Metody obliczeniowe w nauce i technice 2

Tomasz Zawadzki

13 marca 2019

Zadanie 1. Sumowanie liczb pojedynczej precyzji

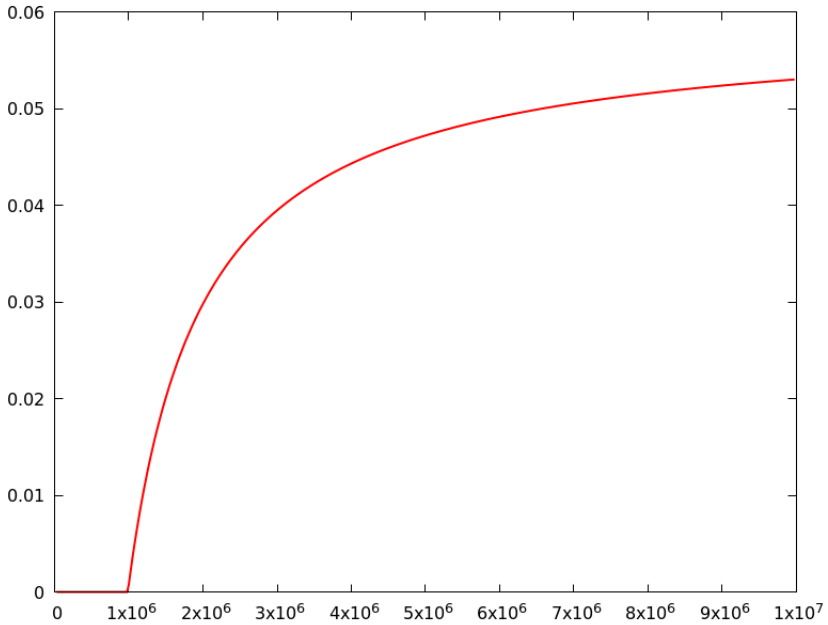
Tablica liczb zmiennoprzecinkowych pojedynczej precyzji (`float`) o rozmiarze 10^7 jest wypełniona tą samą wartością $x = 0.53125$. Suma wszystkich elementów tablicy została obliczona na kilka sposobów.

Prosty algorytm iteracyjny

```
float sum_simple(float t[], int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; i++) sum += t[i];
    return sum;
}
```

```
simple sum: 5030840.500000
absolute error: 281659.500000
relative error: 0.055987 = 5.598657%
0.046875 s
```

Błąd względny wynika z charakterystyki dodawania liczb zmiennoprzecinkowych, które znacznie różnią się wykładnikiem. W kolejnych iteracjach dotychczasowa suma jest znacznie większa od dodawanej wartości, co powoduje utratę precyzji.



Rys. 1. Zależność błędu względnego od liczby przeprowadzonych iteracji

Algorytm rekurencyjny

```
float sum_recursive_lr(float t[], int l, int r) {
    if (l > r) return 0.0f;
    if (l == r) return t[l];
    return sum_recursive_lr(t, l, l+(r-l)/2) + sum_recursive_lr(t, l+(r-l)/2+1, r);
}
float sum_recursive(float t[], int n) {
    return sum_recursive_lr(t, 0, n-1);
}
```

```
recursive sum: 5312500.000000
absolute error: 0.000000
relative error: 0.000000 = 0.000000%
0.109375 s
```

Ponad dwukrotnie dłuższy czas wykonania, ale błąd bezwzględny wynosi zero. Tablica jest wypełniona tą samą wartością, zatem algorytm rekurencyjny zawsze sumuje liczby podobnego rzędu, co przekłada się na większą dokładność uzyskanego wyniku niż w przypadku prostego sumowania iteracyjnego.

Zadanie 2. Algorytm Kahana

```
float sum_kahana(float tab[], int n) {
    float sum = 0.0f;
    float err = 0.0f;
    for (int i = 0; i < n; ++i) {
        float y = tab[i] - err;
        float temp = sum + y;
        err = (temp - sum) - y;
        sum = temp;
    }
    return sum;
}
```

```
Kahan sum: 5312500.000000
absolute error: 0.000000
relative error: 0.000000 = 0.000000%
0.125000 s
```

Algorytm Kahana ma znacznie lepsze własności numeryczne, ponieważ uwzględnia również ostatnie bity liczby zmiennoprzecinkowej w obliczanej sumie. Do akumulowania oraz uwzględniania tych małych błędów wynikających z niedoskonałości arytmetyki zmiennoprzecinkowej służy zmienna `err`. Dla tych samych wejściowych czas wykonania programu z zaimplementowanym algorytmem Kahana jest porównywalny z czasem wykonania programu, w którym zastosowano algorytm sumowania rekurencyjnego.

