МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Чисельні методи на Python

Навчальний посібник для студентів напрямку
„Системна інженерія”

Затверджено Методичною радою НТУУ “КПІ”

Київ
“ПОЛІТЕХНІКА”
2010

Чисельні методи на Python. Навчальний посібник для студентів напрямку "Системна інженерія" / Уклад.: Я.Ю. Дорогий, Є.В.Глушко, А.Ю.Дорошенко. – К.: ІВЦ "Політехніка", 2010. – 281 с.

Гриф надано Методичною радою НТУУ "КПІ"
(Протокол №)

Н а в ч а л ь н е в и д а н н я

Чисельні методи на Python

Навчальний посібник для студентів напрямку
„Системна інженерія"

Затверджено Методичною радою НТУУ "КПІ"

Укладачі: Дорогий Ярослав Юрійович
Глушко Євгеній Васильович
Дорошенко Анатолій Юхимович

Відповідальний редактор Теленик Сергій Федорович

Рецензент Теленик Сергій Федорович

Python – універсальна інтерпретована, об'єктно-орієнтована мова програмування високого рівня, ідеально пристосована для *Web* та швидкої розробки застосувань научного спрямування. Вона розвивається у відкритих вихідних кодах на багатьох платформах та надається безкоштовно для загального користування.

Мова програмування *Python* відкриває шлях до безлічі самих сучасних програмних технологій.

ЗМІСТ

Розділ 1. Вступ

1.1. Про автоматизацію розв’язування задач проектування

У роботі інженера, що проектує новий технологічний процес чи виріб, органічно сполучаються наука та мистецтво. Розробник не просто вивчає досягнення науки і мистецтва, а прагне використовувати їх для потреб практики. Процес інженерної творчості починається звичайно з усвідомлення потреби в новому виробі чи технологічному процесі. Задача інженера – знайти прийнятне рішення та представити його в такому вигляді, щоб його можна було втілити у виробництво.

Проектування починається з ретельного вивчення можливих рішень. Будується *фізична модель чи простий прототип* розроблювального об'єкта. Потім збирається інформація, що дозволяє побудувати модель розроблювального виробу чи процесу, щоб оцінити та перевірити правильність прийнятого рішення. Необхідність цього етапу обумовлена економічними міркуваннями, тому що практична перевірка рішення майже завжди обходиться дуже дорого, віднімає багато часу та вимагає занадто великих матеріальних й енергетичних витрат. Для цього, як правило, розробляється *математична модель* або використовується поєднання простого прототипу та математичної моделі. Побудувавши модель, приступають до вивчення її властивостей, прагнучи з'ясувати, якою мірою розроблювальний виріб відповідає своєму призначенню. При цьому для того, щоб знайти задовільне рішення, будують *обчислювальну модель* проектованого об'єкта. Побудова саме такої моделі та вивчення її властивостей може повторюватись доти, поки не з'явиться впевненість, що знайдене найкраще можливе рішення.

У процесі проектування доводиться виконувати найрізноманітніші обчислення, характер і обсяг яких може змінюватись. Обчислення, для виконання яких доводиться використовувати комп'ютерні програми та системи автоматизації, можна розділити на такі категорії:

1. Обчислення великих обсягів в силу їх *ітераційного характеру*.

2. Громіздкі обчислення, тому що вимагають забезпечення необхідної *точності.*

3. *Візуалізація* графічного представлення даних.

Характер роботи інженера визначає багаторазову повторюваність розв'язуваних їм математичних задач, серед яких – розв’язування алгебраїчних та трансцендентних рівнянь, апроксимація й інтерполяція функцій, розв’язування задач на власні значення, розв’язування звичайних диференціальних рівнянь, розв’язування задач оптимізації та ін.

У цьому посібнику з чисельних методів розглядаються ці задачі, а також роз'ясняються основні поняття та терміни, пов'язані з методами їхнього розв’язування. Обчислювальні методи доводяться до обчислювальних алгоритмів розв’язування інженерних задач, що можуть бути реалізовані різними мовами програмування (в даному посібнику використана мова функціонального програмування Python) на різних обчислювальних пристроях, причому наголос робиться на виборі оптимального методу розв’язування задачі.

1.2. Обчислювальні методи й алгоритми

Побудова математичної моделі об'єкта дозволяє поставити задачу його вивчення як формально-теоретичну. Після цього настає другий етап дослідження – пошук методу розв’язування сформульованої математичної задачі. Варто мати на увазі, що в прикладних роботах нас, як правило, цікавлять кількісні значення величин, тобто відповідь повинна бути доведена "до числа". При цьому всі розрахунки проводяться з числами, записаними у виді скінчених десяткових дробів, і тому результати обчислень завжди принципово носять *наближений характер*. Важливо тільки домогтися того, щоб похибки укладалися в рамки необхідної точності.

1.2.1. Поняття алгоритму та граф-схеми алгоритму

У більшості задач, що зустрічаються в математиці, відповідь дається у вигляді *формули.* Формула, у силу встановлених математичних правил,

визначає послідовність математичних операцій, яку потрібно виконати для обчислення шуканої величини. Наприклад, формула коренів квадратного рівняння дозволяє знайти їх за значеннями коефіцієнтів цього рівняння, формула Герона виражає площу трикутника через довжини його сторін і т.д.

Однак існує багато відомих задач, для яких відповідь легко може бути знайдена, хоча вона і не записується у виді формули. Наприклад, у школі при вивченні цілих чисел та арифметичних операцій над ними замість "формули" обчислення суми декількох чисел розглядають *правило обчислень* за допомогою порозрядного додавання в стовпчик. Це правило цілком вирішує поставлену задачу: воно визначає послідовність математичних операцій, яку потрібно виконати для обчислення шуканої величини.

Обчислювальний метод розв'язування задачі – це визначена послідовність операцій в обчислювальній моделі задачі, тобто обчислювальний алгоритм, мовою якого в кінцевому рахунку є числа й арифметичні дії. Така примітивність мови дозволяє реалізувати чисельні методи на обчислювальних машинах, що робить ці методи потужним та універсальним інструментом дослідження. Однак задачі формулюються, як правило, звичайною математичною мовою (рівнянь, функцій, диференціальних операторів і т.п.). Тому потрібна розробка чисельного методу, що необхідно допускає заміну, апроксимацію задачі іншою, близькою до початкової задачі та сформульованою в термінах об'єктів і операцій предметної області (у найпростішому випадку, чисел та арифметичних операцій).

Таким чином, *алгоритмізація розв'язування задач* припускає перехід від математичної до обчислювальної моделі задачі та побудови послідовності операцій алгоритму для такого розв'язування. Алгоритми розв'язування задач повинні реалізовуватися мовами програмування та виконуватися на комп'ютерах автоматизованої системи. Для *незалежного* (від мови програмування та архітектури комп'ютерної системи) *представлення алгоритмів* розв'язування задач у цьому посібнику будемо користуватися

деякою абстрактною моделлю обчислювача й уніфікованих засобів опису алгоритмів у виді *граф-схем алгоритмів*.

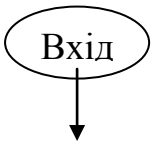
Будемо вважати, що *абстрактний обчислювач* з пам'яттю довільного доступу складається з таких компонентів:

1) процесор з універсальним набором команд (оператори присвоювання, умовний та циклічний оператори) і необхідним набором функцій;

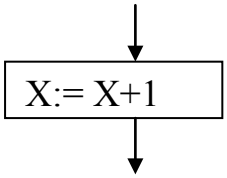
2) оперативна пам'ять з довільним доступом (RAM), що складається із зліченої множини регістрів, для адресації яких використовуються змінні обчислювальної моделі;

3) нескінченної в обидва боки стрічки вводу-виводу (довготермінової пам'яті), відкіля беруться вхідні дані програм та куди поміщаються результати обчислень.

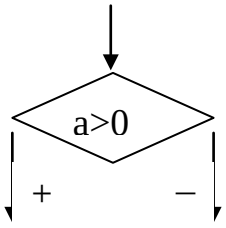
Граф-схема алгоритму є орієнтованим графом, що складається з вершин-операторів та вершин-розпізнавачів, а також стрілок зв’язків, що їх поєднують. Вершини-оператори здійснюють перетворення (обчислення), вершин-розпізнавачі – керують процесом виконання програми. Граф-схеми використовують наступні графічні конструкції програмування:



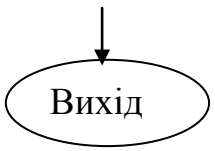
вхідна вершина; єдина крапка входу для алгоритму, куди подаються вхідні дані;



вершина-перетворювач; прямокутник, що містить усередині обчислювальні оператори;



вершина-розпізнавач; реалізує умовні та циклічні оператори;



вихідна вершина, означає завершення обчислень алгоритму.

1.2.2. Особливості чисельних методів

Отже, для розв’язування математичної задачі важливо вказати систему правил, що задає строго визначену послідовність операцій, що приводять до шуканої відповіді. Таку систему правил називають *алгоритмом.* Поняття алгоритму в його загальному вигляді відноситься до основних понять математики.

Алгоритми розв’язування багатьох математичних задач, для яких не вдається одержати відповідь у виді формули, засновані на наступній процедурі: будується нескінченний процес, що збігається до шуканого розв’язку. Він обривається на деякому кроці (оскільки обчислення не можна продовжувати нескінченно), і отримана в такий спосіб величина приймається за наближений розв’язок розглянутої задачі. Збіжність процесу гарантує, що для будь-якої заданої точності $\varepsilon > 0$ знайдеться такий номер кроку n_ε, що при цьому похибка у визначенні розв’язку задачі не перевищить ε. Слова "наближений розв’язок" не означають "розв’язок другого сорту". Якщо буде отримана в якій-небудь задачі відповідь у вигляді формули та потрібно буде підрахувати по ній значення потрібної величини, то через представлення чисел при обчисленнях скінченими десятковими дробами можна одержати для неї наближене значення бажаної точності. Проблема застосування алгоритмів, що використовують нескінченний збіжний процес не в наближеному характері відповіді, а у великому обсязі необхідних обчислень. Не випадково такі алгоритми прийнято називати обчислювальними алгоритмами, а засновані на них методи розв’язування математичних задач – *чисельними методами.* Широке застосування обчислювальних алгоритмів стало можливим тільки завдяки ЕОМ. До їхньої появи чисельні методи використовувалися рідко і тільки в порівняно простих випадках через надзвичайну трудомісткість обчислень, що виконувались вручну.

Поняття чисельного методу розв'язування задачі як правило означає заміну початкової задачі іншою, близькою до неї та сформульованою в термінах чисел і обчислювальних операцій. Незважаючи на всю розмаїтість способів такої заміни, деякі загальні властивості притаманні їм усім. Звернемося до найпростішого прикладу.

Нехай потрібно знайти розв'язок рівняння:

$$x^2 - a = 0, \; a>0, \tag{1.1}$$

тобто добути квадратний корінь із заданого числа a. Можна, звичайно, написати $x = \sqrt{a}$, але символ $\sqrt{\ }$ не вирішує справи, бо не дає способу обчислення величини x.

Вчинимо в такий спосіб. Задамося яким-небудь початковим наближенням x_0 (наприклад, $x_0 = 1$) та будемо послідовно за допомогою формули

$$x_n = \frac{1}{2}(x_{n-1} + \frac{a}{x_{n-1}}) \tag{1.2}$$

обчислювати значення $x_1, x_2, \ldots$ Перервемо цей процес на деякому $n = N$ та отримане в результаті значення x оголосимо наближеним розв'язуванням вихідної задачі (1.1), тобто покладемо $\sqrt{a} = x_N$.

Правомірність такого припущення залежить, очевидно, від вимог, пропонованих до точності розв'язування, від величини a та від параметра N. Якщо мати на увазі будь-які вимоги, то потрібно довести, що для будь-якого $a \geq 0$ відповідним вибором N можна домогтися будь-якої близькості x до точного значення $\sqrt{a}$.

Доведемо, що наш алгоритм (1.2) задовольняє цій умові. Покладемо

$$\frac{x_n}{\sqrt{a}} = 1 + \varepsilon_n \qquad (1.3)$$

Розділимо рівність (1.2) на $\sqrt{a}$ та підставимо в нього (1.3), одержимо

$$1 + \varepsilon_n = \frac{1}{2}(1 + \varepsilon_{n-1} + \frac{1}{1 + \varepsilon_{n-1}}),$$

звідки маємо

$$\varepsilon_n = \frac{1}{2}(\varepsilon_{n-1} - 1 + \frac{1}{1 + \varepsilon_{n-1}}) = \frac{1}{2}\frac{\varepsilon_{n-1}^2}{\varepsilon_{n-1} + 1} \qquad (1.4)$$

Оскільки $1 + \varepsilon_0 = \frac{x_0}{\sqrt{a}} = \frac{1}{\sqrt{a}} > 0$, то з останньої рівності випливає, що всі ε_n, починаючи з першого, позитивні. Використовуючи це, одержуємо з (1.4)

$$\frac{1}{2}\frac{\varepsilon_{n-1}^2}{\varepsilon_{n-1} + 1} < \frac{1}{2}\frac{\varepsilon_{n-1}^2}{\varepsilon_{n-1}} = \frac{1}{2}\varepsilon_{n-1}.$$

І, отже,

$$\varepsilon_n < \frac{1}{2}\varepsilon_{n-1} \qquad (1.5)$$

тобто ε_n спадає з ростом n швидше, ніж геометрична прогресія зі знаменником ½. Отже,

$x_N \to \sqrt{a}$, при $N \to \infty$ і наше твердження доведено.

Нижченаведений рис. 1.1 ілюструє ітераційний процес (1.2).Тут зображені два графіки: лівої $y_л(x)$ та правої $y_п(x)$ частин (1.2). Оскільки, очевидно, $y_л(\sqrt{a}) = y_п(\sqrt{a})$, то ці графіки перетинаються в точці $x = \sqrt{a}$. Проведення ітерацій за формулою (1.2) еквівалентно руху по зображеній на

рисунку ламаній лінії, затиснутої між $y_л(x)$ та $y_п(x)$. Це ще раз переконує нас у збіжності ітерацій до $\sqrt{a}$ при $N \to \infty$.

При дослідженні збіжності була допущена деяка ідеалізація алгоритму, мовчки припустивши можливість точної реалізації обчислень за формулою (1.2). Проте ні людина, ні машина не можуть оперувати з довільними дійсними числами. Обчислення завжди ведуться з обмеженою кількістю десяткових знаків, і точність результату не може перевершувати точність розрахунків. Важливо встановити, у якому відношенні ці точності знаходяться, і чи не будуть похибки, що допускаються при округленні, накопичуватись, позбавляючи результат якої-небудь цінності.

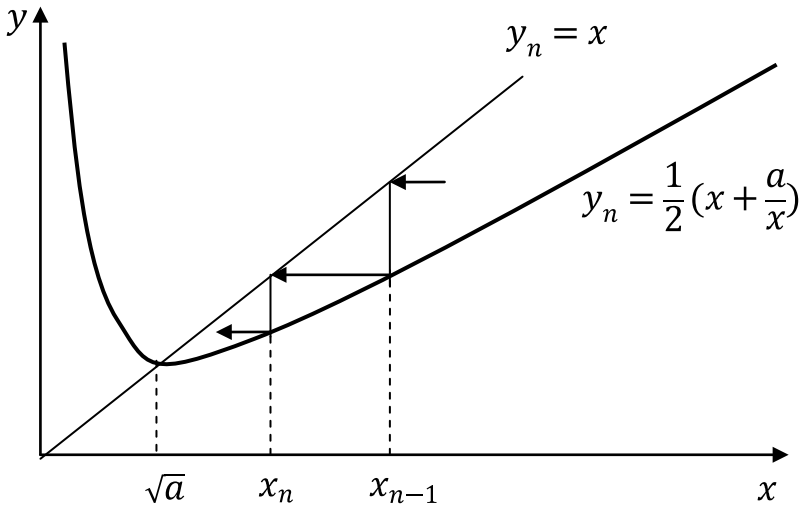


Рис. 1.1. Графічна інтерпретація збіжності

Проведемо формальну перевірку впливу зазначеного фактора. Роль округлень зводиться до того, що фактично замість формули (1.2) ми користуємося формулою

$$x_n' = \frac{1}{2}(x_{n-1}' + \frac{a}{x_{n-1}'})(1 + \partial_n) \qquad (1.6)$$

де множник $1 + \partial_n$ ефективно враховує помилку, що вводиться округленнями на даному n-му кроці розрахунку, а x_n' – фактично

одержувана послідовність. Величина ∂_n<<1 характеризує точність обчислень, Заміняючи $x_n^{'}$ на $\sqrt{a}(1+\varepsilon_n)$, одержимо замість (1.6)

$$\sqrt{a}(1+\varepsilon_n)=\frac{1}{2}(\sqrt{a}(1+\varepsilon_{n-1})+\frac{a}{\sqrt{a}(1+\varepsilon_{n-1})})(1+\partial_n),\qquad 1+\varepsilon_n=\frac{2+2\varepsilon_{n-1}+\varepsilon_{n-1}^2}{2(1+\varepsilon_{n-1})}(1+\partial_n)$$

$$\varepsilon_n=\frac{2+2\varepsilon_{n-1}+\varepsilon_{n-1}^2}{2(1+\varepsilon_{n-1})}(1+\partial_n)-1=\frac{\partial_n(2+2\varepsilon_{n-1}+\varepsilon_{n-1}^2)+\varepsilon_{n-1}^2}{2(1+\varepsilon_{n-1})}=\partial_n+\frac{1}{2}\frac{\varepsilon_{n-1}^2}{\varepsilon_{n-1}+1}(1+\partial_n).$$

Звідси видно, що з ростом n ε_n спадає до величини порядку ∂_n, тобто точність результату відповідає точності обчислень.

Незважаючи на свою елементарність, розглянутий приклад цілком демонструє наступні принципи, загальні для всіх чисельних методів:

— вихідна задача (1.1) заміняється іншою задачею – обчислювальним алгоритмом (1.2);

— задача (1.2) містить параметр *N*, якого немає у вихідній задачі;

— вибором цього параметра можна домогтися, у принципі, будь-якої близькості розв’язку другої задачі до розв’язку першої;

— нарешті, неточна реалізація алгоритму, викликана округленнями, не змінює істотно його властивостей.

1.3 Контрольні запитання

1. Які типи моделей бувають?
2. Що таке абстрактний обчислювач?
3. Що таке граф-схема алгоритму?
4. Які основні елементи граф-схеми алгоритму?
5. У чому суть чисельних методів?
6. Що таке збіжність?
7. Назвіть принципи чисельних методів.

Розділ 2. Наближені числа й оцінка похибок обчислень

2.1. Наближені числа. Класифікація похибок

У процесі обчислень часто треба мати справу з наближеними числами. Джерелами неточностей при цьому є такі:

— математичний опис задачі, зокрема, *неточно задані початкові дані* такого опису;

— застосований *обчислювальний метод не є точним* та дає наближений результат;

— при вводі, обчисленнях та виводі даних здійснюються *округлення* чисел.

Відповідно, виникають і три види похибок:

— *неусувна похибка* – що виникає з природи неточності самої математичної моделі;

— *похибка обчислювального методу;*

— *машинна похибка.*

Нехай A – точне значення деякої величини, надалі будемо називати його точним числом A. Наближеним значенням величини A, або наближеним числом, називається число a, що заміняє точне значення величини A. Якщо $a<A$, то a називається *наближенням A з недостачею,* якщо ж $a>A$, – то *з надлишком.* Наприклад, значення *3,14* є наближеним значенням числа π з недостачею, а *3,15* – з надлишком. Для характеристики ступеня точності даного наближення користаються поняттям похибки або помилки.

Похибкою Δa наближеного числа a називається різниця виду:

$$\Delta a = A - a, \qquad (2.1)$$

де A – відповідне точне число.

Абсолютною похибкою Δ наближеного числа *a* називається абсолютна величина похибки цього числа

$$\Delta = |A - a|. \qquad (2.2)$$

У силу того, що точне число *A*, як правило, невідоме, то користуються поняттям граничної абсолютної похибки. Граничною абсолютною похибкою Δ_a наближеного числа *a* називається число, не менше абсолютної похибки цього числа, тобто:

$$\Delta_a \geq \Delta. \qquad (2.3)$$

З (2.3) маємо

$$\Delta_a \geq |A - a|, \qquad (2.4)$$

отже,

$$a - \Delta_a \leq A \leq a + \Delta_a \qquad (2.5)$$

тобто $a - \Delta_a$ є наближенням числа *A* з недостачею, а $a + \Delta_a$ – наближенням числа *A* з надлишком. Формулу (2.5) коротко записують у вигляді $A = a \pm \Delta_a$.

На практиці під точністю вимірів звичайно розуміють граничну абсолютну похибку. Наприклад, якщо відстань між двома пунктами, що дорівнює $S = 900$ м, отримано з точністю до 0,5 м, то точне значення величини *S* знаходиться в границях $899{,}5 \text{ м} \leq S \leq 900{,}5$ м.

Введення абсолютної чи граничної абсолютної похибки звичайно недостатньо для характеристики ступеня точності наближених чисел. Істотним показником точності наближених чисел є їхня відносна похибка.

Відносною похибкою δ наближеного числа *a* називається відношення абсолютної похибки Δ цього числа до модуля відповідного точного числа *A* ($A \neq 0$), тобто.

$$\delta = \frac{\Delta}{|A|} \qquad (2.6)$$

Граничною відносною похибкою наближеного числа a називається число δ_a, не менше від відносної похибки цього числа, тобто.

$$\delta_{a\geq}. \delta \qquad (2.7)$$

З (2.7) маємо

$$\Delta \leq |A|\ \delta_a \qquad (2.8)$$

Отже, можна вважати, що гранична абсолютна похибка числа a дорівнює

$$\Delta_a = |A|\ \delta_a \qquad (2.9)$$

Якщо прийняти $A \approx a$, то формула (2.9) прийме вид

$$\Delta_a = |a|\ \delta_a \qquad (2.10)$$

Отже, точне число A лежить у межах:

$$a(1- \delta_a) \leq A \leq a(1 + \delta_a). \qquad (2.11)$$

Формула (2.10) дозволяє визначати граничну абсолютну похибку за заданою граничною відносною похибкою і навпаки.

2.2. Значуща цифра. Число вірних знаків

Будь-яке число a може бути представлене у виді скінченої чи нескінченної суми доданків:

$$a = \pm(a_m 10^m + a_{m-1} 10^{m-1} + ... + a_{m-n+1} 10^{m-n+1} + ...), \qquad (2.12)$$

де a_i - цифри числа a (a_i = 0, 1, 2, . . ., 9), m – деяке ціле число, що називається старшим десятковим розрядом числа a, n – кількість значущих цифр. Наприклад, число 476,93 може бути представлене у вигляді:

$476{,}93 = 4\cdot 10^2 + 7\cdot 10^1 + 6\cdot 10^0 + 9\cdot 10^{-1} + 3\cdot 10^{-2}$

При проведенні реальних обчислень усяке число, що має вид нескінченної суми доданків, заміняється сумою скінченого числа доданків, тобто замість суми (2.12) записують суму виду:

$b = \pm(b_m 10^{m} + b_{m-1} 10^{m-1} + ... + b_{m-n+1} 10^{m-n+1} + ...), \quad b_m \neq 0$ (2.13);

Значущими цифрами наближеного числа b називаються всі цифри (десяткові знаки) b_i, починаючи з першої ненульової зліва $b_m \neq 0$.

Значущу цифру називають *вірною*, якщо абсолютна похибка числа не перевищує ½ одиниці розряду, що відповідає цій цифрі. Наприклад,

$0{,}009801 = 9 \cdot 10^{-3} + 8 \cdot 10^{-4} + 0 \cdot 10^{-5} + 1 \cdot 10^{-6}$, значущі цифри – 9801;

$980010 = 9 \cdot 10^{5} + 8 \cdot 10^{4} + 0 \cdot 10^{3} + 1 \cdot 10 + 0 \cdot 10^{0}$, значущі цифри – 980010.

Якщо в даному числі 0,073040 остання цифра не є значущою, то це число повинне бути записане у виді 0,07304.

По виду чисел при звичайному їхньому записі важко судити про точну кількість значущих цифр цих чисел, однак цієї невизначеності можна уникнути в такий спосіб. Наприклад, якщо число 827 000 має три значущі цифри, то його варто записати у вигляді $8{,}27\ 10^{6}$, якщо ж воно має чотири значущі цифри, то у вигляді $8{,}270 \cdot 10^{5}$.

Перші n значущих цифр наближеного числа називаються *вірними*, якщо абсолютна похибка цього числа не більше половини одиниці розряду, що виражається n-ою значущою цифрою.

Наприклад, для точного числа $A = 14{,}298$ число a = 14,300 є наближеним числом з чотирма вірними знаками, тому що

$$\Delta = |A - a| = 0{,}002 < \frac{1}{2} \cdot 0{,}01 = 0{,}005$$

У загальному випадку $\Delta \leq \frac{1}{2} \cdot 10^{m-n+1}$, де m – порядок старшої цифри, n – число вірних значущих цифр. Звідси, зв'язок числа вірних знаків наближеного числа з відносною похибкою цього числа виражається формулою

$$\delta = \frac{\Delta}{|A|} \leq \frac{10^{m-n+1}}{2|a_m|10^{m}} \leq \frac{1}{|a_m|} 10^{1-n}, \qquad (2.14)$$

де δ – відносна похибка наближеного числа *a*, *n* – число вірних десяткових знаків числа *a*, a_m — перша значуща цифра числа *a*. У якості граничної відносної похибки можна брати величину

$$\delta_a = 0.5\frac{1}{a_m}10^{1-n}, \qquad (2.15)$$

Приклад 2.1. Знайти граничну відносну похибку, якщо точне число 14,298 заміняється наближеним 14,300. У даному випадку $a_m = 1$, $n = 4$. По формулі (2.15) маємо $\delta_a = 5\cdot10^{-4}$ і навпаки, знаючи граничну відносну похибку, можна визначати кількість вірних десяткових знаків.

Приклад 2.2. Визначити число вірних десяткових знаків числа $\sqrt{5}$ при відносній похибці δ=0,001.

Очевидно, що $a_m = 2$. Далі $0{,}25\cdot10^{1-n} \le 0{,}001$. Звідси $1-n \le -3+lg4$, $n \ge 4$.

2.3. Правила округлення чисел

Округленням даного числа *a* (точного чи наближеного) називається заміна його числом *b* з меншою кількістю значущих цифр. Ця заміна виконується таким чином, щоб похибка округлення *a* - *b* була мінімальною.

Для округлення числа до *n*-ї значущої цифри відкидають усієї його цифри праворуч, починаючи з *n* + 1-й, або, при необхідності збереження розрядів, заміняють їх нулями. Зазначене відкидання цифр здійснюється за наступними правилами округлення:

1) якщо (*n* + 1)-а цифра більша 5, то до *n*-ої цифри додається одиниця;

2) якщо (*n* + 1)-а цифра менша 5, то всі цифри, що залишилися, зберігаються без зміни;

3) якщо (*n* + 1)-я цифра дорівнює 5 і серед цифр, що відкидаються, є ненульові, то до *n*-ої цифри додається одиниця;

4) якщо (n + 1)-а цифра дорівнює 5, а всі інші цифри, що відкидаються, дорівнюють нулю, то n-а зберігається без зміни, якщо вона парна, і до n-ої додається одиниця, якщо вона непарна.

Приклад 2.3. Округлити число 7,18342950 до восьми, семи та шести значущих цифр.

Відповідні округлені числа рівні: 7,1834295 (за правилом 2), 7,183430 (за правилом 4), 7,18343 (за правилом 2).

2.4. Обчислення похибки функції від n аргументів

Нехай задана деяка функція $y = f(x_1, x_2,..., x_n)$, від n аргументів і нехай значення кожного з аргументів x_i визначені з деякими похибками $|\Delta x_i|$, i=1,2,..., n. Потрібно знайти похибку даної функції.

Для вирішення цієї задачі будемо припускати, що функція $y = f(x_1, x_2,..., x_n)$ є диференційованою у деякій області D. Абсолютна похибка $|\Delta y|$ функції в при заданих абсолютних похибках $|\Delta x_1|$, $|\Delta x_2|,...,|\Delta x_n|$ аргументів дорівнює

$$|\Delta y| = | f(x_t + \Delta x_1, x_2 + \Delta x_2, \ldots, x_n + \Delta x_n) - f(x_1, x_2,..., x_n)|. \qquad (2.16)$$

Припускаючи, що величини $|\Delta x_i|$, i = 1, 2,..., n досить малі, можна записати наближені рівності $|\Delta y| \approx |dy|$

$$|\Delta y| \approx \left| \sum_{i=1}^{n} \frac{df}{dx_i} \Delta x_o \right| \le \sum_{i=1}^{n} \left| \frac{df}{dx_i} \right| \cdot |\Delta x_o| \qquad (2.17)$$

Отже, гранична абсолютна похибка Δy функції y дорівнює

$$|\Delta y| = \sum_{i=1}^{n} \left| \frac{df}{dx_i} \right| \cdot |\Delta x_o| \qquad (2.18)$$

де Δx_i - гранична абсолютна похибка аргументу x_i.

Оцінка для відносної похибки функції виходить шляхом ділення обох частин нерівності (2.16) на |y|:

$$\delta \le \sum_{i=1}^{n} \frac{1}{|y|} \left| \frac{df}{dx_i} \right| \cdot |\Delta x_o| = \sum_{i=1}^{n} \left| \frac{d}{dx_i} \ln y \right| \cdot |\Delta x_o| \qquad (2.19)$$

З формули (2.19) одержуємо вираз для граничної відносної похибки функції *y:*

$$\delta_y = \sum_{i=1}^{n} \left| \frac{d}{dx_i} \ln y \right| \cdot |\Delta x_o| \qquad (2.20)$$

Розглянемо окремі приклади на обчислення похибок різних функціональних співвідношень. Будемо припускати, що в кожнім прикладі задані ті чи інші види похибок аргументів.

1. Нехай $y=x_1+x_2+...+x_n$. За формулою (2.18) гранична абсолютна похибка суми *n* доданків дорівнює:

$\Delta y=\Delta x_1+ \Delta x_2+...+\Delta x_n$

Відносна похибка суми невід'ємних доданків не перевищує найбільшої з відносних похибок цих доданків.

2. Нехай $y = x_1 — x_2$. Тоді гранична абсолютна похибка різниці двох чисел дорівнює $\Delta y=\Delta x_1+ \Delta x_2$.

3. Нехай $y= x_1 x_2 ... x_n$, причому x_i (i =1, 2, ..., n) – додатні. Відповідно до формули (2.20) проведемо перетворення з метою одержання виразу для граничної відносної похибки добутку *n* співмножників:

$$\ln y = \sum_{i=1}^{n} \ln x_o$$

$\delta_y= \delta x_1+ \delta x_2+...+\delta x_n$.

4. Нехай $y = \frac{x_1}{x_2}$. За формулоюі (2.20) гранична відносна похибка частки дорівнює

$$\delta_y = \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2} = \delta x_1 + \delta x_2$$

5. Нехай $y = x^n$. Тоді $\ln y = n \ln x$, і $\delta_y = n \delta_x$.

6. Нехай $y = \sqrt[n]{x}$. Тоді, $\ln y = \frac{\ln x}{n}$ і $\delta_y = \frac{\delta_x}{n}$.

2.5. Завдання для самостійної роботи

1. Округлити сумнівні цифри чисел, залишивши вірні значущі цифри;

2. Знайти граничні абсолютні та відносні похибки результату вказаних операцій від двох аргументів $y=y(a_1,a_2)$ чи трьох аргументів $y=y(a_1,a_2,a_3)$.

1) $a_1 = 0{,}235 \pm 0{,}002$; $a_2 = 2{,}751 \pm 0{,}025$; $y = a_1 + a_2$; $y = a_2/a_1$.

2) $a_1 = 11{,}44 \pm 0{,}01$; $a_2 = 2{,}036 \pm 0{,}015$; $y = a_1 - a_2$; $y = a_1 \cdot a_2$.

3) $a_1 = л \pm 0{,}001$; $a_2 = \cos 25° \pm 0{,}001$; $y = a_1 - a_2$; $y = a_1 \cdot a_2$.

4) $a_1 = 2{,}8 \pm 0{,}3$; $a_2 = 25{,}8 \pm 0{,}05$; $y = a_1 + a_2$; $y = a_2/a_1$.

5) $a_1 = 2{,}56 \pm 0{,}005$; $a_2 = 1{,}2+0{,}05$; $y = a_1 - a_2$; $y = a_1 \cdot a_2$.

6) $a_1 = 3{,}85 \pm 0{,}01$; $a_2 = 2{,}043 \pm 0{,}0004$; $a_3 = 962{,}6 \pm 01$; $y = a_1 \cdot a_2/a_3$.

7) $a_1 = 7{,}27 \pm 0{,}01$; $a_2 = 5{,}205 \pm 0{,}002$; a3 = 87,32 ± 0,03; $y = a_1/(a_2+a_3)$.

3. Кут, зміряний теодолітом, виявився рівним 22°20'30"±30". Яка відносна похибка вимірювання?

4. Визначити число вірних знаків та дати відповідний запис наближеної величини прискорення сили тяжіння g = 9,806 при відносній похибці 0,5%.

5. Відомо, що гранична відносна похибка числа 19 рівна 0,1%. Скільки вірних знаків міститься в цьому числі?

6. Скільки вірних знаків містить число A= 3,7563, якщо відносна похибка рівна 1%?

7. Площа квадрата рівна 25,16 см^2 (з точністю до 0,01 см^2). З якою відносною похибкою та з скількома вірними знаками можна визначити сторону квадрата?

8. З скількома вірними знаками можна визначити радіус круга, якщо відомо, що його площа рівна 124,35 см^2 (з точністю до 0,01 см^2)?

9. Знайти граничну відносну похибка при обчисленні повної поверхні зрізаного конуса, якщо радіуси його підстав R = 23,64 ±0,01 (см), r = 17,31 ±0,01 (см), створююча l= 10,21 ±0,01 (см); число p = 3,14.

10. Число $g=9,8066$ є наближеним значенням прискорення сили тяжіння (для широти 45°) з п'ятьма вірними знаками. Знайти його відносну похибка.

11. Обчислити площу прямокутника, сторони якого 92,73±0,01 (м) і 94,5 ± 0,01 (м). Визначити відносну похибка результату і число вірних знаків.

2.6. Контрольні запитання

1. Перерахуйте види похибок.

2. Назвіть причини виникнення похибок.

3. Чи може похибка бути негативним числом?

4. Що таке неусувна похибка?

5. У чому суть машинної похибки?

6. Розкрийте поняття абсолютної похибки.

7. Що таке відносна похибка. Як вона вимірюється?

8. Дайте визначення поняття «значущі цифри».

9. Перерахуйте правила округлення чисел.

10. Як обчислити похибку від ***n*** аргументів?

Розділ 3 . Елементи векторної і матричної алгебри

3.1. Вектори

Вектори виникають з розгляду *n*-вимірних просторів і є направленими відрізками (які мають довжину, напрям та положення) в таких просторах. Вони розглядаються як вектори-стовпці, якщо не обумовлене інше:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

Операція *транспонування,* яка міняє стовпці на рядки і навпаки, позначається верхнім індексом Т. Вектори зазвичай виражаються через базис; для цього вибирається стандартна сукупність векторів $\mathbf{b}_1$ $\mathbf{b}_2$,...,$\mathbf{b}_n$, і всі інші вектори виражаються через цей базис:

$$\boldsymbol{y}=y_1\boldsymbol{b}_1+ y_2\boldsymbol{b}_2+\ldots+ y_n\boldsymbol{b}_n$$

Коефіцієнти y_i в цьому виразі називаються компонентами вектора у, а сам вираз записується в компактній формі

$$\mathbf{y}=(y_1, y_2,\ldots y_n)^{\mathrm{T}}$$

Базисні вектори визначають систему координат, і компоненти y_i є координатами точки кінця вектора в цій системі координат. Стандартні арифметичні операції (для векторів *x, y, z* і скаляра *a*):

- додавання: $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x} = (x_1 + y_1,\ x_2+y_2,\ \ \ldots.\ x_n + y_n)^{\mathrm{T}}$,

$$\mathbf{x} - \mathbf{y} = -(\mathbf{y} - \mathbf{x}); \quad (\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z});$$

- множення на скаляр: $a\mathbf{x} = (ax_1, ax_2, \ldots, ax_{\mathrm{n}})$.

Сукупність векторів $\mathbf{x}_1, \mathbf{x}_2,\ldots,\mathbf{x}_m$ – *лінійно незалежна*, якщо жодна їх лінійна комбінація не дорівнює нулю, за винятком нульової комбінації.

Простір називається *натягнутим* на сукупність векторів $\mathbf{x}_1$, $\mathbf{x}_2$,...,$\mathbf{x}_m$, якщо кожен вектор простору може бути виражений через лінійну комбінацію цих векторів. Розмірність векторного простору є мінімальне число з цих векторів, потрібних для отримання простору; кожен базис *N*-мірного простору повинен мати в своєму складі *N* векторів. *N*-мірні векторні простори скорочено часто називаються *N*-пространствами, і *N*-вектори є векторами в *N*-просторі. *Скалярний*, або внутрішній, добуток двох векторів **x** і **y** має вигляд:

$$\mathbf{x}^{\mathrm{T}}\mathbf{y} = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{m} x_i y_i$$

де x_i y_i — компоненти векторів **x** та **y**. Два вектори називаються *ортогональними (перпендикулярними)*, якщо $\mathbf{x}^{\mathrm{T}}\mathbf{y}=0$. Розмір вектора може бути виміряний нормою $\|\mathbf{x}\|$, так, що:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{m} x_i^2}$$

У випадку трьох вимірів це – звичайна евклідова довжина. Подвійні вертикальні риски позначають знак норми. Часто бувають зручні дві інші норми:

$$\|\mathbf{x}\|_\infty = \max|x_i|,$$

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{m} |x_i|$$

Кут α між двома векторами визначається співвідношенням:

$$\cos\alpha = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

Формат запису векторів повинен бути узгоджений з форматом запису матриць, тому вектори зазвичай записуються як вектори-стовпці. Це ускладнює виклад тексту, тому вектори записуються горизонтально із

знаком транспонування *T*, якщо вказівка формату стовпця не є необхідною.

3.2. Матриці

Абстрактні об'єкти, які приводять до матриць, є лінійними відображеннями, перетвореннями, або функціями, визначеними над векторами. Оскільки для векторів були введені координати, то ці лінійні функції можуть бути конкретно представлені за допомогою двовимірної таблиці чисел, тобто деякою матрицею $A=(a_{ij})$.

Якщо **y** є лінійна функція від **x**, то кожна компоненту $y_к$ вектора **y** є лінійна функція компонент x_j вектора **x**. Таким чином, для кожного до ми маємо:

$$y_k=a_{k1}\mathbf{x}_1+ a_{k2}\mathbf{x}_2+\ldots+ a_{kn}\mathbf{x}_n$$

Коефіцієнти збираються в матрицю *A*, яка і представляє лінійну функцію (відображення, перетворення, або співвідношення) між векторами **x** і **y**. Лінійна функція позначається $A\boldsymbol{x}$.

Дії над матрицями визначаються правилами, що виконуються для лінійних відображень. Так, $A+B$ є представленням суми двох лінійних функцій, представлених у свою чергу матрицями *A* та *B*. Маємо: $A+B=C$, де $c_{ij}=a_{ij} + b_{ij}$. Добуток AB є ефектом застосування відображення *B* та подальшого застосування відображення *A*. Покажемо, що AB=C, де $c_{ij}= \sum_k a_{ik}b_{kj}$. Для цього позначимо $\boldsymbol{y}=B\boldsymbol{x}$ та $\boldsymbol{z}=A\boldsymbol{y}$. Отже, нам необхідно визначити матрицю таку, що $\boldsymbol{z}=C\boldsymbol{x}$. Виразимо це співвідношення на мові компонент:

$$y_{k=}\sum_{j=1}^{n} b_{kj}x_j\ , \qquad z_{i=}\sum_{k=1}^{n} a_{ik}y_k$$

Таким чином:

$$z_i = \sum_{k=1}^{n} a_{ik} \left(\sum_{j=1}^{n} b_{kj} x_j \right) = \sum_{k=1}^{n} a_{ik} (b_{k1} x_1 + b_{k2} x_2 + \ldots + b_{kn} x_n) =$$

$$= a_{i1}(b_{11} x_1 + b_{12} x_2 + \ldots + b_{1n} x_n) + a_{i2}(b_{21} x_1 + b_{22} x_2 + \ldots + b_{2n} x_n) + , , , + a_{in}(b_{n1} x_1 + b_{n2} x_2 + \ldots + b_{nn} x_n)$$

Перегрупуємо тепер ці n^2 одночленів, розкривши дужки та зібравши коефіцієнти при кожному x_1, x_2,...,x_n, окремо. В результаті отримаємо:

$$a_{i1}(b_{11} x_1 + b_{12} x_2 + \ldots + b_{1n} x_n) + a_{i2}(b_{21} x_1 + b_{22} x_2 + \ldots + b_{2n} x_n) + \ldots + a_{in}(b_{n1} x_1 + b_{n2} x_2 + \ldots + b_{nn} x_n) =$$

$$= x_1(a_{i1} b_{11} + a_{i2} b_{21} + \ldots + a_{in} b_{n1}) + x_2(a_{i1} b_{12} + a_{i2} b_{22} + \ldots + a_{in} b_{n2}) + \ldots + x_n(a_{i1} b_{1n} + a_{i2} b_{2n} + \ldots + a_{in} b_{nn}) =$$

$$= \sum_{j=1}^{n} \left(\sum_{k=1}^{n} a_{ik} b_{kj} \right) x_j$$

і тому c_{ij} визначається вказаною вище формулою. Елемент, що стоїть на перетині *i*-го рядка та *j*-го стовпця матриці *C*, є скалярний добуток *i*-го рядка матриці *A* та *j*-го стовпця матриці *B*.

Справедливі такі арифметичні правила:

- $A+B=B+A$
- $AB \neq BA$ (за винятком спеціальних випадків).

Додавання та множення матриць на мові *Python* виглядає наступним чином (рис. 3.1):

```
>>> import numpy as np
>>> a=np.reshape(np.matrix(range(100)),(10,-1))
>>> a
matrix([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
        [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
```

```
        [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
        [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
        [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
>>> b=np.reshape(np.matrix(range(20,120)),(10,-1))
>>> b
matrix([[ 20,  21,  22,  23,  24,  25,  26,  27,  28,  29],
        [ 30,  31,  32,  33,  34,  35,  36,  37,  38,  39],
        [ 40,  41,  42,  43,  44,  45,  46,  47,  48,  49],
        [ 50,  51,  52,  53,  54,  55,  56,  57,  58,  59],
        [ 60,  61,  62,  63,  64,  65,  66,  67,  68,  69],
        [ 70,  71,  72,  73,  74,  75,  76,  77,  78,  79],
        [ 80,  81,  82,  83,  84,  85,  86,  87,  88,  89],
        [ 90,  91,  92,  93,  94,  95,  96,  97,  98,  99],
        [100, 101, 102, 103, 104, 105, 106, 107, 108, 109],
        [110, 111, 112, 113, 114, 115, 116, 117, 118, 119]])
>>> print a*b
[[ 3750  3795  3840  3885  3930  3975  4020  4065  4110  4155]
 [10250 10395 10540 10685 10830 10975 11120 11265 11410 11555]
 [16750 16995 17240 17485 17730 17975 18220 18465 18710 18955]
 [23250 23595 23940 24285 24630 24975 25320 25665 26010 26355]
 [29750 30195 30640 31085 31530 31975 32420 32865 33310 33755]
 [36250 36795 37340 37885 38430 38975 39520 40065 40610 41155]
 [42750 43395 44040 44685 45330 45975 46620 47265 47910 48555]
 [49250 49995 50740 51485 52230 52975 53720 54465 55210 55955]
 [55750 56595 57440 58285 59130 59975 60820 61665 62510 63355]
```

```
 [62250 63195 64140 65085 66030 66975 67920 68865 69810 70755]]
>>> print a+b
[[ 20  22  24  26  28  30  32  34  36  38]
 [ 40  42  44  46  48  50  52  54  56  58]
 [ 60  62  64  66  68  70  72  74  76  78]
 [ 80  82  84  86  88  90  92  94  96  98]
 [100 102 104 106 108 110 112 114 116 118]
 [120 122 124 126 128 130 132 134 136 138]
 [140 142 144 146 148 150 152 154 156 158]
 [160 162 164 166 168 170 172 174 176 178]
 [180 182 184 186 188 190 192 194 196 198]
 [200 202 204 206 208 210 212 214 216 218]]
>>> print b/a
[[ 0 21 11  7  6  5  4  3  3  3]
 [ 3  2  2  2  2  2  2  2  2  2]
 [ 2  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1  1  1]]
```

Рис. 3.1. Множення та додавання матриць на мові *Python*

Транспонована матриця A^T одержується віддзеркаленням матриці A щодо її діагоналі (елементів a_{ii}), тобто $a_{ij}=aji$. Одинична E матриця має одиниці на діагоналі, а інші її елементи рівні нулю.

Транспоновану матрицю за допомогою *Python* можна отримати на ступним чином (рис. 3.2):

```
>>> a
matrix([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
        [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
        [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
        [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
>>> b
matrix([[ 20,  21,  22,  23,  24,  25,  26,  27,  28,  29],
        [ 30,  31,  32,  33,  34,  35,  36,  37,  38,  39],
        [ 40,  41,  42,  43,  44,  45,  46,  47,  48,  49],
        [ 50,  51,  52,  53,  54,  55,  56,  57,  58,  59],
        [ 60,  61,  62,  63,  64,  65,  66,  67,  68,  69],
        [ 70,  71,  72,  73,  74,  75,  76,  77,  78,  79],
        [ 80,  81,  82,  83,  84,  85,  86,  87,  88,  89],
        [ 90,  91,  92,  93,  94,  95,  96,  97,  98,  99],
        [100, 101, 102, 103, 104, 105, 106, 107, 108, 109],
        [110, 111, 112, 113, 114, 115, 116, 117, 118, 119]])
>>> np.transpose(a)
matrix([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
```

```
       [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
       [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
       [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
       [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
       [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
       [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
       [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
       [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]])
>>> np.transpose(b)
matrix([[ 20,  30,  40,  50,  60,  70,  80,  90, 100, 110],
       [ 21,  31,  41,  51,  61,  71,  81,  91, 101, 111],
       [ 22,  32,  42,  52,  62,  72,  82,  92, 102, 112],
       [ 23,  33,  43,  53,  63,  73,  83,  93, 103, 113],
       [ 24,  34,  44,  54,  64,  74,  84,  94, 104, 114],
       [ 25,  35,  45,  55,  65,  75,  85,  95, 105, 115],
       [ 26,  36,  46,  56,  66,  76,  86,  96, 106, 116],
       [ 27,  37,  47,  57,  67,  77,  87,  97, 107, 117],
       [ 28,  38,  48,  58,  68,  78,  88,  98, 108, 118],
       [ 29,  39,  49,  59,  69,  79,  89,  99, 109, 119]])
>>>
```

Рис. 3.2. Транспонування матриць на мові *Python*

Одинична матриця обов'язково квадратна, тобто має однакове число рядків та стовпців. Легко бачити, що $EA=AE=A$. Обернена матрица A^{-1} є така матриця, для якої виконується рівність $A^{-1}A=E$. Не всяка матриця має обернену. Дійсно, добуток AB може бути рівний нулю, навіть якщо A або B не були нульовими матрицями. Наприклад :

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}=\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Якщо матриця A має обернену, то про неї говорять, що вона *невироджена*. Наступні вислови еквівалентні:

— матриця A - невироджена;

— зворотна матриця A^{-1} існує;

— стовпці матриці A лінійно незалежні;

— рядки матриці A лінійно незалежні;

— рівність $A\boldsymbol{x}=0$ означає, що $\boldsymbol{x}=0$.

Одиничну та нульову матриці можна отримати за допомогою мови *Python* так (рис. 3.3):

```
>>> c=np.zeros((10,10))
>>> c
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> d=np.fromfunction(lambda x,y: (x==y)*1,(100,100),)
>>> d
array([[1, 0, 0, ..., 0, 0, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       ...,
```

```
[0, 0, 0, ..., 1, 0, 0],
[0, 0, 0, ..., 0, 1, 0],
[0, 0, 0, ..., 0, 0, 1]])
```

Рис. 3.3. Створення нульових та одиничних матриць на мові *Python*

Розв'язання лінійних рівнянь полягає в знаходженні такого вектора $\boldsymbol{x}$ по даній матриці A і даному вектору $\boldsymbol{b}$, для якого виконується рівність $A\boldsymbol{x}= \boldsymbol{b}$.

Якщо матриця невироджена (тим самим A – квадратна), то це завдання завжди має єдиний розв'язок для будь-якого $\boldsymbol{b}$. Якщо A має більше рядків, ніж стовпців (тобто існує більше рівнянь, чим змінних), то зазвичай це задоча не має розв'язку. Якщо A має більше стовпців, чим рядків, то, як правило, існує нескінченно багато розв'язків. Найдавнішим і стандартним методом вирішення цієї задачі є метод виключення Гауса (залишимо осторонь правило Крамера із-за його крайньої практичної неефективності).

Система рівнянь називається однорідною, якщо права частина рівна нулю, наприклад, $A\boldsymbol{x}=0$.

Матриця U ортогональна, якщо її стовпці $\boldsymbol{u}_1, \boldsymbol{u}_2,..., \boldsymbol{u}_n$ ортогональні як вектори і їх довжина рівна 1. Іншими словами, $\boldsymbol{u}_i^T\boldsymbol{u}_j = 0$, якщо $i\neq j$, і $\boldsymbol{u}_i^T\boldsymbol{u}_i=1$.

Системи $U\boldsymbol{x}=\boldsymbol{b}$ з ортогональними матрицями легко розв'язується, оскільки U^TU – одинична матриця (рядки U^T є стовпцями матриці U). Якщо $U\boldsymbol{x}=\boldsymbol{b}$, то $U^TU\boldsymbol{x}=\boldsymbol{x}=U^T\boldsymbol{b}$ і $\boldsymbol{x}$ обчислюється безпосередньо. Таким чином, якщо відомий метод перетворення матриці A до ортогональної матриці, то цей метод може бути застосований для вирішення системи $A\boldsymbol{x}=\boldsymbol{b}$.

Матрицею перестановок називається така матриця, у якої всі елементи або 0, або 1, а в кожному рядку і в кожному стовпці є тільки одна 1. Наприклад

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Множення матриці зліва або справа на матрицю перестановок має результатом перестановку рядків або стовпців початкової матриці. Ця властивість дає їм свою назву; матриці перестановок застосовуються у формулах для вказівки перестановок рядків і стовпців і рідко використовуються в обчисленнях як таких.

Власні значення матриці A – це такі числа λ, для яких виконується рівність $A\boldsymbol{x} = \lambda\boldsymbol{x}$ для деякого ненульового вектора $\boldsymbol{x}$; вектор $\boldsymbol{x}$ також називається власним вектором матриці A. Ефект лінійного відображення власного вектора полягає в безпосередньому множенні цього вектора на константу λ, що є власним значенням. Квадратна матриця порядку n має n власних значень і, як правило, але не завжди, n власних векторів. *Спектральний радіус(A)* матриці A – це найбільше по абсолютній величині власне значення A. Спектральний радіус грає фундаментальну роль в дослідженні збіжності ітераційних процесів, що включають матриці.

Існують різні способи вимірювання величини, або норми, матриці. Ми використовуватимемо тільки одну норму, а саме:

$$\|A\| = \max_{\|x\|=1} \|Ax\| .$$

Норма – це найбільша довжина вектора, що належить одиничній сфері, яку він має після застосування лінійного перетворення, визначуваного матрицею A. Така норма визначається в термінах векторної норми, і, отже, різні векторні норми дають різні норми для матриці A. Норма може бути виражена явно через три раніше введені векторні норми:

$$\|A\|_1 = \max_{\|x\|_1=1} \|Ax\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|$$

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2 = \text{(найбільше власне значення } A^T A^{½}\text{)},$$

$$\|A\|_\infty = \max_{\|x\|_\infty=1} \|Ax\|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|$$

1-норма і ∞-норма широко застосовуються, оскільки вони легко обчислюються. Легко показати, що $\rho(A) \le //A//$ для будь-якої норми, тому *1*-норма і ∞-норма забезпечують просту оцінку спектрального радіусу матриці.

Для будь-якої норми матриці A справедливі наступні властивості:

а) $//A//>0$, причому $//A//=0$, коли A – нульова матриця;

б) $a//A// = |a|\cdot//A//$, де a – дійсне число;

в) $//A+B//\le//A//+//B//$ – нерівність трикутник;

г) $//A\cdot B//\le//A//\cdot//B//$ – нерівність Коші-Буняковського.

Між нормою матриці і її власними числами існує певний зв'язок, який виражається наступною теоремою.

Теорема 3.1. Модуль кожного власного числа матриці A не перевищує її норми.

Нехай λ – власне число матриці A. Тогда для будь-якого $\boldsymbol{x}\neq\boldsymbol{0}$ маємо $A\boldsymbol{x}=\lambda\boldsymbol{x}$. Далі, однакові вектори мають рівні норми: $//A\boldsymbol{x}//=//\lambda\boldsymbol{x}//$. За властивістю норми маємо: $//\lambda\boldsymbol{x}//=|\lambda|//\boldsymbol{x}//\le//A//\cdot//\boldsymbol{x}//$. Звідси $|\lambda|\le//A//$.

