

Metody Obliczeniowe w Nauce i Technice

Laboratorium 1

Arytmetyka Komputerowa

Sprawozdanie

Yurii Vyzhha

13 października 2017

Zadanie 1 Sumowanie liczb pojedynczej precyzji

Zwykły algorytm sumowania

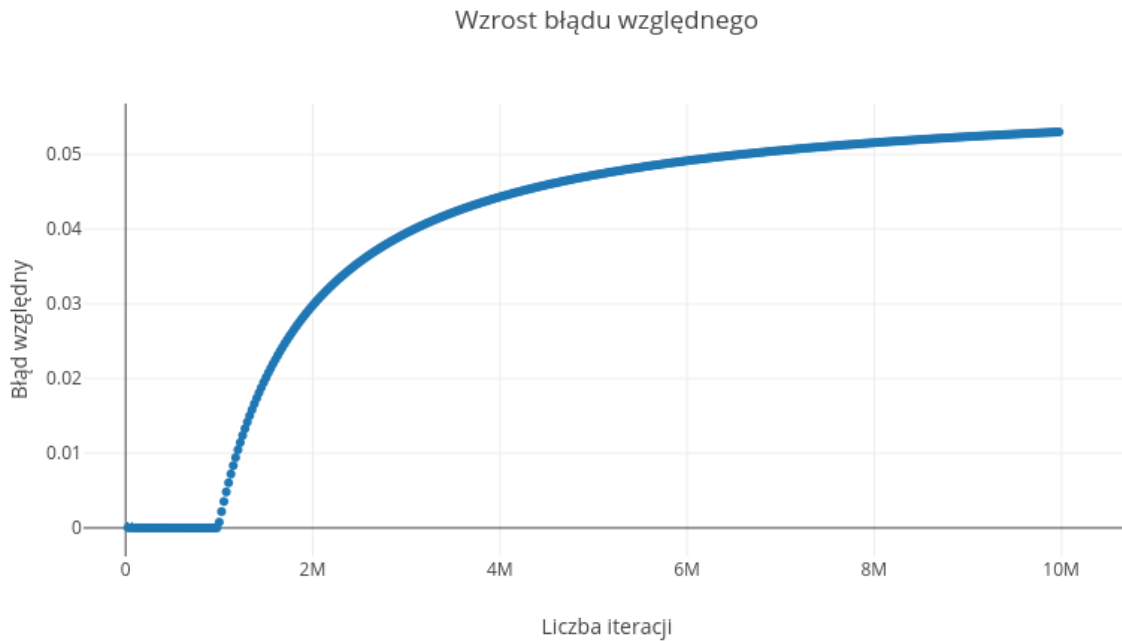
```
float[] ar = new float[10000000];  
float v = 0.53125f;  
for (int i = 0; i < ar.length; i++) {  
    ar[i] = v;  
}  
float sum = 0.0f;  
for (int i = 0; i < ar.length; i++) {  
    sum += ar[i];  
}
```

Bezwzględny błąd: $\sim 5.3\%$.

Względny błąd: 281659.5.

Wielkość błędu sumowania jest związana z reprezentacją liczb zmiennoprzecinkowych oraz z implementacją sumowania dwóch l.z. Dla obliczenia sumy dwóch l.z. mantysę mniejszej z liczb musimy doprowadzić do postaci większej z liczb co wiąże się z utratą mniej znaczących cyfr cechy.

Poniżej przedstawiamy wykres który pokazuje zależność pomiędzy błędem bezwzględnym a ilością iteracji wykonanych przez program.



Jak widzimy na wykresie, przy pierwszych 10^6 iteracji nie mamy błędu. To jest związane z tym, że wielkość cechy jest wystarczająco duża, a różnica między liczbami, które dodajemy, jest wystarczająco mała. Później błąd bezwzględny się pojawia, ale rośnie co raz z mniejszą prędkością. To jest spowodowane tym, że z każdym dodawaniem mylimy się na stałą liczbę, a suma rośnie, więc udział błędu w sumie jest co raz mniejszy.

Rekurencyjny algorytm dodawania

```
public static float recSum(float[] ar, int a, int b) {  
    if (a == b) return ar[a];  
    if (b - a == 1) {  
        return ar[a] + ar[b];  
    }  
    return recSum(ar, a, (b + a)/2) + recSum(ar, (b + a)/2 + 1, b);  
}
```

Bezwzględny błąd: 0.0%.

Względny błąd: 0.0.

Korzystając z rekurencyjnego algorytmu, nie dostaliśmy błędu. Ten algorytm jest zaimplementowany tak, że dodaje liczby wyłącznie o tej samej wielkości, więc przesunięcie cechy i, co za tym idzie, utrata precyzji nie występują.

