Obliczenia naukowe
Sprawozdanie
Lista 2

Mateusz Laskowski

11.11.2018

1. Zadanie 1

1.1. Opis problemu

Powtórzyć zadanie 5 z listy nr 1, lecz zmianą w tym zadaniu jest usunięcie ostatniej cyfry 9 z x_4 oraz ostatniej cyfry 7 z x_5. Sprawdzić jaki wpływ na wyniki mają wykonane niewielkie zmiany w danych.

Przypomnienie, jaki cel miało zadanie 5 z listy nr 1.

Napisać program w języku Julia realizujący następujący eksperyment 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]

(a) „w przód" $\sum_{i=1}^{n} x_i y_i$
(b) „w tył" $\sum_{i=n}^{1} x_i y_i$
(c) Suma wszystkich częściowych iloczynów skalarnych zaczynając od **największego** do **najmniejszego** wyniku częściowego
(d) Suma wszystkich częściowych iloczynów skalarnych zaczynając od **najmniejszego** do **największego** wyniku częściowego

1.2. Rozwiązanie problemu

Uruchomiłem program z pierwszej listy po dokonaniu zmian w wektorach, usunąłem ostatnią cyfrę 9 z x_4 oraz usunąłem ostatnią cyfrę 7 z x_5. Wyniki zestawiłem z wynikami programu bez modyfikacji.

1.3. Wyniki

Ryc. 1.1 – tabela przedstawia zestawienie wyników w arytmetyce **Float32**

Algorytm	Suma bez zmian wektorów	Suma z modyfikacją wektorów
(a)	-0.4999443	-0.4999443
(b)	-0.4543457	-0.4543457
(c)	-0.5	-0.5
(d)	-0.5	-0.5

Ryc. 1.2 – tabela przedstawiająca zestawienie wyników w arytmetyce **Float64**

Algorytm	Suma bez zmian wektorów	Suma z modyfikacją wektorów
(a)	1.0251881368296672e-10	-0.004296342739891585
(b)	-1.5643308870494366e-10	-0.004296342998713953
(c)	0.0	-0.004296342842280865
(d)	0.0	-0.004296342842280865

Ryc. 1.3 – tabela przedstawiająca dokładne wyniki sumowania wektorów

Float64 δ_x	*Float32* δ_x
-0.0042963428423	0.0000000

1.4. Wnioski

W przypadku arytmetyki Float32 usunięcie ostatnich cyfr nie miało wpływu na wynik ostateczny, ponieważ arytmetyka nie jest wystarczająco dokładna. Liczby pojedynczej precyzji pozwalają na zapis do 7-miu cyfr znaczących w reprezentacji dziesiętnej. Modyfikacja wektora w zadaniu dotyczyła cyfry na pozycji 10-tej, więc nie doprowadziło to widocznej zmiany na cyfrach znaczących.

Natomiast w przypadku arytmetyki Float64 lekka miana wektorów przyczyniła się do zmiany wyjściowego wyniku. Wynik ten wciąż odbiega od poprawnego wyniku, lecz jest dużo bardziej dokładny niż bez modyfikacji wektorów. Zmniejszenie dokładności składowych wektora pozwoliło na bliższe sobie wyniki z każdego osobnego algorytmu sumowania. Jak widać poprzez 10-tą cyfrę po przecinku wyniki zaczęły się mocno rozbiegać.

2. Zadanie 2

2.1. Opis problemu

Narysować wykres funkcji $f(x) = e^x ln(1 + e^{-x})$ w co najmniej dwóch dowolnych programach do wizualizacji. Następnie policzyć granicę funkcji $lim_{x\to\infty} f(x)$. Porównać wykres funkcji z policzoną granicą oraz wyjaśnić zjawisko jakie zachodzi.