Інтерес до матриць виникає з їх зв'язку з лінійними функціями декілька змінних (такі функції є природним способом опису простих математичних моделей об'єктів, залежних від декількох змінних). Лінійні функції з багатьма змінними «існують» як абстрактні функції точно так, як і вектори «існують» як абстрактні об'єкти. Для векторів були визначені

поняття, з якими можна маніпулювати і робити обчислення за допомогою введення систем координат; те ж саме має місце і для функцій з багатьма змінними. Якщо виразити $\boldsymbol{x}$ та $\boldsymbol{y}$ в позначеннях деякої системи координат, то повинна існувати формула для обчислення координат $(y_1, y_2,\ldots y_n)^T$ через координати $\boldsymbol{x} = (x_1, x_2,\ldots, x_n)^T$. Ця формула включає двовимірну таблицю чисел, скажемо $A=(a_{ij})$, і насправді має вигляд:

$$\boldsymbol{y} = A\boldsymbol{x}.$$

Таким чином, матриця є конкретне представлення лінійної функції з багатьма змінними, отримане від введення системи координат для векторів. Різні вибори систем координат дають різні матричні уявлення для однієї і тієї ж функції.

Якщо $B=C^{-1}AC$, то A і B називаються *подібними* матрицями; ця формула виражає ефект зміни систем координат (або іншого вибору базисних векторів). Тому всі властивості матриці A, які є властивостями лінійної функції F, що лежить в її основі, не міняються після застосування подібного перетворення $B=C^{-1}AC$. Ці властивості стосуються і власних значень і власних векторів.

Як приклад корисності власних значень та власних векторів розглянемо, що відбудеться, якщо як базисні вектори вибрані власні вектори (припустимо, що їх досить для цієї мети). В цьому випадку матричне представлення лінійної функції є просто діагональна матриця. Далі, компоненти вектора $A\boldsymbol{x}$ є компонентами вектора $\boldsymbol{x}$, помноженими на відповідні власні значення. Тому аналіз матриць та матричне числення істотно спрощуються, якщо отримати власні вектори матриці A і використовувати їх як базисні вектори.

3.2.1. Обчислення визначника

Задача обчислення визначника матриці зустрічається досить рідко, тому розглянемо її коротко. Нагадаємо, що визначником матриці $\boldsymbol{A}$ називають величину

$$D = \sum (-1)^k a_{\alpha 1,1} \cdot a_{\alpha 2,2} \cdot \ldots \cdot a_{\alpha n,n} , \qquad (3.1)$$

де α_i замінюються довільною перестановкою чисел ***1,2,..., n***.

Оскільки кількість перестановок з ***n*** чисел дорівнює ***n!,*** то кількість доданків в (3.1) дорівнює ***n!***. Знак кожного доданку визначається ***k***-числом "безладь" у послідовності $a_1, a_2, \ldots, a_n$. "Безлад" – це таке положення індексів, коли старший стоїть раніше молодшого. Наприклад, у добутку $a_{21}a_{12}a_{33}$ перші індекси утворюють послідовність 2,1,3 і число "2" розташовано раніше "1". Число "безладу" тут дорівнює одиниці і весь добуток в визначнику *3-го* порядку беруть з знаком "мінус". У добутку $a_{31}a_{12}a_{23}$ перші індекси утворюють послідовність 3, 1, 2. Оскільки число "3" розташовано раніше "1" і "2", то число "безладь" дорівнює 2 і добуток підсумовується з знаком "плюс".

Для обчислення визначника порядку ***n*** за формулою (3.1) треба додати ***n!*** додатків, тобто виконати ***(n!-1)*** операцій додавання. Для обчислення кожного доданку треба виконати (***n-1***) операцій множення. Тобто загальна кількість операцій для обчислення визначника за формулою (3.1) дорівнює

$$N = \text{n!}\cdot(-1) + \text{n!} - 1 = \text{n} \cdot \text{n!} - 1 \approx \text{n} \cdot \text{n!}$$

Із зростанням порядку ***n*** матриці кількість потрібних операцій зростає дуже швидко. Наприклад, при ***n=3*** кількість операцій ***N=17,*** при ***n=10*** $N=3.6*10^7$ **,** при ***n=20*** **–** $N=5*10^{19}$. ЕОМ середньою швидкодією 1 млн операцій за секунду обчислить за формулою (3.1) визначник *10-го* порядку за *36с*, а *20-го* порядку – за $5*10^8$ діб [1]. Зрозуміло, що в разі необхідності обчислити визначник, користуватися формулою (3.1) недоцільно, тому застосовують інші методи. Нагадаємо деякі означення.

Мінором елемента a_{ij} називають визначник (***n-1***)-го порядку, утворений з даного визначника виключенням ***i****-го* рядка та ***j****-го* стовпця:

$$M_{ij} = \begin{pmatrix} a_{1,1}\ldots & a_{1,j-1} & a_{1,j+1}\ldots & a_{1,n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{i-1,1}\ldots & a_{i-1,j-1} & a_{i-1,j+1}\ldots & a_{i-1,n} \\ a_{i+1,1}\ldots & a_{i+1,j-1} & a_{i+1,j+1}\ldots & a_{i+1,n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n,1}\ldots & a_{n,j-1} & a_{n,j-1}\ldots & a_{n,n} \end{pmatrix}$$

Алгебраїчним доповненням A_{ij} елемента a_{ij} називають його мінор, взятий зі знаком

$$A_{ij} = (-1)^{i+j} M_{ij}.$$

Нагадаємо також основні властивості визначника, які використовують при його обчисленні:

1. Визначник не зміниться, якщо замінити його рядки стовпцями:

$$|A| = |A'|,$$

де A' – транспонована матриця. На основі цієї властивості всі наступні твердження, які відносяться до рядків, справедливі і для стовпців матриці A.

2. Визначник дорівнює нулю, якщо два його рядка рівні або пропорційні, або якщо один з рядків є лінійна комбінація яких-небудь інших рядків.

3. Множник, загальний для всіх елементів якого-небудь рядка, можна винести за знак визначника.

4. Якщо визначники, які відрізняються тільки елементами *i-го* рядка, скласти, то одержимо визначник, у якого *i-й* рядок дорівнює сумі відповідних елементів *i-х* рядків доданків:

$$\left|a_{i1} \overset{C}{\underset{D}{\ldots}} a_{in}\right| + \left|b_{i1} \overset{C}{\underset{D}{\ldots}} b_{in}\right| = \left|a_{i1}+b_{i1} \overset{C}{\underset{D}{\ldots}} a_{in}+b_{in}\right|$$

5. Визначник не зміниться, якщо до якого-небудь рядка додати елементи іншого рядка або лінійну комбінацію інших рядків.

6. Визначник можна розкласти за елементами *i-го* рядка:

$$|A| = a_{i1}A_{i1} + a_{i2}A_{i2} + \ldots + a_{in}A_{in}$$

де A_{ij} – алгебраїчне доповнення.

7. Сума добутків усіх елементів *i-го* рядка на алгебраїчні доповнення другого рядка дорівнює нулю:

$$\sum_{j=1}^{n} a_{ij} A_{kj} = 0 \quad при\, i \neq k$$

8. При переставленні двох рядків визначника його знак змінюється на протилежний.

9. На основі означення (3.1) можна зробити висновок, що визначник діагональної або трикутної матриці дорівнює добутку діагональних елементів. Тому на практиці для обчислення визначника матриці спочатку використовують метод Гауса або його модифікацію, а потім обчислюють визначник одержаної трикутної або діагональної матриці. При цьому знак не змінюється, якщо кількість переставлень рядків при виборі головного елемента парна, а інакше змінюється на протилежний.

Для визначення детермінанта матриці на мові Python потрібно виконати наступне (рис. 3.4):

```
>>> c=np.zeros((10,10))
>>> c
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> d=np.fromfunction(lambda x,y: (x==y)*1,(100,100),)
```

```
>>> d
array([[1, 0, 0, ..., 0, 0, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 0, 1]])
```

Рис. 3.4. Визначення детермінанта матриці на мові *Python*

3.2.2. Обернення матриць

Обчислення елементів оберненої матриці називається *оберненням* матриці.

Зупинимося на деяких точних обчислювальних методах обернення матриць. На початку розглянемо питання про обернення матриць спеціального вигляду – трикутних матриць.

Визначення 3.1. Квадратна матриця називається верхньою (нижньою) трикутною, якщо елементи, що стоять вище (нижче) за головну діагональ, рівні нулю. Так, матриця

$$A_1 = \begin{Bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{mn} \end{Bmatrix}$$

де $a_{ij}=0$ при $i>j$, є верхньою трикутною матрицею. Аналогічно, матриця

$$A_2 = \begin{Bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{Bmatrix}$$

де a_{ij}=0 при $j > i$, є нижньою трикутною матрицею. В курсах лінійної алгебри доводиться, що:

1) сума і добуток двох верхніх (нижніх) трикутних матриць одного і того ж порядку є також верхня (нижня) трикутна матриця того ж порядку;

2) обернена матриця невиродженої верхньої (нижньої) трикутної матриці є також верхня (нижня) трикутна матриця.

Якщо $U=[a_{ij}]$ – верхня трикутна матриця, то для обчислення елементів її оберненої матриці $U^{-1}=[u_{ij}]$ послідовно (уздовж рядків зверху вниз і зліва направо) можна скористатися достатньо простими формулами:

1) u_{ij} =0, при $i>j$

2) $u_{ij} = \frac{1}{a_{ii}}$, при $i=j$ (3.2)

3) $u_{ij} = -\frac{1}{a_{jj}} \sum_{k=1}^{j-1} u_{ik} a_{rj}$, при $i<j$.

Аналогічно для нижньої матриці L=[a_{ij}] її обернена матриця L^{-1}=[l_{ij}] визначається:

1) l_{ij} =0, при $i<j$

2) $l_{ij} = \frac{1}{a_{ii}}$, при i=j (3.3)

3) $l_{ij} = -\frac{1}{a_{ii}} \sum_{k=1}^{i-1} l_{kj} a_{ik}$, при i>j.

Порядок обернення верхніх (нижніх) трикутних матриць полягає в послідовному визначенні за формулами (3.2) або (3.3) елементів обернених матриць спочатку уздовж *1*-го рядка, потім *2*-го і т.д. до *n*-го рядка.

Загальне число дій множень та ділень, потрібних для обернення трикутних матриць, визначається з виразу : $\frac{n^3 + 3n^2 + 2n}{6}$.

3.2.3. Матричні рівняння

В системі лінійних алгебраїчних рівнянь

Ax=B, (3.4)

матриця ***A*** найчастіше відображає модель фізичного об'єкта (побудову моста, схему електричного ланцюга), а вектор ***B*** – зовнішні відносно моделі об'єкти (навантаження на міст, джерела електричного сигналу). Тому для фіксованої моделі (матриці ***A***) може виникнути необхідність у розгляданні різних зовнішніх впливів, тобто необхідність у розв'язанні декількох систем рівнянь з однаковою матрицею і різними правими частинами:

$$Ax_1 = B_1,\ \ Ax_2 = B_2,\quad Ax_m = B_m. \tag{3.5}$$

Об'єднання цих систем і називають *матричним рівнянням*:

Ax=B (3.6)

Тут ***A*** – квадратна ***n*n*** матриця; ***x*** - матриця ***n*m***, яка складена з векторів невідомих $x_1, x_2, .., x_m$; ***B*** – матриця ***n*m***, яка складена з векторів правих частин $B_1, B_2, ..., B_m$:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \\ & & \dots & \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{pmatrix} = \begin{pmatrix} b_1^{(1)} & b_2^{(2)} & \dots & b_1^{(m)} \\ b_2^{(1)} & b_2^{(2)} & \dots & b_2^{(m)} \\ & & \dots & \\ b_n^{(1)} & b_n^{(2)} & \dots & b_n^{(m)} \end{pmatrix} \tag{3.7}$$

У цьому запису матричного рівняння (3.6) верхній індекс означає номер однієї з систем (3.5), які можуть виникати послідовно одна за іншою при розв'язуванні більш складної задачі. Чи можна у цьому випадку скоротити кількість потрібних операцій для їх розв'язування порівняно з $(2/3)n^3m$, використовуючи незмінність їх матриці? Очевидно, розклавши один раз матрицю ***A*** на добуток двох трикутних матриць і затративши при цьому $(4n^3 - 3n^2 - n)/6$ операцій, можна потім для кожної з ***m*** систем виконувати тільки прямий і зворотний ходи ***LU***–алгоритму (про цей алгоритм буде розповідатись в наступних розділах). Прямий хід потребує ***n(n-1)*** операцій, зворотний – ***n***, всього ***2n - n*** операцій. За такого підходу розв'яжемо системи (3.5) за $(4n^3 - 3n^2-n)/6+m(2n^2 -n)$ операцій, що становить близько $(2/3)n^3$

+2mn. Одержана кількість операцій приблизно в ***m*** разів менша, чим потрібно ***(2/3)n^{3}m*** при незалежному розв'язуванні цієї серії систем.

Системи (3.5) можуть бути також одночасно відомі. У цьому випадку можна об'єднати їх в одну матричну систему (3.7) і розв'язувати не тільки за допомогою ***LU*** - алгоритму, але і методом Гауса. В останньому випадку при відніманні рядків матриці ***A*** віднімаються і ті ж самі рядки матриці ***B***. Тим самим одночасно розв'язуються ***m*** систем. З точки зору кількості операцій метод Гауса і ***LU***-алгоритм еквівалентні.

3.3. Завдання для самоперевірки

1. Обчислити $///x//_1$, $//x//_2$ *та* $//x//_\infty$ для кожного з наступних векторів:

а) $(1, 2, 3, 4)^T$,

б) $(0, — 1, 0, — 2, 0, 1)^T$,

в) $(0, 1, – 2,3, – 4)^T$,

г) $(4.1, – 3.2,8.6, – 1.5, – 2.5)^T$.

2. Обчислити кутα між наступними парами векторів:

а) $(1, 1, 1, 1)^T$, $(—1, 1, – 1, 1)^T$,

б) $(1, 0, 2, 0)^T$, $(0, 1, – 1, 2)^T$,

в) $(1, 2, – 1, 3)^T$, $(2, 4, – 2, 6)^T$,

г) $(1.1, 2.3, – 4.7, 2.0)^T$, $(3.2, 1.2, — 2.3, – 4.7)^T$.

3. Знайти евклідову норму вектора (2, -1, 4) та скалярний добуток векторів $(2, —3, 4, 1)^T$ та $(4, —3, 2)^T$.

4. Визначити, чи належить вектор $(6, 1, —6, 2)^T$ простору, натягнутому на вектори $(1, 1, —1, 1)^T$, $(– 1, 0, 1, 1)^T$ та $(1, —1, —1, 0)^T$. Яка розмірність простору, натягнутого на ці три вектори?

5. Чи утворюють вектори $b_1=(1,0,0,-1)^T$, $b_2=(1,-1,0,0)^T$, $b_3=(1,0,-1,0)^T$, $b_4=(1,0,0,-1)^T$ базис для векторного просторів розмірності 4?

6. Довести, що три вектори $b_1=(1,2,3,4)^T$, $b_2=(2,1,0,4)^T$, $b_3=(0,1,1,4)^T$ лінійно незалежні. Чи утворюють вони базис тривимірного простору?

7. Припустимо, що $x_1, x_2,\ldots,x_m$ – лінійно незалежні, а $x_1, x_2,\ldots,x_m, x_{m+1}$ лінійно залежні. Показати, що x_{m+1} є лінійною комбінацією $x_1, x_2,\ldots,x_m$.

8. Нехай $x_1=(1,2,1), x_2=(1,2,3), x_3=(3,6,5)$. Показати, що ці вектори лінійно залежні, але на них може бути натягнутий двовимірний простір. Знайти базис цього простору.

9. Показати, що для будь-яких двох векторів x та y і будь-яка з трьох норм (1,2 або∞) має місце нерівність

$$|\|x\|-\|y\||\le|x-y|.$$

Ця нерівність виконується для всіх векторних норм.

10. Показати, що для двох ненульових векторів x і y рівність $\|x+y\|_2=\|x\|_2+\|y\|_2$ виконується тоді і тільки тоді, коли $y=ax$, де a – невід'ємна константа.

11. Показати, що для двох n-векторів x і y виконується нерівність $\|x+y\|\le\|x\|+\|y\|$ для будь-якої з трьох норм (1, 2 або ∞). *Вказівка*: для норм 1 і ∞ довести нерівність для двовимірних векторів і потім застосувати індукцію.

12. Показати, що матриця $A=\begin{pmatrix}1 & 2\\ 2 & 4\end{pmatrix}$ невироджена.

13. Знайти ненульове рішення системи:

$$x_1-2x_2+x_3=0,$$

$$x_1+x_2-2x_3=0.$$

14. Нехай

$A=\begin{pmatrix}1 & 2 & 3\\ 0 & -1 & 2\\ 2 & 0 & 2\end{pmatrix}$, $B=\begin{pmatrix}1 & 1 & 2\\ -1 & 1 & -1\\ 1 & 0 & 2\end{pmatrix}$, $C=\begin{pmatrix}1 & 0 & 1\\ 0 & 1 & 2\\ 2 & 0 & 1\end{pmatrix}$

а) Обчислити AB та BA і показати, що $AB \neq BA$.

б) Знайти $(A+B)+C$ та $A+(B+C)$.

в) Показати, що $(AB)C=A(BC)$.

г) Показати, що $(AB)^T=B^TA^T$.

15. Показати, що матриця A вироджена: $A=\begin{pmatrix}3 & 1 & 0\\ 2 & -1 & -1\\ 3 & 3 & 1\end{pmatrix}$

16. Для матриці A завдання 4 обчислити PA та AP, де P – матриця перестановок:

$P=\begin{pmatrix}0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}$.

17. Знайти матрицю перестановок P_1, яка для будь-якої квадратної матриці A четвертого порядку переставляє її другий та четвертий стовпці. Знайти P_2, яка переставляє перший та третій рядки матриці A.

18. Перевірити наступні системи рівнянь на спільність (тобто показати існування принаймні одного рішення):

а) $\begin{cases} x+y+2z+w=5 \\ 2x+3y-z-2w=2 \\ 4x+5y+3z=7 \end{cases}$ б) $\begin{cases} x+y+z+w=0 \\ x+3y+2z+4w=0 \\ 2x+z-w=0 \end{cases}$

19. Показати, що матриця $A=\begin{pmatrix}1 & 1\\ 0 & 2\end{pmatrix}$ невироджена, обчисливши обернену до неї матрицю.

20. Знайти число операцій додавання та множення, необхідних для множення $n \times n$-матриці на n-вектор та перемножування двох $n \times n$-матриць.

21. Обчислити $//A//_1$ $//A//_\infty$ для матриці $A=\begin{pmatrix} 1 & 0 & 2 & -1 \\ 6 & -4 & 3 & 0 \\ 4 & 0 & -4 & 2 \\ 1 & 5 & 1 & 6 \end{pmatrix}$

22. Побудувати ненульову *3x3*-матрицю *A* та ненульовий вектор *x*, такі, що *Ax=0*.

12. Нехай $A=E-2xx^T$, де *x* – вектор-стовпчик . Показати, що *A* – ортогональна матриця і $A_2=1$.

23. Для невиродженої матриці *A* показати, що $x^TA^TAx=0$ тоді і тільки тоді, коли $x^Tx=0$.

24. Нехай *x* – вектор-рядок. Показати, що $xx^T=xx=(||x||^2)$. Чому рівне x^Tx?

25. Нехай для будь-якого вектора *x* функція *F* визначена одним з наступних способів:

а) $F(x)=(x_3, x_1)^T$;

б) $F(x)=(x_1+x_2, 0, x_3)^T$;

в) $F(x)=(x_2, 0, x_1—x_2, x_3, 0)^T$.

Показати, що кожний з цих способів визначає *F* як лінійну функцію. Знайти для кожної з лінійних функцій матрицю, що її представляє.

26. Нехай вектор *x* має компоненти $(x_1\ x_2)^T$ в звичайній евклідової системі координат на площині (тобто базисними векторами є $(1,0)^T$ та $(0,1)^T$). Які компоненти має вектор *x* в базисі $(1,1)^T$, $(1,-1)^T$? Дати геометричну інтерпретацію співвідношення між цими системами координат.

27. Показати, що $//AB//\leq//A//\cdot//B//$ для матричної норми, визначеної в цьому розділі вище. Довести правильність даної формули для норми $||A||_\infty$ $||A||_1$.

28. Показати для *nxn*-матриці *A*, що

$$\max_{i,j}\left|a_{ij}\right| \le \left\|A\right\| \le n \cdot \max_{i,j}\left|a_{ij}\right| .$$

3.4. Контрольні запитання

1. Що таке вектор?

2. У чому суть операції транспонування?

3. Яка сукупність векторів називається лінійно незалежною?

4. Який простір вважається натягнутим на сукупність векторів?

5. Які вектори ортогональні?

6. Поняття матриці.

7. Перерахуйте властивості одиничної матриці.

8. Яка матриця є виродженою?

9. Що таке ортогональність матриці?

10. Розкрийте поняття матриці перестановок.

11. Власні значення матриці.

12. Що таке спектральний радіус матриці?

13. Які норми має матриця?

14. Які властивості мають норми матриці?

15. Як обчислити визначник матриці?

16. Перерахувати основні властивості визначника матриці.

17. Дати визначення мінору матриці.

18. Дати визначення алгебраїчного доповнення.

19. Що таке трикутна матриця? Які вони бувають?

20. Розкрийте поняття матричного рівняння.

Розділ 4. Чисельне розв'язування алгебричних та трансцендентних рівнянь

4.1. Вступ

Інженеру часто доводиться вирішувати рівняння алгебри та трансцендентні рівняння, що може бути самостійним завданням або бути складовою частиною складніших завдань. У обох випадках практична цінність чисельного методу значною мірою визначається швидкістю та ефективністю отримання розв'язку.

Вибір відповідного алгоритму для розв'язування рівнянь залежить від характеру даного завдання. Завдання, що зводяться до розв'язування рівнянь алгебри та трансцендентних рівнянь, можна класифікувати по числу рівнянь та залежно від передбачуваного характеру і числа розв'язків. На рис. 4.1 представлена схема класифікації рівнянь.

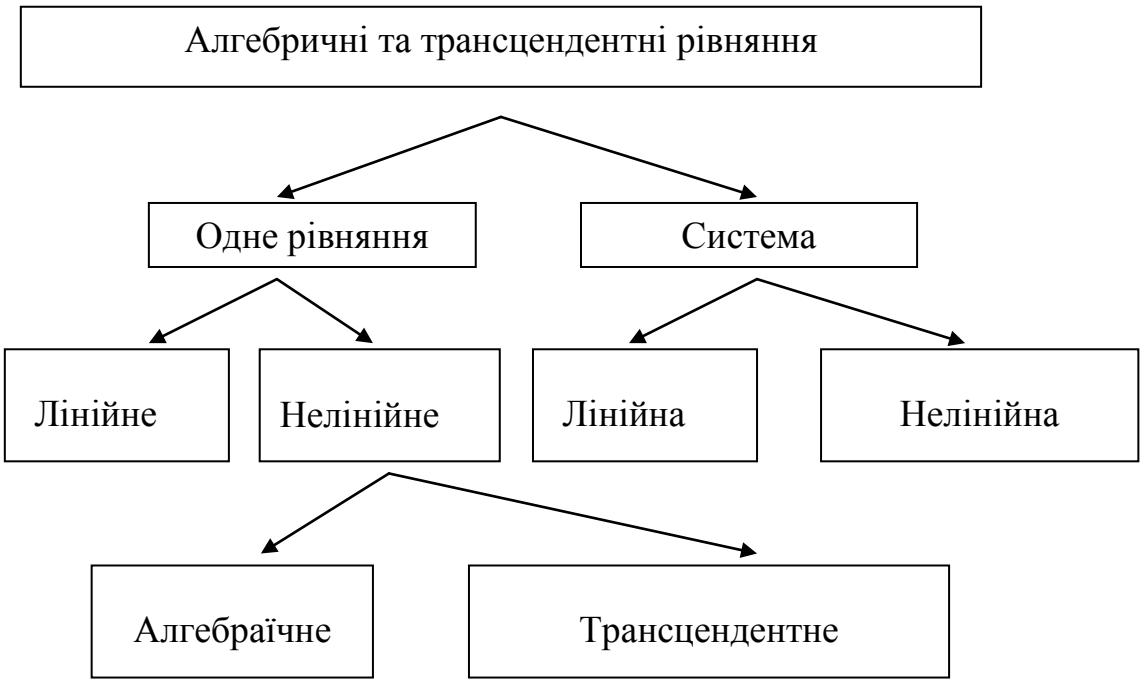


Рис. 4.1. Класифікація рівнянь

Рівняння називатимемо *лінійним, алгеброю* або *трансцендентним* залежно від того, чи має воно один розв'язок, *n* розв'язків або невизначене число розв'язків. Систему рівнянь називатимемо *лінійною* або *нелінійною* залежно від математичної природи вхідних в неї рівнянь. Розв'язування лінійного рівняння з

одним невідомим виходить досить просто і тут не розглядається. Мета даного розділу – виклад різних методів розв'язування рівнянь, що відносяться до решти чотирьох типів.

4.2. Корені нелінійного рівняння

Зазвичай нелінійні рівняння поділяють на трансцендентні та алгебричні. Хоча вони часто вирішуються одними і тими ж методами, в даному випадку методи їх рішення розглядаються окремо, оскільки розв'язування алгебричних рівнянь має особливі властивості. Цей розділ присвячений трансцендентним рівнянням; алгебричні рівняння розглядаються нижче.

Нелінійні рівняння, що містять тригонометричні функції або інші спеціальні функції, наприклад $lg(x)$ або e^x, називаються *трансцендентними.*

Розв'язування рівнянь проходить у два етапи: 1) знаходження наближених значень коренів (відділення коренів) і 2) уточнення наближених значень коренів.

Загальних методів відділення коренів не існує. Методи уточнення наближених значень коренів при розв'язуванні нелінійних рівнянь такого типу діляться на *прямі* та *ітераційні*. Перші дозволяють знайти рішення безпосередньо за допомогою формул і завжди забезпечують отримання точного розв'язку. Відомим прикладом такого роду є формула для коренів квадратного рівняння. В ітераційних методах задається процедура розв'язування у вигляді багатократного застосування деякого алгоритму. Отриманий розв'язок завжди є наближеним, хоча може бути скільки завгодно близьким до точного. Ітераційні методи найбільш зручні для реалізації на комп'ютері і тому детально розглядаються в цьому розділі. У кожному з цих методів вважається, що задача полягає у відшуканні дійсних коренів (нулів) рівняння $f(x)= 0$. Хоча подібні рівняння також можуть мати комплексні корені, способи їх відшукання зазвичай розглядаються тільки для алгебричних рівнянь.

4.2.1. Метод половинного ділення

Блок-схема алгоритму методу половинного ділення представлена на рис. 4.2.

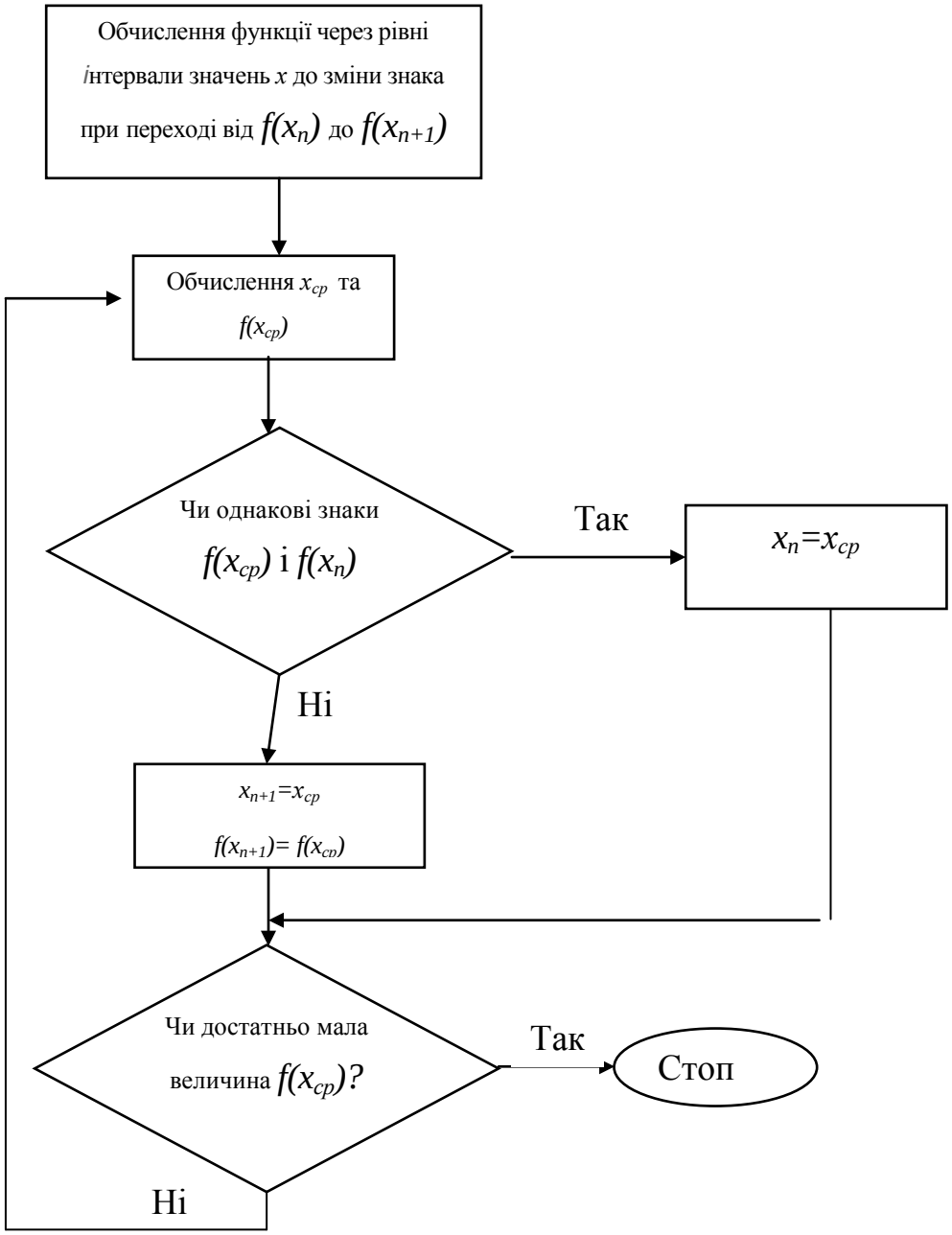


Рис. 4.2. Блок-схема алгоритму методу половинного ділення

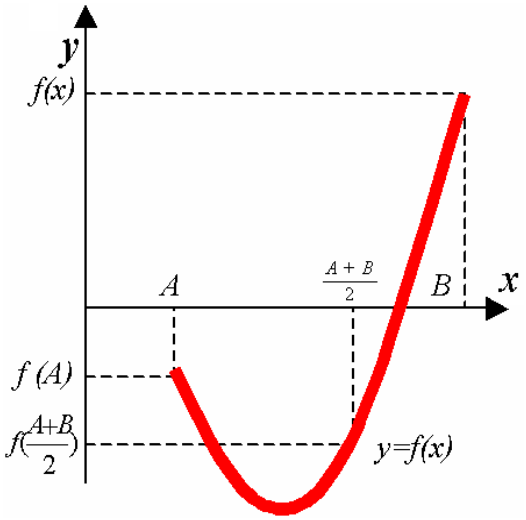


Рис. 4.3. Графічна інтерпретація алгоритму методу половинного ділення

Алгоритм складається з наступних операцій. Спочатку обчислюються значення функцій в точках, розташованих через рівні інтервали на осі *x*. Це робиться до тих пір, поки не будуть знайдено два послідовні значення функції $f(x_n)$ та $f\ \{x_{n+1})$, що мають протилежні знаки (нагадаємо, якщо функція неперервна, то зміна знака вказує на існування кореня).

Потім за формулою обчислюється середнє значення *x* в інтервалі значень $[x_n,\ x_{n+1}]$ та знаходиться значення функції $f(x_{cp})$. Якщо знак $f(x_{cp})$ співпадає із знаком $f(x_n)$, то надалі замість $f(x_n)$ використовується $f(x_{cp})$. Якщо ж $f(x_{cp})$ має знак, протилежний знаку $f(x_n)$, тобто її знак співпадає із знаком $f(x_{n+1})$, то на $f(x_{cp})$ замінюється це значення функції. В результаті інтервал, в якому поміщено значення кореня, звужується. Якщо $f(x_{cp})$ достатньо близько до нуля, процес закінчується, інакше він продовжується. На рис. 3.3. ця процедура показана графічно. Хоча метод половинного ділення не має високої обчислювальної ефективності, із збільшенням числа ітерацій він забезпечує набуття все більш точного наближеного значення кореня. Після того, як вперше знайдений інтервал, в якому знаходиться корінь, його ширина після *N* ітерацій спадає в 2^N разів.

Реалізація наведеного алгоритму на мові *Python* представлена на рис. 4.4.

```
def dihotomy(a,b,eps):
```

```
counter=0;
while (b-a>eps):
    c=(a+b)/2.0;
    if (f(b)*f(c)<0):
        a=c;
    else:
        b=c;
    print "%d %.4f %.4f %.4f %.4f %.4f %.4f" %(counter,a,b,f(a),f(b),c,f(c))
    counter+=1
```

Рис. 4.4. Реалізація алгоритму методу половинного ділення на мові Python

Функція dihotomy має три аргументи: *a*,*b* – інтервал пошуку рішення, *eps* – задана точність пошуку.

4.2.2. Метод хорд

У основі цього методу лежить лінійна інтерполяція по двох значеннях функції, що мають протилежні знаки. При відшуканні кореня цей метод нерідко забезпечує швидшу збіжність, ніж попередній. Блок-схема алгоритму методу хорд вельми схожа на попередню і тому тут не наводиться. Метод виконується таким чином. Спочатку визначаються значення функції в точках, розташованих на осі *x* через рівні інтервали. Це робиться до тих пір, поки не буде знайдена пара послідовних значень функції $f(x_n)$ та $f(x_{n+1})$, що мають протилежні знаки. На рис. 4.5 процес розв’язування показаний графічно.

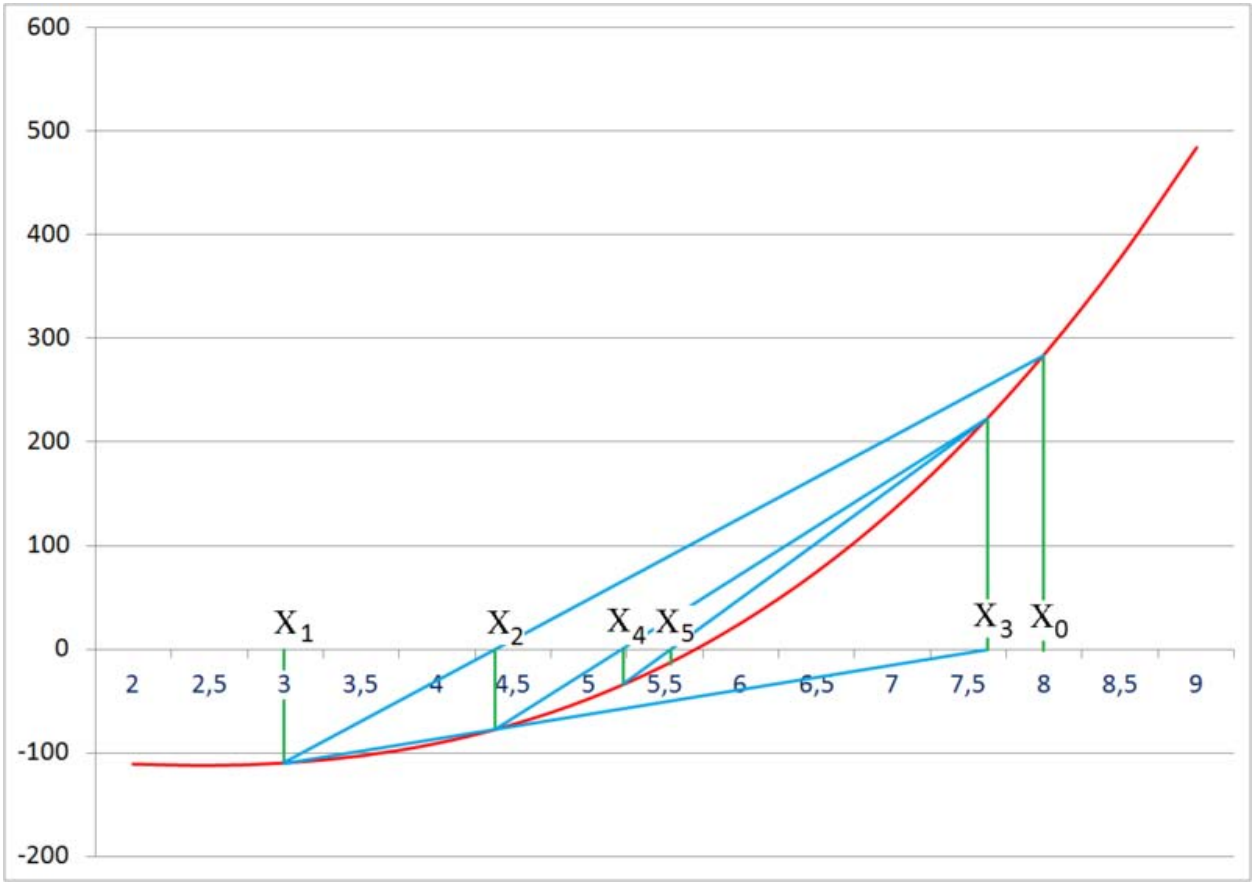


Рис. 4.5. Метод хорд

Пряма, проведена через ці дві точки, перетинає вісь *x* при значенні

$$x^* = x_n - f(x_n)\frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \qquad (4.1)$$

Це значення використовується для визначення значення функції $f(x^*)$, яке порівнюється із значеннями функцій $f(x_n)$ та $f(x_{n+1})$ і надалі використовується замість того з них, з яким воно співпадає по знаку. Якщо значення $f(x^*)$ недостатньо близько до нуля, то вся процедура повторюється до тих пір, поки не буде досягнутий необхідний ступінь збіжності. Блок-схема алгоритму методу хорд представлено на рис. 4.6.

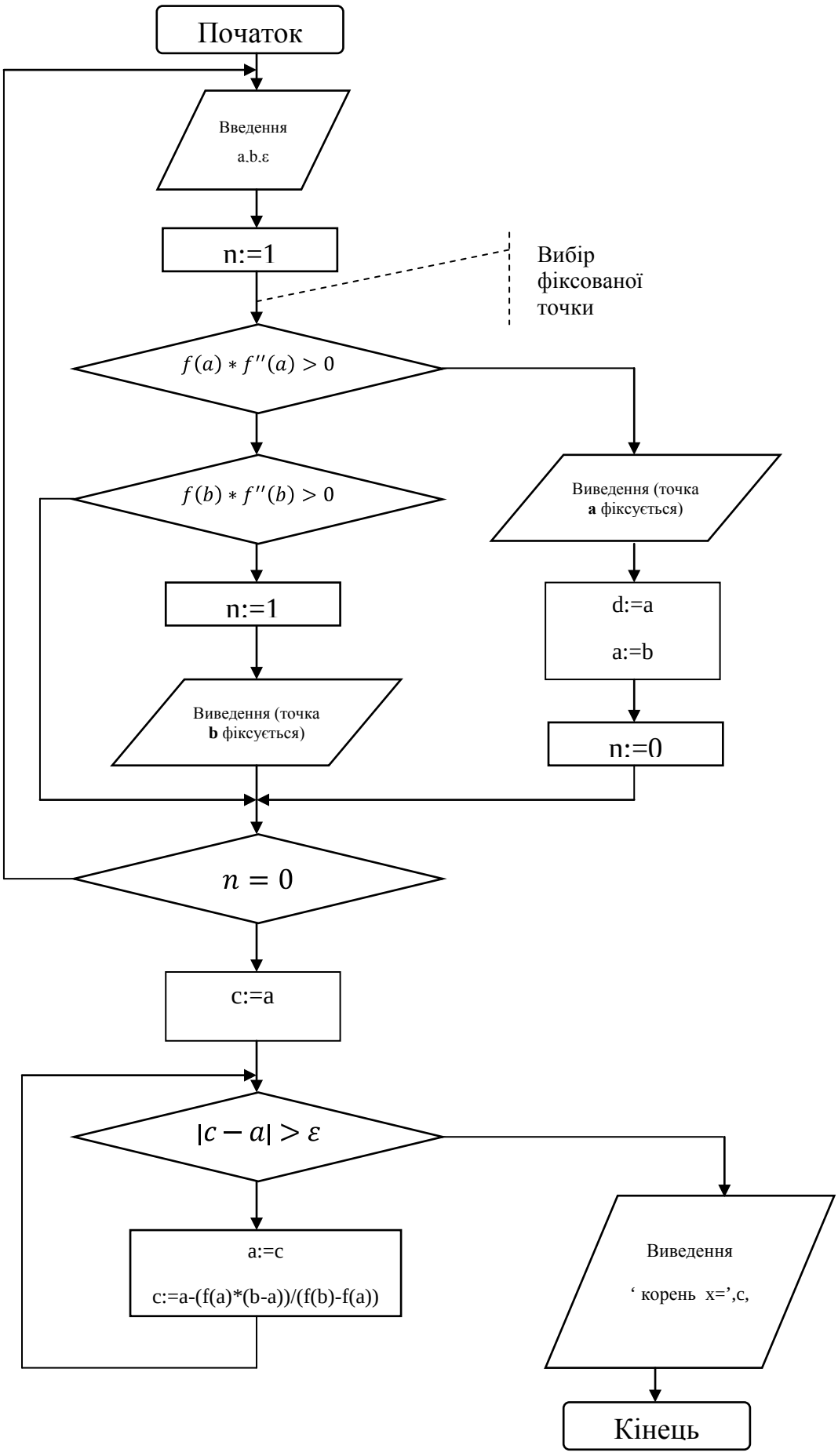


Рис. 4.6. Блок-схема алгоритму методу хорд

Оптимізована під мову *Python* реалізація алгоритму методу хорд представлена на рис. 4.7.

```
def horda(a,b,eps):
    x=lambda a,b: a-f(a)*(b-a)/(f(b)-f(a))

    counter=0;

    if f(a)*f(x(a,b))<=0:
        b=x(a,b)
    else:
        a=x(a,b)

    while (math.fabs(f(x(a,b)))>eps):
        if f(a)*f(x(a,b))<=0:
            b=x(a,b)
        else:
            a=x(a,b)
        print "%d %.4f %.4f %.4f %.4f %.4f %.4f" %(counter,a,b,f(a),f(b),x(a,b),f(x(a,b)))
        counter+=1
```

Рис. 4.7. Реалізація алгоритму методу хорд

на мові Python

Функція horda має три аргументи: *a,b* – інтервал пошуку рішення, *eps* – задана точність пошуку.

4.2.3. Метод Ньютона

Метод послідовних наближень, розроблений Ньютоном, дуже широко використовується при побудові ітераційних алгоритмів. Його популярність обумовлена тим, що на відміну від двох попередніх методів для визначення

інтервалу, в якому поміщений корінь, не потрібно знаходити значення функції з протилежними знаками. Замість інтерполяції по двох значеннях функції в методі Ньютона здійснюється екстраполяція за допомогою дотичної до кривої в даній точці. На рис. 4.8 показана блок-схема алгоритму цього методу, в основі якого лежить розкладання функції *f(x)* в ряд Тейлора

$$f(x_n + h) = f(x_n) + hf'(x_n) + \frac{h}{2} f''(x_n) + \ldots \qquad (4.2)$$

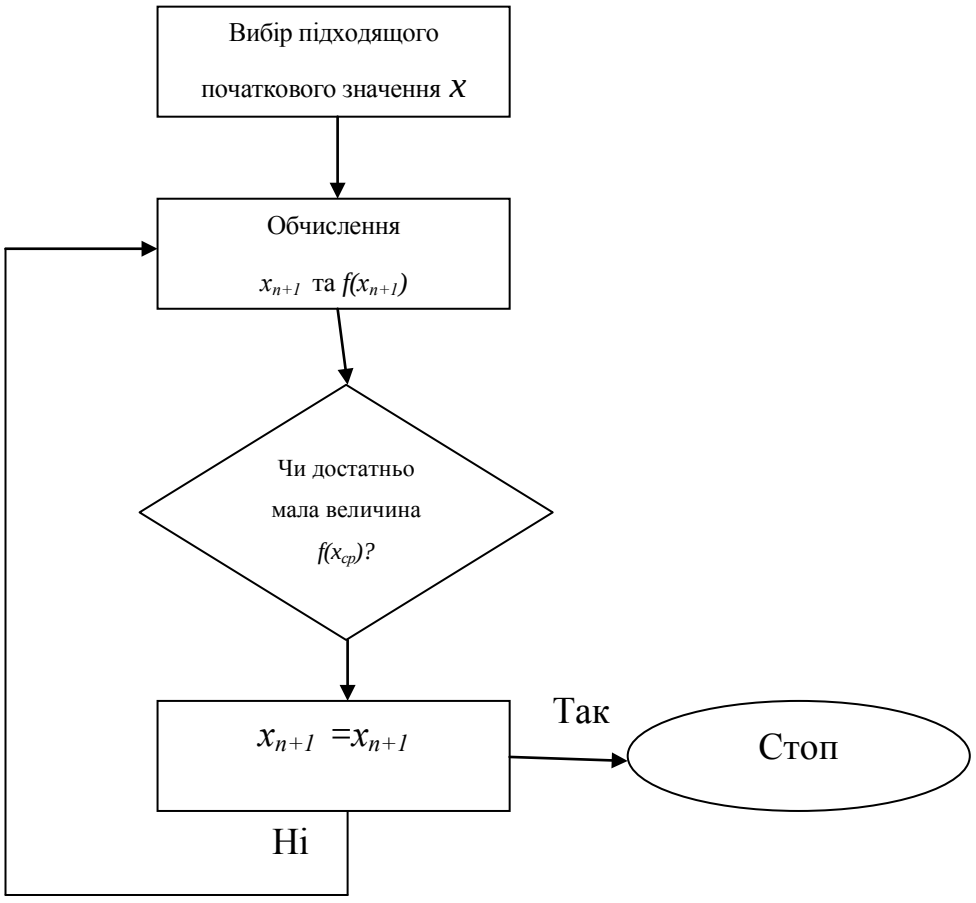


Рис. 4.8. Блок-схема алгоритму методу Ньютона

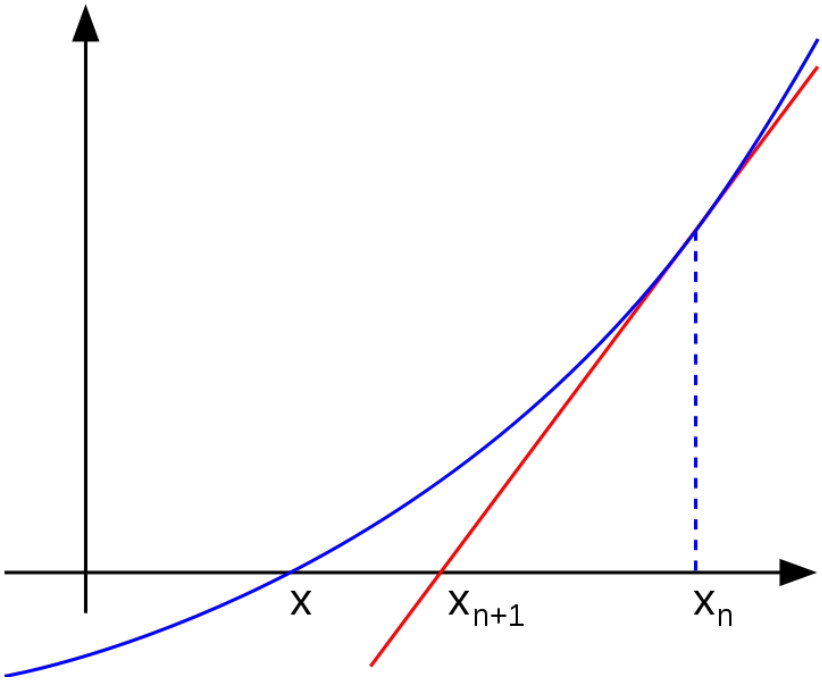


Рис. 4.9. Метод Ньютона

Члени, що містять h, в другій та вищих ступенях похідних, відкидаються; використовуючи співвідношення $x_n+h=x_{n+1}$ Передбачається, що перехід від x_n до x_{n+1} наближає значення функції до нуля так, що при деякому n $f(x_n+h) = 0$. Тоді

$$x_{n+1} = x_n + h = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (4.3)$$

Значення x_{n+1} відповідає точці, в якій дотична до кривої в точці x_n перетинає вісь x. Оскільки крива $f(x)$ відмінна від прямої, те значення функції $f(x_{n+1})$ швидше за все не буде точно дорівнювати нулю. Тому вся процедура повторюється, причому замість x_n використовується x_{n+1}. Процес припиняється після досягнення достатньо малого значення $f(x_{n+1})$. На рис. 4.9 процес розв'язування рівняння методом Ньютона показаний графічно. Ясно, що швидкість збіжності великою мірою залежить від вдалого вибору початкової точки. Якщо в процесі ітерацій тангенс кута нахилу дотичної $f(x)$ перетворюється в нуль, то застосування методу ускладнюється. Можна також показати, що у разі нескінченного великого $f''(x)$ метод також не буде достатньо ефективним. Оскільки умова кратності кореня має вид $f(x)= f'(x)= 0$, то в

цьому випадку метод Ньютона не забезпечує збіжність. Варіант реалізації алгоритму на мові *Python* представлений на рис. 4.10.

```
def newton(a,b,eps):
   counter=0;
   x=a
   t=f(x)/df(x)

   while ((math.fabs(t))>eps):
      t=f(x)/df(x)
      x-=t
      print "%d %.4f %.4f " %(counter,x,t)
      counter+=1
```

Рис. 4.10. Реалізація алгоритму методу Ньютона

на мові Python

Функція newton має три аргументи: *a,b* – інтервал пошуку рішення, *eps* – задана точність пошуку.

4.2.4. Метод простої ітерації

Для застосування цього методу рівняння $f(x)=0$ представляється у вигляді $x=g(x)$. Відповідна ітераційна формула має вигляд:

$$x_{n+1}=g(x_n) \qquad (4.4)$$

Блок-схема алгоритму методу подана на рис. 4.11. Простота методу простої ітерації приваблює, проте не слід забувати, що і цьому методу властиві недоліки, оскільки він не завжди забезпечує збіжність. Тому для будь-якої програми, в якій використовується цей алгоритм, необхідно передбачати контроль збіжності та припиняти рахунок, якщо збіжність не забезпечується.

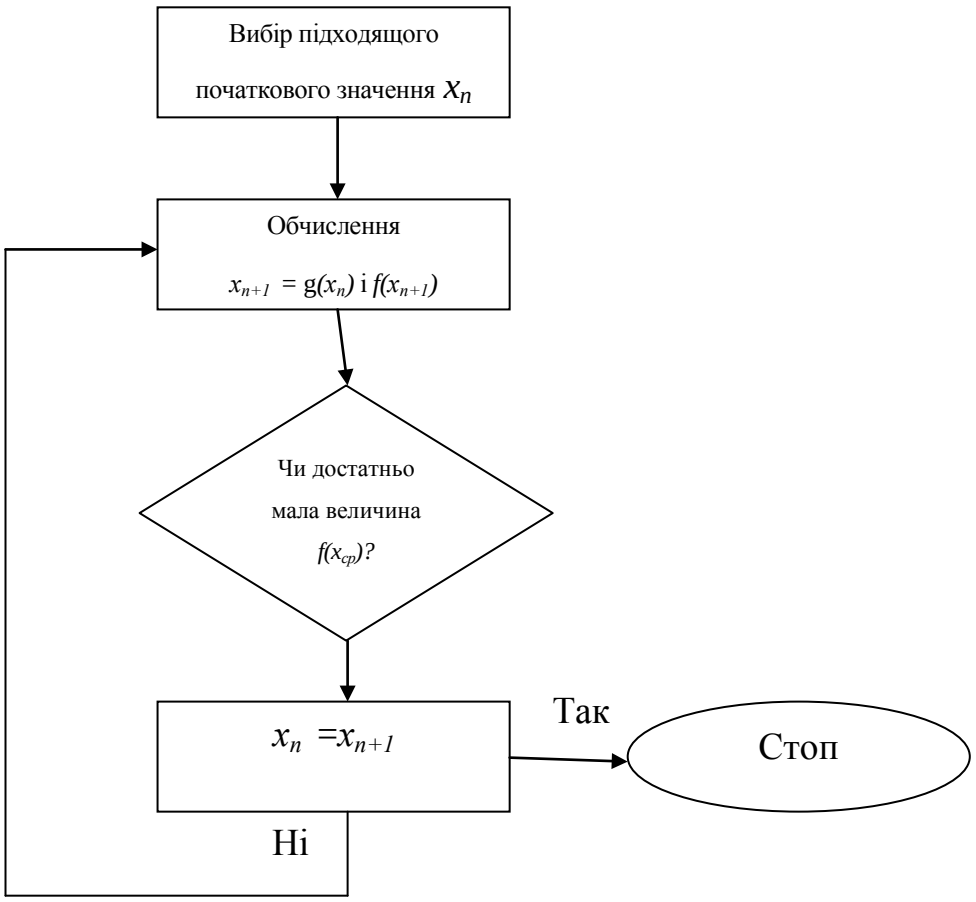


Рис. 4.11. Блок-схема алгоритму методу простої ітерації

Реалізація алгоритму на мові Python представлена на рис. 4.12.

```
def simple_iter(a,b,eps):

    counter=0;
    x=(a+b)/2
    z=x
    x=f(x)
    while ((math.fabs(x-z))>=eps):
        z=x
        x=f(x)
        print "%d %.4f " %(counter,x)
        counter+=1
```

Рис. 4.12. Реалізація алгоритму методу простої ітерації

Функція simple_iter має три аргументи: *a,b* – інтервал пошуку рішення, *eps* – задана точність пошуку. Функція *f(x)* – це представлення рівняння у вигляді *x=f(x)*(наприклад, для рівняння $e^{2x} + 3x - 4 = 0$ функція *f(x)* має вигляд $\frac{\ln(4-3x)}{2}$).

