Obliczenia naukowe - Lista 2

Wojciech Pakulski (250350)

1 Wpływ niewielkiej zmiany danych na wyniki dla iloczynu skalarnego wektorów

Zadanie polegało na powtórzeniu zadania 5 z listy 1, ale przy usunięciu ostatniej 9 z x_4 i ostatniej 7 z x_5.

1.1 Przypomnienie zadania 5 z listy 1

Należało zaimplementować 4 algorytmy liczące iloczyn skalany dwóch wektorów:

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

y = [1486.2497, 878366.9879, −22.37492, 4773714.647, 0.000185049].

Trzeba było napisać poniższe algorytmy, licząc sumę na cztery sposoby:

(a) "w przód" $\sum_{i=1}^{n} x_i y_i$, tj. algorytm

```
S := 0
for i := 1 to n do
        S := S + x_i · y_i
end for
```

(b) "w tył" $\sum_{i=1}^{n} x_i y_i$, tj. algorytm

```
S := 0
for i := n downto 1 do
        S := S + x_i · y_i
end for
```

(c) 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)

(d) od najmniejszego do największego (przeciwnie do metody (c)).

1.2 Jaki wpływ na wyniki mają te niewielkie zmiany danych?

Ta minimalne zmiany, czyli usunięcie ostatniej 9 z x_4 i ostatniej 7 z x_5. Najpierw porównajmy wyniki dla danych:

x_1 = [2.718281828, −3.141592654, 1.414213562, 0.5772156649, 0.3010299957]

y = [1486.2497, 878366.9879, −22.37492, 4773714.647, 0.000185049].

x_2 = [2.718281828, −3.141592654, 1.414213562, 0.577215664, 0.301029995]

Lista 1	$x_1 \cdot y$ w Float32	$x_1 \cdot y$ w Float64
Algorytm (a)	- 0.4999443	1.0251881368296672e-10

Algorytm (b)	- 0.4543457	- 1.5643308870494366e-10
Algorytm (c)	- 0.5	0.0
Algorytm (d)	- 0.5	0.0

Kod źródłowy programu znajduje się w pliku zad1.jl.

Lista 2	$x_2 \cdot y$ **w Float32**	$x_2 \cdot y$ **w Float64**
Algorytm (a)	- 0.4999443	-0.004296342739891585
Algorytm (b)	- 0.4543457	-0.004296342998713953
Algorytm (c)	- 0.5	-0.004296342842280865
Algorytm (d)	- 0.5	-0.004296342842280865

Można zauważyć, że wyniki przeprowadzone we Float32 są takie same dla różniących się wartości wektora x. Spowodowane jest to zbyt małą precyzją tej arytmetyki, żeby wprowadzona przez nas zmiana była widoczna.

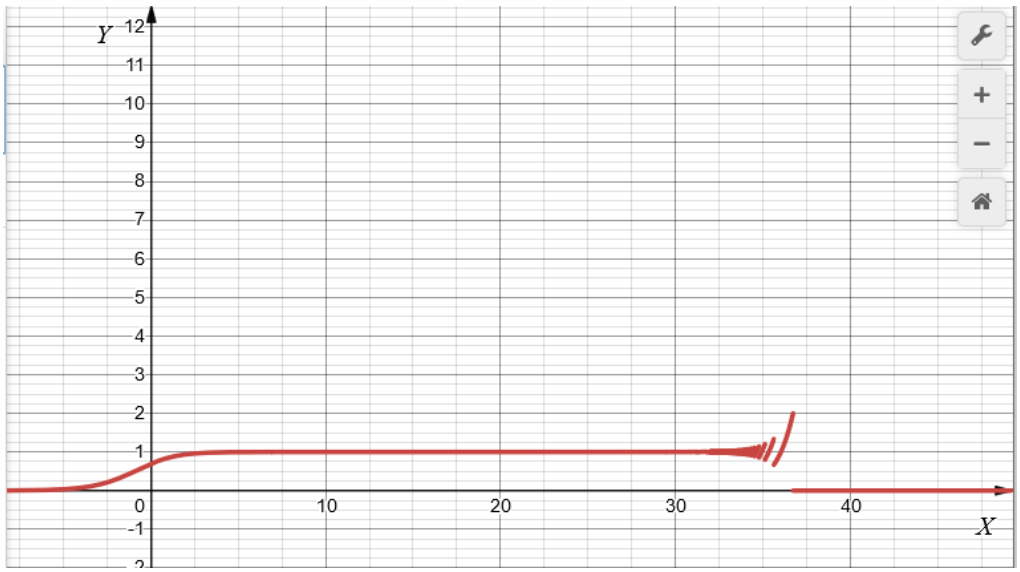
Pozornie powinno się wydawać, że taka minimalna zmiana niewiele wpłynie na wynik. Okazuje się jednak, że nawet one rażąco wpłynęły na wynik. Znaczy to, że zadanie jest źle uwarunkowane.