Zadanie 3. Sumy częściowe

Sumy częściowe szeregu definiującego funkcję dzeta Riemanna

$$\zeta(s) = \sum_{k=0}^{n} \frac{1}{k^s}$$

w punktach $x = 2, 3.6667, 5, 7.2, 10$ zostały obliczone czterema sposobami – w pojedynczej i podwójnej precyzji oraz sumując w przód, a następnie wstecz.

s	n	float forward	float backward	double forward	double backward
2.00	50	1.625132918357849	1.625132799148560	1.625132733621529	1.625132733621529
2.00	100	1.634984016418457	1.634983897209167	1.634983900184892	1.634983900184893
2.00	200	1.639946699142456	1.639946460723877	1.639946546014997	1.639946546014997
2.00	500	1.642935991287231	1.642935991287231	1.642936065514894	1.642936065514894
2.00	1000	1.643934845924377	1.643934488296509	1.643934566681561	1.643934566681560
3.67	50	1.109399437904358	1.109399795532227	1.109399755154195	1.109399755154194
3.67	100	1.109408617019653	1.109408855438232	1.109408797342147	1.109408797342148
3.67	200	1.109408617019653	1.109410285949707	1.109410242333231	1.109410242333231
3.67	500	1.109408617019653	1.109410524368286	1.109410490844071	1.109410490844073
3.67	1000	1.109408617019653	1.109410524368286	1.109410510842358	1.109410510842359
5.00	50	1.036927461624146	1.036927700042725	1.036927716716712	1.036927716716711
5.00	100	1.036927461624146	1.036927700042725	1.036927752692955	1.036927752692953
5.00	200	1.036927461624146	1.036927700042725	1.036927754988678	1.036927754988676
5.00	500	1.036927461624146	1.036927700042725	1.036927755139386	1.036927755139386
5.00	1000	1.036927461624146	1.036927700042725	1.036927755143122	1.036927755143120
7.20	50	1.007227659225464	1.007227659225464	1.007227666476282	1.007227666476282
7.20	100	1.007227659225464	1.007227659225464	1.007227666480654	1.007227666480655
7.20	200	1.007227659225464	1.007227659225464	1.007227666480714	1.007227666480716
7.20	500	1.007227659225464	1.007227659225464	1.007227666480714	1.007227666480717
7.20	1000	1.007227659225464	1.007227659225464	1.007227666480714	1.007227666480717
10.00	50	1.000994563102722	1.000994563102722	1.000994575127818	1.000994575127818
10.00	100	1.000994563102722	1.000994563102722	1.000994575127818	1.000994575127818
10.00	200	1.000994563102722	1.000994563102722	1.000994575127818	1.000994575127818
10.00	500	1.000994563102722	1.000994563102722	1.000994575127818	1.000994575127818
10.00	1000	1.000994563102722	1.000994563102722	1.000994575127818	1.000994575127818

W pojedynczej precyzji wyniki otrzymane metodą sumowania w przód zauważalnie różnią się od tych obliczonych metodą sumowania wstecz. W podwójnej precyzji efekt ten jest również widoczny, ale różnica pomiędzy wartościami jest znacznie mniejsza niż w przypadku precyzji pojedynczej (liczby są równe z dokładnością do 10^{-16}).

Metoda sumowania w przód polega na dodawaniu kolejnych wyrazów ciągu (o rosnących indeksach) do dotychczasowej sumy, która jest znacznie większa niż dodawane wartości, co jest powodem utraty precyzji. Z kolei metoda sumowania tego szeregu wstecz w każdej iteracji do dotychczasowej sumy dodaje większą wartość niż w przypadku sumowania w przód, więc wynik jest obarczony znacznie mniejszym błędem.

Dla mniejszych wartości s wyniki zdecydowanie zależą od liczby iteracji, natomiast dla większych s wartość uzyskana dla $n = 50$ iteracji jest identyczna jak dla $n = 1000$ iteracji. Wynika to z własności szeregu definiującego tę funkcję.

