Obliczenia naukowe - Lista nr 1

Paweł Narolski

16 października 2018 r.

1 Rozpoznanie arytmetyki

1.1 Wyznaczanie epsilonów maszynowych

Epsilonem maszynowym *macheps* (ang. machine epsilon) nazywamy najmniejszą liczbę *macheps* <0 taką, że spełniona jest nierówność:

$$fl(1.0 + macheps) > 1.0. \tag{1}$$

Naszym pierwszym zadaniem w ramach tej części listy laboratoryjnej było napisanie programu w języku Julia, który w sposób iteracyjny będzie wyznaczał epsilony maszynowe dla wszystkich dostępnych typów zmiennopozycyjnych: *Float16, Float32* oraz *Float64* zgodnych ze standardem IEEE 754 (odpowiednio *half, single i double*).

Przygotowany na potrzeby ćwiczenia program, którego kod źródłowy dostępny jest w dołączonym pliku *ex1a.jl* realizuje następujący algorytm:

1. Przypisz $macheps := 1$
2. Dopóki spełniona jest nierówność (1) przypisz *macheps := macheps/2*
3. Zwróć otrzymaną wartość *macheps*

W kroku (2) wykonując dzielenie $macheps/2$dokonujemy przesunięcia bitowego w prawo. Bity przesuwamy do momentu, kiedy następne z przesunięć zostanie rozpoznane jako zero maszynowe.

Następnie otrzymane wyniki należało porównać z wartościami zwracanymi przez funkcję *eps()* dostępną dla programistów języka Julia dla poszczególnych typów zmiennopozycyjnych oraz z danymi zawartymi w pliku nagłówkowym *float.h* dowolnej instalacji języka *C*.

Wyniki przeprowadzonych działań zamieszczono w poniższej tabeli:

	***macheps* iteracyjnie**	***eps()***	**Wartość z *float.h***
Float16	0.0009765625	0.000977	nd.
Float32	1.1920928955078125e-7	1.1920929f-7	1.1920928955078125e-7
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460492503131e-16

1.2 Wyznaczanie liczby eta

Kolejnym naszym zadaniem było wyznaczenie w sposób iteracyjny liczby *eta* takiej, że dla wszystkich typów zmiennopozycyjnych *Float16, Float32* oraz *Float64* zgodnych ze standardem IEEE 754 (odpowiednio *half, single i double*):

$$eta > 0.0. \tag{2}$$

Aby wyznaczyć wartości liczby *eta* dla poszczególnych typów zmiennopozycyjnych w języku *Julia* zaimplementowany został algorytm (do wglądu w pliku źródłowym *ex1b.jl*) działający w następujący sposób:

1. Przypisz $eta := 1$

2. Dopóki wyrażona w danym systemie zmiennopozycyjnym wartość *eta/2 >0* przypisz *eta := eta/2*

3. Zwróć otrzymaną wartość *eta*

Wyznaczona w powyższy sposób liczba *eta* to najmniejsza wartość zmiennoprzecinkowa różna od zera, którą możemy zapisać w danym systemie zmiennopozycyjnym.

Następnie porównaliśmy otrzymane wartości liczbowe zwrócone przez iteracyjny algorytm wraz z wynikiem działania funkcji *nextfloat()* dla odpowiednich typów zmiennopozycyjnych:

	***eta* wyznaczona iteracyjnie**	***eta* zwrócona przez *nextfloat()***
Float16	5.960464477539063e-8	6.0e-8
Float32	1.401298464324817e-45	1.0e-45
Float64	5.0e-324	5.0e-324

1.3 Związek liczby macheps z precyzją arytmetyki

Jeśli liczba rzeczywista nie może zostać zapisana poprzez rozwinięcie dwójkowe na co najwyżej p bitach to musi zostać ona przybliżona poprzez liczbę zmiennoprzecinkową posiadającą taką dwójkową reprezentację. Problem ten nazywamy **błędem przybliżenia**.

