Sumowanie wyrazów ciągów za pomocą algorytmu Gilla-Möllera

```
u=0;p=0;
for i=1:n
v=i-ty wyraz sumowanego ciagu;
s=u+v;
p=u-s+v+p;
u=s;
end
z=s+p
```

Algorytm Gilla-Möllera

Opis:

Algorytm Gilla-Möllera to metoda numeryczna opracowana w celu dokładniejszego sumowania szeregów, zwłaszcza tych z szybko zanikającymi elementami. Głównym celem tego algorytmu jest minimalizacja błędów zaokrągleń, które mogą się pojawić podczas sumowania wielu małych liczb.

Historia:

Algorytm ten został opracowany przez Philippe'a Gilla i Helgi Möllera w latach 60. XX wieku. Był częścią ich pracy nad metodami numerycznymi, które miały na celu poprawę dokładności obliczeń w zastosowaniach naukowych i inżynierskich.

```
u=0;p=0;
for i=1:n
v=i-ty wyraz sumowanego ciagu;
s=u+v;
p=u-s+v+p;
u=s;
end
z=s+p
```

Algorytm Gilla-Möllera

Działanie:

Algorytm działa poprzez wprowadzenie dodatkowych zmiennych, które śledzą błędy zaokrągleń i poprawiają bieżącą sumę. Kluczowym elementem jest tutaj resztkowa suma, która jest korygowana na każdym etapie iteracji, aby uwzględnić błędy wynikające z ograniczonej precyzji arytmetyki zmiennoprzecinkowej.

Zalety:

- **Dokładność:** Zwiększa dokładność sumowania poprzez minimalizację błędów zaokrągleń.
- **Stabilność:** Jest bardziej stabilny numerycznie w porównaniu do prostego sumowania sekwencyjnego.
- **Elastyczność:** Może być stosowany do różnych typów szeregów, w tym tych z szybko zanikającymi elementami.

SUMA SZEREGU KAHANA

Algorytm Kahana jest techniką kompensowanego sumowania, która minimalizuje błędy zaokrągleń poprzez utrzymywanie dodatkowej zmiennej kompensacyjnej. Został opracowany przez Williama Kahana w 1965 roku, aby poprawić dokładność sumowania w obliczeniach numerycznych. Poprawia dokładność poprzez śledzenie błędów zaokrągleń.

```
function suma = suma_szeregu_kahana(a)
    % Inicjalizacja sumy i błędu
    suma = 0.0;
    err = 0.0;

    % Iteracja przez wszystkie elementy szeregu
    for k = 1:length(a)
        y = a(k) - err;       % Skorygowanie bieżącego wyrazu o poprzedni błąd
        t = suma + y;         % Tymczasowa suma
        err = (t - suma) - y; % Obliczenie nowego błędu
        suma = t;             % Aktualizacja sumy
    end
end
```

SUMA SZEREGU NEUMAIRA

Została opracowana jako ulepszenie i rozszerzenie algorytmu Kahana, aby poprawić jego stabilność.

```
function suma = suma_szeregu_neumaira(a)
    % Inicjalizacja sumy i błędu
    suma = 0.0;
    err = 0.0;

    % Iteracja przez wszystkie elementy szeregu
    for k = 1:length(a)
        temp = suma;
        suma = suma + a(k);
        if abs(temp) >= abs(a(k))
            err = err + ((temp - suma) + a(k));
        else
            err = err + ((a(k) - suma) + temp);
        end
    end

    suma = suma + err; % Dodanie skorygowanego błędu
end
```

SUMA SZEREGU CASCADE

Stosowany w różnych formach od lat 80., algorytm ten optymalizuje proces sumowania dużych zbiorów danych. Sumuje elementy w parach, co zmniejsza błędy zaokrągleń przy sumowaniu dużych zbiorów liczb.

```
function suma = suma_szeregu_cascade(a)
    while length(a) > 1
        n = length(a);
        m = floor(n / 2);
        a = [a(1:2:m*2-1) + a(2:2:m*2), a(m*2+1:end)];
    end
    suma = a;
end
```

SUMA SZEREGU PAIRWISE

Sumuje elementy w parach, minimalizując błędy zaokrągleń. Jest często stosowany w obliczeniach naukowych i inżynierskich od lat 90. Efektywny sposób na zmniejszenie błędów zaokrągleń.

```
function suma = suma_szeregu_pairwise(a)
    n = length(a);
    if n == 0
        suma = 0;
    elseif n == 1
        suma = a;
    else
        mid = floor(n / 2);
        left = suma_szeregu_pairwise(a(1:mid));
        right = suma_szeregu_pairwise(a(mid+1:end));
        suma = left + right;
    end
end
```

SUMA SZEREGU BLOCK

Dzieli sumowanie na bloki, minimalizując błędy zaokrągleń. Skutecznie stosowany w przetwarzaniu dużych zbiorów danych w obliczeniach równoległych.

```
function suma = suma_szeregu_block(a, blockSize)
    if nargin < 2
        blockSize = 1000; % Domyślna wielkość bloku
    end

    n = length(a);
    suma = 0;
    for i = 1:blockSize:n
        blockEnd = min(i + blockSize - 1, n);
        suma = suma + sum(a(i:blockEnd));
    end
end
```

