## Metody Obliczeniowe w Nauce i Technice

Laboratorium 1

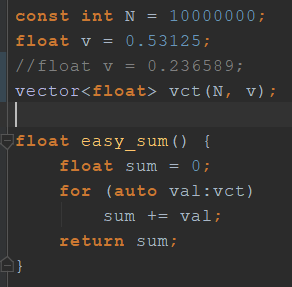
Arytmetyka komputerowa

Sprawozdanie

# Zadanie 1 – Sumowanie liczb pojedynczej precyzji

## 1.1 prosta iteracja

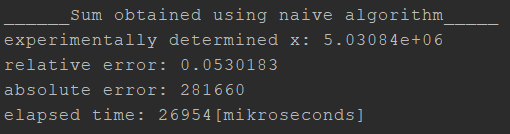
Kod programu:



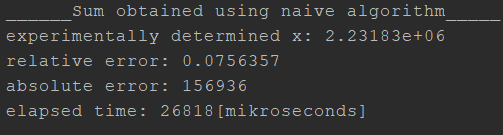
## 1.2 – 1.3 błąd względny i bezwzględny

Wykorzystując powyższą funkcję uzyskano następujące wyniki dla podanych danych

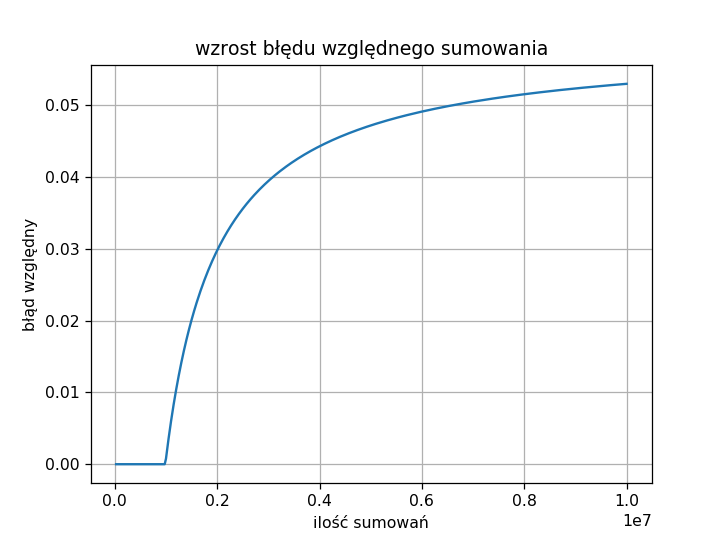
(N= 10**­7,** v =0.53125):

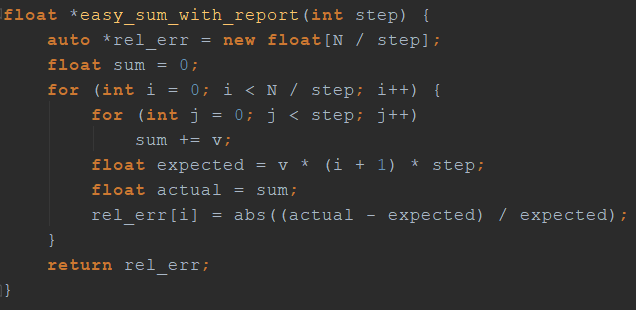


Wyniki dla (N= 10**­7,** v = 0.207489):



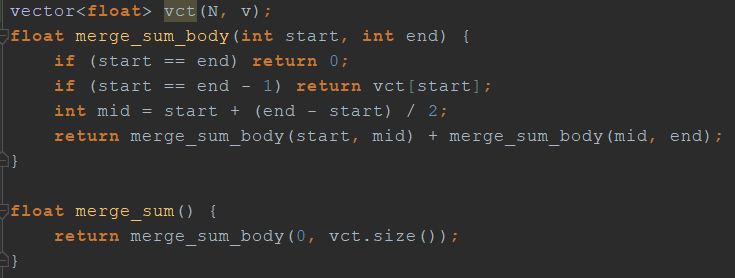
Wysoka wartość błędu względnego wynika z wielokrotnego dodawania w arytmetyce float liczb bardzo dużych do bardzo małych- gdyż za każdym razem dokładność wyniku jest dopasowywana do większej liczby, „gubiąc” dokładną wartość liczby mniejszej. Błąd ten kumuluje się z każdą iteracją. W przedstawionym zadaniu zachodzi to od momentu pokazanego na wykresie (ok 106 iteracji).