Czas działania obu algorytmów jest mniej więcej taki sam dla różnych danych wejściowych.

Zadanie 2 Algorytm Kahana

Implementacja algorytmu Kahana w języku Java

```
float sum = 0.0f;  
float err = 0.0f;  
for (int i = 0; i < ar.length; i++) {  
    float y = ar[i] - err;  
    float temp = sum + y;  
    err = (temp - sum) - y;  
}
```

```

    sum = temp;
}

```

Algorytm Kahana działa w następujący sposób. Każdego razu jak dodajemy nową liczbę do ogólnej sumy ($sum + y$), obliczamy korektę (err), który stosujemy w następnej iteracji. Najpierw odejmujemy korektę err , którą obliczyliśmy w poprzedniej iteracji pętli i otrzymujemy poprawiony składnik y . Później dodajemy ten składnik do sumy sum . Najmniej znaczące bity y straciliśmy przy sumowaniu. Potem obliczamy najbardziej znaczące bity y za pomocą $temp - sum$. Kiedy odejmniemy y od otrzymanej różnicy, odzyskamy najmniej znaczące bity. Te właśnie bity straciliśmy przy obliczaniu $sum + y$. One będą korektą w następnej iteracji. Czas działania algorytmów Kahana oraz rekurencyjnego jest mniej więcej taki sam dla różnych danych wejściowych.

Zadanie 3 Sumy częściowe

Tablice należy czytać w następujący sposób: najpierw podany jest wynik, otrzymany przy sumowaniu wprzód, a niżej w tej samej komórce wynik, otrzymany przy sumowaniu wstecz. *Funkcje policzone z pojedynczą precyzją:*

Funkcja dzeta Riemanna:

		n				
		50	100	200	500	1000
s	2	1.6251329	1.634984	1.6399467	1.642936	1.6439348
		1.6251328	1.6349839	1.6399465	1.642936	1.6439345
	3.6667	1.1093994	1.1094086	1.1094086	1.1094086	1.1094086
		1.1093998	1.1094089	1.1094103	1.1094105	1.1094105
	5	1.0369275	1.0369275	1.0369275	1.0369275	1.0369275
		1.0369277	1.0369277	1.0369277	1.0369277	1.0369277
	7.2	1.0072277	1.0072277	1.0072277	1.0072277	1.0072277
		1.0072277	1.0072277	1.0072277	1.0072277	1.0072277
	10	1.0009946	1.0009946	1.0009946	1.0009946	1.0009946
		1.0009946	1.0009946	1.0009946	1.0009946	1.0009946

Funkcja eta Dirichleta:

		n				
		50	100	200	500	1000
s	2	0.822271	0.8224175	0.8224547	0.82246536	0.82246685
		0.82227105	0.8224175	0.8224546	0.82246506	0.8224665
	3.6667	0.93469304	0.9346932	0.9346932	0.9346932	0.9346932
		0.93469304	0.93469334	0.93469334	0.93469334	0.93469334
	5	0.9721198	0.9721198	0.9721198	0.9721198	0.9721198
		0.97211975	0.97211975	0.97211975	0.97211975	0.97211975
	7.2	0.99352705	0.99352705	0.99352705	0.99352705	0.99352705
		0.993527	0.993527	0.993527	0.993527	0.993527
	10	0.99903953	0.99903953	0.99903953	0.99903953	0.99903953
		0.99903953	0.99903953	0.99903953	0.99903953	0.99903953

Funkcje policzone z podwójną precyzją:

Funkcja dzeta Riemanna:

		n		
		50	100	200
s	2	1.625132733621529	1.6349839001848923	1.6399465460149971
		1.6251327336215293	1.634983900184893	1.6399465460149973
	3.6667	1.1093997551541945	1.1094087973421474	1.1094102423332313
		1.1093997551541943	1.1094087973421476	1.109410242333231
	5	1.036927716716712	1.0369277526929555	1.0369277549886775
		1.0369277167167108	1.0369277526929532	1.036927754988676
	7.2	1.0072276664762816	1.007227666480654	1.0072276664807145
		1.0072276664762823	1.007227666480655	1.0072276664807163
	10	1.0009945751278182	1.0009945751278182	1.0009945751278182
		1.000994575127818	1.000994575127818	1.000994575127818

		n	
		500	1000
s	2	1.642936065514894	1.6439345666815615
		1.6429360655148941	1.6439345666815597
	3.6667	1.1094104908440712	1.1094105108423578
		1.1094104908440725	1.1094105108423593
	5	1.0369277551393863	1.0369277551431222
		1.0369277551393858	1.0369277551431204
	7.2	1.0072276664807145	1.0072276664807145
		1.0072276664807172	1.0072276664807172
	10	1.0009945751278182	1.0009945751278182
		1.000994575127818	1.000994575127818

