Samuel Hełdak

Metody Obliczeniowe w Nauce i Technice

Laboratorium 1
Arytmetyka komputerowa

1. Sumowanie liczb pojedynczej precyzji

1.1. Suma N liczb iteracyjnie

Kod programu:

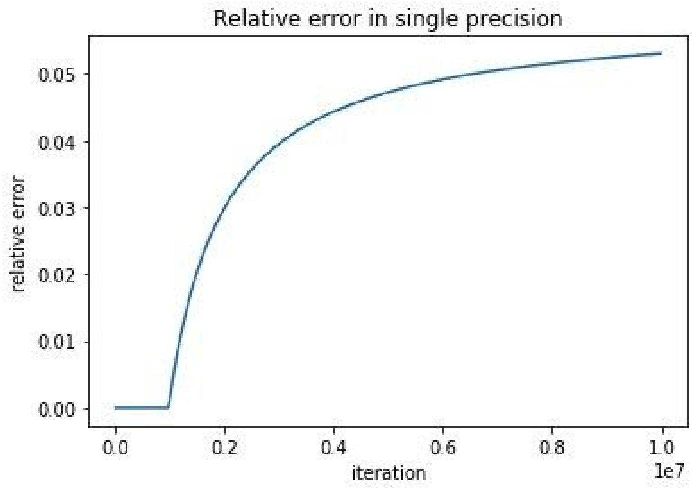
```
def sum_of_floats(arr, steps=False):
   acc = np.float32(0.0)

   if steps:
       arr_steps = []

       for i in range(len(arr)):
           if i % 25000 == 0 and i > 0:
               exact = np.float32(i) * arr[i]
               arr_steps.append(abs((acc - exact)/exact))
           acc += arr[i]

       return arr_steps
   else:
       for i in range(len(arr)):
           acc += arr[i]

       return acc
```

Wyniki dla N = 10^7:

```
one element:  0.236589
result:  2483399.0
accurate result:  2365890.0
absolute error  117509.0
relative error  0.049667988
```

```
one element:  0.53125
result:  5030840.5
accurate result:  5312500.0
absolute error  281659.5
relative error  0.053018257
```

Błędy względne są tak duże, ponieważ przy dodawaniu liczb zmiennoprzecinkowych o dużej różnicy cech wyniki nie są dokładne. Dzieje się tak ze względu na za małą liczbę bitów mantysy. Zatem od pewnego momentu (pokazanego na rys. 1) z każdą kolejną iteracją niedokładność obliczeń staje się coraz wyższa co prowadzi do dużego błędu względnego wyniku.



Rys. 1. Błąd względny w zależności od liczby iteracji algorytmu sumowania liczb pojedynczej precyzji

Na początku błąd względny jest równy 0, ponieważ różnica cech akumulatora (zmienna acc w programie, która przechowuje sumę elementów listy arr) i elementu listy jest na tyle niewielka, że po ich dodaniu wynik nadal jest dokładny w reprezentacji float32. Od pewnego momentu (okolice 10^6 iteracji) liczby są na tyle różne, że obliczenia stają się niedokładne w reprezentacji float32.

1.2. Suma N liczb rekurencyjnie

Kod programu:

```python
def sum_recursive(arr):
   return sum_rec(arr, 0, len(arr)-1)


def sum_rec(arr, i, j):
   if i == j:
       return np.float32(arr[i])
   elif j-i == 1:
       return np.float32(arr[i]) + np.float32(arr[j])

   return sum_rec(arr, i, (i+j)//2 - 1) + sum_rec(arr, (i+j)//2, j)
```

Wyniki dla N = 10^7 :

```
one element:  0.236589
result:  2365890.2
accurate result:  2365890.0
absolute error  0.25
relative error  1.0566848e-07
```

```
one element:  0.53125
result:  5312500.0
accurate result:  5312500.0
absolute error  0.0
relative error  0.0
```

W algorytmie rekurencyjnym zawsze dodawane są do siebie dwie liczby o bliskiej (często takiej samej) cesze co przekłada się na bardzo niski (lub nawet równy 0) błąd względny.