Kod generujący powyższy wykres.

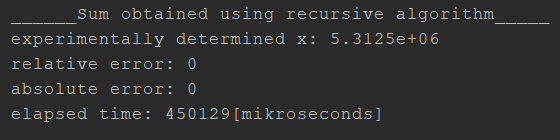
## 1.4- 1.5 rekurencyjny algorytm sumowania

Kod programu wykonujący sumowanie rekurencyjnie:

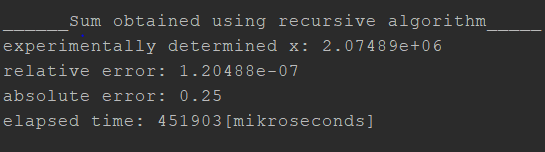


Wyniki dla danych jak z pkt. 1.1

(N= 10**­7,** v =0.53125):

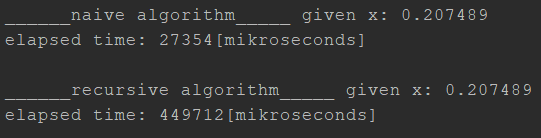


Wyniki dla (N= 10**­7,** v = 0.207489):



W przypadku zastosowania przy sumowaniu algorytmu rekurencyjnego obserwujemy znaczne zmniejszenie się błędu względnego. Wynika to z faktu usunięcia zjawiska dodawania do siebie liczb o różnych rzędach wielkości – na każdym etapie algorytmu dodawane liczby są niemalże takie same.

## 1.6 czasy działania algorytmów



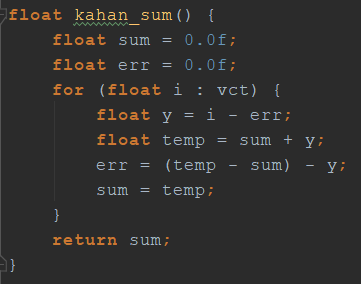
Jak widać w powyższym zestawieniu (i z poprzednich wyników), algorytm rekurencyjny jest znacznie wolniejszy od algorytmu naiwnego. Wynika to z odkładania danych na stos rekurencyjny oraz sprawdzania warunków wyjścia przy każdej iteracji.

## 1.7 niezerowy błąd w algorytmie rekurencyjnym

Pkt 1.7 został zaprezentowany w poprzednich przykładach – dla danych N=10**­7,** v=0.207489.

# Zadanie 2 – Algorytm Kahana

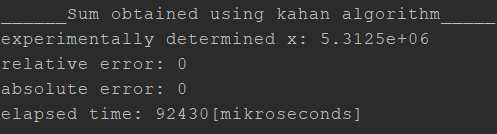
Kod programu:



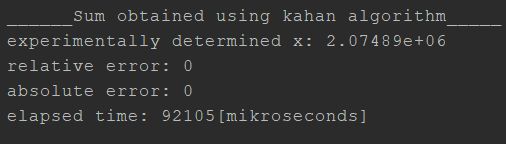
## 2.1 – 2.2 błąd względny i bezwzględny

Dla danych wejściowych jak w zad. 1:

(N= 10**­7,** v =0.53125):



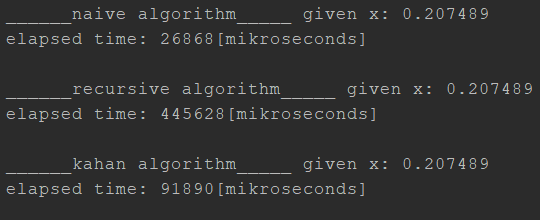
Wyniki dla (N= 10**­7,** v = 0.207489):



Algorytm Kahana jest dokładniejszy od obu poprzednich algorytmów. Wynika to z faktu korygowania przy każdej iteracji błędu dodawania dwóch liczb zmiennoprzecinkowych -zmienna *err* przechowuje aktualny błąd dodawania – różnicę między wartością rzeczywiście dodaną do sum, a liczbą która powinna być dodana (y). W kolejnej iteracji wartość *err* zostaje uwzględniona przez odjęcie jej od następnej sumowanej wartości (i).