Funkcja eta Dirichleta:

		n		
		50	100	200
s	2	0.8222710318260295	0.8224175333741286	0.822454595922551
		0.8222710318260289	0.8224175333741282	0.8224545959225509
	3.6667	0.9346930600307106	0.9346933211400662	0.9346933421086845
		0.934693060030711	0.934693321140067	0.9346933421086852
	5	0.9721197689267979	0.9721197703981592	0.972119770445367
		0.9721197689267976	0.9721197703981589	0.9721197704453663
	7.2	0.9935270006613486	0.9935270006616185	0.9935270006616201
		0.9935270006613481	0.9935270006616179	0.9935270006616198
	10	0.9990395075982718	0.9990395075982718	0.9990395075982718
		0.9990395075982715	0.9990395075982715	0.9990395075982715

		n	
		500	1000
s	2	0.8224650374240963 0.8224650374240972	0.8224665339241114 0.8224665339241127
	3.6667	0.9346933438558745 0.934693343855875	0.9346933439141353 0.9346933439141354
	5	0.9721197704468947 0.9721197704468933	0.9721197704469091 0.9721197704469088
	7.2	0.9935270006616201 0.9935270006616198	0.9935270006616201 0.9935270006616198
	10	0.9990395075982718 0.9990395075982715	0.9990395075982718 0.9990395075982715

Zadanie 4 Błędy zaokrągleń i odwzorowanie logistyczne

Niżej podajemy kod programu w języku Julia, który rysuje diagram bifurkacyjny.

using Plots

```
@everywhere function almostequal(a::Float64, b::Float64, epsilon::Float64)
    return abs(a-b) <= epsilon
end
```

```
function calcarr(xa::Float64, p::Int64, start = 0.0, fin = 4.0)
    lambdaarr = collect(linspace(start, fin, p))
    const iterconst = 50000
    const epsilon = 1e-6
    xarr = Array{Array{Float64,1},1}(p)
    yarr = Array{Array{Float64,1},1}(p)
    for j = 1:p
        lambda = lambdaarr[j]
        x = xa
        xarr[j] = Array{Float64,1}(0)
        yarr[j] = Array{Float64,1}(0)
        for i = 1:iterconst
            x = lambda*x*(1-x)
        end
        push!(yarr[j], x)
        found = false
        while !found
            x = lambda*x*(1-x)
            for y in yarr[j]
                if almostequal(x, y, epsilon)
                    found = true
                    break
                end
            end
        end
        if !found
            push!(yarr[j], x) # if not, add it to the array
        end
    end
end
```

```

    push!(xarr[j], lambda)
end
return xarr, yarr
end

function plotdiag(x, y)
    pyplot()
    verts = [(0, 0.1), (0, 0)]
    scatter(x, y, marker = (Shape(verts), 1, RGBA(0,0,0,0)), leg = false)
end

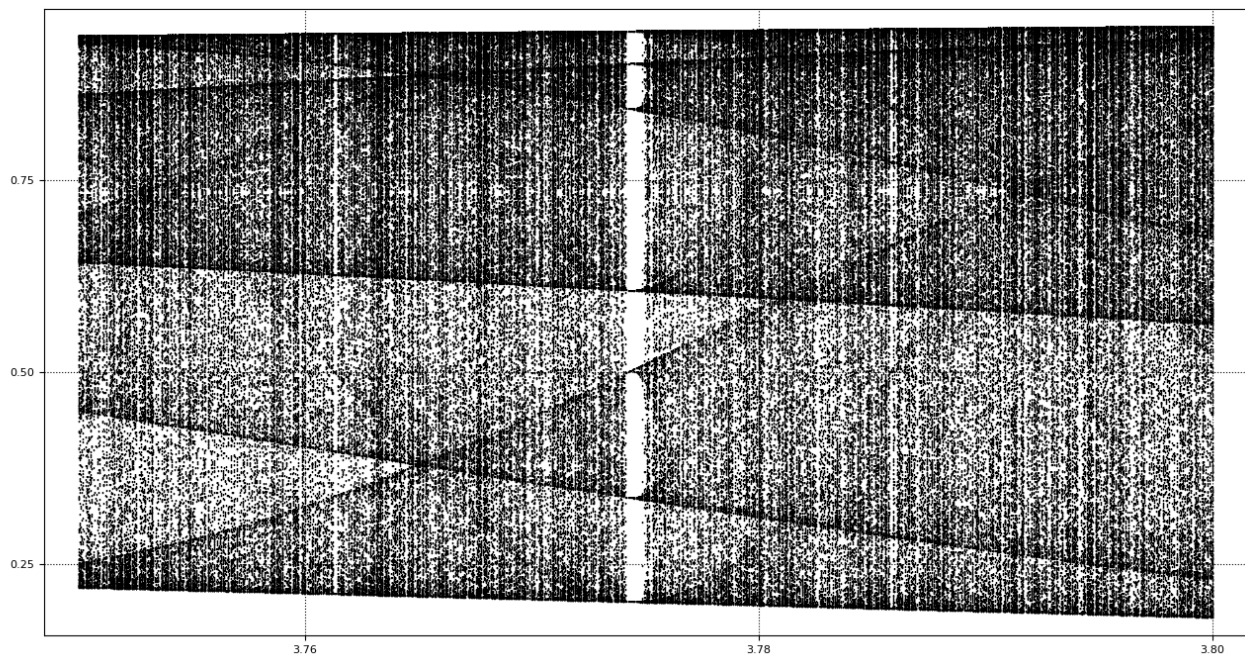
x,y = calcarr(0.5, p, 2.75)
plotdiag(x, y)