2.2. Rozwiązanie problemu

Granica funkcji $f(x)$:

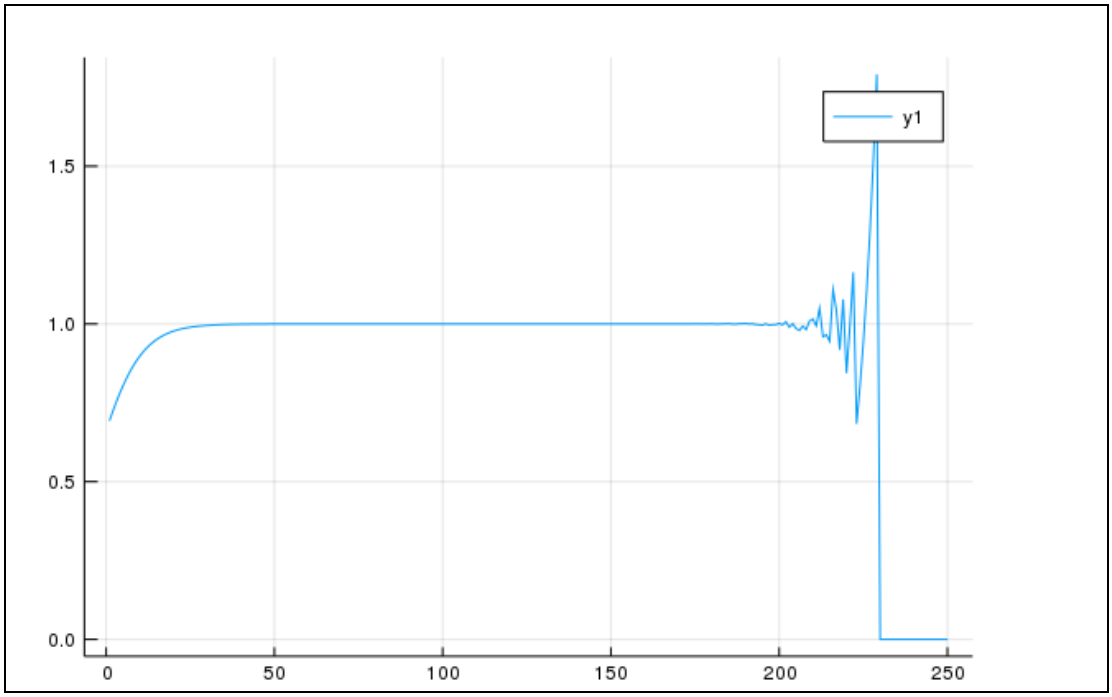
$$lim_{x\to\infty} e^x ln(1 + e^{-x}) = 1$$

Wykonałem obliczenia funkcji $f(x)$ dla x od 0 do 40 oraz wykonałem wykresy z otrzymanych danych.

