Agnieszka Dutka

Metody Obliczeniowe w Nauce i Technice

Laboratorium 1
Arytmetyka komputerowa

Sprawozdanie

Zadanie 1 – Sumowanie liczb pojedynczej precyzji

1.1 prosta iteracja

Kod programu:

```
const int N = 10000000;
float v = 0.53125;
//float v = 0.236589;
vector<float> vct(N, v);

float easy_sum() {
    float sum = 0;
    for (auto val:vct)
        sum += val;
    return sum;
}
```

1.2 – 1.3 błąd względny i bezwzględny

Wykorzystując powyższą funkcję uzyskano następujące wyniki dla podanych danych

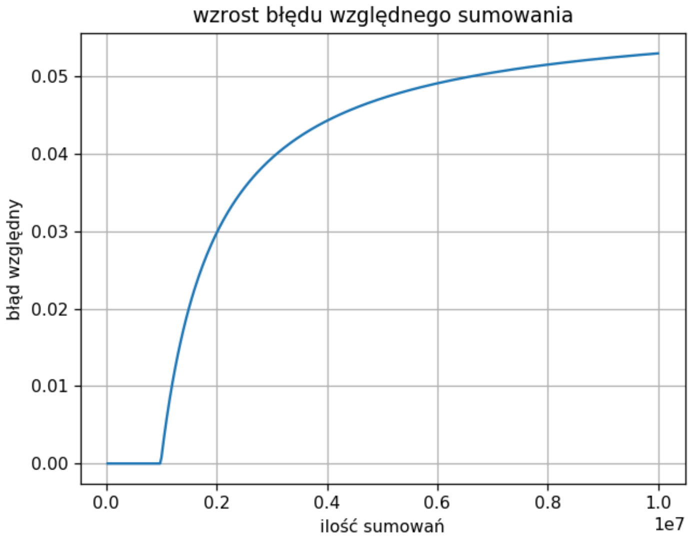
(N= 10^7, v =0.53125):

```
______Sum obtained using naive algorithm_____
experimentally determined x: 5.03084e+06
relative error: 0.0530183
absolute error: 281660
elapsed time: 26954[mikroseconds]
```

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

```
______Sum obtained using naive algorithm_____
experimentally determined x: 2.23183e+06
relative error: 0.0756357
absolute error: 156936
elapsed time: 26818[mikroseconds]
```

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 10^6 iteracji).




```
float *easy_sum_with_report(int step) {
    auto *rel_err = new float[N / step];
    float sum = 0;
    for (int i = 0; i < N / step; i++) {
        for (int j = 0; j < step; j++)
            sum += v;
        float expected = v * (i + 1) * step;
        float actual = sum;
        rel_err[i] = abs((actual - expected) / expected);
    }
    return rel_err;
}
```

Kod generujący powyższy wykres.

1.4- 1.5 rekurencyjny algorytm sumowania

Kod programu wykonujący sumowanie rekurencyjnie:

```
vector<float> vct(N, v);
float merge_sum_body(int start, int end) {
    if (start == end) return 0;
    if (start == end - 1) return vct[start];
    int mid = start + (end - start) / 2;
    return merge_sum_body(start, mid) + merge_sum_body(mid, end);
}

float merge_sum() {
    return merge_sum_body(0, vct.size());
}
```

Wyniki dla danych jak z pkt. 1.1

(N= 10^7, v =0.53125):

```
______Sum obtained using recursive algorithm_____
experimentally determined x: 5.3125e+06
relative error: 0
absolute error: 0
elapsed time: 450129[mikroseconds]
```

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

```
______Sum obtained using recursive algorithm_____
experimentally determined x: 2.07489e+06
relative error: 1.20488e-07
absolute error: 0.25
elapsed time: 451903[mikroseconds]
```

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

```
______naive algorithm_____ given x: 0.207489
elapsed time: 27354[mikroseconds]

______recursive algorithm_____ given x: 0.207489
elapsed time: 449712[mikroseconds]
```

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:

```
float kahan_sum() {
    float sum = 0.0f;
    float err = 0.0f;
    for (float i : vct) {
        float y = i - err;
        float temp = sum + y;
        err = (temp - sum) - y;
        sum = temp;
    }
    return sum;
}
```

