Analiza matematyczna dla informatyków.

Mieczysław Cichoń, ver. 4.2/2023

Mieczysław Cichoń - WMI UAM

Strony do lektury na wykłady

Czytamy najpierw:

[K] : motywacje - strony 21-24

i dalej

[K] : strony 189-194

[W] : strony 42-50 oraz 54-55, 57-61
(lub alternatywnie: z tego wykładu strony 38-48).

Można też: te kilka stron...

Szeregi - motywacje...

Czy należy rozważać jakieś formy "zliczania" nieskończonej ilości liczb? **Tak!** Inaczej nie moglibyśmy unikać wielu paradoksów, ale czy to oznacza "sumowanie" nieskończone? **Nie!**

Gdyby miało to być dodawanie, to miałoby jego własności, czyli musielibyśmy się zgodzić na taki paradoks:

$$0 = 0$$

$$0 = 0 + 0 + 0 + \ldots$$

$$0 = (1 - 1) + (1 - 1) + (1 - 1) + \ldots$$

$0 = 1-1+1-1+1-1\ldots$ przecież *dodawanie* jest łączne, więc możemy??

$0 = 1-(1-1)-(1-1)-\ldots$ przecież *dodawanie* jest łączne, j.w. możemy??

$0 = 1-0-0-0-\ldots$ na to chyba jest powszechna zgoda, obliczamy

$0 = 1$ z tym chyba się nie zgadzamy? $\Rightarrow$ wszystkie liczby są równe...

Uwagi.

Takie działanie nieskończone **nie może więc być sumą!** (niech się nikt nie waży tak pisać, bo dostanie: w swoim rozumieniu "5", a ja wpiszę "2" - w końcu przy takim podejściu to byłyby takie same oceny :-))

Jak się wkrótce okaże - w informatyce to m.in. znakomite narzędzie pozwalające operować na przybliżeniach liczb rzeczywistych, bez tego pojęcia nie byłoby sensownych obliczeń na komputerze.

ALE: jak już wiemy komputer nie może operować na zbiorach nieskończonych, czyli to informatyk musi pogodzić przydatność operownia szeregami z możliwościami komputera. Czyli musimy się sami tego nauczyć i poznać własności tego pojęcia...

Będą dwa główne powody stosowania szeregów w informatyce:

1) reprezentowanie wartości rzeczywistych (m.in. wartości funkcji) z dowolnie rosnącą dokładnością. Mając dana wartość (np,. liczbę niewymierną) obliczać ją z rosnącą precyzją.

2) reprezentowanie funkcji! Mając daną funkcję wskazać metodę wyliczania jej wartości w dowolnych punktach.

Pamiętając o tych 2 celach: pojawi się **kolejne** pytanie, czy mając dany szereg on reprezentuje jakąś liczbę (jako jego sumę)? Jak to sprawdzić i obliczyć?

Odpowiedź na jedno z pytań jest jasna: liczba może mieć **różne reprezentacje** w postaci szeregów. Jedno z nich to np. klasyczne jej rozwinięcie w szereg przybliżeń dziesiętnych (lub o innej podstawie), ale są też o wiele bardziej użyteczne.

Nasze cele...

1. Mamy liczbę rzeczywistą. Czy istnieje jej reprezentacja w postaci sumy szeregu? Czy taki szereg jest tylko jeden?

To da nam możliwość obliczania jej z dowolną dokładnością (w przypadku programu: osiagalną na komputerze)! Co z liczbami wymagającymi bardzo wysokiej precyzji obliczeń, np. symulacjami na komputerze wielkości bardzo małych (np. fizyka kwantowa czy informatyka kwantowa) lub bardzo dużych (astronomia)? *Tak przy okazji: komputer kwantowy ma oznaczać dla nas nie tylko szybsze obliczenia, ale i większą ich precyzję (i z tym mamy na razie problem, bo jest dość podatny na zakłócenia...).* Ile jest takich reprezentacji danej liczby? Czym się różnią?

