SPRAWOZDANIE I

METODY OBLICZENIOWE W NAUCE I TECHNICE

ARYTMETYKA KOMPUTEROWA

DAWID BIAŁKA

2019/2020

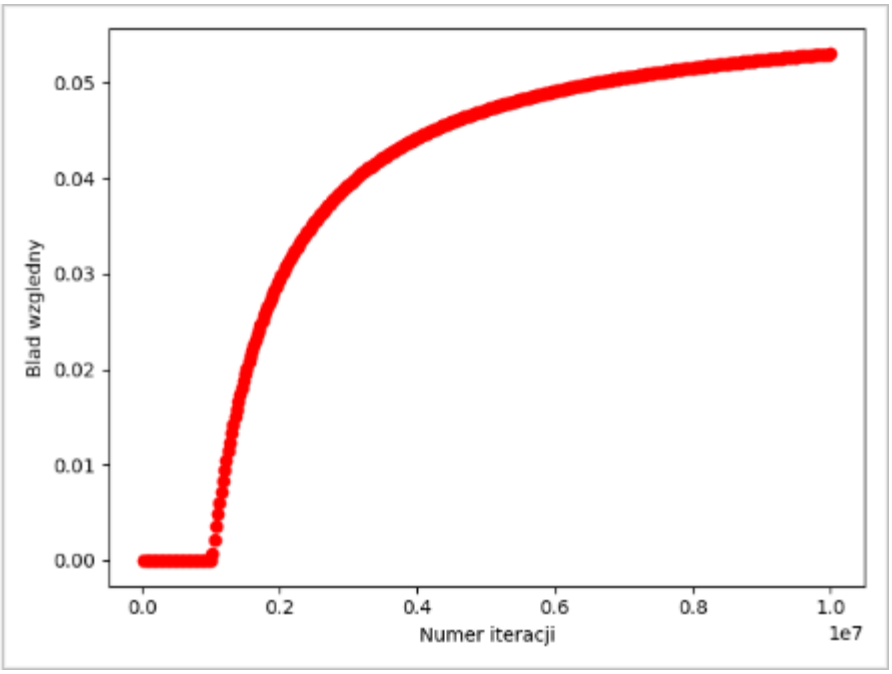
Zadanie 1 Sumowanie liczb pojedynczej precyzji

1. Napisz program, który oblicza sumę N liczb pojedynczej precyzji przechowywanych w tablicy o N = 107 elementach. Tablica wypełniona jest tą samą wartością v z przedziału [0.1, 0.9] np. v = 0.53125.
2. Wyznacz bezwzględny i względny błąd obliczeń. Dlaczego błąd względny jest tak duży?
3. W jaki sposób rośnie błąd względny w trakcie sumowania? Przedstaw wykres (raportuj wartość błędu co 25000 kroków) i dokonaj jego interpretacji.

```
import numpy as np
import time
import matplotlib.pyplot as plt

#Zad 1, 2, 3 ======================================
v = np.float(0.53125)
sum=np.float32(0.0)
precise = np.float32(v * (10 ** 7))
tmp_precise = np.float32(1.0)

tab = np.full((10**7), v, dtype=np.float32)
x_axis = np.arange(25000, 10**7 + 25000, 25000)
errors = np.zeros(10**7//25000)

start = time.time()
for i in range(10**7):
   sum = sum + tab[i]
   if i % 25000 == 0 and i > 0:
       tmp_precise =np.float32(v * i)
       errors[i//25000] = np.float32((tmp_precise - sum) / tmp_precise)
end = time.time()

print("Blad bezwzgledny")
print(abs(precise - sum))
print("Blad wzgledny")
print(abs((precise - sum) / precise * 100), "%")
print("Czas trwania sumowania:", (end - start))

plt.plot(x_axis, errors, 'ro')
plt.ylabel("Blad wzgledny")
plt.xlabel("Numer iteracji")
plt.show()
```

Dla pierwszego algorytmu sumowania otrzymujemy:

```
Blad bezwzgledny
281659.5
Blad wzgledny
5.301825702190399 %
Czas trwania sumowania: 8.372185468673706
```



Rys 1. Wykres błędu względnego od liczby iteracji

Błąd względny jest tak duży, ponieważ w miarę sumowania akumulator jest coraz większy i w pewnym momencie dodawane liczby różnią się znacznie rzędem wielkości i przy dodawaniu pewna część bitów małej liczby zostaje odcięta. W pewnym momencie może dojść do sytuacji, że przy sumowaniu bardzo dużej i bardzo małej liczby, ta mała liczba będzie „kasowana" a wynik będzie taki sam, jak przed operacją sumowania. Z wykresu widzimy, że na początku, gdy liczby są porównywalnego rzędu, błąd nie rośnie w ogóle.

4. Zaimplementuj rekurencyjny algorytm sumowania, działający jak na rysunku poniżej.
5. Wyznacz bezwzględny i względny błąd obliczeń. Dlaczego błąd względny znacznie zmalał?
6. Porównaj czas działania obu algorytmów dla tych samych danych wejściowych.

```
# Zad 4, 5, 6 ================================
def suma(l, r, array):
    if l==r: return array[r]

    q = (l+r)//2
    return suma(l, q, array) + suma(q+1, r, array)


start = time.time()
secondResult = suma(0, 10**7-1, tab)
end = time.time()
print("Blad bezwzgledny w rekurencyjnym")
print(abs(precise - secondResult))
print("Blad wzgledny w rekurencyjnym")
print(abs(precise - secondResult)/precise)
print("Czas trwania sumowania w rekurencyjnym")
print(end - start)
```

Dla algorytmu rekurencyjnego:

```
Blad bezwzgledny w rekurencyjnym
0.0
Blad wzgledny w rekurencyjnym
0.0
Czas trwania sumowania w rekurencyjnym
11.839621782302856
```

Błąd bezwzględny i względny w tym przypadku jest równy zero, ponieważ w tym algorytmie dodajemy za każdym razem do siebie dwie takie same liczby, więc nie występuje zjawisko odcinana bitów jak w przypadku wcześniejszego algorytmu. Algorytm rekurencyjny jest nieco wolniejszy.

