Metody Obliczeniowe w Nauce i Technice

Laboratorium 1

Wojciech Łącki

1. Sumowanie liczb pojedynczej precyzji

from datetime import datetime  
from matplotlib import pyplot  
import numpy  
  
n = 10 \*\* 7  
x = numpy.single(0.53125)   
T = [x for \_ in range(n)]  
  
  
def zad1():  
 relativeErrors = []  
 X = []  
  
 def sumIter(T):  
 suma = numpy.single(0)  
 for i in range(n):  
 suma += T[i]  
 if ((i + 1) % 25000 == 0):  
 X.append(i)  
 realVal = i \* x  
 relativeError = abs(realVal - suma) / realVal \* 100  
 relativeErrors.append(relativeError)  
 return suma  
  
 def sumReq(T, start, end):  
 middle = (end + start) // 2  
 if (start < end):  
 return sumReq(T, start, middle) + sumReq(T, middle + 1, end)  
 return T[start]  
  
 real = 10 \*\* 7 \* x  
 start\_time = datetime.now()  
 sumaIter = sumIter(T)  
 end\_time = datetime.now()  
 err1 = abs(real - sumaIter)  
 err2 = err1 / real  
 print("X:", x)  
 print("DOKLADNA SUMA:", real)  
 print("ITERACYJNIE:")  
 print("SUMA:", sumaIter, "BEZWZGLEDNY:", err1, "WZGLEDNY:", err2)  
 print('CZAS: {}'.format(end\_time - start\_time))  
 print("%:", err2 \* 100)  
 start\_time = datetime.now()  
 sumaReq = sumReq(T, 0, n - 1)  
 end\_time = datetime.now()  
 err1 = abs(real - sumaReq)  
 err2 = err1 / real  
 print("REKURENCYJNIE:")  
 print("SUMA:", sumaReq, "BEZWZGLEDNY:", err1, "WZGLEDNY:", err2)  
 print('CZAS: {}'.format(end\_time - start\_time))  
 print("%:", err2 \* 100)  
 pyplot.plot(X, relativeErrors, marker=".", markersize=1, linestyle='None')  
 pyplot.show()  
  
zad1()

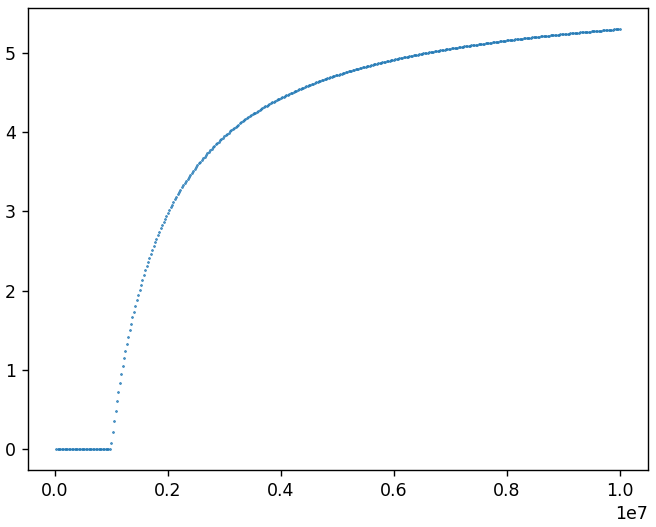
Poniżej przedstawione są wyniki, które otrzymaliśmy.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Jak możemy zauważyć błąd względny dla sumy obliczonej w sposób iteracyjny wyszedł około 5.3%, czyli całkiem duży. Jest to spowodowane tym, że po pewnej liczbie iteracji zaczynamy dodawać stosunkowo małe liczby do dużego akumulatora, przez co tracimy informacje na temat mniej znaczących bitów.

Poniższy wykres przedstawia jak zmienia się błąd względny z kolejnymi iteracjami.