2. Mamy dany szereg. Czy jest zbieżny, a więc czy reprezentuje pewną liczbę (swoją sumę)?

Jeśli tak, to jego sumy częściowe mogą posłużyć do obliczania wartości przybliżonej tej liczby (sumy). Jak to sprawdzić?

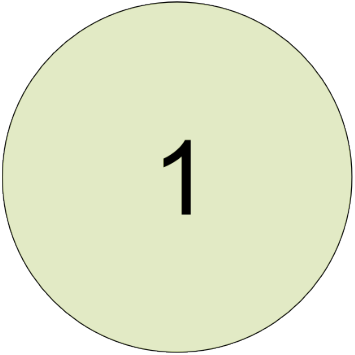
3. Jakie są sytuacje w informatyce, w których pojawi się w naturalny sposób szereg i do czego to może służyć?

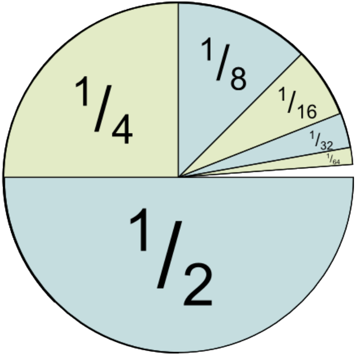
Jakie rodzaje szeregów i operacje na nich będą szczególnie istotne?

Jednym z ciekawych osiągnięć matematyki było odkrycie, że funkcje trygonometryczne nie muszą być obliczane zgodnie z zagadnieniami geometrycznymi, ale można (i jest to celowe) je wyznaczać poprzez przybliżone wartości sumy szeregu! I dotyczy to nie tylko funkcji trygonometrycznych! Jeśli nie wiemy czy szereg jest zbieżny, to nie możemy z sensownym celem wykorzystać go w obliczeniach numerycznych.

I rzeczywiście: badanie zbieżności szeregu będzie jednym z naszych zasadniczych celów.

A jednak *coś* musimy liczyć...



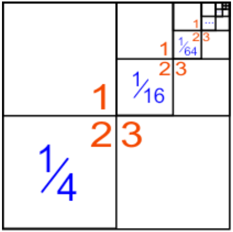


Czy "dodane" ułamki po prawej stronie nie powinny dać wyniku po lewej?

Szeregi liczbowe...

... stanowią uogólnienie sum skończonych (ale nie są wynikiem dodawania!). Niech dany będzie ciąg liczb rzeczywistych $(a_n)_{n=1}^{\infty}$ oraz niech $(s_n)_{n=1}^{\infty}$ będzie ciągiem (sum częściowych danego ciągu), którego wyrazy określone są następująco

$$
\begin{aligned}
s_1 &= a_1 \\
s_2 &= a_1 + a_2 \\
&\ldots\ldots\ldots\ldots\ldots \\
s_n &= a_1 + a_2 + \ldots + a_n \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{aligned}
$$



Szeregi liczbowe.

Jedna z możliwych definicji szeregu liczbowego mówi, że jest to para $((a_n),(s_n))$. Oznaczamy go

$$a_1 + a_2 + \ldots + a_n + \ldots \equiv \sum_{n=1}^{\infty} a_n \ .$$

Jak się za chwilę okaże nie będziemy specjalnie odróżniali odrębnym symbolem szeregu od jego sumu: bardziej interesować nas będzie to co nazwiemy **sumą szeregu.** Widzimy więc, że jeśli obliczamy sumę szeregu to w rzeczywistości danemu ciągowi (a_n) przyporządkowujemy liczbę rzeczywistą $s = \lim_{n\to\infty} s_n = \sum_{n=1}^{\infty} a_n$, czyli szereg to funkcjonał, który oznaczamy

$$\sum_{n=1}^{\infty} : \ (a_n) \longrightarrow s = \sum_{n=1}^{\infty} a_n \ .$$