Як приклад пригадаємо раніше розглянуте рівняння x^2-*a=0.* Його коренями є числа X=± $\sqrt{a}$. Для представлення рівняння у вигляді *x=g(x)*, яке може бути неоднозначним, можна запропонувати, наприклад, рівняння $x = \frac{a}{x}$ або $x = 2x - \frac{a}{x}$, які мають такі ж корені. Неважко встановити, що ні та, ні інша формула не годиться для ітерацій. Перша дає незбіжну послідовність $x_1 = a/x_0$, $x_2=x_0$, $x_3 = a/x_0$, а друга породжує послідовність x_n, що прямує до нескінченності. Проте рівняння $x = \frac{1}{2}(x - \frac{a}{x})$, з такими же коренями, як було показано вище, вже дає збіжну послідовність. Таким чином, далеко не всякий запис рівняння у вигляді (4.4) приводить до мети. З'ясуємо, які властивості *g(x)* впливають на збіжність процесу. Оскільки збіжність означає, що x_n — X → 0 при $n \to \infty$, то потрібно, щоб величина $|x_n — X|$ спадала із зростанням *n*. Нехай для будь-якого *n* справедлива нерівність:

$$|x_n — X| \le c|x_{n-1} — X|,\ c<1. \qquad (4.5)$$

Тоді, очевидно $|x_n - X|$ спадає як геометрична прогресія із знаменником *c*, і має місце збіжність $x_n \to X$. Підставимо в ліву частину (4.5) замість x_n і *X* відповідно $g(x_{n-1})$ і $g(X)$. Отримаємо:

$$| g(x_{n-1}) — g(X)| \le c|x_{n-1} — X|,\ c<1. \qquad (4.6)$$

Оскільки корінь *X* невідомий, то останню умову безпосередньо перевірити не можна і доводиться декілька підсилити його. Вище ми припускали, що корінь локалізований усередині деякого інтервалу. Якщо для будь-якої пари точок x', x'' з цього інтервалу виконана умова

$$| g(x') — g(x'')| \le c| x' — x''|,\ c<1. \qquad (4.7)$$

то тим паче виконується і (4.6). Відображення $x = g(x)$, що задовольняє (4.7), називають *стискаючим відображенням* (оскільки воно стискає відрізки $x'' - x'$). Очевидно, умова (4.7) є достатньою умовою збіжності ітераційного процесу (4.4). Якість того або іншого вибору $g(x)$, очевидно, слід оцінювати за швидкістю збіжності. Кращим в цьому сенсі природно вважати той, для якого величина коефіцієнта c буде найменшою.

Приклад 4.1. Методом хорд визначити корінь рівняння

$$f(x) = x^3 + x - 12 = 0. \qquad (4.8)$$

Для наближеного знаходження кореня рівняння (4.8) можна намалювати ескізи графіків функцій $y = x^3$ та $y = 12 - x$. Грубою прикидкою можна виявити, що шуканий корінь, тобто абсциса точки перетину графіків, знаходиться в інтервалі (2,3). Дійсно:

$f(2) = 2^3 + 2 - 12 = -2$

$f(3) = 3^3 + 3 - 12 = +18.$

Тому можна прийняти $x_n = 2$ і $x_{n+1} = 3$.

Застосовуючи формулу (4.1), набудемо наближеного значення кореня:

$$x^* = x_n - f(x_n)\frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} = 2 - \frac{-2(3-2)}{18+2} = 2{,}1$$

Відмітимо, що $f(x^*) = 9{,}261 + 2{,}1 - 12 = -0{,}639$ (тобто корінь лежить правіше). Тому для уточнення значення x* формулу (4.1) слід далі застосувати до відрізка [2,1; 3].

Приклад 4.2. Методом Ньютона визначити корінь того ж рівняння (4.1), що лежить в інтервалі (2,3).

Тут слід дотримуватися правила: якщо друга похідна функції $f''(x)$ зберігає постійний знак в інтервалі (a, b), то дотичну слід проводити в тій кінцевій точці дуги AB, для якої знак функції співпадає із знаком її другої похідної.

Тут $f''(x)= 6x>0$ при $2\le x\le 3$, причому $f(3)= + 18$. Тому у формулі для методу Ньютона (4.3) вважаємо $x_n = 3$. Оскільки $f'(x)= 3x^2+1$ і $f'(3)= 28$, то маємо

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = 3 - \frac{18}{28} = 2{,}36\,.$$

Для контролю відмітимо, що $f(2{,}36)=13{,}14+2{,}36-12=+3{,}35$. Отже для наступної ітерації слід вибирати відрізок (2,36;3) і знову з тією ж точкою $x_n=3$ для дотичної.

Приклад 4.3. Вирішити рівняння

$$f(x) = e^{2x} + 3x - 4 = 0 \qquad (4.9)$$

с точністю $\varepsilon = 10^{-3}$.

Рішення. Для локалізації коренів застосуємо графічний спосіб. Перетворимо вихідне рівняння до наступного еквівалентного вигляду:

$$e^{2x} = 4 - 3x$$

Побудувавши графіки функцій $f_1(x) = e^{2x}$ та $f_2(x) = 4 - 3x$ (рис. 4.13), визначаємо, що в розв'язуваного рівняння є тільки один корінь, що заходиться в інтервалі $0.4 < x^{(*)} < 0.6$.

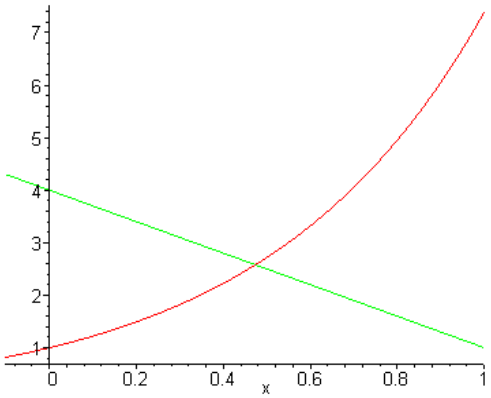


Рис. 4.13. Графіки функцій

Уточнимо значення кореня з необхідною точністю, користуючись методами наведеними вище.

Метод половинного ділення. Як вихідний відрізок виберемо [0.4, 0.6]. Результати подальших обчислень, відповідно до наведеного вище алгоритму містяться в таблиці 4.1.

Табл. 4.1. Результати обчислень методом половинного ділення

k	$a^{(k)}$	$b^{(k)}$	$f(a^{(k)})$	$f(b^{(k)})$	$\frac{a^{(k)}+b^{(k)}}{2}$	$f\left(\frac{a^{(k)}+b^{(k)}}{2}\right)$
0	0.4000	0.6000	-0.5745	1.1201	0.5000	0.2183
1	0.4000	0.5000	-0.5745	0.2183	0.4500	-0.1904
2	0.4500	0.5000	-0.1904	0.2183	0.4750	0.0107
3	0.4500	0.4750	-0.1904	0.0107	0.4625	-0.0906
4	0.4625	0.4750	-0.0906	0.0107	0.4688	-0.0402
5	0.4688	0.4750	-0.0402	0.0107	0.4719	-0.0148
6	0.4719	0.4750	-0.0148	0.0107	0.4734	-0.0020
7	0.4734	0.4750	-0.0020	0.0107	[0.4742]	

$x^{(*)} \approx 0.474$

Метод Ньютона. Для коректного використання даного методу необхідно визначити поведінку першої та другої похідної функції $f(x)$ на інтервалі уточнення кореня та правильно вибрати початкове наближення $x^{(0)}$.

Для функції $f(x) = e^{2x} + 3x - 4 = 0$ маємо:

$f'(x) = 2e^{2x} + 3$, $f''(x) = 4e^{2x}$ - позитивні у всій області визначення функції. Як початкове наближення можна вибрати праву границю інтервалу $x^{(0)} = 0.6$, для якої виконується нерівність:

$$f(0.6)\, f''(0.6) > 0$$

Подальші обчислення проводяться по формулі (3.3), де

$$f(x^{(k)}) = e^{2x^{(k)}} + 3x^{(k)} - 4,\ f'(x^{(k)}) = 2e^{2x^{(k)}} + 3.$$

Ітерації завершуються при виконанні умови $\left|x^{(k+1)} - x^{(k)}\right| < \varepsilon$.

Результати обчислень містяться в таблиці 4.2.

Табл. 4.2. Результати обчислень методом Ньютона

k	$x^{(k)}$	$f(x^{(k)})$	$f'(x^{(k)})$	$-f(x^{(k)})/f'(x^{(}$
0	0.6000	1.1201	9.6402	-0.1162

1	0.4838	0.0831	8.2633	-0.0101
2	0.4738	0.0005	8.1585	-0.0001
3	[0.4737]			

$x^{(*)} \approx 0.474$

Метод простої ітерації. Рівняння (4.9) можна записати у вигляді

$$x = \frac{4 - e^{2x}}{3} \qquad (4.10)$$

або

$$x = \frac{\ln(4 - 3x)}{2} \qquad (4.11).$$

Із двох цих варіантів прийнятним є варіант (4.11), тому що, взявши за основний інтервал (0.4,0.55) та поклавши $g(x) = \frac{\ln(4-3x)}{2}$, будемо мати:

1. $g(x) \in [0.4,0.55] \quad \forall\, x \in [0.4,0.55]$
2. $g'(x) = -\frac{3}{2(4-3x)}$.

Звідси, на інтервалі (0.4,0.55) $|g'(x)| < 0{,}64 = q$.

В якості початкового наближення покладемо $x_0 = (0.4 + 0.55)/2 = 0.475$.

Обчислюємо послідовні наближення x_n з одним запасним знаком по формулі (4.4), де $g(x_n) = \frac{\ln(4 - 3x_n)}{2}$.

Відповідно до (4.7) досягнення необхідної точності контролюється умовою $\frac{q}{1-q}\left|x_{n+1} - x_n\right| \le \varepsilon$.

Результати обчислень наведені в таблиці 4.3.

Табл. 4.3. Результати обчислень методом простої ітерації

n	x_n	$g(x_n)$
0	0.4750	0.4729
1	0.4729	0.4741
2	0.4741	0.4734
3	0.4734	0.4738

4	[0.4738]	

$x^{(*)} \approx 0.474$

4.3 Теорема про стискаючі відображення

Для доведення загального результату про збіжність методу простої ітерації введемо декілька додаткових визначень.

Визначення 4.1. Сукупність елементів $\{x\} = X$ утворює *дійсний лінійний простір*, якщо для будь-яких елементів $x \in X$ введено поняття суми з властивостями

$$x_1 + x_2 = x_2 + x_1;$$

$$x_1 + 0 = x_1;$$

$$x_1 + (-x_1) = 0;$$

і добутку на будь-яке дійсне число а з властивостями

$$a\,(x_2 + x_2) = ax_x + ax_2;$$

$$(a_1 + a_2)\,x = a_1x + a_2x;$$

$$a_1\,(a_2x) = (a_1a_2)\,x.$$

Визначення 4.2. Дійсний лінійний простір називається *метричним*, якщо в ньому введено певним чином поняття відстані між його елементами – *метрика*.

Визначення 4.3. Скалярну невід'ємну величину $\rho(x,y)$ називають метрикою простору X, якщо для будь-яких $x,y \in X$ виконується:

$$\rho(x, y) = \rho(y,x) \text{ для } x \neq y;$$

$$\rho(x, x) = 0;$$

$$\rho(x, y) \leq \rho(x, z) + \rho(z, y);\ x < z < y.$$

Визначення 4.4. Дійсний лінійний простір називається *нормованим*, якщо

кожному елементу $x \in X$ поставлено у відповідність ненегативне число ||x||, що називається нормою, яке задовольняє наступним умовам:

$$||x|| \geq 0,$$

$$a||x|| = |a|||x||,$$

$$||x+y|| \leq ||x|| + ||y||$$

У лінійному нормованому просторі метрику вводять як норму різниці між елементами

$$\rho(x, y) = ||x - y||.$$

Найбільш уживаними є наступні види норм: L^1-норма – середнє значення на інтервалі [0, T]:

$$\|f(x)\|_{L^1} = \frac{1}{T}\left|\int_0^T f(x)dx\right|$$

L^2-норма – ефективне значення погрішності на [0, T]:

$$\|f(x)\|_{L^2} = \frac{1}{T}\left|\int_0^T f^2(x)dx\right|$$

M-норма – мажоранта для всіх значень абсолютної похибки: $\|f(x)\|_M = \max_x |f(x)|$

Послідовність $\{x_n\}$ елементів простору X називається збіжною до елементу $x_0 \in X$, якщо з того, що $||x_m - x_n|| \to 0$, випливає існування такого $x_0 \in X$, що $x_n \to 0$.

Нормований простір X називається *повним,* якщо кожна послідовність $\{x_n\}$ збігається до елемента цього простору. Повний нормований простір ще називають Банаховим.

Метод простої ітерації полягає в тому, що система рівнянь $f(\mathbf{x})=0$ перетворюється до вигляду $\boldsymbol{x} = g(\boldsymbol{x})$ s ітерації проводяться за формулою: $\boldsymbol{x}^{n+1} = g(\boldsymbol{x}^n)$.

З більш загальних позицій цей метод можна представити таким чином. Нехай H – повний метричний простір, а оператор $\boldsymbol{y = g(x)}$ відображає H в себе. Розглядається ітераційний процес $\boldsymbol{x^{n+1} = g(x^n)}$, для вирішення рівняння $\boldsymbol{x=g(x)}$. Якщо при деякому $q<1$ відображення $\boldsymbol{y=g(x)}$ задовольняє умові:

$$\rho(g(\mathbf{x}_1), g(\mathbf{x}_2)) \le q\cdot\rho(\mathbf{x}_1, \mathbf{x}_2) \qquad (4.12)$$

при всіх $\mathbf{x}_1$, $\mathbf{x}_2$, то таке відображення називають стискаючим.

Теорема 4.1. Якщо відображення $\boldsymbol{y = g(x)}$ є стискаючим, те рівняння $\boldsymbol{x=g(x)}$ має єдиний розв'язок $\boldsymbol{X}$ і $\rho(X, x^n) \le \frac{q^n \rho(x^1, x^0)}{1-q}$.

Доведення. З визначення стискаючого відображення (4.12) маємо

$\rho(x^{n+1}, x^n) = \rho(g(x^n), g(x^{n-1})) \le q\rho(x^n, x^{n-1})$ та тому $\rho(x^{n+1}, x^n) \le q^n \rho(x^1, x^0)$. При $l > n$ маємо ланцюжок нерівностей:

$$\rho(x^l, x^n) \le \rho(x^l, x^{l-1}) + ... + \rho(x^{n+1}, x^n) \le$$

$$\le q^{l-1}\rho(x^1, x^0) + ... + q^n \rho(x^1, x^0) \le$$

$$\le q^n \rho(x^1, x^0) \sum_{i=0}^{\infty} q^i = \frac{q^n \rho(x^1, x^0)}{1-q} \qquad (4.13)$$

Згідно з критерієм Коші про фундаментальну послідовність послідовність $\{\boldsymbol{x}^n\}$ має границю $\boldsymbol{X}$. Переходячи до границі в (4.13) при $l \to \infty$, отримуємо:

$$\rho(X, x^n) \le \frac{q^n \rho(x^1, x^0)}{1-q}$$

С іншого боку, справедливий ланцюжок співвідношень:

$$\rho(X, g(X)) \le \rho(X, x^{n+1}) + \rho(x^{n+1}, g(X)) = \rho(X, x^{n+1}) + \rho(g(x^n), g(X)) \le$$

$$\le \rho(X, x^{n+1}) + q\rho(x^n, X) \le 2\frac{q^{n+1}\rho(x^1, x^0)}{1-q} \qquad (4.14)$$

Переходячи в (3.14) до границі, маємо $\rho(X, g(X)) = 0$, і, отже, $X = g(X)$. Якби рівняння мало б два розв'язки X_1 и X_2, то $\rho(X_1, X_2) = \rho(g(X_1), g(X_2)) \le q\rho(X_1, X_2) < \rho(X_1, X_2)$ і ми приходимо до суперечності. Теорема доведена.

4.4 Завдання для самостійної роботи

1. Знайти розв'язки наступних трансцендентних рівнянь:

1) 0,25X – sinX = 0;

2) sinX– 1/X = 0;

3) lnZ–sinZ = 0.

2. Знайти хоч би один з дійсних коренів наступних алгебричних рівнянь:

а) x4 + 7x3 + 3x2 + 4x + 1=0;

б) 7x4 + 5x3 + 2x2 + 4x+1 =0;

в) $X^4 + 5X^3 + 5X^2 – 5X + 6 = 0;$

г) $X^5 + x^4 + 2X^2 – x – 2 = 0.$

4.5 Контрольні запитання

1. У чому відмінність алгебраїчного рівняння від трансцендентного?

2. Чи має метод половинного ділення гарантовану збіжність?

3. Яким чином вибирається початкове наближення в методі Ньютона?

4. Для яких функцій не рекомендується застосовувати методо Ньютона?

5. Чи може інтервал в методі хорд знаходитися з однієї сторони від корня?

6. У чому суть методу половинного ділення?

7. Розкрийте алгоритм методу простої ітерації.

8. Що таке стискаюче відображення?

9. Розкрити поняття дійсного лінійного простору.

10. Який простір називається метричним?

11. Який простір називається нормованим?

12. Який нормований простір називається повним?

Розділ 5. АПРОКСИМАЦІЯ ФУНКЦІЙ

5.1.Основні поняття

5.1.1. Постановка задачі

Задача апроксимації (наближення) функцій не тільки часто виникає в інженерній практиці, але зустрічається як допоміжна при розв'язуванні більш складних задач, наприклад диференціальних рівнянь.

Нехай величини ***x*** та ***y*** зв'язані залежністю ***y=f(x).*** Однак часто вигляд функції ***f(x)*** невідомий, а існує лише таблиця її значень в ряді точок, одержана, наприклад, експериментальним шляхом. Нам же можуть знадобитися значення величини **y** в інших точках, відмінних від вузлів x_i. Замість того, щоб проводити надто дорогий експеримент для розширення таблиці, використовують відносно нескладні формули апроксимації функції для наближеного визначення її значень у проміжних точках.

Зустрічається й інша ситуація, коли залежність ***y=f(x)*** відома, але дуже складна (містить важко обчислювальні вирази, складні інтеграли та ін.) і обчислювання кожного значення вимагає великих зусиль. В цьому випадку також складають таблицю значень, а для обчислення функції в проміжних точках використовують метод апроксимації.

Отже, необхідно використати таблицю значень функції на відрізку ***[a,b]*** для приблизного обчислення її значень в будь-якій точці цього відрізка. Для цієї мети існує задача апроксимації (наближення) функції ***f(x)***: цю функцію приблизно замінити (апроксимувати) деякою функцією ***φ(x)*** так, щоб відхилення ***φ(x)*** від ***f(x)*** були найменшими. Функцію ***φ(x)*** називають апроксимуючою. Якщо одержана ця функція, є можливість для будь-якого значення ***x*** обчислити відповідне значення ***y***, приблизно таке, що дорівнює ***f(x).*** Як функцію ***φ(x)*** беруть узагальнений многочлен:

$$\varphi(x) = a_0\psi_0(x) + a_1\psi_1(x) + ... + a_n\psi_n(x)$$

де $\psi_i(x)$ – система деяких функцій (тригонометричних, степеневих та ін.). Частіше функцію ***y=f(x)*** апроксимують звичайним многочленом:

$$\varphi(x) = a_0 + a_1 x + ... + a_n x^n = \sum_{i=0}^{n} a_i x^i \qquad (5.1)$$

тобто за функції $\psi_i(x)$ беруть степеневі функції:

$$\psi_0(x) = 1, \quad \psi_1(x) = x, \quad \psi_2(x) = x^2, \ ..., \ \psi_n(x) = x^n$$

Природньо, коефіцієнти $\boldsymbol{a_i}$ многочлена (5.1) підбирають так, щоб $\boldsymbol{\psi(x)}$ мало відрізнявся від $\boldsymbol{f(x)}$, тобто відхилення многочлена від функції було найменшим.

Залежно від того, що розуміють під цим відхиленням, розрізняють такі задачі апроксимації: інтерполяція, середньоквадратичне наближення, рівномірне наближення.

5.1.2. Інтерполяція, середньоквадратичне та рівномірне наближення

Інтерполяція є одним з основних видів апроксимації. Вона полягає в тому, що потрібно побудувати многочлен $\boldsymbol{\psi(x)}$, який набирає у заданих точках $\boldsymbol{x}$, ті самі значення $\boldsymbol{y_i}$, що і функція $\boldsymbol{f(x)}$, тобто

$$\varphi(x_i) = y_i, \ i = 0, 1, ..., n, \quad x_0 = a, \quad x_n = b.$$

При цьому вважається, що серед значень $\boldsymbol{x_i}$, немає однакових, тобто $\boldsymbol{x_i \neq x_i}$ при $\boldsymbol{i \neq j}$. Точки x_i називають вузлами інтерполяції, многочлен $\boldsymbol{\varphi(x)}$ – інтерполяційним многочленом.

Таким чином, особливістю інтерполяції є те, що апроксимуючий многочлен $\boldsymbol{\varphi(x)}$ проходить через табличні точки, а в інших точках відрізка $\boldsymbol{[a,b]}$ відображає функцію $\boldsymbol{y=f(x)}$ з деякою точністю (рис. 5.1).

Якщо один і той самий многочлен (його степінь дорівнює $\boldsymbol{n}$) інтерполює $\boldsymbol{f(x)}$ на усьому відрізку $\boldsymbol{[a,b]}$, то говорять про глобальну інтерполяцію.

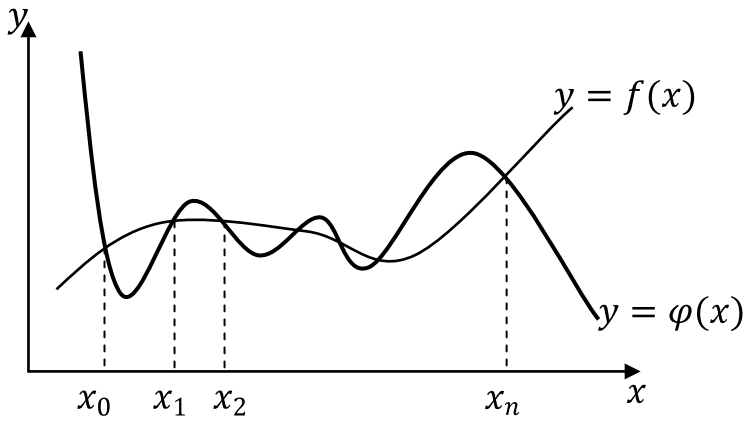


Рис. 5.1. Інтерполяція

Якщо для окремих частин відрізка ***[a,b]*** будують різні многочлени, то такий підхід називають кусковою (локальною, багатоінтервальною) інтерполяцією.

Як правило, інтерполяційний многочлен ***φ(x)*** використовують для апроксимації функції в точках, які належать відрізку ***[a,b]***. Але інколи його використовують для обчислення значень функції поза цим відрізком, тобто при ***x<a*** або ***x>b***. Таке наближення називають екстраполяцією. Однак ним слід користуватися обережно, тому що поблизу кінців відрізка ***[a,b]*** інтерполяційний поліном має коливальний характер із зростаючою амплітудою, а поза відрізком швидко зростає.

Обов'язкове проходження інтерполяційного полінома через табличні точки обумовлює деякі недоліки цього підходу. Оскільки степінь інтерполяційного полінома зв'язаний з кількістю вузлів (при ***n+1*** вузлі степінь дорівнює ***n***), то за великої кількості вузлів степінь полінома буде високою. Це збільшує час обчислення значень цього поліному та помилки округлення. Крім того, табличні дані можуть бути одержані вимірюванням та містити в собі помилки. Інтерполяційний поліном буде точно повторювати ці помилки. Тому інколи вимагають, щоб графік апроксимуючої функції проходив не через табличні точки, а поряд з ними.

Такий підхід використовують при середньоквадратичному наближенні, коли мірою відхилення многочлена *φ(x)* від функції *f(x)* є

$$S = \sum_{i=0}^{n} [\varphi(x_1) - y_1]^2 \qquad (5.2)$$

тобто сума квадратів різниць значень многочлена та функції в вузлових точках (рис. 5.2). Степінь ***m*** апроксимуючого многочлена ***φ(x)*** беруть, як правило, невисокою (*m=1,2,3*), а коефіцієнти цього многочлена беруть так, щоб міра відхилення *S* була мінімальною. В цьому полягає метод найменших квадратів.

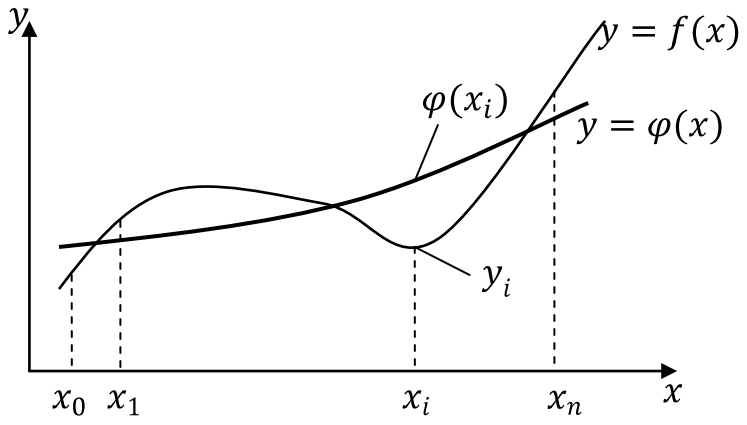


Рис. 5.2. Середньоквадратичне наближення

Поліном ***φ(x)***, одержаний при середньоквадратичному наближенні, може в деяких вузлах сильно відрізнятися від функції ***f(x)***, як наприклад, у вузлі x_k на рис. 5.3.

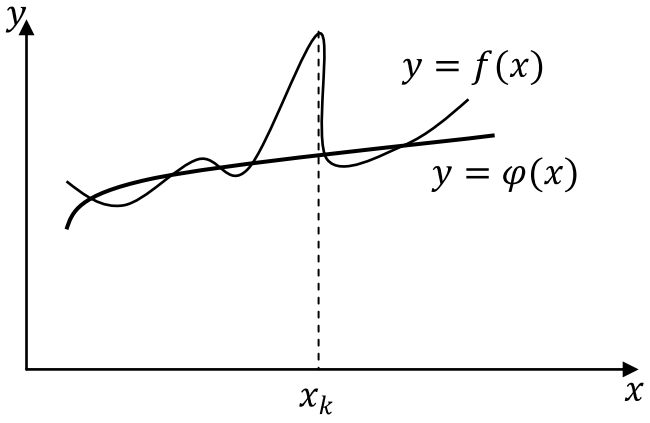


Рис. 5.3. Середньоквадратичне наближення

Тому інколи ставлять більш жорстку вимогу: многочлен повинен відхилятися від функції *f(x)* не більше, ніж на *ε>0*, тобто

$$|f(x) - \varphi(x)| < \varepsilon, \quad a \le x \le b \qquad (5.3)$$

У цьому полягає задача рівномірного наближення. Тоді говорять, що многочлен ***φ(x)*** рівномірно апроксимує функцію ***f(x)*** на відрізку ***[a,b].*** Графік функції в такому випадку розміщений в "трубі" шириною 2ε відносно многочлена ***φ(x)*** (рис. 5.4).

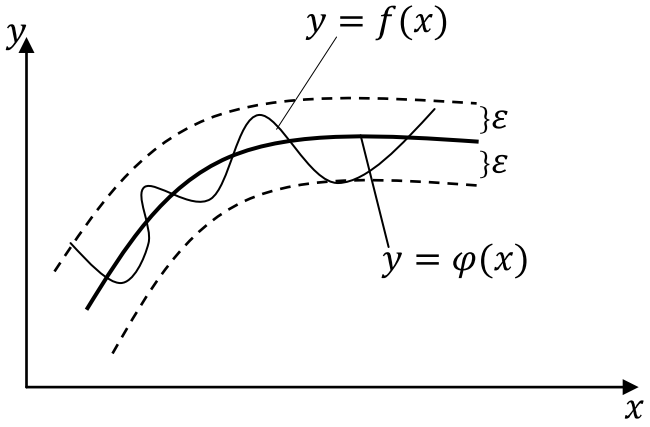


Рис. 5.4. Рівномірне наближення

Можливість побудови такого многочлена для будь-якої величини ε випливає з теореми Вейєрштрасса [3]. Нагадаємо, що абсолютним відхиленням ***Δ*** многочлена ***φ(x)*** від функції ***f(x)*** на відрізку ***[a,b]*** називають максимальну відмінність їх значень:

$$\Delta = \max_{a \le x \le b} \left| f(x) - \varphi(x) \right|$$

Теорема 5.1. Якщо функція ***f(x)*** неперервна на відрізку ***[a,b]***, то для будь-якого ***ε>0*** існує многочлен ***φ(x)*** степені ***m = m(ε)***, абсолютне відхилення якого від функції ***f(x)*** на відрізку ***[a,b]*** менше за ***ε***.

З іншого боку існує поняття найкращого наближення функції ***f(x)*** многочленом ***φ(x)*** при фіксованому степені. Тут серед многочленів степеня ***m*** необхідно знайти такий, щоб абсолютне відхилення було найменшим. Такий многочлен називають многочленом найкращого рівномірного наближення. Його існування виходить з такої теореми [3].

Теорема 5.2. Для будь-якої функції ***f(x)***, неперервної на відрізку ***[a,b]***, та будь-якого натурального ***m*** існує многочлен ***φ(x)*** степеня не вище ***m***, абсолютне відхилення якого від функції ***f(x)*** мінімальне, причому такий многочлен єдиний.

5.2. Глобальна інтерполяція

5.2.1. Лінійна та квадратична інтерполяція

Хоча на практиці лінійні та квадратичні функції використовують як інтерполяційні лише при багатоінтервальній інтерполяції, розглянемо спочатку ці прості випадки.

Нехай таблиця значень функції ***y=f(x)*** (аналітичний вираз ***f(x)*** вважаємо невідомим) складається лише з двох точок ***(x_0,y_0)*** та ***(x_1,y_1)***. Потрібно одержати формулу наближеного обчислення її значень для будь-якого ***$x \neq x_i$ (i=0,1)***, тобто необхідно побудувати інтерполяційний многочлен ***y=φ(x)***, такий, що ***φ(x)≈f(x)***.

Природним розв'язанням цієї задачі є проведення прямої через дві точки (рис. 5.5) та складання її рівняння.

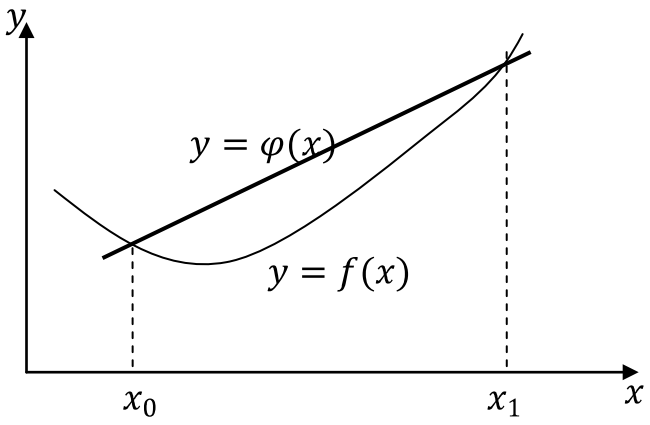


Рис. 5.5. Лінійна інтерполяція

Як відомо, таке рівняння за умови, що $y_0 \neq y_1$, має вигляд:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}$$

або

$$y = \left(y_0 - x_0 \frac{y - y_0}{x_1 - x_0} \right) + x \left(\frac{y - y_0}{x_1 - x_0} \right) \qquad (5.4)$$

Позначивши

$$a_0 = y_0 - x_0 \frac{y - y_0}{x_1 - x_0}, \qquad a_1 = \frac{y - y_0}{x_1 - x_0} \qquad (5.5)$$

одержуємо:

$$y = a_0 + a_1 x = \varphi(x) \qquad (5.6)$$

тобто рівняння зведено до стандартного вигляду запису многочленів.

У випадку, якщо $y_0 = y_1$, пряма буде горизонтальною і їй відповідає рівняння $y = y_0$, тобто інтерполяційний поліном має нульовий степінь. Відмітимо, що через дві точки можна провести скільки завгодно поліномів другого, третього та т.д. степенів.

Таким чином, для двох вузлів $\boldsymbol{x_0, x_1}$ $(n=1)$ можна побудувати єдиний інтерполяційний многочлен $\boldsymbol{\varphi(x)}$ степеня не вище першого та несчислену множину поліномів більш високих степенів.

Це саме рівняння многочлена першого степеня можна одержати інакше. Шукатимемо функцію вигляду $\boldsymbol{y = a_0 + a_1 x}$, графік якої проходить через точки (x_0, y_0) та (x_1, y_1). Із цих умов інтерполяції одержимо систему двох лінійних рівнянь відносно невідомих a_0, a_1:

$$a_0 + a_1 x_0 = y_0,$$
$$a_0 + a_1 x_1 = y_1$$

або

$$\begin{pmatrix} 1 & x_0 \\ 1 & x_0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix}$$

Віднімемо з другого рівняння перше та знайдемо невідомі:

$$a_1 x_1 - a_1 x_0 = y_1 - y_0$$

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0}, \quad a_0 = y_0 - a_1 x_0 = y_0 - x_0 \frac{y_1 - y_0}{x_1 - x_0},$$

що збігається з (5.5) та (5.6).

Блок-схема алгоритму лінійної інтерполяції представлена на рис. 5.6.

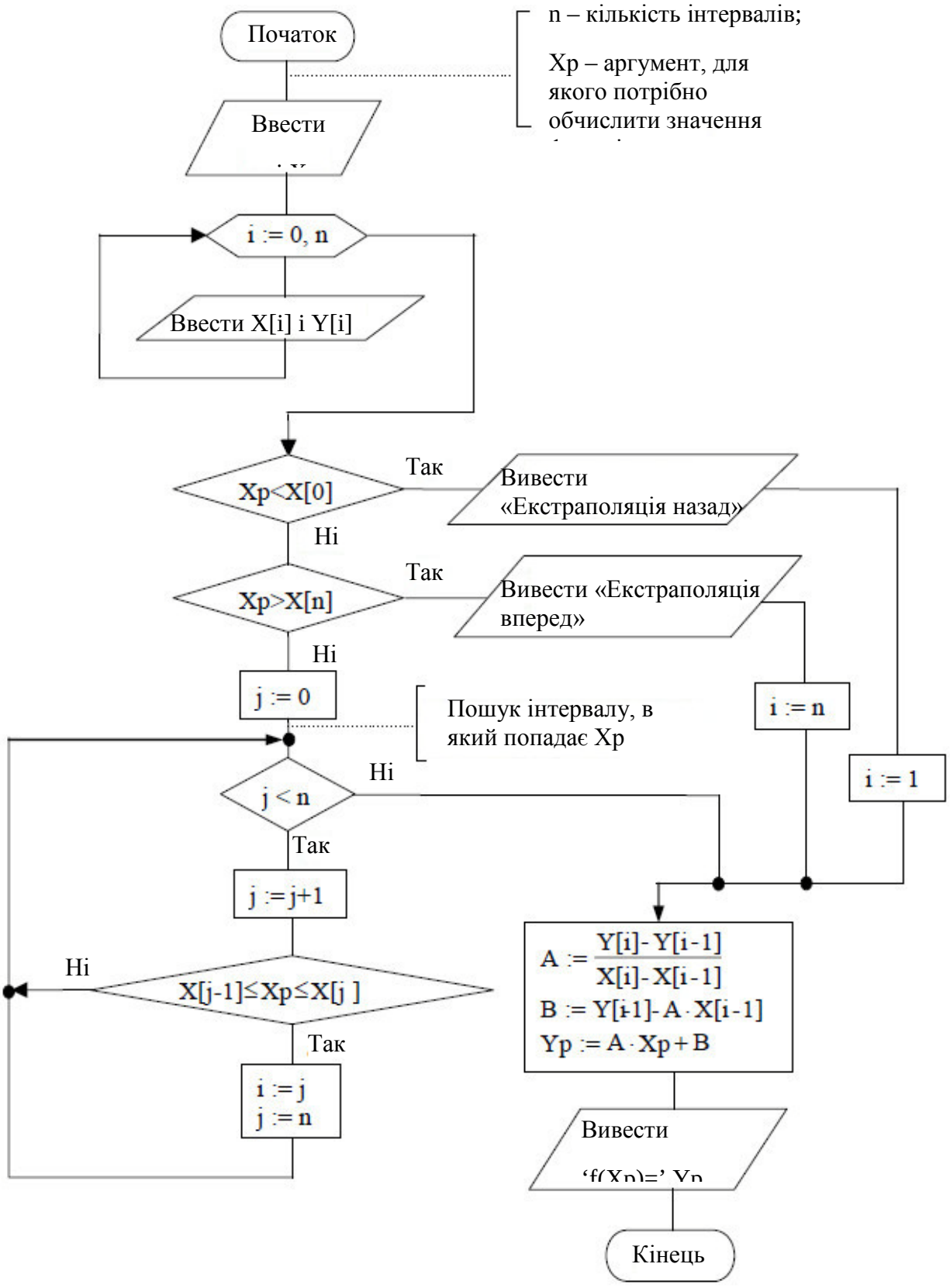


Рис. 5.6. Блок-схема алгоритму лінійної інтерполяції

Реалізація даного алгоритму мовою Python представлена на рис. 5.7.

```
# -*- coding: cp1251 -*-

import numpy as np
```

```
def linear_interp(X,Y,Xp,n=100):
    if (Xp<X[0]) or (Xp>X[n-1]):
        if Xp<X[0]:
            print "Екстраполяція назад"
            i=1
        if Xp>X[n-1]:
            print "Екстраполяція вперед"
            i=n-1
    else:
        for j in xrange(1,n):
            if (X[j-1]<=Xp) and (Xp<=X[j]):
                i=j
                j=n
    A=(Y[i]-Y[i-1])/(X[i]-X[i-1])
    B=(Y[i-1]-A*X[i-1])
    Yp=A*Xp+B
    return A,B,Yp

X=np.array([10.,20.,30.,40.,50.,60.])
Y=np.array([0.17365,0.34202,0.5,0.64279,0.76604,0.86603])
Xp=23.

A,B,Yp=linear_interp(X,Y,Xp,n=3)
print "A=",A, "B=",B, "Yp=",Yp
```

Рис. 5.7. Реалізація алгоритму лінійної інтерполяції

Якщо таблиця значень функції ***y=f(x)*** складається з трьох точок: ***(x_0,y_0), (x_1,y_1), (x_2,y_2)*** маємо можливість скористатися останнім підходом. Будемо шукати інтерполяційний многочлен у вигляді

$$\varphi(x) = a_0 + a_1 x + a_2 x^2 \qquad (5.7)$$

Виходячи з умови проходження параболи (5.7) через три точки, одержимо систему трьох рівнянь з трьома невідомими a_0, a_1, a_2:

$$a_0 + a_1 x_0 + a_2 x_0^{\ 2} = y_0,$$
$$a_0 + a_1 x_1 + a_2 x_1^{\ 2} = y_1,$$
$$a_0 + a_1 x_2 + a_2 x_{20}^{\ 2} = y_2$$

або

$$\begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}$$

Після її розв'язування многочлен (5.7) буде визначений та ним можна скористатися для обчислення функції ***y = f(x)≈φ(x)*** в проміжних точках (рис. 5.8).

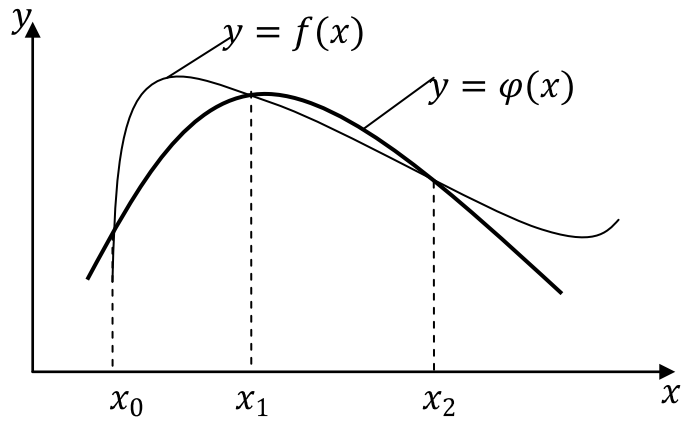


Рис. 5.8. Квадратична інтерполяція

Через три точки можна провести тільки одну параболу та нескінченну множину графіків поліномів більш високих степенів. Якщо три точки розміщено на прямій, тоді єдиний поліном, який проходить через них, матиме перший степінь. Якщо, крім того, ця пряма горизонтальна, то цей поліном має нульовий степінь. Таким чином, за трьома точками можна

побудувати єдиний поліном степеня не вище другого та нескінчену множину поліномів третього, четвертого і т.д. степенів.

5.2.2. Інтерполяційна формула Лагранжа

У п.5.2.1 нами розглянуті прості випадки, коли таблиця значень функції ***y=f(x)*** складається тільки з двох або трьох точок, за якими ми будували інтерполяційний поліном ***f(x)≈φ(x)*** першого або другого степеня. Нехай тепер таблиця значень функції ***y=f(x)*** на відрізку ***[a,b]*** задана в ***(n+1)***-й точці $x_0 = a, x_1, x_2, ..., x_{n-1}, x_n = b.$ Виконаємо глобальну інтерполяцію, тобто для всього відрізку ***[a,b]*** побудуємо єдиний інтерполяційний поліном, який шукатимемо у вигляді

$$\varphi(x) = a_0 + a_1 x + ... + a_n x^n \qquad (5.8)$$

Однак чи можливе розв'язування цієї задачі? Чи існує такий поліном? Відповідь на це запитання про існування та єдиність інтерполяційного полінома дає така теорема [2].

Теорема 5.3. Якщо вузли $x_0, x_1, x_2, ..., x_n$ різні, тоді для будь-яких $y_0, y_1, y_2, ..., y_n$ існує єдиний поліном $\varphi(x)$ степеня не вище n такий, що

$$\varphi(x_i) = y_i, \ \ i = 0, 1, 2, ..., \ n. \qquad (5.9)$$

Як же побудувати такий поліном ***φ(x)***? Достатньо очевидним є наступний підхід, який називають *методом невизначених коефіцієнтів*.

З умови (5.9) рівності значень полінома ***φ(x)*** у вузлах x_i, відповідним табличним значенням y_i можна одержати систему ***n+1****-го* рівнянь відносно ***n+1****-ї* невідомої $a_0, ..., a_n$:

$$\begin{aligned} a_0 + a_1 x_0 + ... + a_n x_0^n &= y_0, \\ a_0 + a_1 x_1 + ... + a_n x_1^n &= y_1, \\ &... \\ a_0 + a_1 x_n + ... + a_n x_n^n &= y_n \end{aligned} \qquad (5.10)$$

аналогічну простим системам п.5.2.1. Матрицю цієї системи

$$A = \begin{pmatrix} 1 & x_0 & K & x_0^n \\ 1 & x_1 & K & x_{01}^n \\ & & K & \\ 1 & x_n & K & x_n^n \end{pmatrix} \qquad (5.11)$$

називають матрицею Вандермонда. В літературі показано, що її визначник не дорівнює нулю, якщо серед вузлів $\boldsymbol{x_j}$ немає таких, що збігаються ($x_i \neq x_j$ при $i \neq j$). Тому система (5.10) має єдиний розв'язок, який містить коефіцієнти $\boldsymbol{a_i}$ інтерполяційного полінома.

Такий підхід інколи корисний для теоретичних досліджень, але для практичного використання він мало придатний внаслідок своєї трудомісткості та поганої обумовленості системи при $\boldsymbol{n \geq 5}$.

Єдиний інтерполяційний поліном, про який йшлося в попередній теоремі, може бути побудований декількома способами. Розглянемо лише одну форму запису цього полінома – *формулу Лагранжа.*

Розглянемо спочатку деякі допоміжні поліноми $l_i(x)$, які називають *поліномами Лагранжа* та мають такі властивості:

$$l_i(x_j) = \begin{cases} 1 \ при & i = j \\ 0 \ при & i \neq j \end{cases} \qquad (5.12)$$

Тобто поліном $\boldsymbol{l_j(x)}$ з номером $\boldsymbol{i}$ дорівнює одиниці тільки у вузлі з таким самим номером $\boldsymbol{x_j}$, а в інших вузлах він дорівнює нулю (рис. 5.9). Для деякої нерівномірної сітки $\boldsymbol{x_i}$ *(i=0,1,...,n)* кожний з $\boldsymbol{n+1}$*-го* поліномів $\boldsymbol{l_i(x)}$ можна подати у вигляді

$$l_i(x) = c_i(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n) \qquad (5.13)$$

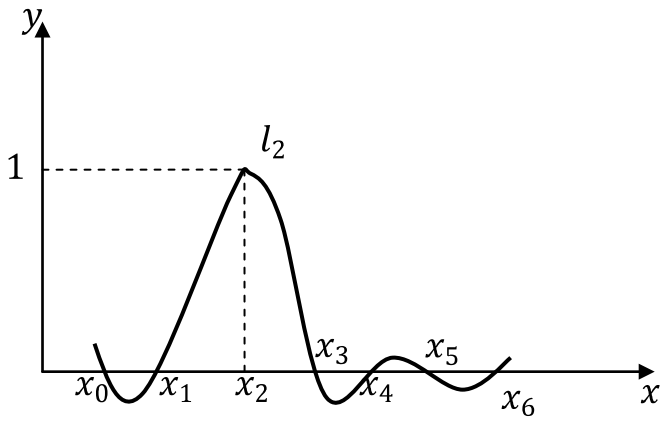


Рис. 5.9. Поліном Лагранжа

Перевіряючи, можна впевнитися, що такий поліном дійсно дорівнює нулю в усіх вузлах, крім вузла x_i. Підберемо тепер коефіцієнт c_i так, щоб у вузлі x_j поліном $l_j(x)$ дорівнював одиниці згідно з (5.12). Підставимо в (5.13) $x = x_i$:

$$1 = c_i (x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)$$

та знайдемо c_i:

$$c_i = \frac{1}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)}$$

Підставимо одержаний вираз у (5.13):

$$l_i(x) = \frac{(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)} \qquad (5.14)$$

Перейдемо до побудови за допомогою поліномів ***n**-го* степеня $l_i(x)$ інтерполяційного полінома, який задовольняє умови (5.9). Розглянемо такий многочлен:

$$\varphi(x) = y_0 l_0(x) + y_1 l_1(x) + ... + y_n l_n(x) = \sum_{i=0}^{n} y_i l_i(x) =$$
$$= \sum_{i=0}^{n} y_i \frac{(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)} = L_n(x) \qquad (5.15)$$

Чи буде він для довільного вузла x_k дорівнювати $y_к$? Знайдемо значення цього полінома при $x = x_к$:

$$\varphi(x_k) = y_0 l_0(x_k) + y_1 l_1(x_k) + ... + y_k l_k(x_k) + ... + y_n l_n(x_k)$$

У цій сумі відповідно (5.12) всі доданки дорівнюють нулю, крім одного, який дорівнює $y_k l_k(x_k) = y_k$. Отже, одержаний поліном (5.15) проходить через табличні точки, тобто є інтерполяційним. Оскільки кожна складова в (5.15) є поліномом ***n**-го* степеня, то і вся сума є поліномом ***n**-го* степеня.

Формулу (5.15) називають *інтерполяційною формулою Лагранжа*. Вона визначає єдиний інтерполяційний поліном ***n**-го* степеня $L_n(x)$.

У випадку двох вузлів x_0, x_1 *(n=1)* ця формула має вигляд

$$\varphi(x) = L_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_1}{x_1 - x_0}$$

та збігається з формулою лінійної інтерполяції, одержаною в п.5.2.1.

У випадку трьох вузлів x_0, x_1, x_2 $(n=2)$ формула Лагранжа має вигляд

$$\varphi(x) = L_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + \\ + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \qquad (5.16)$$

Після зведення подібних членів одержимо рівняння параболи

$$y = a_0 + a_1 x + a_2 x^2 \qquad (5.17)$$

яка проходить через три вузли.

Приклад 5.1. За допомогою інтерполяційної формули Лагранжа обчислимо при $x=2$ значення функції $y=f(x)$, яка задана таблицею з трьох точок.

x	0	1	3
y	2	0	1

Розв'яжемо цю задачу двома способами. У формулу (5.16) підставимо табличні значення $x_0, y_0, x_1, y_1, x_2, y_2$, також значення $x=2$:

$$L_2(2) = 2\frac{(2-1)(2-3)}{(0-1)(0-3)} + 0\frac{(2-0)(2-3)}{(1-0)(0-3)} + 1\frac{(2-0)(2-1)}{(3-0)(3-1)} = \\ = \frac{2 \cdot 1(-1)}{3} + \frac{2 \cdot 1}{3 \cdot 2} = -\frac{1}{3}$$

В другому випадку підставимо в формулу (5.16) тільки табличне значення та одержимо інтерполяційний поліном другого степеня у вигляді (5.17):

$$L_2(x) = 2\frac{(x-1)(x-3)}{(0-1)(0-3)} + 0\frac{(x-0)(x-3)}{(1-0)(0-3)} + 1\frac{(x-0)(x-1)}{(3-0)(3-1)} = \\ = \frac{2}{3}(x-1)(x-3) + \frac{1}{6}x(x-1) = \frac{5}{6}x^2 - \frac{17}{6}x + 2.$$

Після цього підставимо значення $x=2$ в одержаний вираз:

$$L_2(2) = \frac{5}{6}2^2 - \frac{17}{6}2 + 2 = -\frac{1}{3},$$

що збігається зі значенням, одержаним першим способом.

На рис. 5.10 показано графік функції $y = L_2(x)$ для цієї задачі. Природно виникає запитання про точність заміни функції $y=f(x)$ поліномом $L_n(x)$. Відмітимо, що у тому випадку, коли функція $f(x)$ є поліномом n-*го* степеня, одержаний за допомогою формули (5.15) інтерполяційний поліном $L_n(x)$ в разі відсутності помилок округлення збігається з $f(x)$ на всій числовій осі.

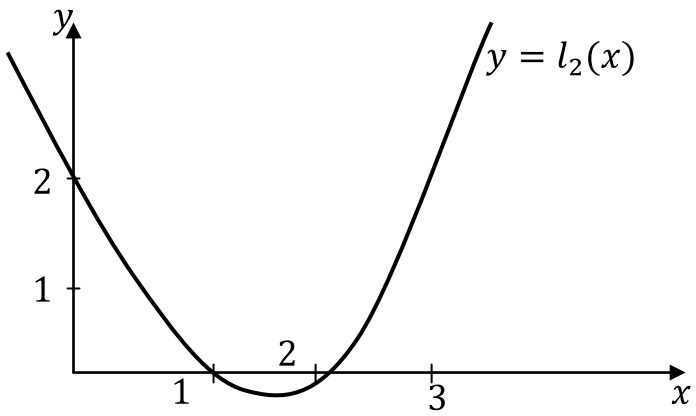


Рис. 5.10. Інтерполяція за трьома точками

У випадку довільної функції $f(x)$ збіг буде тільки у вузлах інтерполяції, а в проміжних точках різниця між $f(x)$ та $L_n(x)$ на дорівнюватиме нулю:

$$R_n(x) = f(x) - L_n(x)$$

Цю величину називають *залишковим членом інтерполяційної формули*. Наступна теорема дає оцінку $R_n(x)$ через старші похідні функції $f(x)$ [2].

Теорема 5.4 (помилка поліномної інтерполяції). Нехай функція $f(x)$ має $(n+1)$ неперервну похідну на деякому інтервалі, що містить відрізок $[a,b]$, та нехай $a = x_0, x_1, \dots, x_n$ – різні вузли.

Тоді, якщо $L_n(x)$ – інтерполяційний многочлен Лагранжа степеня не вище n, що задовольняє співвідношенню

$$L_n(x_i) = f(x_i), \quad i = 0,1,\dots,n$$

тоді для будь якого x із відрізка $[a,b]$

$$R_n(x) = f(x) - L_n(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_n)}{(n+1)!} f^{(n+1)}(z) \qquad (5.18)$$

де z – деяка точка відрізку $[a,b]$.

Позначимо $M = max\ |f^{(n+1)}(z)|$.

Тоді

$$a \le z \le b$$

$$|R_n(x)| \le \frac{|(x-x_0)(x-x_1)...(x-x_n)|}{(n+1)!} M. \qquad (5.19)$$

Якщо вузли x_i розміщено рівномірно з кроком h, то ця оцінка має вигляд: $|R_n(x)| \le Mh^{n+1}$.

Розглянемо на прикладі, як можна використати оцінку похибки (5.19)

Приклад 5.2. З якою точністю можна обчислити $\sqrt{117}$ за допомогою інтерполяційної формули Лагранжа для функції $y = \sqrt{x}$, вибравши вузли інтерполяції $x_0=100, x_1=121, x_2=144$?

