Eliza Zawisza, 244967

SPRAWOZDANIE 2

ZADANIE 1

Opis problemu:

W zadaniu należało powtórzyć zadanie 5 z listy 1, w którym obliczana była suma iloczynu skalarnego dwóch wektorów na 4 różne sposoby, ale z usunięciem ostatniej 9 z x_4 i ostatniej 7 z x_5.

x = [2.718281828, −3.141592654, 1.414213562, 0.577215664, 0.301029995]
y = [1486.2497, 878366.9879, −22.37492, 4773714.647, 0.000185049].

Rozwiązanie:

W celu rozwiązania zadania użyte zostały algorytmy zawarte w pliku *zadanie1.jl*. Poniżej dwa przykładowe kody:

```
function a(Type)
   sum = Type(0.0)
   for i = 1:n
      sum+= Type(x[i])*Type(y[i])
   end
   return sum
end
```

```
function c(Type)
   productArray = computeScalarProduct(Type)
   negativeElements=[]
   positiveElements=[]

   for i = 1:length(productArray)
      if productArray[i] > 0.0
         push!(positiveElements, productArray[i])
      else
         push!(negativeElements, productArray[i])
      end
   end

   sort!(negativeElements)
   sort!(positiveElements, rev = true)

   sum = computeSortSum(negativeElements, positiveElements, Type)
   return sum
end
```

Wyniki oraz ich interpretacja:

	Zadanie 5 z Listy 1		Zadanie 1 z Listy 2	
Sposób	**Float32**	**Float64**	**Float32**	**Float64**
A – w przód	-0.4999443	1.0251881368296672e-10	-0.4999443	-0.004296342739891585
B – w tył	-0.4543457	-1.5643308870494366e-10	-0.4543457	-0.004296342998713953
C – od największego do najmniejszego	-0.5	0.0	-0.5	-0.004296342842280865
D – od najmniejszego do największego	-0.5	0.0	-0.5	-0.004296342842280865

Z powyższej tabeli łatwo można dostrzec, że dla Float32 usunięcie ostatniej cyfry nie zmieniło ostatecznego wyniku sumy iloczynu skalarnego. Natomiast w przypadku arytmetyki Float64 zaszły znaczące zmiany.

Wnioski:

Identyczność wyników pomimo zmienionych danych w arytmetyce Float32 wynika ze zbyt małej precyzji zapisu liczb zmiennopozycyjnych w tej arytmetyce. Poszczególne składowe wektorów nie są przechowane w sposób dokładny, co rzutuje na wynik. W arytmetyce Float64 precyzja jest już znacznie lepsza, dzięki czemu liczby są przechowywane dokładniej. Wyniki nawet przy niewielkiej zmianie danych wejściowych różnią się od siebie. Dużą różnicę można zobaczyć porównując ze sobą wyniki dwóch pierwszych algorytmów, których rząd wielkości jest różny aż 10-krotnie (w przypadku algorytmu A i B). Duża wrażliwość algorytmu na niewielkie zmiany w danych wejściowych, jest dowodem na to, że zadanie to zostało źle uwarunkowane.

ZADANIE 2

Opis problemu:

W zadaniu należało narysować wykres funkcji $f(x) = e^x \ln(1 + e^{-x})$ w dwóch dowolnych programach do wizualizacji oraz obliczyć $\lim_{x\to\infty} f(x)$ i porównać otrzymane wyniki.

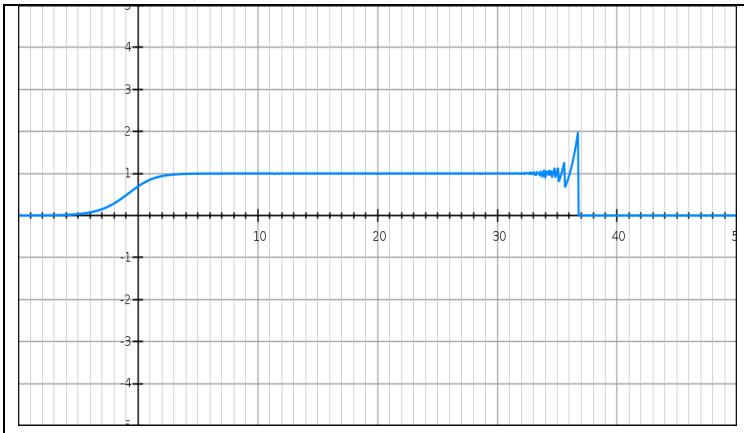
Rozwiązanie:

W celu wygenerowania wykresów użyłam programu gnuplot oraz pakietu Plots.jl oraz GraphSketch. Wygenerowane zostały wykresy porównawcze dla arytmetyki Float64, dla przedziału [30,40] oraz [-10,50]. Za pomocą programu gnuplot i GraphSketch wygenerowane zostały wykresy dla przedziałów [-10, 50] oraz [30,40]. Do obliczenia granicy funkcji został użyty pakiet SymPy. Poniżej zaimplementowana funkcja: Kody źródłowe programu znajdują się w pliku *zadanie2.jl*

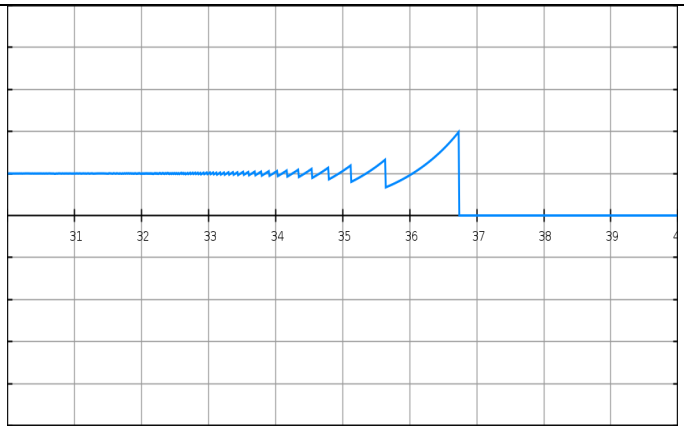
```
@vars x real=true
f(x)= exp.(x) * log.(1.0 + exp.(-x))

print("Limit of f(x) -> oo is " ,limit(f(x), x,oo))
```

Wyniki oraz ich interpretacja:

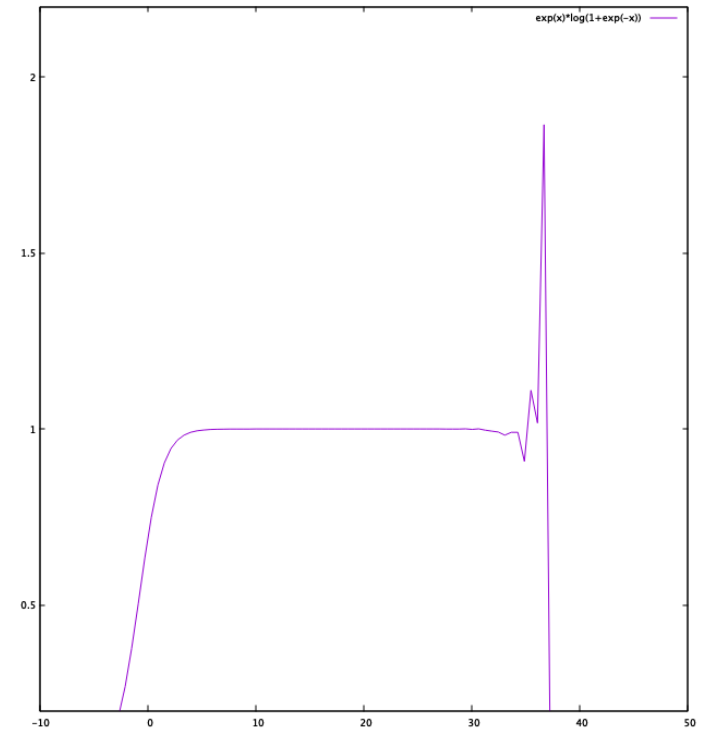


Zakres [-10, 50]

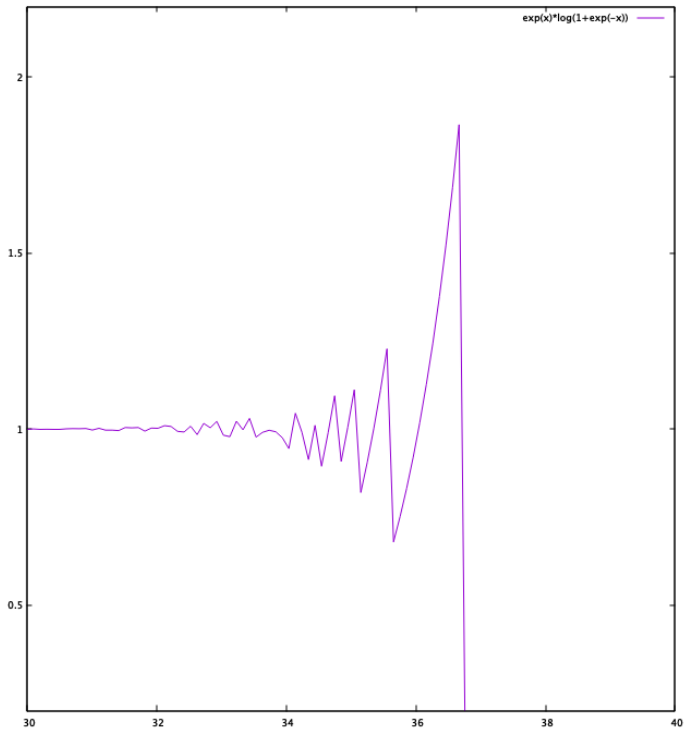


Zakres [30, 40]

Wykresy wygenerowane za pomocą programu GraphSketch

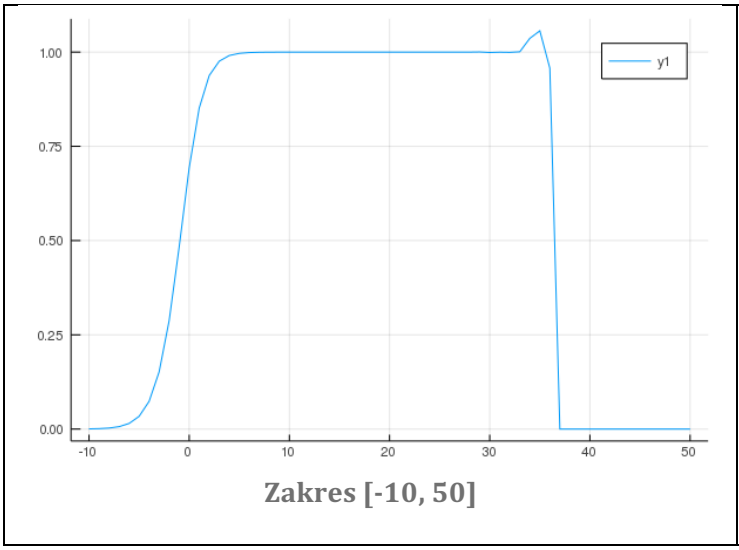


Zakres [-10, 50]

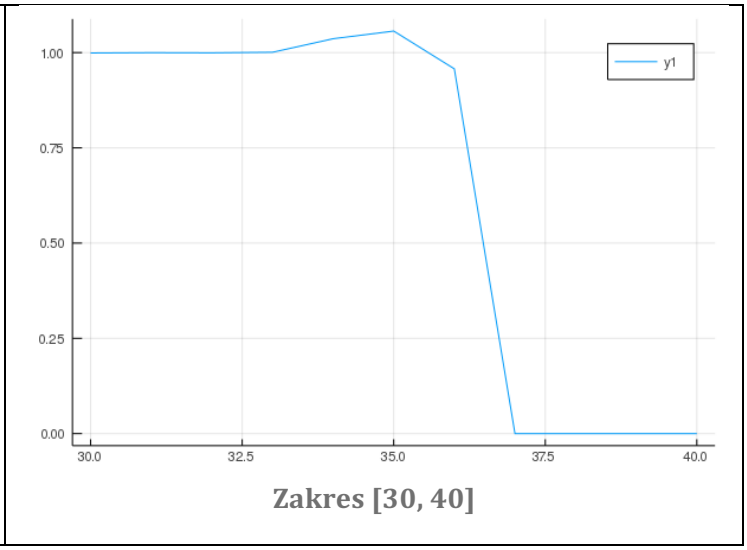


Zakres [30, 40]

Wykresy wygenerowane za pomocą programu Gnuplot



Zakres [-10, 50]



Zakres [30, 40]

Wykresy wygenerowane za pomocą pakietu Plots.jl

Na powyższych wykresach widać, że funkcja f(x) na początku rośnie. Na przedziale (3, 30) jest stała, a potem jej wartości dla przedziału [30, 37) zaczynają oscylować pomiędzy (0.6, 2), aż do x = 37, gdzie wartość funkcji spada do 0.