W tych samych punktach, również czterema sposobami, zostały obliczone sumy częściowe szeregu definiującego funkcję eta Dirichleta

$$\eta(s) = \sum_{k=0}^{n} (-1)^{k-1} \frac{1}{k^s}$$

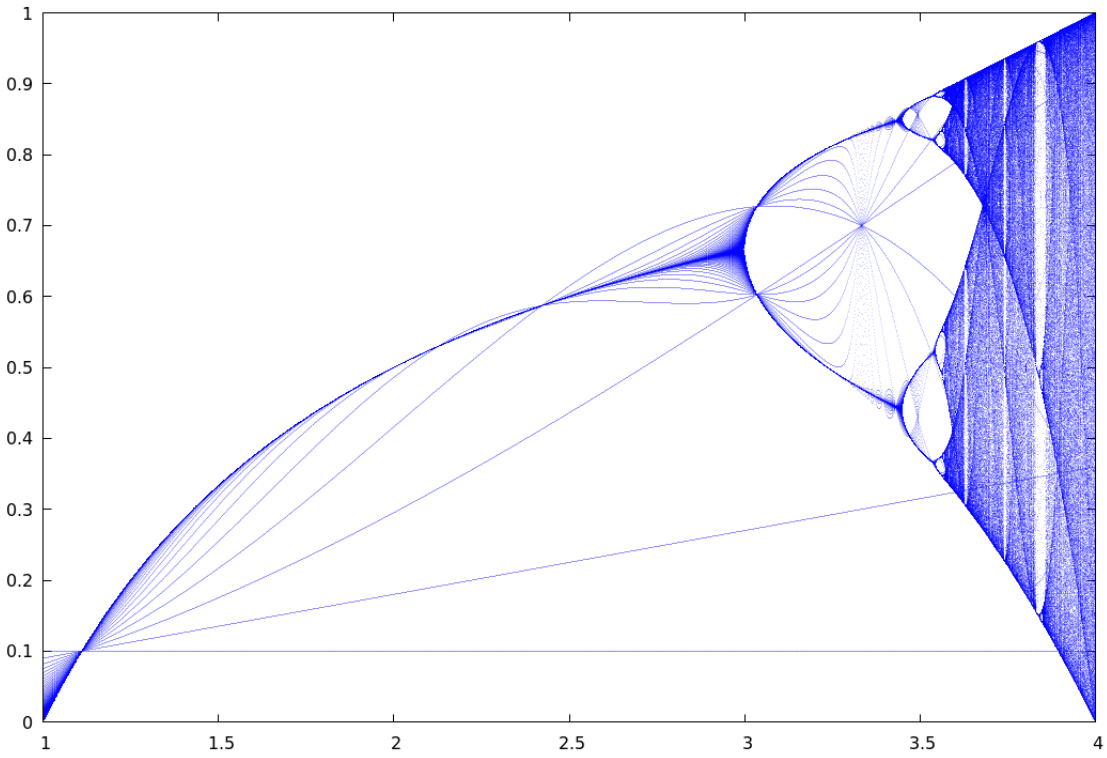
s	n	float forward	float backward	double forward	double backward
2.00	50	0.8222709894180298	0.8222710490226746	0.8222710318260295	0.8222710318260289
2.00	100	0.8224174976348877	0.8224174976348877	0.8224175333741286	0.8224175333741282
2.00	200	0.8224546909332275	0.8224545717239380	0.8224545959225510	0.8224545959225509
2.00	500	0.8224653601646423	0.8224650621414185	0.8224650374240963	0.8224650374240972
2.00	1000	0.8224668502807617	0.8224664926528931	0.8224665339241114	0.8224665339241127
3.67	50	0.9346930384635925	0.9346930384635925	0.9346930600307106	0.9346930600307110
3.67	100	0.9346932172775269	0.9346933364868164	0.9346933211400662	0.9346933211400670
3.67	200	0.9346932172775269	0.9346933364868164	0.9346933421086845	0.9346933421086852
3.67	500	0.9346932172775269	0.9346933364868164	0.9346933438558745	0.9346933438558750
3.67	1000	0.9346932172775269	0.9346933364868164	0.9346933439141353	0.9346933439141354
5.00	50	0.9721198081970215	0.9721197485923767	0.9721197689267979	0.9721197689267976
5.00	100	0.9721198081970215	0.9721197485923767	0.9721197703981592	0.9721197703981589
5.00	200	0.9721198081970215	0.9721197485923767	0.9721197704453670	0.9721197704453663
5.00	500	0.9721198081970215	0.9721197485923767	0.9721197704468947	0.9721197704468933
5.00	1000	0.9721198081970215	0.9721197485923767	0.9721197704469091	0.9721197704469088
7.20	50	0.9935270547866821	0.9935269951820374	0.9935270006613486	0.9935270006613481
7.20	100	0.9935270547866821	0.9935269951820374	0.9935270006616185	0.9935270006616179
7.20	200	0.9935270547866821	0.9935269951820374	0.9935270006616201	0.9935270006616198
7.20	500	0.9935270547866821	0.9935269951820374	0.9935270006616201	0.9935270006616198
7.20	1000	0.9935270547866821	0.9935269951820374	0.9935270006616201	0.9935270006616198
10.00	50	0.9990395307540894	0.9990395307540894	0.9990395075982718	0.9990395075982715
10.00	100	0.9990395307540894	0.9990395307540894	0.9990395075982718	0.9990395075982715
10.00	200	0.9990395307540894	0.9990395307540894	0.9990395075982718	0.9990395075982715
10.00	500	0.9990395307540894	0.9990395307540894	0.9990395075982718	0.9990395075982715
10.00	1000	0.9990395307540894	0.9990395307540894	0.9990395075982718	0.9990395075982715

