Uniwersytet Wrocławski

Dokładność obliczeń numerycznych

Zbigniew Koza

Wydział Fizyki i Astronomii

Wrocław, 2016

MOTYWACJA

Komputer czasami produkuje nieoczekiwane wyniki

```
>> 10*(1-0.9)-1          # powinno być 0
ans =  -2.2204e-016

>> 1 + 1e-17 - 1         # powinno być 1e-17
ans =   0

>> n = int32(100000);
ans = 100000
>> n*n                   # powinno być 10000000000 lub 1e10
ans = 2147483647
```

There and back (?) again

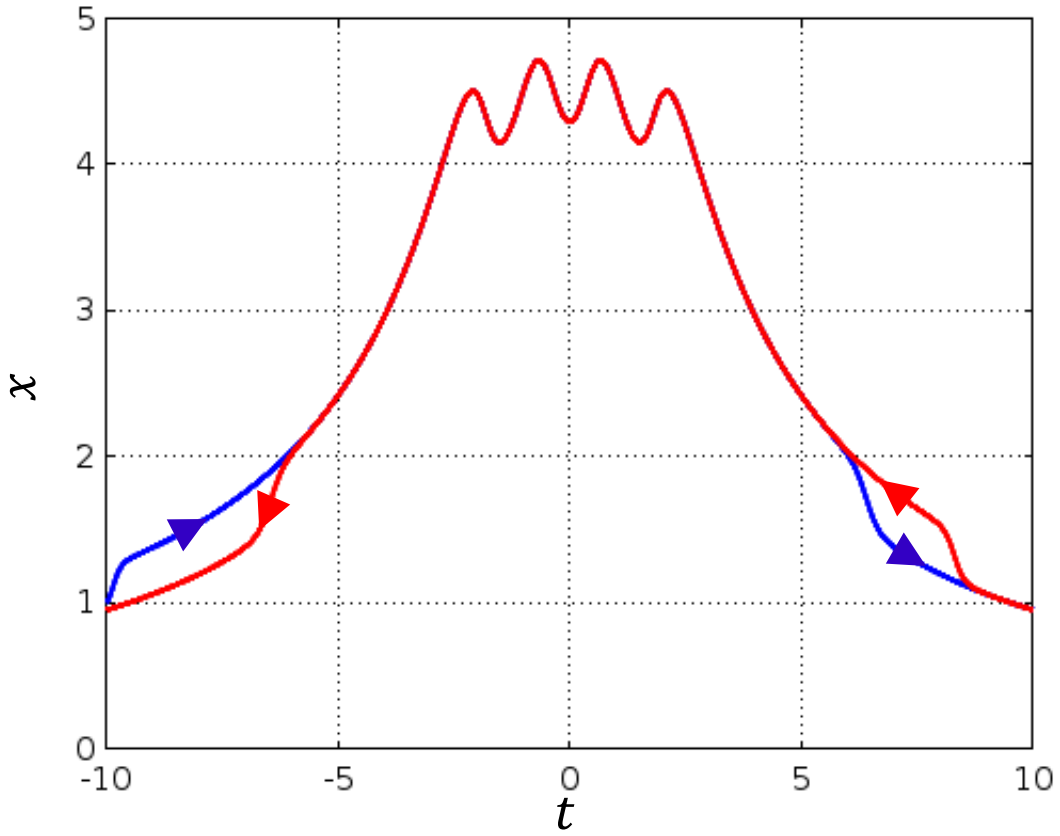
$$\frac{dx}{dt} = \sin(xt)$$
$$x_0 = 1$$
$$-a \leq t \leq a$$

```
f = @(x,t) sin(x*t);
>> N = 10000;
>> a = 10;
>> t = linspace(-a, a, N);
>> y=lsode(f, 1, t);
>> tt = -t  # odwracamy czas
>> yy = lsode(f, y(N), tt);
>> yy(N)
ans =  0.95222  # powinno być 1
```