## 2.3 czas wykonywania

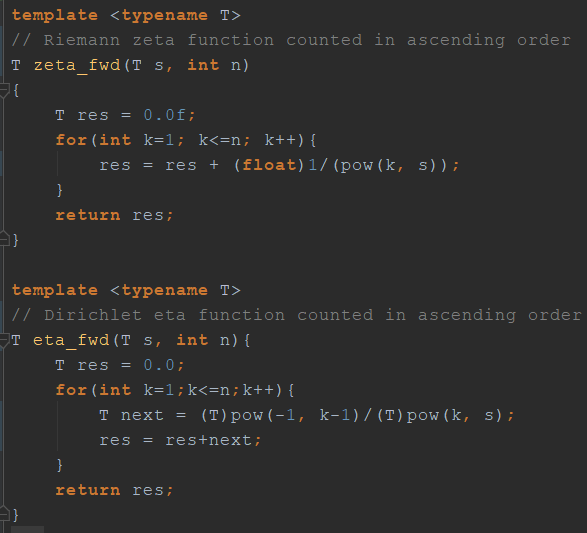
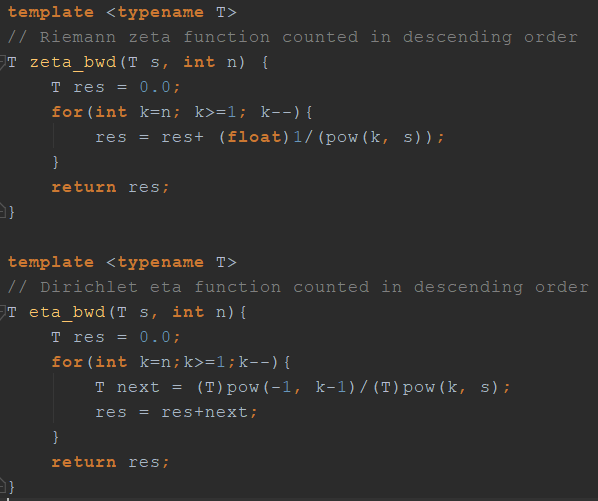
Zestawienie dla 3 algorytmów:



Algorytm Kahana jest wolniejszy od algorytmu naiwnego ze względu na większą ilość działań do wykonania w każdej iteracji. Jest jednak szybszy od algorytmu rekurencyjnego.

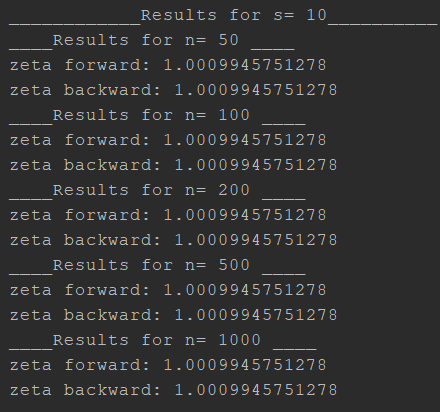
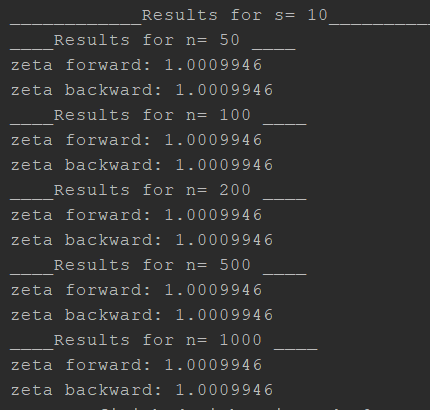
# Zadanie 3 – Sumy częściowe