2.1 – 2.2 błąd względny i bezwzględny

Dla danych wejściowych jak w zad. 1:

(N= 10^7, v =0.53125):

```
______Sum obtained using kahan algorithm_____
experimentally determined x: 5.3125e+06
relative error: 0
absolute error: 0
elapsed time: 92430[mikroseconds]
```

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

```
______Sum obtained using kahan algorithm_____
experimentally determined x: 2.07489e+06
relative error: 0
absolute error: 0
elapsed time: 92105[mikroseconds]
```

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:

```
______naive algorithm_____ given x: 0.207489
elapsed time: 26868[mikroseconds]

______recursive algorithm_____ given x: 0.207489
elapsed time: 445628[mikroseconds]

______kahan algorithm_____ given x: 0.207489
elapsed time: 91890[mikroseconds]
```

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:

```cpp
template <typename T>
// Riemann zeta function counted in ascending order
T zeta_fwd(T s, int n)
{
    T res = 0.0f;
    for(int k=1; k<=n; k++){
        res = res + (float)1/(pow(k, s));
    }
    return res;
}

template <typename T>
// Dirichlet eta function counted in ascending order
T eta_fwd(T s, int n){
    T res = 0.0;
    for(int k=1;k<=n;k++){
        T next = (T)pow(-1, k-1)/(T)pow(k, s);
        res = res+next;
    }
    return res;
}
```

```cpp
template <typename T>
// Riemann zeta function counted in descending order
T zeta_bwd(T s, int n) {
    T res = 0.0;
    for(int k=n; k>=1; k--){
        res = res+ (float)1/(pow(k, s));
    }
    return res;
}

template <typename T>
// Dirichlet eta function counted in descending order
T eta_bwd(T s, int n){
    T res = 0.0;
    for(int k=n;k>=1;k--){
        T next = (T)pow(-1, k-1)/(T)pow(k, s);
        res = res+next;
    }
    return res;
}
```

Agnieszka Dutka

Wyniki działania dla zadanych danych:

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

```
____________Results for s= 2__________
____Results for n= 50 ____
zeta forward: 1.6251329
zeta backward: 1.6251328
____Results for n= 100 ____
zeta forward: 1.634984
zeta backward: 1.6349839
____Results for n= 200 ____
zeta forward: 1.6399467
zeta backward: 1.6399465
____Results for n= 500 ____
zeta forward: 1.642936
zeta backward: 1.642936
____Results for n= 1000 ____
zeta forward: 1.6439348
zeta backward: 1.6439345

____________Results for s= 3.6666999______
____Results for n= 50 ____
zeta forward: 1.1093994
zeta backward: 1.1093998
____Results for n= 100 ____
zeta forward: 1.1094086
zeta backward: 1.1094089
____Results for n= 200 ____
zeta forward: 1.1094086
zeta backward: 1.1094103
____Results for n= 500 ____
zeta forward: 1.1094086
zeta backward: 1.1094105
____Results for n= 1000 ____
zeta forward: 1.1094086
zeta backward: 1.1094105
```

```
____________Results for s= 2__________
____Results for n= 50 ____
zeta forward: 1.6251327336215
zeta backward: 1.6251327336215
____Results for n= 100 ____
zeta forward: 1.6349839001849
zeta backward: 1.6349839001849
____Results for n= 200 ____
zeta forward: 1.639946546015
zeta backward: 1.639946546015
____Results for n= 500 ____
zeta forward: 1.6429360655149
zeta backward: 1.6429360655149
____Results for n= 1000 ____
zeta forward: 1.6439345666816
zeta backward: 1.6439345666816

____________Results for s= 3.666699886322___
____Results for n= 50 ____
zeta forward: 1.1093997659869
zeta backward: 1.1093997659869
____Results for n= 100 ____
zeta forward: 1.1094088081791
zeta backward: 1.1094088081791
____Results for n= 200 ____
zeta forward: 1.109410253171
zeta backward: 1.109410253171
____Results for n= 500 ____
zeta forward: 1.109410501682
zeta backward: 1.109410501682
____Results for n= 1000 ____
zeta forward: 1.1094105216803
zeta backward: 1.1094105216803
```

```
____________Results for s= 5__________
____Results for n= 50 ____
zeta forward: 1.0369275
zeta backward: 1.0369277
____Results for n= 100 ____
zeta forward: 1.0369275
zeta backward: 1.0369277
____Results for n= 200 ____
zeta forward: 1.0369275
zeta backward: 1.0369277
____Results for n= 500 ____
zeta forward: 1.0369275
zeta backward: 1.0369277
____Results for n= 1000 ____
zeta forward: 1.0369275
zeta backward: 1.0369277

____________Results for s= 7.1999998___
____Results for n= 50 ____
zeta forward: 1.0072277
zeta backward: 1.0072277
____Results for n= 100 ____
zeta forward: 1.0072277
zeta backward: 1.0072277
____Results for n= 200 ____
zeta forward: 1.0072277
zeta backward: 1.0072277
____Results for n= 500 ____
zeta forward: 1.0072277
zeta backward: 1.0072277
____Results for n= 1000 ____
zeta forward: 1.0072277
zeta backward: 1.0072277
```

```
____________Results for s= 5__________
____Results for n= 50 ____
zeta forward: 1.0369277167167
zeta backward: 1.0369277167167
____Results for n= 100 ____
zeta forward: 1.036927752693
zeta backward: 1.036927752693
____Results for n= 200 ____
zeta forward: 1.0369277549887
zeta backward: 1.0369277549887
____Results for n= 500 ____
zeta forward: 1.0369277551394
zeta backward: 1.0369277551394
____Results for n= 1000 ____
zeta forward: 1.0369277551431
zeta backward: 1.0369277551431

____________Results for s= 7.1999998092651__
____Results for n= 50 ____
zeta forward: 1.0072276674688
zeta backward: 1.0072276674688
____Results for n= 100 ____
zeta forward: 1.0072276674732
zeta backward: 1.0072276674732
____Results for n= 200 ____
zeta forward: 1.0072276674733
zeta backward: 1.0072276674733
____Results for n= 500 ____
zeta forward: 1.0072276674733
zeta backward: 1.0072276674733
____Results for n= 1000 ____
zeta forward: 1.0072276674733
zeta backward: 1.0072276674733
```

Agnieszka Dutka

```
____________Results for s= 10_________
____Results for n= 50 ____
zeta forward: 1.0009946
zeta backward: 1.0009946
____Results for n= 100 ____
zeta forward: 1.0009946
zeta backward: 1.0009946
____Results for n= 200 ____
zeta forward: 1.0009946
zeta backward: 1.0009946
____Results for n= 500 ____
zeta forward: 1.0009946
zeta backward: 1.0009946
____Results for n= 1000 ____
zeta forward: 1.0009946
zeta backward: 1.0009946
```

```
____________Results for s= 10__________
____Results for n= 50 ____
zeta forward: 1.0009945751278
zeta backward: 1.0009945751278
____Results for n= 100 ____
zeta forward: 1.0009945751278
zeta backward: 1.0009945751278
____Results for n= 200 ____
zeta forward: 1.0009945751278
zeta backward: 1.0009945751278
____Results for n= 500 ____
zeta forward: 1.0009945751278
zeta backward: 1.0009945751278
____Results for n= 1000 ____
zeta forward: 1.0009945751278
zeta backward: 1.0009945751278
```

