Доля П.Г.
Харківський національний університет імені В. Н. Каразіна
факультет математики і інформатики
кафедра теоретичної і прикладної інформатики
2018 г.

Вступ до чисельних методів з Octave

Глава 1. Особливості комп'ютерної математики

Зміст

1. Особливості комп'ютерної математики

При числовому розв'язанні задач однією з важливіших характеристик є похибка результату. Зазвичай вона складається з трьох частин:

- неусувної похибки розв'язку, яка обумовлена неточністю математичної моделі та вхідних даних;
- похибкою метода розв'язання задачі;
- обчислювальної похибки, яка є результатом округлювання під час підрахунків.

Перша похибка привноситься в задачу ззовні (від інженерів або фізиків) і математик її не контролює. Тому вона називається неусувною похибкою.

Друга частина виникає, коли нескінченна послідовність математичних операцій, що становлять метод, замінюється скінченною. Неістотні помилки кожного кроку, накопичуючись, можуть призводити до руйнівних наслідків. Алгоритми, цілком задовільні для малих завдань, можуть бути неефективними для великих задач того ж типу.

Обчислювальна похибка виникає тоді, коли величину неможливо зобразити обмеженою кількістю цифр в пам'яті компьютера, наприклад, число π. Навіть число 0.1 задається з похибкою, бо в двійковій системі, яка використовується комп'ютерами, воно представляється нескінченним рядом двійкових цифр `0.1=0.0001100110011001100110011...`, і яка б розрядність комп'ютера не була, ми все одно змушені цю послідовність обривати.

При розв'язанні задачі треба враховувати всі її особливості і використовувати метод, який привносить найменшу похибку в розв'язок. Отже

чисельні методи складаються не лише з розробки алгоритмів, але і з методів оцінювання їх похибок. Поточна глава присвячена дослідженню і можливому усуненню похибок, яки виникають в простих задачах.

1.1. Машинна арифметика.

Маючи справу з арифметикою, ми звикли користуватися позиційною десятковою системою числення. Назва «десяткова» пояснюється тим, що в основі лежить десять цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Термін «позиційна» вказує на те, що значення цифри в запису числа залежить від її положення (позиції). Наприклад, в десятковому числі 2749 цифра 9 представляє кількість одиниць, цифра 4 – кількість десятків, цифра 7 – кількість сотень, і цифра 2 – кількість тисяч. Це стає зрозумілим, якщо число записати у вигляді суми $2\cdot10^3+7\cdot10^2+4\cdot10^1+9\cdot10^0$, в якій 10 є основою системи числення. Якщо десяткове число дробове, то воно теж записується у вигляді суми, в якій ступінь 10 для кожної цифри після десяткової крапки від'ємна (в англомовних математичних джерелах ціла частина відокремлюється від дробової крапкою і ми будемо дотримуватися цього правила). Наприклад, десяткове число 384.9506 виразиться сумою $3\cdot10^2 + 8\cdot10^1 + 4\cdot10^0 + 9\cdot10^{-1} +5\cdot10^{-2} + 0\cdot10^{-3} + 6\cdot10^{-4}$. Ступінь десятки, перед якою стоїть цифра, називається вагою цифри. Ліворуч від десяткової крапки вона приймає послідовно додатні значення 0, 1, 2, 3, ..., а праворуч – від'ємні -1, -2, -3, ...

В комп'ютерах внутрішнє представлення чисел двійкове, тобто використовується лише дві цифри (нуль та одиниця). Як і десяткове число, будь яке двійкове число можна записати у вигляді суми, яка явно відбиває відмінність ваг цифр, що входять в двійкове число. Наприклад, двійкове число 110101.101 в формі суми має вигляд

$$1\cdot2^5 + 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-1} + 0\cdot2^{-2} + 1\cdot2^{-3}.$$

Ця сума записується за тими ж правилами, що і сума для десяткового числа. Для того, щоб відрізнити двійкове число від десяткового, в запису числа зручно додавати ознаку системи числення, наприклад, 110101.111_2. Виконуючи в останній сумі арифметичні операції за правилами десяткової системи, отримаємо число 53.625. Тобто $110101.101_2=53.625_{10}$.

Кожен розряд (цифру) двійкового числа називають бітом. У комп'ютерах кількість двійкових розрядів (бітів) для запису чисел фіксована. Вона називається розрядністю або довжиною розрядної сітки комп'ютера і зазвичай дорівнює 32 або 64. Фіксована довжина розрядної сітки призводить до появи похибок представлення чисел. Наприклад, десяткове число 0.1 не можна точно зберегти в пам'яті комп'ютера, бо

$$\frac{1}{10}=\frac{0}{2^1}+\frac{0}{2^2}+\frac{0}{2^3}+\frac{1}{2^4}+\frac{1}{2^5}+\frac{0}{2^6}+\frac{0}{2^7}+\frac{1}{2^8}+\frac{1}{2^9}+\frac{0}{2^{10}}+\frac{0}{2^{11}}+\frac{1}{2^{12}}+\frac{1}{2^{13}}+\frac{0}{2^{14}}+\frac{0}{2^{15}}+\ldots$$

Таким чином, $0.1_{10}=0.0001100110\ 01100\ldots_2$, і при будь якій розрядності комп'ютера ми змушені цю послідовність обривати і отримувати похибку. Коли

підсумовується десять таких дробів, результат не є точною одиницею, тобто $\underbrace{0.1+0.1+...+0.1}_{10\ разів} \neq 1$.

Таким же чином число 0.2_{10} буде зображено, як 0.200000003_{10} в одинарній точності. Тому `0,2 + 0,2 ≈ 0,4 ≠ 0.4`.

Коли в мові програмування описується тип змінної, наприклад,

`float x;`

то цим визначається кількість розрядів і спосіб збереження двійкових цифр числа `x`. Так в стандарті IEEE 754 для представлення дійсних чисел одинарної точності використовується 32 біти і застосовується формула

$$(-1)^S \times 1.\mathrm{M} \times 2^P = (-1)^S \times 1.\mathrm{M} \times 2^{X-127},$$

де S – біт знака числа (S=0 – число додатне, S=1 – від'ємне), P – порядок числа (степінь двійки старшого розряду), $X = P + 127_{10}$ – характеристика, 1.M – нормалізована мантиса.

Розглянемо докладніше формат дійсного числа одинарної точності. Воно займає 4 байти (32 біти). Знак кодується в сьомому розряді старшого байту. Коди рівних по абсолютній величині додатних і від'ємних чисел відрізняються лише цим розрядом. Характеристика X числа займає 8 розрядів (молодший розряд зберігається в сьомому біти третього байта) і зміщена відносно справжнього порядку P на $7F_{16}$ (127_{10}), тобто $X = P + 7F_{16}$ або $P = X - 7F_{16}$. Порядок P може змінюватися в межах від $-7E_{16}$ до $7F_{16}$ (від -126_{10} до 127_{10}). Мантиса завжди нормалізована і має 23 явних розряди і один неявний. Нормалізація виконується так, що старша значуща цифра завжди розташовується перед двійковою крапкою (тобто має нормалізовану вагу 2^0). Цей розряд в мантисі не вказується, але завжди мається на увазі. Тобто нормалізована мантиса має вигляд $1.\mathrm{M} = 1.\alpha_1\alpha_2\alpha_3...\alpha_{23}$, де α_i - двійкова цифра в i-му розряді мантиси (з нормалізованою вагою 2^{-i}), M – числове (двійкове) значення дробової частини нормалізованої мантиси.