7. Przedstaw przykładowe dane wejściowe, dla których algorytm sumowania rekurencyjnego zwraca niezerowy błąd.

```
# Zad 7 ===================================

tab1 = np.arange(0.0, (10 ** 4) + 0.2, 0.2)
precise2 = np.float32((10 ** 4)/2 * len(tab1))

result2 = suma(0, len(tab1) - 1, tab1)

print("Blad wzgledny Zad 7")
print(abs(precise2 - result2))
print("Blad bezwzgledny Zad 7")
print((abs(precise2 - result2)/precise2))
```

Przykładem takich danych są dane posortowane.

```
Blad wzgledny Zad 7
8.0
Blad bezwzgledny Zad 7
3.199936103675882e-08
```

Zadanie 2 Algorytm Kahana

1. Zaimplementuj algorytm Kahana
2. Wyznacz bezwzględny i względny błąd obliczeń dla tych samych danych wejściowych jak w przypadku testów z Zadania 1.
3. Wyjaśnij dlaczego w algorytm Kahana ma znacznie lepsze własności numeryczne? Do czego służy zmienna err?
4. Porównaj czasy działania algorytmu Kahana oraz algorytmu sumowania rekurencyjnego dla tych samych danych wejściowych.

```python
import numpy as np
import time

# 2 Zad 1, 2, 3 ==================================

v = np.float(0.53125)
precise = np.float32(v * (10 ** 7))
tab = np.full((10**7), v, dtype=np.float32)

sum=np.float32(0.0)
err=np.float32(0.0)
y=np.float32

temp=np.float32
start = time.time()
for i in range(10**7):
    y = tab[i] - err
    temp = sum + y
    err = (temp - sum) - y
    sum = temp
end = time.time()

print("Blad bezwzgledny w Kahanie")
print(precise - sum)
print("Blad wzgledny w Kahanie")
print((precise - sum)/precise)
print(end - start)
```

```
Blad bezwzgledny w Kahanie
0.0
Blad wzgledny w Kahanie
0.0
16.823793649673462
```

Dla algorytmu Kahana błąd względny i bezwzględny wynoszą 0. Jest to spowodowane tym, że do zmiennej err przypisujemy bity y, które są tracone podczas sumowania. W następnej iteracji te utracone bity dodajemy do kolejnej liczby z tablicy, która w stosunku do err nie jest na tyle duża, aby spowodować ucięcie bitów.

Zadanie 3 Sumy częściowe

Rozważ sumy częściowe szeregu definiującego funkcję dzeta Riemanna oraz funkcję eta Dirichleta. Dla s = 2, 3.6667, 5, 7.2, 10 oraz n = 50, 100, 200, 500, 1000 oblicz wartości funkcji ζ(s) i η(s) w pojedynczej precyzji sumując w przód, a następnie wstecz. Porównaj wyniki z rezultatami uzyskanymi dla podwójnej precyzji. Dokonaj interpretacji otrzymanych wyników.

```
import numpy as np


# 3 Suma Riemanna, sumowanie w przod, pojedyncza precyzja
rieman_sum = np.float32(0.0)
k_to_power = np.float32(1.0)
s_tab = np.array([2, 3.6667, 5, 7.2, 10], dtype=np.float32)
n_tab = np.array([50, 100, 200, 500, 1000])
tab_length = len(s_tab) * len(n_tab)
forward_sum = np.empty(tab_length, dtype=np.float32)
backward_sum = np.empty(tab_length, dtype=np.float32)


counter = 0
for s in s_tab:
    for n in n_tab:
        for k in range(1, n+1):
            tmp_k = np.float32(k)
            k_to_power = tmp_k ** s
            rieman_sum = rieman_sum + 1 / k_to_power
        forward_sum[counter] = rieman_sum
        counter = counter +1
        rieman_sum = 0

# Suma Riemanna, sumowanie w tyl, pojedyncza precyzja

counter = 0
for s in s_tab:
    for n in n_tab:
```

```
        for k in reversed(range(1, n+1)):
            tmp_k = np.float32(k)
            k_to_power = tmp_k ** s
            rieman_sum = rieman_sum + 1 / k_to_power
        backward_sum[counter] = rieman_sum
        counter = counter +1
        rieman_sum = 0

counter = 0
for s in s_tab:
    for n in n_tab:
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w przod",
forward_sum[counter])
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w tyl  ",
backward_sum[counter])
        counter = counter + 1

# 3 Suma Dirichleta, sumowanie w przod, pojedyncza precyzja

forward_sum = np.empty(tab_length, dtype=np.float32)
backward_sum = np.empty(tab_length, dtype=np.float32)
counter = 0

for s in s_tab:
    for n in n_tab:
        for k in range(1, n+1):
            tmp_k = np.float32(k)
            k_to_power = tmp_k ** s
            if k % 2 == 0:
                k_to_power = (-1) * k_to_power
            rieman_sum = rieman_sum + 1 / k_to_power
        forward_sum[counter] = rieman_sum
        counter = counter +1
        rieman_sum = 0

# Suma Dirichleta, sumowanie w tyl, pojedyncza precyzja

counter = 0
for s in s_tab:
    for n in n_tab:
        for k in reversed(range(1, n+1)):
            tmp_k = np.float32(k)
            k_to_power = tmp_k ** s
            if k % 2 == 0:
                k_to_power = (-1) * k_to_power
            rieman_sum = rieman_sum + 1 / k_to_power
        backward_sum[counter] = rieman_sum
        counter = counter +1
        rieman_sum = 0

counter = 0
for s in s_tab:
    for n in n_tab:
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w przod",
forward_sum[counter])
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w tyl  ",
backward_sum[counter])
        counter = counter + 1
```

```python
import numpy as np
import xlwt
from xlwt import Workbook

rieman_sum = np.float64(0.0)
k_to_power = np.float64(1.0)
s_tab = np.array([2, 3.6667, 5, 7.2, 10], dtype=np.float64)
n_tab = np.array([50, 100, 200, 500, 1000])
tab_length = len(s_tab) * len(n_tab)
forward_sum = np.empty(tab_length, dtype=np.float64)
backward_sum = np.empty(tab_length, dtype=np.float64)

wb = Workbook()
sheets = []


def sheet_init(name):

    sheet = wb.add_sheet(name)

    sheet.write(0, 0, 'Wartość s')
    sheet.write(0, 1, 'ζ w przód ')
    sheet.write(0, 2, 'ζ w tył')
    sheet.write(0, 3, 'η w przód')
    sheet.write(0, 4, 'η w tył')
    for i in range(5):
        sheet.write(i + 1, 0, s_tab[i])
    return sheet

# 3 Suma Riemanna, sumowanie w przod, podwojna precyzja

counter = 0
counter_excel = 0
i = 0
for n in n_tab:
    sheets.append(sheet_init(str(n)))
    for s in s_tab:
        for k in range(1, n+1):
            tmp_k = np.float64(k)
            k_to_power = tmp_k ** s
            rieman_sum = rieman_sum + 1 / k_to_power
        forward_sum[counter] = rieman_sum
        sheets[i].write(counter_excel + 1, 1, rieman_sum)
        counter = counter +1
        counter_excel = counter_excel + 1
        rieman_sum = 0
    i = i + 1
    counter_excel = 0
```

```
# Suma Riemanna, sumowanie w tyl, podwojna precyzja

counter = 0
i = 0
for n in n_tab:
    for s in s_tab:
        for k in reversed(range(1, n+1)):
            tmp_k = np.float64(k)
            k_to_power = tmp_k ** s
            rieman_sum = rieman_sum + 1 / k_to_power
        backward_sum[counter] = rieman_sum
        sheets[i].write(counter_excel + 1, 2, rieman_sum)
        counter = counter +1
        counter_excel = counter_excel + 1
        rieman_sum = 0
    i = i + 1
    counter_excel = 0

counter = 0
for n in n_tab:
    for s in s_tab:
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w przod,
podwojna precyzja ", forward_sum[counter])
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w tyl, podwojna
precyzja   ", backward_sum[counter])
        counter = counter + 1

# 3 Suma Dirichleta, sumowanie w przod, podwojna precyzja

forward_sum = np.empty(tab_length, dtype=np.float64)
backward_sum = np.empty(tab_length, dtype=np.float64)

counter = 0
i = 0
for n in n_tab:
    for s in s_tab:
        for k in range(1, n+1):
            tmp_k = np.float64(k)
            k_to_power = tmp_k ** s
            if k % 2 == 0:
                k_to_power = (-1) * k_to_power
            rieman_sum = rieman_sum + 1 / k_to_power
        forward_sum[counter] = rieman_sum
        sheets[i].write(counter_excel + 1, 3, rieman_sum)
        counter = counter +1
        counter_excel = counter_excel + 1
        rieman_sum = 0
    i = i + 1
    counter_excel = 0
```

```
# Suma Dirichleta, sumowanie w tyl, podwojna precyzja

counter = 0
i = 0
for n in n_tab:
    for s in s_tab:
        for k in reversed(range(1, n+1)):
            tmp_k = np.float64(k)
            k_to_power = tmp_k ** s
            if k % 2 == 0:
                k_to_power = (-1) * k_to_power
            rieman_sum = rieman_sum + 1 / k_to_power
        backward_sum[counter] = rieman_sum
        sheets[i].write(counter_excel + 1, 4, rieman_sum)
        counter = counter +1
        counter_excel = counter_excel + 1
        rieman_sum = 0
    i = i + 1
    counter_excel = 0

wb.save('podwojna_precyzja.xls')

counter = 0
for n in n_tab:
    for s in s_tab:
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w przod,
podwojna precyzja ", forward_sum[counter])
        print(f's = {s}, n = {n},', "Wartosc funkcji dla sumowania w tyl, podwojna
precyzja   ", backward_sum[counter])
        counter = counter + 1
```