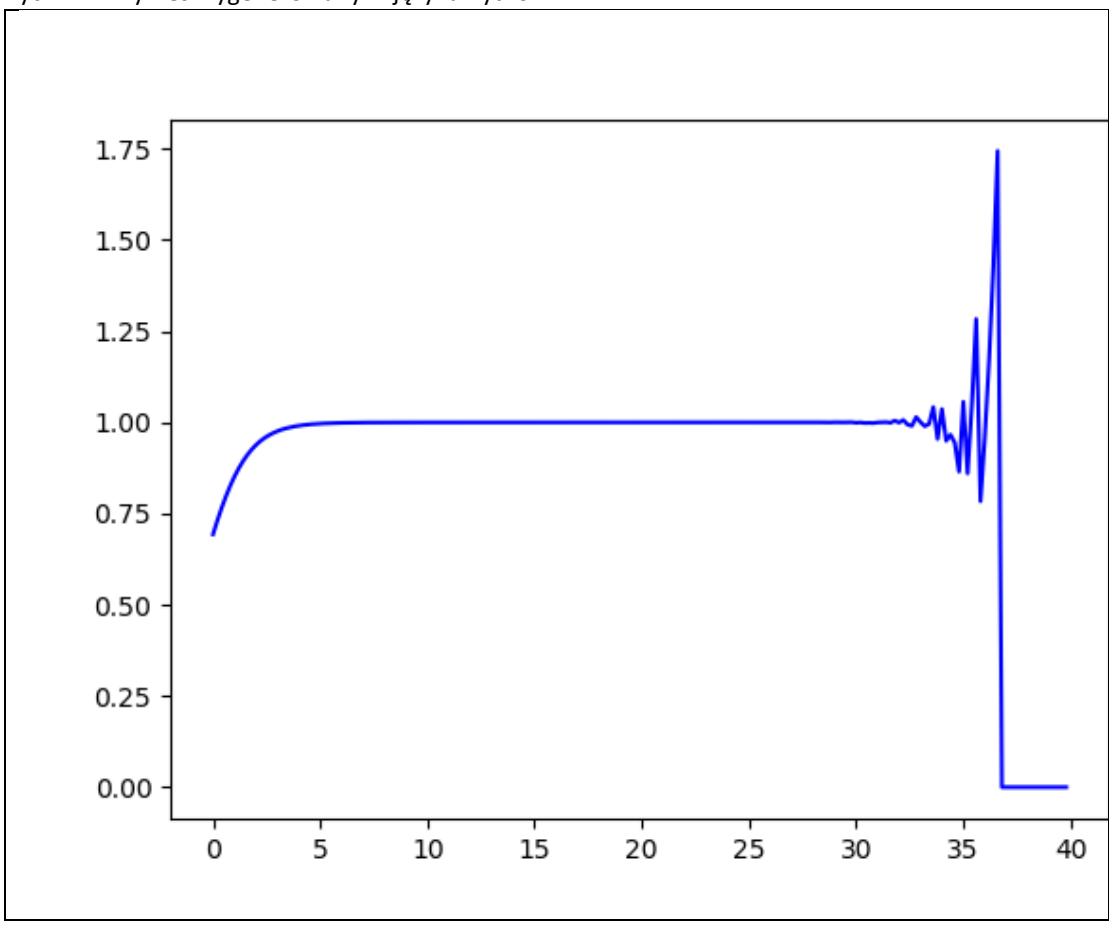
2.3. Wyniki

Wykresy wygenerowane za pomocą dwóch programów do wizualizacji. Pierwszy wykres Ryc. 2.1 powstał w języku Julia za pomocą biblioteki Plots, zaś drugi wykres Ryc. 2.2 powstał w języku Python za pomocą biblioteki matplotlib.

Ryc. 2.1 – wykres wygenerowany w języku Julia



Ryc. 2.2 – wykres wygenerowany w języku Python



2.4. Wnioski

Na podanych wykresach widać, że przy $x = 30$, na wykresie zaczynają się pojawiać duże odchyły wartości, gdzie dla *x* większego od 36 funkcja ma wartość 0. Głównym czynnikiem takich wariacji wartości jest operacja mnożenia bardzo dużej liczby e^x. Aż w pewnym momencie funkcja wynosi zero.

3. Zadanie 3

3.1. Opis problemu

Rozwiązać układ równań liniowych $\boldsymbol{Ax} = \boldsymbol{b}$ za pomocą dwóch algorytmów: eliminacji Gaussa $(\boldsymbol{x} = \boldsymbol{A}\backslash\boldsymbol{b})$ oraz $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ $(\mathrm{x} = \mathrm{inv(A)} * \mathrm{b})$. Eksperymenty przeprowadzić dla macierzy Hilberta H_n z rosnącym stopniem $n > 1$ oraz dla macierzy losowej $\boldsymbol{R_n}$, $n = 5, 10, 20$ z rosnącym wskaźnikiem uwarunkowania $c = 1, 10, 10^3, 10^7, 10^{12}, 10^{16}$. Policzyć błędy względne i porównać z rozwiązaniem dokładnym.

3.2. Rozwiązanie problemu

W celu utworzenia podanych macierzy wykorzystałem funkcje hilb(n) – generująca macierz Hilberta stopnia *n* – oraz matcond(n, c) – generująca macierz losową stopnia *n* z zadanym wskaźnikiem uwarunkowania *c*. Błędy względne obliczeń wyliczyłem przy pomocy normy wektora:

$$\delta = \frac{\|\tilde{x} - x\|}{\|x\|}$$

3.3. Wyniki

Ryc. 3.1 – tabela przedstawia błędy względne dla **macierzy Hilberta**

n	***cond***	***gauss***	***inverted***
1	1.0	0.0	0.0
2	19.28147006790397	5.661048867003676e-16	1.1240151438116956e-15
3	524.0567775860644	8.022593772267726e-15	9.825526038180824e-15
4	15513.73873892924	4.4515459601812086e-13	2.950477637286781e-13
5	476607.25024259434	1.6828426299227195e-12	8.500055777753297e-12
6	1.4951058642254665e7	2.618913302311624e-10	3.3474135070361745e-10
7	4.75367356583129e8	1.2606867224171548e-8	5.163959183577243e-9
8	1.5257575538060041e10	1.026543065687064e-7	2.698715074276819e-7
9	4.931537564468762e11	4.83235712050215e-6	9.175846868614517e-6
10	1.6024416992541715e13	0.0006329153722983848	0.00045521422517408853
11	5.222677939280335e14	0.011543958596122112	0.00804446677343116
12	1.7514731907091464e16	0.2975640310734787	0.34392937091205217
13	3.344143497338461e18	2.375017867706776	5.585796893150773
14	6.200786263161444e17	5.281004646755168	4.800641929017436
15	3.674392953467974e17	1.177294734836712	4.8273577212576475
16	7.865467778431645e17	20.564655823804095	31.736467496266126
17	1.263684342666052e18	17.742214635179074	15.910335962604142
18	2.2446309929189128e18	4.2764564411159425	6.281223433472033
19	6.471953976541591e18	22.119937292648906	22.92561401563632
20	1.3553657908688225e18	14.930069669294001	21.53949860251383