Приклад 1. Дешифрувати послідовність байт $00_{16}\ 00_{16}\ 20_{16}\ 3E_{16}$ в форматі короткого дійсного числа.

Записуємо вміст байтів. Маємо

Оскільки `S=0`, то число додатне

Порядок обчислимо в Octave (використовуючи функцію перетворення `bin2dec('рядок нулів та одиниць')`.

```
>> bin2dec('1111100')-127
ans = -3
```

Тобто порядок $P=001111100_2 - 127_{10} = -3$.

Нормалізована мантиса дорівнює $1.M=1.01_2$ (з урахуванням неявної першої одиниці). Тоді число F дорівнює

$$F = 1.01 \cdot 2^{-3} = 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0.125_{10} + 0.03125_{10} = 0.15625_{10}.$$

Приклад 2. Дешифрувати послідовність байт $00_{16}\ 00_{16}\ 5A_{16}\ C1_{16}$ в форматі короткого дійсного числа.

Послідовність біт має вигляд

1100 0001 0101 1010 0000 0000 0000 0000

$S=1$, отже число від'ємне.

Характеристика $X=10000010_2$

```
>> bin2dec('10000010')
ans = 130
```

Порядок $P=130_{10}-127_{10}=3$.

Тоді число $F = (-1)^1 \cdot 1.101101 \cdot 2^3 = -1101.101_2 = -13.625_{10}$

■

У використовуваному способі є недолік – в ньому немає можливості зобразити нуль, бо в нормалізованій мантисі завжди присутня неявна одиниця. В цьому форматі діє домовленість, що дійсний нуль має нульову мантису і порядок $P=-127_{10}$. При чому існують два нулі: +0 і −0. Хоча апаратно реалізацію операцій з нулем потрібно реалізовувати окремо, зазвичай вони виконуються швидше, ніж зі звичайними числами.

Також в IEEE 754 передбачено представлення спеціальних чисел: нескінченність та невизначеність (NaN – not a number)

Формат довгого дійсного числа побудовано аналогічно, але збільшена розрядність (характеристика 11 біт, зміщення 1023_{10}, мантиса 52 біт) забезпечує більш високу точність і більш широкий діапазон представлення чисел.

Десяткове число з рухомою крапкою має вигляд $M \times 10^P$, де M – мантиса, P – показник степеня (ціле число). При чому число M зліва від десяткової крапки містить лише одну цифру. У обох чисел M і P кількість розрядів скінченна, тому існує лише скінченна множина чисел з рухомою крапкою і, зокрема, в цій множині існують найменше і найбільше число.

Найбільше дійсне число одинарної точності з рухомою крапкою дорівнює

$$1.11111111_2 \times 2^{127} = 11111111_2 \times 2^{119} \approx 1.6948 \cdot 10^{38}.$$

Дійсно

```
>> bin2dec('11111111')*2^119
ans = 1.6948e+038
```

Якщо результат операції перевищує це число, то відбувається переповнення.

Найменше число ε

$$\varepsilon = 1.00000001_2 \times 2^{-126} \approx 2^{-126} \approx 1.1755 \cdot 10^{-38}.$$

Між нулем і найменшим додатним числом ε існує розрив. Якщо результатом операції є надто близьке до нуля число (менше за ε) то кажуть, що виникає машинний нуль.

Мантиса з 24 біт відповідає приблизно 7 десятковим розрядам.

Отже результат операції над числами з рухомою крапкою зазвичай не буває точним. Різниця між точною відповіддю і відповіддю, отриманою в машинній арифметиці, називається помилкою округлювання.

Найменше число с рухомою крапкою, яке при додаванні до 1.0 повертає результат більший за 1.0, називається машинним епсілоном і позначається $\varepsilon_{маш}$. Конкретне значення $\varepsilon_{маш}$ залежить від того, яка арифметика використовується – з округленням чи з усіканням. Машинний епсілон визначає відносну похибку арифметики комп'ютера. Якщо *x* і *y* два додатних числа з рухомою крапкою і $x > y$, то їх суму можна записати у вигляді $x + y = x \cdot \left(1 + \frac{y}{x}\right)$. Якщо $\frac{y}{x} < \varepsilon_{маш}$, то сума співпаде з *x*. Для 32-бітової арифметики з рухомою крапкою, що задовольняє стандарту IEEE, $\varepsilon_{маш} = 2^{-23} \approx 1.19 \cdot 10^{-7}$. Тому немає сенсу розраховувати більш ніж на 7 правильних десяткових знаків або на те, що можна розрізнити відносні відмінності менші за цей рівень.

В стандарті IEEE є ще одна особливість. Коли порядок `P=-127` (для чисел типу `float`) і мантиса нульова, то число вважається рівним нулю. Якщо мантиса ненульова, то число вважається ненульовим, його порядок дорівнює `-127`, а неявний старший біт мантиси вважається рівним нулю. Такі числа називаються денормалізованими.

Підведемо короткий підсумок найважливіших властивостей обчислювань з рухомою крапкою для чисел одинарної точності ([10]).

1. Множина чисел з рухомою крапкою скінченна.
2. Існує найбільше число OFL (overflow level)$\approx 10^{38}$.
3. Існує найменше число UFL (underflow level)$\approx 10^{-38}$.
4. Числа з рухомою крапкою між 0 і OFL розподілені нерівномірно. Між кожними сусідніми степенями двійки розташовано 2^{23} чисел з рухомою крапкою. Наприклад між 2^{-126} і 2^{-125} розташовано стільки ж чисел (тобто 2^{23}), скільки і між 2^{126} і 2^{127}. Таким чином, числа з рухомою крапкою розташовані щільніше біля нуля.
5. Арифметичні операції з числами з рухомою крапкою не завжди призводять до точно зображувальних результатів. Тому результати доводиться усікати або округляти до найближчого числа з рухомою крапкою.
6. Константа $\varepsilon_{маш}$ є найменшим числом з рухомою крапкою, для якого $1.0 + \varepsilon_{маш} > 1.0$ в арифметиці з рухомою крапкою. Для 32-бітового комп'ютера $\varepsilon_{маш} \approx 10^{-7}$. Це число характеризує відносну похибку машинної арифметики.
7. Константи OFL і UFL визначаються головним чином кількістю бітів показника, тоді як $\varepsilon_{маш}$ - кількістю бітів мантиси. Мають місце наступні нерівності

$$0 < UFL < \varepsilon_{маш} < OFL.$$

1.2. Приклади помилок обчислень.

В цьому параграфі ми розглянемо приклади, які наочно пояснюють необхідність оцінювання похибки і вибору «оптимального» способу обчислювання. Але спершу нагадаємо деякі визначення.

Різницю між точним x і наближеним значенням $\tilde{x}$ називають абсолютною похибкою $\Delta x = x - \tilde{x}$.

Відносною похибкою називають відношення абсолютної похибки до абсолютного значення наближеної величини $\delta x = |\Delta x| / |\tilde{x}|$. На відміну від абсолютної похибки, яка зазвичай буває розмірною величиною, відносна похибка – безрозмірне число, і тому краще характеризує справжню точність.

Всі цифри десяткового запису числа, починаючи з першої ненульової зліва, звуться значущими цифрами цього числа. Нулі в кінці числа вважають значущими, інакше їх не пишуть.

Машинна арифметика є наближенням до того, що ми зазвичай розуміємо під арифметикою. Відмінність між ними є причиною дивних ефектів, які ми розглянемо в наступних прикладах. Щоб наочніше їх проілюструвати ми будемо використовувати невелику кількість значущих цифр.