2 Rysowanie wykresów funkcji za pomocą programów do wizualizacji

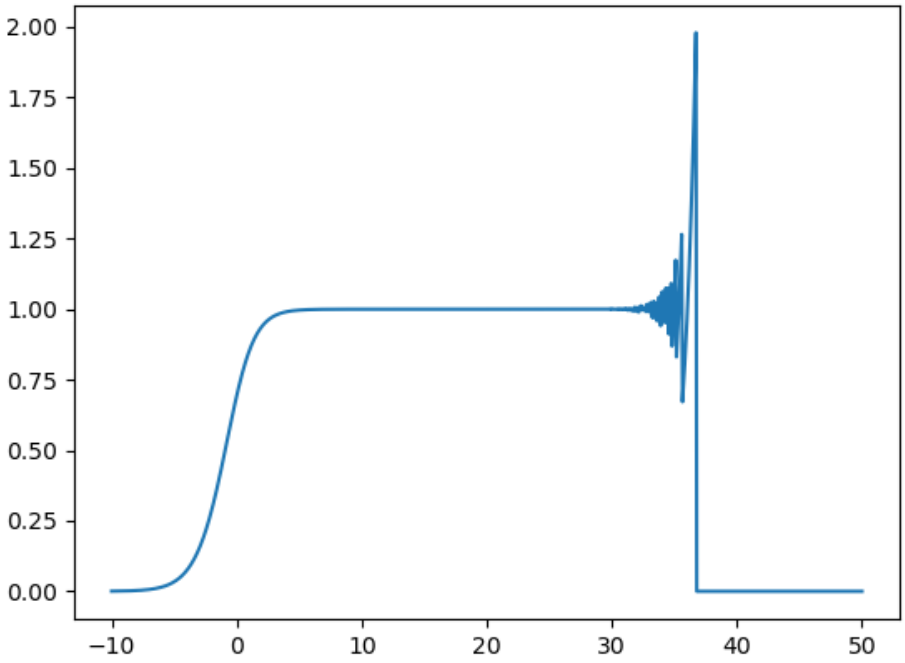
To zadanie polegało na narysowaniu wykresu funkcji *f(x)* za pomocą dwóch różnych programów do wizualizacji.

$$f(x) = e^x \, ln(1 + e^{-x})$$

Użyłem GeoGebry i biblioteki matplotlib (PyPlot).



Wykres funkcji f(x) w GeoGebrze



Wykres PyPlot funkcji f(x) w matplotlib

Granica *f(x)* w nieskończoności: $\lim_{x\to\infty} = e^x\, ln(1 + e^{-x}) = 1$

Wykresy zachowują się poprawnie do około x = 30. Mimo, że granica *f(x)* w nieskończoności jest równa 1, to według wykresu wynosi ona 0. Dzieje się tak, ponieważ dla większych iksów człon $ln(1 + e^{-x})$ arytmetyka zaokrągla tę wartość do $ln(1)$ co jest równe 0. Zanim wartość funkcji ustatkowuje się na zerze występują duże wahania, gdyż dokonujemy mnożenia bardzo dużej liczby (e^x) i liczby bliskiej 0.

Na podstawie tych obserwacji można stwierdzić, że algorytm wyliczania wartości tej funkcji jest bardzo niestabilny, bo małe błędy znacząco wpływają na wynik poprzez nawarstwianie. Należy, więc uważać na programy do wizualizacji, bo mogą wyświetlać niepoprawne i mylące wykresy przez redukcje cyfr znaczących rozwinięcia.

3 Rozwiązywanie układu równań liniowych

W zadaniu 3. mamy do rozwiązania układ równań liniowych $Ax = b$ za pomocą metody:

- eliminacji Gaussa (x =A\b)
- $wykorzystującej\ inwersję\ macierzy\ A\ x = A^{-1}b,$

gdzie **A** to macierz współczynników, a **b** – wektor prawych stron.

Macierz A można generować w następujący sposób:
(a) A = H_n, gdzie H_n jest macierzą Hilberta stopnia n wygenerowaną za pomocą funkcji A = *hilb(n)*

Kod źródłowy programu znajduje się w pliku zad3.jl. Wyniki obliczeń dla macierzy Hilberta H_n z rosnącym stopniem:

Rank	Cond()	Błąd przy metodzie eliminacji Gaussa	Błąd przy metodzie inwersji macierzy
2	19.28147006790397	5.661048867003676e-16	1.4043333874306803e-15
3	524.0567775860644	8.022593772267726e-15	0.0
4	15513.73873892924	4.137409622430382e-14	0.0
5	476607.25024259434	1.6828426299227195e-12	3.3544360584359632e-12
6	1.4951058642254665e7	2.618913302311624e-10	2.0163759404347654e-10
7	4.75367356583129e8	1.2606867224171548e-8	4.713280397232037e-9
8	1.5257575538060041e10	6.124089555723088e-8	3.07748390309622e-7
9	4.931537564468762e11	3.8751634185032475e-6	4.541268303176643e-6
10	1.6024416992541715e13	8.67039023709691e-5	0.0002501493411824886
11	5.222677939280335e14	0.00015827808158590435	0.007618304284315809
12	1.7514731907091464e16	0.13396208372085344	0.258994120804705
13	3.344143497338461e18	0.1103970111786826	5.331275639426837
14	6.200786263161444e17	1.4554087127659643	8.71499275104814
15	3.674392953467974e17	4.696668350857427	7.344641453111494
16	7.865467778431645e17	54.15518954564602	29.84884207073541
17	1.263684342666052e18	13.707236683836307	10.516942378369349
18	2.2446309929189128e18	9.134134521198485	7.575475905055309
19	6.471953976541591e18	9.720589712655698	12.233761393757726
20	1.3553657908688225e18	7.549915039472976	22.062697257870493

Jak widać macierz Hilberta jest macierzą źle uwarunkowaną, ponieważ błąd względny przeprowadzonych obliczeń rośnie wraz ze stopnia macierzy (i wzrostem uwarunkowania macierzy) dla obu metod.