Ryc. 3.2 – tabela przedstawia błędy względne dla **macierzy losowej** o współczynniku *cond*

n	*cond*	*gauss*	*inverted*
5	1.0	9.930136612989092e-17	5.528866075183428e-16
5	3.0	1.2872172993348367e-14	1.6187493345516222e-14
5	7.0	9.891015218792042e-11	3.4375612071191857e-10
5	12.0	7.295011118019287e-6	4.8888371474598434e-5
5	16.0	0.20525387226858713	0.25856403398852085
10	1.0	7.982804644978158e-16	8.158439733063109e-16
10	3.0	7.543089232057327e-14	6.407947240619581e-14
10	7.0	8.969538169778931e-10	4.325562062623882e-10
10	12.0	8.37407170623145e-5	6.06374931824181e-5
10	16.0	1.1234520622231834	0.8337349142359708
20	1.0	6.454588798442909e-16	8.203639191033913e-16
20	3.0	5.754394216095572e-15	2.8106496957992408e-14
20	7.0	5.348362088472877e-10	2.559855198677472e-10
20	12.0	1.3621359328727006e-5	1.6463748074688822e-5
20	16.0	0.043939229124012054	0.563291610564292

3.4. Wnioski

W przypadku macierzy losowych, błąd bezwzględny, niezależnie od stopnia macierzy oscyluje w granicach 10^{-16}. Jednakże w przypadku macierzy Hilberta można zaobserwować wiele więcej.

Macierz Hilberta jest tworzona z elementami danymi wzorem $h_{ij} = \frac{1}{i+j-1}$. Macierz Hilberta jest podręcznikowym przykładem macierzy źle uwarunkowanej, ponieważ duża ilość komórek z macierzy nie da się zapisać w postaci skończonej w reprezentacji binarnej, przykładowo $\frac{1}{3}$. Wykonując obliczenia z jej użyciem jesteśmy narażeni na duży wpływ błędu wynikającego z reprezentacji numerycznej, dlatego rozwiązanie w numeryczny sposób nawet dla niewielkich układów równań z macierzą jest praktycznie nie możliwe.

Po obserwacji, wywnioskowałem, że wraz ze wzrostem stopnia macierzy, błąd obliczeń wykonywanych metodą eliminacji Gaussa, jak i przy użyciu macierzy odwrotnej, rośnie wraz ze wzrostem stopnia macierzy. Jednakże, wynik w przypadku metody Gaussa jest opatrzony mniejszym błędem, ponieważ algorytm ten pozwala na dojście do wyniku w mniejszej ilości kroków (mniejszej ilości operacji) w algorytmie, a wiadomo, że mniej operacji to mniej zaokrągleń w arytmetyce numerycznej. W przypadku macierzy Hilberta i macierzy losowej, metoda Gaussa była lepsza, z powodu mniejszej ilości kroków w algorytmie, lecz błąd wciąż rósł wraz ze wzrostem stopnia *n* macierzy.

4. Zadanie 4

4.1. Opis problemu

(„złośliwy wielomian”, Wilkinson) Zainstalować pakiet Polynomials.

(a) Użyć funkcji roots (z pakietu Polynomials) do obliczenia 20 zer wielomianu *P* w postaci naturalnej, podanego w treści zadania. Sprawdzić obliczone pierwiastki $z_k, 1 \leq k \leq 20$ obliczając $|P(z_k)|, |p(z_k)|$ i $|z_k - k|$. Wyjaśnić rozbieżności.