Wnioski:

Obliczając granicę funkcji otrzymałam prawidłowy wynik, czyli 1. Jak widać na powyższych wykresach jest on inny od wyników, które można z nich odczytać. Z powyższych wykresów można wywnioskować, że granicą funkcji jest 0, a nie jak zostało to policzone 1. Oscylacje wartości funkcji na pewnym odcinku wynikają z mnożenia bardzo dużej wartości funkcji e^x przez bardzo małą wartość logarytmu. Dla coraz większych x: $e^{-x} \approx 0$, co zostaje całkowicie pochłonięte przez 1 (mamy tutaj do czynienia z utratą cyfr znaczących). dlatego obliczamy wartość funkcji ln(1), która wynosi 0, co powoduje zbieżność funkcji do 0. Łatwo można zauważyć, że małe błędy popełnione na poszczególnych etapach obliczenia znacząco rzutują na wynik, co powoduje, ze zadanie to zostało źle uwarunkowane.

ZADANIE 3

Opis problemu:

W zadaniu należało obliczyć układ równań liniowych $Ax = b$.
Dla danej macierzy współczynników, którą należało wygenerować na dwa sposoby:

1. $A = H_n$ gdzie H_n jest macierzą Hilberta stopnia n wygenerowaną za pomocą funkcji A=hilb(n)
2. $A = R_n$ gdzie R_n jest losową macierzą stopnia n z zadanym wskaźnikiem uwarunkowania c, wygenerowaną za pomocą funkcji A=matcond(n, c)

Oraz wektora prawych stron, który zadany jest następująco: $b = Ax$, gdzie A jest wygenerowaną macierzą, a $x = (1, \ldots, 1)^T$.

Równanie należało rozwiązać za pomocą dwóch algorytmów: **eliminacji Gaussa** (x=A\b) oraz $\mathbf{x=A^{-1}b}$ (x=inv(A)*b). Zadanie należało wykonać dla macierzy Hilberta z rosnącym stopniem n>1 oraz dla macierzy losowej n = 5,10,20 z rosnącym współczynnikiem uwarunkowania c = 1, 10, 10^3, 10^7, 10^{12}, 10^{16}. Oraz policzyć błędy względne, porównując otrzymane wyniki z rozwiązaniem dokładnym: $x = (1, \ldots, 1)^T$.

Rozwiązanie:

W celu wygenerowania macierzy użyłam podane przez wykładowcę funkcje: *hilb.jl* oraz *matcond.jl.* Następnie używając poniższej funkcji obliczyłam wyniki dla podanych macierz na dwa różne sposoby i wyliczyłam błędy względne. Wszystkie kody zawarte są w pliku *zadanie3.jl*

```
function gaussElimination(A,b)
    return A\b
end
```

```
function inversionMethod(A,b)
    return inv(A)*b
end
```

```
function compute(A,n)
    x = ones(Float64, n)
    b = A*x
    gaussX = gaussElimination(A,b)
    inversionX = inversionMethod(A,b)
    gaussAproxError = approximationError(gaussX,x)
    inversionApproxError = approximationError(inversionX,x)

    println("Size: $(n)x$(n)\tRank: $(rank(A))")
    println("Cond: $(cond(A))")
    println("Gauss error: $(gaussAproxError)\nInversion error: $(inversionApproxError)\n")
end
```

Wyniki oraz ich interpretacja:

Macierz Hilberta:

Rozmiar macierzy	Rząd macierzy	Wskaźnik uwarunkowania	Błąd względny metody eliminacji Gaussa	Błąd względny metody inwersji
1x1	1	1.0	0.0.	0.0
2x2	2	19.28147006790397	5.661048867003676e-16	1.4043333874306803e-15
3x3	3	524.0567775860644	8.022593772267726e-15	0.0
4x4	4	15513.73873892924	4.137409622430382e-14	0.0
5x5	5	476607.25024259434	1.6828426299227195e-12	1.6828426299227195e-12
6x6	6	1.4951058642254665e7	2.618913302311624e-10	2.0163759404347654e-10
7x7	7	4.75367356583129e8	1.2606867224171548e-8	1.2606867224171548e-8
8x8	8	1.5257575538060041e10	6.124089555723088e-8	3.07748390309622e-7
9x9	9	4.931537564468762e11	3.8751634185032475e-6	4.541268303176643e-6
10x10	10	1.6024416992541715e13	8.67039023709691e-5	0.0002501493411824886
11x11	10	5.222677939280335e14	0.00015827808158590435	0.007618304284315809
12x12	11	1.7514731907091464e16	0.13396208372085344	0.258994120804705
13x13	11	3.344143497338461e18	0.11039701117868264	5.331275639426837
14x14	11	6.200786263161444e17	1.4554087127659643	8.71499275104814
15x15	12	3.674392953467974e17	3.674392953467974e17	7.344641453111494
16x16	12	7.865467778431645e17	54.15518954564602	29.84884207073541
17x17	12	1.263684342666052e18	13.707236683836307	10.516942378369349
18x18	12	2.2446309929189128e18	9.134134521198485	7.575475905055309
19x19	13	6.471953976541591e18	9.720589712655698	12.233761393757726
20x20	13	1.3553657908688225e18	7.549915039472976	22.062697257870493

Analizując powyższe wyniki łatwo można zauważyć, że wzrost współczynnika uwarunkowania macierzy, podobnie jak błędy względne obu metod rożnie wraz ze wzrostem rozmiaru macierzy.