(b) A = R_n, gdzie R_n jest losową macierzą stopnia n z zadanym wskaźnikiem uwarunkowania c wygenerowaną za pomocą funkcji A = *matcond(n, c).*

Wyniki obliczeń dla macierzy losowej R_n, $n \in \{5, 10, 20\}$ z rosnącym wskaźnikiem uwarunkowania $c \in \{1, 10, 10^3, 10^7, 10^{12}, 10^{15}\}$:

Rank	Cond()	Błąd przy metodzie eliminacji Gaussa	Błąd przy metodzie inwersji macierzy
5	1.0	1.7901808365247238e-16	1.7901808365247238e-16
5	10.0	3.2934537262255424e-16	2.0471501066083611e-16
5	1000.0	3.7525702466146135e-14	3.32325934484418e-14
5	1.0e7	3.5408574044522506e-10	2.8515813671787943e-10
5	1.0e12	1.6469258557206415e-5	1.90925502872348e-5
5	1.0e16	0.4158431148697917	0.4601290579826491

10	1.0	3.6988925808851116e-16	2.220446049250313e-16
10	10.0	2.830524433501838e-16	3.8459253727671276e-16
10	1000.0	1.0067709052191802e-14	1.0029192180646287e-14
10	1.0e7	1.7430422591515025e-10	1.5980626821159785e-10
10	1.0e12	1.6640510046704745e-5	1.4472616550517232e-5
10	1.0e16	0.03782557004895977	0.057250221751753934
20	1.0	5.506527130999414e-16	3.623139315696182e-16
20	10.0	4.742874840267547e-16	4.877405731848759e-16
20	1000.0	1.1904806100949086e-14	1.9646117158410984e-14
20	1.0e7	4.6279131645103426e-10	5.171742278393873e-10
20	1.0e12	8.11402139763181e-6	9.510480960041806e-6
20	1.0e16	0.6168393006207931	0.5708473519416114

Widać, że dla losowej macierzy błąd przeprowadzonych obliczeń również rośnie wraz ze stopnia macierzy (i wzrostem uwarunkowania macierzy) dla obu metod. Jednak sytuacja wygląda o wiele lepiej i błąd rośnie o znacznie wolniej, niż dla macierzy Hilberta. Co świadczy, że zadanie rozwiązywania układu równań liniowych dla macierzy Hilberta jest zadaniem źle uwarunkowanym, a macierze o wysokim wskaźniku uwarunkowania generują duże błędy w obliczeniach.

4 „Złośliwy” wielomian Wilkinsona

W tym zadaniu należało sprawdzić obliczone pierwiastki z k, 1 < k < 20, obliczając $|P(z_k)|$, $|p(z_k)|$ i $|z_k - k|$, gdzie P jest postacią naturalną wielomianu Wilkinsona p:

$$p(x) = (x-20)(x-19)(x-18)(x-17)(x-16)(x-15)(x-14)(x-13)(x-12)(x-11)$$
$$(x-10)(x-9)(x-8)(x-7)(x-6)(x-5)(x-4)(x-3)(x-2)(x-1)$$

Kod źródłowy programu znajduje się w pliku zad4.jl – jako funkcja o nazwie *wilkinson ()*.

k	z_k	$\lvert P(z_k)\rvert$	$\lvert p(z_k)\rvert$	$\lvert z_k - k\rvert$
1	0.9999999999996989	35696.50964788257	36626.425482422805	3.0109248427834245e-13
2	2.0000000000283182	176252.60026668405	181303.93367257662	2.8318236644508943e-11
3	2.9999999995920965	279157.6968824087	290172.2858891686	4.0790348876384996e-10
4	3.9999999837375317	3.0271092988991085e6	2.0415372902750901e6	1.626246826091915e-8
5	5.000000665769791	2.2917473756567076e7	2.0894625006962188e7	6.657697912970661e-7
6	5.999989245824773	1.2902417284205095e8	1.1250484577562995e8	1.0754175226779239e-5
7	7.000102002793008	4.805112754602064e8	4.572908642730946e8	0.00010200279300764947
8	7.999355829607762	1.6379520218961136e9	1.5556459377357383e9	0.0006441703922384079
9	9.002915294362053	4.877071372550003e9	4.687816175648389e9	0.002915294362052734
10	9.990413042481725	1.3638638195458128e10	1.2634601896949205e10	0.009586957518274986
11	11.025022932909318	3.585631295130865e10	3.300128474498415e10	0.025022932909317674
12	11.953283253846857	7.533332360358197e10	7.388525665404988e10	0.04671674615314281
13	13.07431403244734	1.9605988124330817e11	1.8476215093144193e11	0.07431403244734014
14	13.914755591802127	3.5751347823104315e11	3.5514277528420844e11	0.08524440819787316
15	15.075493799699476	8.21627123645597e11	8.423201558964254e11	0.07549379969947623
16	15.946286716607972	1.5514978880494067e12	1.570728736625802e12	0.05371328339202819
17	17.025427146237412	3.694735918486229e12	3.3169782238892363e12	0.025427146237412046

18	17.99092135271648	7.650109016515867e12	6.34485314179128e12	0.009078647283519814
19	19.00190981829944	1.1435273749721195e13	1.228571736671966e13	0.0019098182994383706
20	19.999809291236637	2.7924106393680727e13	2.318309535271638e13	0.00019070876336257925

k – pierwiastek wielomianu p, z_k – pierwiastek wielomianu P

Powyższa tabela przedstawia wyniki dla wielomianu Wilkinsona. Zauważmy, że miejsca zerowe wielomianu w postaci naturalnej różnią się od tych w postaci iloczynowej. Funkcja ta jest w takim razie bardzo czuła na odchylenia, ponieważ z matematycznego punktu widzenia $|P(z_k)|$ i $|p(z_k)|$ powinny być równe 0, a np. dla k = 20 wynosi 2.7924106393680727e13, co jest ogromnym odchyleniem.