Powyższe wnioski mają znajdują również zastosowanie w tym przypadku. Należy jednak uwzględnić fakt, że szereg definiujący tę funkcję jest naprzemienny (kolejne wyrazy mają przeciwne znaki), co w sposób znaczący wpływa na błąd bezwzględny. Przykładowo dla $s = 5$, wartość funkcji obliczona w pojedynczej precyzji dla $n = 50, 100, 200, 500, 1000$ iteracji jest taka sama, ale zależy od sposobu sumowania (w przód albo wstecz). Natomiast dla precyzji podwójnej wartości uzyskane w kolejnych iteracjach nie są równe, ale ich różnica jest znacznie mniejsza niż w przypadku pojedynczej precyzji.

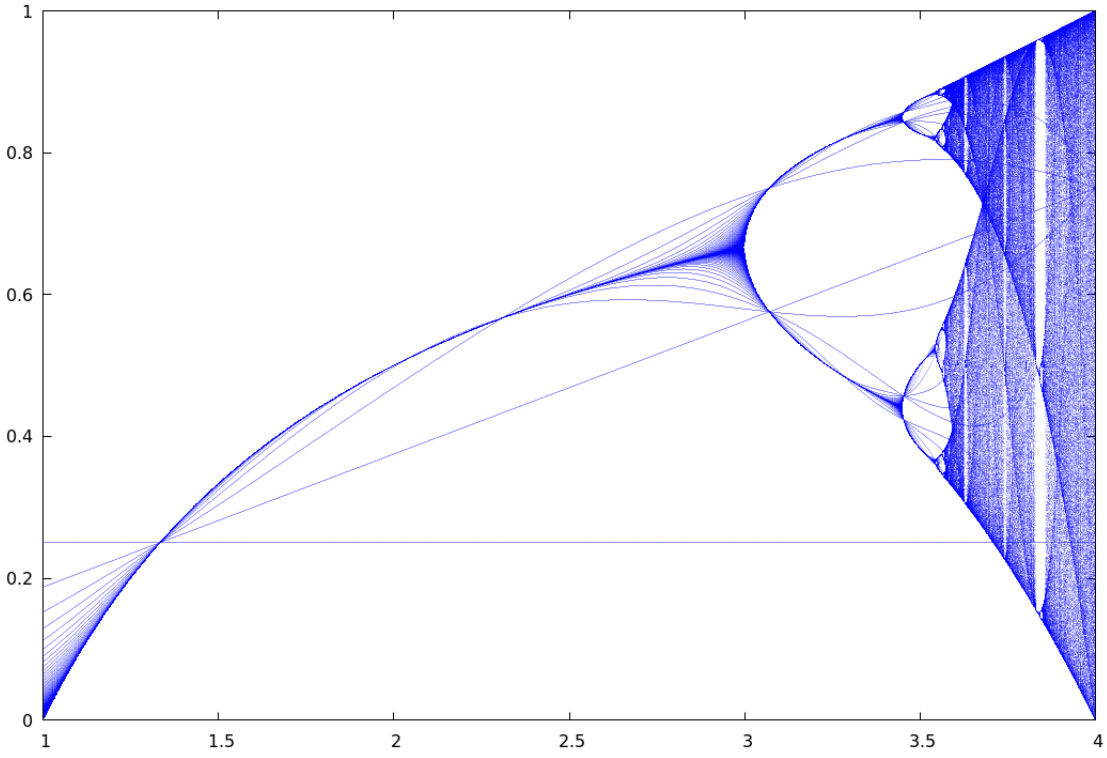
Utrata precyzji związana z naprzemiennym charakterem szeregu definiującego tę funkcję może zostać wyeliminowana na przykład przez osobne sumowanie wyrazów o różnych znakach (dodatnich i ujemnych), a potem zsumowaniu otrzymanych wyników częściowych.