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

```
____________Results for s= 2________
____Results for n= 50 ____
eta forward: 0.822271
eta backward: 0.822271
____Results for n= 100 ____
eta forward: 0.8224175
eta backward: 0.8224175
____Results for n= 200 ____
eta forward: 0.8224547
eta backward: 0.8224546
____Results for n= 500 ____
eta forward: 0.8224654
eta backward: 0.8224651
____Results for n= 1000 ____
eta forward: 0.8224669
eta backward: 0.8224666
____________Results for s= 3.6667___
____Results for n= 50 ____
eta forward: 0.9346931
eta backward: 0.934693
____Results for n= 100 ____
eta forward: 0.9346933
eta backward: 0.9346933
____Results for n= 200 ____
eta forward: 0.9346933
eta backward: 0.9346933
____Results for n= 500 ____
eta forward: 0.9346933
eta backward: 0.9346933
____Results for n= 1000 ____
eta forward: 0.9346933
eta backward: 0.9346933
```

```
____________Results for s= 2__________
____Results for n= 50 ____
eta forward: 0.822271031826
eta backward: 0.822271031826
____Results for n= 100 ____
eta forward: 0.822417533374
eta backward: 0.822417533374
____Results for n= 200 ____
eta forward: 0.822454595923
eta backward: 0.822454595923
____Results for n= 500 ____
eta forward: 0.822465037424
eta backward: 0.822465037424
____Results for n= 1000 ____
eta forward: 0.822466533924
eta backward: 0.822466533924
____________Results for s= 3.66669988632_
____Results for n= 50 ____
eta forward: 0.934693055395
eta backward: 0.934693055395
____Results for n= 100 ____
eta forward: 0.934693316504
eta backward: 0.934693316504
____Results for n= 200 ____
eta forward: 0.934693337473
eta backward: 0.934693337473
____Results for n= 500 ____
eta forward: 0.93469333922
eta backward: 0.93469333922
____Results for n= 1000 ____
eta forward: 0.934693339278
eta backward: 0.934693339278
```

Agnieszka Dutka

```
____________Results for s= 5_____
____Results for n= 50 ____
eta forward: 0.9721198
eta backward: 0.9721197
____Results for n= 100 ____
eta forward: 0.9721198
eta backward: 0.9721197
____Results for n= 200 ____
eta forward: 0.9721198
eta backward: 0.9721197
____Results for n= 500 ____
eta forward: 0.9721198
eta backward: 0.9721197
____Results for n= 1000 ____
eta forward: 0.9721198
eta backward: 0.9721197
____________Results for s= 7.2____
____Results for n= 50 ____
eta forward: 0.9935271
eta backward: 0.993527
____Results for n= 100 ____
eta forward: 0.9935271
eta backward: 0.993527
____Results for n= 200 ____
eta forward: 0.9935271
eta backward: 0.993527
____Results for n= 500 ____
eta forward: 0.9935271
eta backward: 0.993527
____Results for n= 1000 ____
eta forward: 0.9935271
eta backward: 0.993527
```

```
____________Results for s= 5__________
____Results for n= 50 ____
eta forward: 0.972119768927
eta backward: 0.972119768927
____Results for n= 100 ____
eta forward: 0.972119770398
eta backward: 0.972119770398
____Results for n= 200 ____
eta forward: 0.972119770445
eta backward: 0.972119770445
____Results for n= 500 ____
eta forward: 0.972119770447
eta backward: 0.972119770447
____Results for n= 1000 ____
eta forward: 0.972119770447
eta backward: 0.972119770447
____________Results for s= 7.19999980927__
____Results for n= 50 ____
eta forward: 0.993526999829
eta backward: 0.993526999829
____Results for n= 100 ____
eta forward: 0.993526999829
eta backward: 0.993526999829
____Results for n= 200 ____
eta forward: 0.993526999829
eta backward: 0.993526999829
____Results for n= 500 ____
eta forward: 0.993526999829
eta backward: 0.993526999829
____Results for n= 1000 ____
eta forward: 0.993526999829
eta backward: 0.993526999829
```

```
____________Results for s= 10________
____Results for n= 50 ____
eta forward: 0.9990395
eta backward: 0.9990395
____Results for n= 100 ____
eta forward: 0.9990395
eta backward: 0.9990395
____Results for n= 200 ____
eta forward: 0.9990395
eta backward: 0.9990395
____Results for n= 500 ____
eta forward: 0.9990395
eta backward: 0.9990395
____Results for n= 1000 ____
eta forward: 0.9990395
eta backward: 0.9990395
Process finished with exit code 0
```

```
____________Results for s= 10__
____Results for n= 50 ____
eta forward: 0.999039507598
eta backward: 0.999039507598
____Results for n= 100 ____
eta forward: 0.999039507598
eta backward: 0.999039507598
____Results for n= 200 ____
eta forward: 0.999039507598
eta backward: 0.999039507598
____Results for n= 500 ____
eta forward: 0.999039507598
eta backward: 0.999039507598
____Results for n= 1000 ____
eta forward: 0.999039507598
eta backward: 0.999039507598
```