Такі математичні пакети як MatLab і Octave в арифметичних обчислюваннях використовують процесорну точність. Але символьні обчислення в них можна виконувати з будь якою кількістю значущих цифр. Тому для ілюстрації деяких ефектів ми будемо вживати символьні константи, і використовувати потрібну кількість значущих цифр.

Щоб загрузити пакет символьної математики в Octave потрібно виконати наступну команду

```
>> pkg load symbolic
```

Для задання кількості значущих цифр, яка буде використовуватися в арифметичних операціях з символьними константами, призначена функція `digits`. Наприклад, команда

```
>> digits 12
```

встановлює 12 значущих цифр для символьних констант. Для створення символьної константи з числа типа `double` або рядка, що представляє число, призначена функція `vpa(...)`. Її можна використовувати двома способами: `vpa(число)` та `vpa(рядок)`. Наприклад,

```
>> a=vpa(123.456)
a = (sym)123.456000000
```

Тут створена символьна константа `a`, яка містить 12 значущих цифр.

Функція `vpa` може мати другий аргумент, який задає кількість значущих цифр для символьного числа, відмінну від встановленого функцією `digits`. Але будьте уважні, коли перетворюєте `double` число, до символьної константи при великій кількості значущих цифр.

```
b=vpa(0.0123456789, 20)
b = (sym)0.012345678900000000427
```

Але

```
b=vpa('0.0123456789', 20)
```

```
b = (sym) 0.012345678900000000000
```
Функція digits тільки встановлює ознаку відображати результат з заданою кількістю цифр, а в пам'яті після виконання арифметичних обчислень результат зберігає більшу кількість цифр.
```
digits 4
x1=vpa(1364.0)
x1 = (sym) 1364.
x2=vpa(26.46)
x2 = (sym) 26.46
x3=vpa(1.475)
x3 = (sym) 1.475
x12=x1+x2
x12 = (sym) 1390.
x12+x3
ans = (sym) 1392.
```
Останнє додавання показує, що в робочому просторі системи x12 зберігається у вигляді 1390.46 і додаваня числа 1.475 після округлювання дає результат 1392.

Щоб наші обчислювання моделювали скінченнозначну арифметику і зберігали потрібну кількість значущих цифр, результат треба перевести до рядкового вигляду, а потім знову до символьного. Наприклад, якщо суму x1 + x2 спочатку перетворити до рядкового типу, а потім знову до символьного, то результат додавання буде дійсно зберігати лише потрібну кількість значущих цифр.
```
x12=vpa(char(x1+x2))
x12 = (sym) 1390.
```
Тоді
```
x12+x3
ans = (sym) 1391.
```
Нам буде зручно написати відповідну функцію перетворення
```
function y=vpachar(x)
  y=vpa(char(x));
```
яка буде залишати в відповіді арифметичної операції тільки потрібну кількість значущих цифр. Наприклад
```
x12=vpachar(x1+x2)
x12 = (sym) 1390.
x12+x3
ans = (sym) 1391.
```
Отже щоб в Octave арифметична операція залишала в відповіді таку ж кількість значущих цифр, що встановлено функцією digits, результат треба піддавати перетворенню функцією vpachar.

Зауваження. В системі символьних обчислювань Maple схожа інструкція Digits:=n дійсно обмежує кількість значущих цифр в арифметичних операціях і ніяких перетворень результатів робити не потрібно.

Для демонстрації прикладів з обмеженою кількістю значущих цифр в арифметичних операціях інколи краще не вживати символьні константи, а після кожної арифметичної операції використовувати функцію `fround(X,n)`, яка буде округляти число `X` до `n` значущих цифр. Для неї можна запропонувати наступний код.

```
function r=fround(x,n)
  sgn=sign(x);
  ax=abs(x);
  c=ceil(log10(ax));
  r=sgn*round(ax*10^(n-c))*10^(c-n);
```

Нагадаємо, що функція `ceil(z)` повертає найближче ціле, більше за `z`, а `round(...)` – найближче ціле. Вираз `ceil(log10(abs(ax)))` повертає кількість цифр в цілій частині модуля числа. З використанням створеної функції `fround(x,n)` обчислення з заданою кількістю значущих цифр можна виконувати, наприклад, наступним чином.

```
x1=1364
x1 =  1364
x2=26.46
x2 =  26.460
x3=1.475
x3 =  1.4750
x12=fround(x1+x2,4)
x12 =  1390.0
fround(x12+x3,4)
ans =  1391
```

Незважаючи на штучний вигляд створених функцій, вони в наших прикладах будуть імітувати арифметичні обчислення з заданою кількістю цифр. Використання функції `vpachar` буде виконувати обчислення з відкиданням незначущих цифр, а функції `fround` буде округляти останню значущу цифру числа. Різні процесори можуть виконувати арифметичні обчислення як з відкиданням незначущих цифр, так і з округленням останньої.

Приклад 1. Нехай обчислення виконуються з точністю до 4 – х значущих цифр. Виконаємо віднімання двох чисел `123.4-0.0345`, використовуючи наступний код Octave.

```
digits 4
x1=vpa(123.4)
x1 =(sym)123.4
x2=vpa(0.0345)
x2 =(sym)0.03450
x1-x2
ans = (sym)123.4
```

Хоча кожний доданок використовує 4 значущих цифри, в результаті зменшуване число не змінилося. Аналогічно

```
>> x1+x2
ans =(sym)123.4
```

Не треба думати, що наведений приклад штучний. Аналогічні ефекти ми спостерігаємо, коли додаємо числа з плаваючою крапкою.

```
format long
x1=123456.7890
x1 = 123456.789000000
x2=3.5678e-10
x2 = 3.56780000000000e-010
x1+x2
ans = 123456.789000000
```

Додавання числа `x2` до `x1` не змінило `x1`!

Спосіб запису числа `x2` при введенні не має значення, бо комп'ютер зберігає дійсні числа в формі з рухомою крапкою.

```
x2=0.00000000035678
x2 = 3.56800000000000e-010
x1+x2
ans = 123456.789000000
```

Висновок, який можна зробити з цього прикладу, полягає в тому, що при додаванні або відніманні наближених чисел бажано, щоб ці числа мали однакові абсолютні похибки, тобто мали однакову кількість розрядів після десяткової точки (а не однакову кількість значущих цифр).

Приклад 2. Використовуючи 4-х розрядну арифметику (4 значущі цифри) виконаємо додавання п'яти чотирирозрядних чисел в прямому і зворотному порядку.

```
S = 1364 + 26.46 + 1.475 + 0.3944+ 0.2764
```

Ось відповідний сценарій Octave.

```
digits(4)
s1=vpa(0.0);
X=vpa([1364 26.46 1.475 0.3944 .2764]);
for i = 1:5 # додавання по черзі
  s1=vpachar(s1+X(i));
endfor
s1
s2=vpa(0.0);
for i = 5:-1:1  # додавання в зворотньому порядку
  s2=vpachar(s2+X(i));
endfor
s2
```

Виконаємо цей сценарій.

```
>> ex006
s1 = (sym) 1391.
s2 = (sym) 1393.
```

Отже додавання чисел по черзі дає `1391`, а в зворотному порядку – `1393`. Це пояснюється тим, що округлення результатів при обчислюванні на комп'ютері

відбувається після кожного додавання. Дійсно, складемо числа послідовно від першого до останнього

```
x12=vpachar(X(1)+X(2))
x13=vpachar(x12+X(3))
x14=vpachar(x13+X(4))
x15=vpachar(x14+X(5))
x12 = (sym) 1390.
x13 = (sym) 1391.
x14 = (sym) 1391.
x15 = (sym) 1391.
```

Аналіз процесу обчислень показує, що втрата точності тут відбувається через те, що додавання малих чисел до великого не відбувається, тобто $a+b=a$, якщо $a>>b$. Цих малих чисел може бути дуже багато, але на результат вони не вплинуть, оскільки додаються по одному.