Mówimy, że szereg $\sum_{n=1}^{\infty} a_n$, jest **zbieżny** do skończonej liczby s, lub, że ma sumę równą s, co zapisujemy

$$\sum_{n=1}^{\infty} a_n = s = \lim_{n\to\infty} s_n,$$

jeżeli ciąg sum częściowych (s_n) jest **zbieżny** do granicy s, przy $n \longrightarrow \infty$. Jeżeli ciąg sum częściowych (s_n) nie jest zbieżny, to szereg nazywamy **rozbieżnym**.

Warunek konieczny zbieżności szeregu.

Twierdzenie. *Szereg* $\sum_{n=1}^{\infty} a_n$ *może być zbieżny tylko wtedy, gdy ciąg jego wyrazów* (a_n) *dąży do zera, przy* $n \to \infty$*, tj. gdy*

$$\lim_{n\to\infty} a_n = 0 \ .$$

Zauważmy, że warunek podany w powyższym twierdzeniu jest **warunkiem koniecznym** zbieżności szeregu. Aby się przekonać, że nie jest to warunek dostateczny wystarczy rozważyć szereg harmoniczny $\sum_{n=1}^{\infty} \frac{1}{n}$.

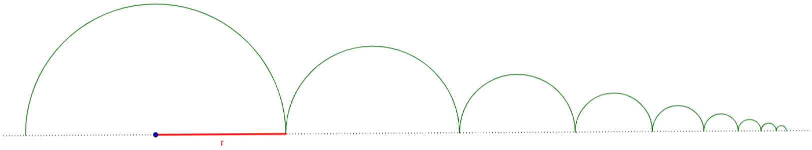
Wyraz ogólny szeregu harmonicznego spełnia warunek podany w powyższym twierdzeniu, jednak - jak pokażemy później - szereg ten nie jest zbieżny.

Uwaga: symulacje komputerowe wcale nam nie pomogą. Skrypt ilustracyjny szeregu harmonicznego w "Mathematica" - potrzebny darmowy *CDF Player* lub *Mathematica*.

Szereg geometryczny.

Dla wszystkich: określić, czy łączna długość łuku krzywej jak na rysunku (proces kontynuujemy w nieskończoność, promień kolejego okręgu stanowi 2/3 długości promienia poprzedniego) jest skończona? Jeśli tak, to wyrazić zadanie poprzez szereg i obliczyć jego sumę.

A dla chętnych: napisać taki program (w Pythonie)...



W informatyce...

Tego jeszcze nie wprowadziliśmy, ale warto już teraz pamiętać, że to szeregi będą podstawą do obliczeń zarówno wartości funkcji w punktach, jak i wartości przybliżonych liczb rzeczywistych - por. przybliżanie π w tekście [K]... Tak będzie najwygodniej reprezentować liczby niewymierne - zamiast kumulować błąd można wielkość przybliżać raz - na końcu, a wcześniej operować wzorem na wyraz ogólny szeregu a_n.

I tu uwaga: nadal nie ma "najlepszego" wzoru na obliczanie komputerowe liczby poprzez szereg.

“Concrete Mathematics: A Foundation for Computer Science”

Dla przypomnienia: wcześniej pojawiła się rekurencja i ciągi rekurencyjne w informatyce. To doskonały przykład jak wszystkie omawiane działy się łączą: ciągom rekurencyjnym odpowiadają funkcje tworzące, a właściwie ich postacie zapisane jako szeregi (potęgowe).

Zainteresowani mogą znaleźć **więcej** w książce *Graham, R. L., D. E. Knuth, O. Patashnik. "Matematyka konkretna" PWN, Warszawa (2006)* (lub w oryginalnej wersji “Concrete Mathematics: A Foundation for Computer Science”): rozdziały 7 i 9. Tam jest też mnóstwo przykładów wskazujących na konieczność stosowania matematyki w informatyce (wśród autorów m.in. D. Knuth...).