На відміну від попереднього прикладу нам відома функція $f(x) = \sqrt{x}$. Тому для використання оцінки (5.19) знайдемо похідні цієї функції

$$(n=2): \qquad y' = \frac{1}{2}x^{-\frac{1}{2}}, \quad y'' = -\frac{1}{4}x^{\frac{3}{2}}, \quad y''' = \frac{3}{8}x^{\frac{5}{2}} = \frac{3}{8\sqrt{x^5}}$$

Знайдемо максимальне значення y''' на відрізку *[100,144]*. Це спадна функція, тому її максимум досягається при $x=100$:

$$M = \max_{100 \le Z \le 144} \left| f^{(2+1)}(Z) \right| = \max_{100 \le Z \le 144} \left| f'''(Z) \right| = \frac{3}{8\sqrt{100}} = \frac{3}{8}10^{-5}$$

Згідно з оцінкою (5.19) одержимо

$$|R_n(x)| \le \frac{|(117-100)(117-121)(117-144)|}{3} \frac{3}{8}10^{-5} \qquad (5.20)$$

На практиці однак складно скористатися оцінкою (5.19), оскільки часто сама функція не задана, а відома тільки таблиця значень, що не дає можливості знайти величину M. Але ця оцінка часто буває корисна для розуміння внутрішньої природи виникаючих помилок. Ми до неї повернемось, коли розглядатимемо кускову інтерполяцію.

Складемо програму на мові *Python* для наближення функції за допомогою інтерполяційної формули Лагранжа. Тут можливі два підходи.

Перший – це підстановка, як і в прикладі 5.1, в формулу (5.15) значення аргументу та табличних значень одночасно. Він простий з точки зору програмування, але тут не формується поліном $\varphi(x)$ у канонічному вигляді (5.8) і тому кількість арифметичних операцій для обчислення $L_n(x)$ велика. В

наведеному далі фрагменті програми (рис. 5.11) масиви ***X*** та ***Y*** містять таблицю значень функції, ***(N+1)*** – довжина таблиці, ***XR*** – масив робочих точок, для яких треба обчислити значення полінома. Вихідні дані накопичуються у масиві *YR*.

```
…
for K in range(1,MR+1):
    Z=XR[K]
    F=0
    for I in range(1,N+2):
        P=1;P1=1;
        for J in range(1,N+2):
            if I not j:
                P=P*(Z - X[J])
                P1=P1*(X[I]-X[J])
        F=F+Y[I]*P/P1
    YR[K]=F
    …
```

Рис. 5.11. Фрагмент програми для інтерполяції методом Лагранжа

Підрахуємо кількість арифметичних операцій, потрібних для обчислення $L_n(x)$ за формулою (5.15) на підставі таблиці, яка складається з ***n+1***-*ї* точки. Для обчислення чисельника кожного дробу потрібно ***n*** операцій віднімання та ***(n-1)*** операцій множення, тобто всього ***(2n-1)*** операцій. Саме стільки операцій потрібно для обчислення знаменника. Для обчислення одного дробу необхідно виконати ***2(2n-1)+2=4n*** операцій. Оскільки в формулі Лагранжа є ***(n+1)***таких дробів, то для їх обчисленя та накопичення суми потрібно $4n(n+1)+n=4n^2+5n$ операцій.

Якщо ця таблиця буде використана декілька разів для різних проміжних (робочих) точок, є рація спочатку одержати одноразово поліном у канонічному вигляді, а після того скористатися ним для обчислення значень

полінома ***φ(x)*** у робочих точках, що потребує набагато менше арифметичних операцій. Перед тим, як перейти до цього другого способу використання полінома Лагранжа для апроксимації функцій, розглянемо схему Горнера для обчислення многочленів.

5.2.3 Обчислення значень многочленів

Необхідність обчислення значень многочленів

$$P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_1 x^n \qquad (5.21)$$

виникає не тільки при апроксимації функції, але і в інших задачах. Нехай коефіцієнти a_i многочлена ***P(x)*** зберігаються в пам'яті ЕОМ у вигляді масиву ***A***:

	1	**2**	**3**		**n+1**
A =	a_0	a_1	a_2	…	a_n

Послідовність розміщення у ньому коефіцієнтів може бути взята і протилежною, що залежить від використаного алгоритму.

Не складним, але малоефективним розв'язуванням цієї задачі є такий алгоритм:

```
...
P=A[1]
for i in range(1,N+1):
    P=P+A[I+1]*X**I
...
```

що реалізує формулу $P = a_0 + \sum_{i=1}^{n} a_i x^i$. Тут при обчислені x^k не враховують уже знайдену величину x^{k-1}, що збільшує загальну кількість арифметичних операцій до $2n + \frac{n(n-1)}{2}$.

Цей недолік ліквідується у такому алгоритмі:

```
...
P=A[1]
```

```
SX=X
for l in range(1,N+1):
    P=P+A[l+1]*SX
    SX=SX*X
...
```

У цьому випадку потрібно виконати тільки ***3n*** арифметичних операцій.

Однак найбільш економним способом обчислення значень многочлена є схема Горнера, яка дає можливість отримати результат за ***2n*** операцій. Вона заснована на тому, що многочлен (5.21) можна подати у вигляді

$$P(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-2}) + x(a_{n-1}))...)). \qquad (5.22)$$

Наприклад, при ***n=3***

$$a_0 + x(a_1 + x(a_2 + xa_3)) = a_0 + x(a_1 + x(a_2 + xa_2 + x^2 a_3)) =$$
$$= a_0 + a_1 x + a_2 x^2 + a_3 x^3 = P(x)$$

Можна показати, що не існує способу обчислення алгебраїчного многочлена ***n-го*** степеня менше, чим за ***2n*** арифметичних операцій. Тому надалі будемо використовувати такі оператори для обчислення значень полінома за схемою Горнера:

```
...
P=A[N+1]
for l in range(1,N+1):
    P=A[N+1-l]+X*P
...
```
(5.23)

5.2.4. Побудова многочлена за допомогою формули Лагранжа

Якщо передбачається багаторазове використання даної таблиці для обчислення значень функції, то недоцільно користуватися безпосередньо формулою Лагранжа, як це зроблено у програмі (5.20). Можна одноразово побудувати за таблицею за допомогою формули Лагранжа многочлен у

канонічному вигляді (5.21), тобто знайти коефіцієнти a_i, а потім його використовувати для кожного значення x.

Оскільки при послідовному множенні двочленів у чисельнику формули (5.15) породжуються многочлени другого, третього і т.д. степенів, кожний з яких повинен бути помноженим на двочлен, розглянемо таку допоміжну задачу: складемо програму множення многочлена степеня l

$$P(x) = \sum_{i=1}^{l} a_i x^i$$

на двочлен $(x-c)$.

Нехай коефіцієнти многочлена $P(x)$ зберігаються в масиві A:

	1	2	3		l+1	
A =	a_0	a_1	a_2	…	a_l	…

Коефіцієнти многочлена $(x-c)P(x)$,тобто результат,будемо накопичувати в масиві B:

	1	2	3		l+1	l+2	
B =	b_0	b_1	b_2	…	b_l	b_{l+1}	…

Оскільки

$$(a_0 + a_1 + ... + a_l x^l)(x - c) = x(a_0 + a_1 + ... + a_l x^l) - c(a_0 + a_1 + ... + a_l x^l) =$$
$$= a_0 x + a_1 x_2 + ... + a_l x^{l+1} - c(a_0 + a_1 + ... + a_l x^l),$$

то стає очевидним, що для одержання масиву B необхідно переписати в нього масив A зі зсувом на одну позицію праворуч, після чого до елементів масиву B додати елементи масиву A, помножені на $(-c)$. Ці два етапи схематично можна зобразити так (рис. 5.12):

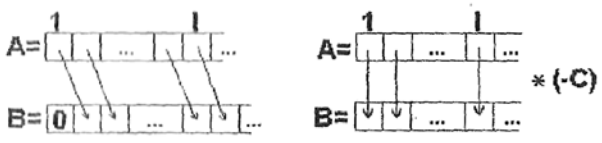


Рис. 5.12. Схема алгоритму

Фрагмент програми, що реалізує ці дії, має вигляд:

```
...
for K in range(1,L+1):
```

```
        B[K+1]=A[K]
    B[1]=0
    for K in range(1,L+1):
        B[K]=B[K]-C*A[K]
    ...
```

У нашій основній задачі ця процедура виконуватиметься багаторазово, причому після кожного множення на двочлен результат знову буде вноситися в масив ***A***. Одержаний для кожного дробу формули Лагранжа многочлен степеня ***n*** у вигляді масиву його коефіцієнтів будемо складати з аналогічними многочленами, одержаними для інших дробів та накопичувати коефіцієнти результуючого многочлена $L_n(x)$ у масиві ***POL***. У цілому програма складатиметься з двох функцій: функції Polynom (формування $L_n(x)$ у канонічному вигляді), функції WPOL (обчислення $L_n(x)$ для кожної робочої точки, які зберігаються в масиві ***XR***). Вихідна таблиця значень невідомої функції нехай складається з ***M*** точок. Програма мовою *Python* має вигляд (рис. 5.13):

```
# -*- coding: cp1251 -*-
import numpy as np
def LAGR2(X,Y,M,XR,YR,MR):
  """ВХІДНІ ДАНІ:   X, Y - таблиця значень функції
                M - довжина таблиці
                XR - масив робочих точок
                MR - кількість робочих точок
    ВИХІДНІ ДАНІ:  YR - масив значень полінома Лагранжа в робочих точках"""


  def Polynom(X,Y,M,POL):
    """підпрограма формування інтерполяційного многочлена"""
```

```
POL=np.array(np.zeros(M+1))
A=np.array(np.zeros(M+1))
B=np.array(np.zeros(M+1))

for i in xrange(1,M+1):
    P=1
    for j in xrange(1,M+1):
        if j==i: continue
        P=P*(X[i]-X[j])
    P=Y[i]/P
    A[1]=1
    L=1
    for j in xrange(1,M+1):
        if j==i: continue
        C=X[j]
        for k in xrange(1,L+1):
            B[k+1]=A[k]
        B[1]=0
        for k in xrange(1,L+1):
            B[k]=B[k]-C*A[k]
        for k in xrange(1,L+2):
            A[k]=B[k]
        L+=1
    for j in xrange(1,M+1):
        POL[j]=POL[j]+A[j]*P;
return POL
```

```
def WPOL(POL,M,XR,YR,MR):
    """підпрограма обчислення значень полінома в робочих точках XR[K]"""
    for k in xrange(1,MR+1):
        P=POL[M]
        Z=XR[k]
        for i in xrange(1,M):
            P=POL[M-i]+Z*P
        YR[k]=P
    return YR


POL=np.array(np.zeros(M+1))
POL=Polynom(X,Y,M,POL)
YR=WPOL(POL,M,XR,YR,MR)
return YR
```

Рис. 5.13. Реалізація алгоритму

Перевірка алгоритму на даних з прикладу 5.1 дала тіж результати. Як видно з останнього рядку значення функції в робочій точці ***x=2*** дорівнює - ***0,(33)***.

```
>>>X=np.array([0.,0.,1.,3.])
>>>Y=np.array([0.,2.,0.,1.])
>>>M=3
>>>XR=np.array([0.,1.,2.,3.])
>>>MR=3
>>>YR=np.array(np.zeros(M+1))
>>>LAGR2(X,Y,M,XR,YR,MR)
>>>>>> YR
array([  0.00000000e+00,   2.22044605e-16,  -3.33333333e-01,
```

```
1.00000000e+00])
```

Ще один варіант реалізації методу Лагранжа представлений на рис. 5.14. Масиви X та Y представляють функцію, яка задана таблично. Параметр argx задає точку, в якій потрібно знайти значення функції.

```
import numpy as np


def lagrange_pol(X,Y,M,argx):
    s=0
    for i in xrange(M):
        c=1
        for j in xrange(M):
            if i!=j:
                c*=(argx-X[j])/(X[i]-X[j])
        s+=c*Y[i]
    return s
```

Рис. 5.14. Реалізація алгоритму методу Лагранжа

Перевірка алгоритму на тих же даних дала ті ж результати. Як видно з останнього рядку значення функції в робочій точці ***x=2*** дорівнює ***-0,(33)***.

```
>>>X=np.array([0.,1.,3.])
>>>Y=np.array([2.,0.,1.])
>>>argx=2
>>>M=3
>>>result=lagrange_pol(X,Y,M,argx)
>>>result
-0.33333333333333331
```

5.2.5 Кінцеві різниці різних порядків

Ми припускатимемо зараз, що початкова таблиця значень функції має

рівновіддалені вузли, тобто $x_{i+1}-x_i = \Delta x_i = h = const\ (i = 0,1,2,...,n-1)$. Константа h називається кроком таблиці.

При вирішенні поставлених задач інтерполяції велике значення мають так звані *кінцеві різниці* даної функції $f(x)$. Кінцевими різницями *1-го* порядку називаються величини:

$$\Delta y_0 = y_1 - y_0 = f(x_0 + h) - f(x_0),$$
$$\Delta y_1 = y_2 - y_1 = f(x_0 + 2h) - f(x_0 + h),$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\Delta y_{n-1} = y_n - y_{n-1} = f(x_0 + nh) - f(x_0 + (n-1)h)$$

Очевидно, що якщо вузлів інтерполяції $n+1$, то кінцевих різниць *1-го* порядку n штук. По ним складаються $n-1$ кінцевих різниць другого порядку:

$$\Delta^2 y_0 = \Delta y_1 - \Delta y_0$$
$$\Delta^2 y_1 = \Delta y_2 - \Delta y_1$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\Delta^2 y_{n-2} = \Delta y_{n-1} - \Delta y_{n-2}$$

Взагалі кінцева різниця порядку $k \le n$ визначається через різниці попереднього порядку: . $\Delta^k y_i = \Delta^{k-1} y_{i+1} - \Delta^{k-1} y_i$. Таких, очевидно, є $n+1-k$ штук. Найвищий порядок можливої різниці є n.

Символ Δ можна розглядати як оператор, що ставить у відповідність функції $y=f(x)$ функцію $\Delta y = f(x + h) — f(x)$. Легко перевірити прості властивості лінійності кінцевих різниць, тобто оператора Δ:

1. Якщо $C = const$, то $\Delta C = 0$

2. $\Delta[f_1(x) + f_2(x)] = \Delta f_1(x) + \Delta f_2(x)$

3. $\Delta[Cf(x)] = C\Delta f(x)$

Кінцеві різниці дозволяють мати інформацію про важливу для інтерполяції «ступінь гладкості» даної функції $f(x)$. При цьому велику роль грають наступні теореми.

Теорема 5.4. Якщо $y = f(x)$ – поліном ст епеня п із ст аршим членом $a_o x^n$, т о для будь-якого $k \le n$ кінцева різниця $\Delta^k y$ є поліном від х ст упеня п - к із ст аршим членом: $n(n-1)(n-2)\ldots(n-k+1)a_o h^k x^{n-k}$, де h – крок т аблиці.

Дійсно, нехай $y = f(x) = a_o x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$.

Тоді згідно властивостям оператора Δ маємо:

$\Delta y = a_0 \Delta(x^n) + a_1 \Delta(x^{n-1}) + ... + a_{n-1} \Delta x$

Знайдемо $\Delta(x^n)$. Маємо:

$$\Delta(x^n) = (x+h)^n - x^n =$$

$$= x^n + nhx^{n-1} + \frac{n(n-1)}{2!}h^2x^{n-2} + ... + nh^{n-1}x + h^n - x^n.$$

І таким чином, $= nhx^{n-1} + \frac{n(n-1)}{2!}h^2x^{n-2} + ... + nh^{n-1}x + h^n$

З цих виразів отримаємо поліном, старший член якого має вид a_0nhx^{n-1}. Таким чином, *1-а* різниця полінома степеня n із старшим членом a_0 є поліном степеня $n-1$ із старшим членом a_0nhx^{n-1}. Продовжуючи подібні обчислення послідовно для різниць будь-якого порядку, можна переконатися в правильності теореми.

Наслідок 5.1. Для полінома $y = f(x)$ степені n кінцева різниця n-го порядку є постійна величина, рівна $\Delta^n y = n!a_0h^n$, а всі різниці вищого порядку, ніж n, тотожно рівні нулю.

На закінчення відзначимо залежність між кінцевими різницями і похідними функції, цікаву з погляду зв'язку дискретного і безперервного аналізу. Оскільки $f'(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} = \lim_{h\to 0}\frac{\Delta y}{h}$, то . $f'(x) \approx \frac{\Delta y}{h}$.

Легко показати також, що і взагалі $f^{(n)}(x) \approx \frac{\Delta^n y}{h^n}$, хоча похибка формули дуже швидко росте із збільшенням n.

5.2.6 Поняття про розділені різниці

Нехай вузли інтерполяції $x_0, x_1,.,., x_n$ не обов'язково рівновіддалені. В цьому випадку аналогічну кінцевим різницям роль грають так звані розділені різниці, або «підйоми» функції.

Розділеними різницями 1-го порядку називаються величини, що мають смисл середніх швидкостей зміни функції:

$[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$; $[x_1, x_2] = \frac{y_2 - y_1}{x_2 - x_1}$; і взагалі $[x_i, x_{i+1}] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$.

Розділеними різницями *2-го* порядку називаються величини

$[x_0, x_1, x_2] = \frac{[x_1, x_2] - [x_0, x_1]}{x_2 - x_0}$; $[x_1, x_2, x_3] = \frac{[x_2, x_3] - [x_0, x_1]}{x_3 - x_1}$ і т.д.

Вони, очевидно, пов'язані із зміною середньої швидкості зміни функції при переході від попереднього інтервалу (x_{i-1}, x_i) до наступного (x_i, x_{i+1}).

Розділені різниці k-го порядку визначаються через розділені різниці $(k-1)$-го порядку за допомогою рекурентного співвідношення

$$[x_i, x_{i+1},...x_{i+k}] = \frac{[x_{i+1},..., x_{i+k}] - [x_i,..., x_{i+k-1}]}{x_{i+k} - x_i}.$$

Властивості лінійності залишаються справедливими і для розділених різниць будь-яких порядків.

Надалі нам корисно мати вираз значення функції $f(x_k)$ в довільному вузлі інтерполяції x_k через значення функції $f(x_0)$ в початковому вузлі x_0 і початкові значення розділених різниць $[x_0, x_1]$, $[x_0, x_1, x_2]$, $[x_0, x_2, x_3]$,.. . Воно виводиться по індукції і має вигляд:

$$f(x_k) = f(x_0) + (x_k - x_0)\cdot[x_0, x_1] + (x_k - x_0)\cdot(x_k - x_1)\cdot[x_0, x_1, x_2] + ...$$
$$+(x_k - x_0)\cdot(x_k - x_1)\cdot...\cdot(x_k - x_{k-1})[x_0, x_1,..., x_k].$$

Якщо таблиця значень функції має постійний крок h, то по індукції легко встановити зв'язок між розділеними і кінцевими різницями одного і того ж порядку k.

$$[x_0, x_0, +h, x_0, +2h,..., x_0, +kh] = \frac{\Delta^k y_0}{k! h^k}$$

Для розділених різниць мають місце теореми, аналогічні теоремам про кінцеві різниці і відповідним наслідками з них. Сформулюємо найважливіше твердження.

Теорема 5.5. Якщо f(x) – поліном ступеня n, то розділені різниці n-го порядку є константами, незалежними від вузлів x_0 x_1,..., x_n і рівними коефіцієнту при старшому степені x в поліномі f(x). Всі розділені різниці більшого, ніж n, порядку рівні 0.

Ця теорема лежить в основі практичного правила, що дозволяє призначати ступінь інтерполяційного полінома у разі нерівновіддалених вузлів так, щоб вона співпадала з порядком практично постійних розділених різниць функції.

Для осмислення зв'язку дискретного і безперервного аналізу треба мати на увазі наступне. У інтервалі (x_0, x_k) існує така точка ξ, що: $[x_0, x_1, ..., x_k] = \frac{f^{(k)}(\xi)}{k!}$. Тому

$$\lim_{\substack{x_1 \to x_0 \\ x_2 \to x_0 \\ \cdots\cdots \\ x_k \to x_0}} [x_0, x_1, ..., x_k] = \frac{f^{(k)}(\xi)}{k!}$$

Звідси випливає, що для наближених «прикидок» можна використовувати співвідношення: $f^{(k)}(x) \approx k![x_0, x_1, ..., x_k]$.

5.2.7 Інтерполяційна формула Ньютона

Істотним недоліком поліномів Лагранжа, як було відмічено, є те, що вони часто значно утрудняють практичні обчислення. Інше уявлення, засноване на використанні розділених різниць, дає можливість для послідовного уточнення результатів інтерполяції і часто не вимагає попереднього знання степеня полінома.

Теорема 5.6. Інтерполяційний поліном $P_k(x)$ може бути записаний у вигляді, який називається формулою Ньютона

$P_k(x) = y_o + [x_o, x_1](x - x_0) + [x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots$

$+ [x_o x_1, ..., x_k](x - x_0)(x - x_1)...(x - x_{k-1})$.

Доведення. По-перше, ясно, що вказаний поліном має ступінь не вищий k. По-друге, він приймає значення функції у вузлах інтерполяції. Дійсно, $P(x_o) = y_o$, оскільки при $x = x_0$ всі члени полінома, починаючи з другого, перетворюються в нуль. При $x = x_1$ з таких же причин на підставі виразу

$$f(x_k) = f(x_0) + (x_k - x_0) \cdot [x_0, x_1] + (x_k - x_0) \cdot (x_k - x_1) \cdot [x_0, x_1, x_2] + ...$$

$$+ (x_k - x_0) \cdot (x_k - x_1) \cdot ... \cdot (x_k - x_{k-1})[x_0, x_1, ..., x_k]$$

отримаємо $P_k(x_1) = y_0 + [x_0, x_1](x - x_0)$, що співпадає з y_1. Аналогічні міркування переконують, що і для будь-якого $i = 2,3...,n$, $P_k(x_i) = f(x_i) = y_i$.

Формули Лагранжа і Ньютона дають лише різні форми запису одного і того ж інтерполяційного полінома. Проте формула Ньютона зручна тим, що при додаванні до вузлів $x_0, x_1, \ldots, x_k$ нового вузла x_{k+1} всі раніше знайдені

члени залишаються без зміни і в поліномі лише додається один додатковий член вигляду $[x_o\ x_1,...,x_{k+1}](x—x_0)(x—x_1)...(x — x_k)$. Це дозволяє, послідовно додаючи поодинці додаткові вузли, поступово нарощувати точність результату інтерполяції.

Оскільки табличні різниці швидко зменшуються із збільшенням порядку, то найближчі до даної точки x'' вузли інтерполяції дадуть основний внесок в шукану величину, а інші даватимуть лише невеликі поправки. В цьому випадку легше уникнути прорахунків і встановити, на якій різниці слід закінчити обчислення.

Через тотожність поліномів Лагранжа і Ньютона залишковий член $R_k(x)$ у них однаковий. З погляду зв'язку між аналізом дискретним і безперервним цікаво відзначити аналогію представлення функції $f(x)=P_k(x)+ R_k(x)$, де права частина визначається формулами Лагранжа і Ньютона, з відомою формулою Тейлора. Добуток різниць $(x - x_0)(x - x_1) ...(x - x_k)$ є узагальненням степеня бінома, а розділені різниці виступають як узагальнені похідні. Якщо вузли $x_0, x_2, . . ., x_k$ стягуються в одну точку, наприклад x_0, то, через наявність границі

$$\lim_{\substack{x_1 \to x_0 \\ x_2 \to x_0 \\ \cdots\cdots \\ x_k \to x_0}} [x_0, x_1, \ldots, x_k] = \frac{f^{(k)}(\xi)}{k!}$$

формула Ньютона перетворюється у формулу Тейлора. Таким чином, формулу Ньютона можна розглядати як узагальнення формули Тейлора на випадок дискретного аналізу.

Реалізація методу інтерполяції за формулою Ньютона у вигляді окремого модуля *Python* представлена на рис. 5.15.

```
# -*- coding: cp1251 -*-

## module newtonPoly

''' p = evalPoly(a,xData,x).

    Визначає значення формули Ньютона p в точці x. Вектор коефіцієнтів

    'a' можна розрахувати за допомогою функції 'coeffts'.
```

```
    a = coeffts(xData,yData).
    Визначає коефіцієнти полінома Ньютона.
"""
def evalPoly(a,xData,x):
    n = len(xData) - 1  # Степінь полінома
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xData[n-k])*p
    return p


def coeffts(xData,yData):
    m = len(xData)  # Кількість точок даних
    a = yData.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xData[k:m] - xData[k-1])
    return a
```

Рис. 5.15. Реалізація алгоритму методу Ньютона

Розглянемо використання створеного модуля (рис. 5.15) на прикладі.

Приклад 5.3. Функція $f(x) = 4{,}8cos\frac{\pi x}{20}$ задана таблично.

x	0,15	2,30	3,15	4,85	6,25	7,95
y	4,79867	4,49013	4,2243	3,47313	2,66674	1,51909

Знайти за допомогою методу Ньютона значення функції в точках: $x = 0; 0{,}5; 1{,}0; \ldots; 8{,}0$ і порівняти результати з точними даними $y_i = f(x_i)$. Програма (написана з використанням раніше розробленого модуля), яка вирішує поставлене завдання, представлена на рис. 5.16.

```
# -*- coding: cp1251 -*-
```

```
from numpy import array,arange
from math import pi,cos
from newtonPoly import *

xData = array([0.15,2.3,3.15,4.85,6.25,7.95])
yData = array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
a = coeffts(xData,yData)
print " x    yInterp   yExact"
print "----------------------"
for x in arange(0.0,8.1,0.5):
    y = evalPoly(a,xData,x)
    yExact = 4.8*cos(pi*x/20.0)
    print "%3.1f %9.5f %9.5f"% (x,y,yExact)
raw_input("\nНатиснути ввід для виходу")
```

Рис. 5.16. Приклад застосування модуля

Результат роботи програми 5.16 представлено на рис. 5.17.

```
>>>
 x    yInterp   yExact
----------------------
0.0   4.80003   4.80000
0.5   4.78518   4.78520
1.0   4.74088   4.74090
1.5   4.66736   4.66738
2.0   4.56507   4.56507
2.5   4.43462   4.43462
3.0   4.27683   4.27683
3.5   4.09267   4.09267
```

4.0 3.88327 3.88328

4.5 3.64994 3.64995

5.0 3.39411 3.39411

5.5 3.11735 3.11735

6.0 2.82137 2.82137

6.5 2.50799 2.50799

7.0 2.17915 2.17915

7.5 1.83687 1.83688

8.0 1.48329 1.48328

Натиснути ввід для виходу

Рис. 5.17. Результат виконання програми

Як бачимо, результати майже не відрізняються від точних.

5.2.8 Інтерполяція для рівновіддалених вузлів

Розглянемо важливий окремий випадок, коли $x_{i+1} - x_i = h = const$, $i = 0,1...,k-1$. Якщо ввести нову змінну, $x = x0 + th$, то при будь-якому h вузли інтерполяції завжди приймають стандартні значення відповідно $t_0 = 0$, $t_1 = 1$, $t_2 = 2...$, $t_k = k$. Змінна t має сенс числа кроків h від x_0 до x. Це дозволяє уніфікувати відповідні формули інтерполяції. Так, коефіцієнти Лагранжа вдається представити у вигляді:

$$L_i(t) = (-1)^{k-1} C_k^i \frac{t(t-1)(t-2)...(t-k)}{(t-i)n}$$

який не залежить ні від $f(x)$, ні від h. Це дозволяє раз і назавжди скласти таблиці коефіцієнтів Лагранжа $L_i(t)$. Формула для інтерполяції поліномами Лагранжа приймає вигляд: $P_k(t) = \sum_{i=1}^{k} y_i L_i(t)$.

Формула Ньютона також спрощується до вигляду:

$$P_k(x) = y_0 + \frac{\Delta y_0}{1!h}(x - x_0) + \frac{\Delta^2 y_0}{2!h^2}(x - x_0)(x - x_1) + ...$$

$$+\frac{\Delta^k y_0}{k!h^k}(x-x_0)(x-x_1)...(x-x_{k-1}).$$

Приклад 5.4. Нехай є таблиця значень функції ***y=sin x.***

x_i **град**	y_i
10	0,17365
20	0,34202
30	0,50000
40	0,64279
50	0,76604
60	0,86603

Потрібно знайти ***y*** при ***x=23°*** методом розділених різниць. За допомогою початкових даних складемо таблицю різниць.

x_i **град**	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$	$\Delta^4 y_i$	$\Delta^5 y_i$
10	**0,17365**	**-**				
		0,16837				
20	**0,34202**		**0,01039**			
		0,15798		**0,00480**		
30	**0,50000**		**0,01519**		**0,00045**	
		0,14279		**0,00435**		**0,00018**
40	**0,64279**		**0,01954**		**0,00063**	
		0,12325		**0,00372**		
50	**0,76604**		**0,02326**			
		0,09999				
60	**0,86603**					

За x_0 можна прийняти будь-яке значення ***x:*** наприклад ***x=20°.*** Необхідні різниці стоять на діагоналі, що йде від x_0 вниз. Число використовуваних

різниць вищих порядків може бути будь-яким, але чим воно більше, тим вище точність.

Одне з достоїнств даного методу полягає в тому, що він дозволяє уточнювати результат, використовуючи додаткові різниці, причому немає необхідності починати обчислення спочатку. Тому у випадку, якщо невідомо, скільки членів слід узяти, їх число можна збільшувати до тих пір, поки їх внесок не нехтуватиме малим. В даному випадку h=10°. Використовуючи тільки першу різницю, знайдемо:

$$y(23) = y + \frac{\Delta y_0}{h}(23 - x_0) = 0{,}34202 + \frac{0{,}15798}{10} \cdot 3 = 0{,}38941.$$

Ввівши додатково другу різницю, отримаємо:

$$y(23) = 0{,}38941 + \frac{\Delta^2 y_0}{2h^2}(23 - x_0)(23 - x_1) = 0{,}39100.$$

Нарешті, за допомогою третьої різниці знайдемо:

$$y(23) = 0{,}39100 + \frac{\Delta^3 y_0}{6h^3}(23 - x_0)(23 - x_1)(23 - x_2) = 0{,}39074$$

Це значення y дуже близько до табличного (точного) значення, рівного *0,39073*.

5.3. Багатоінтервальна інтерполяція

5.3.1. Властивості багатоінтервальної інтерполяції

Як вказувалося в п.5.2.2, степінь інтерполяційного полінома, побудованого за формулою Лагранжа, залежить від кількості табличних точок, а саме: якщо таблиця складається з $(n+1)$ точок, то степінь інтерполяційного полінома дорівнює n. Звідси витікає, що при великій кількості точок потрібно використовувати поліном високого степеня, що призведе до збільшення кількості необхідних арифметичних операцій, зростання похибки округлення та часу обчислень. З цими труднощами стикаються при інтерполяції на великих відрізках: при великих відстанях між вузлами точність дуже мала, а в разі зменшення цих відстаней збільшується кількість вузлів та відповідно степінь полінома.

У таких випадках доцільно проводити інтерполяцію функції за допомогою кускових поліномів невисокого степеня. Нехай $a = \gamma_0 < \gamma_1 < ... \gamma_m < \gamma_{m+1} = b$ – деяке розбиття відрізка ***[a,b]*** на підінтервали, кожний з яких містить у собі декілька вузлових точок (рис. 5.18). Всі точки розбиття нехай збігаються з деякими вузлами таблиці. Побудуємо на кожному відрізку $l_k = [\gamma_k, Y_{k+1}]$ *(k=0,1,...m)* за вузловими точками, які в ньому містяться, інтерполяційний многочлен $g_k(x)$. Оскільки на кожному підінтервалі кількість вузлів невелика, то степінь многочлена буде невисокою.

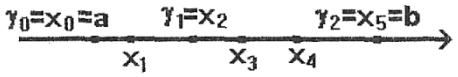


Рис. 5.18. Розбиття відрізка при кусковій інтерполяції

Функцію ***g(x)***, яка є об'єднанням цих поліномів, називають кусково-поліноміальною і саме її використовують для обчислення наближених значень функції ***f(x)*** у проміжних точках. Такий підхід називають кусковою (локальною, багатоінтервальною) інтерполяцією. Загальний алгоритм у цьому випадку можна представити у вигляді:

1. Визначити, якому підінтервалу $[\gamma_k, \gamma_{k+1}]$ належить робоча точка ***x***.
2. За вузловими точками, які належать до цього інтервалу (включаючи точки стику підінтервалів), за допомогою якої-небудь інтерполяційної формули обчислити $y = g_k(x) \approx f(x)$.

Багатоінторвальна інтерполяція має такі властивості:

1. Степінь інтерполюючого многочлена не залежить від кількості вузлів. Дійсно, із зростанням кількості вузлів можна збільшити кількість ***m*** точок розбивання γ_k, тобто кількість підінтервалів, з тим, щоб кількість вузлових точок x_k на кожному підінтервалі не збільшувалась.

2. При незмінному відрізку ***[a,b]*** помилка інтерполяції із зростанням кількості вузлів прямує до нуля, що обумовлено зменшенням відстані між точками.

3. Час обчислень невеликий внаслідок низького степеня полінома. Відмітимо, що хоча функція $g_k(x)$ є неперервною, в точках γ_k стикання підінтервалів, як правило, має розрив вже перша її похідна. Це проілюстровано прикладом у п.5.3.3. Цей недолік багатоінтервальної інтерполяції зникає при використанні сплайнів.

Найпростішим видом кускової інтерполяції є кусково-лінійна інтерполяція.

5.3.2. Кусково-лінійна інтерполяція

Нехай задано таблицю $\{x_i,y_i\}$ значень невідомої функції $y=f(x)$, де $i=0,1,...,n$. З'єднавши сусідні точки $\{x_i,y_i\}$ прямолінійними відрізками, одержуємо ламану лінію, яка і апроксимує функцію $y=f(x)$ (рис. 5.19). Таким чином, при кусково-лінійній інтерполяції точки розбиття γ_k збігаються з вузлами x_i. Функції $g_k(x)$ для кожного з n відрізків $\{x_i,y_i\}$ становлять поліноми першого степеня і можуть бути одержані з рівняння прямої, яка проходить через точки (x_k,y_k) та (x_{k+1},y_{k+1}) (див. п.5.2.1)

$$g_k = a_0 + a_1 x,\ a_1 = \frac{y_{k+1} - y_k}{x_{k+1} - x_k};\ a_0 = y_k - x_k a_1 \qquad (5.24)$$

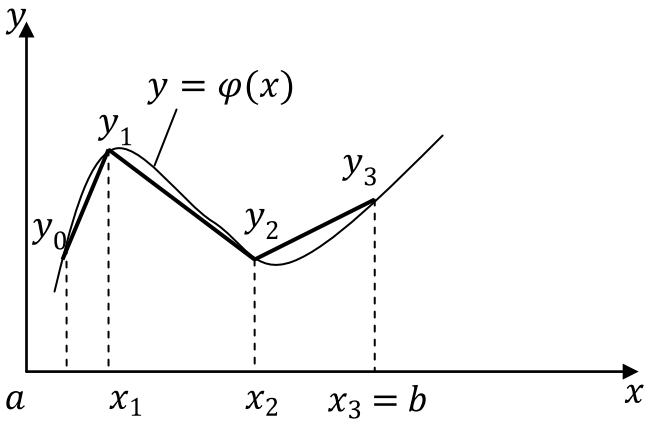


Рис. 5.19. Кусково-лінійна інтерполяція

При використанні кусково-лінійної інтерполяції, як і в загальному випадку, необхідно визначити спочатку, якому відрізку $[x_k,x_{k+1}]$ належить робоча точка x, а потім за допомогою формул (5.24) обчислити $y = g_k(x) \approx f(x)$.

При програмуванні цього алгоритму слід урахувати випадок, коли x знаходиться за межами таблиці, тобто $x < x_0$ або $x > x_n$. Залежно від постановки задачі більш вищого рівня тут можуть бути передбачені:

- виведення діагностичного повідомлення про некоректне значення x;
- продовження ламаної поза відрізком $[a,b]$ горизонтальними прямими $y = y_0$ та $y = y_n$;
- продовження першого і останнього відрізків ламаної при $x<a$ або $x>b$.

Наведемо приклад програми, де передбачений третій варіант, тобто при $x<a$ значення y обчислюють за тією самою формулою, що і для $x \in [x_0,x_1]$, а при $x>b$ за тією самою формулою, що і при $x \in [x_{n-1},x_n]$.

У цій програмі (рис. 5.20) початковими даними є:

- таблиця значень функції в вигляді масивів X та Y;
- робоча точка XR.

Результатом обчислень є змінна YR – наближене значення функції у робочій точці.

```
…
for i in xrange(2,N+1):
    if ( (XR and X[i-1]) and (XR < X[i] ):
        A1=( Y[i]-Y[i-1])/(X[i]-X[i-1])
        A0=Y[i-1]-X[i-1]*A1
if (XR<X[1]) :
    A1=( Y[2]-Y[1])/(X[2]-X[1])
    A0=Y[1]-X[1]*A1
if (XR>X[N]):
    A1=( Y[N] -Y[N -1])/(X[N] - X[N -1])
    A0=Y[N-1]-X[N-1]*A1
YR=A0+A1*XR;
```

...

Рис. 5.20. Фрагмент реалізації алгоритму

5.3.3. Кусково-нелінійна інтерполяція

Як зазначалося в п.5.3.1, весь відрізок ***[a,b]***, на якому задана таблиця значень функції ***y=f(x)***, може бути розбитий на часткові відрізки, кожний з яких має невелику кількість точок x_i. На кожному частковому відрізку можна побудувати свій інтерполяційний поліном $g_k(x)$ невисокого степеня з використанням цих поліномів для находження наближених значень функції ***y=f(x)*** у проміжних точках, що не збігаються в загальному випадку з табличними точками x_i. У п.5.3.2 кожен частковий відрізок містить тільки дві точки і для апроксимації використовувались поліноми першого степеня, яким відповідають прямолінійні відрізки.

Але часто трапляється, що потрібно збільшити точність апроксимації. Це можна забезпечити використанням поліномів хоча б другого степеня, тобто замість відрізків прямих з'єднати табличні точки відрізками парабол. Якщо кожен частковий відрізок має тільки дві точки, то для однозначного визначення параболи, яка проходить через них, необхідно накласти, окрім двох умов інтерполяції, ще одну – допоміжну. Характер цієї умови та особливості одержаної кусково-нелінійної функції будуть розглянути у п.5.3.4. Якщо ж ніяких додаткових умов не накладати, тоді для інтерполяції функції кусковим поліномом другого степеня в кожен частковий відрізок слід об'єднувати три сусідні точки (дві крайні та одну внутрішню). Аналогічно роблять при побудові інтерполюючої кусково-нелінійної функції більш вищого степеня: в частковий відрізок включають чотири точки при використанні полінома третього степеня і т.д.

Для ілюстрації такого підходу розглянемо приклад.

Приклад 5.5. Невідома функція ***y=f(x)*** задана на відрізку ***[0,1]*** таблицею з семи точок:

X	0	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{5}{6}$	1
Y	1	3	2	1	0	2	1

Необхідно побудувати кусково-нелінійну функцію другого степеня, що інтерполює функцію ***y=f(x)***.

Для розв'язування задачі згрупуємо табличні точки в часткові відрізки по три точки в кожному. На кожному з часткових відрізків [0, $\frac{1}{3}$], [$\frac{1}{3}$, $\frac{2}{3}$], [$\frac{2}{3}$, 1] за допомогою формули Лагранжа побудуємо інтерполяційний поліном другого степеня.

Для першого відрізка l_1:

$$g_1(x) = L_2(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

На другому відрізку $\boldsymbol{l_2=[\frac{1}{3}, \frac{2}{3}]}$ скористаємося цією самою функцією, але як величину x_0, x_1, x_2 тепер будемо використовувати чергові точки таблиці $x_0=\frac{1}{3}$, $x_1=\frac{1}{2}$, $x_2=\frac{2}{3}$, $y_0=1$, $y_1=1$, $y_2=0$:

$$g_2(x) = L_2(x) = 2\frac{\left(x-\frac{1}{2}\right)\left(x-\frac{2}{3}\right)}{\left(\frac{1}{3}-\frac{1}{2}\right)\left(\frac{1}{3}-\frac{2}{3}\right)} + 1\frac{\left(x-\frac{1}{3}\right)\left(x-\frac{2}{3}\right)}{\left(\frac{1}{2}-\frac{1}{3}\right)\left(\frac{1}{2}-\frac{2}{3}\right)} + 0\frac{\left(x-\frac{1}{3}\right)\left(x-\frac{1}{2}\right)}{\left(\frac{2}{3}-\frac{1}{3}\right)\left(\frac{2}{3}-\frac{1}{2}\right)} = -6x+4$$

Одержимо поліном першого, а не другого степеня. Це зв'язано з тим, що на відрізку $\boldsymbol{l_2}$ три точки лежать на одній прямій. На третьому частковому відрізку $\boldsymbol{l_3 = [\frac{2}{3}, 1]}$ як вузли інтерполяції будуть використані такі три точки:

$x_0=\frac{2}{3}$, $x_1=\frac{5}{6}$, $x_2=1$, $y_0=1$, $y_1=2$, $y_2=1$. Виконавши аналогічні дії для відрізка $\mathbf{l_3}$, одержимо в результаті кусково-параболічну функцію (рис. 5.21):

$$g(x)=\begin{cases} -54x^2+21x+1 & 0<x<\frac{1}{3} \\ -6x+4 & \frac{1}{3}<x<\frac{2}{3} \\ -54x^2+93x-38 & \frac{2}{3}<x<1 \end{cases}$$

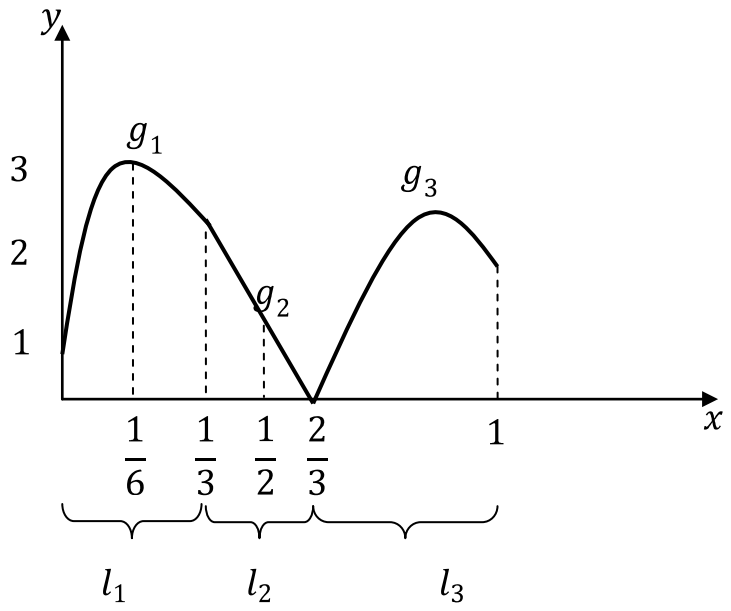


Рис. 5.21. Кусково-параболічна інтерполяція

Виникає природне запитання про точність апроксимації функції ***f(x)*** функцією ***g(x)***, що одержана. Як відомо (див.п.5.2.2), помилку апроксимації поліномом Лагранжа при рівновіддалених вузлах визначають співвідношенням

$$R=\left|f(x)-L_n(x)\right|\geq M\cdot h_{n+1},\ \ де\ M=\max_{x_0\leq z\leq x_1}\left|f^{(n+1)}(z)\right|$$

У цьому випадку ***n=2, h=***$\frac{1}{6}$, тому

$$R=\left|f(x)-g(x)\leq Mh^3=\frac{M}{6^3}\right|,\ \ де\ M=\max_{0\leq z\leq 1}\left|f'''(z)\right|$$

Однак ця оцінка не дозволяє одержати конкретне значення, що обмежує ***R***, оскільки величина ***M*** невідома. Але за її допомогою можна одержати

відповідь на запитання, як впливає на помилку величина кроку. Якщо зменшимо крок вдвічі, тобто розіб'ємо відрізок ***[0,1]*** не на три, а на шість підінтервалів, то помилка апроксимації зміниться:

$$R' \le \left(\frac{h}{2}\right)^3 M = \frac{h^3 M}{8} = \frac{R}{8}$$

Таким чином, при зменшенні кроку вдвічі помилка при параболічній інтерполяції зменшиться в 8 разів.

З рис. 5.21 видно, що в точках стикання поліномів $g_k(x)$ функція $g(x)$ не є диференційованою. В ряді випадків це може бути суттєвим і тоді використовують спеціальну форму кускової апроксимації – сплайни, які розглядаються далі.

5.3.4. Параболічні сплайни

Сплайном називають кусково-задану функцію, яка разом з декількома похідними неперервна на всьому відрізку ***[a,b]***, а на кожному частковому відрізку $[x_i, x_{i+1}]$ окремо є деяким алгебраїчним многочленом.

Максимальний за всіма частковими відрізками степінь многочленів називають *степенем сплайна*, а різницю між степенем сплайна і порядком вищої неперервної на ***[a,b]*** похідної – *дефектом сплайна*. Значення першої похідної сплайна у вузлі таблиці називають його *нахилом* в цьому вузлі. Назва такого роду кривих походить від англійського слова *spline* – гнучка лінійка, що використовується креслярами для проведення ліній. Це зв'язано з тим, що якщо на площині відмітити табличні точки і розмістити гнучку лінійку ребром до площини так, щоб вона була зафіксована в цих точках, то лінійка набере форми, що мінімізує її потенційну енергію. Одержана крива математично є природним кубічним сплайном, який розглядається в п.5.3.5. Розглянемо побудову простих сплайнів – сплайнів другого степеня з дефектом, що дорівнює одиниці.

В п.5.3.3 при проведенні кусково-параболічної інтерполяції ми розбивали відрізок ***[a,b]***, на якому задана таблиця $\{x_i, y_i\}$, на часткові відрізки по три вузли в кожному. Ці три вузли однозначно визначають параболу.

Набір таких парабол і є інтерполюючою функцією. Оскільки правий вузол чергової трійки збігався з лівим вузлом наступної трійки вузлів, то внаслідок вимоги інтерполяції набір парабол у цілому виявляється неперервною функцією. Однак, як видно з рис. 5.21, параболи не переходять плавно одна в одну, а утворюють у точках стикання злом, тобто кускова функція не являє собою диференційовану в цих точках. На кожному частковому відрізку три умови інтерполяції $(g_k(x_i)=y_i)$ повністю вичерпують можливості накладання обмежень на параболу, оскільки вона характеризується своїми трьома коефіцієнтами. Тому, якщо потрібно побудувати диференційовану всюди кускову функцію – сплайн, то слід змінити підхід до кускової інтерполяції. Розглянемо на прикладі, як можна це зробити.

Нехай невідома функція $y=f(x)$ задана в чотирьох точках x_1,x_2,x_3,x_4 своїми значеннями y_1,y_2,y_3,y_4 (рис. 5.22). Як часткові відрізки виберемо $l_i=[x_i,x_{i+1}]$, $i=1,2,3$

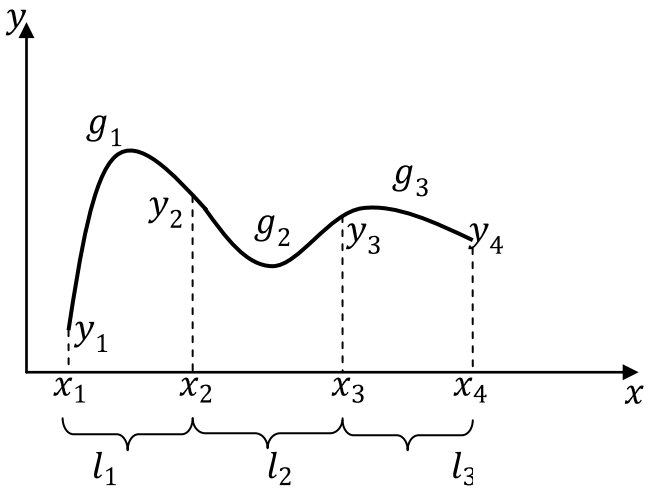


Рис. 5.22. Параболічний сплайн

На кожному частковому відрізку маємо тільки два вузли і для визначення трьох коефіцієнтів параболи можна скласти тільки два рівняння. Тому у нас з'явиться можливість сформулювати додаткову, третю вимогу до параболи – *плавність переходу* її в чергову параболу. На кожному відрізку l_i відповідну параболу шукатимемо у вигляді

$$g_i(x) = a_{i2}x^2 + a_{i1}x + a_{i0}, \qquad l = 1,2,3 \tag{5.25}$$

У цьому випадку сплайн, як сукупність трьох парабол, визначається дев'ятьма коефіцієнтами a_{ij}, тому необхідно скласти дев'ять рівнянь. Перші шість одержуємо з умов проходження кожної параболи через табличні точки:

$$\begin{aligned} &g_1(x_1) = y_1,\ \ g_1(x_2) = y_2, \\ &g_2(x_2) = y_2,\ \ g_2(x_3) = y_3, \\ &g_3(x_3) = y_3,\ \ g_3(x_4) = y_4 \end{aligned} \qquad (5.26)$$

Щоб кускова функція була диференційована в точках стикання часткових відрізків, повинні виконуватися також такі дві умови:

$$\begin{aligned} &g_1^l(x_2) = g_2^l(x_2), \\ &g_2^l(x_3) = g_3^l(x_3) \end{aligned} \qquad (5.27)$$

Вісім рівнянь (5.26) та (5.27) визначають множину сплайнів, які проходять через табличні точки та диференційовані в точках стикання. Для конкретизації сплайна вводиться ще одна, додаткова умова. Найчастіше вказують нахил сплайна в якому-небудь вузлі, наприклад,

$$g_1^l(x_1) = d \qquad (5.28)$$

де d – відома величина. Розв'язавши одержану систему з дев'яти лінійних алгебраїчних рівнянь з дев’ятьма невідомими a_{ij}, одержуємо сплайн

$g(x) = g_i(x) \quad при \quad x \in l_i$.

У загальному випадку, коли функція $y=f(x)$ задана таблицею в n точках, сплайн будують аналогічно. Множину вузлів розбиваємо на $(n-1)$ частковий відрізок $l_i = [x_i, x_{i+1}],\ i=1,...n-1$. Оскільки кількість парабол також дорівнює $(n-1)$, то для визначення $3(n-1)$ невідомих необхідно скласти систему з $3(n-1)$ рівнянь. Записавши для кожної параболи дві умови інтерполяції, одержуємо $2(n-1)$ рівнянь:

$$g_i(x_i) = y_i,\ \ g_1(x_{i+1}) = y_{i+1}, l = 1,..., n-1 \qquad (5.29)$$

З n вузлів x_i внутрішніми, тобто точками стикання, є $(n-2)$ вузли. Записавши для кожного з них умову диференційованості сплайну, одержимо ще $n-2$ рівнянь:

$$g_i^l(x_{i+1}) = g_{i+1}^l(x_{i+1}),\ l = 1,..., n-2 \qquad (5.30)$$

В результаті маємо ***2(n-1)+(n-2)=3n-4*** рівнянь з ***3n-3*** невідомими. Доповнивши їх ще одним рівнянням (5.28), отримаємо шукану СЛАР, яка визначає сплайн.

Вектор невідомих цієї системи складається з коефіцієнтів квадратних тричленів, що утворюють сплайн. Зважаючи на те, що три перших елементи цього вектора є коефіцієнтами першого тричлена, наступні три – другого і т.д., тоді для докладного запису рівнянь зручно використати ступеневу форму. Перші ***2(n-1)*** рівнянь (5.29), що забезпечують вимоги інтерполяції, мають вигляд:

$$\begin{aligned}
a_{12}x_1^{\,2} + a_{11}x_1 + a_{10} &= y_1 \\
a_{12}x_2^{\,2} + a_{11}x_2 + a_{10} &= y_2
\end{aligned}$$

$$\begin{aligned}
a_{22}x_2^{\,2} + a_{21}x_1 + a_{20} &= y_2 \\
a_{22}x_3^{\,2} + a_{21}x_3 + a_{20} &= y_3 \\
\ldots & \ldots \\
a_{n-1,2}x_{n-1}^{\,2} + a_{n-1,1}x_{n-1} + a_{n-1,0} &= y_{n-1} \\
a_{n-1,2}x_n^{\,2} + a_{n-1,1}x_n + a_{n-1,0} &= y_n
\end{aligned}$$

Зважаючи на те, що $g_i^l(x) = 2a_{i2}x + a_{i1}$, рівняння (5.30) набирають вигляду

$$2a_{i2}x_{i+1} + a_{i1} = 2a_{i+1,2}x_{i+1} + a_{i+1,1}$$

або

$$2a_{i2}x_{i+1} + a_{i1} - 2a_{i+1,2}x_{i+1} - a_{i+1,1} = 0.$$

Тому наступні рівняння формуючої системи, що відображають вимоги диференційованості сплайну, мають вигляд:

$$\begin{aligned}
2a_{12}x_2 + a_{11} - 2a_{22}x_2 - a_{21} &= 0 \\
2a_{22}x_3 + a_{21} - 2a_{23}x_3 + a_{31} &= 0 \\
\ldots & \ldots \\
2a_{n-2,2}x_{n-1}^{\,2} + a_{n-2,1} - 2a_{n-1,2}x_{n-1} - a_{n-1,1} &= 0 \\
2a_{12}x_1 + a_{11} &= d
\end{aligned}$$

Більш зручною для програмування є матрична форма СЛАР:

$$\begin{bmatrix} x_1^2 & x_1 & 1 & & & & & & & \\ x_2^2 & x_2 & 1 & & & & & & & \\ & & & x_3^2 & x_2 & 1 & & & & \\ & & & x_3^2 & x_3 & 1 & & & & \\ & & & & \ldots & & & & & \\ & & & & & & x_{n-1}^2 & x_{n-1} & 1 & \\ & & & & & & x_n^2 & x_n & 1 & \\ 2x_2 & 1 & 0 & -2x_2 & -1 & & & & & \\ & & & 2x_3 & 1 & 0 & -2x_3 & -1 & & \\ & & & & & \ldots & & & & \\ & & & & & & 2x_{n-1} & 1 & 0 & -2x_{n-1} & -1 \\ 2x_1 & 1 & & & & & & & & \end{bmatrix} \begin{bmatrix} a_{12} \\ a_{11} \\ a_{10} \\ a_{22} \\ a_{21} \\ a_{20} \\ K \\ a_{n-1,2} \\ a_{n-1,1} \\ a_{n-1,0} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_2 \\ y_3 \\ \ldots \\ y_{n-1} \\ y_n \\ 0 \\ 0 \\ \ldots \\ 0 \\ d \end{bmatrix} \quad (5.31)$$

Незаповнені позиції матриці дорівнюють нулю. Наведемо програму на мові *Python* для побудови та використання параболічного сплайну.