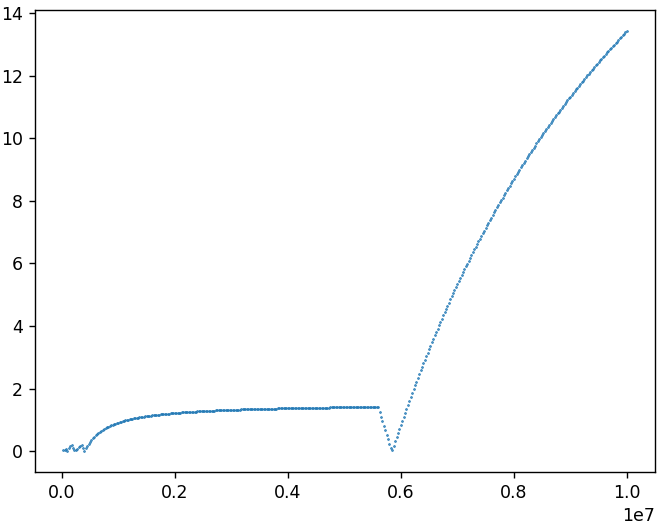


Można zauważyć, że błąd ten rośnie znacząco z kolejnymi przejściami pętli. W początkowej fazie błąd jest niezauważalny, po czym drastycznie zaczyna rosnąć, co oznaczają duże przerwy między kolejnymi punktami. Później błąd zaczyna coraz wolniej rosnąć.

Analizując teraz algorytm rekurencyjny, możemy zauważyć, że w tym przypadku nie zaobserwowano błędu. Wynika to z faktu, że za każdym razem dodajemy do siebie zbliżone liczby (tutaj takie same), przez co nie gubimy mniej znaczących bitów w takim stopniu jak to było w przypadku podejścia iteracyjnego.

Jednak analizując czas działania, nie ma wątpliwości, że sumowanie iteracyjne będzie szybsze, niże rekurencyjne, w tym przypadku około dwa razy. Niestety rekurencja kosztuje.

Jednak można znaleźć wartość początkową, którą wypełniamy tablicę taką, że wynik sumowania rekurencyjnego będzie niezerowy. Stanie się tak na przykład, gdy zmienną „x” ustawimy na „0.368373”. Wykres błędu względnego również nie wygląda już tak ładnie jak w przypadku powyższym.



1. Algorytm Kahana

import numpy   
from datetime import datetime  
  
n = 10 \*\* 7  
x = numpy.single(0.53125)   
T = [x for \_ in range(n)]  
  
  
def zad2():  
 def sumKahan(T):  
 suma = numpy.single(0)  
 err = numpy.single(0)  
 for add in T:  
 y = add - err  
 temp = suma + y  
 err = (temp - suma) - y  
 suma = temp  
 return suma  
  
 real = numpy.single(10 \*\* 7 \* x)  
 start\_time = datetime.now()  
 suma = sumKahan(T)  
 end\_time = datetime.now()  
 err1 = abs(real - suma)  
 err2 = err1 / real  
 print("X:", x)  
 print("DOKLADNA SUMA:", real)  
 print("KAHAN:")  
 print("SUMA:", suma, "BEZWZGLEDNY:", err1, "WZGLEDNY:", err2, 'CZAS: {}'.format(end\_time - start\_time))  
 print("%:", err2 \* 100)  
  
zad2()

Uzyskaliśmy następujący wynik.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Widzimy, że nie wystąpiły błędy, co oznacza, że ten algorytm ma dobre własności numeryczne, a na pierwszy rzut oka nie różni się wiele od zwykłego podejścia iteracyjnego. Wynika to z faktu, że w kolejnych iteracjach dodajemy utracone mniej znaczące bity, do czego służy zmienna „err”. Dzięki temu zabiegowi wynik otrzymujemy dokładny wynik.

Porównując czasy, to algorytm Kahana jest trochę wolniejszy od zwykłego sumowania iteracyjnego, co jest oczywiste, gdyż są wykonywane dodatkowe operacje. Ale nie dość, że daje bardzo dokładny wynik, to jeszcze jest szybszy od podejścia rekurencyjnego, które czasami pokazało niewielkie błędy względne.