Wykorzystanie szeregów w nowych algorytmach nadal jest rozwijane i tworzy się nowe algorytmy. Np. dla obliczenia liczby π bardzo dobrym algorytmem (miarą tego jest bardzo duży przyrost cyfr dokładnych wyniku uzyskiwanych w kolejnych iteracjach) jest ten oparty o

a) wzór Ramanujana:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=1}^{\infty} \frac{(4n)!(1103 + 26390n)}{(n!)^4 396^{4n}}.$$

Dla liczby π (jak każdej innej) mamy oczywiście *wiele innych* możliwości obliczeniowych m.in. za pomocą szeregów.

Oto klasyczne:

b) wzór Newtona

$$\frac{\pi}{4} = \sum_{n=1}^{\infty} \frac{(-1)^n}{2n-1},$$

c) wzór Eulera

$$\frac{\pi^2}{6} = \sum_{n=1}^{\infty} \frac{1}{n^2},$$

d) wzór Chudnovskiego (rekord świata?)

$$\frac{1}{\pi} = 12 \sum_{n=1}^{\infty} \frac{(-1)^n (6n)!(545140134n + 13591409)}{(3n)!(n!)^3 640320^{\frac{3n+3}{2}}}$$

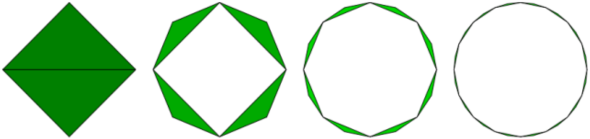
(w jakim sensie algorytm Chudnovskiego to "rekord świata": to aktualnie (?) największy przyrost dokładnie obliczonych miejsc po przecinku w jednym kroku).

Inne przykłady dla liczby π - por. materiał [K].

1.3.2 Liczba π poprzez szereg.

Powróćmy do obliczania liczby π. Widzieliśmy, że metoda oparta o przybliżanie obwodu koła daje niezbyt zadowalający wynik. Często jednak, jeśli jeden algorytm zastąpimy innym, to możemy uzyskać lepsze wyniki przy zastosowaniu tej samej arytmetyki komputerowej. Wiemy, że liczba π jest nie tylko obwodem koła o średnicy jeden, ale również polem koła o promieniu jeden.

Zastępując pole tego koła polami wpisanych w nie 2^n-kątów foremnych również otrzymamy ciąg przybliżeń liczby π. Oznaczmy przez P_n pole 2^n-kąta foremnego wpisanego w koło. Dla $n = 1$ rozważamy $2-kt$ jako zdegenerowaną, pozbawioną pola figurę redukującą się do średnicy koła, zatem $P_1 = 0$.



Rysunek 1.8: Kolejne dodatki do pola koła.

Zauważmy, że 2^{n+1}-kąt powstaje z 2^n-kąta poprzez dorysowanie figury złożonej z 2^n trójkątów równoramiennych wpisanych w koło o podstawach leżących na bokach 2^n-kąta (patrz rys. 1.8). Niech A_n oznacza pole tej figury. Zatem Zatem

$$P_{n+1} = P_n + A_n,$$

czyli A_{n+1} jest poprawką, która dodana do pola 2^n-kąta daje pole 2^{n+1}-kąta. Stąd

$$P_n = A_1 + A_2 + A_3 + A_4 + \ldots + A_{n-1},$$

a pole koła P_K otrzymamy sumując wszystkie poprawki:

$$P_K = A_1 + A_2 + A_3 + A_4 + \ldots .$$

Wartości A_i można wyliczyć. Aby wykorzystać wzory z podrozdziału 1.2.5, zamiast koła o promieniu jeden rozważymy ponownie koło o średnicy jeden, a więc o promieniu $1/2$. Zatem ze wzoru (1.3) jego pole to $\frac{\pi}{4}$. Zatem jako przybliżenie π interesuje nas ciąg $s_n := 4P_n$. Niech T_n oznacza pole trójkąta równoramiennego o podstawie równej bokowi 2^n-kąta, a ramionach równych bokowi 2^{n+1}-kąta. Mamy