There and back (?) again

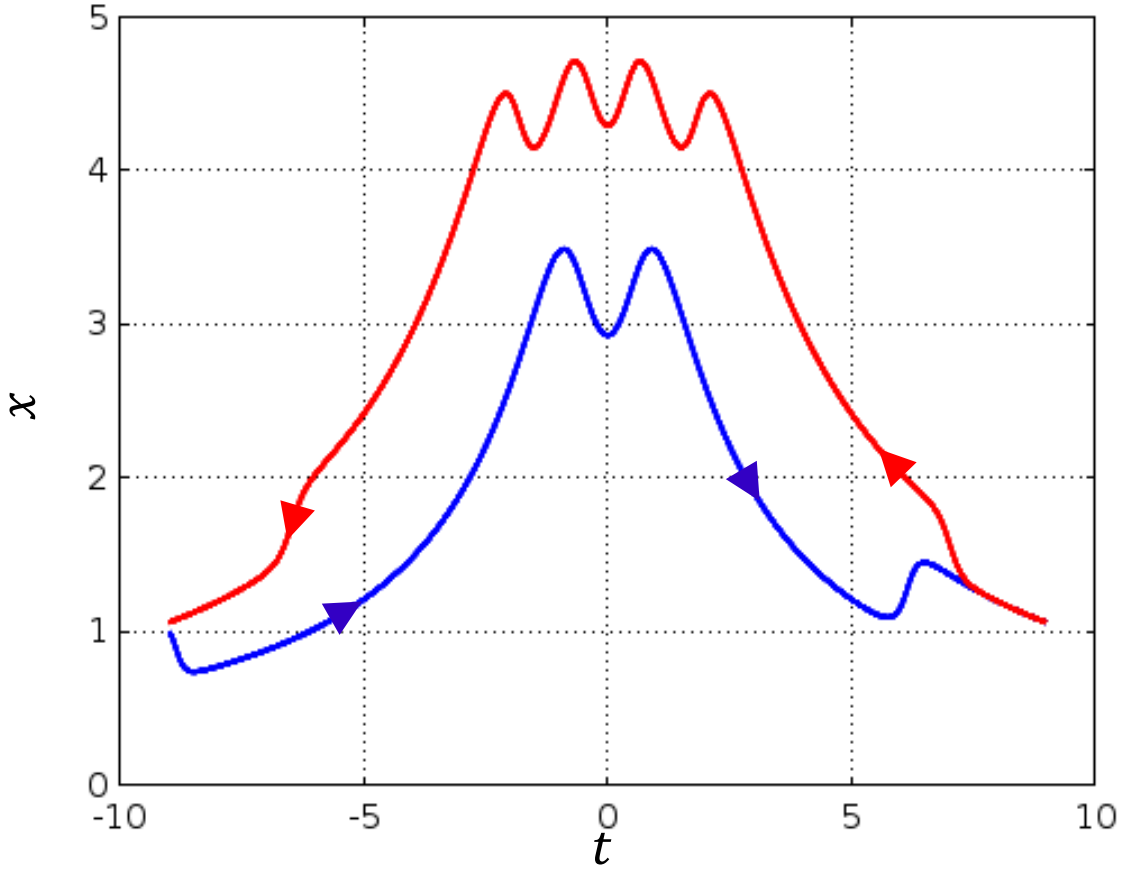
plot (t, y, tt, yy);

$$\frac{dx}{dt} = \sin(xt)$$
$$x_0 = 1$$
$$-a \leq t \leq a$$
$$\boldsymbol{a = 10}$$



There and back (???) again

plot (t, y, tt, yy);



$$\frac{dx}{dt} = \sin(xt)$$

$$x_0 = 1$$

$$-a \le t \le a$$

$$\boldsymbol{a = 9}$$

REPREZENTACJA MASZYNOWA LICZB – „LICZBY CAŁKOWITE”