Wartości dokładne:

$\zeta(2) = 1.64493406684822643647$ $\eta(2) = 0.822467033424113218236$

$\zeta(3.6667) = 1.10941051458645335745$ $\eta(3.6667) = 0.934693343919125072926$

$\zeta(5) = 1.03692775514336992633$ $\eta(5) = 0.972119770446909305935$

$\zeta(7.2) = 1.00722766648071711473$ $\eta(7.2) = 0.993527000661619787574$

$\zeta(10) = 1.00099457512781808533$ $\eta(10) = 0.999039507598271565639$

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 oprócz dodawania 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. „catastrophic cancellation”– 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:

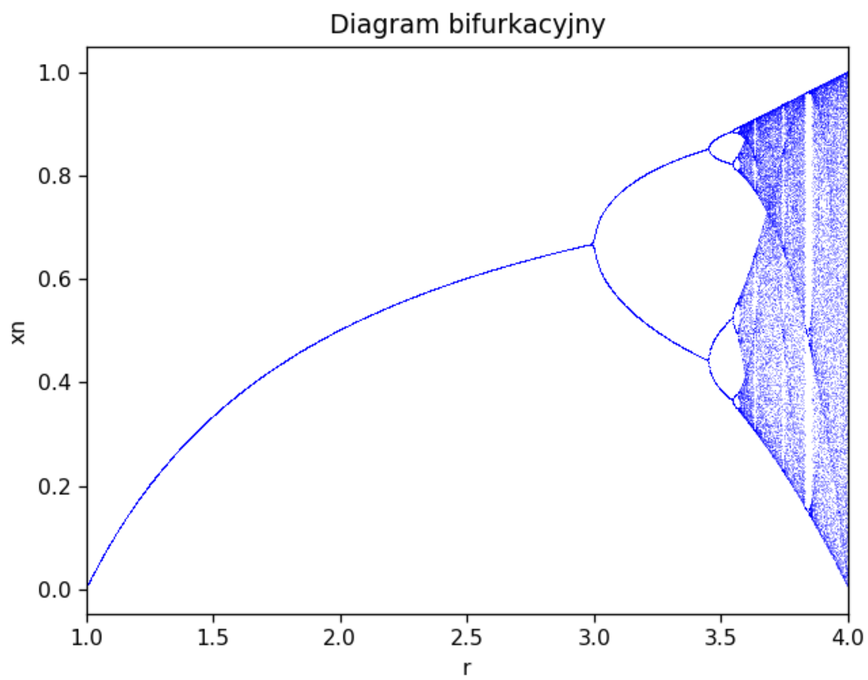
W przypadku działania na liczbach o podwójnej precyzji, uzyskane wyniki są niezależne od sposobu sumowania. Dla s<7.2 wyniki są dokładniejsze dla coraz większych n, a w przypadku s>=7.2 bardzo dobre oszacowanie dostajemy już przy n=50. Dwukrotne zwiększenie precyzji znacznie zwiększyło dokładność obliczeń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

Zadanie 4 -Błędy zaokrągleń i odwzorowanie logistyczne

4.1 Diagram bifurkacyjny

Diagram bifurkacyjny dla (1 <= r <= 4) i wybranego x_0=0.6 (kształt diagramu jest niezależny od wybranych x0)

Agnieszka Dutka



Kod programu (napisany w pythonie dla wygodniejszej wizualizacji):

```python
def logistic_map(r, x):
    return r*x*(1-x)
```