Macierz losowa:

Rozmiar macierzy	Rząd macierzy	Wskaźnik uwarunkowania	Błąd względny metody eliminacji Gaussa	Błąd względny metody inwersji
5x5	5	1	1.85775845048325e-16	2.895107444979072e-16
5x5	5	10	1.2161883888976234e-16	1.7901808365247238e-16
5x5	5	10^3	1.4721741505275717e-14	1.6889408021831846e-14
5x5	5	10^7	2.9588895888438525e-10	2.815710991259624e-10
5x5	5	10^{12}	1.4737300087690685e-5	8.905500434821131e-6
5x5	4	10^{16}	0.4930636538596192	0.4822913854390829
10x10	10	1	1.7554167342883504e-16	2.016820280180126e-16
10x10	10	10	3.040470972244059e-16	2.248030287623391e-16
10x10	10	10^3	3.7155169009665004e-16	4.6997984367617645e-15
10x10	10	10^7	1.3641218363250239e-10	1.3575072949601916e-10
10x10	10	10^{12}	7.42501814094327e-6	6.62797217715222e-6
10x10	9	10^{16}	0.18659040146181743	0.1989965059366621
20x20	20	1	7.554413022338835e-16	5.324442579404919e-16
20x20	20	10	6.661338147750939e-16	5.219229170424792e-16
20x20	20	10^3	1.5551050487244147e-15	1.9459014222361975e-15
20x20	20	10^7	3.329481956383614e-11	4.7815554091702925e-11
20x20	20	10^{12}	3.5186509845981234e-5	3.620698362381902e-5
20x20	19	10^{16}	0.1407580964105188	0.1518923601189342

W przypadku macierzy losowej, można zauważyć podobne zależności jak w przypadku macierzy Hilberta, tzn. wraz ze wzrostem rozmiaru macierzy i jej wskaźnika uwarunkowania rośnie błąd względny obu metod, jednakże wzrost ten jest znacznie mniejszy niż w przypadku macierzy Hilberta.

Wnioski:

Na podstawie powyższych tabel można łatwo wysnuć wniosek, że błędy względne obliczeń zależą os wskaźnika uwarunkowania macierzy. Ponadto analizując tabelę z wynikami dla macierzy Hilberta widać, że zadanie to jest źle uwarunkowane, że wzrost stopnia macierzy powoduje, gwałtowny wzrost wskaźnika uwarunkowania, co sprawia, że błędy względne w obliczeniach są duże, dzięki temu można wysnuć wniosek, że dla macierzy Hilberta to zadanie jest źle uwarunkowane. Lekko inaczej wyglądają wyniki dla macierzy losowej. Podobnie wskaźnik uwarunkowania macierzy wpływa na błędy względne obliczeń, ale można łatwo dostrzec, że są one zdecydowanie mniejsze, a ich tempo wzrostu jest znacznie wolniejsze.

ZADANIE 4

Opis problemu:

Celem zadania było obliczyć 20 zer wielomianu Wilkinsona, zadanego na dwa sposoby:

1. W postaci kanonicznej:

$P(x) = x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} - 135585182899530x^{11} + 1307535010540395x^{10} - 10142299865511450x^9 + 63030812099294896x^8 - 311333643161390640x^7 + 1206647803780373360x^6 - 3599979517947607200x^5 + 8037811822645051776x^4 - 12870931245150988800x^3 + 13803759753640704000x^2 - 8752948036761600000x + 2432902008176640000$

2. W postaci iloczynowej:
$P(X) = (x-20)(x-19)(x-18)(x-17)(x-16)(x-15)(x-14)(x-13)(x-12)(x-11)$
$(x-10)(x-9)(x-8)(x-7)(x-6)(x-5)(x-4)(x-3)(x-2)(x-1)$.

W podpunkcie a należało obliczyć pierwiastki wielomianu używając funkcji *roots* z pakietu *Polynomials,* a następnie sprawdzić obliczone pierwiastko obliczając odpowiednio $|P(z_k)|$, $|p(z_k)|$ i $|z_k - k|$, dla k będącego z przedziału [1,20] oraz wyjaśnić poszczególne rozbieżności w wynikach.

W podpunkcie b należało powtórzyć eksperyment dokonując zmian w danych wejściowych, zmieniając współczynnik -210 na $-210-2^{-23}$.

Rozwiązanie:

W celu wygenerowania wielomianów zostały użyte funkcje ***Poly*** – dla wielomianu w postaci kanonicznej oraz ***poly*** – dla wielomianu w postaci iloczynowej. Następnie w celu obliczenia pierwiastków wielomianu użyłam funkcji ***roots***. Do obliczenia $|P(z_k)|$, $|p(z_k)|$ wykorzystałam funkcję ***polyval***. Wszystkie wykorzystane funkcje są z biblioteki *Polynomials*. Poniżej kilka przykładowych funkcji. Cały kod programu znajduje się w pliku *zadanie4.jl*.

```
coefficient = reverse(givenCoefficient)

P = Poly(coefficient)
p = poly(Float64[1.0:20.0;])

computedRoots = roots(P)


for k in 1:20
    zk = computedRoots[k]
    Pzk = abs(polyval(P, zk))
    pzk = abs(polyval(p, zk))
    n = abs(zk - k)
    println("zk = $zk, \t|P(xk)| = $Pzk, \t |p(zk)| = $pzk, \t |zk-k| = $n")
    # println("$zk;$Pzk;$pzk;$n")
end
```

Wyniki oraz ich interpretacja:

Poniżej tabela zawierające dane dla podpunktu a:

k	z_k	$\|P(z_k)\|$	$\|p(z_k)\|$	$\|z_k-k\|$
1	0.9999999999996989	36352.0	38400.0	3.0109248427834245e-13
2	2.0000000000283182	181760.0	198144.0	2.8318236644508943e-11
3	2.9999999995920965	209408.0	301568.0	4.0790348876384996e-10
4	3.9999999837375317	3.106816e6	2.844672e6	1.626246826091915e-8
5	5.000000665769791	2.4114688e7	2.3346688e7	6.657697912970661e-7
6	5.999989245824773	1.20152064e8	1.1882496e8	1.0754175226779239e-5
7	7.000102002793008	4.80398336e8	4.78290944e8	0.00010200279300764947
8	7.999355829607762	1.682691072e9	1.67849728e9	0.0006441703922384079
9	9.002915294362053	4.465326592e9	4.457859584e9	0.002915294362052734
10	9.990413042481725	1.2707126784e10	1.2696907264e10	0.009586957518274986
11	11.025022932909318	3.5759895552e10	3.5743469056e10	0.025022932909317674
12	11.953283253846857	7.216771584e10	7.2146650624e10	0.04671674615314281