Liczby naturalne

- Liczb naturalnych jest nieskończenie wiele, a pojemność pamięci komputera jest ograniczona
- Działania matematyczne na bardzo dużych liczbach są znacznie bardziej praco- i czasochłonne niż działania na liczbach niewielkich
- Większość energii, jaką do pracy potrzebuje procesor, związana jest z transferem danych
- Potrzebny KOMPROMIS

Kompromis

- Z jednej strony chcielibyśmy móc posługiwać się jak największym podzbiorem liczb naturalnych, oczywiście obejmującym liczby najczęściej używane, czyli stosunkowo niewielkie (tysiące, miliony, może miliardy)
- Z drugiej strony im bardziej ograniczymy ten podzbiór, tym obliczenia będą bardziej efektywne (skrócimy czas obliczeń, oszczędzimy na pamięci DRAM oraz rachunkach za prąd).

Kompromis – stan współczesny

- W ramach kompromisu definiuje się liczby naturalne, których ***reprezentacja dwójkowa*** ograniczona jest do N cyfr, gdzie $N \leq 64$
- Ze względów technicznych obecnie $N = 8 \cdot 2^k$

Nazwa (Octave)	C/C++ (64 bity, 2016)	N	Bajty (=N/8)	min	max (= $2^N - 1$)
uint8, char	unsigned char	8	1	0	255
uint16	unsigned short	16	2	0	65 535
uint32	unsigned int	32	4	0	4 294 967 295
uint64	unsigned long long	64	8	0	18 446 744 073 709 551 615

Procesor 64-bitowy

- 64-bitowe **rejestry** całkowitoliczbowe (do operowania na liczbach całkowitych)
- 64-bitowe **adresowanie pamięci** RAM (64-bitowe wskaźniki)
- 64-bitowe jednostki funkcjonalne (*datapath*) w procesorze (w tym **rejestry**, takie jak wskaźnik instrukcji, licznik programu, etc.)

16, 32 czy 64?

- Procesor 64-bitowy może w jednym cyklu zegara dodać dwie liczby 64-bitowe, o wartości rzędu $18 \cdot 10^{18}$; komputer 32-bitowy operuje na liczbach rzędu $4 \cdot 10^{9}$, a komputery PC z późnych lat 80. – ok. $6 \cdot 10^{5}$.
- Gdy pojawiły się komputery (PC) 32-bitowe, wszyscy natychmiast przerzucili się z liczb 16-bitowych na 32-bitowe; gdy wprowadzono komputery 64-bitowe, standardem pozostały liczby całkowite 32-bitowe. Dlaczego?

32 jest OK

- Na maszynach 16-bitowych ograniczenie liczb całkowitych do ok. 30 000 prowadziło do licznych kłopotów i błędów
- Ograniczenie rzędu 2 miliardy w komputerach 32-bitowych wystarcza w zdecydowanej większości „normalnych” programów
- Liczby 64-bitowe zajmują dwa razy więcej miejsca w pamięci niż liczby 32-bitowe

32 – dane, 64 – pamięć

- Wąskim gardłem współczesnych procesorów jest **prędkość transferu danych** z/do pamięci
- Jeśli danych nie ma w pamięci podręcznej (**cache**), to procesor może oczekiwać na ich pobranie setki (≈200-300) cykli zegara zamiast 4 (bufor L1), 10 (bufor L2), lub 40-80 (L3)
- Użycie liczb 64-bitowych zamiast 32-bitowych zmniejsza efektywną pojemność L1/L2/L3
- Jednak architektura 64-bitowa pozwala usunąć bariery w ilości adresowalnej pamięci (4-64 GB)

64 > 32

Komputery 64-bitowe:

- obsługują *znacznie więcej* pamięci operacyjnej
- mają więcej rejestrów
- znacznie szybciej przetwarzają liczby całkowite większe niż ok. 2 miliardy