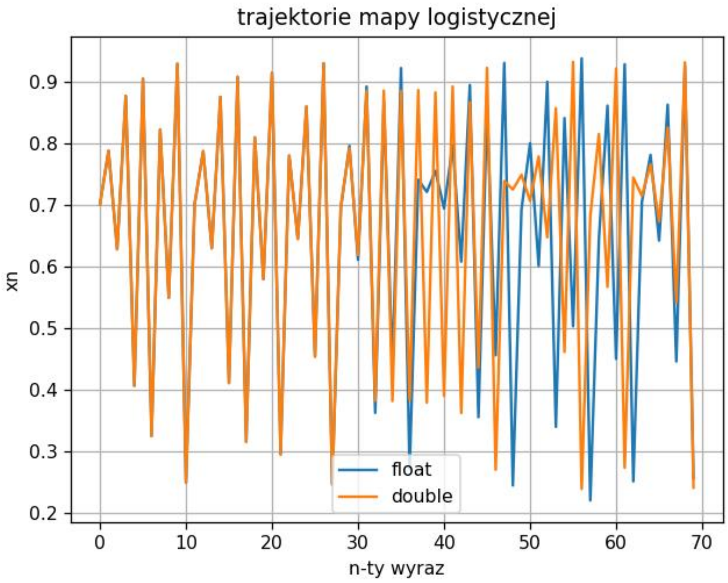
```python
def plot_bifurcation(x_density, r_range, y_density, x):
    r = np.linspace(r_range[0], r_range[1], x_density)

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

    skip = 300
    for i in range(skip):
        x = logistic_map(r, x)

    for i in range(y_density):
        x = logistic_map(r, x)
        ax.plot(r, x, ',b', alpha=.3)

    ax.set_xlim(r_range[0], r_range[1])
    ax.set_title("Diagram bifurkacyjny")
    plt.xlabel("r")
    plt.ylabel("xn")
```

Z wygenerowanego wykresu widać, że granica, do której dąży x_n nie różni się znacząco dla różnych wartości początkowych x_0, jest zależna za to od parametru r. Dla 1<= r<3 funkcja dąży do jednej wartości, następnie oscyluje między 2 określonymi wartościami, oraz zaobserwować możemy momenty chaosu – funkcja nie dąży do żadnych konkretnych wartości.

4.2 Porównanie trajektorii

```cpp
template <class T>
//// calculate trajectory of first @res_len iterations
void logistic_map_trajectory(T x0, T r, T *res, int res_len){
    res[0]=x0;
    for(int i=1; i<res_len; i++){
        res[i] = res[i-1]*r*(1-res[i-1]);
    }
}
```

```
void compare_trajectories(float x0, float r, int N){
    auto *t_float = new float[N];
    logistic_map_trajectory<float>(x0, r, t_float, N);
    auto *t_double = new double[N];
    logistic_map_trajectory<double>((double)x0, (double)r, t_double, N);
    int *xs = new int[N];
    for(int i=0;i<N;i++) xs[i]=i;
    export_to_file("t_float.json", parseArray(t_float, N));
    export_to_file("t_double.json", parseArray<double>(t_double, N));
    export_to_file("t_xs.json", parseArray(xs, N));
}
```



Wykres przedstawia trajektorie pierwszych 70 iteracji x_n dla danych $x_0=0.7$ oraz $r = 3.75$ obliczonych z użyciem pojedynczej(niebieski) i podwójnej(pomarańczowy) precyzji. Początkowo obie trajektorie nakładają się, z czasem jednak różnice stają się coraz bardziej widoczne. Wynika to ze sposobu liczenia funkcji- algorytm nie jest stabilny numerycznie, gdyż dla r>=3.75 (czyli dla r>1) $|\varepsilon_{n+1}|>=|\varepsilon_n|$. Z każdym kolejnym etapem obliczeń jakość wyniku pogarsza się.

4.3 Osiąganie zera

```python
# count iterations to zero
def iter_to_zero(x, r=np.float32(4.0)):
    x = np.float32(x)
    it = 0
    while x > 0 and it < 10000:
        it += 1
        x = np.float32(logistic_map(r, x))
    if it == 10000:
        return -1
    return it

for x in np.linspace(0, 1, num=50):
        print('x0 = %.2f'%x, " iter= ",iter_to_zero(x))
```

Dla r=4 i różnych wartości x_0: (iter to liczba iteracji po której $x_n=0$)