Ważnym czynnikiem, za pomocą którego możemy opisać precyzję zapisu danej liczby w systemie zmiennopozycyjnym jest *epsilon maszynowy.* Wielkość *macheps* jest równa różnicy pomiędzy liczbą 1 a następną w kolejności większą liczbą zapisaną w systemie zmiennopozycyjnym.

Im mniejsza jest wartość *epsilona maszynowego*, tym większa jest względna precyzja obliczeń. Wiedząc, ile wynosi *macheps* możemy określić, że zapis liczby z precyzją *double* pozwoli nam na osiągnięcie dokładności rzędu do około 16 liczb po przecinku. Analogicznie w precyzji *single* osiągniemy dokładność rzędu około 7 liczb po przecinku, a *half* - około trzech.

Reasumując, $precision = macheps$.

1.4 Związek liczby *eta* z liczbą MIN_{sub}

Jak już wcześniej wspomnieliśmy, liczba *eta* to najmniejsza wartość zmiennoprzecinkowa różna od zera, którą możemy zapisać w danym systemie zmiennopozycyjnym.

Liczba *eta* jest zdenormalizowaną liczbą zmiennoprzecinkową. Oznacza to, że wszystkie bity cechy mają wartość zerową. W liczbie *eta* ostatni bit mantysy wynosi 1.

Reasumując, $eta = MIN_{sub}$.

1.5 Wyznaczanie maksymalnej wartości możliwej do zapisania

Ostatnim zadaniem w ramach tej części listy laboratoryjnej było wyznaczenie w sposób iteracyjny maksymalnej wartości możliwej do zapisania (MAX) dla wszystkich typów zmiennopozycyjnych (*Float16, Float32 i Float64*) zgodnych ze standardem IEEE 745.

Aby wyznaczyć *MAX* stworzony został prosty program w języku Julia (którego kod źrodłowy dostępny jest w pliku *ex1c.jl*) realizujący algorytm:

1. Przypisz $MAX := 1$

2. Dopóki nieprawda, że *isinf(2MAX)* przypisz *MAX:= 2MAX*

3. Przypisz MAX := (2 - *macheps* [dla odpowiedniej arytmetyki zmiennoprzecinkowej]) * MAX i zwróć otrzymaną wartość

Otrzymane wyniki dla poszczególnych typów zmiennopozycyjnych mieliśmy następnie za zadanie porównać z wartościami zwracanymi przez funkcję *realmax()* oraz z danymi zawartymi w pliku nagłówkowym *float.h* dowolnej instalacji języka C.

Porównanie otrzymanych wartości z odczytanymi wartościami MAX prezentuje się następująco:

	MAX iteracyjnie	**Wynik *rea tlmax()***	**Wartość z *float.h***
Float16	6.55e4	6.55e4	nd.
Float32	3.4028235e38	3.4028235e38	3.40282347e+38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e+308

2 Sprawdzanie poprawności wyrażenia Kahan'a

William Kahan, matematyk i informatyk specjalizujący się w metodach numerycznych, który za swój wkład w opracowanie standardu liczb zmiennoprzecinkowych otrzymał w 1989 roku nagrodę Turinga, stwierdził, że epsilon maszynowy *macheps* można otrzymać obliczając wyrażenie:

$$macheps = 3 \cdot (4/3 - 1) \tag{3}$$

Eksperymentalnie w języku Julia dokonaliśmy weryfikacji słuszności powyższego twierdzenia dla wszystkich typów zmiennopozycyjnych: *Float16, Float32 i Float64*.

Wykonując obliczenia w programie *ex2.jl* napisanym w języku Julia otrzymane zostały następujące wyniki:

	***macheps* wg. wzoru Kahana**	***eps()***
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-1	2.220446049250313e-16

Zauważamy, że dla typu zmiennopozycyjnego *Float32* wzór okazuje się być poprawny, jednakże już dla typów *Float32* oraz *Float16* otrzymujemy wyniki, które od tych zwracanych przez funkcję *eps()* różnią się nieprawidłowym bitem znaku.

