWyświetlonych * step)
     ilośćWyświetlonych += 1
```

Listing 3.1 pseudokod pętli wyświetlającej liczby

3.3. Wyniki

Dla przedziału [½, 1] oraz [2, 4] wyniki zostały przedstawione na listingu 3.2. Zaprezentowano

osiem pierwszych liczb z tych przedziałów.

```
Przedział [1.0,2.0]
Krok dla przedziału: 2.220446049250313e-16
0011111111110000000000000000000000000000000000000000000000000000
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000000111
Przedział [2.0,4.0]
Krok dla przedziału: 4.440892098500626e-16
0100000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
0100000000000000000000000000000000000000000000000000000000000101
0100000000000000000000000000000000000000000000000000000000000110
0100000000000000000000000000000000000000000000000000000000000111
```

Listing 3.2 rozmieszczenie liczb w danych przedziałach

Można zauważyć że w drugim przedziale liczby rozstawione są z dwa razy większym krokiem.

3.4 Wnioski

Rozkład liczb zmiennopozycyjnych w komputerze jest nierównomierny. Między kolejnymi potęgami dwójki znajduje się tyle samo liczb maszynowych. Dlatego znaczna ich część skupia się w pobliżu zera, ale pewne otoczenie zera stanowi lukę.

4.1 Opis problemu

Ćwiczenie polega na znalezieniu najmniejszej liczby x z przedziału (1, 2) w arytmetyce float64, takiej że x * (1/x) != 1.

4.2 Rozwiązanie

W pierwszej kolejności do zmiennej x należy przypisać najmniejszą wartość większa od 1. Następnie w pętli należy zwiększać liczbę x oraz sprawdzać czy warunek jest już spełniony. Pseudokod funkcji został przedstawiony na listingu 4.1.

```
while x * (1 / x) == 1
     x = nextfloat(x)
```

Listing 4.1 pętla obliczająca najmniejszą liczbę x taką, że x * (1 / x) != 1

4.3 Wyniki

Najmniejsza taka liczba jak i wynik wykonania programu zostały przedstawione na listingu 4.2.

```
1.000000057228997
```

Listing 4.2 wynik wykonania programu

Wykonanie programu zajmuje około minuty.

4.4 Wnioski

Dokładny wynik działań arytmetycznych na liczbach zmiennopozycyjnych na ogół nie jest taką liczbą. Przybliżenie liczby rzeczywistej bliską liczbą maszynową powoduje błędne wyniki. Błędy zaokrągleń są nieuniknione.

5.1 Opis problemu

Ćwiczenie polega na obliczeniu iloczynu skalarnego dwóch wektorów na cztery różne sposoby oraz porównaniu wyników z poprawną wartością.

5.2 Rozwiązanie

Aby wykonać ćwiczenie należy zaimplementować cztery algorytmy. Pierwsze dwa są bardzo proste. Polegają na sumowaniu kolejnych iloczynów składowych wektorów. Przy ostatnich dwóch sposobach najpierw należy obliczyć iloczyny odpowiadających sobie komponentów. Następnie trzeba posortować otrzymaną tablicę wartości. Po posortowaniu można zsumować wszystkie wartości dodatnie oraz ujemne a na koniec dodać oba wyniki.

5.3 Wyniki

Wynik wykonania programu przedstawiono na listingu 5.1.

```
Float64
a: 1.0251881368296672e-10
b: -1.5643308870494366e-10
c: 0.0
d: 0.0
Float32
a: -0.4999443
b: -0.4543457
c: -0.5
d: -0.5
```

Listing 5.1 obliczone iloczyny skalarne wektorów

Prawidłowy wynik sumy wynosi `-1.00657107000000e-11`. Wszystkie wyliczone wartości są różne od prawidłowej.

5.4 Wnioski

Kolejność wykonywania działań ma znacznie. Można zauważyć, że czym więcej wykonujemy działań na liczbach zmiennopozycyjnych tym większy jest błąd względny. Ponadto dodawanie bardzo małej liczby do dużo większej powoduje powstawanie bardzo dużych błędów. Utratę bitów można zminimalizować poprzez użycie podwójnej precyzji.

6.1 Opis problemu

Celem ćwiczenia jest obliczenie wartości tej samej funkcji, różnie zapisanej, dla kolejnych wartości argumentu oraz porównanie wyników ze sobą. Obie funkcje zaprezentowano na listingu 6.1.

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

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

Listing 6.1 funkcje przedstawione w treści zadania

6.2 Realizacja

Aby wykonać ćwiczenie należy zaimplementować funkcje podane w treści zadania oraz wykonać je dla kilku kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$.

6.3 Wyniki

Wyniki wykonania programu dla pięciu pierwszych wartości argumentu `x` zostały przedstawione na listingu 6.2.