1. Sumy częściowe

import numpy  
  
s = [2, 3.6667, 5, 7.2, 10]  
n = [50, 100, 200, 500, 1000]  
len\_s = len(s)  
len\_n = len(n)  
  
  
def zad3():  
 def dzeta(n, s):  
 sumForwardSingle = numpy.single(0)  
 sumBackwardSingle = numpy.single(0)  
 sumForwardDouble = numpy.double(0)  
 sumBackwardDouble = numpy.double(0)  
 for k in range(1, n + 1):  
 sumForwardSingle += numpy.single(1 / (k \*\* s))  
 sumForwardDouble += numpy.double(1 / (k \*\* s))  
 for k in range(n, 0, -1):  
 sumBackwardSingle += numpy.single(1 / (k \*\* s))  
 sumBackwardDouble += numpy.double(1 / (k \*\* s))  
 return sumForwardSingle, sumBackwardSingle, sumForwardDouble, sumBackwardDouble  
  
 def eta(n, s):  
 sumForwardSingle = numpy.single(0)  
 sumBackwardSingle = numpy.single(0)  
 sumForwardDouble = numpy.double(0)  
 sumBackwardDouble = numpy.double(0)  
 for k in range(1, n + 1):  
 sumForwardSingle += numpy.single((-1) \*\* (k - 1) / (k \*\* s))  
 sumForwardDouble += numpy.double((-1) \*\* (k - 1) / (k \*\* s))  
 for k in range(n, 0, -1):  
 sumBackwardSingle += numpy.single((-1) \*\* (k - 1) / (k \*\* s))  
 sumBackwardDouble += numpy.double((-1) \*\* (k - 1) / (k \*\* s))  
 return sumForwardSingle, sumBackwardSingle, sumForwardDouble, sumBackwardDouble  
  
 for i in range(len\_n):  
 for j in range(len\_s):  
 print("n:", n[i], "s:", s[j])  
 dz = dzeta(n[i], s[j])  
 et = eta(n[i], s[j])  
 print("dzeta")  
 print("forwardSingle backwardSingle forwardDouble backwardDouble")  
 print(dz)  
 print("Difference single:", abs(dz[1] - dz[0]))  
 print("Difference double:", abs(dz[3] - dz[2]))  
 print("eta")  
 print("forwardSingle backwardSingle forwardDouble backwardDouble")  
 print(et)  
 print("Difference single:", abs(et[1] - et[0]))  
 print("Difference double:", abs(et[3] - et[2]))  
 print()  
  
zad3()

Jednym z wyników jest ukazany poniżej.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Wyniki liczenia wprzód i w tył różnią się od siebie dla tych samych parametrów, jednak zdarzają się przypadki, gdzie wyniki pojedynczej precyzji dla funkcji „dzeta” między tymi dwoma metodami są równe 0 (na przykład dla n=1000 oraz s=7.2). Pokażemy różnicę rozpisując i porównując oszacowania względnych błędów działań dwóch poniższych wzorów. Używamy maszynowego oraz założenia, że .

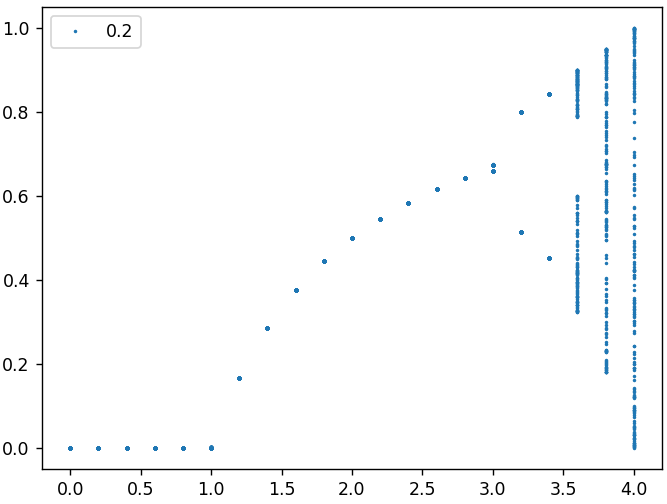
Wzór 1.