Бажано дотримуватися правила, відповідно до якого додавання чисел потрібно проводити в міру їх зростання. Для машинної арифметики через похибки округлення важливо дотримуватися «оптимального» порядку виконання операцій. Відомі з алгебри закони асоціативності і дистрибутивності тут не виконуються.

Отже в нашому прикладі правильно буде додавати числа по порядку, починаючи з кінця.

Приклад 3. Використовуючи чотирьохзначну арифметику, обчислимо суму

$$S=\sum_{i=1}^{10000}\frac{1}{i^2}.$$

Підсумувати цей ряд можна від першого елемента до останнього або в зворотному порядку. Виявляється, що в другому випадку сумарна обчислювальна похибка буде значно менше.

```
s1=0.0;
N=10000;
M=4;          # кількість значущих цифр
for i = 1:N
   s1=fround(s1+fround(1/i^2,M),M);  # s1=s1+1/i^2;
endfor
s1
s2=0.0;
for i = N:-1:1
  s2=fround(s2+fround(1/i^2,M),M);   # s2=s2+1/i^2;
 endfor
 s2
>> ex009
s1 = 1.62500000000000
s2 = 1.64500000000000
```

Якщо виконати обчислення з процесорною точністю, яку використовує Octave, то результатом буде число `1.64483407184806`. Як бачимо, сума `s2` дає значно краще наближення, ніж `s1`.

Підсумовування ряду від меншого числа до більшого дає кращу точність!

Приклад 4. Обчисліть суму $\sum_{k=1}^{10\,000\,000} 0.1$ Мається на увазі додавання числа $\frac{1}{10}$ 10 000 000 разів. Відповідь 1 000 000.

```
s=0.0;
N=10000000;
for i = 1:N
  s=s+0.1;
endfor
s
>> ex010
s = 999999.999838975
```

Отже

$$\sum_{k=1}^{10\,000\,000} 0.1 = 999999.999838975\ .$$

Помилка виникає через неточності представлення числа `0.1` в пам'яті комп'ютера.

Ось той же приклад, але обчислення виконуються для дійсних чисел одинарної точності.

```
format long
s=single(0.0);
N=10000;
x=single(0.1);
for i = 1:N
  s=s+x;
endfor
s
s =  999.902893066406
```

Для `N=100000` ми отримуємо `s = 9998.55664062500`

Дійсні числа одинарної точності зберігають приблизно 7 десяткових значущих цифр, але, як бачите, похибка з'являється в 4-й значущій цифрі.

Приклад 5. Обчислимо значення полінома по формулі $P(x) = x^3 - 3x^2 + 3x - 1$ та по формулі $Q(x) = ((x-3)x+3)x-1$ в точці $x_0 = 2.19$, використовуючи трьохрозрядну арифметику.

Спершу перевіримо, що ми маємо справу з одним і тим же поліномом.

```
>> syms x
>> Q=((x-3)*x+3)*x-1
Q =(sym) x*(x*(x-3)+3)-1
>> expand(Q)
ans = (sym)
   3      2
  x  - 3*x  + 3*x - 1
```

Тепер обчислимо значення цих поліномів, використовуючи трирозрядну арифметику.

```
digits(3)
x=vpa(2.19,3);
P=vpachar(vpachar(x^3)-3*vpachar(x^2)+3*vpachar(x)-1)
Q=vpachar((vpachar(vpachar(x-3)*x)+3)*x)-1
P = (sym) 1.67
Q = (sym) 1.69
```

Зауваження. При обчислюванні величини `P` потрібно було б застосовувати функцію `vpachar` після кожної арифметичної операції.

Тепер знайдемо справжнє значення полінома та помилки трьохрозрядних обчислювань.

```
digits(10)
x=vpa('2.19',10);
P0=x^3-3*x^2+3*x-1;
Q0=((x-3)*x+3)*x-1;
P2=subs(P0,x,vpa('2.19',10))
Q2=subs(Q0,x,vpa('2.19',10))
dP=P2-P
dQ=Q2-Q
ans = (sym) 1.685159000
ans = (sym) 1.685159000
dP = (sym) 0.01523712493
dQ = (sym) -0.004782406235
```

Помилки дорівнюють `0.01523712493` та `-0.004782406235` відповідно. Отже наближення $Q(2.19)\approx 1.69$ має меншу помилку.

Приклад 6. Нехай дано вираз $\left(3-2\sqrt{2}\right)^3$. Оскільки

$$\left(3-2\sqrt{2}\right)^3 = 3^3 - 3\cdot 3^2\cdot 2\sqrt{2} + 3\cdot 3\cdot\left(2\sqrt{2}\right)^2 - \left(2\sqrt{2}\right)^3 = 99-70\sqrt{2},$$

то обчислимо його десяткове значення по лівій і правій формулах при наближеннях кореня зверху та знизу : `1.4,1.41,1.42,1.414,1.415, 1.4142,1.4143`. Наприклад, для наближення $\sqrt{2}\approx 1.4$ виконайте послідовність команд Octave

```
>> format long
>> sq2=1.4; disp((3-2*sq2)^3);disp(99-70*sq2)
0.00800000000000002
1
```

Отриманими значеннями заповніть перший рядок наступної таблиці. Виконавши аналогічні інструкції, заповніть інші рядки таблиці (або напишіть сценарій).

$\sqrt{2}$	$\left(3-2\sqrt{2}\right)^3$	$99-70\sqrt{2}$
1.4	0.00800000000000002	1
1.41	0.00583200000000002	0.300000000000011

1.42	0.00409600000000001	-0.399999999999991
1.414	0.00508844800000001	0.0200000000000102
1.415	0.00491299999999999	-0.0499999999999972
1.4142	0.00505302969600002	0.00600000000000023
1.4143	0.00503538234400002	-9.99999999990564e-004

Ми отримали дуже відмінні результати, і зразу не зрозуміло, який з них ближче до правильної відповіді. Навіть знак деяких значень невірний, бо $3>2\sqrt{2}$ (порівняйте квадрати цих чисел) і зрозуміло, що шукане число повинно бути додатним. В той же час бачимо, що значення в стовпці для виразу $\left(3-2\sqrt{2}\right)^3$ відрізняються одне від одного значно менше.

■

Розглянемо два досить близьких числа `p=3.1415957341` і `p=3.1415926536`, які мають однакову кількість (11) значущих цифр. Обчислимо їх різницю `p-q=0.0000030805`. Вона містить тільки 5 значущих цифр. Це явище називається втратою значущих цифр або втратами із-за віднімання. Воно може зустрітися тоді, коли ви про нього і гадки не маєте.