```
f(8^-1)= 0.0077822185373186414
g(8^-1)= 0.0077822185373187065

f(8^-2)= 0.00012206286282867573
g(8^-2)= 0.00012206286282875901

f(8^-3)= 1.9073468138230965e-6
g(8^-3)= 1.907346813826566e-6

f(8^-4)= 2.9802321943606103e-8
g(8^-4)= 2.9802321943606116e-8

f(8^-5)= 4.656612873077393e-10
g(8^-5)= 4.6566128719931904e-10
```

Listing 6.2 wyniki kolejnych wywołań funkcji `f` oraz `g`

Mimo, że funkcje są sobie równe to wartości zwracane przez te funkcje są od siebie różne. Poprawne wyniki zwraca funkcja g. Błąd pierwszej funkcji bierze się z odejmowania bliskich sobie liczb.

6.4 Wnioski

Gdy tylko to jest możliwe, powinniśmy unikać ryzykownego odejmowania bliskich sobie liczb. Aby pozbyć się utraty znaczących bitów można spróbować poszukać alternatywnej postaci wyrażenia lub zastosować podwójną precyzję, która można pomóc w pewnych obliczeniach.

7.1 Opis problemu

Celem ćwiczenia jest obliczenie przybliżonej wartości pochodnej funkcji `f` oraz błędu tego przybliżenia. Poniżej zaprezentowano funkcję `f`.

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

Listing 7.1 funkcja f przedstawiona w treści zadania

7.2 Rozwiązanie

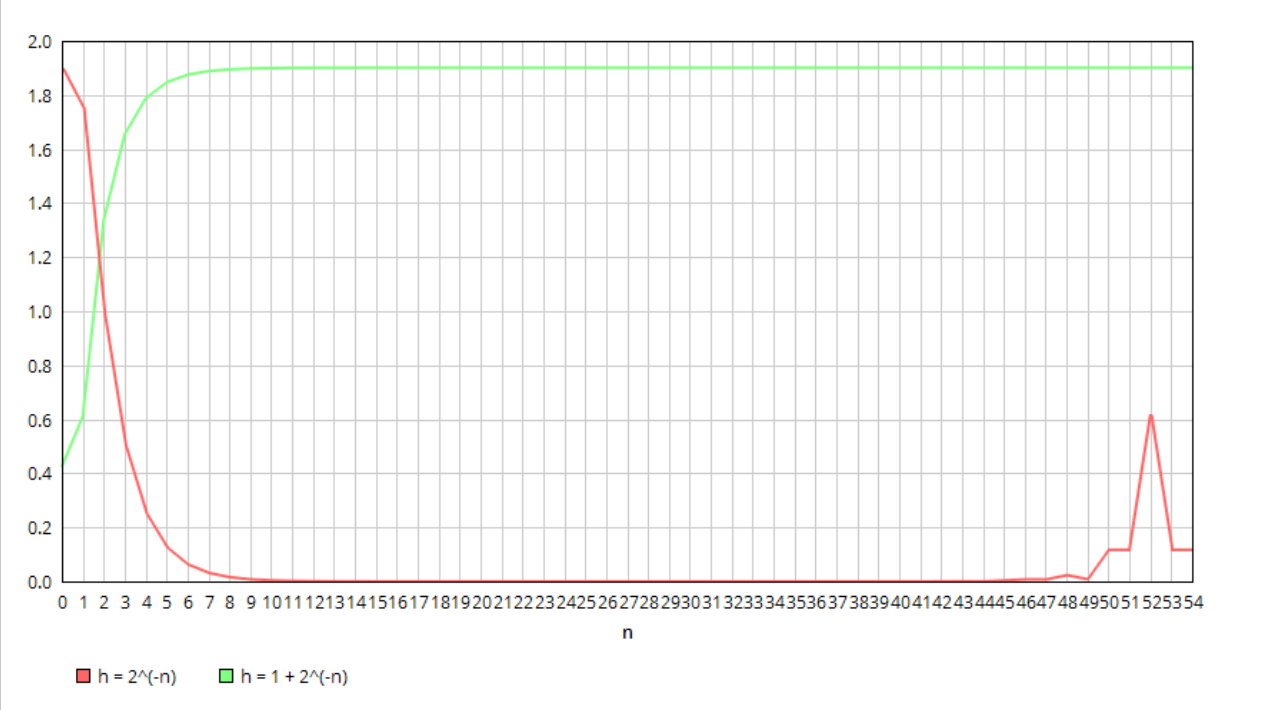
Aby obliczyć przybliżoną wartość pochodnej w punkcie x_0 skorzystano z następującego wzoru:

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

Jako h przyjęto wartość 2^{-n} dla n = 0,…,54. Sprawdzono również jak zachowują się wartości 1+h.

7.3 Wyniki

Wynik eksperymentu przedstawiono na poniższym wykresie.



Jak widać błąd dla wartości 1+h jest ogromny. Pochodzi on z tego, że obliczane były wartości funkcji trygonometrycznych dla zbyt dużych argumentów. Od wartości n = 53 dalsze zmniejszanie wartości h nie poprawia przybliżenia pochodnej. Wynika to z tego, że wartość h jest dużo mniejsza od x0 i wynik operacji x0+h jest równy x0.

7.4 Wnioski

Drastyczne zmniejszenie liczby cyfr znaczących występuje, gdy obliczamy wartości pewnych funkcji dla bardzo dużych argumentów. Można tego uniknąć poprzez zredukowanie argumentu przed wykonaniem funkcji. Jeżeli różnica pomiędzy wykładnikami liczb jest zbyt duża to suma tych liczb jest równa liczbie większej.