Reasumując, możliwe jest wyznaczenie *macheps* dla artymetyk zmiennopozycyjnych *half, single i double* za pomocą wzoru (3), pod warunkiem, że będziemy pamiętać o możliwości otrzymania nieprawidłowego znaku przed wartością liczbową.

3 Badanie rozmieszczenia liczb zmiennopozycyjnych w arytmetyce *Float64*

Przedmiotem następnego przeprowadzonego eksperymentu było sprawdzenie, że w arytmetyce *Float64* liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale $[1, 2]$ z krokiem $\delta = 2^-52$. Innymi słowy, każda liczba x leżąca w tym przedziale może być reprezentowana w następujący sposób:

$$x = 1 + k * \delta \tag{4}$$

Gdzie $k = 1, 2, ..., 2^52 - 1$ i $\delta = 2^-52$.

Aby przekonać się, że w istocie w arytmetyce *Float64* liczby są rozmieszczone w opisany sposób, przygotowany został program *ex3.jl*, którego zadaniem jest wypisać binarną reprezentację pierwszych 10 liczb *double* w przedziale $[1, 2]$. Wypisanie binarnej reprezentacji liczb odbywa się przy użyciu dostępnej dla programistów języka *Julia* funkcji *bits()*.

Po wykonaniu programu dla przedziału liczbowego $[1, 2]$ ukazuje się następujący wynik:

Pierwsze 10 liczb w przedziale $[1,2]$ w *Float64*
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000000111
0011111111110000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000001001
0011111111110000000000000000000000000000000000000000000000001010

Już po wypisaniu pierwszych 10 liczb w przedziale $[1,2]$ jesteśmy w stanie z powodzeniem zauważyć, że każda liczba $nastpna = poprzednia + 1$ (+01 binarnie).

Następnie dokonaliśmy sprawdzenia, jak rozmieszczone są liczby zmiennopozycyjne w arytmetyce *Float64* w przedziałach $[1/2,1]$ oraz $[2,4]$. Wypisując pierwsze 10 liczb z pierwszego przedziału otrzymaliśmy wynik:

Pierwsze 10 liczb w przedziale $[1/2,1]$ w *Float64*
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000001010
0011111111110000000000000000000000000000000000000000000000001100
0011111111110000000000000000000000000000000000000000000000001110
0011111111110000000000000000000000000000000000000000000000010000
0011111111110000000000000000000000000000000000000000000000010010
0011111111110000000000000000000000000000000000000000000000010100

Zauważamy, że dla przedziału $[1/2,1]$ każda liczba $nastpna = poprzednia + 2$ (+10 binarnie). Wartości liczbowe w przedziale są dwukrotnie bardziej oddalone od siebie, niż wcześniej - stąd $\delta = 2^{-53}$.

Analogicznie dla przedziału $[2,4]$:

Pierwsze 10 liczb w przedziale $[2,4]$ w *Float64*
0011111111110000000000000000000000000000000000000000000000000000
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101

Zauważamy, że dla przedziału $[2,4]$ "gęstość" liczb jest dwukrotnie większa z uwagi na dwukrotnie większy przedział względem przedziału $[1,2]$ - stąd $\delta = 2^{-51}$.

Reasumując, eksperymentalnie przekonaliśmy się, że dla danych przedziałów liczbowych liczby rozmieszczone są z określoną regularnością i da się je przedstawić w postaci $x = 1 + k \cdot \delta$.

4 Eksperymentalne wyznaczanie w arytmetyce *Float64* liczby zmiennopozycyjnej x z przedziału $[1, 2]$, dla której dochodzi do sprzeczności $x * (1/x) \neq 1$

Naszym następnym zadaniem było napisanie w języku *Julia* programu komputerowego, który znajdzie taką liczbę zmiennopozycyjną zapisaną w arytmetyce *Float64*, dla której dojdzie do sprzeczności:

$$x * \frac{1}{x} \neq 1 \tag{5}$$

Przygotowany w tym celu program, którego kod źródłowy dostępny jest w dołączonym do sprawozdania pliku *ex4.jl* wyznacza najmniejszą spośród takich liczb realizując następujący algorytm:

1. Przypisz $x := 1 + eps(Float64)$
2. Dopóki $x * \frac{1}{x} = 1$ przypisz $x := x + eps(Float64)$
3. Zwróć x

Zaczynamy od sprawdzenia, czy dla najmniejszej możliwej liczby zmiennopozycyjnej większej od 1 w arytmetyce *Float64* możliwe jest otrzymanie sprzeczności. Dopóki sprzeczność nie zostanie osiągnięta, powtarzamy krok (2) naszego algorytmu.