Błędy spowodowane niedokładnym liczeniem miejsc zerowych funkcji skutkują różnicami w wynikach. Z początku niewielkie zaczynają się kumulować. Na tej podstawie można stwierdzić, że zadanie obliczania pierwiastków wielomianu Wilkinsona jest zadaniem źle uwarunkowanym.

4.1 Eksperyment Wilkinsona

W tej części zadania mieliśmy powtórzyć eksperyment Wilkinsona tj. zmienić współczynnik -210 na $-210 - 2^{-23}$.

Kod źródłowy programu znajduje się w pliku zad4.jl – jako funkcja o nazwie *wilkinson_experiment()*. Poniżej zestawione wyniki:

k	z_k	$\|P(z_k)\|$	$\|p(z_k)\|$	$\|z_k - k\|$
1	0.9999999999998357 + 0.0im	20259.872313418207	19987.872313406835	1.6431300764452317e-13
2	2.0000000000550373 + 0.0im	346541.4137593836	352369.4138087958	5.503730804434781e-11
3	2.99999999660342 + 0.0im	2.2580597001197007e6	2.4162415582518433e6	3.3965799062229962e-9
4	4.000000089724362 + 0.0im	1.0542631790395478e7	1.1263702300292023e7	8.972436216225788e-8
5	4.99999857388791 + 0.0im	3.757830916585153e7	4.475744423806908e7	1.4261120897529622e-6
6	6.000020476673031 + 0.0im	1.3140943325569446e8	2.1421031658039317e8	2.0476673030955794e-5
7	6.99960207042242 + 0.0im	3.939355874647618e8	1.7846173427860644e9	0.00039792957757978087
8	8.007772029099446 + 0.0im	1.184986961371896e9	1.8686972170009857e10	0.007772029099445632
9	8.915816367932559 + 0.0im	2.2255221233077707e9	1.3746309775142993e11	0.0841836320674414
10	10.095455630535774 - 0.6449328236240688im	1.0677921232930157e10	1.490069535200058e12	0.6519586830380407
11	10.095455630535774 + 0.6449328236240688im	1.0677921232930157e10	1.490069535200058e12	1.1109180272716561
12	11.793890586174369 - 1.6524771364075785im	3.1401962344429485e10	3.2962792355717145e13	1.665281290598479
13	11.793890586174369 + 1.6524771364075785im	3.1401962344429485e10	3.2962792355717145e13	2.0458202766784277
14	13.992406684487216 - 2.5188244257108443im	2.157665405951858e11	9.546022365750216e14	2.518835871190904
15	13.992406684487216 + 2.5188244257108443im	2.157665405951858e11	9.546022365750216e14	2.7128805312847097
16	16.73074487979267 - 2.812624896721978im	4.850110893921027e11	2.742106076928478e16	2.9060018735375106
17	16.73074487979267 + 2.812624896721978im	4.850110893921027e11	2.742106076928478e16	2.825483521349608

18	19.5024423688181 - 1.940331978642903im	4.557199223869993e12	4.2524858765203725e17	2.4540214463129764
19	19.5024423688181 + 1.940331978642903im	4.557199223869993e12	4.2524858765203725e17	2.0043294443099486
20	20.84691021519479 + 0.0im	8.756386551865696e12	1.37437435599976e18	0.8469102151947894

Powyższa tabela przedstawia wyniki dla wielomianu Wilkinsona o zaburzonym niewiele jednym współczynniku. Nasza minimalna zmiana jednego współczynnika, okazuje się że spowodowała duże zmiany w miejscach zerowych funkcji, np. poprzednie miejsce zerowe o wartości 20 zwiększyło się prawie o jeden, a miejsca zerowe przybrały postać zespoloną.

Dowodzi to, że obliczenia na tym wielomianie są niewykonalne, ponieważ niewielka zmiana danych wejściowych całkowicie zmienia wynik. Czyli problem wyznaczania pierwiastków wielomianu Wilkinsona jest źle uwarunkowany.

5 Rozważanie równania rekurencyjnego (model logistyczny, model wzrostu populacji)

W zadaniu piątym należało przeprowadzić dwa eksperymenty na równaniu rekurencyjnym

$$p_{n+1} := p_n + rp_n(1 - p_n), dla\ n = 0,1, \dots :$$

(1) Dla danych p_0 = 0.01 i r = 3 wykonać (w arytmetyce Float32) 40 iteracji wyrażenia, a następnie wykonać ponownie 40 iteracji wyrażenia, z niewielką modyfikacją tj. wykonać 10 iteracji, zatrzymać, zastosować obcięcie wyniku odrzucając cyfry po trzecim miejscu po przecinku i kontynuować dalej obliczenia.