Приклад 7. Розв'яжемо квадратне рівняння $x^2-140x+1=0$ і обчислимо його найменший корінь $x_1=70-\sqrt{70^2-1}=70-\sqrt{4899}$. Те ж саме значення можна знайти по – іншому:

$$70-\sqrt{4899}=70-\sqrt{4899}\,\frac{70+\sqrt{4899}}{70+\sqrt{4899}}=\frac{1}{70+\sqrt{4899}}.$$

Нехай обчислення виконуються з точністю до 5 значущих цифр. Виконайте наступний код Octave.

```
digits(5)
sq=vpachar(sqrt(vpa(4899)));
x1=vpachar(70-sq)
x2=vpachar(1/vpachar(70+sq))
x3=70-sqrt(4899)
x1 = (sym) 0.0069580
x2 = (sym) 0.0071434
x3 =  0.00714322161154257
```

Останній результат `x3` отримано з процесорною точністю.

В розв'язку `x2` 4 значущих цифри вірні на відміну від `x1`, який (після округлювання) містить вірну лише одну цифру, хоча в обох випадках ми використовували значення кореня $\sqrt{4889.0}$ с точністю до 5 значущих цифр.

Ті ж самі результати ми отримаємо, якщо використаємо функцію `fround(...)`. Знайдемо `x1`, виконавши інструкцію (при `format long`).

```
fround(70-fround(sqrt(4899),5),5)
ans = 0.00700000000000001
```

Обчислення по іншій формулі з тією ж кількістю значущих цифр дає

```
fround(1/fround((70+fround(sqrt(4899),5)),5),5)
ans = 0.00714340000000001
```

Приклад 8. Нехай дані два еквівалентних вирази $f(x)=x\left(\sqrt{x+1}-\sqrt{x}\right)$ та $g(x)=\dfrac{x}{\sqrt{x+1}+\sqrt{x}}$. Обчислимо їх для `x=500` при одинарній точності (тобто при 7 значущих десяткових цифрах).

```
x=single(500);
f1=x*(sqrt(x+1)-sqrt(x))
g1=x/(sqrt(x+1)+sqrt(x))
z=500;
f2=z*(sqrt(z+1)-sqrt(z))
g2=z/(sqrt(z+1)+sqrt(z))
f1 =  11.1751556396484
g1 =  11.1747550964355
f2 =  11.1747553007469
g2 =  11.1747553007472
```

Вираз `g1` отриманий по другій формулі при використанні одинарної точності. Результати `f2` і `g2` отримано при подвійній точності (тобто точності Octave за замовчуванням). Порівнюючи відповіді, бачимо, що `g1` (тобто обчислення по другій формулі) містить значно меншу похибку ніж `f1`.

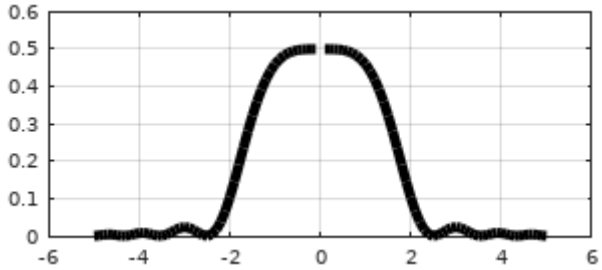
Отже, якщо для розв'язання задачі існує можливість використати формулу, яка не містить віднімання, то це треба зробити.

Приклад 9. Нехай дана функція $f(x)=\dfrac{1-\cos x^2}{x^4}$. Легко бачити, що $\lim\limits_{x\to 0}\dfrac{1-\cos x^2}{x^4}=\dfrac{1}{2}$. Дійсно, використовуючи в Octave символьну функцію `limit`, отримуємо (не забудьте завантажити пакет символьної математики).

```
limit((1-cos(x^2))/x^4,x,0)
ans = (sym) 1/2
```

Побудуємо графік функції $f(x)$, та обчислимо її значення в околі нуля. Для цього створимо наступний сценарій Octave.

```
x=-5:0.1:5;
function z=f(x)
  z=(1-cos(x.^2))./x.^4;
endfunction
y=f(x);
plot(x,y,'-k','Linewidth',5)
grid on
z1=f(0.01)
z2=f(0.001)
z3=f(0.0001)
>> ex015
```




```
z1 = 0.499999996961265
z2 = 0.500044450291171
z3 = 0
```

Перші дві відповіді вірні, а третя `z3=0` – ні. Це сталося через втрату точності.

Зміна кількості значущих цифр в арифметичних обчисленнях дає правильну відповідь.

```
>> digits(20)
>> xx=vpa('0.0001',20)
xx = (sym) 0.00010000000000000000000
>> f(xx)
ans =(sym) 0.50000354876421349015
```

■

В наступному прикладі показано, як зрізаний ряд Тейлора іноді допомагає уникнути помилки втрати значення.

Приклад 10. Нехай дана функція

$$f(x)=\frac{e^x-1-x}{x^2}.$$

Треба обчислити її значення в точці $x_0=0.01$. Чисельник функції містить віднімання і це може бути причиною досить великої похибки обчислень. Можна спробувати виконати обчислення функції $f(x)$ за допомогою її ряду Тейлора.

$$e^x=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}\ldots, \text{ то } f(x)=\frac{1}{2}+\frac{x}{6}+\frac{x^2}{24}+\ldots$$

Отже замість обчислення значень функції по «точній» формулі, при невеликих *x* можна спробувати обчислювати значення полінома

$$F(x)=\frac{1}{2}+\frac{x}{6}+\frac{x^2}{24}.$$

Виконаємо обчислювання в Octave при одинарній і подвійній точності.

```
x0=single(0.01);
x1=0.01;
function y=f(x)
  y=(exp(x)-1-x)./x.^2;
endfunction

function z=P(x)
  z=0.5+x./6+x.^2/24;
endfunction
```

```
y0=f(x0)
z0=P(x0)
y1=f(x1)
z1=P(x1)
y0 =  0.501778006553650
z0 =  0.501670837402344
y1 =  0.501670841679489
z1 =  0.501670833333333
```

Значення `y0` і `z0` отримані при одинарній точності обчислювань, а `y1` та `z1` – при подвійній (отже містять меншу похибку). Порівнюючи `y0`, `z0` з `y1` та `z1` бачимо, що обчислення полінома, тобто з використанням ряду Тейлора, містять значно меншу похибку, ніж обчислення за «точною» формулою.

■

З іншого боку використовувати ряд Тейлора теж треба з розумом.

Приклад 11. Ряд Тейлора для e^x (нестійкий алгоритм).

Ряд Тейлора для функції e^x має вигляд

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

і збігається при всіх x. Отже здається, що цю формулу можна застосовувати для обчислення e^x при будь якому x. Зрозуміло, що для великих значень x результат буде великим і може містити лише максимальну для використовуваного комп'ютера кількість значущих цифр.

Нижче наведено Octave код функції, яка підсумовує ряд Тейлора для функції e^x.

```
function s=texp(x)
  s=0;
  s1=1;
  t=1;
  i=1;
  while(s!=s1)      # закінчити підсумовування при відсутності
                      змін в частковій сумі ряду
    s=s1;
    t=t*(x/i);
    s1=s+t;
    i=i+1;
  endwhile
```

Зверніть увагу на перевірку, яка закінчує підсумовування. Вона враховує той факт, що машинна арифметики є лише наближеною. Вираз `s1=s+t` буде мати однакове значення з `s`, якщо число `t` досить мало.