(b) Powtórzyć eksperyment Wilkinsona,
tj. zmienić współczynnik -210 na $-210 - 2^{23}$

4.2. Rozwiązanie problemu

Funkcja Poly(x) generuje wielomian zależny od współczynnika podanego jako parametr *x.* Metoda poly(x) tworzy wielomian na podstawie pierwiastków wielomianu. Funkcja polyval(p, x) wylicza wartości wielomianu *p* dla wczytywanego parametru *x*. Wykorzystałem w swoim programie wymienione wyżej metody zawarte w bibliotece Polynomials. Wyniki z podpunktu **(a)** zostały zamieszczone na Ryc. 4.1 oraz Ryc. 4.2, zaś wyniki z podpunktu (b) na Ryc. 4.3 i Ryc. 4.4.

4.3. Wyniki

Ryc. 4.1 – tabela z wynikami po eksperymencie Wilkinsona, podpunkt (a)

k	$\|P(z_k)\|$	$\|p(z_k)\|$	$\|z_k - k\|$
1	36352.0	38400.0	3.0109248427834245e-13
2	181760.0	198144.0	2.8318236644508943e-11
3	209408.0	301568.0	4.0790348876384996e-10
4	3.106816e6	2.844672e6	1.626246826091915e-8
5	2.4114688e7	2.3346688e7	6.657697912970661e-7
6	1.20152064e8	1.1882496e8	1.0754175226779239e-5
7	4.80398336e8	4.78290944e8	0.00010200279300764947
8	1.682691072e9	1.67849728e9	0.0006441703922384079
9	4.465326592e9	4.457859584e9	0.002915294362052734
10	1.2707126784e10	1.2696907264e10	0.009586957518274986
11	3.5759895552e10	3.5743469056e10	0.025022932909317674
12	7.216771584e10	7.2146650624e10	0.04671674615314281
13	2.15723629056e11	2.15696330752e11	0.07431403244734014
14	3.65383250944e11	3.653447936e11	0.08524440819787316
15	6.13987753472e11	6.13938415616e11	0.07549379969947623
16	1.555027751936e12	1.554961097216e12	0.05371328339202819
17	3.777623778304e12	3.777532946944e12	0.025427146237412046
18	7.199554861056e12	7.1994474752e12	0.009078647283519814
19	1.0278376162816e13	1.0278235656704e13	0.0019098182994383706
20	2.7462952745472e13	2.7462788907008e13	0.0001907087633625792 5

Ryc. 4.2 – tabela z pierwiastkami root-ów, podpunkt (a)

n	***pierwiastek***
1	0.9999999999996989
2	2.0000000000283182
3	2.9999999995920965
4	3.9999999837375317
5	5.000000665769791
6	5.999989245824773
7	7.000102002793008
8	7.999355829607762
9	9.002915294362053
10	9.990413042481725
11	11.025022932909318
12	11.953283253846857
13	13.07431403244734
14	13.914755591802127
15	15.075493799699476
16	15.946286716607972
17	17.025427146237412
18	17.99092135271648
19	19.00190981829944
20	19.999809291236637

Ryc. 4.3 – tabela z wynikami po eksperymencie Wilkinsona z modyfikacją, podpunkt (b)

k	$\lvert P(z_k)\rvert$	$\lvert p(z_k)\rvert$	$\lvert z_k - k\rvert$
1	19456.0	19456.0	1.6209256159527285e-13
2	54272.0	70656.0	7.66275931596283e-12
3	782848.0	875008.0	1.091470469560818e-9
4	2.555392e6	2.817536e6	2.7770935773219207e-8
5	6.717952e6	7.485952e6	2.3598509635291975e-7
6	1.9252736e7	1.7925632e7	1.293933244994605e-7
7	1.2139264e8	1.18582784e8	1.4538322562707151e-5
8	3.96893184e8	3.92699392e8	9.171398842866552e-5
9	7.4291456e8	4.60736e8	9.836127977891351e-5
10	6.253824e8	2.56111616e9	0.0019448004357265347
11	9.511332864e9	1.828041984e10	0.014794041182474515
12	2.434616064e10	1.10285019136e11	0.0701234129085595
13	8.770431488e10	3.4704067328e11	0.16494035599639645
14	4.1416595000481635e11	3.066962265995617e12	0.5425416609403566
15	4.1416595000481635e11	3.066962265995617e12	0.6388530054672362
16	3.3752053852365005e12	3.677478938782452e13	0.6668188874271295
17	3.3752053852365005e12	3.677478938782452e13	0.6032627181802658
18	1.277755372544e13	2.20104343106048e14	0.21784959517997393
19	2.3017536954368e13	4.451884076032e14	0.08732812123692923
20	5.2019032684032e13	1.2844306462848e15	0.010184050456800264