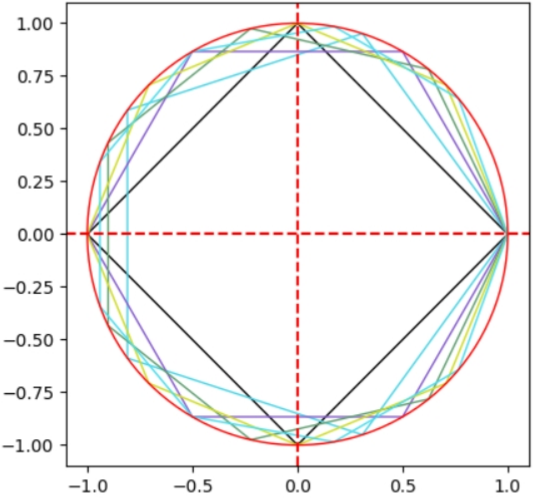
$$A_n = 2^n T_n$$

Można policzyć, że $T_n = \frac{a_n(1-\sqrt{1-a_n^2})}{4}$, gdzie a_n, jak poprzednio, oznacza bok 2^n-kąta foremnego wpisanego w koło o średnicy jeden. Zatem

$$P_{n+1} = P_n + 2^n \frac{a_n(1-\sqrt{1-a_n^2})}{4}$$

oraz

$$s_{n+1} = s_n + 2^n a_n(1-\sqrt{1-a_n^2}). \tag{1.4}$$




```
s[ 2] =  2.000000000000000
s[ 3] =  2.828427124746190
s[ 4] =  3.061467458920718
s[ 5] =  3.121445152258052
s[ 6] =  3.136548490545939
s[ 7] =  3.140331156954753
s[ 8] =  3.141277250932773
s[ 9] =  3.141513801144301
s[10] =  3.141572940367092
s[11] =  3.141587725277160
s[12] =  3.141591421511200
s[13] =  3.141592345570118
s[14] =  3.141592576584873
s[15] =  3.141592634338564
s[16] =  3.141592648776986
s[17] =  3.141592652386592
s[18] =  3.141592653288993
s[19] =  3.141592653514594
s[20] =  3.141592653570994
s[21] =  3.141592653585094
s[22] =  3.141592653588619
s[23] =  3.141592653589501
s[24] =  3.141592653589721
s[25] =  3.141592653589776
s[26] =  3.141592653589790
s[27] =  3.141592653589794
s[28] =  3.141592653589794
s[29] =  3.141592653589795
s[30] =  3.141592653589795
s[31] =  3.141592653589795
s[32] =  3.141592653589795
s[33] =  3.141592653589795
s[34] =  3.141592653589795
```

Liczba π.

W rozdziale 1.3.2 zaprezentowaliśmy metodę przybliżania liczby π poprzez pola wielokątów wpisanych w koło i zauważyliśmy, że stosowny ciąg (1.4) jest szeregiem. Przyjrzyjmy się teraz zachowaniu tego szeregu od strony numerycznej Program wyznaczający wyrazy ciągu (1.4) w Mathematica jest przedstawiony na listingu 12.1, a w języku C++ na listingu 12.2

Uruchamiając program łatwo sprawdzić, że metoda oparta o pola przy zastosowaniu tego samego typu **double** *daje znacznie dokładniejsze wyniki niż metoda poprzednia, bo już 26-ta suma częsciowa daje wartość*

$$s_{26} = 3.141592653589793,$$

która jest poprawnym przybliżeniem liczby π do ostatniej cyfry włącznie.

W informatyce I.