Czasy wykonania dla N = 10^7 oraz odpowiednio v = 0.236589 i v = 0.53125:

```
iterative:
time:  1.1073071956634521

recursive:
time:  2.8383805751800537
```

```
iterative:
time:  1.0839166641235352

recursive:
time:  2.883298635482788
```

Algorytm rekurencyjny jest wolniejszy, ze względu na odkładanie na stos kolejnych rekurencyjnych wywołań funkcji oraz wielokrotne sprawdzanie warunków stopu.

2. Algorytm Kahana

Kod programu:

```python
def kahan_sum(arr):
    result = np.float32(0.0)
    error = np.float32(0.0)

    for i in arr:
        y = np.float32(i) - error
        tmp = result + y
        error = (tmp - result) - y
        result = tmp

    return result
```

Wyniki dla N = 10^7 :

```
one element:  0.236589
result:  2365890.0
accurate result:  2365890.0
absolute error  0.0
relative error  0.0
```

```
one element:  0.53125
result:  5312500.0
accurate result:  5312500.0
absolute error  0.0
relative error  0.0
```

Możliwość otrzymania niedokładnego wyniku występuje gdy cecha sumy (zmienna *result* w programie wyżej) znacząco różni się od cechy dodawanej do sumy liczby (zmienna *i* będąca elementem tablicy). Zmienna *error* przechowuje różnicę między wartością jaka rzeczywiście została dodana do *result* (rezultat tej sumy został zaokrąglony by mieścił się w precyzji float32, więc może być niedokładny) a liczbą, która miała być dodana. Taka różnica często jest niezerowa przy dużych różnicach cech *result* oraz *i*. Następnie przy kolejnej iteracji liczba *i* zostaje skorygowana zgodnie z błędem (*error*) z poprzedniej iteracji. Takie operacje umożliwiają znaczną poprawę własności numerycznych.

Czasy wykonania algorytmów sumowania rekurencyjnego i Kahana dla N = 10^7 oraz odpowiednio v = 0.236589 i v = 0.53125:

```
recursive:
time:  2.810336112976074

Kahan algorithm:
time:  8.643990516662598
```

```
recursive:
time:  2.820634126663208

Kahan algorithm:
time:  7.558794260025024
```

Algorytm Kahana jest najwolniejszy z zaprezentowanych trzech algorytmów, ponieważ przy każdej iteracji jest wykonywanych więcej działań arytmetycznych.

3. Sumy częściowe

Kod programu:

```python
def riemann_zeta(s, n, float_func, reverse=False):
    result = float_func(0)

    if not reverse:
        for k in range(1, n+1):
            result += float_func(1 / (k ** s))
    else:
        for k in range(n, 0, -1):
            result += float_func(1 / (k ** s))

    return result


def dirichlet_eta(s, n, float_func, reverse=False):
    result = float_func(0)

    if not reverse:
        sign = 1
        for k in range(1, n+1):
            result += float_func(sign / (k ** s))
            sign *= -1
    else:
        sign = -1 if n % 2 == 0 else 1
        for k in range(n, 0, -1):
            result += float_func(sign / (k ** s))
            sign *= -1

    return result
```