Накажемо системі відображати максимальну кількість значущих цифр.

```
>> format long
```

Порівняємо результати роботи вбудованої функції `exp` і нашої `texp`. Для цього, наприклад, виконайте інструкцію

```
>> exp(-10),texp(-10)
```

```
ans =   4.53999297624849e-005
ans =   4.53999296704002e-005
```

Отриманими значеннями заповніть четвертий рядок наступної таблиці. Виконавши аналогічні інструкції, заповніть інші рядки таблиці.

x	$\exp(x)$	$\mathrm{texp}(x)$
30	10686474581524.5	10686474581524.5
10	22026.4657948067	22026.4657948067
1	2.71828182845905	2.71828182845905
-10	4.53999297624849e-005	4.53999296704002e-005
-30	9.35762296884017e-014	6.10304247889182e-006
-50	1.92874984796392e-022	2041.83296289762

Як бачите для від'ємних аргументів при зростанні модуля `x` функція `texp(x)` дає все більш недостовірні результати. Для $x > 0$ результати збігаються.

Щоб зрозуміти, що коїться, виконаємо наступний сценарій, обчисливши члени ряду при 5-и значущих цифрах.

```
N=35;  # кількість членів ряду
s=0.0;
x0=-10.0;

function z=ff(x,n)
  z=fround(fround(x.^n,5)./fround(factorial(n),5),5);
endfunction

for n = 0:N
  s=fround(s+fround(ff(x0,n),5),5);
  disp(fround(ff(x0,n),5));
endfor
s
 1.0000
-10.000
 50.000
-166.67
 416.67
-833.33
 1388.9
-1984.1
 2480.2
-2755.7
 . . .
-0.011310
 0.0037700
-0.0012161
s = 0.19304
```

Ми обмежилися при підсумовуванні 35 членами, оскільки наступні доданки вже не міняють суму (перевірте). Отриманий результат незадовільний, оскільки $e^{-10} = 0.0000453999\ 297$.

Аналізуючи доданки, бачимо, що приблизно 20 перших членів та відповідні їм проміжні суми на кілька порядків більше кінцевої відповіді, і вже містять помилку округлення майже рівну по величині остаточному результату. Тому сподіватися на достовірний результат не доводиться.

Це явище іноді називається катастрофічною втратою вірних знаків; воно часто зустрічається в погано продуманих обчисленнях. Звичайно, в обчисленнях можна утримувати більшу кількість значущих цифр з метою уникнути катастрофічної втрати знаків, однак, це завжди обходиться дорого як за часом виконання, так і по пам'яті.

Для розглядуваної задачі є набагато краще вирішення: обчислити суму в п’ятизначній десятковій арифметиці для $x = 10.0$ і потім взяти зворотне число:

$$e^{-10} = \frac{1}{e^{10}} = \frac{1}{1+10+50+166.67+\ldots} = 0.000045397.$$

Змініть в сценарії два рядки: в третьому рядку введіть `x0=10.0`, а в останньому – введіть `1/s`. Виконавши сценарій ви отримаєте

```
ans = 4.5397e-005
```

Приклад 12. Різницеве відношення для похідної. Похідну функції $f(x)$ в точці *x* можна приблизно обчислювати за формулою

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \equiv \Delta_h f(x).$$

Напишемо функцію Octave для обчислення похідної функції e^x.

```
function r=tdif(x,h)
  r=(exp(x+h)-exp(x))/h
```

Використовуючи її, побудуємо наступну таблицю. Для цього, наприклад, виконайте послідовність інструкцій

```
h=0.1, tdif(1,h), abs(tdif(1,h)-exp(1))
```

і отриманими результатами заповніть перший рядок таблиці. Виконавши аналогічні інструкції, змінюючи лише `h`, заповніть інші рядки таблиці.

h	$\Delta_h f(1)$	Абсолютна похибка
0.1	2.85884195487388	0.140560126414838
0.0001	2.71841774708292	1.35918623878961e-004
0.000001	2.71828318743061	1.35897156949838e-006
0.00000001	2.71828182185629	6.60275079056305e-009
0.0000000001	2.71828337616853	1.54770948368466e-006
0.000000000001	2.71871414270208	4.32314243038245e-004
0.00000000000001	2.70894418008538	0.00933764837366313
0.0000000000000001	0	2.71828182845905

Оскільки $\left(e^x\right)' = e^x$, то $\left(e^x\right)'\big|_{x=1} = e = 2.71828182845905$, і в ідеалі в середньому стовпці повинно стояти це значення. Але ми бачимо, що при зменшенні `h`

наближення до похідної, яке дається різницевим відношенням, спочатку покращується, а потім стає все гірше і гірше. Нуль, який стоїть в останньому рядку означає, що було досягнуто границю точності машинної арифметики (чисельник дробу дорівнює нулю).

Приклад 13. Розв'яжемо систему двох лінійних рівнянь

$$\begin{cases} 10x + 100y = 1, & \text{пряма L1} \\ 1000x + 10001y = 101, & \text{пряма L2} \end{cases}$$

за допомогою наступного коду Octave.

```
A=[10 100; 1000  10001];
b=[1; 101];
x=A^(-1)*b
x = -9.90000
     1.00000
```

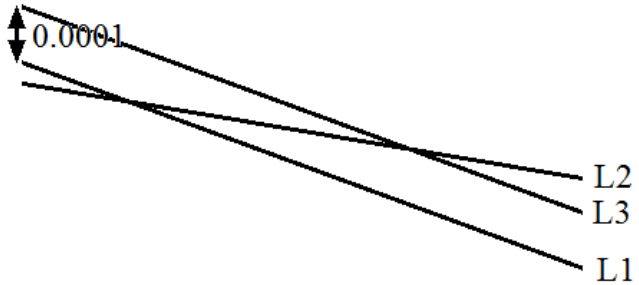
Розв'язком є пара чисел $(-9.9, 1.0)$. Змінимо в першому рівнянні праву частину на 0.01 і знову розв'яжемо «збурену» систему

$$\begin{cases} 10x + 100y = 1.01, & \text{пряма L3} \\ 1000x + 10001y = 101, & \text{пряма L2} \end{cases}.$$

Її розв'язком є числа $x = 0.101,\ y = 0.0$, які сильно відрізняються від розв'язку незбуреної системи.

Цей приклад показує, що невелика похибка в початкових даних може внести суттєві зміни в розв'язок. Може здатися, що ми використали поганий алгоритм, але це не так. В даному випадку кажуть, що задача погано обумовлена (некоректно поставлена або надчутлива). Це означає, що малі зміни коефіцієнтів задачі можуть привести до великих змін у розв'язку.

Причина чисельної нестійкості розв'язку геометрично інтерпретується тим, що на площині (x,y) прямі L1, L2, які визначаються рівняннями, майже паралельні одна одній, а пряма L3 паралельна L1 та трохи зсунута вгору. Але малий зсув прямої призводить до суттєвого зсуву точки перетину.



Отже точка перетину майже паралельних прямих сильно залежить від початкових даних.

Вправа. Розв'яжіть чисельно нестійку задачу – систему двох рівнянь з двома невідомими.

$$\begin{cases} 300x + 400y = 700 \\ 100x + 133y = B \end{cases}$$

при B=232, 233 та 234.

Відповідь: системи мають розв'язок (x = — 3,y = 4), (x = 2, y = 1/4) та (x = 3, y = —1/2) відповідно. Бачимо, що при зміні коефіцієнтів на частки відсотка розв'язок змінюється на сотні відсотків.

■

Пошук коренів полінома є стандартною обчислювальною задачею. Найчастіше вона вкрай чутлива до зміни даних. Наприклад, розглянемо поліном з майже кратними коренями $(x-1)^4 = 0.00000001$, тобто $1 - 4x + 6x^2 - 4x^3 + x^4 = 0.00000001$. Дійсними коренями цього рівняння є $x_{1,2} = 1 \pm 0.01$. Якщо праву частину замінити на 0.00000016 (тобто збільшити на 0.00000015), то дійсними коренями будуть числа $x_1 = 0.98$ і $x_2 = 1.02$. Тобто зміна вільного члена в сьомій десятковій цифрі змінює корені на 0.01.