Szeregi zastępują ciągi w wielu algorytmach. **Dlaczego?**
Po prostu - to często "lepsze" algorytmy (czyli: szybciej zbieżne).

Prosty przykład: liczba e. Jeżeli obliczamy ją z definicji jako granicę ciągu $a_n = \left(1 + \frac{1}{n}\right)^n$, to kolejne wyrazy są jej przybliżeniami.
Ale do obliczenia kolejnego wyrazu nie korzystamy z poprzedniego ("tracimy obliczenia"), a co gorsza wyniki są wolno zbieżne.

Jeśli skorzystamy z szeregu: $e = \sum_{n=0}^{\infty} \frac{1}{n!}$, to chcąc zwiększyć precyzję wystarczy prosta operacja $S_n + \frac{1}{(n+1)!}$. Co więcej

$$0 \leqslant e - S_n \leqslant \frac{1}{n \cdot n!},$$

a więc algorytm jest szybko zbieżny (i - co ważne - mamy oszacowanie błędu **bez znajomości** dokładnej wartości liczby e).

W informatyce II.

Pamiętajmy, że każda liczba rzeczywista, to właściwie suma szeregu jej rozwinięć dziesiętnych (lub o innej podstawie) rzeczywistych - dla pewnych specjalnie istotnych liczb (np. π) podaje się oczywiście lepsze sposoby reprezentacji. Cały czas pracujemy z szeregami!

Zadanie: poszukać samodzielnie materiałów na ten temat...

Szeregi bardzo dobrze nadają się do przybliżania wartości: każde kolejne przybliżenie powstaje przez dodanie nowo obliczonego wyrazu do poprzedniego przybliżenia!

Zadanie.

Informatycy napotykają raczej *odwrotne zagadnienie*: mamy dany algorytm obliczający daną wartość. **Czy i dlaczego jest poprawny? Czy można go usprawnić?**
Poniżej procedura - ćwiczenie do tych pytań:

```
def approximate_pi():
    EPSILON = 1.0e-7
    term = 1
    n = 0
    sum_pi = 0
    while abs(term) > EPSILON:
        term = 4 * (((-1) ** (n)) / (2 * n + 1))
        sum_pi += term
        n += 1
    print(float(round(sum_pi,10)))
```

lub lepiej:

```
def approximate_pi():
    EPSILON = 1.0e-7
    n = 0
    sum_pi = 0
    sign = 1
    while True:
        term = 1 / (2 * n + 1)
        if term < EPSILON:
            break
        sum_pi += sign * term
        n += 1
        sign = -sign
    return float(round(4 * sum_pi, 10))

print(approximate_pi())
```

Szeregi w informatyce cd.

Pamiętajmy, że na pewno liczb niewymiernych nie da się reprezentować dokładnie na komputerze. Jedną z metod (całkiem niezłą) ich przybliżania będą szeregi. *Można skorzystać z gotowych bibliotek* (i co najmniej poznać ich algorytmy obliczeniowe) albo można tworzyć i badać własne algorytmy...

Czyli trzeba znać np. własności szeregów, aby wyeliminować ograniczenia arytmetyki komputerowej i uniknąć błędów...

Skrypt ilustracyjny obliczania sum szeregów w "Mathematica" - potrzebny darmowy *CDF Player* lub *Mathematica*

Zauważmy, że jeśli dany jest ciąg (s_n), to jest on ciągiem sum częściowych szeregu postaci

$$s_1 + (s_2 - s_1) + (s_3 - s_2) + \ldots = s_1 + \sum_{n=1}^{\infty}(s_{n+1} - s_n) \ .$$

Twierdzenie. *Szeregi* **zbieżne** *mają własność łączności dodawania wyrazów sąsiednich, tzn. jeżeli w szeregu zbieżnym $\sum_{n=1}^{\infty} a_n = s$ połączymy wyrazy sąsiednie w grupy, np.*

$$(a_1 + \ldots + a_{n_1}) + (a_{n_1+1} + \ldots + a_{n_2}) + (a_{n_2+1} + \ldots + a_{n_3}) + \ldots$$

gdzie $1 \leqslant n_1 < n_2 < \ldots$, to otrzymany szereg jest zbieżny do tej samej sumy s.