13	13.07431403244734	2.15723629056e11	2.15696330752e11	0.07431403244734014
14	13.914755591802127	3.65383250944e11	3.653447936e11	0.08524440819787316
15	15.075493799699476	6.13987753472e11	6.13938415616e11	0.07549379969947623
16	15.946286716607972	1.555027751936e12	1.554961097216e12	0.05371328339202819
17	17.025427146237412	3.777623778304e12	3.777532946944e12	0.025427146237412046
18	17.99092135271648	7.199554861056e12	7.1994474752e12	0.009078647283519814
19	19.00190981829944	1.0278376162816e13	1.0278235656704e13	0.0019098182994383706
20	19.999809291236637	2.7462952745472e13	2.7462788907008e13	0.00019070876336257925

Poniżej tabela zawierająca wyniki do podpunktu b, ze zmienionym współczynnikiem.

k	z_k	$\|P(z_k)\|$	$\|p(z_k)\|$	$\|z_k-k\|$
1	0.9999999999998357 + 0.0im	20992.0	22016.0	1.6431300764452317e-13
2	2.0000000000550373 + 0.0im	349184.0	365568.0	5.503730804434781e-11
3	2.99999999660342 + 0.0im	2.221568e6	2.295296e6	3.3965799062229962e-9
4	4.000000089724362 + 0.0im	1.046784e7	1.0729984e7	8.972436216225788e-8
5	4.99999857388791 + 0.0im	3.9463936e7	4.3303936e7	1.4261120897529622e-6
6	6.000020476673031 + 0.0im	1.29148416e8	2.06120448e8	2.0476673030955794e-5
7	6.99960207042242 + 0.0im	3.88123136e8	1.757670912e9	0.00039792957757978087
8	8.007772029099446 + 0.0im	1.072547328e9	1.8525486592e10	0.007772029099445632
9	8.915816367932559 + 0.0im	3.065575424e9	1.37174317056e11	0.0841836320674414
10	10.095455630535774 - 0.6449328236240688im	7.143113638035824e9	1.4912633816754019e12	0.6519586830380406
11	10.095455630535774 + 0.6449328236240688im	7.143113638035824e9	1.4912633816754019e12	1.1109180272716561
12	11.793890586174369 - 1.6524771364075785im	3.357756113171857e10	3.2960214141301664e13	1.665281290598479
13	11.793890586174369 + 1.6524771364075785im	3.357756113171857e10	3.2960214141301664e13	2.045820276678428
14	13.992406684487216 - 2.5188244257108443im	1.0612064533081976e11	9.545941595183662e14	2.5188358711909045
15	13.992406684487216 + 2.5188244257108443im	1.0612064533081976e11	9.545941595183662e14	2.7128805312847097
16	16.73074487979267 - 2.812624896721978im	3.315103475981763e11	2.7420894016764064e16	2.9060018735375106
17	16.73074487979267 + 2.812624896721978im	3.315103475981763e11	2.7420894016764064e16	2.825483521349608
18	19.5024423688181 - 1.940331978642903im	9.539424609817828e12	4.2525024879934694e17	2.454021446312976
19	19.5024423688181 + 1.940331978642903im	9.539424609817828e12	4.2525024879934694e17	2.004329444309949
20	20.84691021519479 + 0.0im	1.114453504512e13	1.3743733197249713e18	0.8469102151947894

Analizując tabelę z wynikami z podpunktu a można zobaczyć, że dla poszczególnych miejsc zerowych wartości wielomianu w postaci kanonicznej różnią się od wartości wielomianu w punkcie x_k dla postaci ilorazowej. Można też dostrzec różnice pomiędzy pierwiastkami rzeczywistymi (w tabeli kolumna k), a obliczonymi (w tabeli kolumna z_k). W przypadku zmiany współczynnika wielomianu przy x^{19} w podpunkcie B jako pierwiastki wielomianu otrzymałam liczby zespolone. Niewielka zmiana we współczynniku wielomianu spowodowała duże zmiany w wynikach końcowych. Zauważmy, że zarówno w przypadku a, jak i b wartości wielomianu w miejscach 0, nigdy nie osiągają 0, które powinno być wartością wielomianu w miejscu zerowym.

Wnioski:

Na podstawie przeprowadzonych eksperymentów i powyższych wyników łatwo można zobaczyć, że problem znalezienia miejsc zerowych wielomianu Wilkinsona jest zadaniem źle uwarunkowanym. W przykładzie B dokładnie można dostrzec, jak minimalna zmiana współczynnika wpływa na pojawiające się błędy obliczeń. Błędy w tym zadaniu wynikają również z ograniczeń arytmetyki FLoat64, w której zadanie było realizowane. Arytmetyka Float64 ma 15-17 cyfr znaczących w systemie dziesiętnym, przez co wartości poszczególnych współczynników nie mogą zostać poprawnie przechowywane i kumulują one błędy, które później rzutują na duży błąd wyników.

ZADANIE 5

Opis problemu:

Celem zadania było rozważyć równanie rekurencyjne modelu wzrostu populacji,
$p_{n+1} \coloneqq p_n + rp_n(1 - p_n)$, dla n=0,1,...; Gdzie r jest pewną daną stałą, $r(1 - p_n)$ czynnikiem wzrostu populacji, a p_0 jest wielkością populacji stanowiącą procent maksymalnej wielkości populacji dla danego środowiska.

W zadaniu należało przeprowadzić dwa eksperymenty:
1. Dla $p_0 = 0.01,\ r = 3$ wykonać 40 iteracji równania rekurencyjnego, a następnie wykonać drobną modyfikację, czyli wykonać 10 iteracji dla tych samych daych, dokonać obcięcia liczby do 3 cyfr po przecinku i kontynuować dalej do 40 iteracji. (Arytmetyka Float64).
2. Dla danych $p_0 = 0.01,\ r = 3$ wykonać 40 iteracji równania rekurencyjnego w arytmetyce Float32 i Float64.