Wartości ζ(s) i η(s) dla pojedynczej precyzji były takie same dla sumowania w przód i w tył.

Wartości ζ(s) i η(s) dla podwójnej precyzji:

Wartość s	ζ w przód	ζ w tył	η w przód	η w tył
2	1,625132733621530	1,625132733621530	0,822271031826029	0,822271031826029
3,6667	1,109399755154190	1,109399755154190	0,934693060030711	0,934693060030711
5	1,036927716716710	1,036927716716710	0,972119768926798	0,972119768926798
7,2	1,007227666476280	1,007227666476280	0,993527000661349	0,993527000661348
10	1,000994575127820	1,000994575127820	0,999039507598272	0,999039507598272

Tabela 1. n = 50

Wartość s	ζ w przód	ζ w tył	η w przód	η w tył
2	1,634983900184890	1,634983900184890	0,822417533374129	0,822417533374128
3,6667	1,109408797342150	1,109408797342150	0,934693321140066	0,934693321140067
5	1,036927752692960	1,036927752692950	0,972119770398159	0,972119770398159
7,2	1,007227666480650	1,007227666480650	0,993527000661618	0,993527000661618
10	1,000994575127820	1,000994575127820	0,999039507598272	0,999039507598272

Tabela 2. n = 100

Wartość s	ζ w przód	ζ w tył	η w przód	η w tył
2	1,639946546015000	1,639946546015000	0,822454595922551	0,822454595922551
3,6667	1,109410242333230	1,109410242333230	0,934693342108684	0,934693342108685
5	1,036927754988680	1,036927754988680	0,972119770445367	0,972119770445366
7,2	1,007227666480710	1,007227666480720	0,993527000661620	0,993527000661620
10	1,000994575127820	1,000994575127820	0,999039507598272	0,999039507598272

Tabela 3. n = 200

Wartość s	ζ w przód	ζ w tył	η w przód	η w tył
2	1,642936065514890	1,642936065514890	0,822465037424096	0,822465037424097
3,6667	1,109410490844070	1,109410490844070	0,934693343855875	0,934693343855875
5	1,036927755139390	1,036927755139390	0,972119770446895	0,972119770446893
7,2	1,007227666480710	1,007227666480720	0,993527000661620	0,993527000661620
10	1,000994575127820	1,000994575127820	0,999039507598272	0,999039507598272

Tabela 4. n = 500

Wartość s	ζ w przód	ζ w tył	η w przód	η w tył
2	1,643934566681560	1,643934566681560	0,822466533924111	0,822466533924113
3,6667	1,109410510842360	1,109410510842360	0,934693343914135	0,934693343914135
5	1,036927755143120	1,036927755143120	0,972119770446909	0,972119770446909
7,2	1,007227666480710	1,007227666480720	0,993527000661620	0,993527000661620
10	1,000994575127820	1,000994575127820	0,999039507598272	0,999039507598272

Tabela 5. n = 1000

Dla funkcji ζ(s) tam gdzie występują rozbieżności zaznaczyłem kolorem zielonym wynik bardziej dokładny, a żółtym mniej. W przypadku funkcji η(s) kolorem niebieskim zaznaczyłem kolorem niebieskim wartości, dla których występują rozbieżności, gdyż w tym przypadku nie jesteśmy w stanie określić, która wartość jest bardziej dokładna (dla ζ(s) można to było zrobić, ponieważ cały czas dodawaliśmy do siebie dodatnie liczby, więc wartość, która jest większa, jest bardziej dokładna. Żeby wytłumaczyć różnice dla funkcji ζ(s), gdy sumujemy w przód i w tył możemy rozpatrzyć błąd operacji zmiennoprzecinkowych ε dla obu sposobów.