Wyniki dla s ∈ {2, 3.6667, 5, 7.2, 10} oraz n ∈ {50, 100, 200, 500, 1000}:

```
sum forwards with single precision
s:  2.0   n:  50   zeta:  1.6251329   eta:  0.822271
s:  2.0   n:  100   zeta:  1.634984   eta:  0.8224175
s:  2.0   n:  200   zeta:  1.6399467   eta:  0.8224547
s:  2.0   n:  500   zeta:  1.642936   eta:  0.82246536
s:  2.0   n:  1000   zeta:  1.6439348   eta:  0.82246685
s:  3.6667   n:  50   zeta:  1.1093994   eta:  0.9346931
s:  3.6667   n:  100   zeta:  1.1094086   eta:  0.9346933
s:  3.6667   n:  200   zeta:  1.1094086   eta:  0.9346933
s:  3.6667   n:  500   zeta:  1.1094086   eta:  0.9346933
s:  3.6667   n:  1000   zeta:  1.1094086   eta:  0.9346933
s:  5.0   n:  50   zeta:  1.0369275   eta:  0.9721198
s:  5.0   n:  100   zeta:  1.0369275   eta:  0.9721198
s:  5.0   n:  200   zeta:  1.0369275   eta:  0.9721198
s:  5.0   n:  500   zeta:  1.0369275   eta:  0.9721198
s:  5.0   n:  1000   zeta:  1.0369275   eta:  0.9721198
s:  7.2   n:  50   zeta:  1.0072277   eta:  0.99352705
s:  7.2   n:  100   zeta:  1.0072277   eta:  0.99352705
s:  7.2   n:  200   zeta:  1.0072277   eta:  0.99352705
s:  7.2   n:  500   zeta:  1.0072277   eta:  0.99352705
s:  7.2   n:  1000   zeta:  1.0072277   eta:  0.99352705
s:  10.0   n:  50   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  100   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  200   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  500   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  1000   zeta:  1.0009946   eta:  0.99903953
```

```
sum forwards with double precision
s:  2.0   n:  50   zeta:  1.625132733621529   eta:  0.8222710318260295
s:  2.0   n:  100   zeta:  1.6349839001848923   eta:  0.8224175333741286
s:  2.0   n:  200   zeta:  1.6399465460149971   eta:  0.822454595922551
s:  2.0   n:  500   zeta:  1.642936065514894   eta:  0.8224650374240963
s:  2.0   n:  1000   zeta:  1.6439345666815615   eta:  0.8224665339241114
s:  3.6667   n:  50   zeta:  1.1093997551541945   eta:  0.9346930600307106
s:  3.6667   n:  100   zeta:  1.1094087973421474   eta:  0.9346933211400662
s:  3.6667   n:  200   zeta:  1.1094102423332313   eta:  0.9346933421086845
s:  3.6667   n:  500   zeta:  1.1094104908440712   eta:  0.9346933438558745
s:  3.6667   n:  1000   zeta:  1.1094105108423578   eta:  0.9346933439141353
s:  5.0   n:  50   zeta:  1.036927716716712   eta:  0.9721197689267979
s:  5.0   n:  100   zeta:  1.0369277526929555   eta:  0.9721197703981592
s:  5.0   n:  200   zeta:  1.0369277549886775   eta:  0.972119770445367
s:  5.0   n:  500   zeta:  1.0369277551393863   eta:  0.9721197704468947
s:  5.0   n:  1000   zeta:  1.0369277551431222   eta:  0.9721197704469091
s:  7.2   n:  50   zeta:  1.0072276664762816   eta:  0.9935270006613486
s:  7.2   n:  100   zeta:  1.007227666480654   eta:  0.9935270006616185
s:  7.2   n:  200   zeta:  1.0072276664807145   eta:  0.9935270006616201
s:  7.2   n:  500   zeta:  1.0072276664807145   eta:  0.9935270006616201
s:  7.2   n:  1000   zeta:  1.0072276664807145   eta:  0.9935270006616201
s:  10.0   n:  50   zeta:  1.0009945751278182   eta:  0.9990395075982718
s:  10.0   n:  100   zeta:  1.0009945751278182   eta:  0.9990395075982718
s:  10.0   n:  200   zeta:  1.0009945751278182   eta:  0.9990395075982718
s:  10.0   n:  500   zeta:  1.0009945751278182   eta:  0.9990395075982718
s:  10.0   n:  1000   zeta:  1.0009945751278182   eta:  0.9990395075982718