Приклад 5.6. Дано таблицю значень функції ***y=f(x)*** в 10 точках. Скласти програму побудови параболічного сплайну та обчислення за його допомогою наближених значень функції в чотирьох проміжних точках

Розв'язок. Цю програму доцільно реалізовувати у вигляді трьох функцій.

Функція **FORM** одержує на вході таблицю значень функції ***y=f(x)*** у вигляді масивів ***X*** та ***Y*** і формує систему рівнянь (5.31). Результат її роботи - матриця ***A*** та вектор правої частини цієї системи. Функція **GAUSS**, одержавши на вхід масиви ***A*** та ***B***, розв'язує СЛАР та формує вектор коефіцієнтів сплайну у вигляді масиву ***AS***. Функція **PRES** виконує обчислення ***g(z$_i$)*** та друкує результати. Зважаючи на те, що для обчислення значень сплайну зручніше користуватися двовимірним, а не одновимірним масивом, на початку цього модуля масив ***AS*** перетворюється в масив ***SP*** такої структури (рис. 5.23):

SP =

a_{12}	a_{11}	a_{10}	- коефіцієнти 1-ої параболи
a_{22}	a_{21}	a_{20}	
	…		…
a_{92}	a_{91}	a_{90}	- коефіцієнти 9-ої параболи

Рис. 5.23. Структура масиву

Реалізація програми представлена на рис. 5.24.

```
def FORM(X,Y,A,B):
    np.zeros((N,N))
    k,m=0,0
    for i in xrange(9):
        R=X[i]
        A[k,m]=R**2
        A[k,m+1]=R
        A[k,m+2]=1
        B[k]=Y[i+1]
        k+=1
        R=X[i+1]
        A[k,m]=R**2
        A[k,m+1]=R
        A[k,m+2]=1
        B[k]=Y[i+1]
        k+=1
        m+=3
    m=0
    for i in xrange(8):
        R=X[i+1]
        A[k,m]=2*R
        A[k,m+1]=1
        A[k,m+3]=-2*R
        A[k,m+4]=-1
        B[k]=0
        k+=1
```

```
            m+=3
        A[k,0]=2*X[0]
        A[k,1]=1
        B[k]=0
        return A,B


def GAUSS(A,B,AS,N=27):
    for k in xrange(N-1):
        for i in xrange(k+1,N):
            R=A[i,k]/A[k,k]
            for j in xrange(k,N):
                A[i,j]=A[i,j]-A[k,j]*R
            B[i]-=B[k]*R
    AS[2]=B[2]/A[2,2]
    for i in xrange(N-2,-1,-1):
        s=0
        for j in xrange(i+1,N):
            s+=A[i,j]*X[j]
        AS[i]=(B[i]-s)/A[i,i]
    return AS


def PRES(AS,Z,X):
    SP=np.reshape(AS,(9,3));print SP
    for i in xrange(3):
        R=Z[i]
        for j in xrange(9):
            if (R>=X[j]) and (R<=X[j+1]):
```

```
        G=SP[j,0]*R**2+SP[j,1]*R+SP[j,2]
    print "apr.=%.3f -- %.3f" %(R,G)
```

Рис. 5.24. Реалізація фрагменту програми мовою *Python*

5.3.5. Кубічні сплайни

При апроксимації розв'язків диференціальних рівнянь, а також в інших випадках потрібно, щоб апроксимуюча функція була принаймні двічі безперервно диференційована. Досягти цього за допомогою квадратичного сплайну, розглянутого в п.5.3.4, в загальному випадку неможливо. Тому розглянемо кусково-кубічний поліном **S(X)** (кубічний сплайн), який має такі властивості:

- двічі безперервно диференційований;
- на кожному частковому відрізку з кубічним поліномом.

Існує декілька способів побудови кубічних сплайнів. Розглянемо спочатку спосіб, коли на кожному частковому відрізку $l_i=[x_i,x_{i+1}]$ сплайн записують у вигляді

$$S(x) = S_i(x) = a_{i3}x^3 + a_{i2}x^2 + a_{i1}x + a_{i0}, \quad x \in l_i \qquad (5.32)$$

Нехай маємо таблицю значень функції $y=f(x)$ в n точках x_j, $i=1,2...,n$. Вони розбивають відрізок $[a,b]$ $(x_1 = a, x_n = b)$ на $n-1$ часткових відрізки l_i $i=1,...,n-1$. На кожному з них ми повинні побудувати свій кубічний поліном $S_i(x)$, $i=1,.,.,n-1$. Оскільки поліном $S_i(x)$ визначається чотирма коефіцієнтами a_{ij} $(j=3,2,1,0)$, то в цілому необхідно знайти $4(n-1)$ чисел a_{ij}. Для одержання відповідної системи рівнянь, яка визначає a_{ij}, сформулюємо умови, яким повинні задовольняти $S_i(x)$.

Оскільки будуватимемо інтерполяційний сплайн, то кожен поліном $S_i(x)$ повинен проходити через табличні точки свого часткового відрізка l_i. Ці умови інтерполяції дають перші $2(n-1)$ рівнянь:

$$S_i(x_i) = y_i,\ S_i(x_{+1i}) = y_{i+1},\ i = 1,2,...,n-1 \qquad (5.33)$$

Вимога диференційованості сплайну в точках сполучення часткових відрізків (таких точок $n-2$) породжує ще $2(n-2)$ рівнянь:

$$S_i^l(x_{i+1}) = S_{i+1}^l(x_{i+1}),\ \ S_i^{ll}(x_{i+1}) = S_{i+1}^{ll}(x_{i+1}),\ \ i = 1,2,...n-2 \qquad (5.34)$$

Всього одержали (***4n-6***) рівнянь замість ***4n-4***. Дві умови, які залишились (їх називають крайовими), можемо вибрати по-різному.

Якщо вимагати, щоб кривизна сплайну на кінцях відрізка дорівнювала нулю, тобто

$$S_1^{ll}(x_1) = 0,\ S_{n-1}^{ll}(x_n) = 0, \qquad (5.35)$$

то одержаний сплайн називають природним кубічним сплайном.

Розв'язавши одержану СЛАР (5.33)-(5.35), можна знайти коефіцієнти сплайну. Однак матриця цієї системи має велику розмірність, хоча і є розрідженою.

Розглянемо другий спосіб побудови кубічного сплайну, в якому треба розв'язати СЛАР розмірності лише (***n-2***) відносно других похідних сплайну у внутрішніх вузлах таблиці.

Кубічні поліноми, що утворюють сплайн, шукатимемо у вигляді

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3,\ x \in [x_i, x_{i+1}], i = 1,2,...n-1 \quad (5.36)$$

Похідні сплайну:

$$S_i^l(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2,\ \ S_i^{ll}(x) = 2c_i + 6d_i(x - x_i).$$

Для знаходження коефіцієнтів a_i, b_i, c_i, d_i, запишемо, як і в попередньому випадку, умови інтерполяції та диференціювання:

$$S_i(x_i) = y_i,\ S_i(x_i) + 1 = y_{i+1},\ i = 1,2,...n-1,$$
$$S_i^l(x_{i+1}) = S_{i+1}^l(x_{i+1}),\ S_i^{ll}(x_{i+1}) = S_{i+1}^{ll}(x_{i+1}), i = 1,...n-2.$$

Доповнивши одержані рівняння двома крайовими умовами вигляду (5.35), знову одержимо систему з ***4n-4*** рівнянь. Якщо виключити деякі невідомі, її можна легко спростити. Запишемо систему докладніше, позначивши $h_i = x_{i+1} - x_i$:

$$a_i = y_i, \qquad i = 1,..., n-1, \qquad (5.37)$$

$$a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = y_{i+1}, \qquad i = 1,..., n-1, \qquad (5.38)$$

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1}, \qquad i = 1,..., n-2, \qquad (5.39)$$

$$c_i + 6d_i h_i = 2c_{i+1}, \qquad i = 1,..., n-2, \qquad (5.40)$$

$$c_i = 0, \tag{5.41}$$

$$2c_{n-1} + 6d_{n-1}h_{n-1} = 0, \tag{5.42}$$

Знайдемо із (5.40) та (5.42) величини d_i:

$$d_i = \frac{2c_{i+1} - 2c_i}{6h_j} = \frac{c_{i+1} - c_i}{h_j}, \ i = 1,...1n-2, \ d_{n-1} = \frac{-c_{n-1}}{3h_{n-1}} \tag{5.43}$$

та підставимо d_i та a_i в (5.38):

$$y_i + b_i h_i + c_i h_i^{\,2} + \frac{c_{i+1}c_i}{3h_i} h_i^{\,3} = y_{i+1}, \ i = 1..., n-2,$$

$$y_{n-1} + b_{n-1}h_{n-1} + c_{n-1}h_{n-1}^{\,2} + \frac{c_{n-1}}{3h_{n-1}} h_{n-1}^{\,3} = y_m$$

З цих двох співвідношень виразимо b_i:

$$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{c_{i+1} - 2c_i}{3} h_i, \ i = 1,..., n-2 \tag{5.44}$$

$$b_{n-1} = \frac{y_n - y_{n-1}}{h_{n-1}} - \frac{2}{3} c_{n-1}h_{n-1}$$

Нарешті, в (5.39) підставляємо одержаний вираз b_i та d_i через c_1:

$$\frac{y_{i+1} - y_i}{h_i} - \frac{c_{i+1} + 2c_i}{3} h_i + 2c_i h_i + 3\frac{c_{i+1} - c_i}{3h_i} h_i^{\,2} = \frac{y_{i+2} - y_{i+1}}{h_{i+1}} - \frac{c_{i+2} + 2c_{i+1}}{3} h_{i+1}, \ i = 1,..., n-3,$$

$$\frac{y_{n-1} - y_{n-2}}{h_{n-2}} - \frac{c_{n-1} + 2c_{n-2}}{3} h_{n-2} + 2c_{n-2}h_{n-2} + 3\frac{c_{n-1} - c_{n-2}}{3h_{n-2}} h^2_{n-2} = \frac{y_n - y_{n-1}}{h_{n-1}} - \frac{2}{3} c_{n-1}h_{n-1}.$$

Одержали систему ***n-2*** рівнянь відносно величин C_i ***(i=1,:..,n-1)***. Оскільки із (5.41) $C_i = 0$, то кількість невідомих збігається з кількістю рівнянь. Після нескладних перетворень система набирає вигляду:

$$\begin{bmatrix} 2(h_1 + h_2) & h_2 & & \\ h_2 & 2(h_1 + h_2) & h_3 & \\ & \dots & \dots & \\ & & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{bmatrix} \begin{bmatrix} c_2 \\ c_3 \\ \dots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \dots \\ \gamma_{n-2} \end{bmatrix} \tag{5.45}$$

де $\gamma_i = 3\left[\frac{y_{i+2} - y_{i+1}}{h_{i+1}} - \frac{y_{i+1} - y_i}{h_i}\right]$.

Таким чином, для побудови сплайну у вигляді (5.36) необхідно:

1. Розв'язавши систему (5.45), знайти величини $c_2,\dots,c_{n-1}$. Взяти $c_1 = 0$.
2. За формулами (5.43) обчислити $d_1,...,d_{n-1}$.

3. За формулами (5.44) обчислити $b_1,...,b_{n-1}$.

4. Взяти $a_1 = y_1|, i=1,...,n-1$.

Відмітимо, що за коефіцієнтами сплайну, побудованого у такому вигляді, можна легко оцінити його похідні у вузлах таблиці:

$$b_i = S_i^I(x_i), c_i = \frac{S_i^{II}(x_i)}{2}, i = 1,..., n-1$$

Розглянемо ще один підхід до формування інтерполяційних кубічних сплайнів, відповідно до якого на кожному частковому відрізку $[x_i,x_{i+1}]$ $(i=1,...,n-1)$ кубічний многочлен будують у вигляді:

$$S_i(x) = \frac{(x_{i+1}-x)^2\left[2(x-x_i)+h\right]}{h^3} y_i + \frac{(x-x_i)^2\left[2(x_{i+1}-x)+h\right]}{h^3} y_{i+1} +$$
$$+ \frac{(x_{i+1}-x)^2(x-x_i)}{h^2} m_i + \frac{(x-x_i)^2(x-x_{i+1})}{h^2} m_{i+1}. \qquad (5.46)$$

Тут x_i,y_i – координати табличних точок $(i=1,...,n)$; m_i – нахили сплайна у вузлах таблиці $(S^I(x_i)=m_i$; h – відстань між вузлами $x_i)$, які в цьому підході беруть рівновіддаленими.

Перевіркою можна переконатися, що сплайн у формі (5.46) дійсно є інтерполяційним, тобто $S_j(x_i)=y_i, Sj(x_{i+1})=y_{i+1}$, а його похідні у вузлах дорівнюють нахилам: $S^I_i(x_i)=m_i$, $S^I_{i+1}(x_{i+1})=m_{i+1}$.

Звідси випливає, що при заданій таблиці $\{x_i,y_i\}$ та нахилах набір кубічних многочленів вигляду (5.46) є неперервною та диференційованою кусковою функцією, тобто сплайном третього степеня з дефектом не більше як два.

Існують такі способи задавання нахилів:

1. Виходячи з заданої таблиці обчислюють нахили за формулами чисельного диференціювання другого порядку:

$$m_i = \frac{y_{i+1}-y_{i-1}}{2h}, i = 2,...n-1,$$
$$m_1 = \frac{4y_2 - y_3 - 3y_1}{2h},\ m_1 = \frac{3y_n - y_{n-23} - 4y_{n-1}}{2h}.$$

Одержані значення підставляють у формулу (5.46).

2. Якщо відомі значення похідної інтерпольованої функції у вузлах $y^I_i = f^I(x_i)$, то беруть $m_i=y^I_i$, $i=1,...,n$.

Два наведені способи задавання нахилів називають локальними, бо на кожному частковому відрізку сплайн будується окремо. При цьому неперервність другої похідної не гарантується і дефект такого сплайну звичайно дорівнює двом.

3. Формується СЛАР відносно нахилів. Її рівняння одержують з умов неперервності другої похідної у внутрішніх вузлах таблиці:

$$S^{II}{}_{i-1}(x_i) = S^{II}{}_i(x_i), \quad l = 2,..., n-1$$

В результаті одержимо СЛАР розмірності ***n-2*** відносно ***n*** невідомих нахилів **m**:

$$m_{i-1} + 4m_i + m_{i+1} = \frac{3(y_{i+1} - y_{i-1})}{h},\ i - 2,..., n-1 \qquad (5.47)$$

До системи слід додати ще дві умови, які мають назву крайових, тому що з їх допомогою звичайно задаються значення m_1, і m_n нахилів на кінцях відрізка *[a,b]*. Тоді система набуває вигляду:

$$\begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & & \dots & & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{bmatrix} \begin{bmatrix} m_2 \\ m_3 \\ \dots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = \begin{bmatrix} \gamma_2 - m_1 \\ \gamma_3 \\ \\ \gamma_{n-2} \\ \gamma_{n-1} - m_n \end{bmatrix}$$

де $\gamma_i = \frac{3(y_{i+1} - y_{i-1})}{h}$.

Крайові умови можуть бути задані такими способами:

a) якщо відомі $y^l_i = f^l(x_1)$ і $y^l_n = f^l(x_n)$, то беруть $m_1 = y^l_i$, $m_n = y^l_n$;

b) значення похідних ***f'(x)*** на кінцях відрізка апроксимуємо за формулами чисельного диференціювання третього порядку:

$$m_1 = \frac{1}{6h}(-11y_1 + 18y_2 - 9y_2 + 2y_4),$$
$$m_n = \frac{1}{6h}(-11y_n + 18yn - 1_2 - 9y_{n-2} + 2y_{n-3});$$

c) можуть бути відомі значення другої похідної на кінцях відрізка ***[a,b]***: $y^{II}{}_1 = f^{II}(x_1), y^{II}{}_n = f^{II}(x_n)$. У цьому випадку із співвідношень $S^{II}{}_1(x_1) = y^{II}{}_1$ і $S^{II}{}_{m-1}(x_n) = y^{II}{}_n$ одержимо такі крайові умови:

$$m_1 = -\frac{m_2}{2} + \frac{3}{2} \cdot \frac{y_2 - y_1}{h} - \frac{h}{4} y''_1,$$

$$m_n = -\frac{m_{n-1}}{2} + \frac{3}{2} \cdot \frac{y_n - y_{n-1}}{h} - \frac{h}{4} y''_n.$$

Одна з реалізацій алгоритму інтерполяції за допомогою кубічного сплайну (у вигляді окремого модуля) представлена на рис. 5.25.

```
## module cubicSpline
''' k = curvatures(xData,yData).
    Повертає значення нахилів сплайну в точках.

    y = evalSpline(xData,yData,k,x).
    Вираховує значення сплайну в точці x.
'''
from numpy import zeros,ones,float64,array
from LUdecomp3 import *

def curvatures(xData,yData):
    n = len(xData) - 1
    c = zeros((n),dtype=float64)
    d = ones((n+1),dtype=float64)
    e = zeros((n),dtype=float64)
    k = zeros((n+1),dtype=float64)
    c[0:n-1] = xData[0:n-1] - xData[1:n]
    d[1:n] = 2.0*(xData[0:n-1] - xData[2:n+1])
    e[1:n] = xData[1:n] - xData[2:n+1]
    k[1:n] =6.0*(yData[0:n-1] - yData[1:n]) \
                /(xData[0:n-1] - xData[1:n]) \
              -6.0*(yData[1:n] - yData[2:n+1])   \
                /(xData[1:n] - xData[2:n+1])
```

```
    LUdecomp3(c,d,e)
    LUsolve3(c,d,e,k)
    return k

def evalSpline(xData,yData,k,x):
    def findSegment(xData,x):
        iLeft = 0
        iRight = len(xData)- 1
        while 1:
            if (iRight-iLeft) <= 1: return iLeft
            i =(iLeft + iRight)/2
            if x < xData[i]: iRight = i
            else: iLeft = i

    i = findSegment(xData,x)
    h = xData[i] - xData[i+1]
    y = ((x - xData[i+1])**3/h - (x - xData[i+1])*h)*k[i]/6.0 \
      - ((x - xData[i])**3/h - (x - xData[i])*h)*k[i+1]/6.0   \
      + (yData[i]*(x - xData[i+1])                           \
       - yData[i+1]*(x - xData[i]))/h
    return y
```

Рис. 5.25. Реалізація алгоритму інтерполяції за допомогою кубічного сплайну мовою *Python*

Ця реалізація використовує *LU*-метод розв'язання СЛАР, який описано в розділі 6. В даній реалізація прийнято, що значення нахилів для крайових умов дорівнюють 0. Функція curvatures визначає нахили сплайну в точках. На вхід цієї функції подаються табличні значення xData, yData. Функція evalSpline визначає значення сплайну в заданій точці x. На вхід цієї функції подаються:

табличні значення xData, yData, а також визначені попередньою функцією коефіцієнти нахилу в точках k. На рис. 5.26 представлена програма використання створеного модуля.

```
#!/usr/bin/python
from numpy import array,float64
from cubicSpline import *

xData = array([1,2,3,4,5],dtype=float64)
yData = array([0,1,0,1,0],dtype=float64)
k = curvatures(xData,yData)
while 1:
   try: x = eval(raw_input("\nx ==> "))
   except SyntaxError: break
   print "y =",evalSpline(xData,yData,k,x)
raw_input("OK")
```

Рис. 5.26. Приклад використання створеного модуля

Розглянемо декілька прикладів визначення значень функції методом інтерполяції за допомогою кубічного сплайну.

Приклад 5.7. Використати натуральний кубічний сплайн для визначення значення функції в точці $x=1.5$. Функція задана таблицею:

x	1	2	3	4	5
y	0	1	0	1	0

Враховуючи, що друга похідна натурального сплайну дорівнює 0 в першій та останній точках, знаходимо $m_0=m_4=0$. Другі похідні в інших точках можна отримати з (5.47). Підставляючи $i=1,2,3\ldots$ в рівняння отримуємо наступну систему лінійних рівнянь.

$$\begin{cases} 0 + 4m_1 + m_2 = 6[0 - 2(1) + 0] = -12 \\ m_1 + 4m_2 + m_3 = 6[1 - 2(0) + 1] = 12 \\ m_2 + 4m_3 + 0 = 6[0 - 2(1) + 0] = -12 \end{cases}$$

Рішенням системи рівнянь є: $m_1 = m_3 = -\frac{30}{7}$; $m_2 = \frac{36}{7}$. Точка ***x=1.5*** лежить між точками *1* та *2*. Відповідний інтерполянт можна отримати з (5.46) підставивши ***i=0***. Підставивши $x_i - x_{i+1} = -h = -1$, отримаємо:

$$S_0(x) = -\frac{m_0}{6}[(x-x_1)^3 - (x-x_1)] + \frac{m_1}{6}[(x-x_0)^3 - (x-x_0)] - [y_0(x-x_1) - y_1(x-x_0)]$$

Тому, $y(1{,}5) = S_0(1{,}5) = 0 + \frac{1}{6}\left(-\frac{30}{7}\right)[(1{,}5-1)^3 - (1{,}5-1)] - [0 - 1(1{,}5-1)] =$ 0,7679.

Приклад 5.8. Іноді потрібно змінити одну або обидві крайові умови для сплайну. Використати крайову умову у вигляді $S_0'(0) = 0$ замість $S_0''(0) = 0$ та знайти значення функції в точці ***x=2.6***. Функція задана таблицею:

x	0	1	2	3
y	1	1	0,5	0

Для використання нової крайової умови потрібно дещо змінити рівняння (5.47). Підставивши ***i=0*** в (5.46) та продиференціювавши, отримаємо:

$$S_0'(x) = \frac{m_0}{6}\left[3\frac{(x-x_1)^2}{x_0-x_1} - (x_0-x_1)\right] - \frac{m_1}{6}\left[3\frac{(x-x_1)^2}{x_0-x_1} - (x_0-x_1)\right] + \frac{y_0-y_1}{x_0-x_1}.$$

Так як $S_0'(x_0) = 0$, то

$$\frac{m_0}{3}(x_0-x_1) + \frac{m_1}{6}(x_0-x_1) + \frac{y_0-y_1}{x_0-x_1} = 0,$$

або

$$2m_0 + m_1 = -6\frac{y_0-y_1}{(x_0-x_1)^2}.$$

З заданої умови видно, що $y_0 = y_1 = 1$. І тоді, останнє рівняння перетворюється в: $2m_0 + m_1 = 0$ (a). Інші рівняння з (5.47) залишаються без змін. Враховуючи, що $m_3=0$, маємо:

$$m_0 + 4m_1 + m_2 = 6[1 - 2(1) + 0{,}5] = -3 \ (b)$$

$$m_1 + 4m_2 = 6[1 - 2(0{,}5] + 0] = 0 \ (c)$$

Розв'язком СЛАР *(a)-(c)* є: $m_0 = 0{,}4615; m_1 = -0{,}9231; m_2 = 0{,}2308$.

Тепер можна визначити інтерполянт, застосувавши (5.46). Підставляючи ***i=2*** та $x_i - x_{i+1} = -h = -1$, отримаємо:

$$S_2(x) = \frac{m_2}{6}[-(x-x_3)^3 - (x-x_3)] - \frac{m_3}{6}[-(x-x_2)^3 - (x-x_2)] - y_2(x-x_3) + y_3(x-x_2).$$

$$y(2{,}6) = S_2(2{,}6) = \frac{0{,}2308}{6}[-(-0{,}4)^3 + (-0{,}4)] - 0 - 0{,}5(-0{,}4) + 0 = 0{,}1871.$$

5.4. Середньоквадратичне наближення

5.4.1. Метод найменших квадратів. Нормальні рівняння

Нагадаємо, що при інтерполяції таблично заданої функції ***f(x)*** поліномом ***p(x)*** потрібно, щоб ***f(x)*** та ***p(x)*** збігалися в табличних точках (вузлах інтерполяції). Причому, якщо функція задана в ***(n+1)***-й точці $x_0, x_1, ..., x_n$, то існує єдиний поліном степеня не вище ***n*** такий, що $p(x_i) = f(x_j)$ для $i=0,1,...,n$.

Але в деяких випадках виконання цієї умови є важким або навіть зайве. Наприклад, за великої кількості вузлів одержимо поліном високого степеня, що збільшить похибки обчислень. Крім того, табличні дані можуть бути одержані вимірюванням і містити похибки, інтерполяційний поліном повторював би ці похибки. Тому іноді краще, щоб графік полінома не збігався, а проходив би близько табличних точок. Степінь такого полінома беруть невисоким (1...4) і він менший за ***n***. Нехай функція ***f(x)*** задана в ***m*** точках $x_1 \ldots x_m$:

$$f(x_1) = y_1, ..., f(x_m) = y_m$$

В методі найменших квадратів (МНК) відшукують такий поліном $p(x) = a_0 + a_1 x + ... + a_n x^n$, щоб для нього величина

$$g = \sum_{i=1}^{m} (P(x_i) - y_i)^2$$

була мінімальною серед усіх поліномів степеня ***n***. Тобто треба знайти такі коефіцієнти $a_0, a_1, ..., a_n$, щоб сума квадратів відхилень $(P(x_i) - y_i)$ була найменшою. Це можна зробити декількома способами, наприклад, за допомогою нормальних рівнянь, ортогональних поліномів, методів оптимізації. Розглянемо перший з них.

Величина ***g*** є функцією ***n+1*** аргумента a_i:

$$g(a_0,...,a_n)=\sum(a_0+a_1x_1+...+a_nx_i^n-y_i)^2 \qquad (5.48)$$

Потрібно знайти точку її мінімуму, необхідною умовою чого є рівність нулю часткових похідних $\partial g/\partial a_j$ ($j=0,1,...,n$) у цій точці.

Розглянемо спочатку простий випадок, коли ***n=0***, тобто апроксимуючий поліном є константою: ***p(x)=a₀***. Наприклад, масу деякого предмета вимірювали на ***m*** різних важелях і одержали значення ***y₁,...,yₘ***. Що вважати масою предмета? Функція ***g*** стає функцією одного аргументу ***a₀***: $g(a_0)=\sum(a_0-y_i)^2$.

З курсу математики відомо, що така функція досягає мінімуму в точці ***a*₀***, якщо ***g′(a*₀)=0*** та ***g″(a*₀)>0***. Запишемо першу умову докладніше (сумування проводять в межах від *1* до *m*):

$$g^l(a_0)=\sum_{i=1}^{m}2(a_0-y_i)=2(\sum a_0-\sum y_i)=2(ma_0-\sum y_i)=0$$

Звідси випливає, що $g^{ll}(a_0)>0$, тобто МНК дає середнє арифметичне. Відмітимо, що в будь якій точці: $g^{ll}(a_0)=2m>0$.

Нехай тепер ***n-1***. Тоді апроксимуючий поліном має вигляд $p(x)=a_0+a_1x$. Необхідними умовами мінімуму функції $g(a_0a_1)=\sum_{i=1}^{m}(a_0+a_1x_1-y_i)^2$ є рівність нулю частинних похідних:

$$\frac{\partial g}{\partial a_0}=\sum 2(a_0+a_1x_1-y_i)=2(\sum a_0+\sum a_1x_1-\sum y_i)=2\left[ma_0+\left(\sum x_1\right)a_1-\sum y_i\right]=0,$$

$$\frac{\partial g}{\partial a_1}=\sum 2(a_0+a_1x_1-y_i)x_i=2(\sum a_0x_i+\sum a_1x_1^{\,2}-\sum x_iy_i)=2\left[(\sum x_i)a_0+\left(\sum x_1^{\,2}\right)a_1-\sum x_iy_i\right]=0$$

Одержали систему двох лінійних рівнянь з двома невідомими ***a₀*** та ***a₁***:

$$\begin{cases}ma_0+(\sum x_i)a_i=\sum y_i,\\(\sum x_i)a_0+(\sum x_i^{\,2})a_i=\sum x_iy_i\end{cases}$$

або в матричній формі:

$$\begin{pmatrix} m & \sum x_i \\ \sum x_i & \sum x_i{}^2 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}=\begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix}.$$

За формулою Крамера одержимо розв'язок:

$$a_0=\frac{\sum x_i^2\cdot\sum y_i-\sum x_i y_i\cdot\sum x_i}{m\sum x_i^2-(\sum x_i)^2};\ a_1=\frac{m\sum x_i y_i-\sum y_i\cdot\sum x_i}{m\sum x_i^2-(\sum x_i)^2}$$

У випадку довільного ***n*** цілком аналогічно прирівняємо до нуля $\frac{\partial g}{\partial a_j}$ (***j=0,...,n***) і одержимо СЛАР порядку ***n+1*** відносно ***(n+1)-ї*** невідомої $a_0,...,a_n$:

$$\frac{\partial g}{\partial a_0}=2\sum(a_0+a_1x_1+...+a_nx_i^n-y_i)=0,$$
$$\frac{\partial g}{\partial a_1}=2\sum(a_0+a_1x_1+...+a_nx_i^n-y_i)x_1=0,$$
$$...$$
$$\frac{\partial g}{\partial a_n}=2\sum(a_0+a_1x_1+...+a_nx_i^n-y_i)x^n{}_1=0.$$

Ці рівняння називають *нормальними*. Запишемо цю систему лінійних алгебраїчних рівнянь, групуючи члени, що містять a_j:

$$\begin{cases} ma_0+\left(\sum x_i\right)a_i+\left(\sum x^2{}_i\right)a_2+...+\left(\sum x^n{}_i\right)a_n=\sum y_i \\ \left(\sum x_i\right)a_0+\left(\sum x^2{}_i\right)a_i+\left(\sum x^3{}_i\right)a_2+...+\left(\sum x^n{}_i\right)a_n=\sum x_i y_i \\ ... \\ \left(\sum x^n{}_i\right)a_0+\left(\sum x^{n+1}{}_i\right)a_i+\left(\sum x^{n+2}{}_i\right)a_2+...+\left(\sum x^{2n}{}_i\right)a_n=\sum x^n{}_i y_i \end{cases} \qquad (5.49)$$

або у матричній формі:

$$\begin{pmatrix} m & \sum x_i & \sum x_i^2 & ... & \sum x_i^n \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & ... & \sum x_i^{n+1} \\ & & & ... & \\ \sum x^n{}_i & \sum x_i^{n+1} & \sum x_i^{n+2} & ... & \sum x_i^{2n} \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_m \end{pmatrix}=\begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ ... \\ \sum x_i^n y_i \end{pmatrix} \qquad (5.50)$$

Доведено, що ця система має єдиний розв'язок, отже, задача апроксимації за методом найменших квадратів теж має єдиний розв'язок.

Розглянемо більш загальну постановку задачі середньоквадратичного наближення функцій.

Відмітимо, що апроксимуюча функція не обов'язково повинна бути алгебраїчним многочленом. Нехай $\varphi_0, \varphi_1, \ldots, \varphi_n$ – задані функції однієї змінної, а $w_1, \ldots, w_n$ – задані додатні числа (ваги). Тоді загальна постановка

задачі апроксимації за методом найменших квадратів полягає в тому, що потрібно знайти числа $a_0,a_1,...,a_n$, які мінімізують функцію

$$g(a_0,...,a_n)=\sum_{i=1}^{m} w_i\left[a_0\varphi_0(x_i)+a_1\varphi_1(x_i)+...+a_n\varphi_n(x_i)-y_i\right]^2 \qquad (5.51)$$

Якщо $w_1 = ... = w_m = 1$, а $\varphi_j(x)=x^j$, то одержимо попередню постановку (5.48). Окрім степеневої іноді використовують такі базисні функції:

$\varphi_j(x)=Sin\ j\pi x,\ \varphi_j(x)=e^{\alpha_j x}$ де $j=0,1,...,n;\ \alpha_j$ – задані числа.

Ваги w_i використовують, щоб надати більшу або меншу роль вузлам сітки. Наприклад, якщо значення $y_1,...,y_{10}$ виміряні більш точно, то можна взяти $w_1 =... = w_{10} =5$, а інші ваги взяти меншими: $w_{11} =... = w_m =1$.

Для функції (5.51), як і раніше, можна побудувати нормальні рівняння. Часткові похідні цієї функції мають вигляд:

$$\frac{\partial g}{\partial a_j}=2\sum_{i=1}^{m} w_i\varphi_i(x_i)\left[a_0\varphi_0(x_i)+a_1\varphi_1(x_i)+...+a_n\varphi_n(x_i)-y_i\right]$$

Покладемо їх такими, що дорівнюють нулю і об'єднаємо коефіцієнти при a_i. Одержимо СЛАР:

$$\begin{pmatrix} \sum w_i\varphi_0(x_i)\varphi_0(x_i) & \sum w_i\varphi_1(x_i)\varphi_0(x_i) & ... & \sum w_i\varphi_n(x_i)\varphi_0(x_i) \\ \sum w_i\varphi_0(x_i)\varphi_1(x_i) & ... & & \\ & ... & & \\ \sum w_i\varphi_0(x_i)\varphi_n(x_i) & \sum w_i\varphi_1(x_i)\varphi_n(x_i) & ... & \sum w_i\varphi_n(x_i)\varphi_n(x_i) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_n \end{pmatrix} = \begin{pmatrix} \sum w_i\varphi_0(x_i)y_i \\ \sum w_i\varphi_1(x_i)y_i \\ ... \\ \sum w_i\varphi_n(x_i)y_i \end{pmatrix} \qquad (5.52)$$

За деяких умов, що накладаються на вузли x_i та функції $\varphi_j(x)$, вона має єдиний розв'язок. Але при $n>5$ системи (5.50) та (5.52) погано обумовлені, тому їх використовують лише при малих значеннях n. При більших значеннях n той самий поліном середньоквадратичної апроксимації можна одержати за допомогою ортогональних поліномів.

5.4.2. Застосування ортогональних поліномів у методі найменших квадратів

Нехай $q_0, q_1,\ldots,q_n$ – поліноми степенів $0,1,...,n$ відповідно, їх називають *взаємно ортогональними* на множині точок $x_1,...,x_n$, якщо

$$\sum_{i=1}^{m} q_k(x_i) \cdot q_j(x_i) = 0 \text{ при } k, j = 0,1,...,n \quad i \quad k \neq j$$

Якщо у системі (5.52) базисними функціями будуть $q_k(x)(\varphi_k = q_k)$, то ця система перетвориться в систему з діагональною матрицею і матиме вигляд

$$\sum_{i-1}^{m} [q_k(x_i)]^2 a_k = \sum_{i-1}^{m} q_k(x_i) y_i, \; k = 0,1,...,n.$$

З такої системи легко знаходять невідомі a_k:

$$a_k = \frac{\sum_{i-1}^{m} q_k(x_i) y_i}{\sum_{i-1}^{m} [q_k(x_i)]^2} \tag{5.53}$$

Тобто буде одержаний потрібний апроксимуючий поліном:

$$q(x) = \sum_{k=0}^{n} a_k q_k(x) \tag{5.54}$$

Він є іншим представленням того самого полінома, що одержується за допомогою нормальних рівнянь, бо задача апроксимації МНК має єдиний розв'язок для кожного n. Але як побудувати ортогональні поліноми, що дають тривіальну діагональну СЛАР відносно a_j? Розглянемо один із способів. Покладемо

$$q_0(x) = 1, \; q_1(x) = x - \alpha_1$$

Константу α_1 знайдемо виходячи з умови ортогональності q_0 і q_1 на множині точок x_1:

$$0 = \sum_{i=1}^{m} q_0(x_i) \cdot q_1(x_i) = \sum (x_i - \alpha_1) = \sum x_i - m\alpha_1.$$

Одержимо $\alpha_1 = \frac{1}{m} \sum_{i=1}^{m} x_i$.

Нехай тепер

$$q_2(x) = xq_1(x) - \alpha_2 q_1(x) - \beta_1.$$

Константи α_2 та β_1 знайдемо виходячи з умов ортогональності q_2 до поліномів q_0 та q_1:

$$q_0(x)q_2(x) = \sum_{i=1}^{m}\left[x_i q_1(x_i) - \alpha_2 q_1(x_i) - \beta_1\right] = 0,$$

$$q_1(x)q_2(x) = \sum_{i=1}^{m}\left[x_i q_1(x_i) - \alpha_2 q_1(x_i) - \beta_1\right]q_1(x_i) = 0.$$

Оскільки q_0 та q_1 ортогональні, то $m\sum_{i=1}^{m} q_1(x_i) = 0$ і ці рівняння спрощуються:

$$\sum_{i=1}^{m} x_i q_1(x_i) - m\beta_1 = 0,$$

$$\sum_{i=1}^{m} x_i\left[q_1(x_i)\right]^2 - \alpha_2\sum_{i=1}^{m}\left[q_1(x_i)\right]^2 = 0,$$

Визначимо α_2 та β_1 так:

$$\beta_1 = \frac{1}{m}\sum x_i q_i(x_i),$$

$$\alpha_2 = \frac{\sum x_i\left[q_1(x_i)\right]^2}{\sum\left[q_1(x_i)\right]^2}$$

Аналогічно будують наступні поліноми. Нехай поліноми $q_0, q_1, \ldots, q_j$ вже знайдені. Покладемо

$$q_{j+1}(x) = xq_j(x) - \alpha_{j+1}q_j(x) - \beta_j q_{j-1}(x) \qquad (5.55)$$

де константи α_{j+1} та β_j треба визначити виходячи з умов ортогональності q_{j+1} до поліномів q_j та q_{j-1}. Якщо ці дві умови виконуватимуться, то q_{j+1} буде ортогональний до усіх попередніх поліномів $q_k(x)$, $(k<j-1)$. Дійсно,

$$\sum_{i=1}^{m} q_{j+1}(x_i)q_k(x_i) = \sum\left[x_i q_j(x_i) - \alpha_{j+1}q_j(x_i) - \beta_j q_{j-1}(x_i)\right]q_k(x_i) =$$

$$= \sum x_i q_j(x_i)q_k(x_i) - \alpha_{j+1}\sum q_j(x_i)q_k(x_i) - \beta_j q_{j-1}(x_i)q_k(x_i).$$

Оскільки поліноми $q_0, \ldots, q_j$ ортогональні за припущенням, то два останніх доданки дорівнюють нулю. Поліном $xq_k(x)$ має степінь $(k+1)$ і, отже, може бути представлений як лінійна комбінація поліномів $q_0, \ldots, q_k$. Але $k+1<j-1$, тому перший доданок теж дорівнює нулю. Отже, $q_{j+1}(x)$ ортогональна до усіх поліномів $q_0 \ldots\ldots q_j$.

Знайдемо α_{j+1} та β_j:

$$\sum q_{j+1}(x_i)q_j(x_i) = 0,$$

$$\sum q_{j+1}(x_i)q_{j-1}(x_i) = 0.$$

Підставимо сюди з (5.55) вираз для q_{j+1} і одержимо:

$$\alpha_{j+1} = \frac{\sum_{i=1}^{m} x_i \left[q_j(x_i)\right]^2}{\sum_{i=1}^{m} \left[q_j(x_i)\right]^2} \qquad (5.56)$$

$$\beta_j = \frac{\sum_{i=1}^{m} x_i q_j(x_i) \cdot q_{j-1}(x_i)}{\sum_{i=1}^{m} \left[q_{j-1}(x_i)\right]^2} \qquad (5.57)$$

З викладеного випливає алгоритм середньоквадратичного наближення функцій за допомогою ортогональних поліномів:

1. Покласти $q_{-1}(x) = 0,\ q_0(x) = 1,\ j = 0,\ a_0 = \frac{\sum_{i=1}^{m} y_i}{m}$.

2. Обчислити a_{j+1} за формулою (5.56).

3. Обчислити β_j за формулою (5.57) при $j>0$ або покласти $\beta_0=0$.

4. Обчислити коефіцієнти полінома $q_{j+1}(x)$ за формулою (5.55).

5. Обчислити $a_{j+1} = \frac{\sum_{i=1}^{m} q_{j+1}(x_i) y_i}{\sum_{i=1}^{m} \left[q_{j+1}(x_i)\right]^2}$.

6. Покласти $j=j+1$. Якщо $j \le n+1$, то перехід на п.2.

7. Побудувати поліном

$P_n(x) = a_0 q_0(x) + ... + a_n q_n(x) = c_0 + c_1 x + ... + c_n x^n$.

Переваги цього підходу полягають в тому, що не потрібно розв'язувати СЛАР загального вигляду. Також маємо можливість будувати поліном степінь за степенем. Наприклад, можемо заздалегідь не знати, поліном якого степеня нас задовольнить. Можна почати зі степеня *1*, потім побудувати поліном другого степеня і т.д., поки не одержимо потрібний поліном з невеликим g.

Реалізація алгоритму апроксимації за допомогою ортогональних поліномів (у вигляді модуля) представлена на рис. 5.27.

```
# -*- coding: cp1251 -*-
```

```
## module polyFit
''' c = polyFit(xData,yData,m).
    Повертає коефіцієнти полінома
    p(x) = c[0] + c[1]x + c[2]x^2 +...+ c[m]x^m
    що підходить під задані дані в сенсі МНК.

    sigma = stdDev(c,xData,yData).
    Розраховує стандартне відхилення між p(x)
    і даними.
'''
from numpy import zeros,float64
from math import sqrt
from gaussPivot import *

def polyFit(xData,yData,m):
    a = zeros((m+1,m+1),dtype=float64)
    b = zeros((m+1),dtype=float64)
    s = zeros((2*m+1),dtype=float64)
    for i in range(len(xData)):
        temp = yData[i]
        for j in range(m+1):
            b[j] = b[j] + temp
            temp = temp*xData[i]
        temp = 1.0
        for j in range(2*m+1):
            s[j] = s[j] + temp
            temp = temp*xData[i]
```

```
    for i in range(m+1):
        for j in range(m+1):
            a[i,j] = s[i+j]
    return gaussPivot(a,b)


def stdDev(c,xData,yData):


    def evalPoly(c,x):
        m = len(c) - 1
        p = c[m]
        for j in range(m):
            p = p*x + c[m-j-1]
        return p


    n = len(xData) - 1
    m = len(c) - 1
    sigma = 0.0
    for i in range(n+1):
        p = evalPoly(c,xData[i])
        sigma = sigma + (yData[i] - p)**2
    sigma = sqrt(sigma/(n - m))
    return sigma
```

Рис. 5.27. Реалізація алгоритму апроксимації за допомогою ортогональних поліномів

Модуль складається з двох функцій: polyFit та stdDev. Перша функція призначена для пошуку коефіцієнтів ортогонального полінома та має на вході наступні параметри: xData,yData – вхідна функція, яка задана таблично, m – степінь шуканого полінома. Друга функція визначає стандартне

відхилення отриманого полінома від заданої функції. Для вирішення СЛАР, яку потрібно розв’язати при пошуку полінома в функції polyFit використано метод Гауса з обертанням (gaussPivot). Методи вирішення СЛАР розглядаються в наступному розділі.

Приклад використання створеного модуля представлено на рис. 5.28.

```
#!/usr/bin/python
from numarray import array
from polyFit import *

xData = array([-0.04,0.93,1.95,2.90,3.83,5.0,      \
            5.98,7.05,8.21,9.08,10.09])
yData = array([-8.66,-6.44,-4.36,-3.27,-0.88,0.87, \
            3.31,4.63,6.19,7.4,8.85])
while 1:
   try:
      m = eval(raw_input("\nDegree of polynomial ==> "))
      coeff = polyFit(xData,yData,m)
      print "Coefficients are:\n",coeff
      print "Std. deviation =",stdDev(coeff,xData,yData)
   except SyntaxError: break
raw_input("Finished. Press return to exit")
```

Рис. 5.28. Приклад використання створеного модуля

5.5 Контрольні запитання

1. У чому суть задачі апроксимації?

2. Що таке інтерполяція?

3. Що таке інтерполяційний многочлен?

4. Розкрити суть глобальної інтерполяції.

5. Лінійна інтерполяція.

6. Метод невизначених коефіцієнтів.

7. Метод інтерполяції Лагранжа.

8. Метод інтерполяції Ньютона.

9. Метод кусково-лінійної інтерполяції.

10. Метод кусково-нелінійної інтерполяції.

11. Що таке сплайн? Степінь сплайна, дефект сплайна.

12. Метод інтерполяції параболічними сплайнами.

13. Метод інтерполяції кубічними сплайнами.

14. Метод найменших квадратів.

15. Які рівняння називаються нормальними?

16. Метод ортогональних поліномів.

Розділ 6. Розв’язування систем лінійних рівнянь

6.1. Основні поняття і загальні положення

Рішення багатьох прикладних, економічних і інших інженерно-технічних завдань безпосередньо пов'язано з обчислювальними методами лінійної алгебри. Важливість освоєння даних методів визначається і тим, що до цих методів приводяться багато завдань чисельного аналізу. У цьому розділі розглядаються питання про чисельне рішення систем лінійних рівнянь алгебри, про обернення матриць і знаходження власних значень та власних векторів матриць.

Чисельні методи рішення систем лінійних рівнянь і звернення матриць можна розділити на три групи: 1) *точні* (прямі) методи; 2) *ітераційні* методи (послідовних наближень); 3) *комбіновані* методи. Точні методи характеризуються можливістю набуття дійсних значень невідомих величин. При цьому, звичайно, допускається, що всі обчислення здійснюються над дійсними значеннями величин. Ітераційні методи (послідовних наближень) характеризуються тим, що із самого початку задаються наближеними значеннями невідомих, по яких набувають нових наближених значень і т.д. до тих пір, поки не набувають наближених значень невідомих із заданою точністю. Комбіновані методи є сукупністю перших двох методів.

Систему лінійних рівнянь

$$\left\{\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \end{array}\right\} \qquad (6.1)$$

можна представити у вигляді одного матричного рівняння

$$\mathrm{A}\mathbf{x} = \mathbf{b} \qquad (6.2)$$

де $A = \left\{\begin{array}{cccc} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{array}\right\}$ - матриця системи;

$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix}$ - матриця-стовпець вільних членів;

$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix}$ - матриця-стовпець з невідомих.

За умови, що ***m = n*** і визначник матриці А відмінний від нуля, рішення системи (6.2) єдине і має вигляд

$$\mathbf{x} = A^{-1}\mathbf{b} \qquad (6.3)$$

де A^{-1} матриця, обернена квадратній матриці ***A***.

За визначенням матриця A^{-1} є зворотною до квадратної матриці ***A***, якщо виконується рівність

$$A \cdot A^{-1} = A^{-1} \cdot A = E \qquad (6.4)$$

де

$I = \begin{Bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{Bmatrix}$ – одинична матриця того ж порядку, що і матриця ***A***.

Як відомо з курсу алгебри, за поставлених вище умов розв'язок системи (6.2) визначається за формулами Крамера

$$x_i = \frac{\det A_i}{\det A} \qquad (6.5)$$

де $detA_i$ – визначник матриці A_i, отриманої шляхом заміни *і-го* стовпчика матриці ***A*** стовпчиком вільних членів.

Формула Крамера має досить простій вигляд, проте при її використанні необхідно проводити великий обсяг обчислень (порядку $n!(n^2+1)+n)$, що робить її практично непридатними при рішенні практичних

задач (при ***n=30*** такий обсяг стає недоступним для сучасних ЕОМ). Крім того, навіть при малих ***n*** на результат мають сильний вплив похибки округлення.

Система (6.1) називається *сумісною*, якщо вона має рішення. Система називається *визначеною*, якщо вона має єдине рішення.

6.2. Прямі методи

6.2.1. Метод Гауса

Найбільш стародавнім і відомим прямим методом розв'язування систем лінійних рівнянь є метод виключення невідомих, який пов'язують з ім'ям Гауса. Застосування цього методу засноване на наступній теоремі.

Теорема 6.1. Якщо A - квадратна матриця порядку n, то існують такі нижня матриця L і верхня матриця U того ж порядку такі, що PA=LU, де P – деяка матриця перестановок порядку n.

Якщо матриця A має відмінні від нуля головні (діагональні) мінори

$\Delta_1=a_{11}\neq 0, \Delta_2=\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}\neq 0, \ldots, \Delta_n=\det A\neq 0$

то матриця P може бути вибрана одиничною, тобто A=LU. Причому це розкладання буде єдиним, якщо наперед зафіксувати діагональні елементи однієї з трикутних матриць.

Ідея алгоритму Гауса при рішенні системи (6.2) полягає в тому, що дана система приводиться до еквівалентної системи з верхньою трикутною матрицею (прямий хід). З перетвореної таким чином системи невідомі знаходяться послідовними підстановками, починаючи з останнього рівняння перетвореної системи (зворотний хід).

Таким чином, якщо при розв'язуванні рівняння ***Ax=LUx=b*** позначити

$$U\mathbf{x}=\mathbf{y} \qquad (6.6)$$

те розв'язок одержується з

$$L\mathbf{y}=\mathbf{b}. \qquad (6.7)$$

Перетворення (6.6) називається *прямим ходом* методу Гауса, а

перетворення (6.7) – *зворотним.*

На підставі даної теореми матрицю ***A***, головний діагональний мінор якої відмінний від нуля, можна представити у вигляді добутку двох трикутних матриць:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix} \qquad (6.8)$$

Прямий хід розв'язання може бути здійснений таким чином. Вважаючи, що коефіцієнт $\mathbf{a_{11}\neq 0}$ (це не обмежує загальності міркувань, оскільки інакше досить просто змінити нумерацію рівнянь), ділимо перше з рівнянь системи (6.2) на коефіцієнт a_{11}. Виражаючи невідоме x_1 через інші невідомі першого рівняння і підставляючи це значення в ***n-1*** рівняння, що залишилися, , і т.д.,приходимо до перетвореної системи лінійних рівнянь

Порядок обчислень елементів матриць ***L*** і ***U***, а також векторів ***y*** і ***x***, з урахуванням залежності між ними, задається в наступному вигляді:

$u_{11}=1;\ u_{1i}=a_{1i}/a_{11};\ l_{1i}=a_{1i},\ i=1,...,n;$

$y_1=b_1/a_{11};\ l_{ki}=a_{ki}-\sum_{j=1}^{i-1} l_{kj}u_{ji}\ ;\ k=i,...,n;\ i=2,\dots,n.$

$u_{ik}=(a_{ik}-\sum_{j=1}^{i-1} l_{ij}u_{jk})/l_{ii};\ k=i+1,...,n;$

$y_i=(b_i-\sum_{j=1}^{i-1} l_{ij}y_j)/l_{ii}\ ;$

$x_n=y_n;\ x_{n-i}=y_{n-i}-\sum_{j=n-i+1}^{n} u_{n-i,j}x_j\ ;\ i=2,\dots,n-1;$

Розглянемо як приклад просту систему:

$$\begin{cases} 0.000100x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}$$

яка має точний розв'язок $x_1=1.00010,\ x_2=0.99990$.

Якщо задача розв'язується, наприклад, з використанням арифметики з трьома десятковими цифрами, то це означає, що залишаються тільки три старші значущі десяткові цифри будь-якого результату арифметичних

операцій. Припустимо, що результат округляється. Згідно виключенню Гауса помножимо перше рівняння на –10 000 і додамо до другого. Маємо:

$$\begin{cases} 0.000100x_1 + x_2 = 1 \\ -10000\,x_2 = -10000 \end{cases}$$ і розв'язок $x_1 = 0.000,\ x_2 = 1.000$.

Отже, має місце обчислювальна катастрофа. І справа тут зовсім не в близькості матриці системи до виродженої. Дійсно, спробуємо переставити рівняння (тобто виконавши процес вибору провідного елементу). Отримаємо, що виключення Гауса дає розв'язок (у тих же припущеннях про точність обчислень) $x_1 = 1.00,\ x_2 = 1.00$. Причому цей розв'язок обчислений з тією точністю, яку можна отримати для арифметики з трьома десятковими знаками.

Висновок, який можна зробити з розгляду цього прикладу такий – недостатньо уникати тільки нульових провідних елементів, необхідно також уникати вибору малих провідних елементів при приведенні матриць до трикутного вигляду.

Таким чином, приходимо до модифікованого методу *Гауса-Жордана*, який може використовувати дві стандартні стратегії вибору провідних елементів. Перша полягає в *частковому виборі* провідного елементу: на *k-му* кроці прямого ходу провідним береться найбільший (за абсолютною величиною) елемент в неприведеній частині *k-го* стовпчика, тобто

$|a_{kk}| = \max|a_{lk}|, \quad i = k, k+1, \ldots, n$.

Друга стратегія полягає в *повному виборі* провідного елемента: на до k-му кроці прямого ходу за провідний береться найбільший (за абсолютною величиною) елемент в неприведеній частині матриці, тобто

$|a_{kk}| = \max|a_{ij}|,\ i = k, k+1, \ldots, n;\quad j = k, k+1, \ldots, n$.