$$fl(fl(x+y)+z) = fl((x+y)(1+\epsilon)+z) = ((x+y)(1+\epsilon)+z)(1+\epsilon) = (x+x\epsilon+y+y\epsilon+z)(1+\epsilon)$$

$$fl(fl(x+y)+z) = (x+y+z+\epsilon(x+y))(1+\epsilon)$$

$$fl(x+fl(y+z)) = fl(x+(y+z)(1+\epsilon)) = (x+(y+z)(1+\epsilon))(1+\epsilon) = (x+y+y\epsilon+z+z\epsilon)(1+\epsilon)$$

$$fl(x+fl(y+z)) = (x+y+z+\epsilon(z+y))(1+\epsilon)$$

Jeśli | x + y | < | y + z | to widzimy, że błąd dla drugiego sposobu jest większy. W przypadku sumowania w tył błąd też występuje, dlatego również te wartości mogą nie być dokładne.

Zadanie 4 Błędy zaokrągleń i odwzorowanie logistyczne

Rozważ odwzorowanie logistyczne dane następującym wzorem rekurencyjnym

$$x_{n+1} = r\, x_n\, (1 - x_n)$$

Przy czym $0 =< x_n =< 1$ i $r > 0$. Zbadaj zbieżność procesu iteracyjnego określonego tym równaniem w zależności od wartości parametru r oraz x_0.

Aby zbadać zbieżność procesu iteracyjnego stworzyłem interaktywny wykres wartości kolejnych elementów ciągu dla wartości stałej r z przedziału [0.0, 4.0] i $x_0 = 0.8$. Wykres znajduje się pod adresem:

https://gfycat.com/thirdultimategalapagosdove

```
import matplotlib.pyplot as plt
import numpy as np
import imageio

def logistic(x, r):
    return r * x * (1 - x)

def plot_for_offset(r, y_max):
    t = np.arange(0.0, 100, 1)
    s = []
    x = 0.7
    s.append(x)
    for j in range(99):
        new_x = logistic(x, r)
        s.append(new_x)
        x = new_x

    fig, ax = plt.subplots(figsize=(10,5))
    ax.plot(t, s)
```

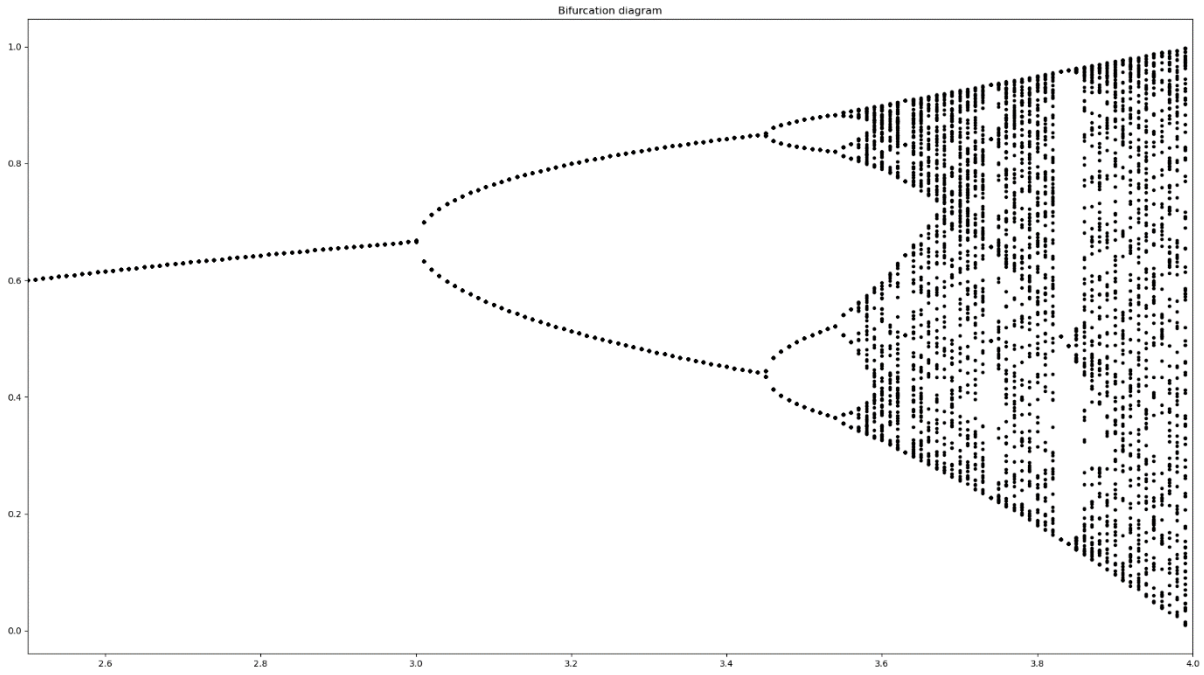
```
    ax.grid()
    ax.set(xlabel=f'n, r = {r}', ylabel='xn',
           title='Odwzorowanie logistyczne')

    ax.set_ylim(0, y_max)


    fig.canvas.draw()
    image = np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
    image  = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))

    return image

kwargs_write = {'fps':1.0, 'quantizer':'nq'}
imageio.mimsave('./zbieznosc.gif', [plot_for_offset(i/100, 1) for i in
range(400)], fps=20)
```

Analizując dany wykres możemy wywnioskować:

- dla r od 0.0 do 2.0 elementy szybko zbiegają do swojej granicy
- dla r od 2.0 do 3.0 ciąg też zbiega do granicy, ale w początkowej fazie występują wahania
- dla r od 3.0 do 3.5 otrzymujemy oscylator
- dla r od 3.5 do 4.0 otrzymujemy drgania chaotyczne

Poniżej diagram bifurkacyjny:

```
import matplotlib.pyplot as plt
import numpy as np

r_tab = np.arange(2.5, 4, 0.01)
x = 0.8

def logistic(x, r):
    return r * x * (1 - x)


fig, ax1 = plt.subplots(figsize=(19.20, 10.80))
for r in r_tab:
    for i in range(1000):
        new_x = logistic(x, r)
        if i >= (900):
            ax1.plot(r, x, 'ko', markersize = 3)
        x = new_x


ax1.set_xlim(2.5, 4)
ax1.set_title("Bifurcation diagram")
plt.show()
fig.savefig('bifurcation_diagram.png')
```



