Lista nr 1

1. Zadanie 1

1.1. Opis problemu

Pierwszy problem polega na wyznaczeniu w sposób iteracyjny istotnych stałych związanych z arytmetyką zmiennoprzecinkową w różnych precyzjach.

- Epsilon maszynowy : najmniejszy $macheps > 0$ taki że $fl(1.0 + macheps) > 1.0$ i $fl(1.0 + macheps) = 1 + macheps$
- Eta : najmniejsza eta taka że $eta > 0.0$
- Max : największa liczba możliwa do zapisania w danym systemie

1.2. Opis algorytmu

1.2.1. Epsilon maszynowy

Szukając $macheps$, sprawdzam kolejne potęgi $\frac{1}{2}$, dopóki nie dojdę do takiej, która w wyniku dodania do 1.0 da 1.0. Według standardu IEEE 754 zaokrąglenia następują do najbliższej parzystej liczbie (czyli kończącej się na bicie 0). Z racji że 1.0 jest zgodnie z tym parzysta, to algorytm zakończy się rzeczywiście przy pierwszej takiej potędze $\frac{1}{2}$, że wynik dodania do 1.0 nie mieści się dokładnie w precyzji. Wtedy poprzednia potęga, to szukany $macheps$.

```
eps = 1.0
while (1.0 + eps / 2.0) > 1.0
    eps /= 2.0
end
return eps
```

Program 1: Algorytm liczenia $macheps$

1.2.2. Eta

Na podobnej zasadzie szukam liczby eta. Sprawdzam kolejne potęgi $\frac{1}{2}$ aż któraś nie zmieści się już w precyzji i zaokrągli do 0.0, które jest parzyste. Wtedy poprzednia potęga to eta.

```
eta = 1.0
while (eta / 2.0) > 0.0
    eta /= 2.0
end
return eta
```

Program 2: Algorytm liczenia eta

1.2.3. Max

Tutaj podzieliłem algorytm na dwie części. Najpierw „maksymalizuje cechę”, szukając największej potęgi 2 mieszczącej się w precyzji. Następnie „wypełniam mantysę”, dodając po kolei coraz mniejsze potęgi 2, aż nie dojdę do liczby, której nie można już zapisać dokładnie w danej precyzji i spowoduje overflow. Wtedy poprzednia liczba to max.

```
max = 1.0
while !isinf(max * 2.0)
    max *= 2.0
end
step = max / 2.0
while !isinf(max + step)
    my_max += step
    step /= 2.0
end
return max
```

Program 3: Algorytm liczenia max

1.3. Wyniki

	Float16	Float32	Float64
$macheps$	0.000977	$1.1920929 \cdot 10^{-7}$	$2.220446049250313 \cdot 10^{-16}$
eta	$6.0 \cdot 10^{-8}$	$1.0 \cdot 10^{-45}$	$5.0 \cdot 10^{-324}$
max	$6.55 \cdot 10^{4}$	$3.4028235 \cdot 10^{38}$	$1.7976931348623157 \cdot 10^{308}$

Tabela 1: Wyniki otrzymane w przeprowadzonych doświadczeniach

	Float16	Float32	Float64
`eps(T)`	0.000977	$1.1920929 \cdot 10^{-7}$	$2.220446049250313 \cdot 10^{-16}$
`nextfloat(T(0.0))`	$6.0 \cdot 10^{-8}$	$1.0 \cdot 10^{-45}$	$5.0 \cdot 10^{-324}$
`floatmax(T)`	$6.55 \cdot 10^{4}$	$3.4028235 \cdot 10^{38}$	$1.7976931348623157 \cdot 10^{308}$

Tabela 2: Wartości wyliczone przy pomocy biblioteki standardowej

	float	double
`EPSILON`	$1.1920928955078125 \cdot 10^{-7}$	$2.22044604925031308 \cdot 10^{-16}$
`MAX`	$3.4028234663852886 \cdot 10^{38}$	$1.79769313486231571 \cdot 10^{308}$