Te twierdzenie ułatwi obliczenia przybliżeń sum szeregów - **o ile tylko będziemy wiedzieli, że są zbieżne!**

Wyrażenia w nawiasach można obliczać w podprocedurach...

Obliczenia symboliczne.

Jeśli nie możemy sobie pozwolić na kumulowanie błędów na komputerze, musimy skorzystać ze specjalizowanych programów matematycznych np. *Mathematica* czy *Maple*. Tam są uwzględnione wszelkie reguły, o których mówimy (i dużo więcej!)... Np.

12.4.4 Szeregi w programie Mathematica

Program Mathematica dość dobrze radzi sobie z liczeniem sum szeregów, a w przypadku braku zbieżności sygnalizuje ten fakt. Do policzenia sumy szeregu $\sum_{i=p}^{\infty} a_n$ używamy instrukcji

```
Sum[a[n],{n,p,Infinity}]
```

Na przykład

```
Sum[1/n!, {n, 2, Infinity}]
```

zwraca $\sum_{n=2}^{\infty} \frac{1}{n!} = -2 + e$. Oznacza to, że Mathematica, w miarę swoich umiejętności, stara się zwrócić wynik dokładny, używając symbolicznych oznaczeń na stałe matematyczne.

Dopiero w końcowym oszacowaniu musimy obliczyć e...

W innych programach, takich jak Python możemy obliczać przybliżenia tej liczby, np.

```
def compute_e(n):
    e = 1
    fact = 1
    for i in range(1, n+1):
        fact *= i
        e += 1 / fact
    return e
```

Np.

```
n = 10
e = compute_e(n)
print("Liczba e:", e)
```

Czy potrafimy jednak sprawdzić precyzję dokonanych obliczeń? Zauważmy, że bez podnoszenia precyzji mamy taki efekt:

[1]:
```
def compute_e(n):
    e = 1
    fact = 1
    for i in range(1, n+1):
        fact *= i
        e += 1 / fact
    return e
```

[2]:
```
n = 10
e = compute_e(n)
print("Liczba e:", e)
```

```
Liczba e: 2.7182818011463845
```

[3]:
```
n = 100
e = compute_e(n)
print("Liczba e:", e)
```

```
Liczba e: 2.718281828459045
```

[4]:
```
n = 1000
e = compute_e(n)
print("Liczba e:", e)
```

```
Liczba e: 2.718281828459045
```

Własności szeregów zbieżnych.

Twierdzenie. Szeregi **zbieżne** mają własność łączności dodawania wyrazów sąsiednich, tzn. jeżeli w szeregu zbieżnym $\sum_{n=1}^{\infty} a_n = s$ połączymy wyrazy sąsiednie w grupy, np.

$$(a_1 + \ldots + a_{n_1}) + (a_{n_1+1} + \ldots + a_{n_2}) + (a_{n_2+1} + \ldots + a_{n_3}) + \ldots,$$

gdzie $1 \leqslant n_1 < n_2 < \ldots$, to otrzymany szereg jest zbieżny do tej samej sumy s.

Te twierdzenie ułatwi obliczenia przybliżeń sum szeregów - o ile tylko będziemy wiedzieli, że są zbieżne!

Własności szeregów zbieżnych.