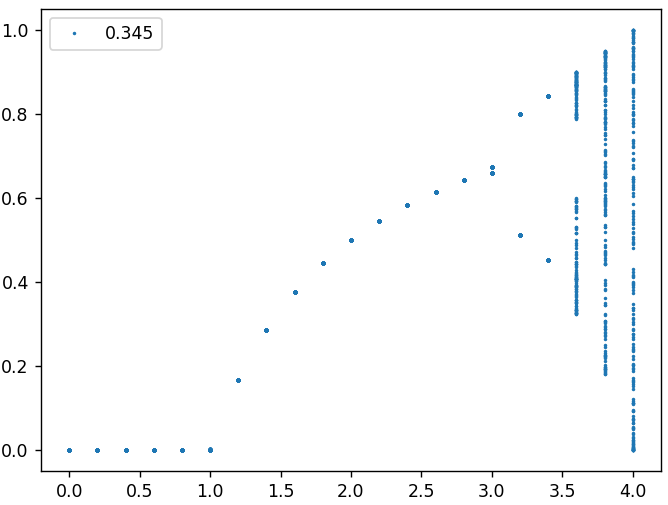
Analizując funkcję „dzeta” można zobaczyć, że kolejne składniki sumy są coraz mniejsze, ale zawsze są dodatnie (czyli x<z, co w naszym przypadku oznacza, że jest dodawaniem do przodu, a w tył). Podstawiając do „Wzór 1.”, otrzymamy wynik dodatni, co oznacza, że dodając od największych liczb (czyli u nas od przodu) będziemy otrzymywać coraz większy błąd. Podobnie wygląda sprawa dla funkcji „eta”.

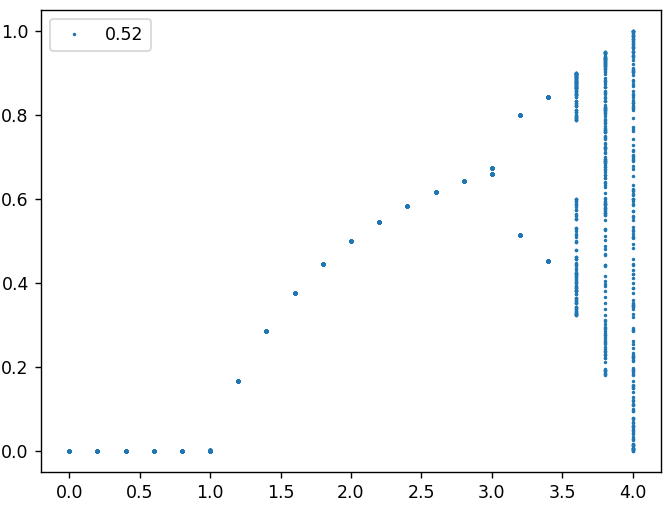
1. Błędy zaokrągleń i odwzorowanie logistyczne

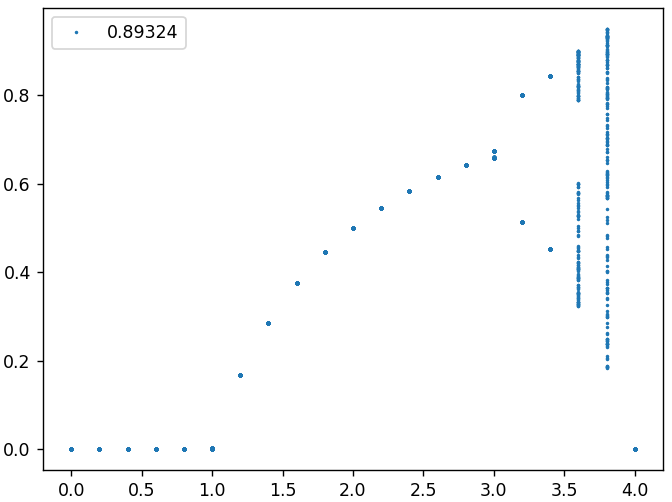
import numpy  
from matplotlib import pyplot  
  
  
def zad4():  
 def logic():  
 return numpy.single(r \* numpy.single(x) \* numpy.single(1 - x))  
  
 x0s = [numpy.single(0.2), numpy.single(0.345), numpy.single(0.52), numpy.single(0.89324)]  
 for x0 in x0s:  
 r = 0  
 X = []  
 Y = []  
 for i in range(21):  
 x = x0  
 for j in range(1000):  
 x = logic()  
 if (j >= 800):  
 X.append(r)  
 Y.append(x)  
 r += 0.2  
 pyplot.clf()  
 pyplot.plot(X, Y, marker=".", markersize=2, linestyle='None', label=str(x0))  
 pyplot.legend()  
 pyplot.show()  
  
zad4()

Poniżej są przedstawione wykresy dla kolejnych wartości .