Tabela 3: Wartości znajdujące się w pliku nagłówkowym float.h

1.4. Wnioski i odpowiedzi

- *Jaki ma związek macheps z precyzją arytmetyki?*

macheps oznacza względną odległość pomiędzy kolejnymi liczbami maszynowymi. Natomiast precyzja arytmetyki oznacza największy względny błąd reprezentacji. Stąd jeżeli mamy do czynienia z zaokrąglaniem to precyzja $= \frac{1}{2} macheps$, a gdy z obcinaniem to precyzja $= macheps$.

- *Jaki ma związek liczba eta z liczbą MIN_{sub}?*

MIN_{sub} to najmniejsza liczba większa od zera możliwa do zapisania w danej precyzji. Należy do liczb zdenormalizowanych. Zgodnie z definicją jest równa naszej liczbie eta.

- *Co zwracają* `floatmin(T)` *i jaki jest ich związek z* MIN_{nor}*?*

`floatmin(Float32)` i `floatmin(Float64)` zwracają kolejno liczby $1.1754944 \cdot 10^{-38}$ i $2.2250739 \cdot 10^{-308}$. Zgodnie z dokumentacją są to najmniejsze liczby znormalizowane możliwe do zapisania w danej precyzji. Tym samym są równe MIN_{nor}.

2. Zadanie 2

Według Kahana można uzyskać epsilon maszynowy wyliczając wyrażenie `3(4/3-1)-1`. Stwierdzenie to zweryfikowałem w wielu precyzjach w języku Julia za pomocą prostych poleceń:

```
Float16(3) * (Float16(4) / Float16(3) - Float16(1)) - Float16(1) == -eps(Float16)
```

→ true

```
Float32(3) * (Float32(4) / Float32(3) - Float32(1)) - Float32(1) == eps(Float32)
```

→ true

```
Float64(3) * (Float64(4) / Float64(3) - Float64(1)) - Float64(1) == -eps(Float64)
```

→ true

Żeby to zrozumieć wystarczy spojrzeć na kolejne etapy liczenia tego wyrażenia. Dla przykładu użyję arytmetyki Float16.

`4/3`	$(1.0101010101)_2 \cdot 2^0$
`4/3-1`	$(1.0101010100)_2 \cdot 2^{-2}$
`3(4/3-1)`	$(1.1111111110)_2 \cdot 2^{-1}$
`3(4/3-1)-1`	$-(1.0000000000)_2 \cdot 2^{-11}$

Tabela 4: Kolejno wykonywane operacje i ich wyniki w Float16

Wiadomo, że wynikiem `3(4/3-1)` powinno być 1.0, a jednak w arytmetyce zmiennoprzecinkowej otrzymałem wynik, który wynosi $1 \pm macheps$. Różnica w znaku jest związana z tym, że okres $\frac{1}{3}$ w rozwinięciu dwójkowym ma długość 2 i zależnie od tego czy precyzja jest parzysta (Float32), czy nieparzysta (Float16, Float64), utniemy dokładny wynik na ostatnim bicie 0 albo na 1.

3. Zadanie 3

Ten problem polega na sprawdzeniu równomiernego rozmieszczenia liczb maszynowych na odcinkach $[1, 2]$, $\left[\frac{1}{2}, 1\right]$, $[2, 4]$. Zgodnie z teorią z wykładu wiem, że odległość pomiędzy dwiema sąsiednimi liczbami maszynowymi z cechą c wynosi $2^{c-(t-1)}$. Podstawiając dla naszych odcinków kolejno $c = 0, -1, 1$ oraz $t - 1 = 52$, otrzymałem $2^{-52}, 2^{-53}, 2^{-51}$. Z racji, że te odcinki mają w sobie bardzo dużo liczb, nie sprawdzałem każdej z nich, a jedynie wybrałem paru przedstawicieli i sprawdzałem ich reprezentacje bitową. Eksperyment potwierdził teorię.