Приклад 14. Дано поліном (приклад Уілкінсона, 1963)

$$p(x) = (x-1)(x-2)...(x-19)(x-20) = x^{20} - 210\ x^{19} + ...$$

Його коренями є числа 1, 2, ..., 19, 20. При введенні в комп'ютер коефіцієнтів такого полінома вони округлюються або відбувається відкидання двійкових розрядів, які не поміщаються в розрядну сітку мантиси. Припустимо, що лише для одного з двадцяти коефіцієнтів робиться помилка в молодшому двійковому розряді. Саме, припустимо, що коефіцієнт при x^{19} змінюється з -210 на $-210 + 2^{-23}$. З'ясуємо, який вплив така мала зміна справить на корені полінома. Для цього створимо і виконаємо наступний сценарій.

```
N=20;                 # степінь полінома
p=[1 -1];             # поліном x-1
for i = 2:N
  p=conv(p,[1 -i]);   # добуток полінома p на поліном x-i
 endfor
 p
dlt=0;
# dlt=2^(-23);
p(2)=p(2)+dlt;        # додаток до коефіцієнта при x^19
roots(p)              # обчислення коренів полінома p
```

При `dlt=0` ми отримуємо (для стислості ми наводимо корені в два стовпці в порядку від більшого до меншого).

```
     19.99987                    9.99119
     19.00130                    9.00271
     17.99367                    7.99939
     17.01854                    7.00010
     15.95972                    5.99999
     15.05933                    5.00000
     13.93019                    4.00000
     13.06266                    3.00000
     11.95887                    2.00000
     11.02246                    1.00000
```

Хоча обчислення виконувалися при подвійній точності, ми спостерігаємо деякі невеликі відхилення від точних значень: `1,2,...,19,20`. Однак всі корені дійсні.

Тепер покладіть `dlt=2^(-23)` і заново обчисліть сценарій. Ви отримаєте

```
  20.47678 +  1.03902i          9.50163 +  0.00000i
  20.47678 -  1.03902i          9.14747 +  0.00000i
  18.18133 +  2.54895i          7.99303 +  0.00000i
  18.18133 -  2.54895i          7.00030 +  0.00000i
  15.30595 +  2.77540i          5.99999 +  0.00000i
  15.30595 -  2.77540i          5.00000 +  0.00000i
  12.82177 +  2.12356i          4.00000 +  0.00000i
  12.82177 -  2.12356i          3.00000 +  0.00000i
  10.89297 +  1.14956i          2.00000 +  0.00000i
  10.89297 -  1.14956i          1.00000 +  0.00000i
```

Мала зміна в коефіцієнті -210 має наслідком те, що десять коренів зробилися комплексними і деякі з них відсунулися від дійсної осі більш ніж на 2 одиниці.

Причина, по якій ці корені так сильно змінилися, полягає не в помилках округлення і не пов'язана з вибором алгоритму обчислення; суть в чутливості самої задачі. Проведемо невеликий аналіз того, що сталося. Запишемо поліном у вигляді $p(x,\alpha)=x^{20}-\alpha x^{19}+\dots$, і потім знайдемо похідну по α для кожного кореня. Це робиться диференціюванням рівняння $p(x,\alpha)=0$ по α.

$$\frac{\partial p(x,\alpha)}{\partial x}\frac{\partial x}{\partial \alpha}+\frac{\partial p(x,\alpha)}{\partial \alpha}=0,$$

$$\frac{\partial x}{\partial \alpha}=-\frac{\partial p/\partial \alpha}{\partial p/\partial x}=\frac{x^{19}}{\sum\limits_{i=1}^{20}\prod\limits_{j=1,\,j\neq i}^{20}(x-j)}.$$

Обчислюючи цей вираз для кожного кореня, отримуємо

$$\left.\frac{\partial x}{\partial \alpha}\right|_{x=i}=\frac{i^{19}}{\prod\limits_{j=1,\,j\neq i}^{20}(i-j)},\ i=1,2,\dots,20.$$

Ці величини зображують швидкість зміни кореня при зміні параметра α. Зміна коефіцієнта α на $\alpha+\varepsilon$ змінює *i*-й корінь приблизно на $\varepsilon\cdot\left.\frac{\partial x}{\partial \alpha}\right|_{x=i}$.

Створимо функцію для обчислення коефіцієнтів швидкості і надрукуємо їх значення.

```
function dxda = difpa(i)
  X1 = i - [1:i-1];
  X2 = i - [i+1:20];
  X = [X1 X2];
  dxda=i^19/prod(X);
```

В наступній таблиці наведено їх значення, які отримано інструкціями типу

```
>> difpa(20)
```

```
ans = 4.3100e+007
```

Корінь	$\left.\frac{\partial x}{\partial \alpha}\right\|_{x=i}$
1	-8.2206e-018
2	8.1890e-011
...	...
10	7.5941e+006
...	...
18	9.9559e+008
19	-3.0901e+008
20	4.3100e+007

■

У наведеному прикладі причина чисельної нестійкості міститься в математичному формулюванні задачі і не може бути усунена вибором методу або алгоритму розв'язання. Якщо повернутися до прикладу 11 обчислення e^{-10} за допомогою ряду Тейлора, то він показав, що навіть для добре поставленої задачі існують погано продумані алгоритми, які можуть приводити до незадовільних результатів. В тому прикладі труднощі були подолані за допомогою зміни алгоритму. Для деяких задач «гарні» відповіді неможливо отримати ніяким алгоритмом, тому що задача чутлива до малих помилок, які допущені при представлені даних. Попередній приклад 13 системи рівнянь наочно продемонстрував таку задачу.

Нестійкі алгоритми і чутливі задачі зустрічаються майже у всіх галузях обчислювальної математики. Нижче ми розглянемо ще один приклад нестійкого алгоритму.

Приклад 15. Припустимо, що ми бажаємо обчислити інтеграли

$$E_n = \int_0^1 x^n e^{x-1}\, dx, \quad n = 1, 2, \ldots.$$

Інтегруючи частинами, отримуємо

$$\int_0^1 x^n e^{x-1}\, dx = x^n e^{x-1}\Big|_0^1 - \int_0^1 n x^{n-1} e^{x-1}\, dx,$$

або $E_n = 1 - nE_{n-1}$, $n = 2, 3, \ldots$, де $E_1 = 1/e$.