Wykres 2. Diagram bifurkacyjny dla r od 2.5 do 4 i $x_0 = 0.8$

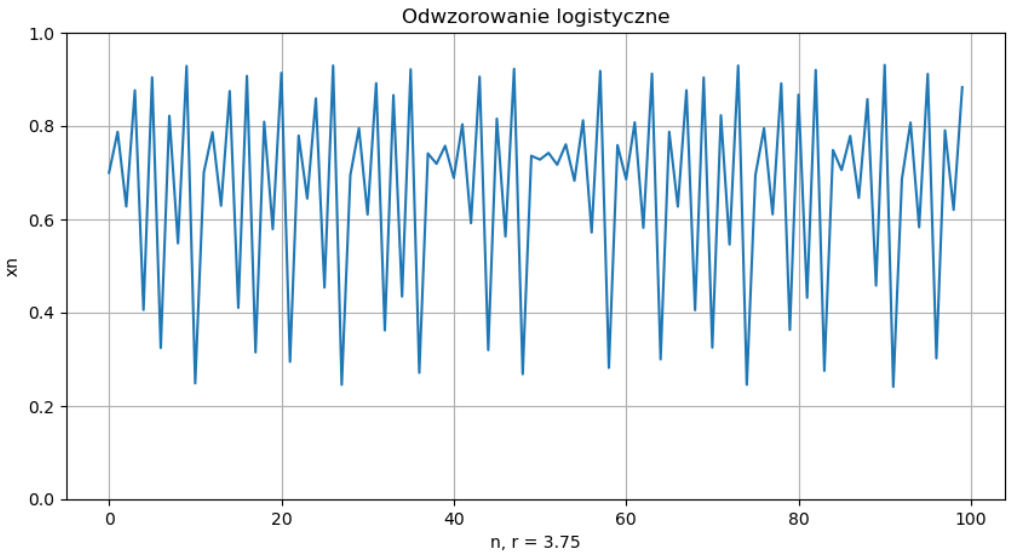
Oryginalna rozdzielczość : 1920 x 1080

Z diagramu bifurkacyjnego możemy zauważyć związek otrzymanego wykresu z wcześniej określonymi przedziałami.

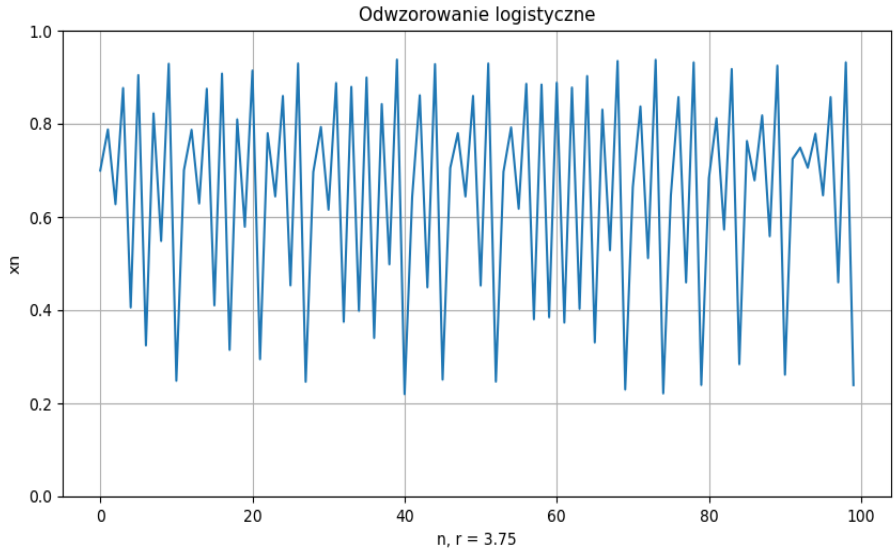
Trajektorie dla r od 3.75 do 3.8 i $x_0 = 0.8$ dla pojedynczej i podwójnej precyzji

```
import matplotlib.pyplot as plt
import numpy as np

r_tab = [3.75, 3.76, 3.77, 3.78, 3.79, 3.80]

def logistic(x, r):
    return np.float32(r * x * (1 - x))

def plot_for_offset_s(r, y_max):
    t = np.arange(0.0, 100, 1)
    s = np.empty(100, dtype=np.float32)
    x = np.float32(0.7)
    s[0] = x
    for j in range(99):
        new_x = logistic(x, np.float32(r))
        s[j+1] = new_x
        x = new_x

    fig, ax = plt.subplots(figsize=(10, 5))
    ax.plot(t, s)
```

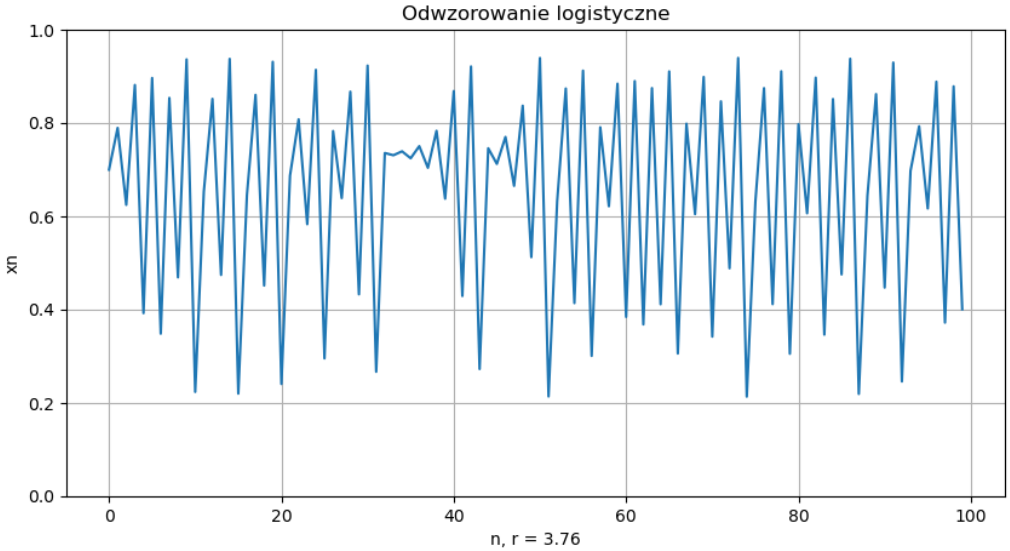
```
    ax.grid()
    ax.set(xlabel=f'n, r = {r}', ylabel='xn',
           title='Odwzorowanie logistyczne')
    ax.set_ylim(0, y_max)
    fig.savefig(f's_r={r}.png')

def plot_for_offset_d(r, y_max):
    t = np.arange(0.0, 100, 1)
    s = np.empty(100, dtype=np.float64)
    x = np.float64(0.7)
    s[0] = x
    for j in range(99):
        new_x = logistic(x, r)
        s[j+1] = new_x
        x = new_x

    fig, ax = plt.subplots(figsize=(10, 5))
    ax.plot(t, s)
    ax.grid()
    ax.set(xlabel=f'n, r = {r}', ylabel='xn',
           title='Odwzorowanie logistyczne')
    ax.set_ylim(0, y_max)
    fig.savefig(f'd_r={r}.png')

for r in r_tab:
    plot_for_offset_s(r, 1)
    plot_for_offset_d(r, 1)
```