Ryc. 4.4 – tabela pierwiastków root-ów z modyfikacją, podpunkt (b)

n	***pierwiastek***
1	1.000000000000162 + 0.0im
2	2.0000000000076628 + 0.0im
3	2.9999999989085295 + 0.0im
4	4.000000027770936 + 0.0im
5	4.999999764014904 + 0.0im
6	5.9999998706066755 + 0.0im
7	7.000014538322563 + 0.0im
8	7.999908286011571 + 0.0im
9	9.000098361279779 + 0.0im
10	10.001944800435727 + 0.0im
11	10.985205958817525 + 0.0im
12	12.07012341290856 + 0.0im
13	12.835059644003604 + 0.0im
14	14.4431091456307 - 0.3130586828605012im
15	14.4431091456307 + 0.3130586828605012im
16	16.540360760741656 - 0.3907143161956746im
17	16.540360760741656 + 0.3907143161956746im
18	18.217849595179974 + 0.0im
19	18.91267187876307 + 0.0im
20	20.0101840504568 + 0.0im

4.4. Wnioski

Wyniki z podpunktu (a) wskazują na pewne rozbieżności w wynikach. Wartości $P(z_k)$ oraz $p(z_k)$ różnią się od siebie. Wyliczone wartości pierwiastków za pomocą funkcji roots(P(x)) różnią się od rzeczywistych pierwiastków. Po obserwacji wnioskuję, że operacje wyznaczenia wielomianu na podstawie współczynników lub utworzeniu za pomocą pierwiastków, są obarczone zaburzeniem dokładności, wynikający z precyzji arytmetyki, na której działają operacje.

Obserwując wyniki z podpunktu (b), można zauważyć, że niewielka zmiana w jednym ze współczynników, była skutkiem wielkich zmian w wynikach, a co za tym idzie wielki błąd w dokładności. Ta niewielka zmiana w skończonej arytmetyce wywołała wielkie zaburzenie w arytmetyce numerycznej. Funkcja roots(P(x)) wywołana na wielomianie ze zmodyfikowanym współczynnikiem zwróciła pierwiastki zespolone.

5. Zadanie 5

5.1. Opis problemu

Równanie rekurencyjne (model logistyczny, model wzrostu populacji)

$$p_{n+1} := p_n + rp_n(1 - p_n)\ dla\ n\ 0, 1, 2, \ldots$$

Należy przeprowadzić następujące eksperymenty:

(a) Dla danych $p_0 = 0.01$ oraz $r = 3$ wykonać 40 iteracji wyrażenia (1), następnie 40 iteracji wyrażenia (1) z niewielką modyfikacją tj. wykonać 10 iteracji, zatrzymać, zastosować obcięcie wyniku odrzucając cyfry po trzecim miejscu po przecinku i kontynuować obliczenia do 40-stej iteracji. Porównać wyniki. Obliczenia wykonać w arytmetyce Float32.

(b) Dla danych $p_0 = 0.01$ i $r = 3$ wykonać 40 iteracji wyrażenia (1) w arytmetyce Float32 i Float64. Porównać wyniki.

5.2. Rozwiązanie problemu

Stworzyłem program w języku Julia, który wylicza wyżej podane rekurencje.

5.3. Wyniki

Ryc. 5.1 – tabela przedstawiająca wyniki z podpunktu (a) (wyniki 40 iteracji z oraz bez modyfikacji w arytmetyce Float32)