Kod źródłowy programu znajduje się w pliku zad5.jl. Wyniki:

Iteracja	p_n normalne	p_n ucięte	Błąd względny
0	0.01	0.01	0.0
1	0.0397	0.0397	0.0
2	0.15407173	0.15407173	0.0
3	0.5450726	0.5450726	0.0
4	1.2889781	1.2889781	0.0
5	0.1715188	0.1715188	0.0
6	0.5978191	0.5978191	0.0
7	1.3191134	1.3191134	0.0
8	0.056273222	0.056273222	0.0
9	0.21559286	0.21559286	0.0
10	0.7229306	0.722	0.0009306073
11	1.3238364	1.3241479	0.00031149387
12	0.037716985	0.036488414	0.0012285709
13	0.14660022	0.14195944	0.004640773
14	0.521926	0.50738037	0.0145456195
15	1.2704837	1.2572169	0.013266802
16	0.2395482	0.28708452	0.047536314
17	0.7860428	0.9010855	0.11504269

18	1.2905813	1.1684768	0.122104526
19	0.16552472	0.577893	0.4123683
20	0.5799036	1.3096911	0.72978747
21	1.3107498	0.09289217	1.2178576
22	0.088804245	0.34568182	0.25687757
23	0.3315584	1.0242395	0.69268113
24	0.9964407	0.94975823	0.046682477
25	1.0070806	1.0929108	0.08583021
26	0.9856885	0.7882812	0.1974073
27	1.0280086	1.2889631	0.2609545
28	0.9416294	0.17157483	0.7700546
29	1.1065198	0.59798557	0.50853425
30	0.7529209	1.3191822	0.56626123
31	1.3110139	0.05600393	1.25501
32	0.0877831	0.21460639	0.12682329
33	0.3280148	0.7202578	0.39224303
34	0.9892781	1.3247173	0.3354392
35	1.021099	0.034241438	0.98685753
36	0.95646656	0.13344833	0.8230182
37	1.0813814	0.48036796	0.6010135
38	0.81736827	1.2292118	0.41184354
39	1.2652004	0.3839622	0.88123816
40	0.25860548	1.093568	0.8349625

Do momentu obcięcia wartości elementów dla obu ciągów były identyczne, ale po naszym zabiegu zaczynają się niewielkie odchylenia, które z każdą iteracją powiększają się, aż do momentu gdzie nie są nawet bliskie sobie.

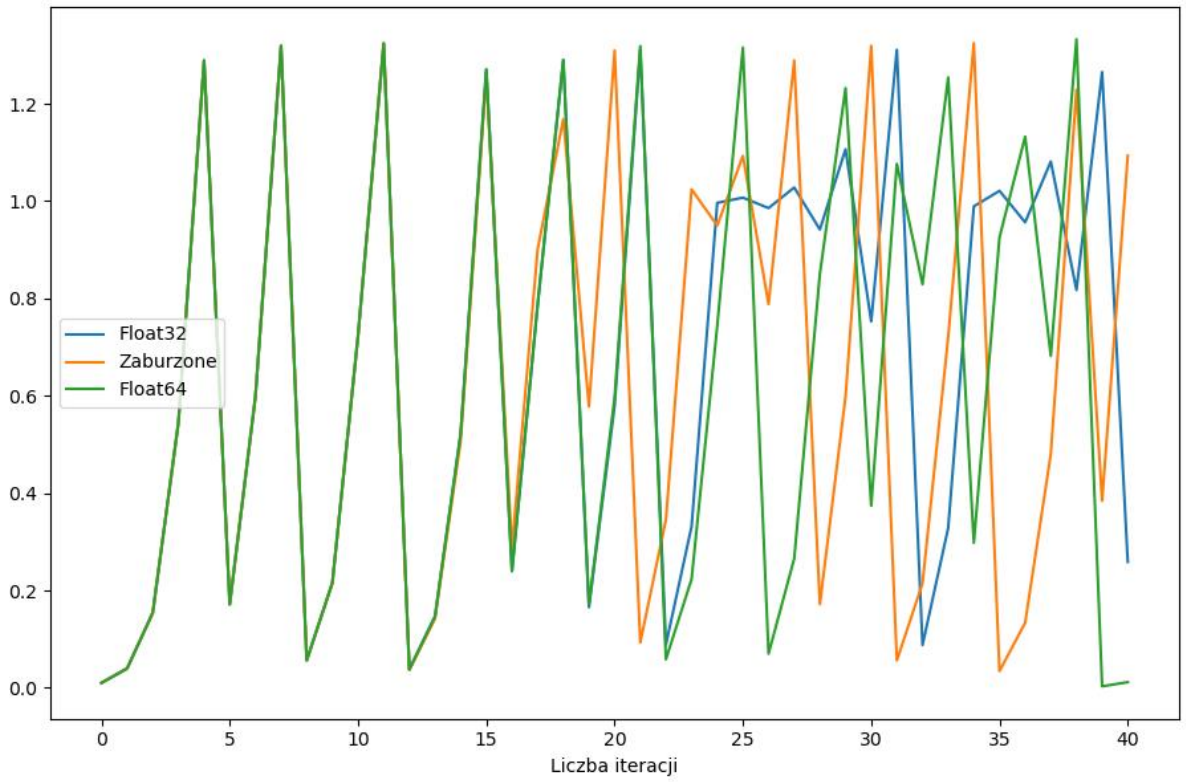
W powyższej tabeli widać, że pozornie mały błąd spowodowany obcięciem w dziesiątej iteracji, spowodował 4-krotnie większy wynik (0.25860548 bez obcięcia, 1.093568 z obcięciem), a błąd względny z każdą iteracją stawał się coraz większy. Na tej podstawie stwierdzam, że zadanie jest źle uwarunkowane, ponieważ mały błąd powoduje napiętrzanie się go rekurencyjnie i bardzo szybko się zwiększa. Zaobserwowane zjawisko wrażliwości rozwiązań na małe zaburzenie parametrów zostało nazwane przez Edwarda Lorenza jako chaos deterministyczny.