Zadanie 4. Błędy zaokrągleń i odwzorowanie logistyczne

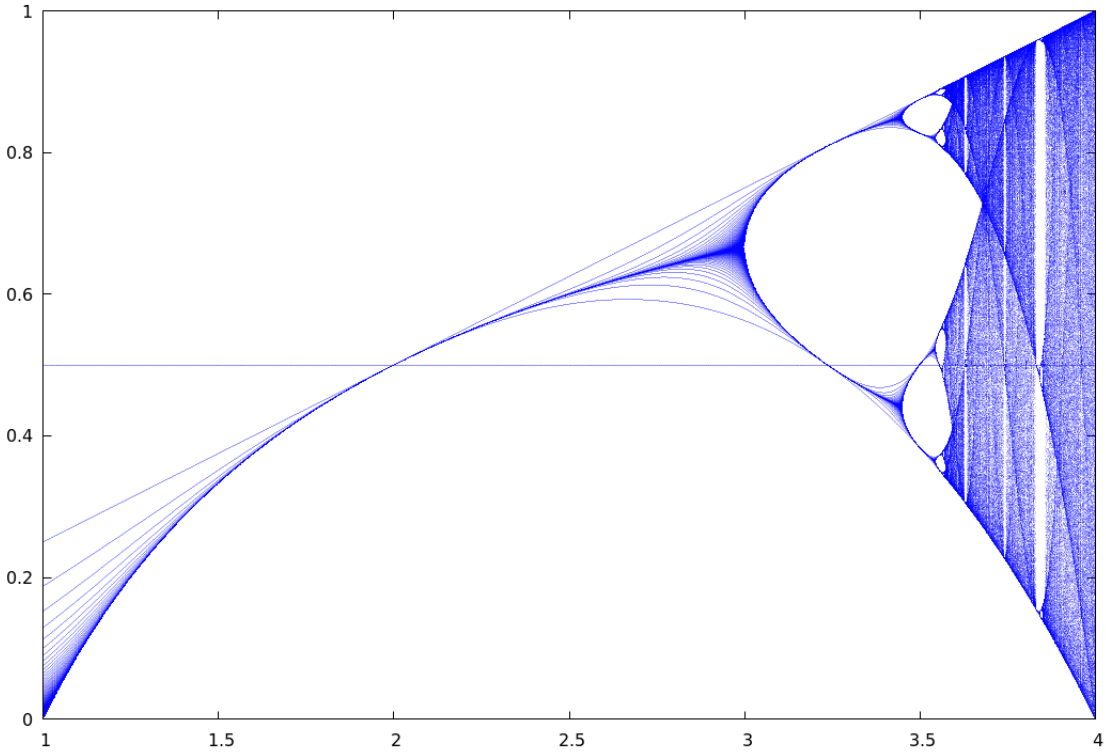
Odwzorowanie logistyczne: $x_{n+1} = rx_n(1 - x_n),\ 0 \leqslant x_n \leqslant 1,\ r > 0$



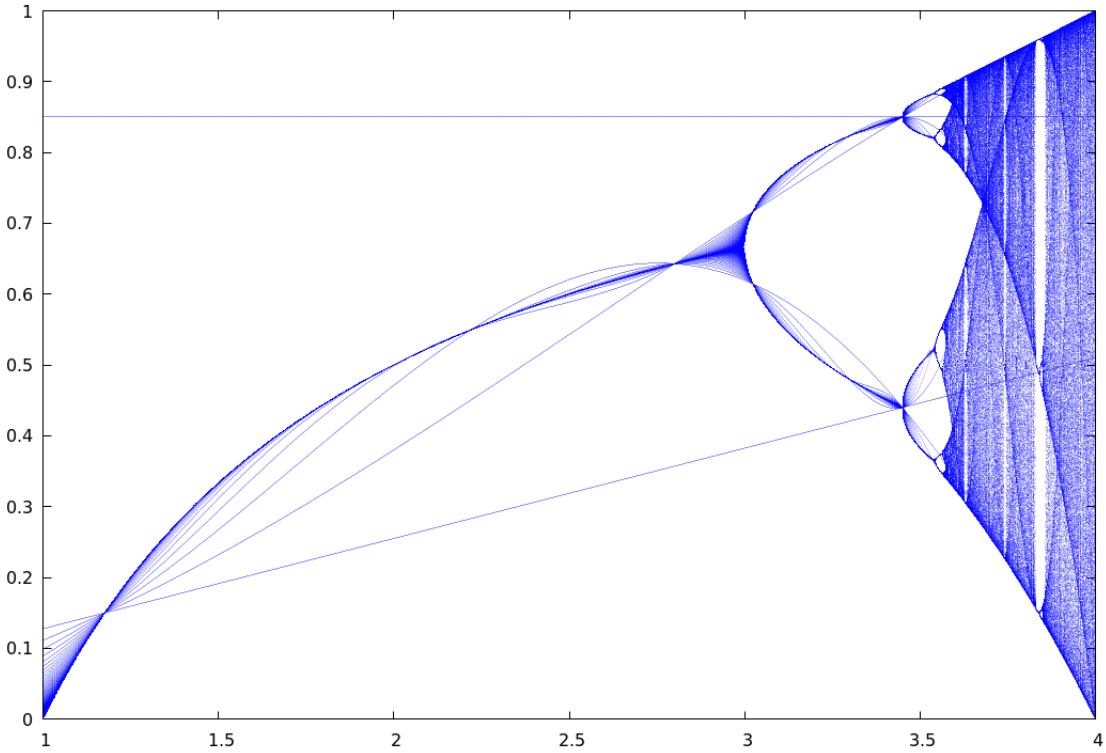
Rys. 2. Diagram bifurkacyjny dla $x_0 = 0.1$, $1 \leqslant r \leqslant 4$