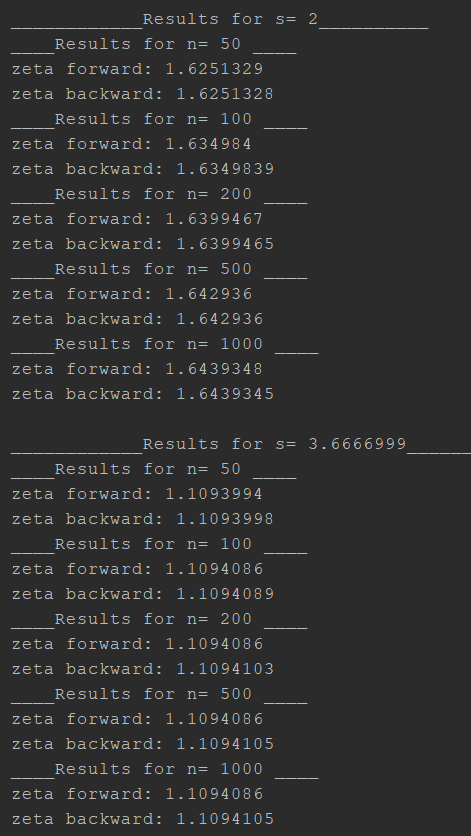
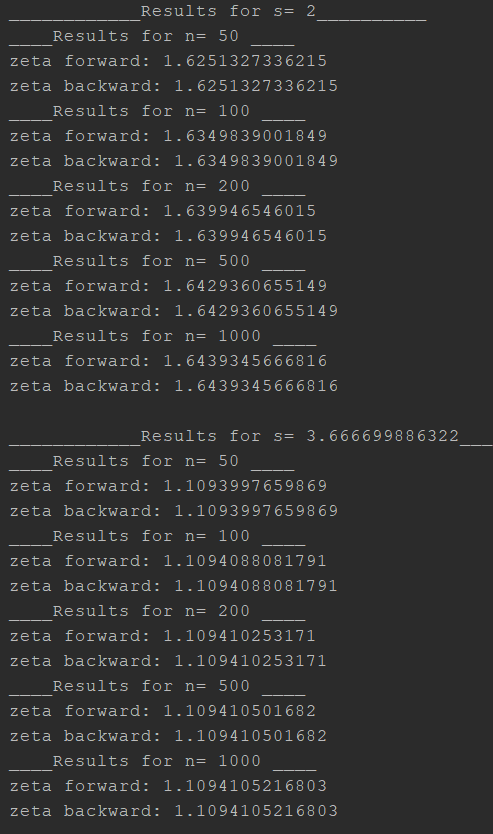
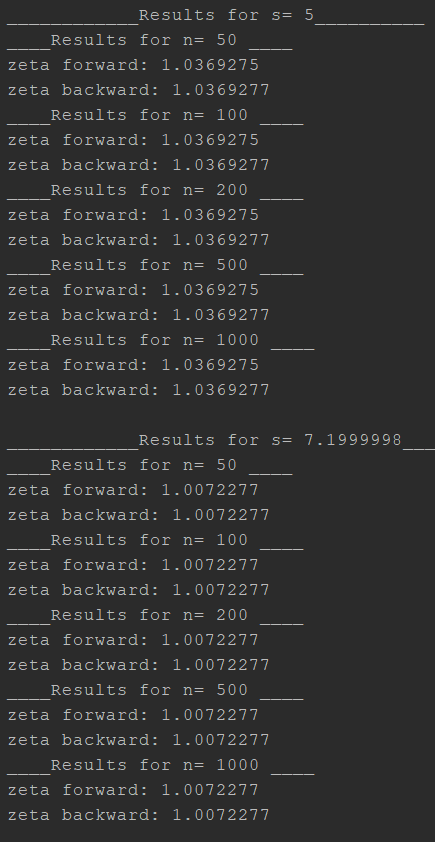
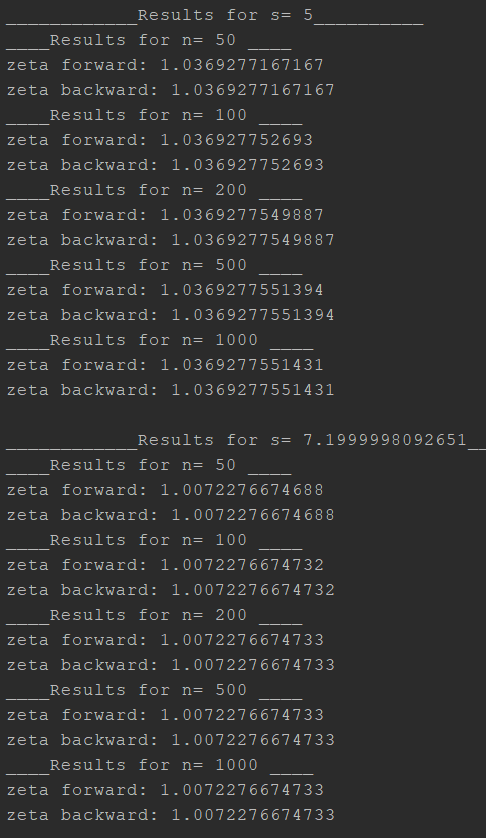
Kod programu:

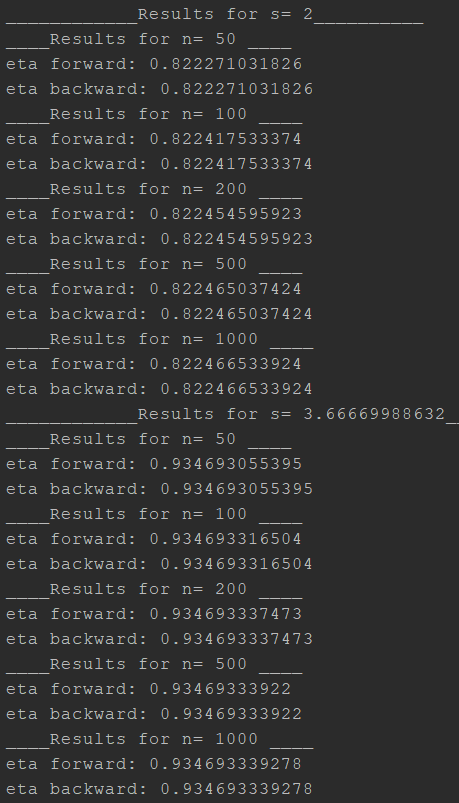
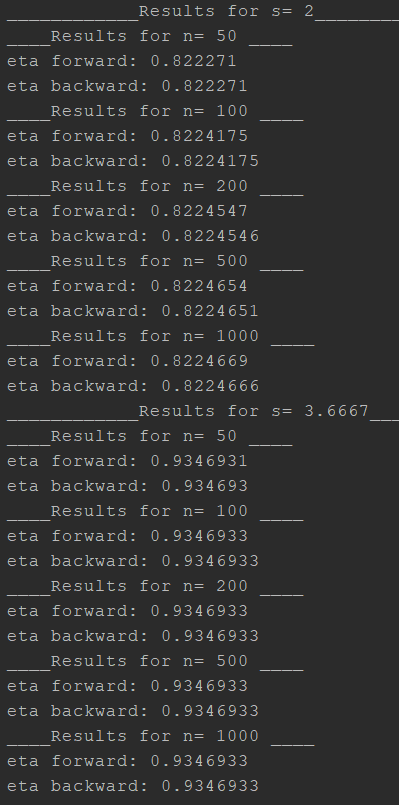
Wyniki działania dla zadanych danych:

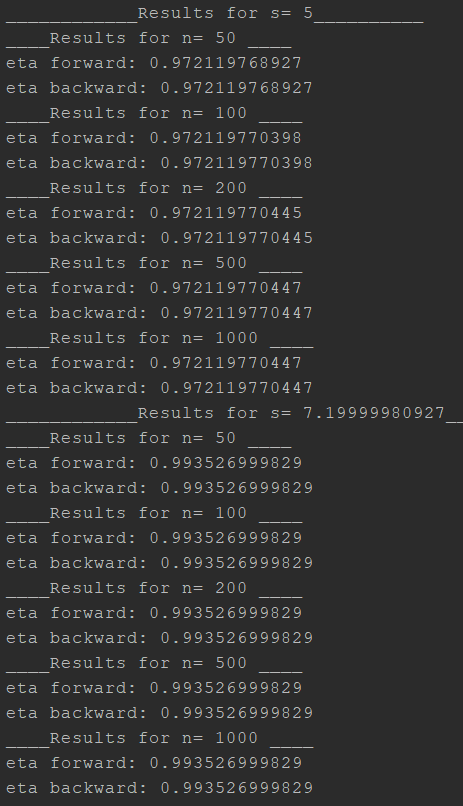
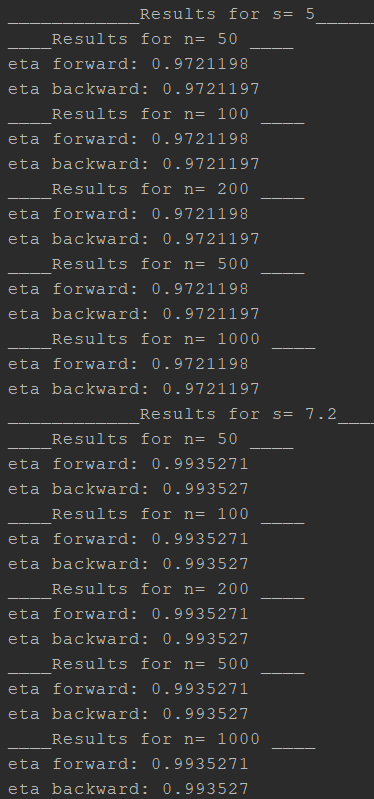
funkcja zeta: (float po lewej, double po prawej)