SUMA SZEREGU FFT

Używa transformacji Fouriera do sumowania szeregu w sposób bardziej stabilny numerycznie. Transformacja Fouriera jest stosowana w analizie sygnałów od połowy XX wieku. Poprawia dokładność sumowania dla dużych zbiorów danych.

```matlab
function suma = suma_szeregu_fft(a)
    n = length(a);
    if mod(n, 2) ~= 0
        a = [a, 0]; % Dopisanie zera, jeśli liczba elementów nieparzysta
    end
    A = fft(a);
    A(2:end) = 0;
    suma = real(ifft(A) * n);
end
```

```
%% Sumowanie szeregu 5^k/k^2 dla 10 kroków
a = 5.^(1:10) ./ (1:10).^2; % Szereg 5^k/k^2 dla k od 1 do 10
```

```
suma_szeregu_kahana1 =

     1.300900368677564e+05


suma_szeregu_neumaira2 =

     1.300900368677564e+05


suma_szeregu_cascade3 =

     1.300900368677564e+05


suma_szeregu_pairwise4 =

     1.300900368677564e+05
```

```
suma_szeregu_block5 =

     1.300900368677564e+05


sumaGillaMollera6 =

     1.300900368677564e+05


sumazwykla7 =

     1.300900368677564e+05
```

```
%% sumowanie szeregu 5^k/k^2 dla 1000000 kroków
a = 5.^(1:1000000) ./ (1:1000000).^2; % Szereg 5^k/k^2 dla k od 1 do 1000000
```

```
suma_szeregu_kahana1 =

   NaN


suma_szeregu_neumaira2 =

   NaN


suma_szeregu_cascade3 =

   Inf


suma_szeregu_pairwise4 =

   Inf
```

```
suma_szeregu_block5 =

   Inf


sumaGillaMollera6 =

   NaN


sumazwykla7 =

   Inf
```

```
%% sumowanie szeregu (-1)^k*(1/(2*k+1)) dla 100000 kroków
a = (-1).^(0:100000) .*(1./(2.*(0:100000)+1)); % Szereg (-1)^k*(1/(2*k+1)) dla k od 0 do 100000
```

```
%% sumowanie szeregu (-1)^k*(1/(2*k+1)) dla 10 kroków
a = (-1).^(0:10) .*(1./(2.*(0:10)+1)); % Szereg (-1)^k*(1/(2*k+1)) dla k od 0 do 10
```

```
suma_szeregu_kahana1 =

   0.808078952351398


suma_szeregu_neumaira2 =

   0.808078952351398


suma_szeregu_cascade3 =

   0.808078952351398


suma_szeregu_pairwise4 =

   0.808078952351398
```

```
suma_szeregu_block5 =

   0.808078952351398


sumaGillaMollera6 =

   0.808078952351398


sumazwykla7 =

   0.808078952351398
```

```
suma_szeregu_kahana1 =

   0.785400663372449


suma_szeregu_neumaira2 =

   0.785400663372449


suma_szeregu_cascade3 =

   0.785400663372449


suma_szeregu_pairwise4 =

   0.785400663372449
```

```
suma_szeregu_block5 =

   0.785400663372448


sumaGillaMollera6 =

   0.785400663372449


sumazwykla7 =

   0.785400663372430
```

```
%% sumowanie szeregu 1/k! dla 10 kroków
a = 1./(factorial((1:10))); % Szereg 1/k! dla k od 1 do 10
```

```
suma_szeregu_kahana1 =

   1.718281801146385


suma_szeregu_neumaira2 =

   1.718281801146385


suma_szeregu_cascade3 =

   1.718281801146385


suma_szeregu_pairwise4 =

   1.718281801146384
```

```
suma_szeregu_block5 =

   1.718281801146385


sumaGillaMollera6 =

   1.718281801146385


sumazwykla7 =

   1.718281801146385
```

```
%% sumowanie szeregu 1/k! dla 10000 kroków
a = 1./(factorial((1:10000))); % Szereg 1/k! dla k od 1 do 10000
```

```
suma_szeregu_kahana1 =

   1.718281828459045


suma_szeregu_neumaira2 =

   1.718281828459045


suma_szeregu_cascade3 =

   1.718281828459045


suma_szeregu_pairwise4 =

   1.718281828459045
```

```
suma_szeregu_block5 =

   1.718281828459045


sumaGillaMollera6 =

   1.718281828459045


sumazwykla7 =

   1.718281828459046
```

Podsumowanie

Algorytm Gilla-Möllera spisuje się porównywalnie do pozostałych sprawdzanych przez nas algorytmów. W niektórych przypadkach wystepowały różnice na dalekich miejscach po przecinku, ale generalnie sumy były poprawne. W przypadku pierwszego przykładu (szereg rozbieżny) niektóre metody, w tym algorytm Gilla-Möllera wyprodukowały wynik NaN (Not a Number), co może świadczyć o problemach w sumowaniu szeregów zbieżnych do nieskończoności.

Dziękujemy za uwagę!