Хоча стратегія повного вибору надійніша, частіше застосовується все ж таки перша стратегія, особливо для вирішення великих систем, оскільки обчислювальні витрати при цьому значно менші. Більш детально цей метод буде розглянуто пізніше.

Реалізація методу Гауса у вигляді модуля мовою *Python* представлена на рис. 6.1.

```
# -*- coding: cp1251 -*-
## module gaussElimin
''' x = gaussElimin(a,b).
    Вирішує [a]{b} = {x} методом Гауса.
'''
from numarray import dot

def gaussElimin(a,b):
    n = len(b)
  # Фаза виключення
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a [i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
  # Зворотній хід
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Рис. 6.1. Модуль з реалізацію алгоритму Гауса

На вхід функції gaussElimin подаються матриця $\boldsymbol{A}$ та вектор-стовпчик $\boldsymbol{B}$.

6.2.2. Число обумовленості методу Гауса

Число обумовленості також грає фундаментальну роль в аналізі помилок округлень, здійснених в процесі гаусового виключення. Припустимо, що всі елементи $\boldsymbol{A}$ і $\boldsymbol{b}$ точно представляються числами з плаваючою точкою, і нехай $\boldsymbol{x^*}$ – вектор розв'язку, складений з чисел з плаваючою точкою. Припустимо також, що точна вирожденность матриці,

якщо вона має місце, не була виявлена і що не було ні машинних нулів, ні переповнювань. Тоді мають місце наступні нерівності:

$$\frac{\left\|b - Ax^*\right\|}{\|A\| \cdot \left\|x^*\right\|} \le \rho \cdot 2^{-t} \quad \text{і} \quad \frac{\left\|x - x^*\right\|}{\left\|x^*\right\|} \le \rho \cdot cond(A) 2^{-t}.$$

Тут ***2*** – основа системи для представлення чисел з плаваючою точкою, а ***t*** – число розрядів дробової частини, так що $\mathbf{2^{-t}}$ має значення «машинного епсилон». Величина $\boldsymbol{\rho}$ нижче визначається точніше, але звичайне її значення не перевершує ***2***.

Перша нерівність говорить, що, як правило, можна розраховувати на те, що відносна нев'язка матиме величину, порівняну з похибкою округлення, незалежно від того, наскільки погано обумовлена матриця.

Друга нерівність вимагає, щоб ***A*** була невироджена, бо в нього входить точне рішення ***x***. Ця нерівність випливає безпосередньо з першого і визначення ***cond(A)***; його сенс – відносна похибка буде мала, якщо мале число ***cond(A)***, але вона може бути дуже велика, якщо матриця майже вироджена. У граничному випадку, коли ***A*** вироджена, але це не було виявлено в процесі обчислень, перша нерівність все ж таки зберігає силу, тоді як друга не має сенсу.

Щоб точніше визначити величину $\boldsymbol{\rho}$, необхідно використовувати поняття матричної норми і встановити деякі допоміжні нерівності. Визначена раніше величина ***M*** називається нормою матриці.

$$\|A\| = \max_x \frac{\|Ax\|}{\|x\|} = \max_j \|a_j\|,$$

де $\boldsymbol{a_j}$ – стовпчики матриці ***A***.

Основний результат в дослідженні помилок округлень в гаусовому виключенні полягає в тому, що обчислений розв'язок ***x**** точно задовольняє систему

$(A+E)\mathbf{x}^* = \mathbf{b};$

де ***E*** – матриця, елементи якої мають величину порядку помилок округлень в елементах матриці ***A***. Звідси можна негайно вивести

нерівності, що відносяться до нев'язки і похибки в обчисленому розв’язку. Нев'язка визначається як $\mathbf{b} - A\mathbf{x}^* = E\mathbf{x}^*$ і, отже

$$||\mathbf{b} - A\mathbf{x}^*|| = ||E\mathbf{x}^*|| \le ||E|| \cdot ||\mathbf{x}^*||$$

У визначенні нев'язки бере участь добуток ***Ax****, так що доречно розглядати відносну нев'язку, яка порівнює норму вектора $\boldsymbol{b - Ax^*}$ з нормами ***A*** та ***x****. Оскільки $\frac{\|E\|}{\|A\|} = \rho \cdot 2^{-t}$, то з приведеної вище нерівності відразу виходить, що $\frac{\|b - Ax^*\|}{\|A\| \cdot \|x^*\|} \le \rho \cdot 2^{-t}$.

*Якщо **A** невироджена, то за допомогою оберненої матриці похибку можна записати у вигляді* $\boldsymbol{x - x^* = A^{-1}(b - Ax^*)}$ і тоді $\boldsymbol{||x - x^*|| = ||Ex^*|| \le ||A^{-1}|| \cdot ||E|| \cdot ||x^*||}$.

Простіше за все порівнювати норму похибки с нормою обчисленого розв’язку. Таким образом, відносна похибка задоволняє нерівність

$$\frac{\|x - x^*\|}{\cdot \|x^*\|} \le \rho \cdot \|A\| \cdot \|A^{-1}\| 2^{-t}.$$

Виявляється, что $||A^{-1}|| = 1/m$, і тоді $cond(A) = ||A||\ ||A^{-1}||$ і, таким чином:

$$\frac{\|x - x^*\|}{\cdot \|x^*\|} \le \rho \cdot cond(A) \cdot 2^{-t}.$$

6.2.3. Метод Гауса з вибором головного елемента

У викладеному вище методі Гауса ми припускали, що діагональні елементи a_{kk}, на які проводяться ділення при обчисленні множників $R = a_{ik}/a_{kk}$, не дорівнюють нулю. При практичному розв'язуванні систем часто $a_{kk} = 0$, що призводить до програмного переривання. Тому в програмах такий випадок повинен бути передбачений.

Нехай ми виконали ***(k-1)*** крок алгоритму Гауса та звели систему до вигляду, в якому піддіагональні елементи перших **(k-1)-**их стовпців дорівнюють нулю (рис. 6.2б) , та нехай $a_{kk} = 0$.

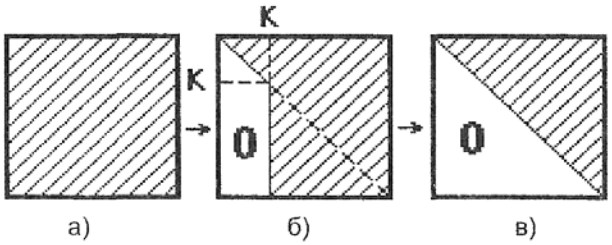


Рис. 6.2. Зміна ненульової структури матриці в процесі процедури Гауса

Тоді серед нижче розташованих елементів ***k****-го* стовпця є хоча б один, що не дорівнює нулю, інакше матриця ***A*** була б вироджена. Нехай він знаходиться в рядку з номером ***h***. Переставивши місцями ***k****-е* та ***h****-е* рівняння системи, що не змінює її розв'язок, помістимо на діагональ ненульовий елемент і зможемо продовжити процедуру Гауса. Відмітимо, що в ***k****-му* та ***h****-му* рядках матриці ***A*** лівіше ***k****-го* стовпця розташовані нулі, тому перестановка цих рядків не змінює структуру матриці, зображеної на рис. 6.2б. Саме за цієї причини пошук ненульового елемента проводимо тільки нижче діагоналі, оскільки в протилежному випадку після перестановки місцями одного з перших рівнянь і рівняння з номером ***k*** в нульовій області з'явились би ненульові елементи і в результаті матриця системи не перетворилась би до верхньої трикутної форми.

Виявляється, що перестановка рівнянь бажана, якщо навіть діагональний елемент і не дорівнює нулю. Метод Гауса є точним, якщо обчислювати без округлення, з необмеженою кількістю розрядів. ЕОМ має обмежену кількість розрядів і похибки округлення є майже завжди. Аналіз похибки обчислень за формулою прямого ходу показує, що похибка тим менше, чим менший множник $R=a_{ik}/a_{kk}$. Щоб зробити множник як можна меншим, треба щоб a_{kk} було як найбільшим. Серед елементів a_{ik} $(i\geq k)$ треба шукати не просто такий, що не дорівнює нулю, а максимальний за модулем (його називають головним) і перестановкою рівнянь помістити його на діагональ. Іншими словами, при перестановці рівнянь необхідно добитися того, щоб $|a_{kk}| \geq |a_{ik}|$. Тоді множник $|R| \leq 1$.

Описаний спосіб розв'язування СЛАР називають *методом Гауса з вибором головного елемента* за стовпцем, або з частковим упорядкуванням.

Алгоритм цього методу складається з наступних кроків:

1) встановлюємо $k=1$;

2) серед піддіагональних елементів k-*го* стовпця та a_{kk} знайти максимальний за модулем елемент $a_{hk}=max\ |a_{ik}|,\ (i\geq k)$. *Я*кщо $k\neq h$, переставити k-*й* та h-*й* рядки матриці A та вектора правої частини B;

3) за допомогою лінійних комбінацій рядка k та рядків $k+1,k+2,...,n$ виключити піддіагональні елементи k-*го* стовпця $a_{k+1},a_{k+2},\ldots,a_{nk}$;

4) встановити $k = k +1$. *Я*кщо $k<n-1$, перейти до п.2. *Я*кщо $k=n$, перейти до п.5;

5) якщо $a_{nn}=0$, то матриця вироджена і розв'язування припиняється; якщо $a_{nn}\neq 0$, то виконується зворотний хід.

*Я*к же впливає вибір головного елемента на точність результатів? Проілюструємо це на прикладі системи двох рівнянь з двома невідомими **[2]:**

$$\begin{pmatrix} -10^{-5} & 1 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} X_1 \\ X_2 \end{pmatrix}=\begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad (6.9)$$

з точним розв'язком $X_1=-0{,}499975,\ X_2=0{,}999995$ (наведені перші шість значущих цифр) або, приблизно, $X_1=-0{,}5,\ X_2=1$. Проаналізуємо, як буде розв'язуватися ця задача на деякій ЕОМ з чотирма десятковими розрядами, тобто на ЕОМ, числа якої зберігаються в формі $0.nnnn\cdot 10^P$, де n – цифри мантиси, P – порядок.

Процедуру Гауса починають з обчислення R:

$$R=\frac{a_{21}}{a_{11}}=\frac{2}{-10^{-5}}=-\frac{0.2\cdot 10}{0.1\cdot 10^{-4}}=-0.2\cdot 10^6.$$

Округлення при цьому не виникає. Віднімаючи з другого рівняння перше, помножене на R, перетворимо в нуль елемент a_{21}, а нове значення a_{22} дорівнюватиме:

$$a_{22}^{(1)}=1-(-0.2\cdot 10^6)\cdot 1=0.1\cdot 10^1-(-0.2\cdot 10^6)(0.1\cdot 10^1)=0.1\cdot 10^1+0.2\cdot 10^6=0.200001\cdot 10^6$$

Оскільки дію виконують на чотирирозрядній ЕОМ, то число округлюють і беруть таким, що дорівнює $0.2\cdot 10^6$. При обчислюванні нового значення b_2 похибок не виникає:

$$b_2 = 0 - (-0.2 \cdot 10^6)(0.1 \cdot 10^1) = 0.2 \cdot 10^6$$

В результаті система (6.9) зводиться до вигляд:

$$\begin{pmatrix} -10^{-5} & 1 \\ 0 & 0.2 \cdot 10^6 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.2 \cdot 10^6 \end{pmatrix}$$

При знаходженні x_1 та x_2 з цієї системи також не виникає похибок округлення:

$$X_2 = \frac{b_2^{(1)}}{a_{22}^{(11)}} = \frac{0.2 \cdot 10^6}{0.2 \cdot 10^6} = 1;$$

$$X_1 = \frac{0.1 \cdot 10^1 - 0.1 \cdot 10^1}{-0.1 \cdot 10^{-4}} = 0.$$

Одержаний розв'язок ***(0;1)*** значно відрізняється від точного ***(-0,5;1).*** Причиною є похибка округлення, допущена при обчислюванні $a_{22}^{(1)}$. Як змогла похибка в 6-му десятковому знаці призвести до катастрофічно неправильного розв'язку? Скористуємося принципом зворотного аналізу похибок, який полягає не в з'ясуванні того, яка допущена похибка, а в пошуку задачі, яка в дійсності розв'язувалась.

Відкинута при обчислюванні $a_{22}^{(1)}$ величина 0,000·001 10 є a_{22} вихідної матриці. Тому одержаний нами розв'язок був би таким самим, якщо б a_{22} дорівнювало нулю. Тобто в дійсності знайдено точний розв'язок системи

$$\begin{pmatrix} 2 & 1 \\ -10^{-5} & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Аналізуючи процес обчислення $a_{22}^{(1)}$, доходимо висновку, що причина у значній величині множника R, який не дозволив a_{22} включитися в загальну суму. Множник R одержали великим, оскільки діагональний елемент a_{11} малий порівняно з a_{21}. Це дозволяє надіятись, що застосувавши процедуру вибору головного елемента за стовпцем, ми одержимо більш точний розв'язок.

Поміняємо в системі (6.9) рівняння місцями. В результаті максимальне з чисел першого стовпця опиниться на діагоналі:

$$\begin{pmatrix} 2 & 1 \\ -10^{-5} & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Виконаємо дії алгоритму Гауса, відзначивши, що похибка округлення допущена, як і раніше, тільки при обчисленні $a_{22}^{(1)}$:

$$R = \frac{-0.1\cdot 10^{-4}}{0.2\cdot 10^{1}} = -0.5\cdot 10^{-5};$$

$$a_{22}^{(1)} = 0.1\cdot 10^{1} - \left(-0.5\cdot 10^{-5}\right)\left(0.1\cdot 10^{1}\right) = 0.1000005\cdot 10^{1} \approx 0.1\cdot 10^{1} = 1;$$

$$b_{2}^{(1)} = 0.1\cdot 10^{1} - (-0.5\cdot 10^{-5})\cdot 0 = 0.1\cdot 10^{1}$$

$$X_2 = \frac{b_2^{(1)}}{a_{22}^{(1)}} = \frac{0.1\cdot 10^{1}}{0.1\cdot 10^{1}} = 1;$$

$$X_1 = \frac{1}{a_{11}}(b_1 - a_{12}\cdot X_2) = -\frac{0.1\cdot 10^{1}}{0.2\cdot 10^{1}} = -0.5$$

Одержаний розв'язок майже збігається з точним. Розглянутий метод Гауса з стратегією часткового упорядкування досить надійний і широко застосовується на практиці. Але в деяких випадках він дає незадовільний результат. Наприклад, помноживши перше рівняння в (6.9) на -10^6, одержимо систему

$$\begin{pmatrix} 10 & -10^{6} \\ 2 & 1 \end{pmatrix}\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} -10^{6} \\ 0 \end{pmatrix}$$

яка повинна мати такий самий розв'язок, як і (6.9).

Розв'яжемо її методом Гауса з вибором головного елемента за стовпцем. В першому стовпці максимальний елемент знаходиться на діагоналі, тому рівняння не переставляють.

$$R = \frac{2}{10} = 0.2; a_{22}^{(1)} = 1 - (-10^{6})\cdot 0.2 = 1 + 0.2\cdot 10^{6} \approx 0.2\cdot 10^{6};$$

$$b_{2}^{(1)} = 0 - (-10^{6})\cdot 0.2 = 0.2\cdot 10^{6};$$

$$X_2 = \frac{b_2^{(1)}}{a_{22}^{(1)}} = \frac{0.2\cdot 10^{6}}{0.2\cdot 10^{6}} = 1;$$

$$X_1 = \frac{1}{a_{11}}(b_1 - a_{12}\cdot X_2) = \frac{1}{10}(-10^{6} - (-10^{6})\cdot 1) = 0.$$

Отже, використовуючи часткове упорядкування, ми знову одержали неправильний розв'язок. Процедура вибору головного елемента за стовпцем дає непоганий результат, якщо матриця системи зрівноважена, тобто

максимальні елементи в кожному стовпці і в кожному рядку мають близькі порядки. Проте, досить простої процедури такого масштабування немає [2].

Виходом з цієї ситуації є повне упорядкування, тобто вибір головного елемента по всьому полю. Для реалізації запропонованого алгоритму спочатку створимо допоміжний модуль, який буде мати дві функції: swapRows – для переміщення рядків матриці та swapCols – для переміщення стовпчиків. Реалізація цього допоміжного модуля представлена на рис. 6.3.

```
# -*- coding: cp1251 -*-
## module swap
''' swapRows(v,i,j).
    Міняє містами рядки і та j вектора або матриці [v].

    swapCols(v,i,j).
    Міняє містами стовпчики і та j матриці [v].
'''
def swapRows(v,i,j):
    if len(v.getshape()) == 1: v[i],v[j] = v[j],v[i]
    else:
        temp = v[i].copy()
        v[i] = v[j]
        v[j] = temp

def swapCols(v,i,j):
    temp = v[:,j].copy()
    v[:,j] = v[:,i]
    v[:,i] = temp
```

Рис. 6.3. Реалізація допоміжного модуля

На рис. 6.4 представлена реалізація запропонованого алгоритму у вигляді модуля мови *Python* з використанням функцій допоміжного модуля.

```
# -*- coding: cp1251 -*-
## module gaussPivot
''' x = gaussPivot(a,b,tol=1.0e-9).
    Вирішує [a]{x} = {b} методом Гауса
    з частковим упорядкуванням
'''
from numpy import *
import swap
import error

def gaussPivot(a,b,tol=1.0e-12):
    n = len(b)
    s = zeros((n),dtype=float64)
    for i in range(n):
        s[i] = max(abs(a[i,:]))

    for k in range(0,n-1):

      # Перестановка рядків, якщо потрібна
        p = int(argmax(abs(a[k:n,k])/s[k:n])) + k
        if abs(a[p,k]) < tol: error.err('Матриця сингулярна')
        if p != k:
            swap.swapRows(b,k,p)
            swap.swapRows(s,k,p)
            swap.swapRows(a,k,p)

      # Фаза виключення
```

```
    for i in range(k+1,n):
      if a[i,k] != 0.0:
        lam = a[i,k]/a[k,k]
        a[i,k+1:n] = a [i,k+1:n] - lam*a[k,k+1:n]
        b[i] = b[i] - lam*b[k]
  if abs(a[n-1,n-1]) < tol: error.err('Матриця сингулярна')

 # Зворотній хід
  for k in range(n-1,-1,-1):
    b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
  return b
```

Рис. 6.4. Реалізація модуля алгоритму

Створений модуль складається з однієї функції, яка і реалізує алгоритм. На вхід функції gaussPivot подаються матриця ***A*** та вектор-стовпчик ***B***.

6.2.4. Метод Гауса з вибором головного елемента по всьому полю

В п.6.2.3 ми показали, що часткове упорядкування не забезпечило прийнятних результатів для системи

$$\begin{pmatrix} 10 & -10^6 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} -10^6 \\ 0 \end{pmatrix} \qquad (6.10)$$

еквівалентної системі (6.9) з розв'язком ***(- 0,5;1)***.

Перевіримо, що дає нам повне упорядкування. Для цього зробимо пошук головного елемента по всьому полю матриці. Очевидно, максимальним за модулем є елемент $a_{12}=-10^6$. Розмістимо його на діагональ. Для цього треба переставити місцями стовпці матриці. Оскільки кожний стовпець відповідає конкретному x_j, то треба переставити і компоненти вектора ***X***. В результаті одержимо систему

$$\begin{pmatrix} 10 & -10^6 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} X_2 \\ X_1 \end{pmatrix} = \begin{pmatrix} -10^6 \\ 0 \end{pmatrix} \qquad (6.11)$$

Розв'яжемо її методом Гауса:

$$R = \frac{1}{-10^6} = -10^{-6};\ \ a_{22}^{(1)} = 2 - 10 \cdot (-10^{-6}) = 2 + 10^{-5} \approx 2;$$

$$b_2^{(1)} = 0 - (-10^{-6}) = -1 \qquad (6.12)$$

$$X_1 = \frac{b_2^{(1)}}{a_{22}^{(1)}} = \frac{-1}{2} = -0.5;$$

$$X2 = \frac{1}{a_{11}}(b_1 - a_{12} \cdot X_2) = \frac{1}{-10^6}(-10^6 - 10 \cdot (-0.5)) = \frac{-10^6 + 5}{-10^6} \approx 1.$$

Одержано розв'язок, майже такий, що збігається з точним. Проте стійкість цього алгоритму до похибок округлення призводить до ускладнення програми та збільшення часу обчислення. Він відрізняється від попереднього алгоритму тим, що на ***k****-му* кроці (перед перетворенням у нуль елементів ***k****-го* стовпця) головний елемент відшуковується на правій нижній субматриці матриці ***A***, яку на рис. 6.5 показано подвійним штрихуванням.

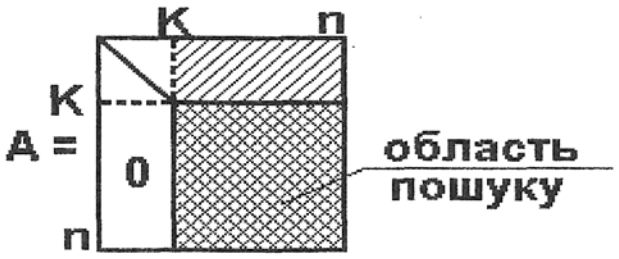


Рис. 6.5. Область пошуку головного елемента в алгоритмі повного упорядкування

Ця субматриця включає елементи a_{ij}, для яких $i,j>k$. Крім того, визначивши головний елемент a_{hp}, необхідно переставити місцями не тільки рядки матриці ***A*** з номерами ***k*** і ***h*** та компоненти b_k та b_h вектора правої частини, але і стовпці матриці ***A*** з номерами ***k*** та ***p***, а також урахувати цю перестановку у векторі невідомих ***X***.

Алгоритм методу Гауса з вибором головного елемента по всьому полю наведено нижче:

1.Встановити $k=1$; сформувати масив перестановок $MP(i)=i,i=1,...,n$.

2. Серед елементів a_{ii}; $(i,j\geq k)$ знайти максимальний за модулем

$$a_{hp} = \max\left|a_{ij}\right|, i, j \geq k$$

3. Якщо $h \neq k$, поміняти місцями рядки k та h матриці та елементи з номерами k і h вектора B.

4. Якщо $k \neq p$, поміняти місцями стовпці k та p матриці A, а також елементи з номерами k і p масиву перестановок стовпців MP.

5. Виключити піддіагональні елементи k-*го* стовпця матриці A за допомогою лінійної комбінації рядків.

6. Встановити $k=k+1$. Якщо $k \leq n-1$, перейти до п.2.

7. Виконати зворотний хід і одержати вектор невідомих Z.

8. За допомогою масиву перестановок MP сформувати за вектором Z масив X.

9. Обчислити вектор нев'язок r та надрукувати вектора r і X.

Навіщо потрібен масив MP? Нехай, наприклад, розв'язуємо СЛАР з чотирма невідомими, точний розв'язок якої $x_1=7, x_2=5, x_3=-9, x_4=-2$.

Нехай в процесі виконання процедури Гауса з вибором головного елемента по всьому полю ми тільки один раз переставляємо стовпці матриці A, наприклад, *2-й* та *4-й*. Тоді, виконавши зворотний хід, одержимо деякий вектор

	1	2	3	4
Z=	7	-2	-9	5

в якому порівняно з вектором невідомих X переставлені *2-й* і *4-й* елементи. Тобто одержимо неправильну відповідь. Саме для запам'ятовування проведених перестановок стовпців матриці і потрібен масив MP. У даному випадку нашої системи з чотирьох рівнянь він спочатку має вигляд

	1	2	3	4
MP=	1	2	3	4

Одночасно з перестановкою *2-го* та *4-го* стовпців переставляються *2-й* та *4-й* елементи цього масиву:

	1	2	3	4
MP=	1	4	3	2

Величина ***MP(i)*** означає дійсний номер у векторі невідомих ***X*** числа ***Z(i),*** одержаного після зворотного ходу. Використовуючи інформацію про перестановки, що знаходиться в масиві ***MP,*** перед друкуванням кінцевого результату треба зробити зворотну перестановку одержаного масиву ***Z*** за допомогою операторів:

…

```
for i in range(N):
    X[MP[i]]= z [ i ]                                    (6.13)
```

…

У випадку нашої системи виконують такі дії:

X(MP(1)) = X(1) ← Z(1) = 7,

X(MP)2)) = X(4) ← Z(2) = -2,

X(MP(3)) = X(3) ← Z(3) = -9,

X(MP(4)) = X(2) ← Z(4) = 5,

та, оскільки елементи масиву ***X*** друкуються в порядку зростання номерів елементів, отримаємо правильну відповідь.

Таким чином, в процесі зворотного ходу необхідно визначити масив ***Z***, після чого за допомогою операторів (6.13) знайти масив ***X***.

Відмітимо, що час роботи програми можна суттєво скоротити, якщо не робити насправді перестановку рядків та стовпчиків матриці, а запам'ятовувати ці перестановки за допомогою масиву ***MP*** і ще одного аналогічного масиву для рядків. Хоча такий підхід буде ще більше ускладнювати програму.

6.2.5. LU-алгоритм

Розглянуті вище реалізації методу Гауса (виключення за стовпцями та рядками) не вичерпують усіх можливих варіантів. Існує цілий ряд інших

алгоритмів, що базуються на тій самій ідеї виключення невідомих шляхом лінійної комбінації рівнянь. Сюди відносять, наприклад, методи ***LU***-перетворення, Жордано, оптимального виключення, прогонки, які можна розглядати як модифікації методу Гауса.

Після класичного методу Гауса з реалізацією за стовпцями значне місце в обчислювальній практиці отримав ***LU***-алгоритм.

Нехай ми вміємо перетворювати матрицю ***A*** в добуток двох трикутних матриць: нижньої трикутної ***L***, на діагоналі якої розташовані одиниці ($l_{ii}=1$), й верхньої трикутної ***U***, тобто

$$\mathbf{A = LU} \tag{6.14}$$

Тоді вихідну систему ***Ax=B*** записують у вигляді

$$\mathbf{LUx = B} \tag{6.15}$$

Якщо позначити вектор ***Ux*** через ***y*** (***Ux=y***), остання система набиратиме вигляду

$$\mathbf{Ly = B} \tag{6.16}$$

Оскільки її матриця є нижньою трикутною, то розв'язок такої СЛАР здійснюють просто (див. рис. 6.6).

```
…
X[0]=B[0]/A[0][0]
for i in range(1,N+1):
    s=0
    for j in range(I):
        s+=A[i][j]*X[j]
        X[i]=(B[i]-s)/A[i][i]
…
```

Рис. 6.6. Фрагмент програми вирішення СЛАР

Коли ми вже знайшли в результаті розв'язування цієї системи вектор ***y***, тепер можемо розв'язати систему

$$\mathbf{Ux = y} \tag{6.17}$$

з верхньою трикутною матрицею ***U*** (див. рис. 6.7) і отримати шуканий вектор ***X***.

```
…
X[N-1]=B[N-1]/A[N-1][N-1]
for i in range(N-1,-1,-1):
    s=0
    for j in range(I+1,N+1):
        s+=A[i][j]*X[j]
        X[i]=(B[i]-s)/A[i][i]
…
```

Рис. 6.7. Фрагмент програми вирішення СЛАР

Таким чином, ***LU***-алгоритм має вигляд:

1. Розкласти матрицю ***A*** на добуток верхньої та нижньої матриць ***L*** й ***U*** (***LU***-факторизація).
2. Розв'язати систему (6.16) з нижньою трикутною матрицею (прямий хід).
3. Розв'язати систему (6.17) з верхньою трикутною матрицею (зворотний хід).

Переваги такого підходу особливо очевидні, якщо необхідно розв'язати декілька СЛАР з однією і тією самою матрицею ***A*** й різними векторами ***B***. У цьому випадку ***A*** = ***LU*** розкладується тільки один раз і для кожної системи потрібно виконати тільки прямий і зворотній ходи. Кількість необхідних арифметичних операцій для всього ***LU***-алгоритму є такою самою, як і для методу Гауса, але більша їхня частина припадає на перший етап – ***LU***-факторизацію, і тому одноразове використання цього етапу під час розв'язування серії систем дозволяє суттєво зменшити витрати часу ЕОМ.

Як саме виконати ***LU***-факторизацію? Виявляється, що матриця ***U*** збігається з верхньою трикутною матрицею, яку отримуємо звичайним методом Гауса, а коефіцієнти ***R*** цього методу утворюють матрицю ***L***. І тому для розкладу матриці ***A*** достатньо запам'ятати ці коефіцієнти й, оскільки

клітини нижньої лівої частини матриці A переходять у нульові й не використовуються, саме сюди можна вписувати ці коефіцієнти.

Таким чином, для проведення LU-факторизації не обов'язково використовувати додаткові масиви, а матриці L й U зберігати на полі матриці A, причому діагональні одиниці матриці L взагалі не зберігаються.

Наведемо, як приклад, LU-факторизацію такої матриці:

$$A = \begin{pmatrix} 2 & 7 & 5 \\ 4 & 4 & 3 \\ 6 & 8 & 9 \end{pmatrix}.$$

Будемо використовувати процедуру Гауса з реалізацією за стовпцями й без вибору головного елемента.

Спочатку розглянемо піддіагональні елементи першого стовпчика. Для елемента a_{21} множник R дорівнює

$$R = \frac{a_{21}}{a_{11}} = \frac{4}{2} = 2 = l_{21}.$$

Оскільки елемент a_{21} більше не потрібен, то на його місце можна записати отриманий множник. Починаючи з наступного за діагональним елементом першого рядка, помножимо його елементи на R та віднімемо з другого рядка, як і в звичайному алгоритмі Гауса:

$$a_{22}^{(1)} = a_{22} - a_{12} \cdot R = 4 - 7 \cdot 2 = -10,$$
$$a_{23}^{(1)} = a_{23} - a_{13} \cdot R = 3 - 5 \cdot 2 = -7$$

Ці обидва числа, як і завжди, вносять на місце елементів a_{22}, a_{23}. Тепер необхідно обчислити множник для елемента a_{31}:

$$R = \frac{a_{31}}{a_{11}} = \frac{6}{2} = 3 = l_{31}$$

й внести його замість елемента a_{31}.

Перший рядок, починаючи з другого елемента, множимо на цей множник і віднімаємо з третього рядка, замість якого і вноситься

$$a_{32}^{(1)} = a_{32} - a_{12} \cdot R = 8 - 7 \cdot 3 = -13,$$
$$a_{33}^{(1)} = a_{33} - a_{12} 3 \cdot R = 9 - 5 \cdot 3 = -6.$$

У загальному випадку всі ці перетворення виконуються для другого та усіх наступних стовпчиків матриці, крім останнього. Для нашої матриці, яка тепер набрала вигляду

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ l_{21} & a_{22}^{(1)} & a_{23}^{(1)} \\ l_{31} & a_{32}^{(1)} & a_{33}^{(1)} \end{pmatrix} = \begin{pmatrix} 2 & 7 & 5 \\ 2 & -10 & -7 \\ 3 & -13 & -6 \end{pmatrix}$$

знайдемо множник для елемента $a_{32}^{(1)}$

$$R = \frac{a_{32}^{(1)}}{a_{22}^{(1)}} = \frac{-13}{-10} = \frac{13}{10} = l_{32}.$$

Помножимо елементи другого рядка, що розташовані після діагонального, на ***R*** і віднімемо з третього рядка:

$$a_{33}^{/(2)} = a_{33}^{/(1)} - a_{23}^{/(1)} \cdot R = -6 - (-7) \cdot \frac{3}{10} = \frac{31}{10}.$$

В результаті на полі вихідного масиву ***A*** зберігаються тепер елементи матриць ***L*** та ***U***:

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{pmatrix} = \begin{pmatrix} 2 & 7 & 5 \\ 2 & -10 & 7 \\ 3 & \frac{13}{10} & \frac{31}{10} \end{pmatrix}.$$

Перевіримо, що наведена рівність справедлива. Для цього виконаємо множення двох отриманих матриць:

$a_{11}=l_{11}U_{11}+l_{12}U_{21}+l_{13}U_{31}=1\cdot 2 + 0\cdot 0 + 0\cdot 0 = 2,$

$a_{12}=l_{11}U_{12}+l_{12}U_{22} + l_{13}U_{32}=1\cdot 7 + 0\cdot (-10) + 0\cdot 0 = 7,$

$a_{13} = l_{11}U_{13} + l_{12}U_{23} + l_{13}U_{33} = 1\cdot 5 + 0\cdot (-7) + 0 \cdot \frac{31}{10} = 5,$

$a_{21} = I_{21}U_{11} + l_{22}U_{21} + l_{23}U_{31} = 2\cdot 2 + 1\cdot 0 + 0\cdot 0 = 4,$

$a_{22} = l_{21}U_{12} + I_{22}U_{22} + l_{23}U_{32} = 2\cdot 7 + 1 \cdot (-10) + 0\cdot 0 = 4,$

$a_{23}= I_{21}\ U_{13}+ l_{22}U_{23} + l_{23}U_{33} = 2\cdot 5 + 1 \cdot (-7) + 0 \cdot \frac{31}{10} = 3,$

$a_{31}=l_{31}U_{11}+l_{32}U_{21}+l_{33}U_{31}=3\cdot 2 + \frac{13}{10} \cdot 0 + 1\cdot 0 = 6,$

$a_{32}= l_{32}U_{12}+I_{32}\ U_{22}+I_{33}U_{32}=3\cdot 7 + \frac{13}{10} \cdot (-10) + 1\cdot 0 = 8,$

$$a_{33} = l_{31}U_{13} + l_{32}U_{23} + l_{33}U_{33} = 3 \cdot 5 + \frac{13}{10} \cdot (-7) + 1 \cdot \frac{31}{10} = 9.$$

Дійсно, ми отримали елементи вихідної матриці. Алгоритм ***LU***-факторизації можна сформулювати таким чином:

1. встановити $k=1$;
2. встановити $i=к+1$;
3. знайти $R= a_{ik} / a_{кк}$ та занести його замість елемента $a_{iк}$;
4. встановити $j=k+1$;
5. знайти $a_{ij} = a_{ij} - a_{kj}R$;
6. $j = j + 1$. Якщо $j \le n$, повернення до п. 5);
7. $i = i + 1$. Якщо $i \le n$, повернення до п. 3);
8. $k = k + 1$. Якщо $k \le n$, повернення до п. 2).

Можлива інша, наприклад, порядкова реалізація ***LU***-факторизації. Як і для класичного методу Гауса, тут необхідно проводити вибір головного елемента в тій або іншій формі. Реалізація алгоритму у вигляді модуля на мові *Python* представлена на рис. 6.8.

```
# -*- coding: cp1251 -*-

## module LUdecomp

''' a = LUdecomp(a).

    LU факторизація: [L][U] = [a]. Матриця, яка вертається [a] = [L\U]

    містить [U] в верхньому трикутнику і недіагональні елементи

    з [L] в нижньому трикутнику.


    x = LUsolve(a,b).

    Вирішує [L][U]{x} = b, де [a] = [L\U] матриця з LUdecomp.

'''

from numarray import dot


def LUdecomp(a):
```

```
    n = len(a)
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a [i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                a[i,k] = lam
    return a


def LUsolve(a,b):
    n = len(a)
    for k in range(1,n):
        b[k] = b[k] - dot(a[k,0:k],b[0:k])
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Рис. 6.8. Фрагмент програми вирішення СЛАР

Модуль складається з двох функцій: LUdecomp – функції, яка виконує ***LU***-факторизацію, та LUsolve – функції, що вирішує СЛАР. На вхід першої функції подається тільки один параметр – матриця ***A***. На вхід другої функції подаються матриця ***A*** та вектор-стовпчик ***B***.

6.2.6. Метод Жордано

Пропонована модифікація методу Гауса зводить матрицю системи не до трикутної, а до діагональної форми (рис. 6.9).

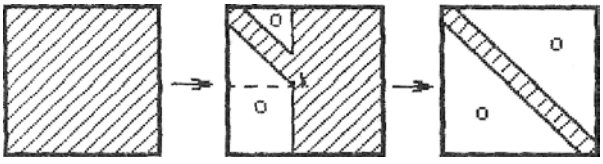


Рис. 6.9. Зміна ненульової структури матриці методом Жордано

При цьому перед виконанням ***k**-го* кроку (обробкою ***k**-го* стовпчика матриці) перші ***(k-1)*** стовпчики матриці є нульовими, окрім діагональних елементів. У процесі ***k**-го* кроку переходять у нуль не тільки піддіагональні елементи ***k**-го* стовпчика, але й його елементи, розташовані над діагоналлю. Виконують все це тими ж самими способами, що і в методі Гауса – відніманням ***k**-го* рядку від інших рядків. Обчислювальна схема методу Жордано для розв'язування СЛАР така:

1) встановити $k=1$;

2) встановити $i=1$;

3) якщо $i=k$, перехід до п.8;

4) знайти $R=a_{ik}/a_{kk}$;

5) встановити $j=k$;

6) знайти $a_{ij}= a_{ij} - a_{kj}\cdot R$, $b_j = b_j - b_k\cdot R$;

7) $j=j+1$. Якщо $j \le n$, повернення до п.6;

8) $i=i+1$. Якщо $i \le n$, повернення до п.3;

9) $k=k+1$. Якщо $k \le n$, повернення до п.2;

10) знайти вектор невідомих $x_i = b_i/a_{ij}$, $i=1,2,\ldots, n$.

Як і попередні методи, на практиці метод Жордано потрібно використовувати з вибором головного елемента. Кількість необхідних обчислювальних операцій у цьому методі більше, ніж у методі Гауса та становить близько n^3. У зв'язку з цим метод Жордано використовують, в основному, тільки для обернення матриць та у задачах лінійного програмування. На рис. 6.10 представлена реалізація методу Жордано в якості функції, яка дозволяє знайти обернену матрицю.

```
# -*- coding: cp1251 -*-

## Module gaussJordan

''' a_inverse = gaussJordan(a).

    Інвертує матрицю 'a' методом Жордано.

'''

def gaussJordan(a):
```

```
    n = len(a)
    for i in range(n):
        temp = a[i,i]
        a[i,i] = 1.0
        a[i,0:n] = a[i,0:n]/temp
        for k in range(n):
            if k != i:
                temp = a[k,i]
                a[k,i] = 0.0
                a[k,0:n] = a[k,0:n] - a[i,0:n]*temp
    return a
```

Рис. 6.10. Реалізація методу Жордано

Модуль складається з однієї функції: gaussJordan – функції, яка виконує пошук оберненої матриці методом Жордано. На функції подається тільки один параметр – матриця A.

6.2.7. Метод оптимального виключення

Як і метод Жордано, даний метод призводить матрицю системи до діагональної форми, але послідовність виключення невідомих тут інша. Його особливістю є те, що в оперативній пам'яті можна зберігати тільки частину матриці, тобто вводити її поступово із зовнішньої пам'яті. Метод використовувався на ранніх етапах розвитку обчислювальної техніки, а зараз втратив свою актуальність. Процес обробки матриці цим методом показано на рис. 6.11.

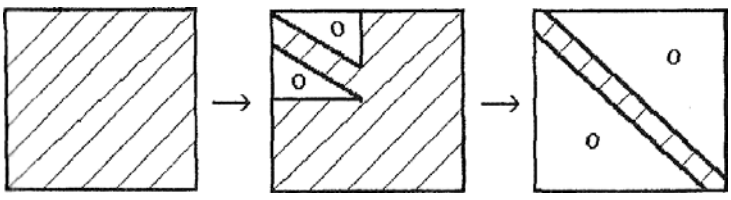


Рис. 6.11. Зміна ненульової структури матриці методом оптимального виключення

6.2.8. Метод прогонки

На практиці часто виникає необхідність розв'язування СЛАР з тридіагональною матрицею. Запишемо таку систему у вигляді

$$\begin{aligned}
b_1x_1 + c_1x_2 &= d_1,\\
a_2x_1 + b_2x_2 + c_2c_3 &= d_2,\\
a_3x_2 + b_3x_3 + c_3x_4 &= d_3,\\
&\dots\\
a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n &= d_{n-1},\\
a_nx_{n-1} + b_nx_n &= d_n
\end{aligned} \qquad (6.18)$$

До розв'язування таких систем може приводити, наприклад, метод скінченних різниць при його використанні для розв'язування крайових задач для диференціальних рівнянь.

Систему вигляду (6.18) можна розв'язати звичайним методом Гауса, враховуючи структуру її матриці. У процесі прямого ходу для ***k = 1,....,(n-1)*** виконуються такі дії:

$$R = \frac{a_{k+1}}{b_k}, \quad b_{k+1}^{(1)} = b_{k+1} - c_kR, \quad d_{k+1}^{(1)} = d_{k+1} - d_kR$$

що вимагає ***3(n-1)*** арифметичних операцій. При зворотному ході обчислюють вектор невідомих ***x*** за формулами:

$$x = \frac{d_n^{(1)}}{b_n^{(1)}}, \quad x_i = \left(d_i^{(1)} - c_ix_{i-1}\right)\cdot\frac{1}{b_i^{(1)}}, \quad i = n-1,\dots,1.$$

що вимагає ***3(n-1)+1*** арифметичних операцій. Всього для розв'язування методом Гауса такої СЛАР необхідно ***8n-7*** арифметичних операцій на відміну від ***2/3n^3*** операцій, потрібних для розв'язування довільної системи за допомогою цього методу.

У практиці обчислень часто використовують для розв'язування СЛАР вигляду (6.18) не метод Гауса, а його модифікацію, яку називають методом прогонки [1]. За кількістю необхідних операцій цей метод еквівалентний методу Гауса. Метод прогонки виконують у два етапи, аналогічних прямому та зворотному ходам методу Гауса.

На першому етапі (пряма прогонка) елементи $\mathbf{x_i(i=1,\ldots,\ n-1)}$ вектора невідомих виражають через $\mathbf{x_{i+1}}$ за допомогою прогонних коефіцієнтів $\mathbf{A_i}$, $\mathbf{B_i}$:

$$x_i = A_i x_{i+1} + B_i,\ i = 1,2,\ldots,n-1 \qquad (6.19)$$

Мета першого етапу – обчислення цих коефіцієнтів. З першого рівняння системи (6.19) знайдемо $\boldsymbol{x_1}$:

$$x_1 = -\frac{c_1}{b_1}\cdot x_2 + \frac{d_1}{b_1}$$

тобто

$$A_1 = -\frac{c_1}{b_1},\ B_1 = \frac{d_1}{b_1}$$

Підставимо ці співвідношення в друге рівняння й виразимо $\boldsymbol{x_2}$ через $\boldsymbol{x_3}$:

$$a_2(A_1x_2 + B_1) + b_2x_2 + c_2x_3 = d_2,$$
$$x_2(a_2A_1 + b_2) + c_2x_3 + a_2B_1 = d_2,$$
$$x_2 = \frac{d_2 - c_2x_3 - a_2B_1}{a_2A_1 + b_2} = -\frac{c_2}{a_2A_1 + b_2}\cdot x_3 + \frac{d_2 - a_2B_1}{a_2A_1 + b_2} = A_2x_3 + B_2.$$

Аналогічно можна отримати інші коефіцієнти:

$$A_i = -\frac{c_i}{e_i},\quad B_i = -\frac{d_i - a_iB_{i-1}}{e_i},\quad e_i = a_iA_{i-1} + b_i,\quad i = 2,3,\ldots,n-1.$$

Останнє з цих співвідношень одержують із передостаннього рівняння. Дійсно,

$$a_{n-1}(A_{n-2}x_{n-1} - B_{n-2}) + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1},$$
$$x_{n-1}(a_{n-1}A_{n-2} + b_{n-1}) + c_{n-1}x_n + a_{n-1}B_{n-2} - d_{n-1},$$
$$x_{n-1} = -\frac{c_{n-1}}{a_{n-1}A_{n-2} + b_{n-1}}x_n + \frac{d_{n-1} - a_{n-1}B_{n-2}}{a_{n-1}A_{n-2} + b_{n-1}} = A_{n-1}x_n + B_{n-1}.$$

Продовжуючи такий процес, підставимо $\boldsymbol{x}_{n-1}$ в останнє рівняння:

$$a_n(A_{n-1}x_n + B_{n-1}) + b_nx_n = d_n,\quad x_n(a_nA_{n-1} + b_n) = d_n - a_nB_{n-1}.$$

Одержаний вираз дає можливість знайти $\boldsymbol{x_n}$. При цьому починається другий етап – зворотня прогонка:

$$x_n = \frac{d_n - a_nB_{n-1}}{a_nA_{n-1} + b_n}.$$

Інші елементи вектора невідомих знаходяться в процесі зворотньої прогонки за формулою (6.19). Таким чином, алгоритм методу прогонки для розв'язування СЛАР з тридіагональною матрицею набирає вигляду:

1. Реалізувати пряму прогонку:

а) знайти $A_1 = -c_{-1} / b_1$, $B_1 = d_1 / b_1$;

б) для $i = 2, 3, \ldots\ldots n-1$ знайти:

$$e_i = a_i A_{i-1} + b_i, \quad A_i = -\frac{c_i}{e_i}, \quad B_i = \frac{d_i - a_i B_{i-1}}{e_i}.$$

2. Реалізувати зворотну прогонку:

а) знайти $x_n = \dfrac{d_n - a_n B_{n-1}}{b_n + a_n A_{n-1}}$

б) для $i = n-1, .., 1$ знайти $x_i = A_i x_{i+1} + B_i$.

Коефіцієнти A_i, B_i при програмній реалізації методу можна запам'ятовувати на місці елементів b_i, c_i матриці A.

Зазначимо, що алгоритм включає операцію ділення, тому необхідно накласти додаткові умови на елементи матриці системи (6.18), щоб виключити ділення на нуль і забезпечити стійкість методу щодо помилок округлення. Ці умови полягають у тому, що матриця повинна бути діагонально домінуючою, тобто $|b_i| \geq |a_i| + |c_i|$, причому хоча б для одного значення i повинна виконуватися строга нерівність [1]. Ця умова стійкості методу прогонки є достатньою, але не є необхідною, тобто в багатьох випадках цей метод стійкий навіть при порушенні умови переваження діагональних елементів. Зазначимо, що у більшості практичних випадків ці умови виконуються, і метод прогонки можна реалізувати без вибору головного елемента. Реалізація методу прогонки представлена на рис. 6.12.

```
# -*- coding: cp1251 -*-
def progonka(a,b):
    N=len(b)
    b[0]/=a[0][0]
    a[0][1]/=-a[0][0]
```

```
for i in xrange(1,N):
    znam=-a[i][i]-a[i][i-1]*a[i-1][i]
    a[i][i+1]/=znam
    b[i]=(a[i][i-1]*b[i-1]-b[i])/znam

b[N-1]=(a[N-1][N-2]*b[N-2]-b[N-1])/(-a[N-1][N-1]- \
      a[N-1][N-2]*a[N-2][N-1])

#зворотній хід
for i in xrange(N-1,-1,-1):
    b[i]+=b[i+1]*a[i][i+1]
```

Рис. 6.12. Реалізація методу прогонки

Модуль складається з однієї функції: progonka – функції, що вирішує СЛАР методом прогонки. На вхід функції подаються два параметри: матриця ***A*** та вектор-стовпчик ***B***.

6.2.9. Погано обумовлені системи

Раніше було встановлено, що ефективність розв'язування задачі, а саме, точність отриманого результату і кількість необхідних операцій, суттєво залежить від методу розв'язування. Наприклад, вибір головного елемента забезпечує більш високу точність при фіксованій довжині розрядної сітки ЕОМ. Чи впливають на точність результату при фіксованому методі властивості самої системи рівнянь? Очевидно, із зростанням розміру системи зростає й кількість необхідних операцій. При цьому помилки округлення стають більш суттєвими. Але й для систем однакового розміру будемо отримувати різні похибки.

Існують так звані погано обумовлені СЛАР, розв'язування яких є важким взагалі будь-яким методом, а в більшості випадків не має сенсу шукати розв'язок [2].

Систему лінійних рівнянь називають погано обумовленою, якщо незначні зміни елементів матриці коефіцієнтів або правих частин призводять до надто значних змін у розв'язку.

Наприклад, розглянемо систему тільки двох рівнянь:

$$\begin{cases} 0{,}832 \cdot x_1 + 0{,}448 \cdot x_2 = 1 \\ 0{,}784 \cdot x_1 + 0{,}421 \cdot x_2 = 0 \end{cases} \qquad (6.20)$$

точним розв'язком якої з точністю до трьох знаків є

$x_1 = -439, \quad x_2 = 817$.

Розв'яжемо її методом Гауса з вибором головного елемента по усьому полю, який, як ми встановили в п.6.2.4, є достатньо надійним. Припустимо, що задачу обчислюють за допомогою гіпотетичної ЕОМ з трьома десятинними розрядами.

Оскільки $\boldsymbol{a_{11}}$ є максимальним за модулем серед елементів матриці, то ніяких перестановок рядків і стовпчиків не проводимо, а зразу виконуємо процедуру Гауса:

$$R = \frac{a_{21}}{a_{11}} = \frac{0{,}784}{0{,}832} \approx 0{,}942308 \approx 0{,}942$$

$$a_{22}^{(1)} = a_{22} - a_{12}R = 0{,}421 - 0{,}448 \cdot 0{,}942 \approx 0{,}421 - 0{,}422016 \approx 0{,}421 - 0{,}422 = -0{,}001,$$

$$b_2^{(1)} = b_2 - b_1 R = 0 - 1 \cdot 0{,}942 = -0{,}942$$

В результаті отримали систему з трикутною матрицею:

$$\begin{cases} 0{,}832 \cdot x_1 + 0{,}448 \cdot x_2 = 1 \\ 0{,}001 \cdot x_1 = -0{,}942 \end{cases}$$

з якої знайдемо вектор невідомих:

$$x_1 = -506, \quad x_2 = 942 \qquad (6.21)$$

Порівнюючи його з точним розв'язком ***(-439;817)***, помітимо, що відмінність становить близько *15%*. Яка ж тому причина?

Використаємо принцип зворотного аналізу помилок, тобто з'ясуємо, яка система має точний розв'язок (6.21). Виявляється [2], насправді ми розв'язали систему

$$\begin{cases} 0{,}832 \cdot x_1 + 0{,}447974... \cdot x_2 = 1 \\ 0{,}783744... \cdot x_1 + 0{,}420992... \cdot x_2 = 0 \end{cases} \qquad (6.22)$$

коефіцієнти якої відрізняються від коефіцієнтів системи (6.20) не більше, ніж на *0,03%*. Така зовсім незначна відмінність цих систем призводить до того, що точні розв'язки відрізняється на *15%*, тобто помилки вихідних даних збільшилися в *500* разів.

Причина полягає в тому, що матриця коефіцієнтів системи (6.20) майже вироджена, тобто прямі лінії, що визначаються рівняннями цієї системи, майже паралельні (рис. 6.13). Визначник цієї матриці наближений до нуля, її рядки майже пропорційні.

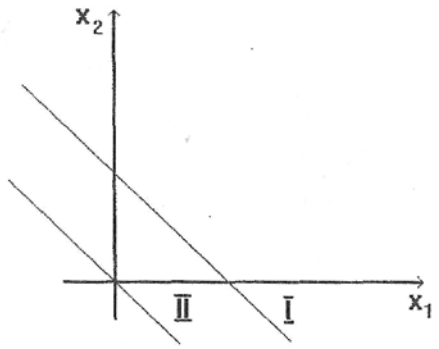


Рис. 6.13. Прямі, що визначаються погано обумовленою системою (6.20), майже паралельні

Розглянемо систему рівнянь, отриману із системи (6.20):

$$\begin{cases} 0{,}832 \cdot x_1 + 0{,}447974... \cdot x_2 = 1 \\ 0{,}784 \cdot x_1 + (0{,}421 + \varepsilon) \cdot x_2 = 0 \end{cases} \qquad (6.23)$$

її друге рівняння визначає сімейство прямих, що залежить від параметра ***ε***.

При ***ε=0*** ця система збігається з системою (6.20). Якщо ***ε*** збільшується від нуля до *0,012*, пряма ***II*** на рис. 6.13 повертається проти годинникової

стрілки до такого положення, коли вона паралельна прямій ***I***. При цьому система (6.23) не матиме розв'язку, визначник дорівнюватиме нулю. В процесі обертання точка перетину двох прямих буде віддалятися в нескінченність. При такому пониженні параметра ε до значення, коли система вироджується, навіть невеликі зміни коефіцієнтів системи призводять до все більших змін щодо розв'язку.

Очевидно, з точки зору стійкості розв'язку система двох рівнянь буде "ідеально обумовленою", якщо визначені нею прямі взаємно перпендикулярні.

Відомо, що коли система вироджена, її визначник дорівнює нулю. Але чи випливає з малої його величини погана обумовленість системи? У загальному випадку відповідь на це запитання негативна.

Наприклад, значення наступних двох визначників суттєво відрізняються:

$$\begin{vmatrix} 10^{-10} & 0 \\ 0 & 10^{-10} \end{vmatrix} = 10^{-20}, \quad \begin{vmatrix} 10^{10} & 0 \\ 0 & 10^{10} \end{vmatrix} = 10^{20}$$

але відповідні системи рівнянь

$$\begin{cases} 10^{-10} \cdot x_1 = 0 \\ 10^{-10} \cdot x_2 = 0 \end{cases} \quad \begin{cases} 10^{10} \cdot x_1 = 0 \\ 10^{10} \cdot x_2 = 0 \end{cases}$$

визначають одні й ті самі прямі – координатні осі й ці системи є "ідеально обумовлені". Проте, якщо матриця системи масштабована, як це вказано далі, визначник може слугувати мірою обумовленості системи.