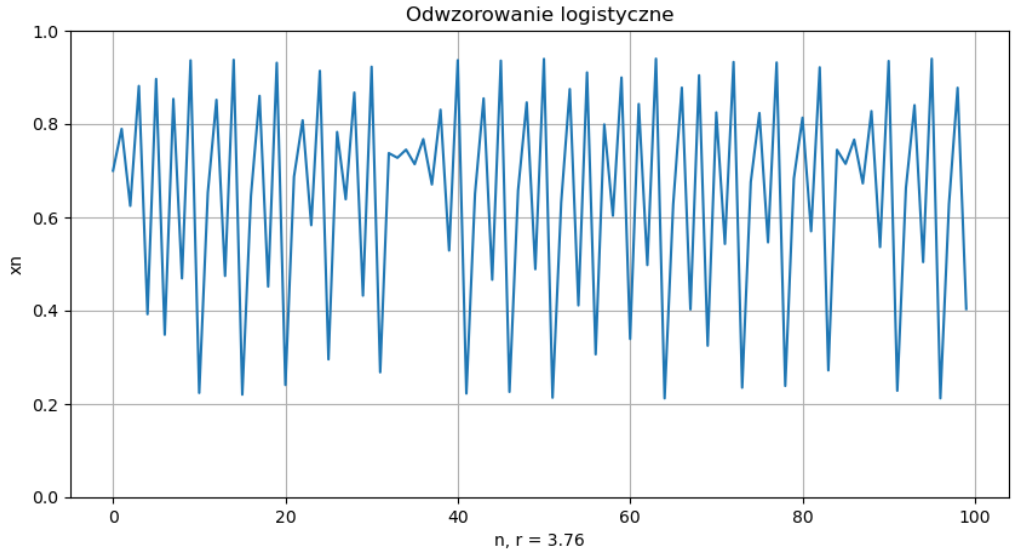
Wykres 3. Pojedyncza precyzja



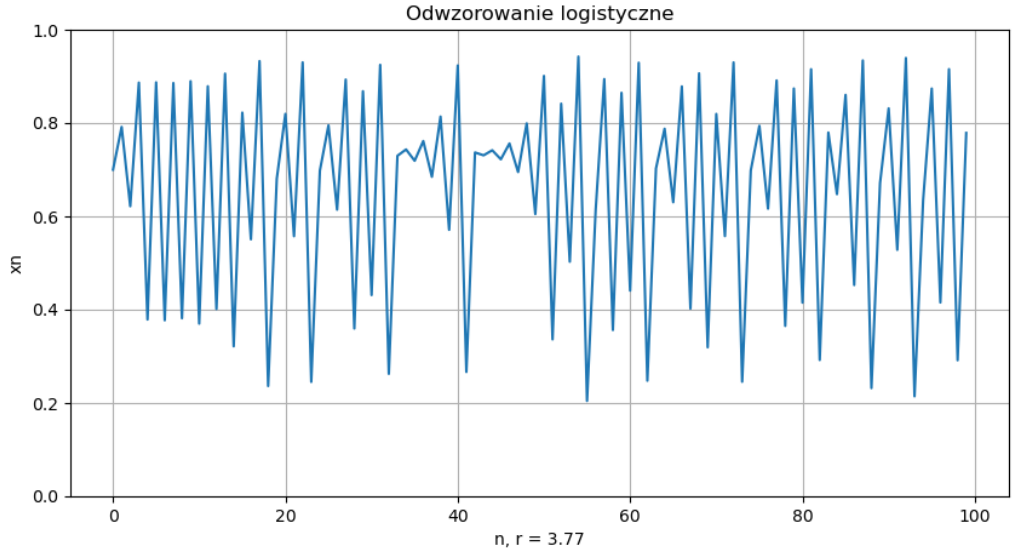
Wykres 4. Podwójna precyzja



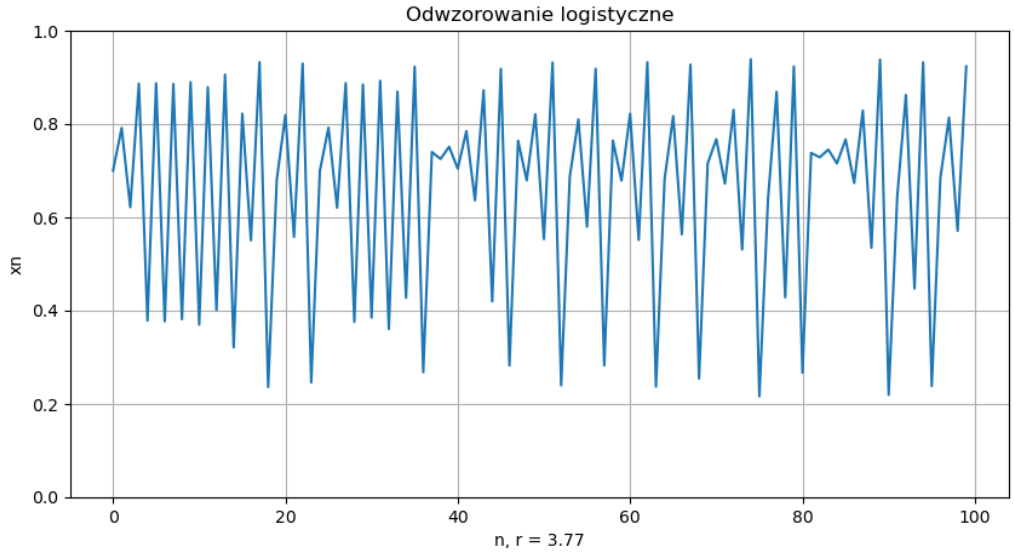
Wykres 5. Pojedyncza precyzja



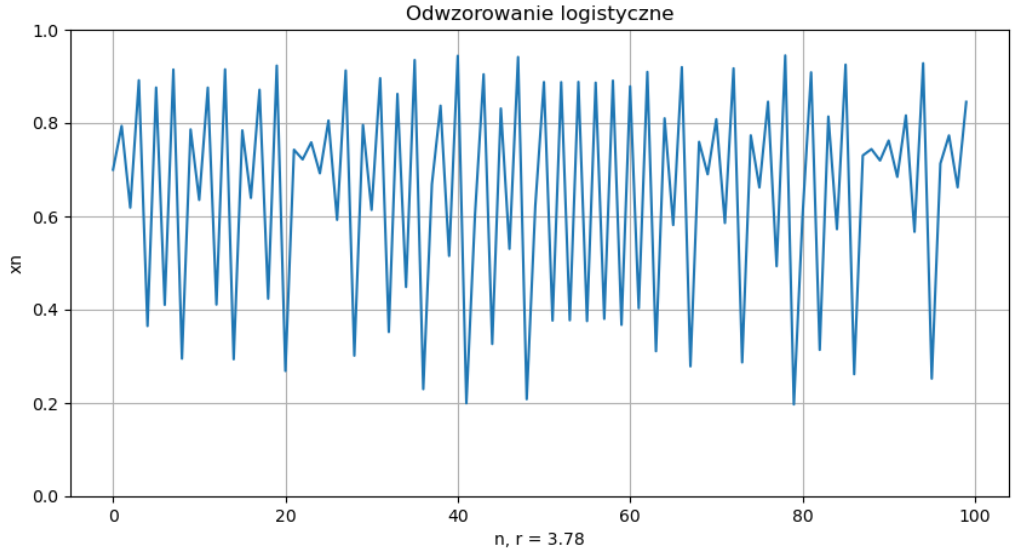
Wykres 6. Podwójna precyzja



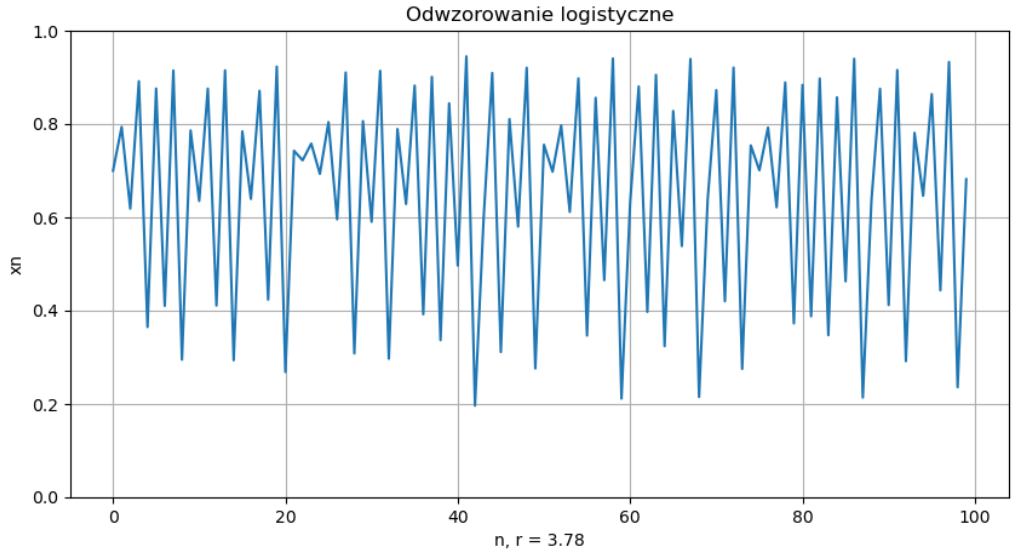
Wykres 7. Pojedyncza precyzja



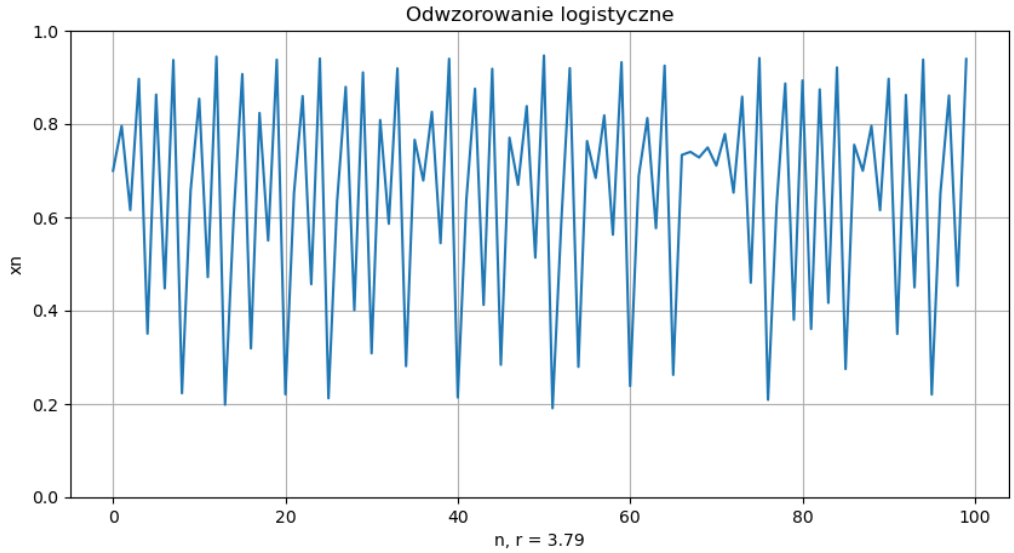
Wykres 8. Podwójna precyzja



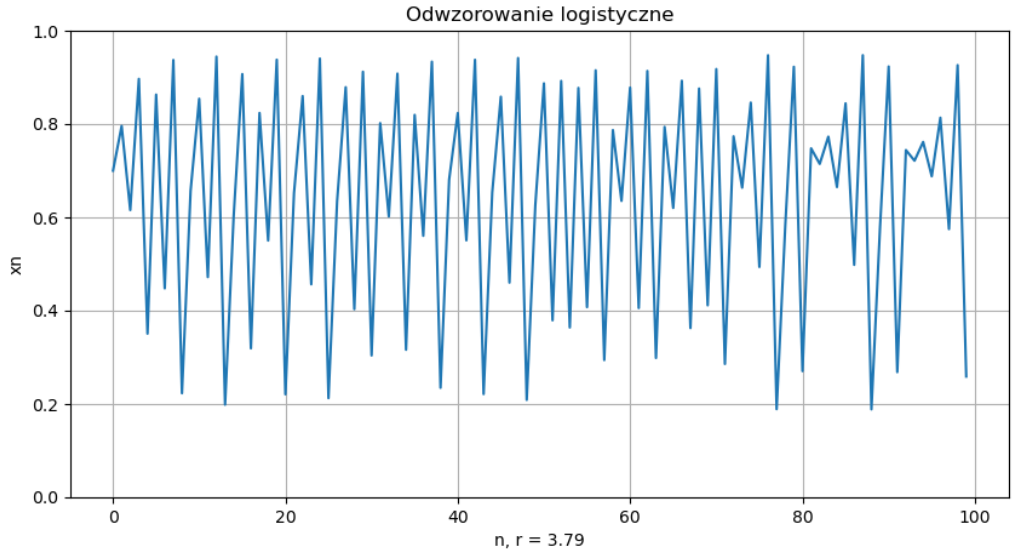
Wykres 9. Pojedyncza precyzja



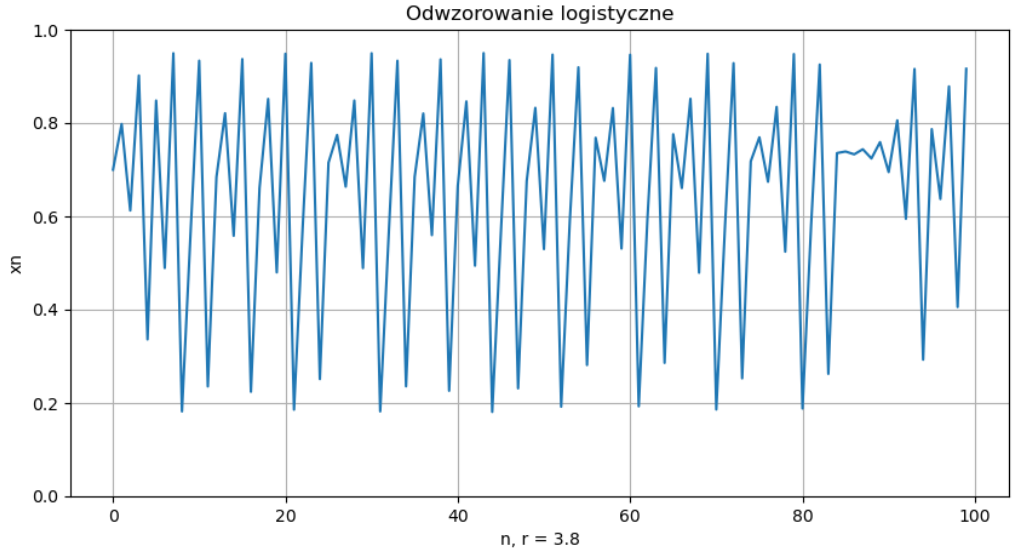
Wykres 10. Podwójna precyzja