Rozwiązanie:

W celu obliczenia zadania użyłam dwóch funkcji, które obliczają mi równanie rekurencyjne. Pierwsza wykonuje kolejno 40 iteracji, druga natomiast uwzględnia zadane w zadaniu modyfikacje. Poniżej kody funkcji, cały kod programu znajduje się w pliku *zadanie5.jl*

```
function basicAlgorithm(Type, p, r)
    results = Type[]
    for i in 1:40
        p = p + Type(r * p * (Type(1.0) - p))
        push!(results, p)
    end
    return results
end
```

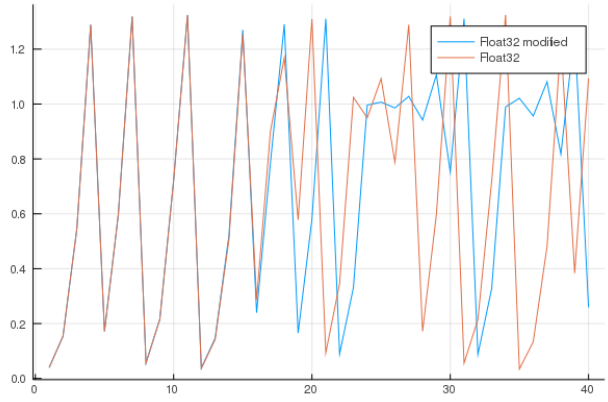
```
function modifiedAlgorithm(Type, p, r)
    results = Type[]
    for i in 1:10
        p = p + Type(r * p * (Type(1.0) - p))
        if i == 10
            p = trunc(p, digits=3)
        end
        push!(results, p)
```

```
    end
    for i in 12:40
        p = p + Type(r * p * (Type(1.0) - p))
        push!(results, p)
    end
    return results
end
```

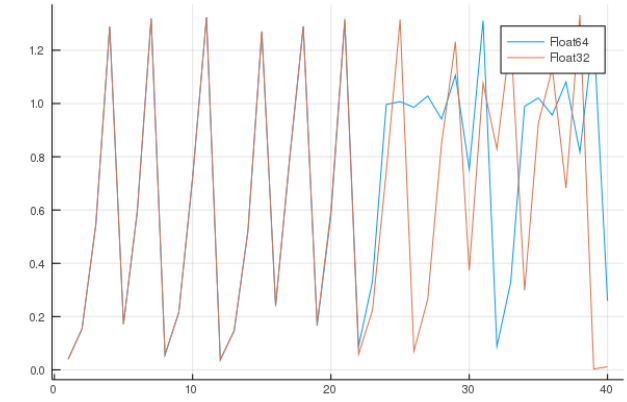
Wyniki oraz ich interpretacja:

n	Sposób 1 arytmetyka Float32	Sposób 2 arytmetyka Float32
1	0.0397	0.0397
2	0.15407173	0.15407173
3	0.5450726	0.5450726
4	1.2889781	1.2889781
5	0.1715188	0.1715188
6	0.5978191	0.5978191
7	1.3191134	1.3191134
8	0.056273222	0.056273222
9	0.21559286	0.21559286
10	***0.7229306***	***0.722***
11	1.3238364	1.3241479
12	0.037716985	0.036488414
13	0.14660022	0.14195944
14	0.521926	0.50738037
15	1.2704837	1.2572169
16	0.2395482	0.28708452
17	0.7860428	0.9010855
18	1.2905813	1.1684768
19	0.16552472	0.577893
20	0.5799036	1.3096911
21	1.3107498	0.09289217
22	0.088804245	0.34568182
23	0.3315584	1.0242395
24	0.9964407	0.94975823
25	1.0070806	1.0929108
26	0.9856885	0.7882812
27	1.0280086	1.2889631
28	0.9416294	0.17157483
29	1.1065198	0.59798557
30	0.7529209	1.3191822
31	1.3110139	0.05600393
32	0.0877831	0.21460639
33	0.3280148	0.7202578
34	0.9892781	1.3247173
35	1.021099	0.034241438
36	0.95646656	0.13344833
37	1.0813814	0.48036796
38	0.81736827	1.2292118
39	1.2652004	0.3839622
40	0.25860548	1.093568

n	Arytmetyka Float32	Arytmetyka Float64
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



Wykres dla Float32 i zmodyfikowanego Float32



Wykres dla Float32 i Float64

Analizując powyższe tabele łatwo można zauważyć, że zarówno obcięcie wyrazu p_{n+1} po 10 operacjach, jak i zmiana arytmetyki sprawia, że wyniki są inne.

Wnioski:

W zadaniu mamy do czynienia ze sprzężeniem zwrotnym, czyli procesem, w którym dane wyjściowe jednej iteracji są zarazem danymi wejściowymi kolejnej. Przeprowadzone eksperyment pokazują, jak niewielkie zmiany (obcięcie w 10 iteracji wyniku do 3 miejsc po przecinku) potrafią się nawarstwiać, by w ostateczności mieć duży wpływ na wynik. Takie zjawisko w matematyce zostało uznane jako chaos sprzężeń zwrotnych. Jak można zauważyć na powyższych wykresach proces sprzężenia zwrotnego jest niestabilny. W drugim eksperymencie należało jedynie przeprowadzić niezmodyfikowaną iterację dla dwóch różnych arytmetyk: Float32 i Float64. Jak widać na wykresie oraz w tabeli zmiana arytmetyki również wpływa na niestabilność wyników. Począwszy od 20-iteracji wyniki zaczynają coraz bardziej od siebie odbiegać, jest to niewątpliwie spowodowane różnicą w precyzjach tych arytmetyk. Jest to kolejne zadanie, które ukazuje jak duży wpływ na wyniki końcowe mają dane wejściowe programu.

ZADANIE 6

Opis problemu:

Celem zadania było rozważyć następujące równanie rekurencyjne: $x_{n+1} \coloneqq x_n^2 + c$, dla n = 0,1, … , gdzie c jest odpowiednio:

1. c = -2, x_0 = 1
2. c = -2, x_0 = 2
3. c = -2, x_0 = 1.99999999999999
4. c = -1, x_0 = 1
5. c = -1, x_0 = -1
6. c = -1, x_0 = 0.75
7. c = -1, x_0 = 0.25

dla arytmetyki Float64, wykonując 40 iteracji oraz przeprowadzić iterację graficzną równania.