Rys. 3. Diagram bifurkacyjny dla $x_0 = 0.25$, $1 \leqslant r \leqslant 4$

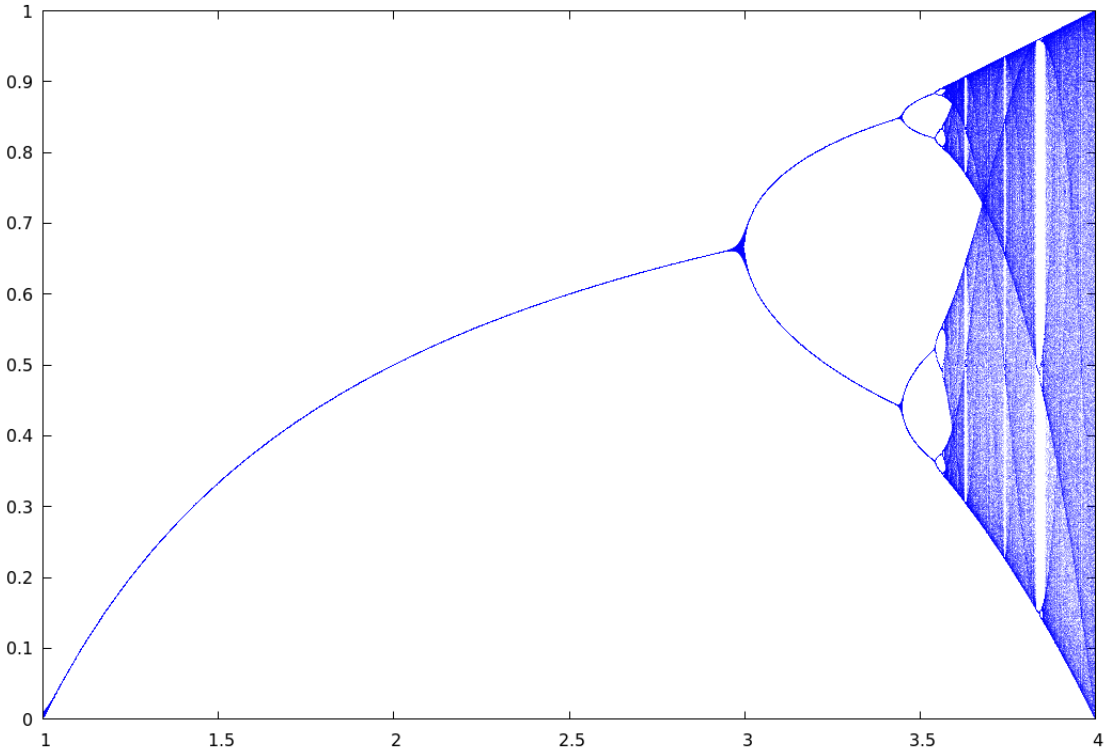


Rys. 4. Diagram bifurkacyjny dla $x_0 = 0.5$, $1 \leqslant r \leqslant 4$



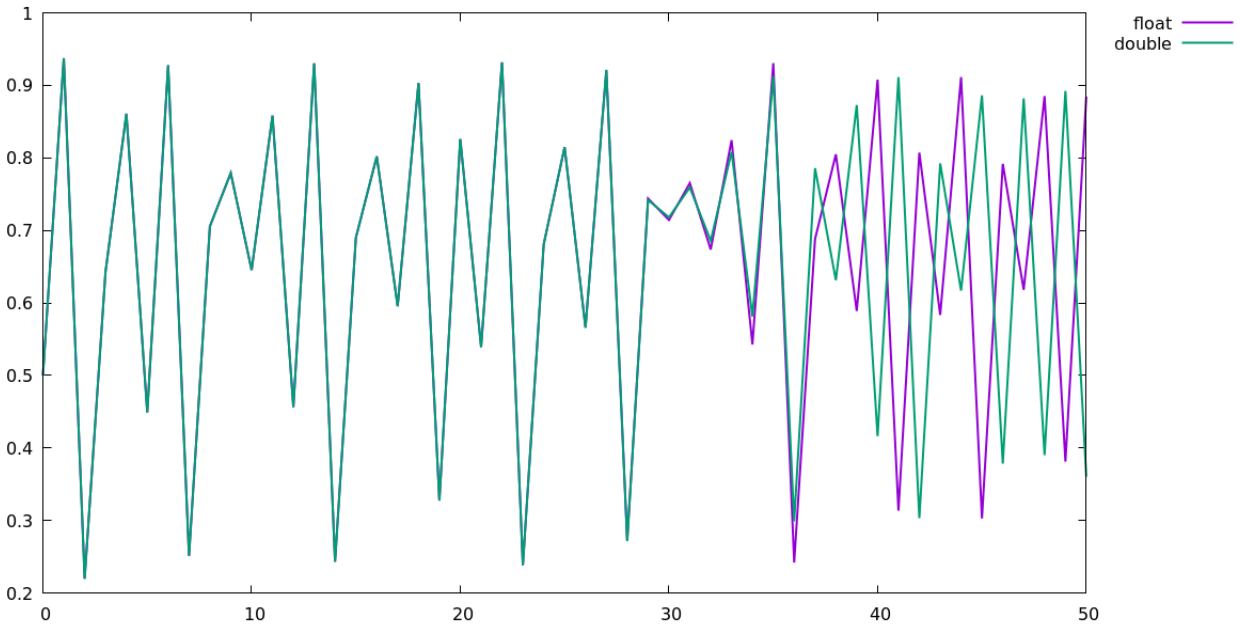
Rys. 5. Diagram bifurkacyjny dla $x_0 = 0.85$, $1 \leqslant r \leqslant 4$

Diagramy bifurkacyjne otrzymane dla różnych x_0 różnią się jedynie niewielką liczbą punktów odpowiadających wartościom osiąganym w początkowych iteracjach. Na poniższym diagramie pominięto pierwszych 100 iteracji.