Liczba	Reprezentacja binarna
1.0	`0:01111111111:0000000000000000000000000000000000000000000000000000`
$1.0 + 2^{-52}$	`0:01111111111:0000000000000000000000000000000000000000000000000001`
$1.0 + 2^{-51}$	`0:01111111111:0000000000000000000000000000000000000000000000000010`
$\frac{4}{3}$	`0:01111111111:0101010101010101010101010101010101010101010101010101`
$\frac{4}{3} + 2^{-52}$	`0:01111111111:0101010101010101010101010101010101010101010101010110`
2.0	`0:10000000000:0000000000000000000000000000000000000000000000000000`
$2.0 - 2^{-52}$	`0:01111111111:1111111111111111111111111111111111111111111111111111`

Tabela 5: Test dla liczb z odcinka $[1, 2]$

Liczba	Reprezentacja binarna
2.0	`0:10000000000:0000000000000000000000000000000000000000000000000000`
$2.0 + 2^{-51}$	`0:10000000000:0000000000000000000000000000000000000000000000000001`
4.0	`0:10000000001:0000000000000000000000000000000000000000000000000000`
$4.0 - 2^{-51}$	`0:10000000000:1111111111111111111111111111111111111111111111111111`

Tabela 6: Test dla liczb z odcinka $[2, 4]$

Liczba	Reprezentacja binarna
0.5	`0:01111111110:0000000000000000000000000000000000000000000000000000`
$0.5 + 2^{-53}$	`0:01111111110:0000000000000000000000000000000000000000000000000001`
1.0	`0:01111111111:0000000000000000000000000000000000000000000000000000`
$1.0 - 2^{-53}$	`0:01111111110:1111111111111111111111111111111111111111111111111111`

Tabela 7: Test dla liczb z odcinka $\left[\frac{1}{2}, 1\right]$

4. Zadanie 4

Celem tego zadania jest wyznaczenie najmniejszej takiej liczby $1 < x < 2$, że $fl\big(x \cdot fl\big(\frac{1}{x}\big)\big) \neq 1$. Zgodnie z poprzednim zadaniem wystarczy iteracyjnie przesuwać się od 1 krokami 2^{-52}, żeby nie pominąć żadnej liczby w przedziale $[1, 2]$.

```
x = 1.0
while x * (1.0 / x) == 1.0
    x += 2^(-52)
end
return x
```

Wynikiem była liczba 1.000000057228997096814282485865987837314605712890625. Albo inaczej 257736490 liczba z kolei następująca po liczbie 1.

5. Zadanie 5

5.1. Opis problemu

Problem polega na zbadaniu wyliczenia iloczynu skalarnego wektorów x i y poprzez dodawanie w różnych kolejnościach.

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

Te dwa wektory są do siebie prawie prostopadłe, przez co iloczyn skalarny jest bliski zeru. Dokładny wynik powinien wynieść $-1.00657107000000 \cdot 10^{-11}$.

5.2. Opis algorytmu

Zacząłem od wyliczenia wektora składającego się z wymnożonych elementów x i y.

$$m = [x_1y_1, x_2y_2, x_3y_3, x_4y_4, x_5y_5]$$
$$m \approx [4040.046, -2759471.3, -31.64292, 2755462.9, 0.0000557]$$

Kolejne metody liczenia iloczynu skalarnego realizuję poprzez sumowanie elementów wektora m w różnych kolejnościach.