```

```
sum backwards with single precision
s:  2.0   n:  50   zeta:  1.6251327   eta:  0.82227105
s:  2.0   n:  100   zeta:  1.6349839   eta:  0.8224175
s:  2.0   n:  200   zeta:  1.6399465   eta:  0.8224546
s:  2.0   n:  500   zeta:  1.642936   eta:  0.82246506
s:  2.0   n:  1000   zeta:  1.6439345   eta:  0.82246655
s:  3.6667   n:  50   zeta:  1.1093998   eta:  0.93469304
s:  3.6667   n:  100   zeta:  1.1094089   eta:  0.93469334
s:  3.6667   n:  200   zeta:  1.1094103   eta:  0.93469334
s:  3.6667   n:  500   zeta:  1.1094105   eta:  0.93469334
s:  3.6667   n:  1000   zeta:  1.1094105   eta:  0.93469334
s:  5.0   n:  50   zeta:  1.0369277   eta:  0.97211975
s:  5.0   n:  100   zeta:  1.0369277   eta:  0.97211975
s:  5.0   n:  200   zeta:  1.0369277   eta:  0.97211975
s:  5.0   n:  500   zeta:  1.0369277   eta:  0.97211975
s:  5.0   n:  1000   zeta:  1.0369277   eta:  0.97211975
s:  7.2   n:  50   zeta:  1.0072277   eta:  0.993527
s:  7.2   n:  100   zeta:  1.0072277   eta:  0.993527
s:  7.2   n:  200   zeta:  1.0072277   eta:  0.993527
s:  7.2   n:  500   zeta:  1.0072277   eta:  0.993527
s:  7.2   n:  1000   zeta:  1.0072277   eta:  0.993527
s:  10.0   n:  50   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  100   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  200   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  500   zeta:  1.0009946   eta:  0.99903953
s:  10.0   n:  1000   zeta:  1.0009946   eta:  0.99903953
```

```
sum backwards with double precision
s:  2.0   n:  50   zeta:  1.6251327336215293   eta:  0.8222710318260289
s:  2.0   n:  100   zeta:  1.634983900184893   eta:  0.8224175333741282
s:  2.0   n:  200   zeta:  1.6399465460149973   eta:  0.8224545959225509
s:  2.0   n:  500   zeta:  1.6429360655148941   eta:  0.8224650374240972
s:  2.0   n:  1000   zeta:  1.6439345666815597   eta:  0.8224665339241127
s:  3.6667   n:  50   zeta:  1.1093997551541943   eta:  0.934693060030711
s:  3.6667   n:  100   zeta:  1.1094087973421476   eta:  0.934693321140067
s:  3.6667   n:  200   zeta:  1.109410242333231   eta:  0.9346933421086852
s:  3.6667   n:  500   zeta:  1.1094104908440725   eta:  0.934693343855875
s:  3.6667   n:  1000   zeta:  1.1094105108423593   eta:  0.9346933439141354
s:  5.0   n:  50   zeta:  1.0369277167167108   eta:  0.9721197689267976
s:  5.0   n:  100   zeta:  1.0369277526929532   eta:  0.9721197703981589
s:  5.0   n:  200   zeta:  1.036927754988676   eta:  0.9721197704453663
s:  5.0   n:  500   zeta:  1.0369277551393858   eta:  0.9721197704468933
s:  5.0   n:  1000   zeta:  1.0369277551431204   eta:  0.9721197704469088
s:  7.2   n:  50   zeta:  1.0072276664762823   eta:  0.9935270006613481
s:  7.2   n:  100   zeta:  1.007227666480655   eta:  0.9935270006616179
s:  7.2   n:  200   zeta:  1.0072276664807163   eta:  0.9935270006616198
s:  7.2   n:  500   zeta:  1.0072276664807172   eta:  0.9935270006616198
s:  7.2   n:  1000   zeta:  1.0072276664807172   eta:  0.9935270006616198
s:  10.0   n:  50   zeta:  1.000994575127818   eta:  0.9990395075982715
s:  10.0   n:  100   zeta:  1.000994575127818   eta:  0.9990395075982715
s:  10.0   n:  200   zeta:  1.000994575127818   eta:  0.9990395075982715
s:  10.0   n:  500   zeta:  1.000994575127818   eta:  0.9990395075982715
s:  10.0   n:  1000   zeta:  1.000994575127818   eta:  0.9990395075982715
```