Najmniejszą eksperymentalnie wyznaczoną liczbą która spełnia sprzeczność (5) okazała się być liczba:

`1.000000057228997.`

Do sprzeczności doszliśmy po `257736490` iteracjach naszego algorytmu, co oznacza, że jest to 257736490 najmniejsza liczba zmiennoprzecinkowa możliwa do zapisania w arytmetyce *Float64* w kolejności począwszy od 1.

Najważniejszym wnioskiem z tego zadania jest fakt, że operacje przeprowadzane na liczbach zmiennoprzecinkowych zawsze obarczone są pewnym błędem przybliżenia, do którego dochodzi w konsekwencji przeprowadzania pewnych zaokrągleń koniecznych do otrzymania prawidłowej reprezentacji liczby. Zwielokrotnienie przybliżeń w skrajnym przypadku, do których należy ten przebadany powyżej, może doprowadzić do poważnych konsekwencji w postaci otrzymania skrajnie niepoprawnych wyników obliczeń.

5 Eksperymentalne porównanie poprawności otrzymanych wyników dla różnych algorytmów obliczania iloczynu skalarnego dwóch wektorach na liczbach zmiennoprzecinkowych

Kolejny spośród przeprowadzonych przez nas eksperymentów miał na celu dokonanie poprawności wyników otrzymanych przy zastosowaniu czterech różnych algorytmów w problemie obliczania iloczynu skalarnego dwóch wektorów:

$$x = [2.718281828, 3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$
$$y = [1486.2497, 878366.9879, 22.37492, 4773714.647, 0.000185049]$$

Zaimplementowane zostały cztery różne algorytmy na różne sposoby obliczające iloczyn skalarny zgodnie ze specyfikacją zadania, których implementacja w języku *Julia* dostępna jest w pliku źródłowy *ex5.jl* dołączonym do niniejszego sprawozdania:

1. Algorytm obliczający sumę "w przód": $\sum_{i=1}^{n} x_i * y_i$
2. Algorytm obliczający sumę "w tył": $\sum_{i=n}^{1} x_i * y_i$

3. Algorytm "od największego do najmniejszego" [malejący]: (1) dodaj dodatnie liczby w porządku od największego do najmniejszego, (2) dodaj ujemne liczby w porządku od najmniejszego do największego, (3) dodaj do siebie obliczone sumy częściowe
4. Algorytm "od najmniejszego do największego" [rosnący]: (1) dodaj dodatnie liczby w porządku od najmniejszego do największego, (2) dodaj ujemne liczby w porządku od największego do najmniejszego, (3) dodaj do siebie obliczone sumy częściowe

Porównanie wyników działania algorytmów podczas operowania na liczbach zmiennoprzecinkowych w artymetyce *Float32* oraz *Float64* prezentuje się następująco:

	Wyniki w pojedyńczej precyzji	**Wyniki w podwójnej precyzji**
Algorytm *w przód*	-0.4999443	1.0251881368296672e-10
Algorytm *w tył*	-0.4543457	-1.5643308870494366e-10
Algorytm *malejący*	-0.5	0.0
Algorytm *rosnący*	-0.5	0.0

Otrzymane wyniki przeprowadzonych operacji dla liczb zmiennoprzecinkowych w arytmetyce *Float32* i *Float64* następnie mieliśmy porównać z prawidłową wartością iloczynu skalarnego wektorów x i y, która z dokładnością do 15 cyfr wynosi:

$$1.0065710700000010 * 10^-11$$

Zauważamy, że zastosowanie żadnego z powyżej opisanych algorytmów nie pozwoliło nam na otrzymanie poprawnego wyniku zarówno w arytmetyce *single*, jak i *double*, jednak wzrost precyzji zapisu miał bezpośredni wpływ na przybliżane wyniku do oczekiwanej, prawidłowej wartości.