Ale

- generują wyraźnie większy strumień danych: program (wskaźniki!) i dane programu

Dlatego normalnie **nie używa** się typu **uint64**

Reprezentacja binarna

Obecnie liczby naturalne powszechnie reprezentuje się w systemie dwójkowym

$$[b_{31}b_{30}b_{29}\dots b_1b_0] = \sum_{k=0}^{31} b_k \cdot 2^k$$

Reprezentacja binarna – przykład

- Dla `uint8` ($N = 8$ bitów)

$$\begin{matrix}[0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0] \\ [\mathbf{128} & \mathbf{64} & \mathbf{32} & \mathbf{16} & \mathbf{8} & \mathbf{4} & \mathbf{2} & \mathbf{1}]\end{matrix} = 2 + 4 + 16 = 22$$

Porównaj z systemem dziesiętnym:

$$\begin{matrix}[\mathbf{8} & \mathbf{2} & 0] \\ [\mathbf{100} & \mathbf{10} & \mathbf{1}]\end{matrix} = 8 \cdot 100 + 2 \cdot 10 = 820$$

Liczby całkowite

- Liczby całkowite mogą być ujemne…

Nazwa (Octave)	C/C++ (64 bity)	N	Bajty (=N/8)	min (= -2^{N-1})	max (= $2^{N-1}-1$)
int8	char	8	1	-128	127
int16	short	16	2	-32 768	32 767
int32	int	32	4	-2 147 483 648	2 147 483 647
int64	long long	64	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

Liczby całkowite

- Liczby całkowite mogą być ujemne…
- Powszechne rozwiązanie:
  waga najstarszego bitu jest *ujemna*

$$[b_{31}b_{30}b_{29}\ldots b_1b_0] = \mathbf{-b_{31}\cdot 2^{31}} + \sum_{k=0}^{\mathbf{30}} b_k \cdot 2^k$$

bit najstarszy

bit najmłodszy

minus

Przekroczenie zakresu

- Co się stanie, gdy wynik operacji arytmetycznej przekroczy dopuszczalny zakres danego typu liczb całkowitych?
- Niemal zawsze: BŁĄD
- Octave: monitoruje operacje i w razie przekroczenia zakresu w dół, przyjmuje że wynikiem jest wartość „minimum” dla danego typu
- Analogicznie dla przekroczenia zakresu w górę

Przekroczenie zakresu

- Przekroczenie zakresu nie powoduje żadnej reakcji alarmowej na procesorze
- Języki z rodziny C/C++ nie monitorują przekroczenia zakresu (to jest zbyt czasochłonne) – cała odpowiedzialność za poprawne działanie programu spada na programistę
- Inne języki: programy mogą zgłaszać wyjątek (np. Ada) lub promować zmienne do typu dowolnej precyzji (np. Python)

„Liczby całkowite”

- Ze względu na ograniczony zakres „liczb całkowitych” dostępnych w komputerach, nie są to dokładne odpowiedniki liczb naturalnych (całkowitych) używanych w matematyce

REPREZENTACJA MASZYNOWA LICZB – „LICZBY RZECZYWISTE”

Problem nieskończoności

- Z liczbami rzeczywistymi problem jest podobny jak z liczbami całkowitymi: jest ich nieskończona liczba
- Zapisanie dokładanej wartości tak użytecznych liczb rzeczywistych, jak π czy $\sqrt{2}$, wymagałoby użycia nieskończonej liczby cyfr (dziesiętnych, dwójkowych etc.)
- Rozpiętość liczb rzeczywistych używanych w inżynierii jest ogromna, co najmniej $10^{-30}…10^{30}$

Kompromis

Znowu potrzebny jest kompromis:

- Nie ma mowy o dokładnej reprezentacji liczb rzeczywistych – zamiast tego używa się ich ***przybliżeń*** (zaokrągleń)
- Chcielibyśmy, by te przybliżenia były ***jak najbardziej dokładne*** i obejmowały bardzo duży zakres liczb (np. $\pm[10^{-100}\ldots10^{100}]$)
- Z drugiej strony chcemy, by operacje na tych liczbach były wykonywane ***szybko***

Kompromis

- Kompromis musi więc obejmować:
  1. Dokładność (jak najmniejsze błędy zaokrągleń)
  2. Zakres (jak najszerszy zakres liczb)
  3. Możliwość zapewnienia efektywnej implementacji maszynowej (najlepiej: jedno dodawanie na cykl zegara)
  4. Jak najmniejsze obciążenie dla pamięci komputera
- Punkty 1-2 kłócą się z punktami 3-4

Liczby zmiennopozycyjne

- Kompromis polega na zastosowaniu **reprezentacji zmiennopozycyjnej**
- W tej reprezentacji do zapisu (przybliżenia) liczby rzeczywistej (x) używa się notacji inżynierskiej:

$$x = z \cdot u \cdot B^w$$

z – znak (-1 lub 1)

u – mantysa (*znormalizowana* ***część ułamkowa***)

B – podstawa (zwykle $B = 2$)

w – wykładnik

Mantysa

- Zakładamy, że część ułamkowa ma postać znormalizowaną

$$1 \leq u < B$$

Przykład:

$$h \approx \mathbf{6{,}626} \cdot \mathbf{10^{-34}} = \mathbf{+6.626E\text{-}34}$$

znak: **+1**
znormalizowana mantysa: **6,626**
wykładnik: **-34**
podstawa: **10**

Mantysa: układ dwójkowy

- $\sqrt{2} \approx 1.4142_{(10)} \approx 1.0110101000_{(2)}$
- $1.0110101_{(2)} = 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} = \frac{181}{128} \approx 1.4141$

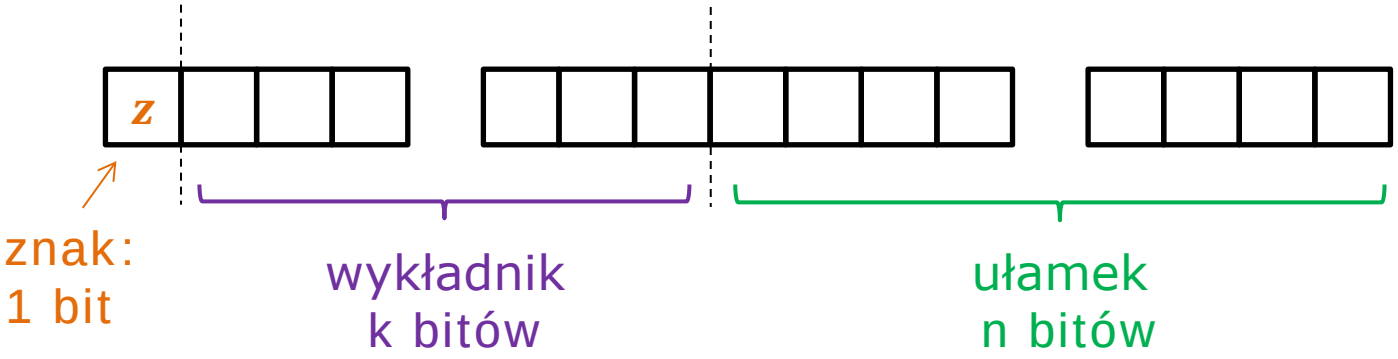
Ukryty bit

- Znormalizowana mantysa
$$1 \leq m < B$$
w układzie dwójkowym ($B = 2$)
zawsze zaczyna się cyfrą 1
– tej cyfry komputery *nie przechowują*
- Jest to tzw. cyfra ukryta/bit ukryty (*hidden bit*)

Wykładnik (cecha)