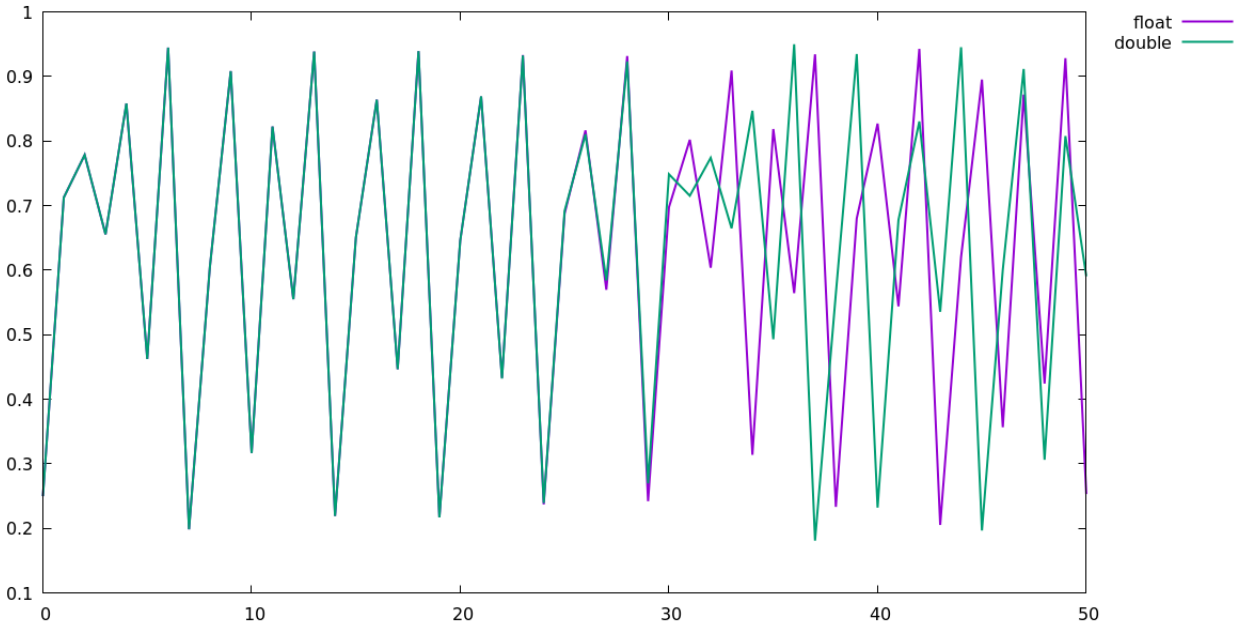


Rys. 6. Diagram bifurkacyjny dla $1 \leqslant r \leqslant 4$

Dla $1 \leqslant r \leqslant 3$ proces iteracyjny jest zbieżny do jednej wartości. Dla $r > 3$ kolejne wyrazy ciągu oscylują między dwoma wartościami – dlatego na diagramie można zauważyć rozdwojenie (zwane również bifurkacją – stąd nazwa diagram bifurkacyjny). Dla większych wartości r można dostrzec kolejne bifurkacje, a powyżej $3.569945\ldots$ kolejne iteracje odwzorowania stają się chaotyczne (chaos deterministyczny). Dla $k = 4$ pokryty jest cały przedział $(0, 1)$.

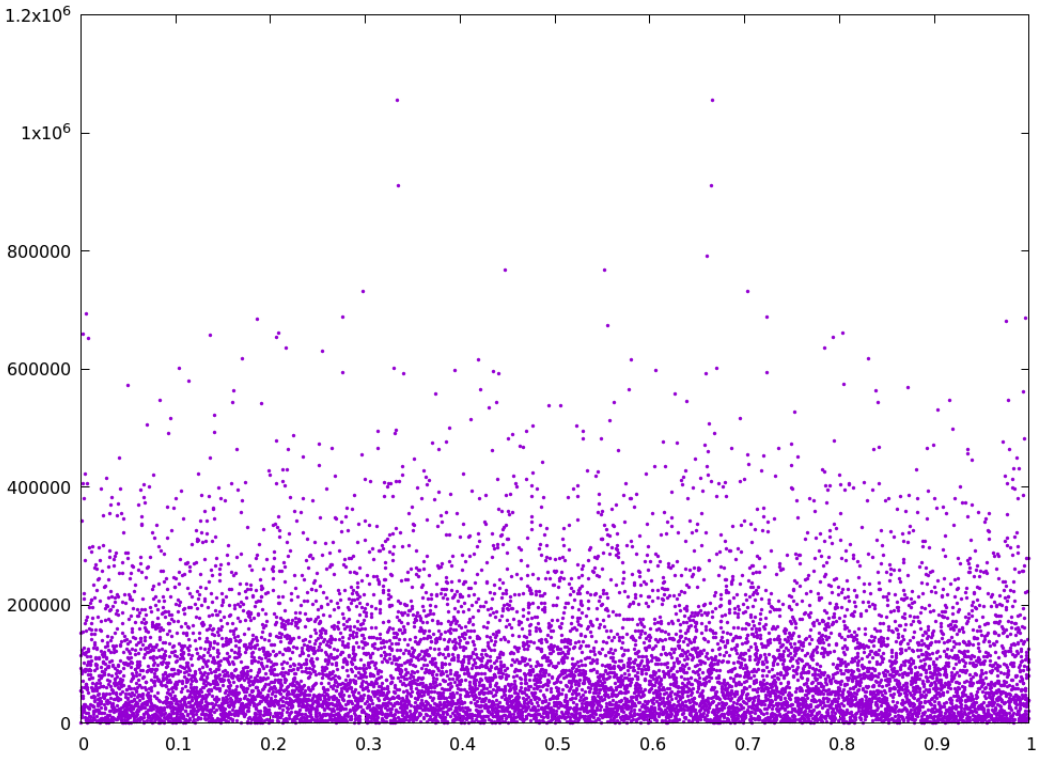


Rys. 7. Trajektorie obliczone przy użyciu liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji dla parametrów $x_0 = 0.5$, $r = 3.75$



Rys. 8. Trajektorie obliczone przy użyciu liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji dla parametrów $x_0 = 0.25$, $r = 3.8$

W początkowych iteracjach wyrazy ciągu obliczone przy użyciu pojedynczej i podwójnej precyzji są zbliżone, ale ich różnice mają znaczący wpływ na kolejne wartości.



Rys. 9. Liczba iteracji potrzebnych do osiągnięcia zera (z dokładnością do 10^{-9})

Bibliografia

[1] `https://en.wikipedia.org/wiki/Kahan_summation_algorithm`

[2] `https://en.wikipedia.org/wiki/Logistic_map`

[3] `https://pl.wikipedia.org/wiki/Odwzorowanie_logistyczne`