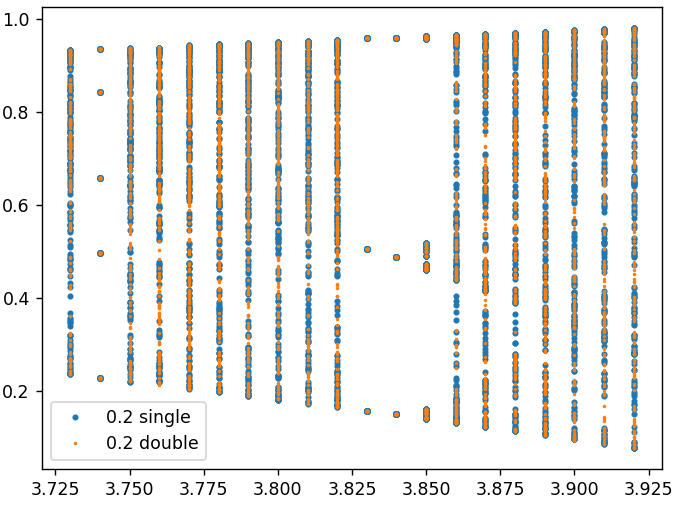


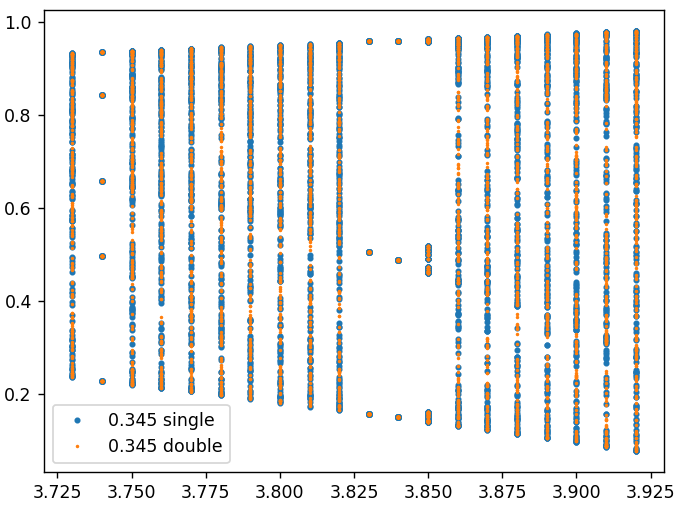


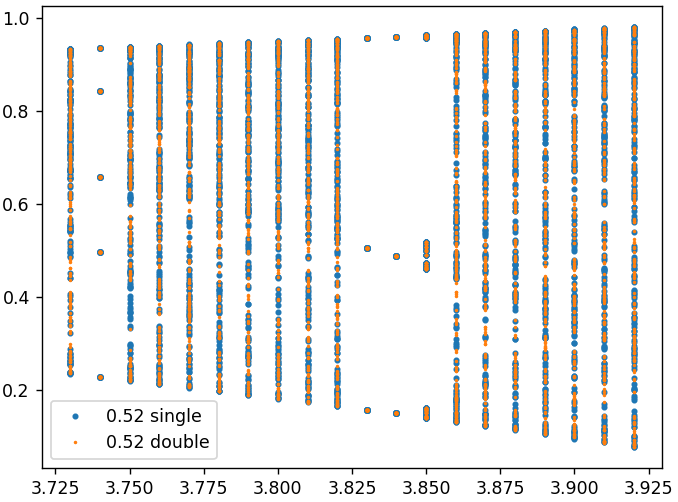
Można zaobserwować, że parametr nie ma dużego wpływu na to jak będą rozmieszczone punkty po wielu iteracjach i jest tak do , natomiast dalej ciężko jest to określić. Widać też, że parametr r ma duży wpływ na wynik. Ciekawe jest to, że dla wartości są przy zerze. Dla r<3 punkty gromadzą się w jednym miejscu, dla wokół dwóch miejsc, a dalej nie da się określić.

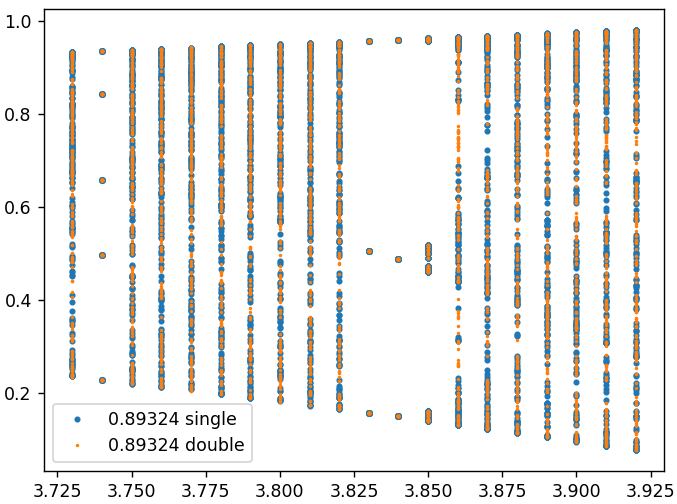
import numpy  
from matplotlib import pyplot  
  