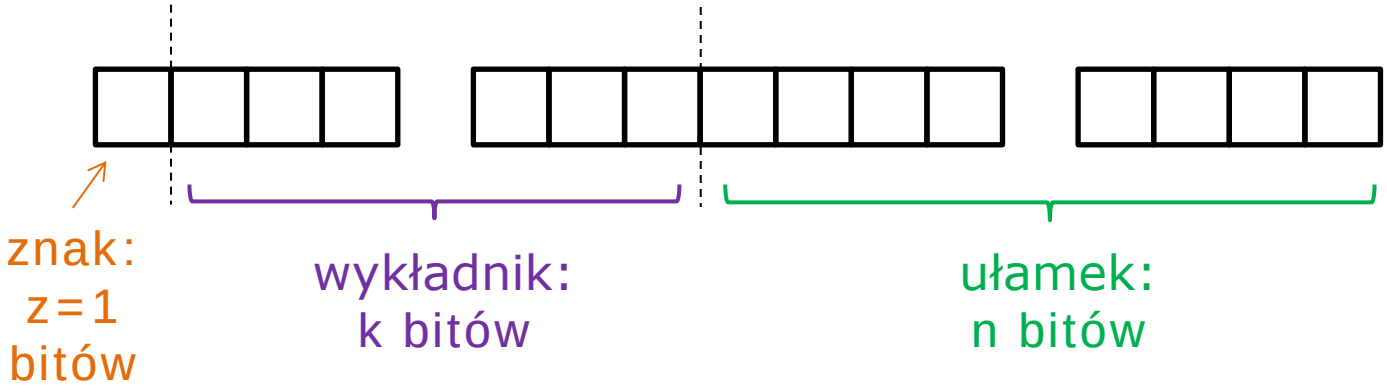
- Zakres wykładnika (w) powinien obejmować liczby ujemne i dodatnie; ujemne wykładniki odpowiadają liczbom < 1
- Z technicznego punktu widzenia korzystnie jest, by maszynowa reprezentacja wykładnika była nieujemna
- Dlatego wprowadza się tzw. *bias* (*b*): komputer przechowuje wartość $w + b$

IEEE Standard for Floating-Point Arithmetic (**IEEE 754**)



- 1 bit znaku
- k bitów wykładnika
- *bias* wykładnika $= 2^{k-1} + 1$
- n bitów części ułamkowej, bez ukrytego bitu
- podstawa arytmetyki $B = 2$ lub $B = 10$

IEEE Standard for Floating-Point Arithmetic (IEEE 754)



typ	bity	z	k	n	bias	cyfry znacz.	ϵ	Max_+	Min_+
single	32	1	8	23	127	≈ 7	≈1e-7	≈3e38	≈1e-38
double	64	1	11	52	1023	≈ 16	≈2e-16	≈2e308	≈2e-308
extended	80	1	15	64	16383	≈ 19	≈1e-19	≈1e4932	≈1e-4932

Octave: `format bit`

```
>> format bit;
>> int32(35)   # 35 = 32+2+1
ans = 00000000000000000000000001000011
>> 6/5
ans = 0011111111110011001100110011001100110011001100110011001100110011
```

znak

wykładnik

część ułamkowa

$$\frac{6}{5} = 1.2_{(10)} = 1.0011001100\ldots_{(2)}$$

hidden bit

wykładnik =
0 + bias =
0 + 1023 =
$01111111111_{(2)}$

Mnożenie

$$x_1 = z_1 \cdot u_1 \cdot B^{w_1}$$
$$x_2 = z_2 \cdot u_2 \cdot B^{w_2}$$

Jak obliczyć $x' = x_1 \cdot x_2$?

$$x' = z' \cdot u' \cdot B^{w'}$$

$x' = x_1 \cdot x_2$

$u' = u_1 \cdot u_2$ (+ zaokrąglenie + normalizacja)

$w' = w_1 + w_2$ (+ normalizacja)