Dokładne wartości funkcji (źródło: keisan.casio.com):

$\zeta(2) = 1.6449340668482264364 72$ $\eta(2) = 0.822467033424113218236 2$

$\zeta(3.6667) = 1.10941051458645335745 1$ $\eta(3.6667) = 0.9346933439191250729261$

$\zeta(5) = 1.0369277551433699263 31$ $\eta(5) = 0.97211977044690930593 57$

$\zeta(7.2) = 1.00722766480717114739$ $\eta(7.2) = 0.99352700066161978757 45$

$\zeta(10) = 1.00099457512781808533 7$ $\eta(10) = 0.99903950759827156563 92$

Dla s >= 5 w pojedynczej precyzji nie było różnicy między wartościami funkcji zeta i eta dla n = 50, 100, 200, 500, 1000. Natomiast dla wartości s = 3.6667 w pojedynczej precyzji wartości funkcji zeta dla n >= 100 już się nie różniły gdy sumowanie odbywało się w przód, a różniły się dla n = 100, 200, 500 gdy sumowano wstecz oraz wartość obliczona wstecz (dla n = 500, 1000) jest bliższa wynikowi dokładnemu. To sugeruje, że funkcja licząca wstecz była w tym przypadku dokładniejsza (była lepsza wraz ze wzrastającym n). Dla podwójnej precyzji tylko przy s = 10 nie miała znaczenia liczba n (dla n >= 50) i również dla liczenia w przód szybciej następowała stabilizacja wartości funkcji zeta niż dla liczenia wstecz (tym razem dla s = 7.2). Dla obydwu funkcji w obydwu precyzjach liczenie wstecz było dokładniejsze od liczenia w przód dla s >= 7.2 (sprawdzone dla n = 1000, ponieważ taki wynik jest najdokładniejszy). Na podstawie powyższych wyników liczenie wstecz wydaje się być dokładniejsze. Również można zaobserwować, że dla większej liczby s, wyniki są bliższe dokładnym wartościom.

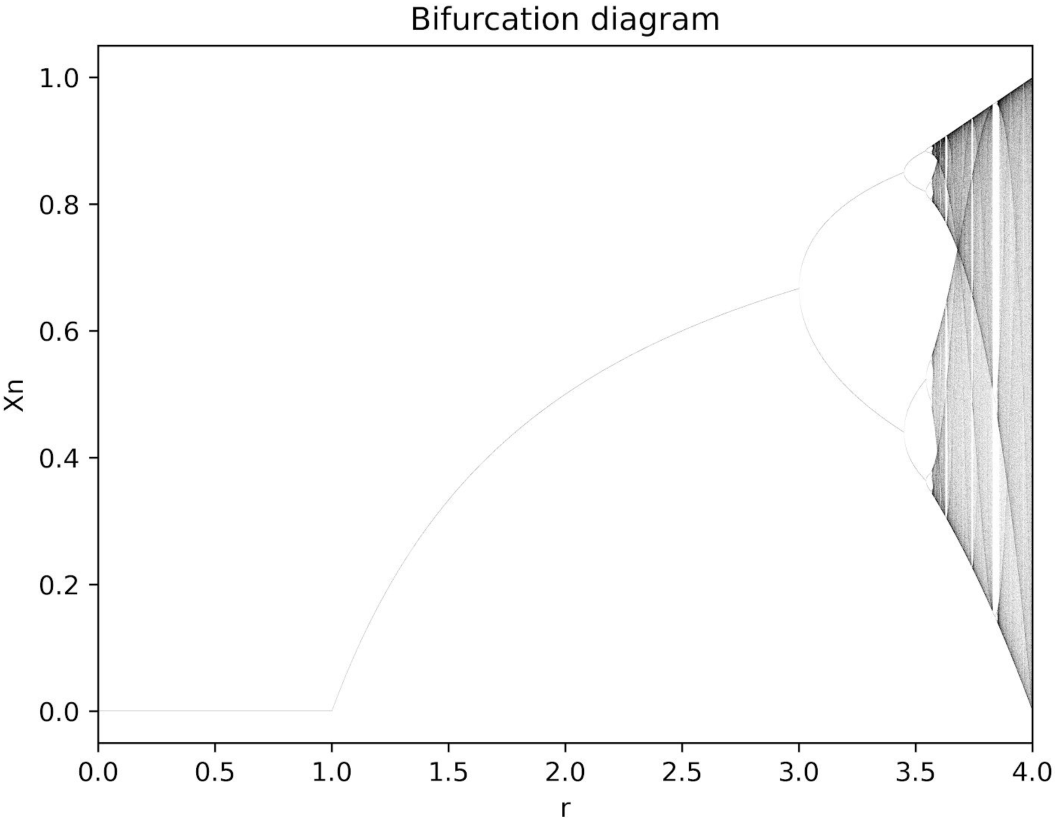
4. Błędy zaokrągleń i odwzorowanie logistyczne