Використовуючи отримане рекурентне співвідношення, обчислимо 14 перших значень E_n з одинарною та подвійною точностями.

```
t1=single(exp(-1));
t2=exp(-1);
for n = 2:14
  t1=1-n.*t1;   # обчислення з одинарною точністю
  t2=1-n.*t2;   # обчислення з подвійною точністю
  disp(["E",num2str(n),"=",num2str(t1,7),"\t ",num2str(t2,15)])
endfor
```

```
E2=0.2642411     0.264241117657115
E3=0.2072767     0.207276647028654
```

```
E4=0.1708932      0.170893411885384
E5=0.145534       0.14553294057308
E6=0.1267958      0.12680235656152
E7=0.1124296      0.112383504069363
E8=0.100563       0.100931967445092
E9=0.09493256     0.0916122929941707
E10=0.05067444    0.0838770700582927
E11=0.4425812     0.0773522293587803
E12=-4.310974     0.0717732476946367
E13=57.04266      0.0669477799697233
E14=-797.5973     0.0627310804238732
```

Хоча підінтегральний вираз $x^{12}e^{x-1}$ додатний на інтервалі (0,1), обчислене значення для E_{12} при одинарній точності від'ємне. Єдина помилка, яка зроблена в наших обчисленнях, – це помилка в E_1, коли `1/e` округлюється до 7 значущих цифр. Оскільки рекурентна формула, яка отримана інтегруванням частинами, є точною, то помилка в E_{12} повністю зобов'язана помилці округлення, зробленій в E_1. Щоб зрозуміти, як помилка в E_1, яка приблизно дорівнює 10^{-8}, стає настільки великою, зауважимо, що вона множиться на –2 при обчисленні E_2, потім помилка в E_2 множиться на –3 при обчисленні E_3 і т. д. Таким чином, помилка в E_{12} є в помилка в E_1 помножена на (–2) (–3)....(–12) = –12! або приблизно `–4.79` (при одинарній точності).

```
>> factorial(12)*1e-8
ans = 4.79001600000000
```

Це величезне збільшення помилки вхідних даних є результатом обраного алгоритму. Чи існує інший алгоритм, який не має подібної нестійкості? Якщо ми перепишемо рекурентне співвідношення у вигляді

$$E_{n-1} = \frac{1-E_n}{n}, \quad n = \ldots, 3, 2,$$

то на кожному кроці обчислення помилка в E_n буде множитися на 1/n. Таким чином, якщо ми почнемо зі значення для деякого E_n при $n >> 1$ і будемо вести обчислювання в зворотному напрямку, то будь-яка початкова помилка або проміжні помилки округлень будуть зменшуватися на кожному кроці. Це і називається стійким алгоритмом. Щоб знайти початкове значення, зауважимо, що

$$E_n = \int_0^1 x^n e^{x-1}\, dx \le \int_0^1 x^n\, dx = \left.\frac{x^{n+1}}{n+1}\right|_0^1 = \frac{1}{n+1}.$$

Отже, E_n прямує до нуля, коли n прямує до ∞. Наприклад, якщо ми апроксимуємо нулем число E_{20} і візьмемо цей нуль в якості стартового значенням, то зробимо початкову помилку, яка не перевищує 1/21. Ця помилка множиться на 1/20 при обчисленні E_{19}, так що помилка в E_{19} не перевищує $(1/20)\cdot(1/21) \approx 0.0024$. До часу обчислення E_{12} початкова помилка зменшиться до величини, яка менша $4\cdot 10^{-8}$, що в свою чергу менше помилки округлення (при одинарній точності). Проводячи ці обчислення, отримуємо

```
t1=single(0.0);
t2=0.0;
for n = 20:-1:10
  t1=(1-t1)/n;    # обчислення з одинарною точністю
  t2=(1-t2)/n;    # обчислення з подвійною точністю
  disp(["E",num2str(n-1),"=",num2str(t1,7),"\t ",num2str(t2,15)])
endfor
E19=0.05         0.05
E18=0.05         0.05
E17=0.05277778   0.0527777777777778
E16=0.05571895   0.055718954248366
E15=0.05901757   0.0590175653594771
E14=0.06273217   0.0627321623093682
E13=0.06694771   0.066947702692188
E12=0.07177325   0.0717732536390625
E11=0.07735223   0.0773522288634115
E10=0.08387707   0.0838770701033262
E9=0.09161229    0.0916122929896674
```

Коли ми дійдемо до E_{15}, початкова помилка в E_{20} вже зовсім придушена стійкістю алгоритму і обчислені значення від E_{15} до E_9 точні у всіх шести значущих цифрах, з похибкою округлення в останній (сьомій) цифрі.

Отже бездоганна з точки зору аналітичної математики формула $E_n = 1 - nE_{n-1}$ абсолютно не придатна з точки зору обчислювальної математики, оскільки неминуча похибка стартового значення E_0 або E_1 при обчислюванні E_n збільшується в $n!$, тобто катастрофічно зростає.

■

В деяких задачах погано збіжний алгоритм можна замінити двома, кожний з яких обчислюється краще.

Приклад 16. Розглянемо функцію

$$f(x) = \sum_{n=1}^{\infty} \left(1 + \frac{1}{n^5}\right) x^n .$$

Цей ряд збігається при $|x| < 1$. Нехай $x = 0.9999$. Збіжність буде поганою і навіть для досягнення відносної похибки $\varepsilon = 10^{-3}$ потрібно просумувати велику кількість членів ряду. Перепишемо формулу в наступному вигляді

$$f(x) = \sum_{n=1}^{\infty} x^n + \sum_{n=1}^{\infty} \frac{x^n}{n^5} .$$

Перша сума являє нескінченно спадною геометричною прогресією, і тому

$$\sum_{n=1}^{\infty} x^n = \frac{x}{1-x},$$

а другий ряд збігається дуже швидко.

■

Підведемо деякий підсумок. Існуючі комп'ютери використовують зображення дійсних чисел у формі з плаваючою крапкою, а значить фіксовану кількість

значущих десяткових цифр, оскільки двійкове подання мантиси містить обмежену кількість двійкових розрядів. Звідси випливає, що при обмеженій кількості значущих цифр чисельні алгоритми розв'язання задач повинні «вижимати» з методів максимально можливу точність.

З розглянутих прикладів випливають деякі правила підрахунку похибок арифметичних обчислень:

- при додаванні або відніманні чисел їх абсолютні похибки складаються;
- при множенні або діленні чисел друг на друга їх відносні похибки складаються;
- при піднесенні в ступінь наближеного числа його відносна похибка збільшується на показник ступеня.

З досвіду обчислень випливають деякі спрощені правила урахування значущих цифр:

- при додаванні і відніманні наближених чисел в результаті слід зберігати стільки десяткових знаків, скільки їх в наближеному даному з найменшою кількістю десяткових знаків;
- при множенні і діленні в результаті варто зберігати стільки значущих цифр, скільки їх має наближене дане з найменшою кількістю значущих цифр;
- у проміжних результатах слід зберігати однією цифрою більш, ніж рекомендують два попередні правила; в остаточному результаті ця "запасна" цифра відкидається.

Література

1. Каханер Д., Моулер К, Нэш С. Численные методы и програмное обеспечение. – М:. Мир, 1998. – 575с.
2. И.С. Березин, Н.П. Жидков Методы вычислений. Т.1, ГИФМЛ, М. – 1962, 464с.
3. Березин, Жидков Т.2, ГИФМЛ, М. – 1959, 620с.
4. Джон Г. Мэтьюз, Куртис Д. Финк. Численные методы. Использование Matlab. – М. + К.: Вильямс, 2001. – 720с.
5. Дж. Форсайт, М. Малькольм, К. Моулер Машинные методы математических вычислений. – М. – 1980, Мир. 280с.
6. Дж. Форсайт, К. Моулер Численное решение систем линейный алгебраических уравнений. – М. – 1969, Мир. 167с.
7. Д. Поттер Вичислительные методы физики., 390с.
8. Турчак Л.И., Плотников П.В. Основы численных методов. Учебное пособие. – М:. Физматлит, 2003. – 304с.
9. Самохин А, Б., Самохина А. С.-Численные методы и программирование на Фортране-1996 .djvu
10. Каханер Д., Моулер К., Нэш С. Численные методы и программное обеспечение. – М.: Мир, 1998, 575с.
11. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. - Численные методы - 2002.djvu