- Skoro $1 \leq u' < 4$, to normalizacja jest prosta
- Dokładność wyniku jest na poziomie ϵ

Dzielenie

- Dzielenie działa podobnie do mnożenia, tzn. dokładność wyniku jest na poziomie epsilona maszynowego i wynika z zaokrąglenia dokładnego wyniku

Dodawanie i odejmowanie

$$x_1 = z_1 \cdot u_1 \cdot B^{w_1}$$
$$x_2 = z_2 \cdot u_2 \cdot B^{w_2}$$

Jak obliczyć $x' = x_1 \pm x_2$?

$$x' = z' \cdot u' \cdot B^{w'}$$

- Sprowadzamy problem do $x_1 \geq x_2 \geq 0$
- Wtedy $w_1 \geq w_2$, czyli $w_2 - w_1 \leq 0$ oraz

$$x_1 \pm x_2 = (u_1 \pm \underbrace{u_2 B^{w_2 - w_1}}_{\text{denormalizacja}}) \cdot B^{w_1}$$

Denormalizacja w dodawaniu

$$x_1 + x_2 = (u_1 + \underbrace{u_2 B^{w_2 - w_1}}_{\text{denormalizacja}}) \cdot B^{w_1}$$

- Przykład:
  $B = 10, k = 2, n = 2, x_1 = 10, x_2 = 0.33$
- x_1=1.0E01, x_2=3.3E-01
- $x_1 + x_2$=(1.0 + **3.3**$\cdot$ **10^{-2}**)E01
  =(1.0 + **0.033**)
  = 1.033E01= 1.0E01 = **10**

Epsilon maszynowy

- Jak widzimy, denormalizacja i zaokrąglanie prowadzi do sytuacji, gdy dla pewnych $x > 0$ zachodzi $1 + x = 1$
- Najmniejszą liczbę reprezentowaną w danym procesorze taką, że $1 + x > 1$ nazywamy **epsilonem maszynowym** i oznaczamy ϵ lub u.

Utrata dokładności w odejmowaniu

$$x_1 - x_2 = (u_1 - u_2 B^{w_2 - w_1}) \cdot B^{w_1}$$

- Przykład:
  $B = 10, k = 2, n = 2$
- x_1 = 1/3 = 3.3E-01, x_2 = 3/10 = 3.0E-01
- $x_1 - x_2$ = (3.3 – 3.0)E-01 = 0.3E-01 = 3.0E-2
- Wartość dokładna: $\frac{1}{3} - \frac{3}{10} = \frac{1}{30} = 0.033(3)$
- Błąd względny: $\frac{0.0333(3) - 0.03}{0.033(3)} = \frac{0.003(3)}{0.03(3)} = 10\%$
- Błąd jest znacznie większy od ϵ (1%)

Znoszenie się składników

- Odjęcie dwóch liczb zmiennopozycyjnych o zbliżonych wartościach prowadzi do utraty dokładności wyniku **znacznie przekraczającej epsilon maszynowy**
- Jest to „**znoszenie się składników**”
- Podobnie jest w przypadku **x+y**, gdy x i y są przeciwnych znaków
- Kilkukrotne zniesienie się składników zwykle czyni wynik bezwartościowym

Konsekwencje…

- W arytmetyce zmiennopozycyjnej:

$$(x \pm y) \pm z \neq x \pm (y \pm z)$$
$$(x * y) * z \neq x * (y * z)$$
$$x * (y + z) \neq x * y + x * z$$

  czyli **kolejność wykonywania operacji ma wpływ na wynik**

- Ten sam program w różnych komputerach (lub kompilowany z różnymi opcjami) może dać **inne wyniki**