Agnieszka Dutka

```
x0 = 0.00  iter=  0
x0 = 0.01  iter=  3339
x0 = 0.02  iter=  1533
x0 = 0.03  iter=  2612
x0 = 0.04  iter=  2560
x0 = 0.05  iter=  2064
x0 = 0.06  iter=  407
x0 = 0.07  iter=  324
x0 = 0.08  iter=  1998
x0 = 0.09  iter=  4027
x0 = 0.10  iter=  1038
x0 = 0.11  iter=  896
x0 = 0.12  iter=  1194
x0 = 0.13  iter=  -1
x0 = 0.14  iter=  1230
x0 = 0.15  iter=  2525
x0 = 0.16  iter=  2856
x0 = 0.17  iter=  1084
x0 = 0.18  iter=  1923
x0 = 0.19  iter=  1155
x0 = 0.20  iter=  617
x0 = 0.21  iter=  1991
x0 = 0.22  iter=  5
x0 = 0.23  iter=  3568
x0 = 0.24  iter=  3658
x0 = 0.25  iter=  174
x0 = 0.26  iter=  2366
x0 = 0.27  iter=  -1
x0 = 0.28  iter=  -1
x0 = 0.29  iter=  2409
x0 = 0.30  iter=  1651
x0 = 0.31  iter=  3147
x0 = 0.32  iter=  -1
x0 = 0.33  iter=  897
x0 = 0.34  iter=  2323
x0 = 0.35  iter=  3922
x0 = 0.36  iter=  3115
x0 = 0.37  iter=  1349
x0 = 0.38  iter=  1386
x0 = 0.39  iter=  759
x0 = 0.40  iter=  4339
x0 = 0.41  iter=  1250
x0 = 0.42  iter=  1297
x0 = 0.43  iter=  1244
x0 = 0.44  iter=  2014
x0 = 0.45  iter=  2030
x0 = 0.46  iter=  4063
x0 = 0.47  iter=  3243
x0 = 0.48  iter=  910
x0 = 0.49  iter=  3825
x0 = 0.51  iter=  3825
x0 = 0.52  iter=  910
x0 = 0.53  iter=  3243
x0 = 0.54  iter=  4063
x0 = 0.55  iter=  2030
x0 = 0.56  iter=  2014
x0 = 0.57  iter=  1244
x0 = 0.58  iter=  1297
x0 = 0.59  iter=  1250
x0 = 0.60  iter=  4339
x0 = 0.61  iter=  759
x0 = 0.62  iter=  1386
x0 = 0.63  iter=  1349
x0 = 0.64  iter=  3115
x0 = 0.65  iter=  3922
x0 = 0.66  iter=  2323
x0 = 0.67  iter=  1576
x0 = 0.68  iter=  729
x0 = 0.69  iter=  3147
x0 = 0.70  iter=  1651
x0 = 0.71  iter=  2409
x0 = 0.72  iter=  -1
x0 = 0.73  iter=  172
x0 = 0.74  iter=  2453
x0 = 0.75  iter=  4457
x0 = 0.76  iter=  3658
x0 = 0.77  iter=  3568
x0 = 0.78  iter=  5
x0 = 0.79  iter=  481
x0 = 0.80  iter=  3386
x0 = 0.81  iter=  1155
x0 = 0.82  iter=  1923
x0 = 0.83  iter=  1084
x0 = 0.84  iter=  2856
x0 = 0.85  iter=  2525
x0 = 0.86  iter=  1230
x0 = 0.87  iter=  1297
x0 = 0.88  iter=  1194
x0 = 0.89  iter=  896
x0 = 0.90  iter=  16
x0 = 0.91  iter=  2295
x0 = 0.92  iter=  2357
x0 = 0.93  iter=  3539
x0 = 0.94  iter=  407
x0 = 0.95  iter=  3136
x0 = 0.96  iter=  2560
x0 = 0.97  iter=  1381
x0 = 0.98  iter=  1552
x0 = 0.99  iter=  3339
x0 = 1.00  iter=  1
```

Możemy zaobserwować, że dla różnych wartości x_0 liczba iteracji potrzebna do osiągnięcia zera jest bardzo różna (wynik -1 oznacza że w ciągu 50000 iteracji 0 nie zostało osiągnięte). Odwzorowanie logistyczne dla r=4 zdaje się przyjmować największy przedział wartości z wszystkich które funkcja może przyjmować (co potwierdza się też na wykresie).