(2) Dla danych $p_0 = 0.01$ i $r = 3$ wykonać 40 iteracji wyrażenia, w arytmetyce Float32 i Float64.
Wyniki:

Iteracja	**p_n w Float32**	**p_n w Float64**
0	0.01	0.01
1	0.0397	0.0397
2	0.15407173	0.15407173000000002
3	0.5450726	0.5450726260444213
4	1.2889781	1.2889780011888006

5	0.1715188	0.17151914210917552
6	0.5978191	0.5978201201070994
7	1.3191134	1.3191137924137974
8	0.056273222	0.056271577646256565
9	0.21559286	0.21558683923263022
10	0.7229306	0.722914301179573
11	1.3238364	1.3238419441684408
12	0.037716985	0.03769529725473175
13	0.14660022	0.14651838271355924
14	0.521926	0.521670621435246
15	1.2704837	1.2702617739350768
16	0.2395482	0.24035217277824272
17	0.7860428	0.7881011902353041
18	1.2905813	1.2890943027903075
19	0.16552472	0.17108484670194324
20	0.5799036	0.5965293124946907
21	1.3107498	1.3185755879825978
22	0.088804245	0.058377608259430724
23	0.3315584	0.22328659759944824
24	0.9964407	0.7435756763951792
25	1.0070806	1.315588346001072
26	0.9856885	0.07003529560277899
27	1.0280086	0.26542635452061003
28	0.9416294	0.8503519690601384
29	1.1065198	1.2321124623871897
30	0.7529209	0.37414648963928676
31	1.3110139	1.0766291714289444
32	0.0877831	0.8291255674004515
33	0.3280148	1.2541546500504441
34	0.9892781	0.29790694147232066
35	1.021099	0.9253821285571046
36	0.95646656	1.1325322626697856
37	1.0813814	0.6822410727153098
38	0.81736827	1.3326056469620293
39	1.2652004	0.0029091569028512065
40	0.25860548	0.011611238029748606

Przy eksperymencie drugim porównywaliśmy wartości elementów ciągu w arytmetykach podwójnej i pojedynczej precyzji. Do około dwudziestej pierwszej iteracji, wyniki dla obu są podobne. Dalej dokładność arytmetyki potrzebna na zapisanie elementów ciągu wyrażonego rekurencyjnie staje się niewystarczające i wyniki stają się rozbieżne. Można zaobserwować, że w tej dokładniejszej wynik wyniósł 0.011611238029748606, czyli wartość prawie 25-krotnie mniejsza, niż w arytmetyce Float32. Natomiast gdyby porównać ten wynik do czterdziestego elementu ciągu dla Float32 z zaburzeniem, mielibyśmy aż stukrotną różnicę.



Wykres zależności wartości elementu ciągu od jego numeru n

Przenoszenie się błędu, dla modelu rozwoju populacji Verhulste'a, jest spowodowane precyzją arytmetyki IEEE-754. Błąd popełniony w początkowym stadium kumuluje się, aż do momentu gdy wynik staje się kompletnie nierzetelny. Zastosowanie większej precyzji obliczeń mogłoby pomóc, jednak tylko chwilowo, bo propagacja błędu spowoduje występowanie błędu przy dalszych iteracjach.

6 Badanie równania rekurencyjnego określonego przez funkcję kwadratową

W tym zadaniu mieliśmy rozważyć równanie rekurencyjne:

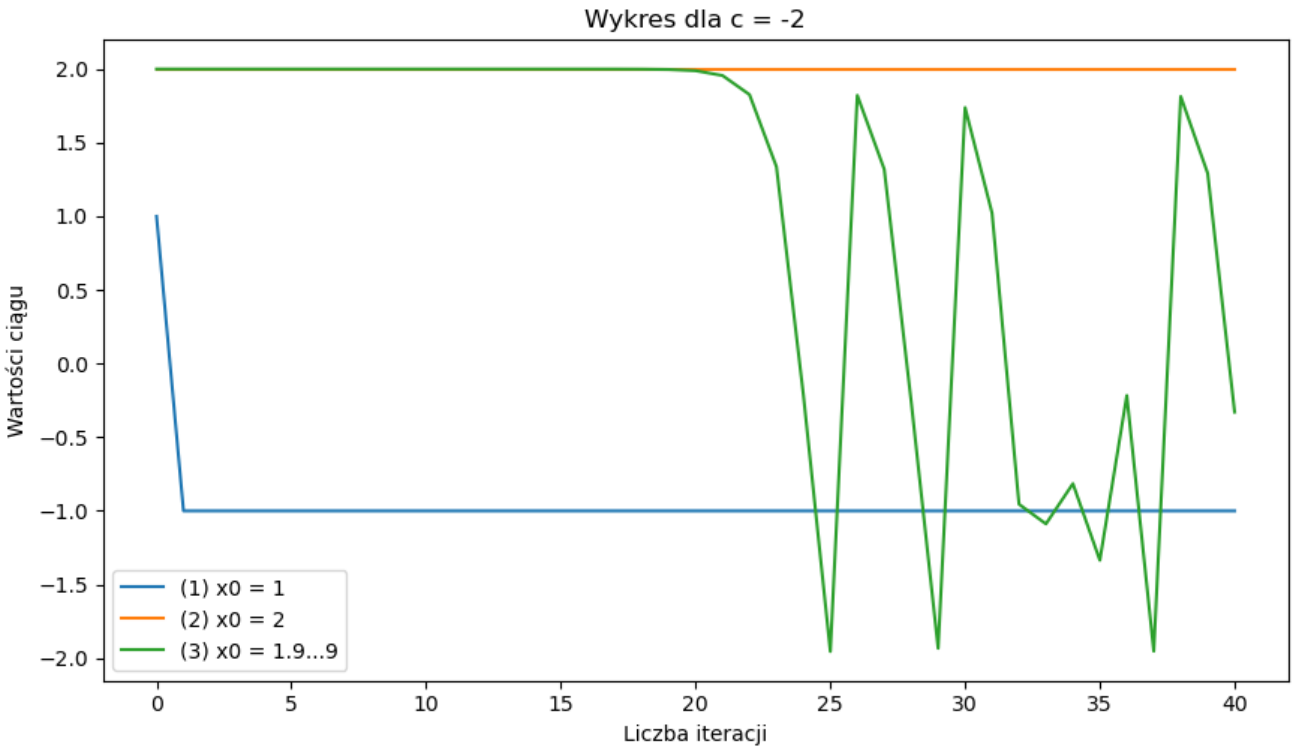
$$x_{n+1} := x_n^2 + c, \text{dla } n = 0,1, \ldots,$$