Wartości specjalne

>> format bit

>> 0/0 **# NaN** (***N**ot **a** **N**umber*)

ans = **1111111111110000000000000000000000000000000000000000000000000000**

>> 1/0 **# Inf** (*Infinity*, czyli nieskończoność)

ans = **0111111111110000000000000000000000000000000000000000000000000000**

>> -1/0 **# -Inf**

ans = **1111111111110000000000000000000000000000000000000000000000000000**

>> 0 **# zero**

ans = **0000000000000000000000000000000000000000000000000000000000000000**

>> -0 **# minus zero**

ans = **1000000000000000000000000000000000000000000000000000000000000000**

Podsumowanie: źródła błędów (na poziomie maszynowym)

- Zależność wyniku od kolejności obliczeń
- Zaokrąglenia (ograniczona dokładność)
- Znoszenie się składników
- Wartości specjalne
- Przepełnienie i niedomiar
  (→ metody numeryczne)

ŹRÓDŁA BŁĘDÓW W INŻYNIERSKICH OBLICZENIACH KOMPUTEROWYCH

Błąd modelu ☹

Błędy danych wejściowych

- Występują praktycznie zawsze, gdy dane wejściowe pochodzą z pomiaru wielkości fizycznych
- Użyty algorytm może „wzmacniać” takie błędy – por. „efekt motyla”

Błędy reprezentacji

- Występują wtedy, gdy reprezentacja maszynowa nie jest w stanie odtworzyć z dostateczną dokładnością liczb znanych dokładnie
- Przykłady: liczby niewymierne, a także większość wymiernych, np. 1/3, 1/7 etc.

Błędy obcięcia

- Powstają wtedy, gdy „obcinamy” dokładne wyrażenie (zwykle nieskończoną sumę), np. zamiast

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

używamy przybliżenia

$$e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{3!}$$

Błędy obcięcia

Błąd obcięcia powstaje także w sytuacji, gdy

- Wartość całki przybliżamy skończoną sumą
- W szeregu Taylora zostawiamy tylko kilka pierwszych wyrazów

Błędy dyskretyzacji

- Wynikają z zastąpienia funkcji ciągłych ich wartościami w dyskretnym (=skończonym) zbiorze punktów
- Przykład:

$$f'(x) = \lim_{h \to \infty} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

- Oczywiście jest to rodzaj błędu obcięcia (szeregu Taylora)
  – tak częsty, że otrzymał specjalną nazwę

Niestabilność numeryczna

- „Efekt motyla”
- Czasami w samej istocie problemu matematycznego zawarta jest ogromna wrażliwość na jakikolwiek błąd w wartościach początkowych (lub pośrednich)

Błędy wynikające z reprezentacji maszynowej

- Zależność od kolejności obliczeń, zaokrąglenia, znoszenie się składników w arytmetyce zmiennopozycyjnej, etc.
- Przekroczenie zakresu w arytmetyce całkowitoliczbowej, etc.

Błędy wynikające z reprezentacji maszynowej

- Błędom tym można próbować przeciwdziałać poprzez zmianę algorytmu i/lub implementacji
- Trzeba jednak rozumieć, co się dzieje „pod maską” procesora – stąd tak długi wstęp do niniejszego wykładu

Pomyłki ☺

TESTY, TESTY, TESTY!!!

- Zwykle można przewidzieć wartości w pewnych przypadkach granicznych lub dostępne są wyniki referencyjne – używaj ich do kontroli poprawności swoich wyników

Przykłady

- **Katastrofa Ariane 5**: *integer overflow*
  http://www.around.com/ariane.html
  utrata rakiety i ładunku, 7 mld USD

- ***Failure at Dhahran:***
  loss of significance
  https://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran
  śmierć 28 żołnierzy *US Army*

Wnioski

- Po dzisiejszym wykładzie można by wnioskować, że to, iż komputerom w ogóle można ufać, zakrawa na cud
- W rzeczywistości nie jest tak źle
- Jednak ocena wiarygodności wyników symulacji należy do fundamentalnych problemów stojących przed każdym „symulantem”
- Potrzeba do tego lat doświadczenia