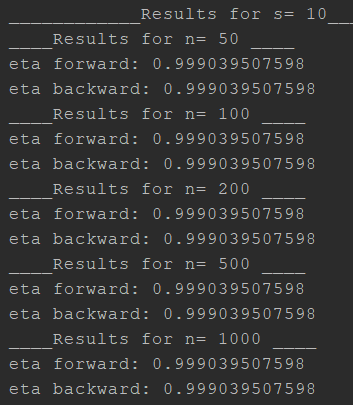
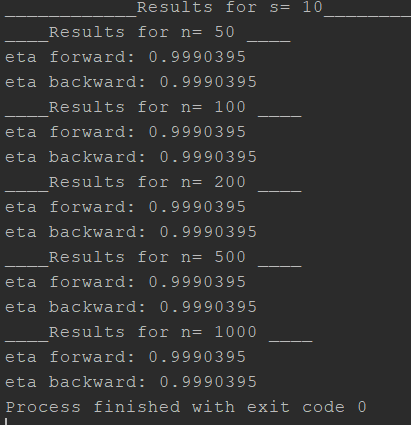


Funkcja eta: (float po lewej, double po prawej)







Wartości dokładne:

ζ(2) = 1.644934066848226436472 η(2) = 0.8224670334241132182362

ζ(3.6667) = 1.109410514586453357451 η(3.6667) = 0.9346933439191250729261

ζ(5) = 1.036927755143369926331 η(5) = 0.9721197704469093059357

ζ(7.2) = 1.007227666480717114739 η(7.2) = 0.9935270006616197875745

ζ(10) = 1.000994575127818085337 η(10) = 0.9990395075982715656392

Interpretacja wyników:

**Precyzja** **float**:

Dla funkcji **zeta** dla s>=7.2 i każdego przyjętego n, nie było różnicy między wynikami uzyskanymi sumowaniem w przód i w tył. Dla s <7.2 wyniki były dokładniejsze, gdy liczyliśmy funkcję sumując od tyłu. Przyczyną różnicy w wynikach jest dodawanie liczb o różnym rzędzie wielkości – od pewnego momentu kolejne elementy sumy były już względnie bardzo małe względem pierwszego elementu, zatem dodawanie kolejnych elementów do dotychczasowej sumy było obarczone bardzo dużym błędem względnym. W przypadku sumowania od tyłu, dodawaliśmy ze sobą elementy od najmniejszych do największych, co zredukowało ilość dodawań liczb różnych w rzędach wielkości, dzięki czemu uzyskane wyniki były dokładniejsze.

W przypadku funkcji **eta,** nie obserwujemy tej samej zależności – wręcz przeciwnie, dla s= 2 oraz 5 przy dużych n (500, 1000) wyniki były nieznacznie bardziej dokładne w przypadku sumowania od przodu. Dzieje się tak, gdyż opróczdodawania do siebie liczb o różnym rzędzie wielkości, gdy sumujemy „w przód”, pojawia się w tym przypadku również błąd spowodowany odejmowaniem od siebie 2 bardzo bliskich liczb, gdy sumujemy „od tyłu”. Jest to tzw. katastrofalne anulowanie – w wyniku odejmowania dwóch bliskich sobie liczb otrzymujemy wiele początkowych zer w mantysie- po jej znormalizowaniu i przesunięciu pierwszej cyfry znaczącej, pozostałe miejsca mantysy zapełnianie są przypadkowymi cyframi, co powoduje pojawienie się niedokładności.

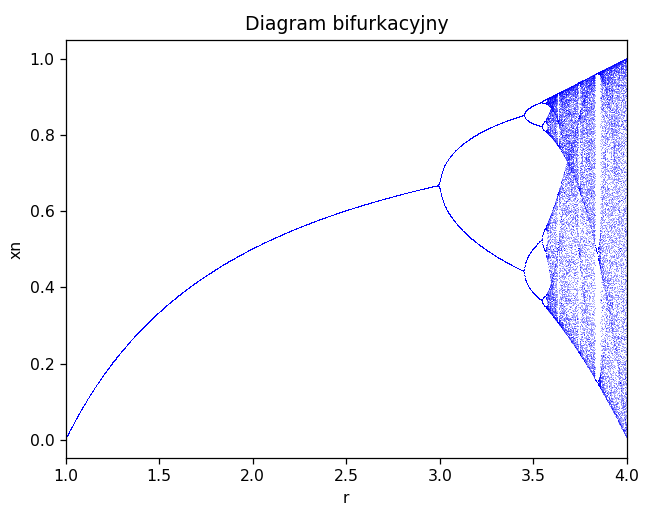
**Precyzja** **double**:

W przypadku działania na liczbach o podwójnej precyzji, uzyskane wyniki są niezależne od sposobu sumowania. Dla s<7.2 wyniki są dokładniejsze dla coraz większych n, a w przypadku s>=7.2 bardzo dobre oszacowanie dostajemy już przy n=50. Dwukrotne zwiększenie precyzji znacznie zwiększyło dokładność obliczeń i uzyskanych wyników.

# Zadanie 4 -Błędy zaokrągleń i odwzorowanie logistyczne

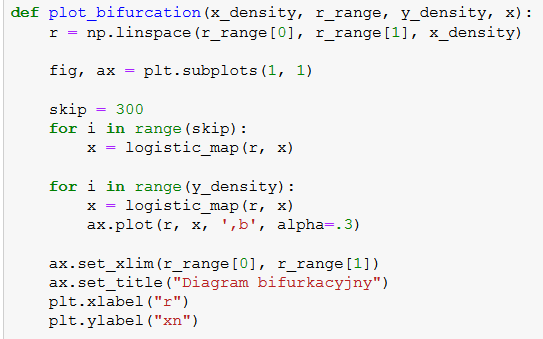
## Diagram bifurkacyjny

Diagram bifurkacyjny dla (1 <= r <= 4) i wybranego x­0­­=0.6 (kształt diagramu jest niezależny od wybranych x0)

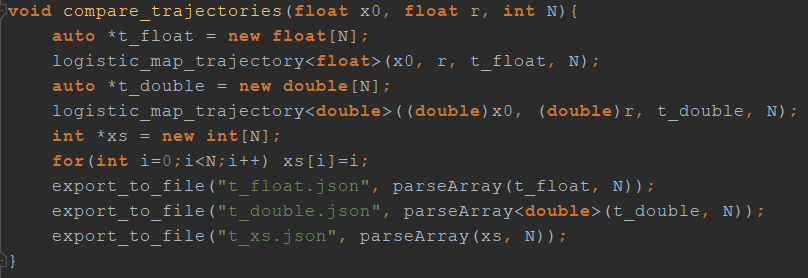
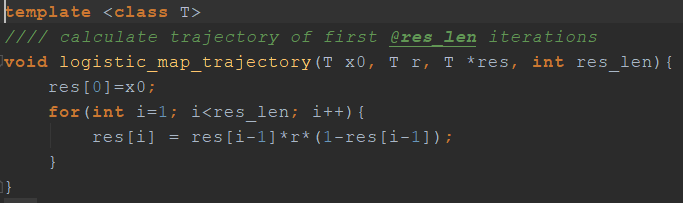


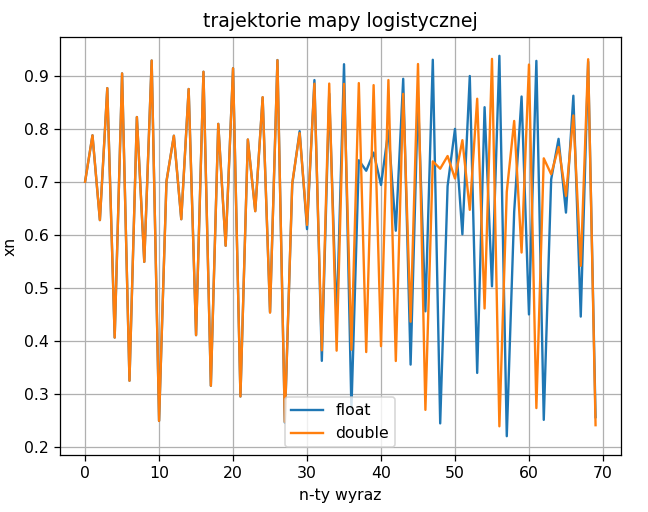
Kod programu (napisany w pythonie dla wygodniejszej wizualizacji):





## Porównianie trajektorii





## 4.3 Osiąganie zera