  
def zad4():  
 def logicSingle(x):  
 return numpy.single(r \* numpy.single(x) \* numpy.single(1 - x))  
  
 def logicDouble(x):  
 return numpy.double(r \* numpy.double(x) \* numpy.double(1 - x))  
  
 x0s = [numpy.single(0.2), numpy.single(0.345), numpy.single(0.52), numpy.single(0.89324)]  
 x0d = [numpy.double(0.2), numpy.double(0.345), numpy.double(0.52), numpy.double(0.89324)]  
 for k in range(len(x0s)):  
 r = 3.73  
 X = []  
 Y = []  
 Y2 = []  
 for i in range(20):  
 x1 = x0s[k]  
 x2 = x0d[k]  
 for j in range(1000):  
 x1 = logicSingle(x1)  
 x2 = logicDouble(x2)  
 if (j >= 800):  
 X.append(r)  
 Y.append(x1)  
 Y2.append(x2)  
 r += 0.01  
 pyplot.clf()  
 pyplot.plot(X, Y, marker=".", markersize=5, linestyle='None', label=str(x0s[k])+" single")  
 pyplot.plot(X, Y2, marker=".", markersize=2, linestyle='None', label=str(x0d[k])+" double")  
 pyplot.legend()  
 pyplot.show()  
  
zad4()

Dla tych samych co w podpunkcie a), zobaczymy co się dzieje dla liczb pojedynczej i podwójnej precyzji dla wartości .









Widzimy, że trajektoria jest podobna w obu przypadkach. Warto zauważyć, że dla niektórych „r” pojedyncza i podwójna precyzja się pokrywają, tak jak na przykład dla .

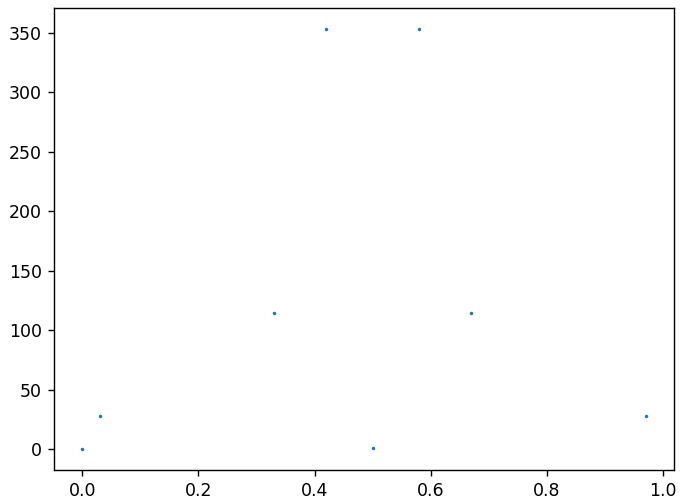
import numpy  
from matplotlib import pyplot  
  
  
def zad4():  
 def logic():  
 return numpy.single(r \* numpy.single(x) \* numpy.single((1 - x)))  
  
 x0 = []  
 y0 = []  
 r = 4  
 for i in range(100):  
 x = i \* 0.01  
 for j in range(1000):  
 x = logic()  
 if x == 0:  
 x0.append(i \* 0.01)  
 y0.append(j)  
 break  
 print("x0:", x0)  
 print("integrations:", y0)  
 pyplot.plot(x0, y0, marker=".", markersize=2, linestyle='None', label=str(x0))  
 pyplot.show()  
  
zad4()

Otrzymujemy następujące punktu wraz z liczbą iteracji potrzebnych do osiągnięcia 0.

Obraz zawierający tekst, zegarek, urządzenie, wskaźnik

Opis wygenerowany automatycznie

Na wykresie wygląda to następująco.



Wykres jest symetryczny względem osi , co pokazuje jak ciekawe jest to zadanie.