4.1. Diagramy bifurkacyjne

Kod programu:

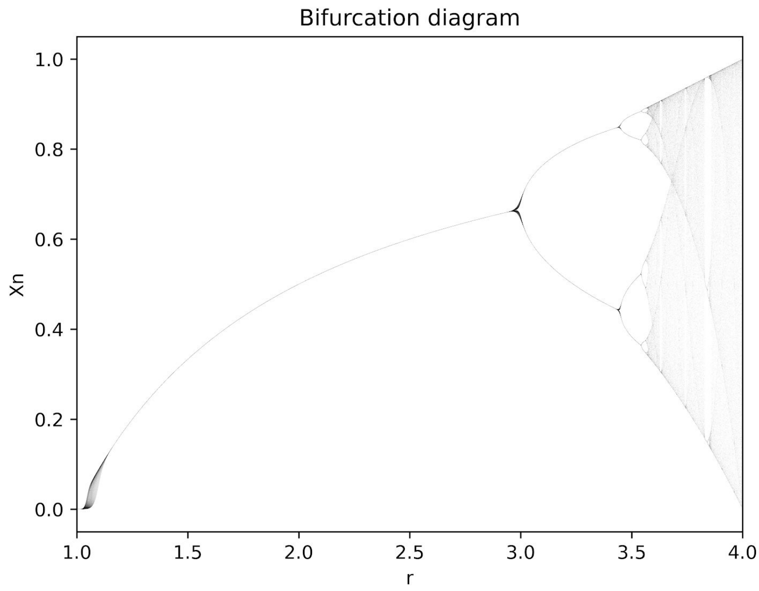
```
def draw_logistic_map(n, r_range, iterations, iterations_to_draw, x):
   r = np.linspace(r_range[0], r_range[1], n)

   fig, ax = plt.subplots(1, 1)

   for i in range(iterations):
       x = logistic(r, x)
       if i >= (iterations - iterations_to_draw):
           ax.plot(r, x, ',k', alpha=.25)

   ax.set_xlim(r_range[0], r_range[1])
   ax.set_title("Bifurcation diagram")
   plt.xlabel("r")
   plt.ylabel("Xn")
   plt.savefig('bifurcation_diagram.png', dpi=800)
```

Diagram bifurkacyjny przedstawiający wartości x_n dla dziesięciu tysięcy *r* z przedziału [0, 4] z zaznaczonymi punktami dla n ∈ {9000, 9001, …, 9999}, $x_0 = 10^{-5}$:



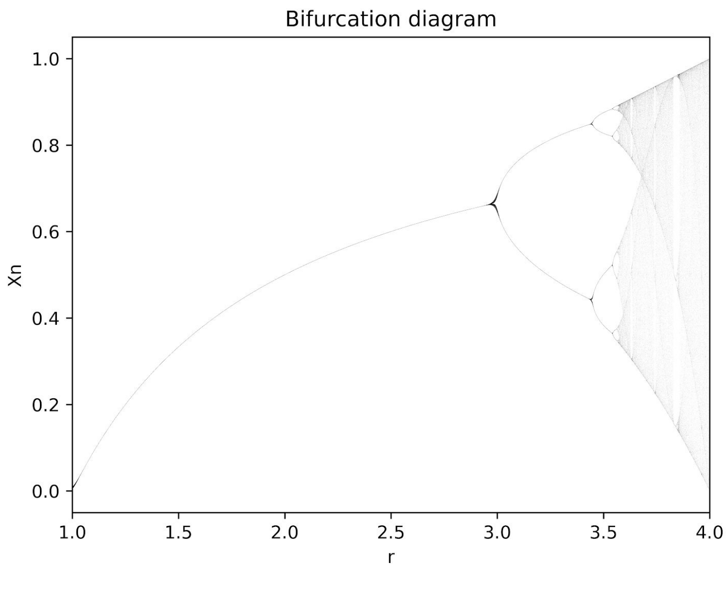
Rys. 2. Diagram bifurkacyjny dla wysokich wartości n; $x_0 = 10^{-5}$

Diagram bifurkacyjny przedstawiający wartości x_n dla dziesięciu tysięcy *r* z przedziału [1, 4] z zaznaczonymi punktami dla n ∈ {100, 101, …, 199}, $x_0 = 10^{-5}$:



Rys. 3. Diagram bifurkacyjny dla niższych n, $x_0 = 10^{-5}$

Diagram bifurkacyjny przedstawiający wartości x_n dla dziesięciu tysięcy *r* z przedziału [1, 4] z zaznaczonymi punktami dla n ∈ {100, 101, …, 199}, $x_0 = 0.5$:



Rys. 4. Diagram bifurkacyjny dla niższych n, $x_0 = 0.5$

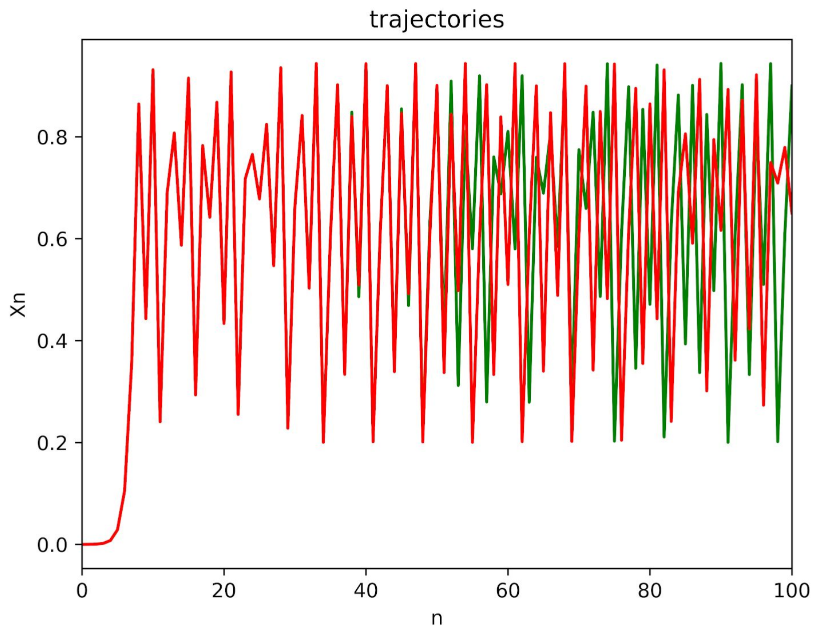
Zaznaczając wiele ostatnich iteracji diagram nie różni się znacząco przy różnych x_0. Najbardziej znacząca różnica jest widoczna w okolicach r = 1, wtedy funkcja logistyczna potrzebuje większej liczby iteracji by ustabilizować się w oczekiwanym położeniu.