Очевидно, для системи двох рівнянь

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = 0 \\ a_{21}x_1 + a_{22}x_2 = 0 \end{cases}$$

хорошою мірою "паралельності" двох відповідних прямих може бути кут між ними або площа ромба з одиничною стороною, побудованого на цих прямих (рис. 6.14).

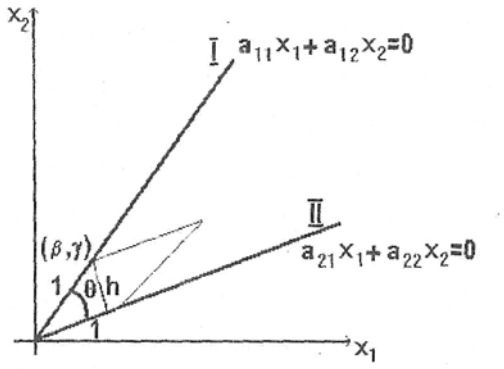


Рис. 6.14. Оцінка обумовленості системи двох рівнянь

Позначимо висоту ромба через ***h***. Тоді його площа також дорівнюватиме ***h***. Оскільки ***h=sinθ*** (***θ*** – кут між прямими), то площа цього ромба буде змінюватися від нуля для прямих, що збігаються, до одиниці, коли прямі перпендикулярні. Як знайти ***h***, якщо відомі коефіцієнти a_{ij}?

Відстань ***h*** від точки ***(β,γ)*** до прямої ***II*** знайдемо за допомогою нормального рівняння цієї прямої

$$h = \frac{\left|-a_{21}\beta + a_{22}\gamma\right|}{\alpha_2}, \quad \alpha_2 = \sqrt{a_{21}^2 + a_{22}^2} \tag{6.24}$$

Залишилося виразити величини β та γ через a_{ij}. З очевидних співвідношень

$$\begin{cases} \beta^2 + \gamma^2 = 1 \\ a_{11}\beta + a_{12}\gamma = 0 \end{cases},$$

при $a_{11}>0$ можна знайти

$$\beta = -\frac{a_{12}}{\alpha_1},\ \gamma = \frac{a_{11}}{\alpha_1},\ \alpha_1 = \sqrt{a_{11}^2 + a_{12}^2}.$$

Підставивши отримані вирази для ***β*** та ***γ*** в (6.24), отримаємо формулу для обчислення ***h***:

$$h = \frac{\left|-a_{21}\dfrac{a_{12}}{\alpha_1} + a_{22}\dfrac{a_{12}}{\alpha_1}\right|}{\alpha_2} = \frac{\left|a_{11}a_{22} - a_{12}a_{21}\right|}{\alpha_1\alpha_2} = \frac{\left|\det A\right|}{\alpha_1\alpha_2},$$

яку можна використовувати для оцінки обумовленості системи. Аналогічна формула є й для системи ***n*** рівнянь:

$$V = \frac{\left|\det A\right|}{\alpha_1\alpha_2 \ldots \alpha_n} \tag{6.25}$$

де $\alpha_1 = \sqrt{a_{i1}^2 + a_{i2}^2 + ... + a_{in}^2}.$

Залежність (6.25) можна записати інакше:

$$V_1 = \det\begin{pmatrix} \frac{a_{11}}{\alpha_1} & \frac{a_{12}}{\alpha_1} & ... & \frac{a_{1n}}{\alpha_1} \\ & ... & & \\ \frac{a_{n1}}{\alpha_n} & \frac{a_{n2}}{\alpha_n} & ... & \frac{a_{nn}}{\alpha_n} \end{pmatrix}; \qquad V = |V_1|,$$

тобто для оцінки обумовленості системи можна знайти визначник її матриці, пронормований за допомогою α_i. Як і величина h, $V \in [0,1]$.

Помилки округлення, що виникають при розв'язуванні СЛАР на ЕОМ, відіграють таку саму роль, як і зміна коефіцієнтів вихідної системи. Як ми переконалися на прикладі систем (6.20) та (6.22) незначні зміни в коефіцієнтах погано обумовленої системи призводять до суттєвих змін у розв'язку. Припустимо, що нам необхідно розв'язати систему (6.22), точний розв'язок якої ***(-506; 942).*** Але для одержання її коефіцієнтів ми повинні були виміряти параметри деякого фізичного процесу за допомогою приладу, який забезпечує три правильні цифри числа. Тому замість системи (6.22) ми почали б розв'язувати систему (6.20) і навіть, якщо б розв'язали її абсолютно точно, то одержали б відповідь ***(-439; 817).*** Тобто для погано обумовлених систем необхідно не лише вміти знаходити точний розв'язок, але й треба достатньо точно виміряти коефіцієнти системи.

Класичним прикладом погано обумовленої матриці є матриця Гільберта:

$$H_n = \begin{pmatrix} 1 & \frac{1}{2} & ... & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & ... & \frac{1}{n+1} \\ ... & & & \\ \frac{1}{n} & \frac{1}{2} & ... & \frac{1}{2n-1} \end{pmatrix}$$

Із збільшенням n вона стає все більш погано обумовленою. Наприклад, якщо при $n=8$ занесемо в пам'ять ЕОМ таку матрицю у вигляді восьмизначних десятичних чисел й обернемо її точно, то отримана зворотня

матриця відрізнятиметься від точної зворотньої до H_8 матриці вже в першому знаці.

По-друге, для погано обумовлених систем обманливість вектора нев'язок $r=Ax-B$.

Якщо ми одержали результат, що збігається з точним, то вектор нев'язок дорівнюватиме нулю. Можна припустити, що коли ми одержали хороше наближення до точного розв'язку, то вектор нев'язок буде малим. І навпаки, якщо вектор нев'язок малий, тоді ми одержали хороше наближення. Але для погано обумовлених систем це не так. Наприклад, розглянемо систему

$$0{,}780x_1 + 0{,}563x_2 = 0{,}217,$$
$$0{,}913x_1 + 0{,}659x_2 = 0{,}254$$

з точним розв'язком $(1, -1)$.

Якби ми одержали результат, що помітно відрізняється від точного і дорівнює $x = \begin{pmatrix} 0{,}341 \\ -0{,}087 \end{pmatrix}$, то нев'язка була б $r = \begin{pmatrix} 10^{-6} \\ 0 \end{pmatrix}$.

Інший приближений розв'язок $x = \begin{pmatrix} 0{,}999 \\ -1{,}001 \end{pmatrix}$, що майже співпадає з точним, дає нев'язку $r = \begin{pmatrix} -0{,}0013 \\ 0{,}0015 \end{pmatrix}$.

Таке значення гірше попереднього, хоча вектор розв'язку ближче до точного. Таким чином, величина нев'язки для погано обумовлених систем тільки наводить на тінь.

Погану обумовленість матриці можна виявити не тільки за допомогою формули (6.25), але й використовуючи поняття норми вектора й матриці. Нагадаємо деякі види норм вектора x й матриці A:

$$\|x\|_1 = \sum_{i=1}^{n} |x_1|; \quad \|x\|_2 = \sqrt{n \sum_{i=1} x_i^2}, \quad \|x\|_\infty = \max_i |x_i| \tag{6.26}$$

$$\|A\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|, \quad \|A\|_\infty = \max_j \sum_{i=1}^{n} |a_{ij}| \tag{6.27}$$

З'ясуємо спочатку, як впливають на вектор розв'язку зміни правої частини системи. Нехай ***x**** **-** точний розв'язок системи ***Ax=B***, а ***x*+Dx*** – розв'язок системи ***Ax=B+DB***, тобто справджуються рівності

$$\begin{aligned} Ax^* &= B, \\ A(x^* + \Delta x) &= B + \Delta B \end{aligned} \qquad (6.28)$$

Підставивши перший вираз у другий, одержимо

$$A \cdot \Delta x = \Delta B$$

або

$$\begin{aligned} \Delta x &= A^{-1} \cdot \Delta B, \\ \|\Delta x\| &\le \|A^{-1}\| \cdot \|\Delta B\| \end{aligned} \qquad (6.29)$$

Звідси випливає, що коли $\|A^{-1}\|$ велика, то невеликі зміни вектора ***B*** можуть призвести до значних змін у розв'язку. Перейдемо до відносних величин. Із (6.28) випливає, що

$$\|B\| \le \|A\| \cdot \|x^*\| \qquad (6.30)$$

Помножимо нерівності (6.29) та (6.30):

$$\|\Delta x\| \cdot \|B\| \le \|A^{-1}\| \cdot \|\Delta B\| \cdot \|A\| \cdot \|x^*\|$$

та розділимо отриманий результат на $\|x^*\| \cdot \|B\|$

$$\frac{\|\Delta x\|}{\|x^*\|} \le \|A\| \cdot \|A^{-1}\| \cdot \frac{\|\Delta B\|}{\|B\|} \qquad (6.31)$$

Звідси випливає, що відносна зміна ***x*,*** обумовлена зміною вектора правої частини ***B***, обмежена відносною зміною ***B***, помноженою на $\|A\| \cdot \|A^{-1}\|$**.** Останню величину називають *числом обумовленості* матриці ***A*** й позначають ***cond(A)*:**

cond(A) **=** $\|A\| \cdot \|A^{-1}\|$**.**

Зазначимо, що ***cond(A)*** $\ge$ ***1***. Матриці, у яких ***cond(A)*** велике, називають *погано обумовленими*.

Розглянемо тепер, як змінюється вектор розв'язку в разі зміни матриці ***A***. Нехай ***x*+δx*** – розв'язок системи ***(A+δA)x=B***, тобто

$$(A + \delta A) \cdot (x^* + \delta x) = B.$$

Використавши (6.28), отримаємо:

$$Ax^* + a \cdot \delta A \cdot x^* + \delta A \cdot \delta x = B,$$
$$A \cdot \delta x + \delta A \cdot x^* + A \cdot \delta x = 0,$$
$$A \cdot \delta x = -\delta A(x^* + \delta x),$$
$$\delta x = -A^{-1} \cdot \delta A(x^* + \delta x),$$
$$\|\delta x\| \le \|A^{-1}\| \cdot \|\delta A\| \cdot \|x^* + \delta x\| = \|A\| \cdot \|A^{-1}\| \cdot \|\delta A\| \cdot \|x^* + \delta x\| \cdot \frac{1}{\|A\|} = cond(A) \cdot \frac{\|\delta A\|}{\|A\|} \cdot \|x^* + \delta x\|.$$

Поділимо це співвідношення на ***||x*+dx||:***

$$\frac{\|\delta x\|}{\|x^* + \delta x\|} \le cond(A) \cdot \frac{\|\delta A\|}{\|A\|} \qquad (6.32)$$

В одержаній оцінці зміни вектора розв'язку знову, як і в (6.31), бере участь ***cond(A).*** Нерівності (6.31), (6.32) необхідно інтерпретувати так, що якщо ***cond(A)*** мале (не набагато перебільшує одиницю), то незначні зміни в СЛАР призводять до незначних змін у розв'язку. Якщо ж ***cond(A)*** велике, то не обов'язково незначні зміни системи призведуть до великих змін щодо розв'язку.

Підсумовуючи, зазначимо, що деякі погано обумовлені системи не потрібно навіть намагатися розв'язати, а потрібно або переформулювати задачу, або провести більш точне вимірювання необхідних даних.

6.3. Ітеративні методи

Основні переваги ітераційних методів полягають в тому, що:

1. Якщо процес ітерації збігається швидко, тобто кількість наближень менша, ніж порядок системи, то виходить виграш у часі розв’язування системи.

2. Метод ітерацій є таким, що самокоректується, тобто окрема помилка обчислення не відбивається на остаточному результаті рішення.

3. Процес ітерацій легко програмується на ЕОМ.

4. Деякі методи ітерацій стають особливо вигідними при розв’язуванні систем, у яких значна кількість коефіцієнтів, розташованих підряд, дорівнює нулю (розріджені матриці).

Таким чином, ітераційні методи особливо ефективні для СЛАР високої розмірності і СЛАР з розрідженою матрицею. Для погано обумовлених систем рівнянь ітераційні методи дають такий самий неправильний результат, як і прямі методи [2].

В ітераційних методах чергове наближення x до вектора розв'язку (k-номер ітерації) залежить від $A, B, x^{(k-1)}, x^{(k-2)}, \ldots, x^{(k-r)}$.

Для економії пам'яті ЕОМ звичайно беруть $r=1$, так що $x=F_k(A,B,x^{(k-1)})$ і такі методи називають *однокроковими* на відміну від *багатокрокових*, у яких $r>1$. Якщо F_k не залежить від k, ітерацію називають *стаціонарною*. Якщо F_k – лінійна функція від $x^{(k-1)}$, ітерацію називають *лінійною*. Далі розглянемо найпростіші лінійні ітераційні методи. Найбільш загальною лінійною функцією є

$$F_k = Hx^{(k-1)} + V$$

де H – деяка матриця; V – вектор.

Природньо, що H і V беруть не довільними, а зв'язаними з матрицею A і вектором B початкової системи. Таким чином, загальна форма лінійного ітераційного процесу має вигляд

$$x^{(k)} = Hx^{(k-1)} + V, \quad k = 1,2,\ldots . \tag{6.33}$$

Умови збіжності таких методів визначаються такими ствердженнями.

Теорема 6.2. Векторна послідовність x, визначена ітераційним методом (6.33), де $x^{(k)}$ – заданий початковий вектор, збігається до вектора розв'язку x, якщо матриця H невироджена і задовольняє нерівність

$$|H|<1. \tag{6.34}$$

Ця теорема визначає достатню умову збіжності. Необхідні і достатні умови формулюють в наступній теоремі, яка є основним теоретичним результатом для методів (6.33).

Теорема 6.3. Для збіжності ітераційного процесу (6.33) при будь-якому початковому наближенні необхідно і достатньо, щоб усі власні значення матриці H були за модулем меншими за одиницю, тобто $\lambda_i<1$.

Іншими словами, спектральний радіус ***r(H)*** матриці ***H*** повинен бути меншим за одиницю: ***r(H)<1***.

Швидкість збіжності ітераційного процесу (6.33) визначають таким співвідношенням (***x**** – точний розв'язок):

$$\left\|x^* - x^{(k)}\right\| \le \|H\| \cdot \left\|x^* - x^{(k-1)}\right\|$$

тобто, чим менша норма матриці ***H***, тим швидше ми наближаємося до точного розв'язку ***x****.

Підставою для використання $\left\|x^* - x^{(k)}\right\|$ як критерію закінчення ітераційного процесу є таке співвідношення:

$$\left\|x^* - x^{(k)}\right\| \le \frac{\|H\|}{1-\|H\|} \cdot \left\|x^{(k)} - x^{(k-1)}\right\|.$$

Конкретні ітераційні методи одержують матрицю ***H*** і вектор ***V***, коли проводять еквівалентні перетворення системи ***Ax=B*** до вигляду ***x=Hx+V***. При цьому точний розв'язок ***x**** є нерухомою точкою лінійного перетворення ***Tx=Hx+V.***

6.3.1. Метод послідовних наближень

Розглянемо СЛАР з трьома невідомими:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \qquad (6.35)$$

або в матричній формі:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$Ax = B$$

Ліву частину кожного рівняння перенесемо праворуч і додамо до обох частин першого рівняння x_1, другого **-** x_2, третього **-** x_3.

Одержимо еквівалентну систему

$$\begin{aligned} x_1 &= x_1 - (a_{11}x_1 + a_{12}x_2 + a_{13}x_3) + b_1, \\ x_2 &= x_2 - (a_{21}x_1 + a_{22}x_2 + a_{23}x_3) + b_2, \\ x_3 &= x_3 - (a_{31}x_1 + a_{32}x_2 + a_{33}x_3) + b_3 \end{aligned} \qquad (6.36)$$

У термінах матриць ці перетворення мають вигляд

$$x = x - Ax + B,$$
$$x = (E - A)x + B.$$

Записавши систему (6.36) у вигляді:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ -a_{21} & 1-a_{22} & -a_{23} \\ -a_{31} & -a_{32} & 1-a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

можемо побудувати для неї ітераційний процес [5]:

$$\begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{pmatrix} = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ -a_{21} & 1-a_{22} & -a_{23} \\ -a_{31} & -a_{32} & 1-a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ x_3^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \qquad (6.37)$$

або

$$x^{(k)} = (E - A)x^{(k-1)} + B \qquad (6.38)$$

Такий запис методу використовують тільки для теоретичних цілей, а при фактичних обчислюваннях використовують покомпонентні формули.

При підготовці задачі до розв'язку на ЕОМ не треба проводити вручну перетворення систем для одержання форми (6.37), бо вони будуть передбачені покомпонентною формулою методу, яка для *i-ї* компоненти вектора розв'язку має в загальному випадку вигляд

$$x_i^{(k)} = x_i^{(k-1)} - \sum_{j=1}^{n} a_{ij} x_j^{k-1} + b_i \qquad (6.39)$$

або

$$x_i^{(k)} = (1 - a_{ii}) x_i^{(k-1)} - \sum_{j=f,\, j \neq i}^{n} a_{ij} x_j^{k-1} + b_i \qquad (6.40)$$

де ***n*** – порядок системи.

Позначивши в (6.38) ***H=E-A***, ***V=B***, одержимо більш загальну формулу (6.33).

Загальна достатня умова збіжності ітераційних методів ***||H||<1*** для нашого випадку має вигляд ***||E-A||<1***. Зважаючи на визначення норм (6.27), дійдемо висновку, що для збіжності методу послідовних наближень при

розв'язуванні системи ***n-го*** порядку достатньо, щоб виконувалася одна з умов:

$$\max_j \sum_{i=1}^{n} |h_{ij}| = \max_j \left\{ |1 - a_{jj}| + \sum_{i=1, i\neq j}^{n} |a_{ij}| \right\} < 1,$$

$$\max_i \sum_{j=1}^{n} |h_{ij}| = \max_i \left\{ |1 - a_{jj}| + \sum_{j=1, j\neq i}^{n} |a_{ij}| \right\} < 1$$

або, грубо кажучи, достатньо, щоб матриця ***A*** була близькою до одиничної. До такого вигляду її необхідно звести вручну за допомогою еквівалентних перетворень системи перед введенням початкових даних в ЕОМ.

Для геометричної ілюстрації методу послідовних наближень розглянемо систему, яка складається тільки з одного рівняння:

0,4x = 1.

Щоб одержати розв'язок ***x=2.5*** даним методом, перетворімо це рівняння

$$0 = -0{,}4x + 1,$$

$$x = x - 0{,}4x + 1,$$

$$x = 0{,}6x + 1.$$

Очевидно, розв'язок цього рівняння, еквівалентного початковому, це абсциса точки перетину двох прямих ***y=x*** та ***y=0,6x+1*** (рис. 6.15а).

Візьмемо деяке початкове наближення $x^{(0)}$ і обчислимо $x^{(1)}=0{,}6x^{(0)}+1$. Цим ми фактично знайдемо ординату точки ***A***, яка лежить на прямій ***y=0.6x+1***. Проведемо з точки ***A*** пряму, паралельну осі ***OX***, до перетину з прямою ***y=x*** у точці ***B***. Ця точка має ту саму ординату, що і точка ***A***, а абсциса точки ***B*** чисельно дорівнює її ординаті, тобто $0.6x^{(0)}+1$. Отже, абсциса точки ***B*** дорівнює $x^{(1)}$. Для обчислення $x^{(2)}$ на другій ітерації треба повторити ті ж самі дії, тобто з точки ***x*** на осі ***OX*** встановити перпендикуляр до його перетину з прямою ***y=0.6x+1*** в точці ***C***, через неї провести горизонтальну пряму до перетину з прямою ***y=x*** в точці ***D***, яка має абсцису ***x***, і т.д.

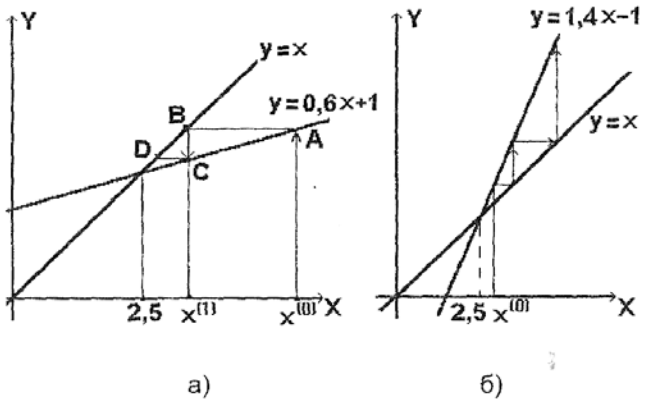


Рис. 6.15. Геометрична інтерпретація методу послідовних наближень

Побудуємо ітераційний процес:

$$x^{(k)} = 0{,}6x^{(k-1)} + 1, \quad k = 1,2,....$$

Як матриця ***H*** тут виступає кутовий коефіцієнт, який дорівнює ***0.6***. Процес на рис. 6.15а збігається, оскільки виконується умова ***||H||<1***. На рис. 6.15б зображений розбіжний ітераційний процес. Його побудовано для того самого рівняння ***0,4x=1***. Дійсно,

$$0 = 0{,}4x - 1,$$
$$x = x + 0{,}4x - 1,$$
$$x = 1{,}4x - 1$$
$$x^{(k)} = 1{,}4x^{(k-1)}.$$

Тут ***||H||=1.4>1*** і не виконується достатня умова збіжності. Крім того, ***r(H)*** теж більший за одиницю і процес повинен розбігатися. Таким чином, для однієї СЛАР ***Ax=B*** ітераційний процес може збігатися чи розбігатися залежно від способу зведення її до вигляду ***x=Hx+V,*** тобто залежно від використаного методу.

Приклад 6.1. Розв'язати методом послідовних наближень СЛАР:

$$\begin{aligned} &1{,}1x_1 - 0{,}2x_2 + 0{,}3x_3 = 1, \\ &0{,}1x_1 + 0{,}9x_2 + 0{,}2x_3 = 3, \\ &0{,}2x_1 - 0{,}1x_2 + 1{,}2x_3 = 2 \end{aligned} \qquad (6.41)$$

або у матричному вигляді:

$$\begin{pmatrix} 1.1 & -0.2 & 0.3 \\ 0.1 & 0.9 & 0.2 \\ 0.2 & -0.1 & 1.2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

Відповідно до даної системи ітераційний процес (6.38) можна записати у вигляді:

$$\begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{pmatrix} = \begin{pmatrix} -0.1 & 0.2 & -0.3 \\ -0.1 & 0.1 & -0.2 \\ -0.2 & 0.1 & -0.2 \end{pmatrix} \cdot \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ x_3^{(k-1)} \end{pmatrix} + \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} \qquad (6.42)$$

Позначимо

$$H = \begin{pmatrix} -0.1 & 0.2 & -0.3 \\ -0.1 & 0.1 & -0.2 \\ -0.2 & 0.1 & -0.2 \end{pmatrix}.$$

Перевіримо, чи виконується умова збіжності (6.34) для нашої матриці

$$\|H\|_1 = \|E - A\|_1 = \max_j \sum_{i=1}^{n} |h_{ij}| = 0.7 < 1.$$

Отже, процес (6.38) буде збіжним. Як початкове наближення візьмемо нульовий вектор і знайдемо з (6.42) перше наближення:

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{pmatrix} = \begin{pmatrix} -0.1 & 0.2 & -0.3 \\ -0.1 & 0.1 & -0.2 \\ -0.2 & 0.1 & -0.2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}.$$

Виконаємо ще одну ітерацію:

$$\begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{pmatrix} = \begin{pmatrix} -0.1 & 0.2 & -0.3 \\ -0.1 & 0.1 & -0.2 \\ -0.2 & 0.1 & -0.2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} + \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 0.9 \\ 2.8 \\ .1.7 \end{pmatrix}.$$

Процес можна продовжувати доти,поки два послідовних наближення не стануть достатньо близькими. Одержана послідовність векторів

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \to \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} \to \begin{pmatrix} 0.9 \\ 2.8 \\ 1.7 \end{pmatrix} \to \begin{pmatrix} 0.96 \\ 2.85 \\ 1.76 \end{pmatrix} \to \ldots$$

буде збігатися до точного розв'язку системи. Результати наведені в табл. 6.1 (див.далі). Природньо, що при програмуванні цього методу не потрібно зберігати в пам'яті ЕОМ усі вектора $x^{(к)}$, які одержуємо, достатньо двох останніх для оцінки різниці між ними. В загальному випадку ітераційний процес закінчується, якщо $\| x^{(К)} - x^{(К-1)} \| < \varepsilon$, тобто для всіх елементів векторів виконується умова $g_i < \varepsilon$, де

$$g_i = \begin{cases} \left| x_i^{(k)} - x_i^{(k-1)} \right| & при \ \left| x_i^{(k)} \right| \le 1, \\ \dfrac{\left| x_i^{(k)} - x_i^{(k-1)} \right|}{\left| x_i^{(k)} \right|} & при \ \left| x_i^{(k)} \right| > 1. \end{cases} \qquad (6.43)$$

Тут ***i*** – номер елемента вектора; ***k*** – номер ітерації; ***ε*** – допустима похибка (наприклад, ***ε =0.001***).

Ця умова перевіряється в кінці кожної ітерації і якщо вона не виконується хоча б для одного елемента вектора ***x***, процес продовжується.

6.3.2. Метод простої ітерації

Цей метод, який також називають методом Якобі, відрізняється від попереднього способом зведення системи ***Ax=B*** до вигляду ***x=Hx+V.*** Проілюструємо його на прикладі системи трьох рівнянь:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2, \\ a_{31}x_1 + a_{32}x_2 + a_{23}x_3 &= b_3. \end{aligned}$$

Припустимо, що діагональні коефіцієнти a_{ii} відмінні від нуля (інакше можна переставити рівняння). Виразимо з першого рівняння x_1, з другого x_2, з третього x_3:

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 - a_{13}x_3\right), \\ x_2 &= \frac{1}{a_{22}}\left(b_2 - a_{21}x_1 - a_{23}x_3\right), \\ x_3 &= \frac{1}{a_{33}}\left(b_3 - a_{31}x_1 - a_{32}x_2\right) \end{aligned}$$

і побудуємо ітераційний процес:

$$\begin{aligned} x_1^{(k)} &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^{(k-1)} - a_{13}x_3^{(k-1)}\right), \\ x_2^{(k-1)} &= \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{(k-1)} - a_{23}x_3^{(k-1)}\right), \\ x_3^{(k-1)} &= \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{(k-1)} - a_{32}x_2^{(k-1)}\right) \end{aligned} \qquad (6.44)$$

який і називають методом простої ітерації.

Запишемо (6.44) у вигляді:

$$x^{(k)} = Hx^{(k-1)} + V \tag{6.45}$$

де

$$H = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & -\frac{a_{13}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & -\frac{a_{23}}{a_{22}} \\ -\frac{a_{31}}{a_{33}} & -\frac{a_{32}}{a_3} & 0 \end{pmatrix}; \quad V = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \frac{b_3}{a_{33}} \end{pmatrix}.$$

Неважко переконатися, що

$$H = E - D^{-1}A, \; V = D^{-1}B.$$

де ***D*** – діагональна матриця елементів a_{11}. Отже, процес (6.45) можна записати через матрицю ***A*** і вектор ***B*** початкової системи ***Ax=B***:

$$x^{(k)} = (E - D^{-1}A)x^{(k-1)} + D^{-1}B \tag{6.46}$$

З загальної умови $||H||<1$ збіжності лінійних ітераційних методів випливає, що метод простої ітерації збігається, якщо виконується хоча б одна з двух умов:

$$\begin{aligned} |a_{ii}| &> \sum_{j=1,\, j\neq i}^{n} |a_{ij}|, \;\; i = 1,2,...,n \\ |a_{jj}| &> \sum_{i=1,\, i\neq i}^{n} |a_{ij}|, \;\; j = 1,2,...,n \end{aligned} \tag{6.47}$$

Іншими словами, для збіжності методу простої ітерації потрібно, щоб матриця СЛАР була діагонально домінуючою, тобто щоб модуль діагонального елемента в кожному рядку був більший за суму модулів решти елементів цього рядка або щоб у кожному стовпці модуль діагонального елемента був більший за суму модулів решти елементів цього стовпця. Ці умови є достатніми, але не необхідними, тобто для деяких систем ітерації збігаються і при порушенні умов (6.47).

Оцінку кількості потрібних ітерацій для одержання розв'язку з точністю є визначають формулою:

$$k \approx \frac{\ln \dfrac{p\varepsilon}{pq+u}}{\ln(1-p)}$$

де

$$p = \max_i \sum_{j=1,\, j\neq i}^{n} \left|\frac{a_{ij}}{a_{ii}}\right|,\ q = \max_i \left|x_{ij}^{(0)}\right|,\ u = \max_i \left|\frac{b_i}{a_{ii}}\right|.$$

Для виповнення однієї ітерації потрібно виконати в даному методі ***n*** операцій ділення, $\boldsymbol{n^2-n}$ операцій множення, ***n*** операцій додавання (***n*** – порядок системи).

Наведемо обчислювальну схему методу простої ітерації. До початку ітераційного процесу задаємо точність є і початкове наближення $\boldsymbol{x^{(0)}}$. Потім:

1. Обчислюємо чергове наближення до вектора розв'язку:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij} x_j^{(k-1)}\right),\ i = 1,2,\ldots,n.$$

2. Перевіряємо умову закінчення процесу:

$$g_i < \varepsilon,\ \ i = 1,2,\ldots,n$$

де

$$g_i = \begin{cases} \left|x_i^{(k)} - x_i^{(k-1)}\right| & \text{при } \left|x_i^{(k)}\right| \le 1, \\ \dfrac{\left|x_i^{(k)} - x_i^{(k-1)}\right|}{\left|x_i^{(k)}\right|} & \text{при } \left|x_i^{(k)}\right| > 1. \end{cases}$$

Якщо умова виконується для всіх ***i***, то $\boldsymbol{x^{(k)}}$ вважатимемо за розв'язок, інакше виконуємо чергову ітерацію з п.1.

Приклад 6.2. Розв'язати методом простої ітерації з точністю до $\boldsymbol{\varepsilon=0.01}$ систему трьох рівнянь:

$$\begin{aligned} &12x_1 - 3x_2 + x_3 = 9, \\ &x_1 + 5x_2 - x_3 = 8, \\ &x_1 - x_2 + 3x_3 = 8, \end{aligned} \qquad (6.48)$$

Як легко переконатися перевіркою, точним розв'язком цієї системи є $\boldsymbol{x_1=1,\ x_2=2,\ x_3=3}$. Перевіримо, чи виконуються достатні умови збіжності методу простої ітерації:

$$|12| > |-3| + |1|,$$
$$|5| > |1| + |-1|,$$
$$|3| > |1| + |-1|$$

Умови виконуються, тому використовуючи формулу(6.44) одержимо результати, наведенні в табл.6.2 (див. далі). На рис. 6.16 представлена одна з можливих реалізацій методу простої ітерації.

```
# -*- coding: cp1251 -*-
from numpy import *
from math import sqrt
from random import *


A=matrix([[12.,-3.,1.],[1.,5.,-1.],[1.,-1.,3.]])
b=matrix([[9.],[8.],[8.]])
n=len(b)

print "Метод Якобі"

def diagonal_prevalence(A,n):
    for i in range(n):
        s=0.0
        for j in range(n):
            s+=abs(A[i,j])
        if 2*A[i,i]<=s:
            return False
    return True

if diagonal_prevalence(A,n):
```

```
    print "Діагональна перевага виконується"
else:
    print "Діагональна перевага не виконується"

x=zeros(n)
print "Вектор початкового наближення"
print x

def next_vector(x):
    y=zeros(n)
    for i in range(n):
        s=0
        for j in range(n):
            if j!=i:
                s+=A[i,j]/A[i,i]*x[j]
        y[i]=-s+b[i]/A[i,i]
    return y

for i in range(5):
    x=next_vector(x)
print x
print "Вектор нев'язок:",dot(A,x)-b
```

Рис. 6.16. Реалізація методу простої ітерації

Ця реалізація використовує методику з частковим впорядкуванням (функція diagonal_prevalence). На даних з прикладу 6.2 ця програма видає наступне (рис. 6.17):

```
>>>
Метод Якобі
```

```
Діагональна перевага виконується
Вектор початкового наближення
[ 0.  0.  0.]
[ 1.00080247  2.00038272  3.0013786 ]
Вектор нев'язок: [[ 0.00986008 -0.99866255 -0.99544444]
 [ 1.00986008  0.00133745  0.00455556]
 [ 1.00986008  0.00133745  0.00455556]]
>>>
```

Рис. 6.17. Вивід попередньої програми

Як бачимо, отримані ті ж самі результати.

6.3.3. Метод Зейделя

За способом зведення системи $\boldsymbol{Ax=B}$ до вигляду $\boldsymbol{x=Hx+V}$ даний метод аналогічний методу послідовних наближень, однак тільки що обчислена компонента $\boldsymbol{x_i^{(k)}}$ вектора невідомих тут же використовується для обчислення наступної компоненти $\boldsymbol{x_{i+1}^{(k)}}$**.** Іншими словами, для обчислення $\boldsymbol{x_{i+1}^{(k)}}$ використовуються нові значення $\boldsymbol{x_1^{(k)}}$, $\boldsymbol{x_2^{(k)}}$,..., $\boldsymbol{x_i^{(k)}}$ і старі значення $\boldsymbol{x_{i+1}^{(k-1)}}$**,…,** $\boldsymbol{x_n^{(k-1)}}$.

Проілюструємо метод на прикладі системи трьох рівнянь:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1,\\ &a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2,\\ &a_{31}x_1 + a_{32}x_2 + a_{23}x_3 = b_3. \end{aligned} \qquad (6.49)$$

Як і в методі послідовних наближень, додамо до обох частин ***i****-го* рівняння $\boldsymbol{x_i}$ і одержимо систему:

$$\begin{aligned} &x_1 = x_1 - (a_{11}x_1 + a_{12}x_2 + a_{13}x_3) + b_1,\\ &x_2 = x_2 - (a_{21}x_1 + a_{22}x_2 + a_{23}x_3) + b_2,\\ &x_3 = x_3 - (a_{31}x_1 + a_{32}x_2 + a_{33}x_3) + b_3 \end{aligned} \qquad (6.50)$$

Задамо деякі початкові значення невідомих: $x_1=x_1^{(0)}$, $x_2=x_2^{(0)}$, $x_3=x_3^{(0)}$. Підставимо ці значення в праву частину першого з рівнянь (6.50) та одержимо нове (перше) наближення для x_1:

$$x_1^{(1)} = x_1^{(0)} - (a_{11}x_1^{(0)} + a_{12}x_2^{(0)} + a_{13}x_3^{(0)}) + b_1.$$

Використаємо тепер це значення x_1 і старі значення $x_2^{(0)}$, $x_3^{(0)}$. За допомогою другого рівняння (6.50) одержимо нове наближення для x_2:

$$x_2^{(1)} = x_2^{(0)} - (a_{21}x_1^{(1)} + a_{22}x_2^{(0)} + a_{23}x_3^{(0)}) + b_2.$$

Нарешті, на основі нових значень $x_1^{(1)}$, $x_2^{(1)}$ і старого значення $x_3^{(0)}$ з третього рівняння (6.50) знаходимо нове (перше) наближення для x_3:

$$x_3^{(1)} = x_3^{(0)} - (a_{31}x_1^{(1)} + a_{32}x_2^{(0)} + a_{33}x_3^{(0)}) + b_3.$$

На цьому закінчена перша ітерація. Наступні ітерації виконують аналогічно. Наближення з номером k можна записати у вигляді:

$$x_1^{(k)} = x_1^{(k-1)} - (a_{11}x_1^{(k-1)} + a_{12}x_2^{(k-1)} + a_{13}x_3^{(k-1)}) + b_1,$$
$$x_2^{(n)} = x_2^{(k-1)} - (a_{21}x_1^{(n)} + a_{22}x_2^{(k-1)} + a_{23}x_3^{(k-1)}) + b_2,$$
$$x_3^{(n)} = x_3^{(k-1)} - (a_{31}x_1^{(n)} + a_{32}x_2^{(n)} + a_{33}x_3^{(k-1)}) + b_3$$

Ітераційний процес продовжується, доки значення $x_1^{(k)}$, $x_2^{(k)}$, $x_3^{(k)}$ не стануть близькими з заданою похибкою до значень $x_1^{(k-1)}$, $x_2^{(k-1)}$, $x_3^{(k-1)}$.

У випадку системи n рівнянь обчислення виконують за формулою:

$$x_i^{(k)} = x_i^{(k-1)} - (\sum_{j=1}^{i-1} a_{ij}x_j^{(k-1)} + \sum_{j=1}^{n} a_{ij}x_j^{(k-1)}) + b_i. \qquad (6.51)$$

Умови збіжності методу Зейделя такі самі, як і для методу послідовних наближень, і в більшості випадків він збігається швидше цього методу. В табл. 6.1. наведені результати розв'язання системи (6.41) методами послідовних наближень і Зейделя.

Табл. 6.1. Порівняння двох ітераційних методів

	Метод послідовних наближень	**Метод Зейделя**

k	x_1	x_2	x_3	x_1	x_2	x_3
1	1,0000	3,0000	2,0000	1,0000	2,9000	2,0900
2	0,9000	2,8000	1,700	0,8530	2,7867	1,6900
3	0,960	2,860	1,760	0,9650	2,8442	1,7534
4	0,9460	2,8370	1,7410	0,9463	2,8391	1,7440
5	0,9505	2,8409	1,7463	0,9500	2,8401	1,7452
6	0,9492	2,8398	1,7447	6,9495	2,8400	1,7451
7	0,9496	2,8401	1,7462	0,9495	2,8400	1,7451
8	0,9495	2,8400	1,7451			
9	0,9495	2,8400	1,7451			

Запишемо формулу ітераційного процесу Зейделя у матричному вигляді. Систему (6.50) можна представити так:

$$x = Hx + V$$

де
$$H = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ -a_{21} & 1-a_{22} & -a_{23} \\ -a_{31} & -a_{32} & 1-a_{33} \end{pmatrix} = E - A,\ V = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

Матрицю ***H*** представимо у вигляді суми двох трикутних матриць

$$H = M + N,$$

$$M = \begin{pmatrix} 0 & 0 & 0 \\ -a_{21} & 0 & 0 \\ -a_{31} & -a_{32} & 0 \end{pmatrix},\quad N = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ 0 & 1-a_{22} & -a_{23} \\ 0 & 0 & 1-a_{33} \end{pmatrix}$$

Тоді можна записати ітераційний процес в матричному вигляді:

$$x^{(k)} = Mx^{(k)} + Nx^{(k-1)} + V$$

або

$$x^{(k)} = (E-M)^{-1}Nx^{(k-1)} + (E-M)^{-1} + V.$$

На одну ітерацію метода Зейделя потрібно n операцій ділення, n^2 операцій множення, n^2 операцій додавання.

6.3.4. Метод Некрасова

Цей метод, який іноді також називають методом Гауса-Зейделя, або методом Лібмана, аналогічний методу простої ітерації за способом зведення системи $Ax=B$ до вигляду $x=Hx+V$ і аналогічний методу Зейделя за способом обліку вже обчислених компонент.

Систему трьох рівнянь (6.49), виразивши з i-*го* рівняння невідому x_i, доведемо до вигляду

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3), \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3), \\ x_3 &= \frac{1}{a_{33}}(b_3 - a_{31}x_1 - a_{32}x_2) \end{aligned} \qquad (6.52)$$

Цей метод полягає у тому, що задавшись деяким початковим наближенням $x_1^{(0)}$, $x_2^{(0)}$, $x_3^{(0)}$, обчислення проводять за такими формулами, використовуючи тільки що обчисленні компоненти вектора **x** для обчислення чергових компонент:

$$\begin{aligned} x_1^{(k)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k-1)} - a_{13}x_3^{(k-1)}), \\ x_2^{(k)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k-1)}), \\ x_3^{(k)} &= \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)}) \end{aligned} \qquad (6.53)$$

У загальному випадку системи n рівнянь i-*а* компонента вектора розв'язку на k-*й* ітерації обчислюється як

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=ш+1}^{m} a_{ij}x_j^{(k-1)}\right) \qquad (6.54)$$

Систему (6.52) у матричній формі можна записати так:

$$x = Hx + V,$$

де

$$H = -D^{-1}(L + R), \quad V = D^{-1}B, \quad A = L + D + R,$$

$$L = \begin{pmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{31} & a_{32} & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 0 & a_{12} & a_{13} \\ 0 & 0 & a_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad D = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}$$

Отже, у матричній формі ітераційний метод Некрасова (6.54) має вигляд

$$x^{(k)} = D^{-1}Lx^{(k)} - D^{-1}Rx^{(k-1)} + D^{-1}B \qquad (6.55)$$

або

$$x^{(k)} = -(D + L)^{-1}Rx^{(k-1)} + (D + L)^{-1}B.$$

Кількість арифметичних операцій для виконання однієї ітерації методу Некрасова така сама, як і в методі простої ітерації. Збігаються у цих методах і достатні умови збіжності (діагональне домінування матриці ***A***). Найчастіше метод Некрасова збігається швидше, ніж метод простої ітерації. Як ілюстрація в табл.1.2 наведені результати розв'язання системи (6.48) методами простої ітерації і Некрасова.

Табл. 1.2. Порівняння двох ітераційних методів

	Метод простої інтерації			**Метод Некрасова**		
k	**x1**	**x2**	**x3**	**x1**	**x2**	**x3**
0	0	0	0	0	0	0
1	0,75	1,60	2,66	0,75	1,45	2,89
2	0,93	1,85	2,95	0,89	2,00	3,03
3	0,97	2,00	2,93	1,00	2,00	3,00
4	1,00	2,00	3,00	1,00	2,00	3,00
5	1,00	2,00	3,00			

Реалізація методу Гауса-Зейделя у вигляді окремого модуля представлена на рис. 6.18.

```
# -*- coding: cp1251 -*-

## module gaussSeidel

''' x,numIter,omega = gaussSeidel(iterEqs,x,tol = 1.0e-9)
```

```
    Рішення методом Зейделя [A]{x} = {b}.
    Матриця [A] повинна бути розріджена. Користувач повинен передати
    функцію iterEqs(x,omega), що повертає покращене {x},
    за наданим {x} ('omega' фактор релаксації).
'''
from numarray import dot
from math import sqrt

def gaussSeidel(iterEqs,x,tol = 1.0e-9):

    omega = 1.0
    k = 10
    p = 1
    for i in range(1,501):
        xOld = x.copy()
        x = iterEqs(x,omega)
        dx = sqrt(dot(x-xOld,x-xOld))
        if dx < tol: return x,i,omega
      # Підрахування фактору релаксації після к+р ітерацій
        if i == k: dx1 = dx
        if i == k + p:
            dx2 = dx
            omega = 2.0/(1.0 + sqrt(1.0 - (dx2/dx1)**(1.0/p)))
    print 'Метод Зейделя не зійшовся'
```

Рис. 6.18. Реалізація методу Гауса-Зейделя

Розглянемо приклад застосування наведеного модуля.

Приклад 6.3. Нехай потрібно вирішити наступну систему рівнянь методом Зейделя. Потрібно написати програму, яка буде працювати з будь-яким числом рівнянь ***n***.

$$\begin{bmatrix} 2 & -1 & 0 & 0 & & 0 & 0 & 0 & 1 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & & 0 & 0 & 0 & 0 \\ & \vdots & & & \ddots & & \vdots & & \\ 0 & 0 & 0 & 0 & & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 2 & -1 \\ 1 & 0 & 0 & 0 & & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Вирішити систему для ***n=20***. Можна показати, що точне рішення системи таке: $x_i = -\frac{n}{4} + \frac{i}{2}$.

Розв'язок. Згідно з (6.55):

$$x_1 = \frac{w(x_2 - x_n)}{2} + (1 - w)x_1,$$

$$x_i = \frac{w(x_{i-1} + x_{i+1})}{2} + (1 - w)x_i, i = 2,3, \dots, n - 1,$$

$$x_n = \frac{w(1 - x_1 + x_{n-1})}{2} + (1 - w)x_n.$$

Ці формули визначаються в функції iterEqs. Програму наведено на рис. 6.19.

```
# -*- coding: cp1251 -*-
from numpy import zeros,float64
from gaussSeidel import *


def iterEqs(x,omega):
    n = len(x)
    x[0] =omega*(x[1] - x[n-1])/2.0 + (1.0 - omega)*x[0]
    for i in range(1,n-1):
        x[i] = omega*(x[i-1] + x[i+1])/2.0 + (1.0 - omega)*x[i]
        x[n-1] = omega*(1.0 - x[0] + x[n-2])/2.0 \
          + (1.0 - omega)*x[n-1]
    return x
```

```
n = eval(raw_input("Кількість рівнянь ==> "))
x = zeros((n),dtype=float64)
x,numIter,omega = gaussSeidel(iterEqs,x)
print "\nКількість ітерацій =",numIter
print "\nФактор релаксації =",omega
print "\nРішення таке:\n",x
raw_input("\nНатисніть ввід для виходу")
```

Рис. 6.19. Реалізація програми застосування модуля

Результат виконання програми показано на рис. 6.20.

```
>>>
Кількість рівнянь ==> 20
Кількість ітерацій = 259
Фактор релаксації = 1.70545231071

Рішення таке:
[ -4.50000000e+00  -4.00000000e+00  -3.50000000e+00  -3.00000000e+00
  -2.50000000e+00  -2.00000000e+00  -1.50000000e+00  -9.99999997e-01
  -4.99999998e-01   2.14047151e-09   5.00000002e-01   1.00000000e+00
   1.50000000e+00   2.00000000e+00   2.50000000e+00   3.00000000e+00
   3.50000000e+00   4.00000000e+00   4.50000000e+00   5.00000000e+00]
Натисніть ввід для виходу
```

Рис. 6.20. Результат виконання програми

6.4 Контрольні запитання

1. Чим відрізняються прямі методи від ітераційних?

2. До якого вигляду зводиться матриця коефіцієнтів прямого ходу методу Гауса.

3. Коли не можна застосувати метод Гауса?

4. Який елемент є провідним в стовпчику матриці?

5. У чому перевага методу Гауса з вибором головного елементу в стовпчику?

6. До якого вигляду зводиться матриця в методі Гауса-Жордано?

7. Чи потрібен зворотній хід в методі Некрасова?

8. Яка умова завершення ітерації в ітераційних методах?

9. Головні переваги методу Гауса-Зейделя перед методом простих ітерацій.

10. Як перевірити істинність чи хибність знайдених корнів?

Розділ 7. Розв'язування систем нелінійних рівнянь

Нехай потрібно вирішити систему рівнянь

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0, \\ f_2(x_1, x_2, ..., x_n) = 0, \\ ..... \\ f_n(x_1, x_2, ..., x_n) = 0, \end{cases} \tag{7.1}$$

де $f_1, f_2, \dots, f_n$ – задані, взагалі кажучи, нелінійні (серед них можуть бути і лінійні) дійснозначні функції n дійсних змінних. Позначивши

$$\bar{x} := \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}, \quad F(x) := \begin{pmatrix} f_1(\bar{x}) \\ f_2(\bar{x}) \\ \dots \\ f_n(\bar{x}) \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, ..., x_n) \\ f_2(x_1, x_2, ..., x_n) \\ \dots \\ f_n(x_1, x_2, ..., x_n) \end{pmatrix}, \quad \bar{0} := \begin{pmatrix} 0 \\ 0 \\ \dots \\ 0 \end{pmatrix}$$

дану систему (7.1) можна записати одним рівнянням

$$F(\bar{x}) = 0 \tag{7.2}$$

відносно векторної функції F векторного аргументу x. Таким чином, вихідне завдання можна розглядати як задачу про нулі нелінійного відображення. В цій постановці вона є прямим узагальненням основного завдання побудови методів знаходження нулів одновимірних нелінійних відображень. Фактично це те ж завдання, лише в просторах більшої розмірності. Тому можна як заново будувати методи її рішення на основі розроблених вище підходів, так і здійснювати формальне перенесення виведених для скалярного випадку розрахункових формул. В будь-якому разі слід подумати про правомірність тих або інших операцій над векторними змінними та векторними функціями, а також про збіжність отримуваних у такий спосіб ітераційних процесів.

Часто теореми збіжності для цих процесів є тривіальними узагальненнями відповідних результатів, отриманих для методів вирішення скалярних рівнянь. Проте не всі результати та не всі методи можна перенести з випадку $n = 1$ на випадок $n \geq 2$. Наприклад, тут вже не працюватимуть методи дихотомії, оскільки безліч векторів не впорядкована. В той же час, перехід від $n = 1$ до $n \geq 2$ вносить до завдання знаходження нулів

нелінійного відображення свою специфіку, облік якої призводить до нових методів і до різних модифікацій тих, що вже є. Зокрема, велика варіативність методів вирішення нелінійних систем пов'язана з різноманітністю способів, якими можна вирішувати лінійні завдання алгебри, що виникають при покроковій лінеаризації даної нелінійної вектор-функції ***F(x)***.

7.1. Метод Ньютона, його реалізації та модифікації

7.1.1. Метод Ньютона

Нехай **(A_k)** — деяка послідовність невироджених дійсних ***nxn***-матриц. Тоді, очевидно, послідовність завдань

$$x = x - A_k F(x), \quad k = 0,1,2,...$$

має ті ж рішення, що і вихідне рівняння (7.2), і для наближеного знаходження цих розв’язків можна формально записати ітераційний процес

$$x^{(k+1)} = x^{(k)} - A_k F(x^{(k)}), \; k = 0,1,2,... \qquad (7.3)$$

що має вигляд методу простих ітерацій (7.17) при $\Phi(x) := \Phi_k(x) := x - A_k F(x)$. У випадку $\boldsymbol{A_k = A}$ – це дійсно метод простої ітерації з лінійною збіжністю послідовності $(x^{(k)})$. Якщо ж $\boldsymbol{A_k}$ різні при різних $\boldsymbol{k}$, то формула (7.3) визначає велике сімейство ітераційних методів з матричними параметрами. Розглянемо деякі з методів цього сімейства.

Нехай $A_k := \left[F'(x^{(k)})\right]^{-1}$, де

$$F'(x) = J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \qquad (7.4)$$

- матриця Якобі вектор-функції ***F(x)***. Підставивши це $\boldsymbol{A_k}$ в (7.3), отримаємо явну формулу методу Ньютона

$$x^{(k+1)} = x^{(k)} - \left[F'(x^{(k)})\right]^{-1} F(x^{(k)}), \qquad (7.5)$$

узагальненого на багатовимірний випадок скалярний метод Ньютона (4.2, 4.3). Цю формулу, що вимагає звернення матриць на кожній ітерації, можна переписати в неявному вигляді:

$$F'(x^{(k)})(x^{(k+1)} - x^{(k)}) = -F(x^{(k)}) \tag{7.6}$$

Вживання (7.6) передбачає при кожному ***k=0,1,2,...*** вирішення лінійної алгебраїчної системи

$$F'(x^{(k)})p^{(k)} = -F(x^{(k)}) \tag{7.7}$$

відносно векторної поправки, а потім збільшення цієї поправки до поточного наближення для здобуття наступного:

$$x^{(k+1)} = x^k + p^{(k)}. \tag{7.8}$$

До вирішення таких лінійних систем можна залучати самі різні методи як прямі, так і ітераційні залежно від розмірності ***n*** вирішуваного завдання і специфіки матриць Якобі $J(x^{(k)})$ (наприклад, можна враховувати їх симетричність, розрідженість і тому подібне).

Порівнюючи (7.6) з формальним розкладанням ***F(x)*** в ряд Тейлора

$$F(x) = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) + \frac{1}{2!}F''(x^{(k)})(x - x^{(k)})^2 + \dots,$$

видно, що послідовність (x_k) в методі Ньютона виходить в результаті заміни при кожному ***k=0,1,2...*** нелінійного рівняння ***F(x)=0*** або, що те ж (при достатній гладкості ***F(x)***), рівняння

$$F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) + \frac{1}{2!}F''(x^{(k)})(x - x^{(k)})^2 + \dots = 0$$

лінійним рівнянням

$$F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) = 0,$$

тобто з покроковою лінеаризацією. Як наслідок цього факту, можна розраховувати, що при достатній гладкості ***F(x)*** і досить гарному початковому наближенні $x^{(0)}$ збіжність породжуваної методом Ньютона послідовності (x_k) до розв'язку x^* буде квадратичною і в багатовимірному випадку. Є ряд теорем, що встановлюють це при тих або інших припущеннях. Зокрема, одна з таких теорем наводиться нижче.

Новим, в порівнянні із скалярним випадком, чинником, що ускладнює застосування методу Ньютона для розв'язування ***n***-мірних систем, є необхідність вирішення ***n***-мірних лінійних завдань на кожній ітерації (звернення матриць в (7.5) або вирішення СЛАР в (7.6)), обчислювальні витрати на яких зростають із зростанням ***n***, взагалі кажучи, непропорційно швидко. Зменшення таких витрат – один з напрямів модифікації методу Ньютона.

7.1.2. Модифікований метод Ньютона

*Я*кщо матрицю Якобі ***F'(x)*** обчислити і обернути лише один раз – в початковій точці, то від методу Ньютона (7.5) прийдемо до модифікованого методу Ньютона

$$x^{(k+1)} = x^{(k)} - \left[F'(x^{(0)})\right]^{-1} F(x^{(k)}) \qquad (7.9)$$

Цей метод вимагає значно менших обчислювальних витрат на один ітераційний крок, але ітерацій при цьому може бути потрібно значно більше для досягнення заданої точності в порівнянні з основним методом Ньютона (7.5), оскільки, будучи окремим випадком МПІ ($A := \left[F'(x^{(0)})\right]^{-1}$), він має лише швидкість збіжності геометричної прогресії.