6 Odejmowanie bliskich wartości liczbowych

Przedmiotem naszych kolejnych rozważań będzie sytuacja, w której zmuszeni jesteśmy odejmować zbliżone do siebie wartości liczbowe. Będziemy analizować wyrażenie:

$$y = \sqrt{x^2 + 1} - 1. \tag{6}$$

Zauważamy, że dla małych x zmuszeni będziemy odejmować od siebie bliskie sobie liczby, czego skutkiem będzie zmniejszenie liczby cyfr znaczących. Możemy jednak przekształcić wyrażenie y tak, aby uniknąć "niebezpiecznego" odejmowania:

$$y = (\sqrt{x^2 + 1} - 1)\frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1} = \frac{x^2}{\sqrt{x^2 + 1} + 1}. \tag{7}$$

Aby przekonać się, że przekształcenie wyrażenia y pozwoli nam na dokonanie bardziej precyzyjnych obliczeń napisany został program w języku *Julia*, który dla kolejnych wartości argumentu $x = 8^-1, 8^-2, ...$ obliczał wartości funkcji y zapisanej w formie wyjściowej i przekształconej w arytmetyce *Float64*.

Wyniki przeprowadzonych obliczeń zaprezentowano w poniższej tabeli:

	$y = \sqrt{x^2 + 1} - 1$	$y = \frac{x^2}{\sqrt{x^2+1}+1}$
1	0.0077822185373186414	0.0077822185373187065
2	0.00012206286282867573	0.00012206286282875901
3	1.9073468138230965e-6	1.907346813826566e-6
4	2.9802321943606103e-8	2.9802321943606116e-8
5	4.656612873077393e-10	4.6566128719931904e-10
6	7.275957614183426e-12	7.275957614156956e-12
7	1.1368683772161603e-13	1.1368683772160957e-13
8	1.7763568394002505e-15	1.7763568394002489e-15
9	0.0	2.7755575615628914e-17
10	0.0	4.336808689942018e-19

	$y = \sqrt{x^2+1} - 1$	$y = \frac{x^2}{\sqrt{x^2+1}+1}$
11	0.0	6.776263578034403e-21
12	0.0	1.0587911840678754e-22
13	0.0	1.6543612251060553e-24
14	0.0	2.5849394142282115e-26
15	0.0	4.0389678347315804e-28
16	0.0	6.310887241768095e-30
17	0.0	9.860761315262648e-32
18	0.0	1.5407439555097887e-33
19	0.0	2.407412430484045e-35
20	0.0	3.76158192263132e-37
21	0.0	5.877471754111438e-39

Zauważamy, że dla $x = (\frac{1}{8})^9$ i mniejszych, choć wyrażenie (6) jest równe wyrażeniowi (7) zmniejszenie liczby cyfr znaczących drastycznie wpływa na poprawność finalnie otrzymanego wyniku.

Dlaczego obliczając wartości funkcji y korzystając z pierwotnego wyrażenia dochodzi do problemów powyżej ósmej iteracji algorytmu? Dzieje się tak, ponieważ dochodzi do przybliżenia $\sqrt{x^2+1} \approx 1$. Wówczas wynikiem operacji $\sqrt{x^2+1} - 1$ jest liczba zero.

W konsekwencji aby otrzymać wiarygodne wyniki przeprowadzonych obliczeń w analogicznych sytuacjach powinniśmy dążyć do takiego przekształcenia danej nam funkcji liczbowej, aby uniknąć "niebezpiecznego" odejmowania.