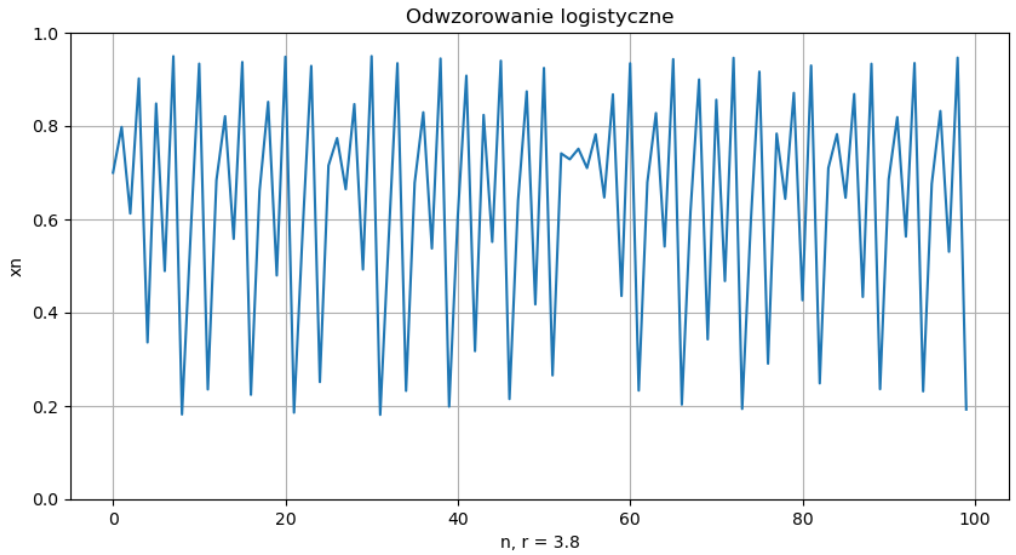
Wykres 11. Pojedyncza precyzja



Wykres 12. Podwójna precyzja



Wykres 13. Pojedyncza precyzja



Wykres 14. Podwójna precyzja

Widać więc, że zastosowanie podwójnej precyzji skutkuje otrzymaniem innej trajektorii w porównaniu z pojedynczą precyzją.

Liczba iteracji dla $x_0 = 0.6$, $x_0 = 0.7$, $x_0 = 0.8$, po której osiągane jest zero:

```
import numpy as np

x0 = np.float32(0.6)
x1 = np.float32(0.7)
x2 = np.float32(0.8)
r = 4
eps = np.float32(1.1922e-07)

def logistic(x, r):
    return r * x * (1 - x)

x = x0
counter = 0
while (x > eps):
    new_x = logistic(x, r)
    x = new_x
    counter = counter + 1

print(x0)
print(counter)
```

```
x = x1
counter = 0
while (x > eps):
    new_x = logistic(x, r)
    x = new_x
    counter = counter + 1

print(x1)
print(counter)

x = x2
counter = 0
while (x > eps):
    new_x = logistic(x, r)
    x = new_x
    counter = counter + 1

print(x2)
print(counter)
```

```
0.6
1227
0.7
18340
0.8
10063
```

Jako warunek przyjąłem, że wartość funkcji musi być mniejsza od liczby nieco większej od epsilona maszynowego dla np.float32. Przyjąłem eps jako 1.1922e-07.