Twierdzenie. *Załóżmy, że dane są dwa szeregi zbieżne* $\sum_{n=1}^{\infty} a_n = a$, $\sum_{n=1}^{\infty} b_n = b$ *oraz że* λ *jest dowolną liczbą.*

Wówczas szeregi $\sum_{n=1}^{\infty}(a_n + b_n)$, $\sum_{n=1}^{\infty}(a_n - b_n)$ *oraz* $\sum_{n=1}^{\infty} \lambda a_n$ *są zbieżne i zachodzą wzory*

(a) $\sum_{n=1}^{\infty}(a_n + b_n) = a + b$,

(b) $\sum_{n=1}^{\infty}(a_n - b_n) = a - b$,

(c) $\sum_{n=1}^{\infty} \lambda a_n = \lambda a$.

Takie działania - TYLKO na szeregach, o których **wiemy**, że są zbieżne.

Szeregi bezwzględnie i warunkowo zbieżne.

Szereg $\sum_{n=1}^{\infty} a_n$ nazywamy bezwzględnie zbieżnym, jeżeli szereg $\sum_{n\to\infty} \mid a_n \mid$ jest zbieżny.

Szereg $\sum_{n=1}^{\infty} a_n$ nazywamy warunkowo zbieżnym jeżeli jest on zbieżny, ale nie jest bezwzględnie zbieżny.

Uwaga: warunkowa zbieżność szeregu może prowadzić do problemów w obliczeniach na komputerze. Dlaczego? Na kolejnym slajdzie...

Twierdzenia mówiące o bezwzględnej i warunkowej zbieżności szeregów liczbowych.

Twierdzenie. *Każdy szereg bezwzględnie zbieżny jest zbieżny.*

Twierdzenie. *Szereg bezwględnie zbieżny pozostaje zbieżny i nie zmienia swojej sumy po dowolnej zmianie porządku wyrazów.*

Twierdzenie. (twierdzenie Riemanna) *W szeregu warunkowo zbieżnym można tak zmienić porządek wyrazów, aby nowy szereg był zbieżny do dowolnie obranej liczby, lub tak aby nowy szereg był rozbieżny .*

Przykładem szeregu warunkowo zbieżnego jest szereg anharmoniczny

$$\sum_{n=1}^{\infty}(-1)^{n+1}\frac{1}{n}.$$

Kluczowy problem w obliczeniach na komputerze: bez wiedzy o bezwzględnej zbieżności szeregu nie wolno zmieniać kolejności sumowania wyrazów - por. tw. Riemanna)

Zobaczmy jak wygląda problem:

a) jeżeli szereg byłby bezwględnie to można to sprawdzić stosując poznane wcześniej kryteria dla $\sum_{n=1}^{\infty} |a_n|$.

b) jeśli nie, lub nie potrafimy sprawdzić taką metodą zbieżności $\sum_{n=1}^{\infty} |a_n|$, to potrzebujemy nowych kryteriów.

Czy to jednak jest możliwe?

Przykład.

Rozważmy następujący ciąg:

$$1, -1, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{8}, \frac{1}{8}, \ldots$$

(pary $\frac{1}{2^n}, -\frac{1}{2^n}$ są przepisane 2^n razy).

Ciąg sum częściowych wygląda tak:

$$1, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, \frac{1}{4}, 0, \frac{1}{4}, 0, \frac{1}{4}, 0, \frac{1}{4}, 0, \frac{1}{8}, 0, \ldots$$

i oczywiście są zbieżne do zera. Zgodnie z definicją szereg utworzony przez nasz ciąg jest zbieżny.

A teraz poprzestawiamy wyrazy następująco:

$$1, -1, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \ldots$$

Teraz ciąg sum częściowych wygląda następująco:

$$1, 0, \frac{1}{2}, 1, \frac{1}{2}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{4}, \frac{1}{2}, \frac{1}{4}, 0, \ldots$$

i każdą z wartości 0 i 1 osiąga **nieskończenie wiele razy**.

Czyli nie jest zbieżny! Jak wiemy - nie jest też bezwzględnie zbieżny (tak naprawdę: dla szeregów liczbowych "bezwarunkowa zbieżność = bezwględna zbieżność").