7 Obliczanie przybliżonej wartości pochodnej funkcji

Finalnym zadaniem w ramach pierwszej listy zadań laboratoryjnych z obliczeń naukowych było skorzystanie ze wzoru na przybliżoną wartość pochodnej $f(x)$ w punkcie x, aby obliczyć wartość pochodnej funkcji $f(x) = sin(x) + cos(3x)$w punkcie $x_0 = 1$:

$$f(x_0) \approx f'(\tilde{x_0}) = \frac{f(x_0+h)f(x_0)}{h} \tag{8}$$

oraz obliczyć błędy przybliżenia $|f'(x_0) - f'(\tilde{x_0})|$ dla $h = (1/2)^n$, gdzie $n = 0, 1, 2, ..., 54$.

Wyniki przeprowadzonych obliczeń prezentują się następująco:

	$f'(\tilde{x_0})$	$\lvert f'(x_0) - f'(\tilde{x_0})\rvert$	$1+h$
0	2.0179892252685967	1.9010469435800585	2.0
1	1.8704413979316472	1.753499116243109	1.5
2	1.1077870952342974	0.9908448135457593	1.25
3	0.6232412792975817	0.5062989976090435	1.125
4	0.3704000662035192	0.253457784514981	1.0625
5	0.24344307439754687	0.1265007927090087	1.03125
6	0.18009756330732785	0.0631552816187897	1.015625
7	0.1484913953710958	0.03154911368255764	1.0078125
8	0.1327091142805159	0.015766832591977753	1.00390625
⋮	⋮	⋮	⋮
26	0.11694233864545822	5.6956920069239914e-8	1.0000000149011612
27	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
28	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
29	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
30	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
31	0.11694216728210449	1.1440643366000813e-7	1.0000000004656613
⋮	⋮	⋮	⋮

	$f'(\tilde{x_0})$	$\lvert f'(x_0) - f'(\tilde{x_0})\rvert$	$1+h$
38	0.116943359375	1.0776864618478044e-6	1.000000000003638
39	0.11688232421875	5.9957469788152196e-5	1.000000000001819
40	0.1168212890625	0.0001209926260381522	1.0000000000009095
41	0.116943359375	1.0776864618478044e-6	1.0000000000004547
42	0.11669921875	0.0002430629385381522	1.0000000000002274
43	0.1162109375	0.0007313441885381522	1.0000000000001137
44	0.1171875	0.0002452183114618478	1.0000000000000568
45	0.11328125	0.003661031688538152	1.0000000000000284
46	0.109375	0.007567281688538152	1.0000000000000142
47	0.109375	0.007567281688538152	1.000000000000007
48	0.09375	0.023192281688538152	1.0000000000000036
49	0.125	0.008057718311461848	1.0000000000000018
50	0.0	0.11694228168853815	1.0000000000000009
51	0.0	0.11694228168853815	1.0000000000000004
52	-0.5	0.6169422816885382	1.0000000000000002
53	0.0	0.11694228168853815	1.0
54	0.0	0.11694228168853815	1.0

Zauważamy, że dokładność przeprowadzonych obliczeń w bardzo dużej mierze zależy od odpowiednio dobranej wartości h - zbyt duże h wpływa na przybliżenie znacznie odbiegające od rzeczywistej wartości pochodnej funkcji w punkcie x_0, jednak dalsze zmniejszanie h dla $n \geqslant 29$ również ostatecznie doprowadza do mniej dokładnych obliczeń.

Wpływ na niedokładne obliczenia ma także odejmowanie bliskich sobie wartości liczbowych, co dokładniej przeanalizowaliśmy w poprzednim zadaniu.

8 Wnioski ogólne z listy laboratoryjnej

Najważniejszym wnioskiem, który nasuwa się po przeprowadzeniu powyższych eksperymentów jest fakt, że działania na liczbach zmiennopozycyjnych w arytmetykach *half, single* czy *double* zawsze są obarczone pewnym błędem przybliżenia, który wynika ze sposobu reprezentacji liczby rzeczywistej w systemach informatycznych.

Dokonaliśmy analizy różnych czynników, które mają wpływ na dokładność przeprowadzanych przez nas obliczeń. Ich znajomość pozwala zminimalizować błędy obliczeniowe, zaś brak ich zrozumienia, jak się przekonaliśmy, może być katastrofalny w skutkach.