5.3. Wyniki

Float64			
Kolejność	**Działanie**	**Wynik**	**Błąd względny**
W przód	$m_1 + m_2 + m_3 + m_4 + m_5$	$1.0251881 \cdot 10^{-10}$	11.185
W tył	$m_5 + m_4 + m_3 + m_2 + m_1$	$-1.5643309 \cdot 10^{-10}$	14.541
Od max do min	$(m_4 + m_1 + m_5) + (m_2 + m_3)$	0.0	1.0
Od min do max	$(m_5 + m_1 + m_4) + (m_3 + m_2)$	0.0	1.0

Tabela 8: Wyniki w precyzji Float64

Float32			
Kolejność	**Działanie**	**Wynik**	**Błąd względny**
W przód	$m_1 + m_2 + m_3 + m_4 + m_5$	-0.4999443	$4.9668 \cdot 10^{10}$
W tył	$m_5 + m_4 + m_3 + m_2 + m_1$	-0.4543457	$4.5138 \cdot 10^{10}$
Od max do min	$(m_4 + m_1 + m_5) + (m_2 + m_3)$	-0.5	$4.9674 \cdot 10^{10}$
Od min do max	$(m_5 + m_1 + m_4) + (m_3 + m_2)$	-0.5	$4.9674 \cdot 10^{10}$

Tabela 9: Wyniki w precyzji Float32

5.4. Wnioski

Najmniejszym błędem zostało obarczone sumowanie od tyłu w precyzji Float64. Natomiast jakakolwiek próba sumowania we Float32 skończyła ogromnym błędem względnym rzędu 10^{10}.

6. Zadanie 6

W tym zadaniu obliczam wartość dwóch różnych wyrażeń $f(x)$ i $g(x)$, które są sobie matematycznie równoważne, dla zmniejszającego się argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$

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

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

x	$f(x)$	$g(x)$
8^{-1}	$7.78221854 \cdot 10^{-03}$	$7.78221854 \cdot 10^{-03}$
8^{-2}	$1.22062863 \cdot 10^{-04}$	$1.22062863 \cdot 10^{-04}$
8^{-3}	$1.90734681 \cdot 10^{-06}$	$1.90734681 \cdot 10^{-06}$
8^{-4}	$2.98023219 \cdot 10^{-08}$	$2.98023219 \cdot 10^{-08}$
8^{-5}	$4.65661287 \cdot 10^{-10}$	$4.65661287 \cdot 10^{-10}$
8^{-6}	$7.27595761 \cdot 10^{-12}$	$7.27595761 \cdot 10^{-12}$
8^{-7}	$1.13686838 \cdot 10^{-13}$	$1.13686838 \cdot 10^{-13}$
8^{-8}	$1.77635684 \cdot 10^{-15}$	$1.77635684 \cdot 10^{-15}$
8^{-9}	0.00000000	$2.77555756 \cdot 10^{-17}$
8^{-10}	0.00000000	$4.33680869 \cdot 10^{-19}$
8^{-11}	0.00000000	$6.77626358 \cdot 10^{-21}$
...	...	...
8^{-48}	0.00000000	$1.00538234 \cdot 10^{-87}$
8^{-49}	0.00000000	$1.57090991 \cdot 10^{-89}$
8^{-50}	0.00000000	$2.45454673 \cdot 10^{-91}$

Tabela 10: Porównanie wyliczania obu wyrażeń

Choć wyrażenia były równoznaczne, to szybko zaczęły od siebie odbiegać. Problem pojawia się w wyrażeniu $x^2 + 1$ i jest powodowany przez dodawanie do 1 liczby względnie bardzo małej. Gdy $x = 8^{-9} = 2^{-27}$ to $x^2 = 2^{-54}$, a jako że typ Float64 przeznacza 53 bity na mantysę, to wynik dodawania $x^2 + 1$ jest obcinany do wartości 1.0. Wyrażenie $g(x)$ nie jest obarczone tym problemem z powodu posiadania dodatkowo x^2 w liczniku i tym samym daje bliższe prawdzie wyniki.