n	***40 iteracji bez modyfikacji***	***40 iteracji z modyfikacją 10-tego kroku***
1	0.039700001478195	0.039700001478195
2	0.154071733355522	0.154071733355522
3	0.545072615146637	0.545072615146637
4	1.288978099822998	1.288978099822998
5	0.171518802642822	0.171518802642822
6	0.597819089889526	0.597819089889526
7	1.319113373756409	1.319113373756409
8	0.056273221969604	0.056273221969604
9	0.215592861175537	0.215592861175537
10	0.722930610179901	0.722000002861023
11	1.323836445808411	1.324147939682007
12	0.037716984748840	0.036488413810730
13	0.146600216627121	0.141959443688393
14	0.521925985813141	0.507380366325378
15	1.270483732223511	1.257216930389404
16	0.239548206329346	0.287084519863129
17	0.786042809486389	0.901085495948792
18	1.290581345558167	1.168476819992065
19	0.165524721145630	0.577893018722534
20	0.579903602600098	1.309691071510315
21	1.310749769210815	0.092892169952393
22	0.088804244995117	0.345681816339493
23	0.331558406352997	1.024239540100098
24	0.996440708637238	0.949758231639862
25	1.007080554962158	1.092910766601563
26	0.985688507556915	0.788281202316284
27	1.028008580207825	1.288963079452515
28	0.941629409790039	0.171574831008911
29	1.106519818305969	0.597985565662384
30	0.752920925617218	1.319182157516479
31	1.311013936996460	0.056003928184509
32	0.087783098220825	0.214606389403343
33	0.328014791011810	0.720257818698883
34	0.989278078079224	1.324717283248901
35	1.021098971366882	0.034241437911987
36	0.956466555595398	0.133448332548141
37	1.081381440162659	0.480367958545685
38	0.817368268966675	1.229211807250977
39	1.265200376510620	0.383962213993073
40	0.258605480194092	1.093567967414856

Ryc. 5.2 – tabela przedstawia porównanie wyników 40-stu iteracji w arytmetyce Float32 oraz Float64

n	*Float32*	*Float64*
1	0.039700001478195	0.039700000000000
2	0.154071733355522	0.154071730000000
3	0.545072615146637	0.545072626044421
4	1.288978099822998	1.288978001188801
5	0.171518802642822	0.171519142109176
10	0.722930610179901	0.722914301179573
15	1.270483732223511	1.270261773935077
20	0.579903602600098	0.596529312494691
25	1.007080554962158	1.315588346001072
30	0.752920925617218	0.374146489639287
35	1.021098971366882	0.925382128557105
36	0.956466555595398	1.132532262669786
37	1.081381440162659	0.682241072715310
38	0.817368268966675	1.332605646962029
39	1.265200376510620	0.002909156902851
40	0.258605480194092	0.011611238029749

5.4. Wnioski

Pierwszym spostrzeżeniem to jak szybko rośnie błąd przy każdej iteracji, ponieważ z każdą kolejną iteracją, ilość cyfr po przecinku rośnie dwukrotnie, zbliżając się do granicy dokładności skończonej arytmetyki. Wartość w 3 iteracji jest już przybliżana, a nie dokładna, poprzez znaczny wzrost cyfr po przecinku w skończonej arytmetyce.

W podpunkcie (a) oczywiście nie było zmiany do 10-tej iteracji w otrzymanych wynikach, które można zobaczyć na Ryc. 5.1. Po obcięciu cyfr znaczących w 10-tej iteracji, wyniki znacznie się zmieniały, aż w końcu w 40-stym kroku otrzymałem kompletnie inny wynik. Jak widać obcięcie znaczących cyfr w obliczeniach, prowadzi do znaczącego błędu, niepoprawnego wyniku. Można stwierdzić, że wynik końcowy różnił się, aż cztero-krotnie.

W podpunkcie (b) porównując wyniki w dwóch różnych arytmetykach Float32 oraz Float64, można zauważyć, że wyniki się różnią od siebie. Pierwsze iteracje dają dość poprawne wyniki, lecz im dalej tym błąd względny rośnie. Każda następna iteracja generuje błąd poprzez przybliżenia cyfr po przecinku. Niestety niezależnie od arytmetyki skończonej w komputerze, za każdym razem obliczenia numeryczne obarczamy błędem obliczeniowym oraz wszelkimi przybliżeniami.

6. Zadanie 6

6.1. Opis problemu

Wykonać eksperymenty dla podanego równania rekurencyjnego:

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

gdzie *c* jest pewną stałą.

Następujące eksperymenty:

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. $c = -1 \ i \ x_0 = 0.25$

Wykonać w języku Julia w arytmetyce Float64, 40 iteracji równania. Zaobserwować zachowanie generowanych ciągów.

6.2. Rozwiązanie problemu

Utworzyłem program w języku Julia, który oblicza wyżej podane eksperymenty. Wyniki z programu załączam w Ryc. 6.1 i Ryc. 6.2.