Rozwiązanie:

W celu rozwiązania zadania użyłam rekurencyjnie wywoływanej funkcji compute. Do wygenerowania wykresów, dla poszczególnych przypadków użyto funkcji generatePlot oraz biblioteki *Plots.jl.* Wszystkie kody źródłowe zawarte są w liku *zadanie6.jl.*

```
function compute(x0, n, c)
   if n == 0 return x0
     else
      x = compute(x0, n-1, c)
      return (x^2 + c)
     end
end
```

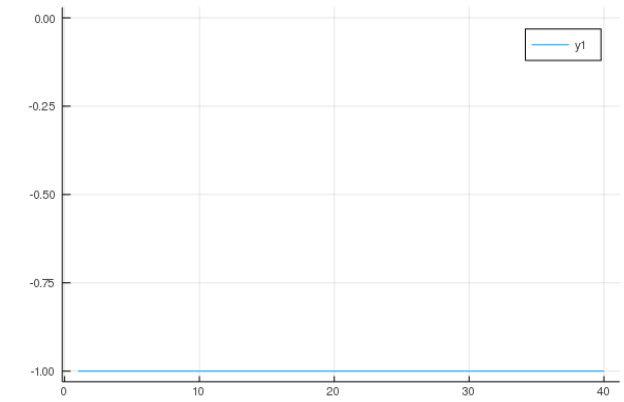
```
function generatePlot(Y, filename)
  plot(x = 1:40, Y)
  savefig(filename)
  println(Y)
end
```

Wyniki oraz ich interpretacja:

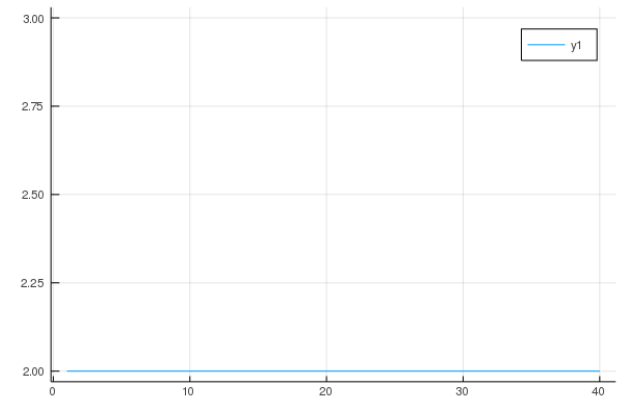
Wyniki dla poszczególnych iteracji wszystkich 7 przypadków.

n	Zestaw 1	Zestaw 2	Zestaw 3	Zestaw 4	Zestaw 5	Zestaw 6	Zestaw 7
1	-1.0	2.0	1.99999999999996	0.0	0.0	-0.4375	-0.9375
2	-1.0	2.0	1.9999999999998401	-1.0	-1.0	-0.80859375	-0.12109375
3	-1.0	2.0	1.9999999999993605	0.0	0.0	-0.3461761474609375	-0.9853363037109375
4	-1.0	2.0	1.999999999997442	-1.0	-1.0	-0.8801620749291033	-0.029112368589267135
5	-1.0	2.0	1.9999999999897682	0.0	0.0	0.2253147218564956	-0.9991524699951226
6	-1.0	2.0	1.9999999999590727	-1.0	-1.0	-0.9492332761147301	-0.0016943417026455965
7	-1.0	2.0	1.999999999836291	0.0	0.0	-0.0989561875164966	-0.9999971292061947
8	-1.0	2.0	1.9999999993451638	-1.0	-1.0	0.9902076729521999	-5.741579369278327e-6
9	-1.0	2.0	1.9999999973806553	0.0	0.0	-0.01948876442658909	-0.9999999999670343
10	-1.0	2.0	1.999999989522621	-1.0	-1.0	-0.999620188061125	-6.593148249578462e-11
11	-1.0	2.0	1.9999999580904841	0.0	0.0	0.0007594796206411569	-1.0
12	-1.0	2.0	1.9999998323619383	-1.0	-1.0	-0.9999994231907058	0.0
13	-1.0	2.0	1.9999993294477814	0.0	0.0	-1.1536182557003727e-6	-1.0
14	-1.0	2.0	1.9999973177915749	-1.0	-1.0	0.9999999999986692	0.0
15	-1.0	2.0	1.9999892711734937	0.0	0.0	-2.6616486792363503e-12	-1.0
16	-1.0	2.0	1.9999570848090826	-1.0	-1.0	-1.0	0.0
17	-1.0	2.0	1.999828341078044	0.0	0.0	0.0	-1.0
18	-1.0	2.0	1.9993133937789613	-1.0	-1.0	-1.0	0.0
19	-1.0	2.0	1.9972540465439481	0.0	0.0	0.0	-1.0
20	-1.0	2.0	1.9890237264361752	-1.0	-1.0	-1.0	0.0
21	-1.0	2.0	1.9562153843260486	0.0	0.0	0.0	-1.0
22	-1.0	2.0	1.82677862987391	-1.0	-1.0	-1.0	0.0
23	-1.0	2.0	1.3371201625639997	0.0	0.0	0.0	-1.0
24	-1.0	2.0	-0.21210967086482313	-1.0	-1.0	-1.0	0.0
25	-1.0	2.0	-1.9550094875256163	0.0	0.0	0.0	-1.0
26	-1.0	2.0	1.822062096315173	-1.0	-1.0	-1.0	0.0
27	-1.0	2.0	1.319910282828443	0.0	0.0	0.0	-1.0

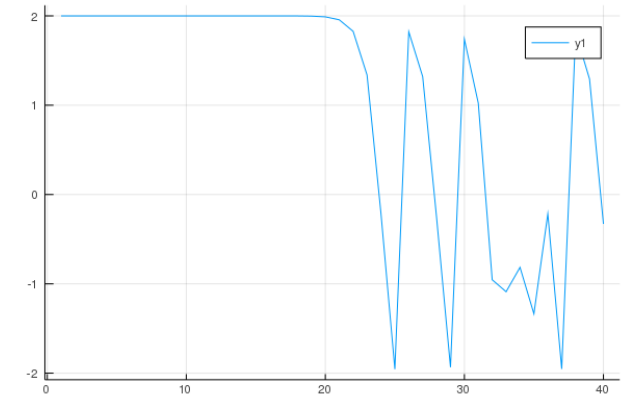
28	-1.0	2.0	-0.2578368452837396	-1.0	-1.0	-1.0	0.0
29	-1.0	2.0	-1.9335201612141288	0.0	0.0	0.0	-1.0
30	-1.0	2.0	1.7385002138215109	-1.0	-1.0	-1.0	0.0
31	-1.0	2.0	1.0223829934574389	0.0	0.0	0.0	-1.0
32	-1.0	2.0	-0.9547330146890065	-1.0	-1.0	-1.0	0.0
33	-1.0	2.0	-1.0884848706628412	0.0	0.0	0.0	-1.0
34	-1.0	2.0	-0.8152006863380978	-1.0	-1.0	-1.0	0.0
35	-1.0	2.0	-1.3354478409938944	0.0	0.0	0.0	-1.0
36	-1.0	2.0	-0.21657906398474625	-1.0	-1.0	-1.0	0.0
37	-1.0	2.0	-1.953093509043491	0.0	0.0	0.0	-1.0
38	-1.0	2.0	1.8145742550678174	-1.0	-1.0	-1.0	0.0
39	-1.0	2.0	1.2926797271549244	0.0	0.0	0.0	-1.0
40	-1.0	2.0	-0.328979123002670	-1.0	-1.0	-1.0	0.0