7. Zadanie 7

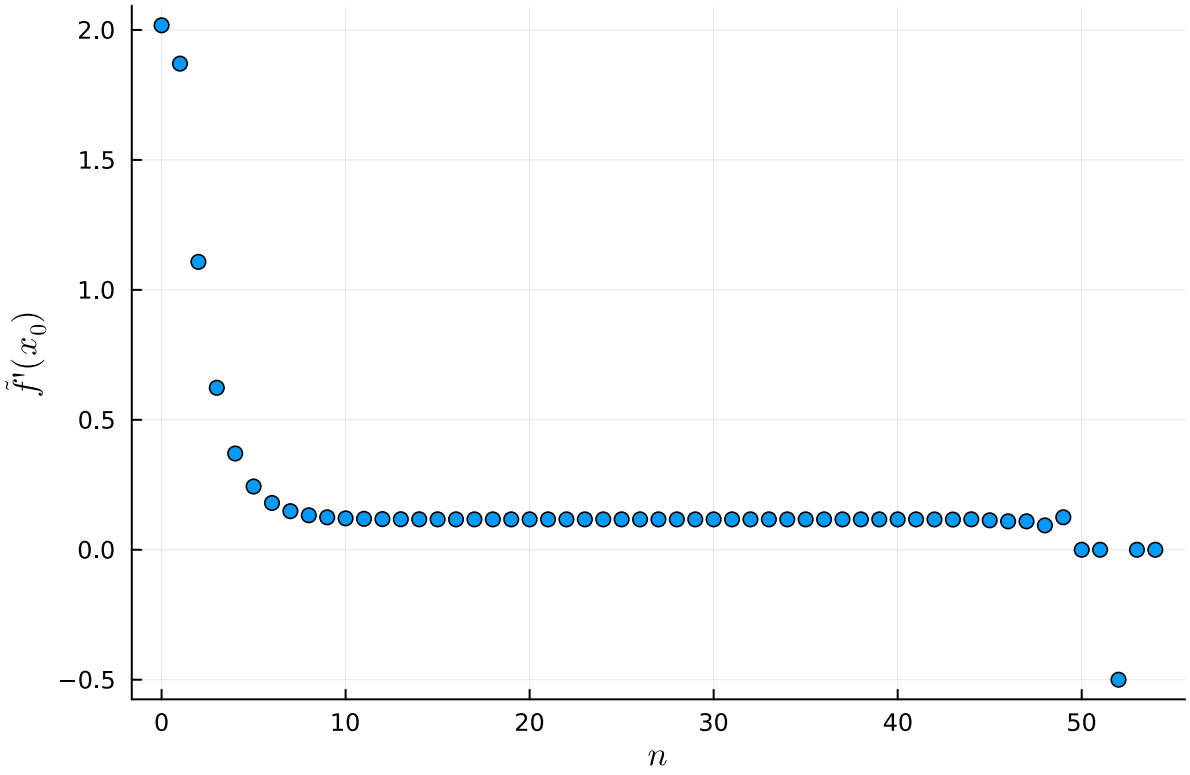
7.1. Opis problemu

Do wyliczenia przybliżonej pochodnej funkcji w punkcie można skorzystać z definicji granicznej. Tym samym otrzymuje się taki wzór:

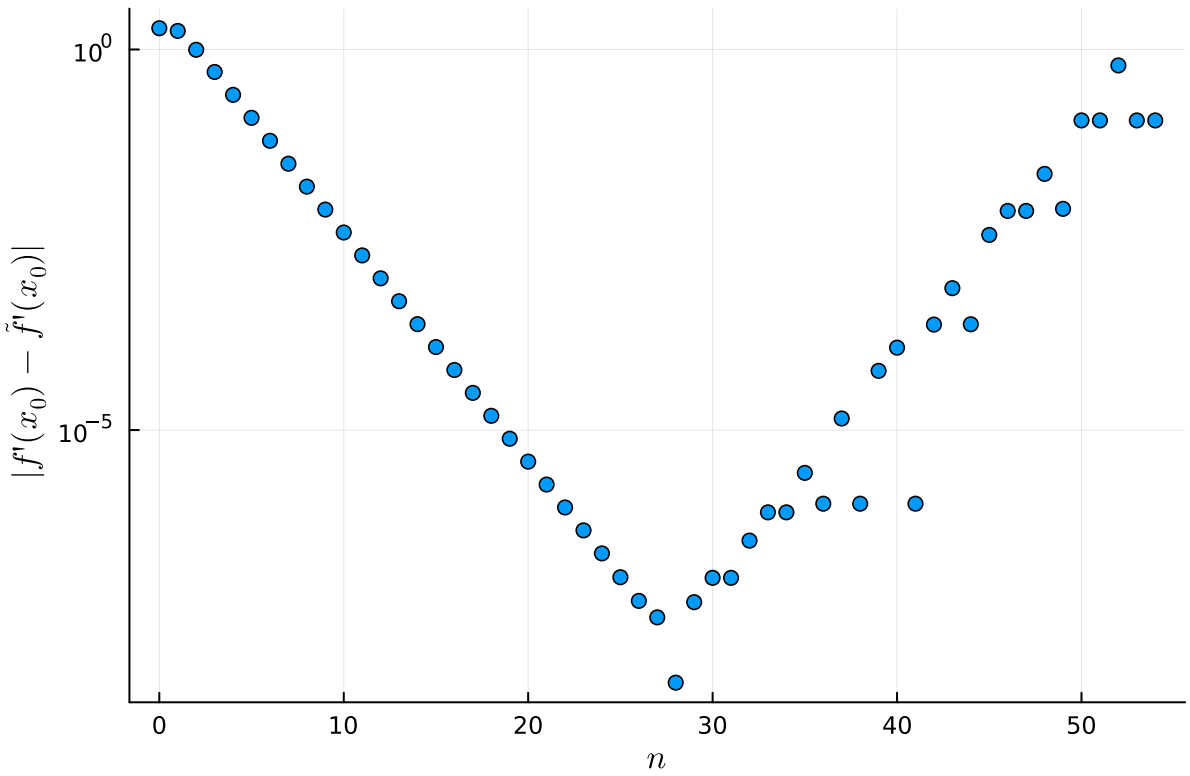
$$f'(x_0) \approx \tilde{f}'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

To zadanie polega na wyliczeniu wartości i błędu pochodnej funkcji $\sin(x) + \cos(3x)$ w punkcie $x_0 = 1$ dla $h = 2, 2^{-1}, 2^{-2}, ..., 2^{-54}$. Pochodną wyliczyłem ręcznie i wyniosła ona $f'(x) = \cos(x) - 3\sin(3x)$, co w punkcie $x_0 = 1$ daje w przybliżeniu wartość 0.1169422817.

7.2. Wyniki



Rysunek 1: Wyliczone przybliżone wartości pochodnej



Rysunek 2: Błąd bezwzgledny przybliżeń (w skali logarytmicznej)

7.3. Wnioski

Z wykresów można odczytać, że od pewnego miejsca błąd zaczyna rosnąć zamiast maleć. Minimalny błąd zostaje osiągnięty dla $n = 28$. Jest to złożenie paru czynników. Pierwszy związany jest z odejmowaniem bliskich siebie liczb w wyrażeniu $f(x_0 + h) - f(x_0)$. Zgodnie z wykładem takie odejmowanie $x - y$ wiąże się z dużym mnożnikiem błędu reprezentacji $\frac{|x| + |y|}{|x-y|}$. Następnie to wyrażenie jest dzielone przez przez $|h|$, i razem z tym błąd dostaje kolejny mnożnik $\frac{1}{|h|}$. Do pewnego momentu algorytm szybciej zbiega niż jest w stanie popsuć to błąd, ale od pewnego momentu zostaje przez niego wyprzedzony.