6.3. Wyniki

Ryc. 6.1 – tabela przestawia wyniki eksperymentów nr 1, 2, 3, 4

iteracja	*Eksperyment (1)*	*Eksperyment (2)*	*Eksperyment (3)*	*Eksperyment (4)*
1	-1.000000000000000	2.000000000000000	1.999999999999960	0.000000000000000
2	-1.000000000000000	2.000000000000000	1.999999999999840	-1.000000000000000
3	-1.000000000000000	2.000000000000000	1.999999999999361	0.000000000000000
4	-1.000000000000000	2.000000000000000	1.999999999997442	-1.000000000000000
5	-1.000000000000000	2.000000000000000	1.999999999989768	0.000000000000000
10	-1.000000000000000	2.000000000000000	1.999999989522621	-1.000000000000000
15	-1.000000000000000	2.000000000000000	1.999989271173494	0.000000000000000
20	-1.000000000000000	2.000000000000000	1.989023726436175	-1.000000000000000
25	-1.000000000000000	2.000000000000000	-1.955009487525616	0.000000000000000
30	-1.000000000000000	2.000000000000000	1.738500213821511	-1.000000000000000
35	-1.000000000000000	2.000000000000000	-1.335447840993894	0.000000000000000
36	-1.000000000000000	2.000000000000000	-0.216579063984746	-1.000000000000000
37	-1.000000000000000	2.000000000000000	-1.953093509043491	0.000000000000000
38	-1.000000000000000	2.000000000000000	1.814574255067817	-1.000000000000000
39	-1.000000000000000	2.000000000000000	1.292679727154924	0.000000000000000
40	-1.000000000000000	2.000000000000000	-0.328979123002670	-1.000000000000000

Ryc. 6.2 – tabela przestawia wyniki eksperymentów nr 5, 6, 7

iteracja	*Eksperyment (5)*	*Eksperyment (6)*	*Eksperyment (7)*
1	0.000000000000000	-0.437500000000000	-0.937500000000000
2	-1.000000000000000	-0.808593750000000	-0.121093750000000
3	0.000000000000000	-0.346176147460938	-0.985336303710938
4	-1.000000000000000	-0.880162074929103	-0.029112368589267
5	0.000000000000000	-0.225314721856496	-0.999152469995123
10	-1.000000000000000	-0.999620188061125	-0.000000000065931
15	0.000000000000000	-0.000000000002662	-1.000000000000000
20	-1.000000000000000	-1.000000000000000	0.000000000000000
25	0.000000000000000	0.000000000000000	-1.000000000000000
30	-1.000000000000000	-1.000000000000000	0.000000000000000
35	0.000000000000000	0.000000000000000	-1.000000000000000
36	-1.000000000000000	-1.000000000000000	0.000000000000000
37	0.000000000000000	0.000000000000000	-1.000000000000000
38	-1.000000000000000	-1.000000000000000	0.000000000000000
39	0.000000000000000	0.000000000000000	-1.000000000000000
40	-1.000000000000000	-1.000000000000000	0.000000000000000

6.4. Wnioski

Po wszystkich przeprowadzonych eksperymentach i obserwacjach na uzyskanych wynikach, można podzielić na ciągi stabilne oraz niestabilne, gdzie wyniki są chwiejne i zmienne. Eksperymenty nr 1 oraz nr 2 dają nam oczekiwane wyniki, stabilne. Zaś już od trzeciego eksperymentu zachodzą zmiany. W Eksperymencie nr 3 operujemy na liczbach, które arytmetyka Float64 nie jest w stanie obliczyć dokładnie, gdzie zachodzi przybliżanie wyników, czyli pojawia się błąd w każdej kolejnej iteracji. Z każdą kolejną iteracją błąd narasta. Takie algorytmy są niestabilne i niechciane przez osoby działające na obliczeniach numerycznych.

Przy eksperymentach nr 4 oraz nr 5 uzyskujemy ciągi składające się z liczb całkowitych, które nie są wstanie wyskoczyć poza precyzję arytmetyki Float64. Taki rodzaj algorytmów jest jak najbardziej stabilny.

Ostatnie dwa eksperymenty (nr 6 oraz nr 7) są szczególnym przypadkiem, ponieważ ciągi w pewnym momencie są tak małe, że arytmetyka Flaot64 wymaga zaokrąglenia danych, co prowadzi do kompletnie błędnych wyników. Można zobaczyć te zjawisko już w trzynastej iteracji działania algorytmu.