4.2. Trajektorie w odwzorowaniu logistycznym

Kod programu:

```
def get_results(x, r, n, float_func):
   x = float_func(x)
   r = float_func(r)
   x_axis = [float_func(i) for i in range(n + 1)]
   y_axis = []

   for i in range(n + 1):
       x = logistic(r, x)
       y_axis.append(x)

   return x_axis, y_axis


def draw_trajectories(x, r, n):
   fig, ax = plt.subplots(1, 1)

   x_axis, y_axis = get_results(x, r, n, np.float32)
   plt.plot(x_axis, y_axis, c="green")

   x_axis, y_axis = get_results(x, r, n, np.float64)
   plt.plot(x_axis, y_axis, c="red")

   ax.set_xlim(0, n)
   ax.set_title("trajectories")
   plt.xlabel("n")
   plt.ylabel("Xn")
   plt.savefig('trajectories.png', dpi=800)
```

Trajektorie dla $x_0 = 0.00001$, r = 3.775, n = 100 dla pojedynczej (na zielono) i podwójnej (na czerwono) precyzji (na początku wartości są identyczne, więc czerwona trajektoria przysłania zieloną):



Rys. 5. Trajektorie funkcji logistycznej dla pojedynczej i podwójnej precyzji, r = 3.775, n = 100, $x_0 = 0.00001$

Różnica między trajektoriami wynika z błędów jakie się gromadzą przy wielokrotnym mnożeniu liczb zmiennoprzecinkowych.

4.3. Osiąganie 0 dla r = 4

Poniżej przedstawiona jest liczba iteracji potrzebnych do osiągnięcia 0 dla wszystkich $x_0 \in$ {0.01, 0.02, …, 0.99}, dla których udawało się znaleźć taką liczbę w czasie co najwyżej kilku sekund dla pojedynczej precyzji obliczeń. Funkcje mające wartość x_0 , która nie pojawia się poniżej najprawdopodobniej nigdy nie osiągają zera (funkcja staje się w pewnym momencie okresowa).

0.01 : 2960	0.02 : 984	0.03 : 832	0.04 : 269	0.05 : 3543	0.06 : 3416	0.07 : 1323	0.08 : 885		
0.11 : 559	0.12 : 620	0.13 : 3094	0.14 : 1954	0.15 : 3253	0.16 : 3009	0.17 : 1102	0.18 : 737	0.19 : 3542	
0.21 : 3484	0.22 : 3550	0.23 : 113	0.24 : 2647		0.26 : 1900		0.28 : 2986	0.29 : 2361	0.3 : 1100
0.31 : 2406	0.32 : 2146	0.33 : 2538	0.34 : 2016	0.35 : 1095		0.37 : 638	0.38 : 3200	0.39 : 3458	0.4 : 2406
0.41 : 1377	0.42 : 767	0.43 : 2494	0.44 : 333	0.45 : 804	0.46 : 2629		0.48 : 763	0.49 : 3252	0.5 : 2
0.51 : 3252	0.52 : 1349		0.54 : 2629	0.55 : 804	0.56 : 333	0.57 : 2494	0.58 : 767	0.59 : 3856	0.6 : 2406
0.61 : 3458	0.62 : 3200	0.63 : 638		0.65 : 862	0.66 : 2016	0.67 : 2009	0.68 : 2146	0.69 : 2406	0.7 : 1100
0.71 : 3852	0.72 : 2051		0.74 : 1900		0.76 : 2647	0.77 : 2963	0.78 : 3550	0.79 : 3484	
0.81 : 3542	0.82 : 737	0.83 : 1102	0.84 : 3009	0.85 : 3253	0.86 : 1954	0.87 : 3094	0.88 : 620	0.89 : 3145	0.9 : 1928
0.91 : 861	0.92 : 3379	0.93 : 2672	0.94 : 3416	0.95 : 1481	0.96 : 2405		0.98 : 2645	0.99 : 803	

Liczba iteracji potrzebnych do osiągnięcia 0 różni się znacząco w zależności od liczby x_0 . Wynika to z dużej rozbieżności liczb x_n w zależności od n dla r = 4 co można dostrzec na rys. 2 (funkcja zdaje się przyjmować każdą wartość z przedziału [0,1]). Również dla r=4 jest największy przedział wartości jakie funkcja logistyczna może przyjmować.