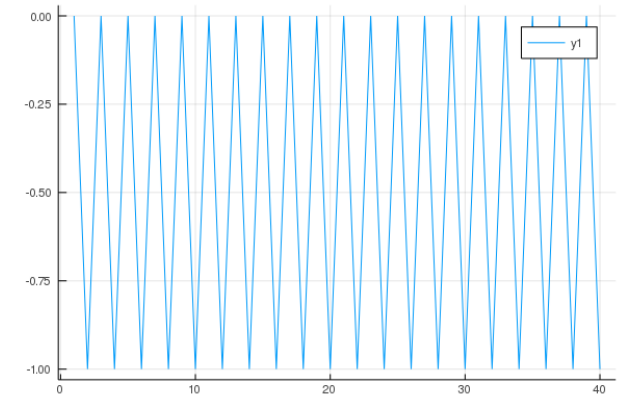
Wykres dla c = -2 i x_0 = 1



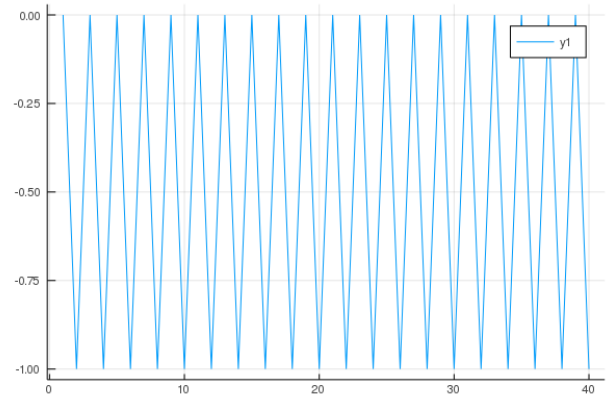
Wykres dla c = -2 i x_0 = 2



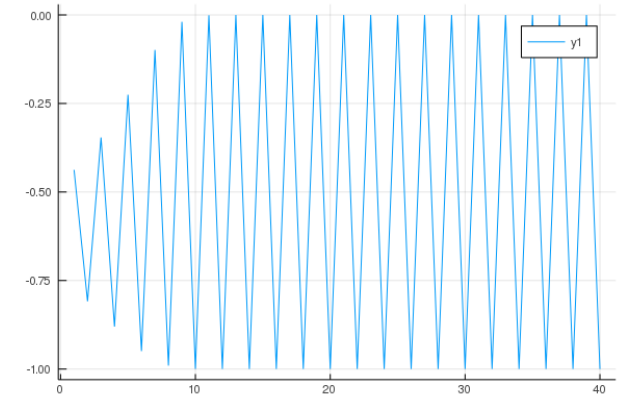
Wykres dla c = -2 i x_0 = 1.99999999999999



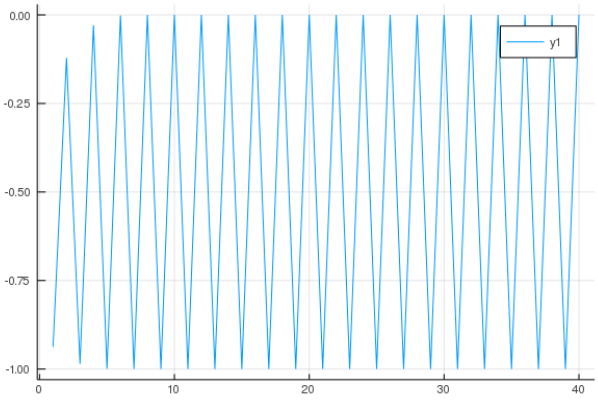
Wykres dla c = -1 i x_0 = 1



Wykres dla c = -1 i x_0 = -1



Wykres dla c = -1 i x_0 = 0.75



Wykres dla c = -1 i x_0 = 0.25

Na widocznych powyżej wykresach widać, że dla większości zestawów danych wyniki zachowują się stabilnie, czyli w określonych interwałach pojawiają się takie same wyniki. Jedynym przypadkiem, w którym można zauważyć kompletny brak stabilności wyników jest wykres 3, gdzie nie mamy do czynienia z okresową powtarzalnością wyników.

Wnioski:

Zadanie 6 jest bardzo podobne do zadania 5, w którym dokonujemy analizy rozwoju populacji, ponieważ w obu zadaniach mamy do czynienia ze sprzężeniem zwrotnym i dwoma przykładami jego zachowań: stabilnością i niestabilnością.

Analizując wyniki można łatwo wyciągnąć wnioski, że za niestabilność wyników odpowiedzialne jest podnoszenie liczb do kwadratu, jak można łatwo zauważyć, ilość cyfr znaczących z każdym podniesieniem liczby do kwadratu zwiększa się dwukrotnie, co przy określonej precyzji danej arytmetyki wpływa na otrzymane wyniki w kolejnych krokach. Dokładnie ilustruje to wykres dla trzeciego zestawu danych, gdzie c = -2 i x_0 = 1.99999999999999 mamy bardzo niestabilne wyniki, podczas gdy można by się spodziewać, że wartość x_0 jest zbliżona do 2 i wyniki będą zbliżone do 2.0 tak się nie dzieje.

Analizując wykresy można zobaczyć, że pozostałych zestawach danych mamy do czynienia ze swoistą stabilnością wyników. W przypadku 1 i 2 wyniki równanie rekurencyjnego są równe odpowiednio -1 i 2, podczas gdy dla przypadków 4,5,6,7 wyniki w każdej iteracji (poza kilkoma pierwszymi w przypadku, gdy c = -1 i x_0 = 0.75 oraz c = -1 i x_0 = 0.25) zbiegają na zmianę odpowiednio do 0 i -1. Łatwo zatem można wyciągnąć wnioski, że niektóre zestawy danych sprowadzają się do stabilnego zachowania wyników, inne natomiast sprawiają, że otrzymane wyniki działanie rekurencyjnego są bardzo niestabilne.