gdzie c jest pewną daną stałą i przeprowadzić następujące eksperymenty. Dla danych:

(1) c = −2 i x_0 = 1
(2) c = −2 i x_0 = 2
(3) c = −2 i x_0 = 1.99999999999999
(4) c = −1 i x_0 = 1
(5) c = −1 i x_0 = −1
(6) c = −1 i x_0 = 0.75
(7) c = −1 i x_0 = 0.25

Kod źródłowy programu znajduje się w pliku zad6.jl.

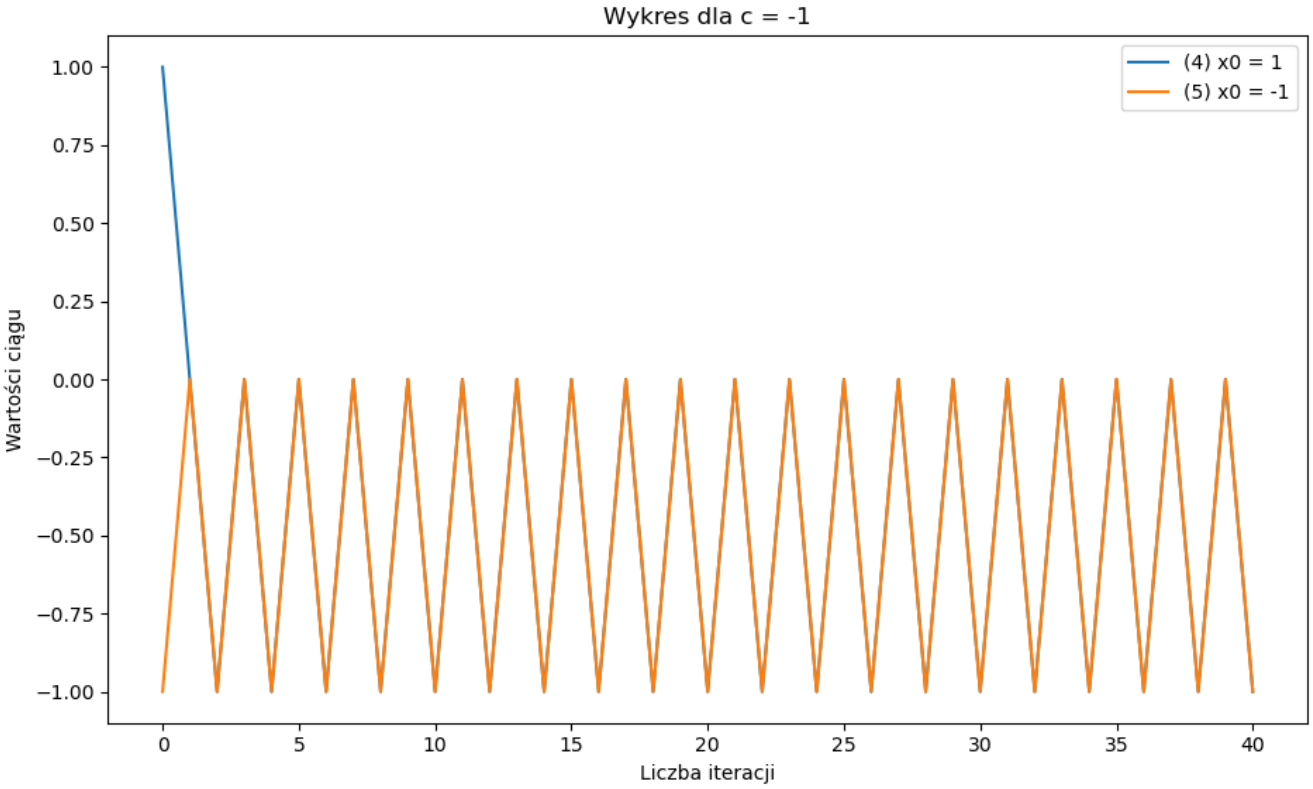
Jako, że eksperymentów jest wiele to omówię je grupami, zaczynając od pierwszych trzech (gdzie c = -2).