```

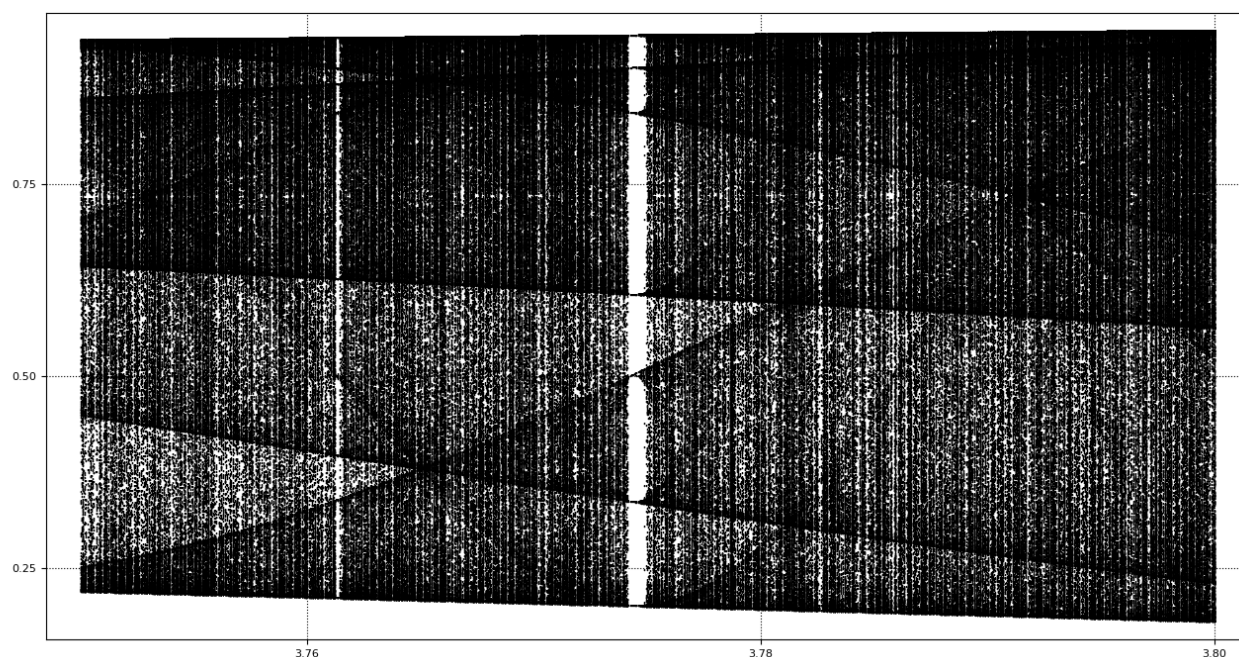
Diagram bifurkacji:

Dalej podajemy dwie części diagramu dla $x_0 = 0.5$ oraz $r = 4$.

Użyto pojedynczej precyzji:



Użyto podwójnej precyzji:



Jak widzimy, diagram, który został wygenerowany przy użyciu podwójnej precyzji wygląda 'gęściej'. Algorytm, który generuje dany diagram, najpierw szuka pierwszego punktu, gdzie funkcja zbiega, a kolejne otrzymuje, porównując do już otrzymanych. Kiedy używamy pojedynczej precyzji, nie możemy wykryć niektórych punktów, bo nie możemy rozróżnić dwóch bardzo bliskich siebie punktów. Dla tego używając l.z. z podwójną precyzją dostajemy więcej punktów.