Jak widać wartości dla x_o = 1 i 2, wartości ciągu generowane są poprawnie, ponieważ ich wartości są stałe i wynoszą odpowiednio -1 i 2. Wartym zwrócenia uwagi jest wykres zielony przedstawiający eksperyment (3), ponieważ wartość x_0 jest niemal równa tej w przypadku (2), a od około 18 iteracji wartości przestają być uporządkowane i stają się coraz bardziej nieprzewidywalne.

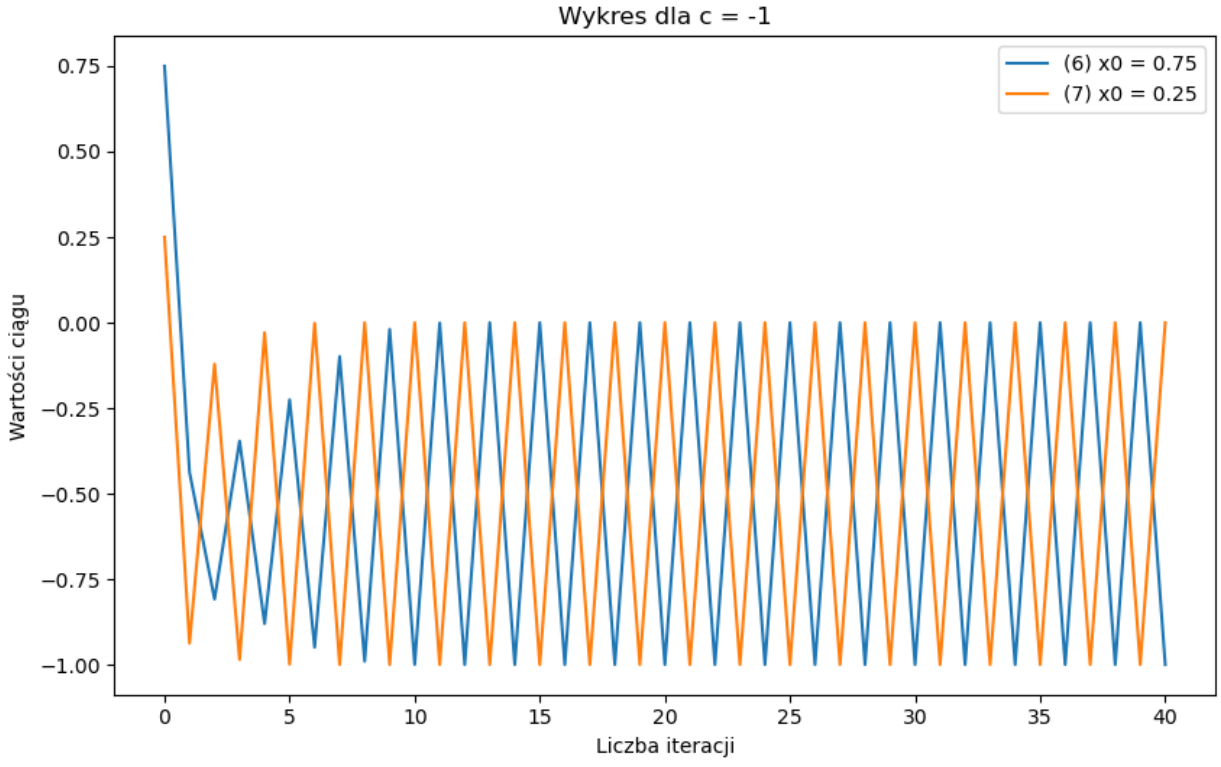
Świadczy to o wystąpieniu akumulacji błędu. Na brak precyzji wyniku wpływa również długość rozwinięcia dziesiętnego, które w wypadku podnoszenia do kwadratu powinna się podwajać. Uniemożliwia to otrzymanie dokładnych wyników dla więcej niż kilku iteracji, gdyż komputery reprezentują liczby tylko ze skończoną, liczbą miejsc po przecinku i dochodzi do zaokrągleń.

Teraz porównajmy ze sobą wartości dla (4) i (5):



Tutaj stwierdzam, że oba wykresy są poprawne i zachowują się stabilnie. Przybierają wartości 0 i -1 jak można by się tego spodziewać.

Na koniec zestawmy ze sobą wartości dla (6) i (7):



Tu dla większej liczby iteracji zauważamy, że wartości w obu przypadkach wahają się raz na 0, a raz -1 w stałych interwałach. Dzieje się tak, ponieważ 0.25 i 0.75 brane do kwadratu szybko maleją, więc w pewnym momencie przeprowadzane jest odejmowanie 1 od liczby bliskiej 1 i komputer zaokrągla je do 0. Od 0 odejmowane jest 1, w ten sposób uzyskujemy -1 i cykl się powtarza. Przypadkowi (6) zajmuje niewiele dłużej, żeby ustabilizować się między 0, a -1, ponieważ dla $x_0 = 0.25$ wartość ciągu szybciej dąży do zera.

Zaobserwowane niestabilności i odchylenia wyników są skutkiem precyzji arytmetyki Float64, która była używana do przeprowadzania obliczeń. Przy braniu liczb do kwadratu liczba potrzebnych miejsc rośnie w bardzo szybkim tempie, co znacząco fałszuje wyniki.