Компромісний варіант – це обчислення і звернення матриць Якобі не на кожному ітераційному кроці, а через декілька кроків (інколи такі методи називають рекурсивними).

Наприклад, просте чергування основного (7.5) і модифікованого (7.9) методів Ньютона призводить до ітераційної формули

$$x^{(k+1)} = x^{(k)} - \left[F'(x^{(k)})\right]^{-1} F(x^{(k)}) \qquad (7.10)$$

де $A_k := \left[F'(x^{(k)})\right]^{-1}$ ***k=0,1,2,...*** За $x^{(k)}$ тут приймається результат послідовного використання одного кроку основного, а потім одного кроку модифікованого методу, тобто двоступінчатого процесу

$$\begin{cases} z^{(k)} = x^{(k)} - A_k F(x^{(k)}), \\ x^{(k+1)} = z^{(k)} - A_k F(z^{(k)}). \end{cases} \qquad (7.11)$$

Доведено, що такий процес за певних умов породжує послідовність, що кубічно сходиться ($x^{(k)}$).

7.1.3. Метод Ньютона з послідовною апроксимацією матриць

Завдання звернення матриць Якобі на кожному ***k***-му кроці методу Ньютона (7.5) можна спробувати вирішувати не точно, а приблизно. Для цього можна застосувати, наприклад, ітераційний процес Шульца, обмежуючись мінімумом – всього одним кроком процесу другого порядку, в якому за початкову матрицю береться матриця, отримана в результаті попереднього (***k-1***) -го кроку. Таким чином приходимо до методу Ньютона з послідовною апроксимацією зворотних матриць:

$$\begin{cases} x^{(k+1)} = x^{(k)} - A_k F(x^{(k)}), \\ \Psi_k = E - F'(x^{(k+1)})A_k, A_{k+1} = A_k + A_k \Psi_k, \end{cases} \qquad (7.12)$$

де ***k=0,1,2,.***, а $x^{(0)}$ і $A^{(0)}$ – початковий вектор і матриця ($\approx \left[F'(x^{(0)})\right]^{-1}$). Цей метод (називатимемо його коротше ААМН — апроксимаційний аналог методу Ньютона) має просту схему обчислень – почергове виконання векторних в першому рядку і матричних в другому рядку його запису (7.12) операцій. Швидкість його збіжності майже така ж висока, як і в методі Ньютона. Послідовність ($x^{(k)}$) може квадратично збігатися до розв'язку x^* рівняння ***F(x)=0*** (при цьому матрична последовательность (A_k) також квадратично збігається до $A^* := \left[F'(x^*)\right]^{-1}$, тобто в ітераційному процесі (7.12), що нормально розвивається, повинна спостерігатися досить швидка збіжність ($\|\Psi_k\|$) до нуля).

Застосування тієї ж послідовної апроксимації зворотних матриць до простого рекурсивного методу Ньютона (7.10) або, що те ж, до двоступінчатого процесу (7.11) визначає його апроксимаційний аналог

$$\begin{cases} z^{(k)} = x^{(k)} - A_k F(x^{(k)}), \quad x^{(k+1)} = z^{(k)} - A_k F(z^{(k)}), \\ \Psi_k = E - F'(x^{(k+1)})A_k, \qquad A_{k+1} = A_k + A_k \Psi_{k,} \end{cases} \qquad (7.13)$$

як і (7.10), також можна віднести до методів третього порядку. Доказ кубічної збіжності цього методу вимагає вже жорсткіших обмежень на властивості ***F(x)*** і близькість $x^{(0)}$ до x^*, A_0 до $\left[F'(x^{(0)})\right]^{-1}$, ніж в попередньому методі. Зазначимо, що до поліпшення збіжності тут може привести підвищення порядку апроксимації зворотних матриць, наприклад, за рахунок додавання ще одного доданку у формулі для підрахунку A_{k+1}:

$$A_{k+1} = A_k + A_k\Psi_k + A_k\Psi_k^2 \qquad (7.14)$$

Розглянемо приклад.

Приклад 7.1. Вирішити систему нелінійних рівнянь

$$\begin{cases} \sin(x+y) - 1.6x = 0 \\ x^2 + y^2 = 1 \end{cases}.$$

Початкове наближення:

$$Z^{(0)} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

Вектор-функція:

$$F(Z) = \begin{bmatrix} \sin(x+y) - 1.6x \\ x^2 + y^2 - 1 \end{bmatrix}.$$

Матриця Якобі вектор-функції:

$$F'(Z) = J(Z) = \begin{bmatrix} \cos(x+y) - 1.6 & \cos(x+y) \\ 2x & 2y \end{bmatrix}.$$

Обчислюємо корінь за формулою методу Ньютона з точністю $\varepsilon = 0.001$:

$$Z^{(k+1)} = Z^{(k)} - \left[F'(Z^{(k)})\right]^{-1} * F(Z^{(k)})$$

k	$Z^{(k)}$	$F(Z^{(k)})$	$F'(Z^{(k)})$	$\left[F'(Z^{(k)})\right]^{-1}$	$Z^{(k+1)}$	$\|Z\|$
0	0 -1	-0.841 0	-1.06 0.54 0 -2	-0.944 -0.255 0 -0.5	-0.794 -1	$0.794 > \varepsilon$
1	-0.794 -1	0.295 0.63	-1.821 -0.221 -1.588 -2	-0.608 0.067 0.482 -0.553	-0.657 -0.794	$0.247 > \varepsilon$
2	-0.657 -0.794	0.058 0.062	-1.48 0.12 -1.314 -1.588	-0.633 -0.048 0.524 -0.59	-0.617 -0.788	$0.040 > \varepsilon$
3	-0.617 -0.788	-0.0000597 0.011	-1.441 0.159 -1.234 -1.588	-0.639 -0.064 0.497 -0.58	-0.616 -0.788	$0.001 = \varepsilon$

4	-0.616 -0.788	0.000522 0.0004	-1.434 -1.232	0.166 -1.576	-0.639 0.5	-0.067 -0.582	-0.616 -0.788	$0<\varepsilon$

Відповідь: $Z = \begin{bmatrix} -0.616 \\ -0.788 \end{bmatrix}$

Блок-схема алгоритму розв'язування систем нелінійних рівнянь методом Ньютона представлена на рис. 7.1.

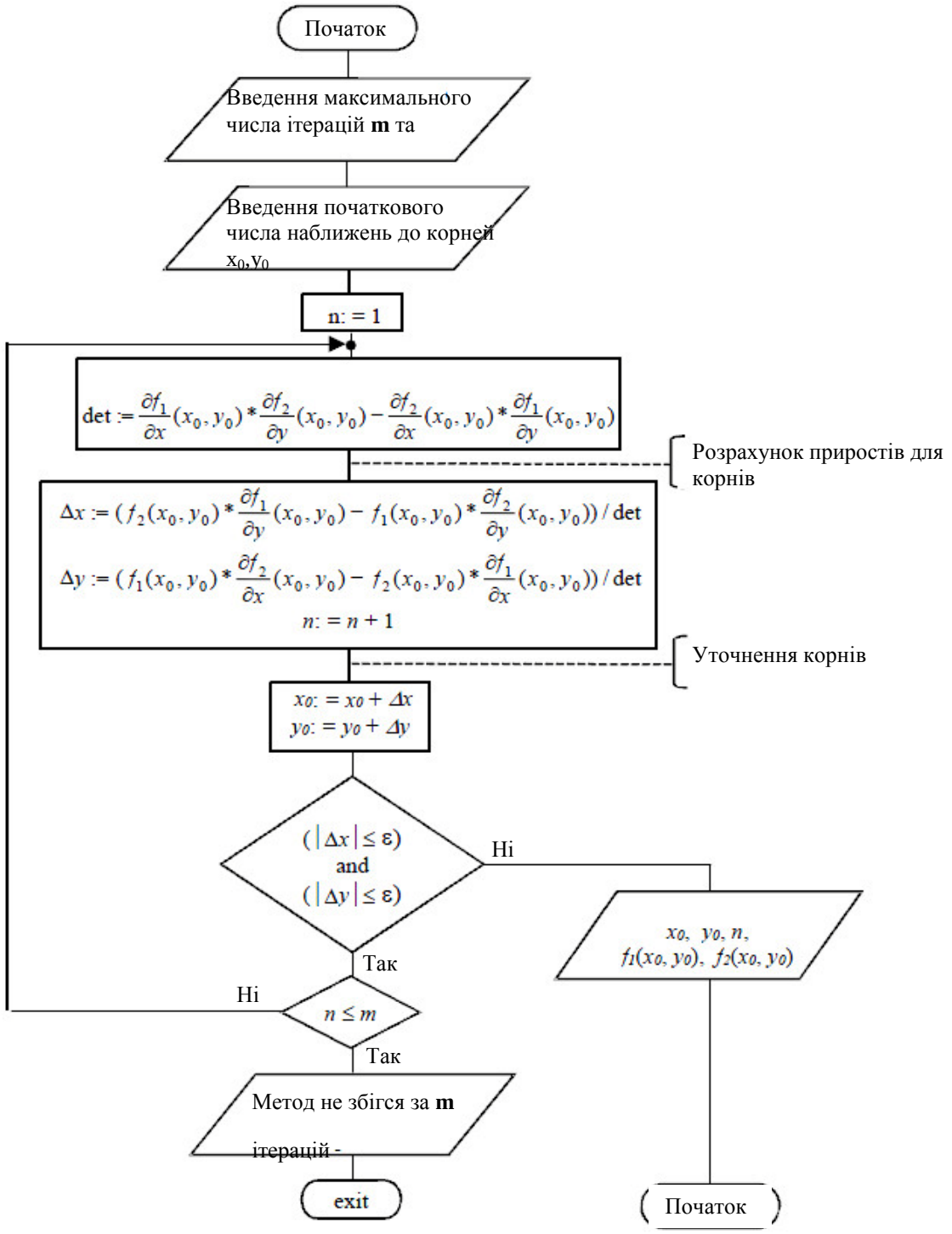


Рис. 7.1. Блок-схема алгоритму

Реалізація запропонованого алгоритму пропонується як самостійна вправа.

7.1.4. Різницевий метод Ньютона

На базі методу Ньютона (7.5) можна побудувати близький до нього за поведінкою ітераційний процес, що не вимагає обчислення похідних. Зробимо це, замінивши частинні похідні в матриці Якобі *J(x)* різницевими відношеннями, тобто підставивши у формулу (7.3) замість A_k матрицю $\left[J(x^{(k)}, h^{(x)})\right]^{-1}$ де

$$J(x,h) := \left(\frac{f_i(x_1,...,x_j + h_j,...,x_n) - f_i(x_1,...,x_j,...,x_n)}{h_j} \right)_{i,j=1}^{n} \qquad (7.15)$$

При вдалому завданні послідовності малих векторів $h^{(k)} = \left(h_1^{(k)},...,h_n^{(k)}\right)^T$ (постійною або такою, що збігається до нуля) отриманий таким шляхом різницевий (або інакше, дискретний) метод Ньютона має надлінійну, аж до квадратичної, швидкість збіжності і узагальнює метод на багатовимірний випадок. При завданні векторного параметра ***h*** — кроку дискретизації — слід враховувати точність машинних обчислень, точність обчислення значень функцій, середні значення отримуваних наближень.

7.2. Інші методи вирішення систем нелінійних рівнянь

7.2.1. Метод простих січних

Можна пов'язати завдання послідовності ($h^{(k)}$) з якою-небудь векторною послідовністю, що збігається до нуля, наприклад, з послідовністю нев'язок ($F(x^{(k)})$) або поправок ($p^{(k)}$). Так, вважаючи $h_j^{(k)} := x_j^{(k-1)} - x_j^{(k)}$ де ***j=1,…,n***, а ***k=1,2.***, приходимо до простого методу січних — узагальнення скалярного методу січних:

$$x^{(k+1)} = x^{(k)} - \left[B(x^{(k)}, x^{(k-1)})\right]^{-1} F(x^{(k)}), \qquad (7.14)$$

де

$$B(x^{(k)}, x^{(k-1)}) := \left(\frac{f_i(x_1^{(k)},...,x_j^{(k-1)},...,x_n^{(k)}) - f_i(x_1^{(k)},...,x_j^{(k)},...,x_n^{(k)})}{x_j^{(k-1)} - x_j^{(k)}} \right)_{i,j=1}^{n} \quad \textbf{k=1,2,3..}$$

Цей метод є двокроковим і вимагає завдання двох початкових точок $x^{(0)}$ і $x^{(1)}$. При ***n=1*** збіжність методу (7.14) має порядок $\frac{1+\sqrt{5}}{2}$. Можна розраховувати на таку ж швидкість і в багатовимірному випадку.

До методу січних так само, як і до методу Ньютона, можна застосувати покрокову апроксимацію зворотних матриць на основі методу Шульца. Розрахункові формули цієї модифікації легко виписати, замінивши в сукупності формул ААМН (7.12) матрицю $F'(x^{(k+1)})$ на матрицю $B(x^{(k+1)}, x^{(k)})$ з (7.14).

7.2.2. Метод простих ітерацій

Нехай система (7.1) перетворена до вигляду:

$$\begin{cases} x_1 = \varphi_1(x_1, x_2,..., x_n) = 0, \\ x_2 = \varphi_2(x_1, x_2,..., x_n) = 0, \\ \quad ..... \\ x_n = \varphi_n(x_1, x_2,..., x_n) = 0, \end{cases} \tag{7.15}$$

або інакше, в компактному записі

$$\bar{x} = \Phi(\bar{x}), \tag{7.16}$$

де

$$\Phi(\bar{x}) := \begin{pmatrix} \varphi_1(\bar{x}) \\ \varphi_2(\bar{x}) \\ ... \\ \varphi_n(\bar{x}) \end{pmatrix} = \begin{pmatrix} \varphi_1(x_1, x_2,..., x_n) \\ \varphi_2(x_1, x_2,..., x_n) \\ ... \\ \varphi_n(x_1, x_2,..., x_n) \end{pmatrix}.$$

Для цього ***задачі про нерухому точку*** нелінійного відображення запишемо формально рекурентну рівність

$$x^{(k+1)} = \Phi(x^{(k)}) \tag{7.17}$$

яка визначає ***метод простих ітерацій (МПІ)*** (або ***метод послідовних наближень***) для завдання (7.15).

Якщо почати процес побудови послідовності $(x^{(k)})$ з деякого вектора $x^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, ..., x_n^{(0)}\right)^T$ і продовжити за формулою (7.17), то за певних умов ця послідовність із швидкістю геометричної прогресії наближатиметься до вектора x^* — нерухомої точки відображення ***Φ(x)***. А саме, справедлива наступна теорема.

Теорема 7.1. Нехай функція Φ(x) і замкнута безліч $M \subseteq D(\Phi) \subseteq R_n$ *такі, що:*

1) $\Phi(x) \in M \quad \forall x \in M$;

2) $\exists q < 1 : \left\|\Phi(x) - \Phi(\tilde{x})\right\| \le q * \left\|x - \tilde{x}\right\| \quad \forall x, \tilde{x} \in M$

Тоді Φ(x) має в M єдину нерухому точку x^**; послідовність* $(x^{(k)})$*, що визначається МПІ (7.17), при будь-якому* $x^{(0)} \in M$ *сходиться до* x^* *і справедливі оцінки*

$$\left\|x^* - x^{(k)}\right\| \le \frac{q}{1-q}\left\|x^{(k)} - x^{(k-1)}\right\| \le \frac{q^k}{1-q}\left\|x^{(1)} - x^{(0)}\right\| \quad \forall k \in N.$$

Теорема 7.2. Нехай функція Φ(x) диференційована в замкнутій кулі $S(x^{(0)}, r) \subseteq D(\Phi)$, *причому* $\exists q \in (0,1)$*:*

$\sup\left\|\Phi'(x)\right\| \le q$. *Тоді, якщо центр* $x^{(0)}$ *і радіус r кулі S такі, що* $\left\|x^{(0)} - \Phi(x^{(0)})\right\| \le r(1-q)$, *то справедливий висновок теореми 7.1 з M=s.*

Якщо вимагати безперервну диференційність ***Φ(x)***, то простіше перейти від теореми 7.1 до теореми 7.2, застосувавши наступне ствердження.

Лема 4.1. Нехай функція $\Phi : R_n \to R_n$ *безперервна і диференційована на множині* $M \subseteq D(\Phi)$ *і нехай* $\left\|\Phi'(x)\right\| \le L \quad \forall x \in M$. *Тоді Φ(x) задовольняє на множині M умові Ліпшіця*

$$\left\|\Phi(x) - \Phi(\tilde{x})\right\| \le L\left\|x - \tilde{x}\right\|.$$

Запис МПІ (7.17) в розгорнутому вигляді, тобто сукупність рекурентної рівності

$$\begin{cases} x_1^{(k+1)} = \varphi_1(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}), \\ x_2^{(k+1)} = \varphi_2(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}), \\ ..... \\ x_n^{(k+1)} = \varphi_n(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}), \end{cases} \quad (7.18)$$

нагадує МПІ для СЛАР, який укладається в цю схему, якщо всі функції — лінійні. Враховуючи, що в лінійному випадку, як правило, в порівнянні з МПІ ефективніший метод Зейделя, тут теж може виявитися корисною модифікація. А саме, замість (7.17) можна реалізувати наступний метод ітерацій:

$$\begin{cases} x_1^{(k+1)} = \varphi_1(x_1^{(k)}, x_2^{(k)}, ..., x_{n-1}^{(k)}, x_n^{(k)}), \\ x_2^{(k+1)} = \varphi_2(x_1^{(k+1)}, x_2^{(k)}, ..., x_{n-1}^{(k)}, x_n^{(k)}), \\ ..... \\ x_n^{(k+1)} = \varphi_n(x_1^{(k+1)}, x_2^{(k+1)}, ..., x_{n-1}^{(k)}, x_n^{(k)}), \end{cases} \quad (7.19)$$

Зазначимо, що як і для лінійних систем, окремі рівняння в методі (7.19) нерівноправні, тобто зміна місцями рівнянь системи (7.15) може змінити в якихось межах число ітерацій і взагалі ситуацію із збіжністю послідовності ітерацій. Аби застосувати метод простих ітерації або його зейделеву модифікацію (7.19) до вихідної системи (7.1), потрібно, як і в скалярному випадку, спочатку тим або іншим способом привести її до вигляду (7.15). Це можна зробити, наприклад, помноживши (7.2) на деяку неособливу ***n-x-n*** матрицю – ***A*** і додавши до обох частин рівняння – ***AF(x)=0*** вектор невідомих ***x***. Отримана система

$$x = x - AF(x) \quad (7.20)$$

еквівалентна даній і має вигляд задачі про нерухому точку (7.16). Проблема тепер полягає лише в підборі матричного параметра ***A*** такого, при якому вектор-функція $\boldsymbol{\Phi}(\boldsymbol{x}) := \boldsymbol{x} - \mathrm{A}\boldsymbol{F}(\boldsymbol{x})$ мала б потрібні властивості.

Блок-схема алгоритму простих ітерацій представлена на рис. 7.2.

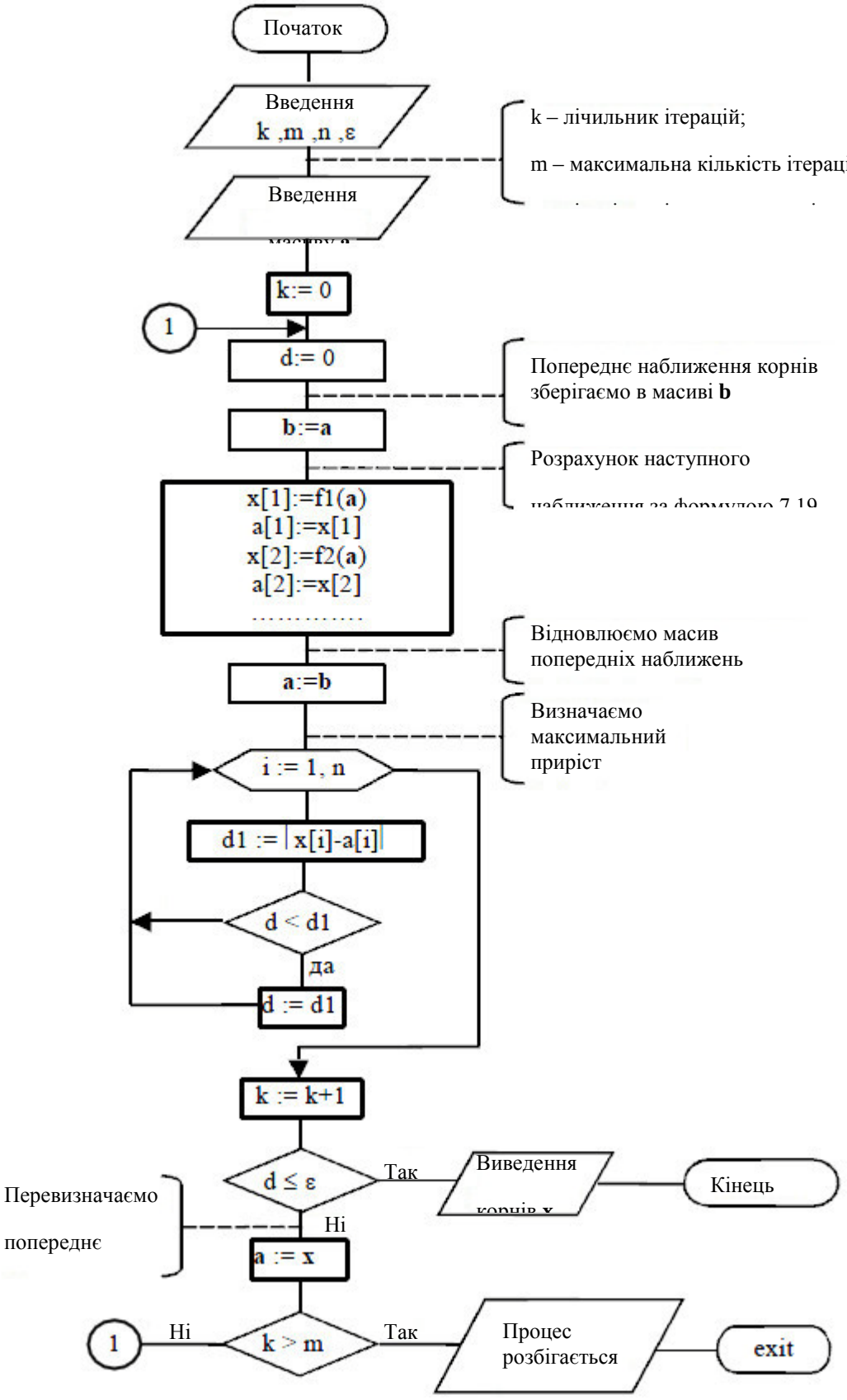


Рис. 7.2. Блок-схема алгоритму

Реалізація запропонованого алгоритму у вигляді модуля *Python* представлена на рис. 7.3.

```
# -*- coding: cp1251 -*-
from math import log,fabs
import numpy as np
from copy import deepcopy

def System(N,X):
    if N==1:
        return -0.1*X[1]**2-0.2*X[2]**2+0.3
    elif N==2:
        return -0.1*X[1]**2+0.1*X[1]*X[2]+0.7

def MPI(n,m,X,eps=1e-3):
    k=0

    while True:

        d=0; b=deepcopy(X); A=deepcopy(b)
        A[1]=System(1,X)
        X[1]=A[1]

        A[2]=System(2,X)
        X[2]=A[2]



        A=deepcopy(b)
```

```
for i in xrange(1,n+1):
    d1=fabs(X[i]-A[i])
    if d<d1:
        d=d1

k+=1

if (d<=eps):
    print "Solution is ",X,"\nnumber of iteration=",k
    break

A=deepcopy(X)

if k>m:
    print "Процес розбігається!"
    exit(0)
```

Рис. 7.3. Реалізація алгоритму

Модуль складається з двох функцій: System та MPI. Перша функція задає систему нелінійних рівнянь, друга – реалізація методу простої ітерації.

Програма використання розробленого модуля представлена на рис. 7.4. В якості прикладу використана система

$$\begin{cases} f_1(x_1, x_2) = 0.1x_1^2 + x_1 + 0.2x_2^2 - 0.3 = 0 \\ f_2(x_1, x_2) = 0.1x_1^2 + x_2 - 0.1x_1x_2 - 0.7 = 0 \end{cases}$$

з початковим наближенням: $x_1^{(0)} = 0.25$, $x_2^{(0)} = 0.75$.

```
# -*- coding: cp1251 -*-
import numpy as np
from Iter import *
```

```
X=np.array([0.,0.25,0.75])


n=2; m=10
MPI(n,m,X)
```

Рис. 7.4. Використання модуля

Результат виконання програми представлено на рис. 7.5.

```
>>>
Solution is  [ 0.          0.19532485  0.71005434]
number of iteration= 3
```

Рис. 7.5. Результат використання програми

7.2.3. Метод Брауна

На відміну від покрокової лінеаризації векторної функції $F(x)$, що привела до методу Ньютона (7.5), Брауном (1966 р.) запропоновано проводити на кожному ітераційному кроці почергову лінеаризацію компонент вектор-функції $F(x)$, тобто лінеаризовати в системі (7.1) спочатку функцію f_1, потім f_2 і так далі, і послідовно вирішувати отримувані таким чином рівняння. Аби не затінювати цю ідею громіздкими викладеннями і зайвими індексами, розглянемо виведення розрахункових формул методу Брауна в двовимірному випадку.

Нехай потрібно знайти вирішення системи

$$\begin{cases} f(x,y)=0, \\ g(x,y)=0, \end{cases} \tag{7.20}$$

і нехай вже отримані наближення x_k, y_k.

Підмінимо перше рівняння системи (7.20) лінійним, отриманим по формулі Тейлора для функції два змінних:

$$f(x,y) \approx f(x_k,x_k) + f'_x(x_k,x_k)(x-x_k) + f'_y(x_k,x_k)(y-y_k) = 0.$$

Звідси виражаємо $\boldsymbol{x}$ (позначимо цей результат через $\tilde{x}$):

$$\tilde{x} = x_k - \frac{1}{f'_x(x_k, y_k)}\left[f(x_k, y_k) + f'_y(x_k, y_k)(y - y_k)\right] \qquad (7.21)$$

При $y := y_k$ знаходимо значення $\tilde{x}_k$ змінної $\tilde{x}$:

$$\tilde{x}_k = x_k - \frac{f(x_k, y_k)}{f'_x(x_k, y_k)},$$

яке рахуватимемо лише проміжним наближенням (тобто не x_{k+1}), оскільки воно не враховує другого рівняння системи (7.20).

Підставивши в $g(x, y)$ замість ***x*** змінну $\tilde{x} = \tilde{x}(y)$, прийдемо до деякої функції *G(y):= g(*$\tilde{x}$*, y)* лише одній змінної ***y***. Це дозволяє лінеаризувати друге рівняння системи (7.20) за допомогою формули Тейлора для функції однієї змінної:

$$g(\tilde{x}, y) \approx G(y_k) + G'(y_k)(y - y_k) = 0 \qquad (7.22)$$

При знаходженні похідної ***G'(y)*** потрібно врахувати, що ***G(y)=g((y), y)*** є складною функцією однієї змінної ***y***, тобто потрібно застосувати формулу повної похідної.

Диференціюючи по ***y*** рівність (7.21), отримуємо вираз

$$\tilde{x}'_y = -\frac{f'_y(x_k, y_k)}{f'_x(x_k, y_k)},$$

підстановка якого в попередню рівність при $x = x_k, y = y_k$ дає

$$G'(y_k) = -g'_x(\tilde{x}_k, y_k) * \frac{f'_y(x_k, y_k)}{f'_x(x_k, y_k)} + g'_y(\bar{x}_k, y_k).$$

При відомих значеннях $G(y_k) = g(\tilde{x}_k, y_k)$ і $\mathbf{G'(}_{y_k}\mathbf{)}$ тепер можна вирішити лінійне рівняння (7.22) відносно ***y*** (назвемо набуті значення y_{k+1}):

$$y_{k+1} = y_k - \frac{G(y_k)}{G'(y_k)} = y_k - \frac{g(\tilde{x}_k, y_k) f'_x(x_k, y_k)}{f'_x(x_k, y_k) g'(\tilde{x}_k, y_k) - f'_y(x_k, y_k) g'_x(\tilde{x}_k, y_k)}$$

Замінюючи в (7.21) змінну ***y*** знайденим значенням y_{k+1}, приходимо до значення

$$x_{k+1} = \tilde{x}(y_{k+1}) = x_k - \frac{1}{f'_x(x_k, y_k)}\left[f(x_k, y_k) + f'_y(x_k, y_k)(y_{k+1} - y_k)\right]$$

Таким чином, реалізація методу Брауна вирішення двовимірних нелінійних систем вигляду (7.20) зводиться до наступного.

При вибраних початкових значеннях кожне x_0, y_0 подальше наближення по методу Брауна знаходиться при ***k=0,1,2***,... за допомогою сукупності формул

$$\tilde{x}_k = x_k - \frac{f(x_k, y_k)}{f'_x(x_k, y_k)},$$

$$q_k = \frac{g(\tilde{x}_k, y_k) f'_x(x_k, y_k)}{f'_x(x_k, y_k) g'(\tilde{x}_k, y_k) - f'_y(x_k, y_k) g'_x(\tilde{x}_k, y_k)}, \qquad (7.23)$$

$$p_k = \frac{f(x_k, y_k) - q_k f'(x_k, y_k)_y}{f'_x(x_k, y_k)},$$

$$x_{k+1} = x_k - p_k, \quad y_{k+1} = y_k - p_k$$

розрахунок за якими повинен виконуватись в тій послідовності, в якій вони записані.

Обчислення в методі Брауна природньо закінчити, коли виконається нерівність $\max\{|p_{k-1}|, |q_{k-1}|\} < \varepsilon$, (з результатом $(x^*, y^*) := \approx (x_k, y_k)$). В ході обчислень слід контролювати немаліть знаменників розрахункових формул. Зазначимо, що функції ***f*** і ***g*** в цьому методі нерівноправні, і зміна їх ролей може змінити ситуацію із збіжністю.

Вказуючи на наявність квадратичної збіжності методу Брауна, відзначають, що розраховувати на його велику ефективність в порівнянні з методом Ньютона в сенсі обчислювальних витрат можна лише у разі, коли частинні похідні, що фігурують в ньому замінюються різницевими відношеннями.

Написання реалізації методу Брауна (7.23) пропонується як самостійна вправа.

7.2.4. Метод січних Бройдена

Аби наблизитися до розуміння ідей, що лежать в основі пропонованого увазі методу, повернемося спочатку до одновимірного випадку.

В процесі побудови методів Ньютона і січних вирішення нелінійного скалярного рівняння

$$f(x, y) = 0 \tag{7.24}$$

функція ***f(x)*** в околиці поточної точки x_k підміняється лінійною функцією (аффінною моделлю)

$$\varphi_k(a_k, x) := f(x_k) + a_k(x - x_k) \tag{7.25}$$

Прирівнювання до нуля останнього, тобто вирішення лінійного рівняння

$$f(x_k) + a_k(x - x_k) = 0,$$

породжує ітераційну формулу

$$x_{k+1} = x_k - a_k^{-1} f(x_k) \tag{7.26}$$

для обчислення наближень до корня рівняння (7.24).

Якщо зажадати, аби замінююча функцію ***f(x)*** поблизу точки x_k аффінна модель $\varphi_k(a_k, x)$ мала в цій точці однакову з нею похідну, то, диференціюючи (7.25), отримуємо значення коефіцієнта

$$a_k = f'(x_k),$$

підстановка якого в (7.26) приводить до відомого методу Ньютона. Якщо ж виходити з того, що разом з рівністю $\varphi_k(a_k, x) = f(x_k)$ повинен мати місце збіг функцій ***f(x)*** і в попередній x_k точці x_{k-1} тобто з рівності

$$\varphi_k(a_k, x_{k-1}) = f(x_{k-1}),$$

або, відповідно до (7.25)

$$f(x_k) + a_k(x_{k-1} - x_k) = f(x_{k-1}), \tag{7.27}$$

то отримуємо коефіцієнт

$$a_k = \frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k},$$

що перетворює (7.26) на відому формулу січних.

Рівність (7.27), переписане у вигляді

$$a_k(x_{k-1} - x_k) = f(x_{k-1}) - f(x_k),$$

називають співвідношенням січних у R_1. Вона легко узагальнюється на ***n***-мірний випадок і лежить в основі виведення методу Бройдена. Опишемо це виведення.

У ***n***-мірному векторному просторі R_n співвідношення січних представляється рівністю

$$B_k(x^{(k-1)} - x^{(k)}) = F(x^{(k-1)}) - F(x^{(k)}), \qquad (7.28)$$

де $x^{(k-1)}, x^{(k)}$ — відомі ***n***-мірні вектори, $F: R_n \to R_n$ — дане нелінійне відображення, а B_k — деяка матриця лінійного перетворення в R_n. З позначеннями

$$s^{(k)} := x^{(k)} - x^{(k-1)}, \quad y^{(k)} := F(x^{(k)}) - F(x^{(k-1)}) \qquad (7.29)$$

співвідношення січних в R_n знаходить коротший запис:

$$B_k s^{(k)} = y^{(k)} \qquad (7.30)$$

Аналогічно одновимірному випадку, а саме, по аналогії з формулою (7.26), шукатимемо наближення до вирішення векторного рівняння (7.2) за формулою

$$x^{(k+1)} = x^{(k)} - B_k^{-1} F(x^{(k)}) \qquad (7.31)$$

Бажаючи, аби ця формула узагальнювала метод січних, зворотню ***n x n***-матрицю B_k в ній потрібно підібрати так, щоб вона задовольняла співвідношенню січних (7.28). Але це співвідношення не визначає однозначно матрицю B_k: дивлячись на рівність (7.30), легко зрозуміти, що при ***n>1*** існує безліч матриць, що перетворюють заданий ***n***-мірний вектор $s^{(k)}$ в інший заданий вектор $y^{(k)}$ (звідси — ясність в розумінні того, що можуть бути різні узагальнення одновимірного методу січних).

При формуванні матриці B_k міркуватимемо таким чином.

Переходячи від наявної в точці $x^{(k-1)}$ аффінної моделі функції ***F(x)***

$$\Phi_{k-1} := F(x^{(k-1)}) + B_{k-1}(x - x^{(k-1)}) \qquad (7.32)$$

до такої ж моделі в точці

$$\Phi_k := F(x^{(k)}) + B_k(x - x^{(k)}) \tag{7.33}$$

ми не маємо про матрицю лінійного перетворення B_k жодних відомостей, окрім співвідношення січних (7.28). Тому виходимо з того, що при цьому переході зміни в моделі мають бути мінімальними. Ці зміни характеризує різниця $\Phi_k - \Phi_{k-1}$. Віднімемо з рівності (7.33) визначальну Φ_{k-1} рівність (7.32) і перетворимо результат, залучаючи співвідношення січних (7.28). Маємо:

$$\Phi_k - \Phi_{k-1} := F(x^{(k)}) - F(x^{(k-1)}) + B_k(x - x^{(k)}) - B_{k-1}(x - x^{(k-1)}) =$$
$$= B_k(x^{(k)} - x^{(k-1)}) - B_k x^{(k)} + B_{k-1}x^{(k-1)} + (B_k - B_{k-1})x = (B_k - B_{k-1})(x - x^{(k-1)})$$

Представимо вектор $x - x^{(k-1)}$ у вигляді лінійної комбінації фіксованого вектора $s^{(k)}$ визначеного в (7.29), і деякого вектора ***t***, йому ортогонального:

$$x - x^{(k-1)} = \alpha s^{(k)} + t, \qquad \alpha \in R_1, t \in R_n : (t, s^{(k)}) = 0.$$

Підстановкою цього представлення вектора $x - x^{(k-1)}$ в різницю $\Phi_k - \Phi_{k-1}$ отримуємо інший її вигляд

$$\Phi_k - \Phi_{k-1} = \alpha(B_k - B_{k-1})s^{(k)} + (B_k - B_{k-1})t \tag{7.34}$$

Аналізуючи вираз (7.34), помічаємо, що перший доданок в ньому не може бути змінений, оскільки

$$(B_k - B_{k-1})s^{(k)} = B_k s^{(k)} - B_{k-1}s^{(k)} = y^{(k)} - B_{k-1}s^{(k)}$$

- фіксований вектор при фіксованому ***k***. Тому мінімальній зміні аффінної моделі Φ_{k-1} відповідатиме випадок, коли другий доданок в (7.34) буде нуль-вектором при будь-яких векторах ***t***, ортогональних векторам $s^{(k)}$, тобто B_k слід знаходити з умови

$$(B_k - B_{k-1})t = 0 \qquad \forall t : (t, s^{(k)}) = 0 \tag{7.35}$$

Безпосередньою перевіркою переконуємося, що умова (7.35) буде виконана, якщо матричну поправку $B_k - B_{k-1}$ взяти у вигляді однорангової ***nxn***-матриці

$$B_k - B_{k-1} = \frac{(y^{(k)} - B_{k-1}s^{(k)})(s^{(k)})^T}{(s^{(k)})^{T(k)}}.$$

Таким чином, приходимо до так званої ***формули перерахунку С. Бройдена*** (1965 р.)

$$B_k = B_{k-1} + \frac{(y^{(k)} - B_{k-1}s^{(k)})(s^{(k)})^T}{(s^{(k)})^T s^{(k)}}, \qquad (7.36)$$

яка дозволяє простими обчисленнями перейти від старої матриці B_{k-1} до нової B_k такої, аби виконувалося співвідношення січних (7.30) в новій точці і при цьому зміни в аффінній моделі (7.32) були б мінімальними.

Сукупність формул (7.31), (7.36) разом з позначеннями (7.29) називають *методом січних Бройдена* або просто методом Бройдена вирішення систем нелінійних числових рівнянь.

Хоча в методах січних звичайним є завдання двох початкових векторів ($x^{(0)}$ і $x^{(1)}$), для методу Бройдена характерний інший початок ітераційного процесу. Тут потрібно задати **один** початковий вектор $x^{(0)}$, початкову матрицю B_0 і далі в циклі по ***k=0,1,2,...*** послідовно виконувати наступні операції:

1. вирішити лінійну систему

$$B_k s^{(k+1)} = -F(x^{(k)}) \qquad (7.37)$$

відносно вектора $s^{(k+1)}$:

2. знайти вектори $x^{(k+1)}$ і $y^{(k+1)}$:

$$x^{(k+1)} = x^{(k)} + s^{(k+1)}, \quad y^{(k+1)} = F(x^{(k+1)}) - F(x^{(k)}); \qquad (7.38)$$

3. зробити перевірку на зупинку (наприклад, за допомогою перевірки на малість величин $\|s^{(k+1)}\|$ і $\|y^{(k+1)}\|$ і якщо потрібна точність не досягнута, обчислити нову матрицю B_k за формулою перерахунку (див. (7.36))

$$B_{k+1} = B_k + \frac{(y^{(k+1)} - B_k s^{(k+1)})(s^{(k+1)})^T}{(s^{(k+1)})^T s^{(k+1)}} \qquad (7.39)$$

В якості матриці B_0, що потребується рівністю (7.37) для запуску ітераційного процесу Бройдена, найчастіше беруть матрицю Якобі $F'(x^{(0)})$ або яку-небудь її апроксимацію. При цьому отримувані далі перерахунком (7.39) матриці B_1, B_2... не завжди можна вважати близькими до відповідних

матриць Якобі $F'(x^{(1)})$, $F'(x^{(2)})$,... (що може інколи зіграти корисну роль при виродженні матриць). Але, в той же час, показується, що при певних вимогах до матриць Якобі $F'(x^{(1)})$ матриці $F'(x^{(2)})$ мають «властивість обмеженого погіршення», що означає, що якщо і відбувається збільшення $\|B_k - F'(x^{(k)})\|$ із збільшенням номера ітерації $\boldsymbol{k}$, то досить повільний. За допомогою цієї властивості доводяться твердження про лінійну збіжність ($x^{(k)}$) до $\boldsymbol{x^*}$ при достатній близькості $x^{(0)}$ до $\boldsymbol{x^*}$ і B_0 до $F'(x^{(0)})$, а в тих припущеннях, при яких можна довести квадратичну збіжність методу Ньютона (7.5), — *про надлінійну збіжність послідовності наближень по методу Бройдена*.

Як і у випадках вживання інших методів вирішення нелінійних систем, перевірка здійснимості якихось умов збіжності ітераційного процесу Бройдена досить важка.

Формулі перерахунку (7.39) в ітераційному процесі можна надати більш простий вигляд.

Оскільки, в силу (7.37) та (7.38)

$$y^{(k+1)} - B_k s^{(k+1)} = F(x^{(k+1)}) - F(x^{(k)}) + F(x^{(k)}) = F(x^{(k+1)}),$$

а

$$(s^{(k+1)})^T s^{(k+1)} = (s^{(k+1)}, s^{(k+1)}) = \|s^{(k+1)}\|_2^2 = \|x^{(k+1)} - x^{(k)}\|_2^2,$$

то з формули (7.39) отримуємо формально еквівалентну до неї формулу перерахунку

$$B_{k+1} = B_k + \frac{F(x^{(k+1)})(x^{(k+1)} - x^{(k)})^T}{\|x^{(k+1)} - x^{(k)}\|_2^2}, \qquad (7.40)$$

яку можна використовувати замість (7.39) в сукупності з формулою (7.31) або з (7.37), (7.38) (без обчислення вектора $y^{(k+1)}$). Таке перетворення ітераційного процесу Бройдена трохи скорочує об'єм обчислень (на одне матрично-векторне множення на кожній ітерації). Але не потрібно забувати, що при заміні формули (7.39) формулою (7.40) може змінитися ситуація з обчислювальною стійкістю методу; на щастя, це трапляється тут украй рідко, а саме, в тих випадках, коли для здобуття рішення з

потрібною точністю потрібно багато ітерацій по методу Бройдена, тобто коли і застосовувати його не варто.

Реалізація цього методу також пропонується як самостійна вправа.

7.3. Про вирішення нелінійних систем методами спуску

Загальний недолік всіх розглянутих вище методів вирішення систем нелінійних рівнянь — це суто локальний характер збіжності, що утрудняє їх вживання у випадках, коли є проблеми з вибором хороших початкових наближень. Допомога тут може прийти з боку чисельних методів оптимізації — гілки обчислювальної математики, що зазвичай виділяється в самостійну дисципліну. Для цього потрібно поставити завдання знаходження розв'язків даної нелінійної системи як оптимізаційне або, інакше, екстремальне завдання. Для геометричної інтерпретації міркувань, що проводяться нижче, і їх результатів обмежимося розглядом системи, що складається з двох рівнянь з двома невідомими (7.41).

З функцій *f* u *g* системи (4.3.1) утворюємо нову функцію

$$\Phi(x, y) := f^2(x, y) + g^2(x, y). \qquad (7.41)$$

Оскільки ця функція ненегативна, то знайдеться точка ***(x*, y*)*** така, що

$$\Phi(x, y) \geq \Phi(x^*, y^*) \geq 0 \quad \forall (x, y) \in R_2,$$

тобто ***(x*, y*)= argminΦ(x, y)***. Отже, якщо тим або іншим способом вдається отримати точку ***(x*, y*)***, що мінімізує функцію ***Φ(x,y)***, і якщо при цьому виявиться, що ***minΦ(x,y)=Φ(x*, y*)=0***, то ***(x*, y*)*** – шуканий розв'язок системи (7.20), оскільки

$$\Phi(x^*, y^*) = 0 \Leftrightarrow \begin{cases} f(x^*, y^*) = 0, \\ g(x^*, y^*) = 0. \end{cases}$$

Послідовність точок (x_k, y_k) — наближень до точки ***(x*; y*)*** мінімуму ***Φ(x, y)*** — зазвичай отримують по рекурентній формулі

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + a_k \begin{pmatrix} p_k \\ q_k \end{pmatrix}, \quad k=0,1,2,..., \qquad (7.42)$$

де $(p_k; q_k)^T$ — вектор, що визначає напрям мінімізації, а a_k – скалярна величина, що характеризує розмір кроку мінімізації (кроковий множник). Враховуючи геометричний сенс завдання мінімізації функції двох змінних ***Ф(x,y)*** — «спуск на дно» поверхні *z=Ф(x,y)* (див. рис. 7.5), ітераційний метод (7.42) можна назвати ***методом спуску***, якщо вектор $(p_k; q_k)^T$ при кожному ***k*** є напрямом спуску (тобто існує $\alpha > 0$ таке, що $\Phi(x_k + \alpha p_k, y_k + \alpha q_k) < \Phi(x_k, y_k)$ і якщо множник a_k підбирається так, щоб виконувалася ***умова релаксації*** $\Phi(x_{k+1}, y_{k+1}) < \Phi(x_k, y_k)$, що означає перехід на кожній ітерації в точку з меншим значенням функції, що мінімізується.

Отже, при побудові чисельного методу вигляду (7.42) мінімізації функції ***Ф(x, y)*** слід відповісти на два головні питання: як вибирати напрям спуску a_k і як регулювати довжину кроку у вибраному напрямі за допомогою скалярного параметра — крокового множника a_k. Приведемо найбільш прості міркування із цього приводу.

При виборі напряму спуску природним є вибір такого напряму, в якому функція, що мінімізується, убуває найшвидше. Як відомо з математичного аналізу функцій декількох змінних, напрям найбільшого зростання функції в даній точці показує її градієнт в цій точці. Тому приймемо за напрям спуску вектор

$$\begin{pmatrix} p_k \\ q_k \end{pmatrix} := -grad\Phi(x_k, y_k) = -\begin{pmatrix} \Phi'_x(x_k, y_k) \\ \Phi'_y(x_k, y_k) \end{pmatrix}$$

— антиградієнт функції ***Ф(x, y)***. Таким чином, з сімейства методів (7.42) виділяємо градієнтний метод

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} := \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \alpha_k \begin{pmatrix} \Phi'_x(x_k, y_k) \\ \Phi'_y(x_k, y_k) \end{pmatrix}. \qquad (7.43)$$

Оптимальний крок у напрямі антиградієнта — це такий крок, при якому значення $\Phi(x_{k+1}, y_{k+1})$ — найменше серед всіх інших значень ***Ф(x,y)*** в цьому фіксованому напрямі, тобто коли точка (x_{k+1}, y_{k+1}) є точкою умовного мінімуму. Отже, можна розраховувати на найбільш швидку збіжність методу (7.43), якщо використати в ньому такий вибір крокового множника, який

називається *вичерпним спуском*. Такий вибір кроку разом з формулою (7.43) визначає *метод найшвидшого спуску*.

Геометричну інтерпретацію цього методу добре видно з рис. 7.5, 7.6. Характерні дев'яностоградусні злами траєкторії найшвидшого спуску, що пояснюється вичерпністю спуску та властивістю градієнта (а значить, і антиградієнта) бути перпендикулярним до дотичної до лінії рівня у відповідній точці.

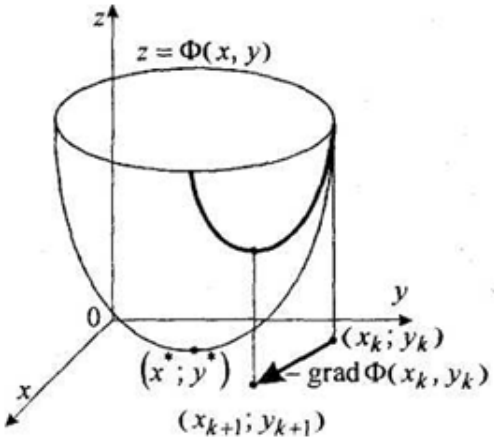


Рис. 7.5. Просторова інтерпретація методу найшвидшого спуску для функції (7.41)

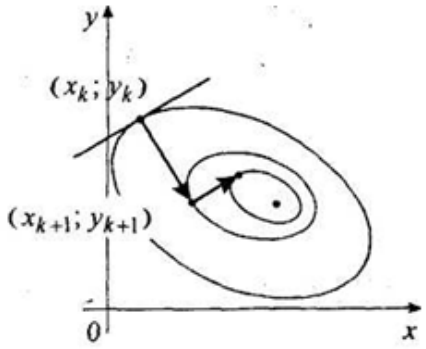


Рис. 7.6. Траєкторія найшвидшого спуску для функції (7.41)

Найбільш типовою є ситуація, коли знайти точне (аналітичними методами) оптимальне значення $\boldsymbol{a_k}$ не вдається. Отже, доводиться робити ставку на використання яких-небудь чисельних методів одновимірної мінімізації і знаходити $\boldsymbol{a_k}$ лише приблизно.

Не дивлячись на те, що завдання знаходження мінімуму функції однієї змінної $\varphi_k(\alpha) = \Phi(x_k - \alpha\Phi'_x(x_k, y_k), y_k - \alpha\Phi'_y(x_k, y_k))$ набагато простіше, ніж

вирішуване завдання, вживання тих або інших чисельних методів знаходження значень $\alpha_k = \arg\min \varphi_k(\alpha)$ з тією або іншою точністю вимагає обчислення декількох значень функції, що мінімізується. Оскільки це потрібно робити на кожному ітераційному кроці, то при великому числі кроків реалізація методу найшвидшого спуску в чистому вигляді є досить високовитратною. Існують ефективні схеми наближеного обчислення квазіоптимальних $\boldsymbol{a_k}$, в яких враховується специфіка функцій, що мінімізуються (типа сум квадратів функцій).

Частенько успішною є така стратегія градієнтного методу, при якій кроковий множник $\boldsymbol{a_k}$ в (7.43) береться або відразу досить малим постійним, або передбачається його зменшення, наприклад, діленням навпіл для задоволення умови релаксації на черговому кроці. Хоча кожен окремий крок градієнтного методу при цьому, взагалі кажучи, далекий від оптимального, такий процес по кількості обчислень функції може виявитися ефективнішим, ніж метод найшвидшого спуску.

Головна перевага градієнтних методів вирішення нелінійних систем — глобальна збіжність. Неважко довести, що процес градієнтного спуску приведе до якої-небудь точки мінімуму функції з будь-якої початкової точки. За певних умов знайдена точка мінімуму буде шуканим вирішенням вихідної нелінійної системи.

Головний недолік — повільна збіжність. Доведено, що збіжність таких методів — лише лінійна, причому, якщо для багатьох методів, таких як метод Ньютона, характерне прискорення збіжності при наближенні до рішення, то тут має місце швидше зворотне. Тому є резон в побудові гібридних алгоритмів, які починали б пошук шуканої точки, — вирішення даної нелінійної системи — градієнтним методом, що глобально сходиться, а потім проводили уточнення якимсь швидкозбіжним методом, наприклад, методом Ньютона (зрозуміло, якщо дані функції володіють потрібними властивостями).

На даний момент розроблено ряд методів вирішення екстремальних завдань, які сполучають в собі низьку вимогливість до вибору початкової точки і високу швидкість збіжності. До таких методів, що називають квазіньютонівськими, можна віднести, наприклад, метод змінної метрики (Девідона-Флетчера-Пауела), симетричний і позитивно визначений методи січних.

За наявності нерівних функцій у вирішуваному завданні слід відмовитися від використання похідних або їх апроксимацій і вдатися до так званих методів прямого пошуку (циклічного покоординатного спуску, Хука і Дживса, Розенброка і тому подібне).

7.4 Контрольні запитання

Розділ 8. Чисельне інтегрування функцій

Відомо, що для переважної більшості функцій не вдається обчислити первісні функції, внаслідок чого доводиться вдаватися до методів наближеного і чисельного інтегрування функцій.

При чисельному інтегруванні по заданій підінтегральній функції будується сіткова функція :

x_i	x_0	x_1	...	x_n
y_i	y_0	y_1	...	y_n

яка потім за допомогою формул локальної інтерполяції з контрольованою похибкою замінюється інтерполяційним многочленом, і інтеграл від якої добре обчислюється і порівняно легко оцінюється похибка.

Нехай на відрізку $x \in [a,b]$ задана неперервна функція $y = f(x)$, і потрібно на цьому відрізку обчислити визначений інтеграл:

$$I = \int_a^b f(x)dx \tag{8.1}$$

Замінимо дану функцію на сіткову. Замість точного значення інтеграла I шукатимемо його наближене значення за допомогою суми $I \approx I_h = \sum_{i=0}^{n} A_i h_i$,

де $h_i = x_i - x_{i-1},\ i = \overline{1,n},\ x_0 = a,\ x_n = b$, в якій необхідно визначити коефіцієнти A_i і похибку формули.

8.1. Метод прямокутників

Найбільш простою (і неточною) є формула прямокутників. Вона може бути отримана на основі визначення інтеграла, як границі послідовності інтегральних сум:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(\xi_i)\Delta x_i,\ \xi_i \in [x_i, x_{i-1}],\quad \Delta x_i = x_i - x_{i-1}\ . \tag{8.2}$$

Якщо в цьому визначенні зняти знак границі і покласти $\Delta x_i = h_i$, $i = \overline{1,n}$, то з'явиться похибка R_{np} (за ξ_i можна прийняти лівий або правий кінець відрізка Δx_i), тобто

$$\int_a^b f(x)dx = \sum_{i=1}^{n} f(x_{i-1})h_i + R_{np} \quad \text{або} \quad \int_a^b f(x)dx = \sum_{i=1}^{n} f(x_i)h_i + R_{np}\,. \tag{8.3}$$

Одержані формули (8.3) називаються, відповідно, *формулами лівих і правих прямокутників* чисельного інтегрування.

Розглянемо похибку R_i формули лівих прямокутників на одному кроці $[x_{i-1}, x_i]$ чисельного інтегрування.

Для цього припустимо, що первісна функція $F(x)$ для підінтегральної функції $f(x)$ (вона існує, оскільки $f(x)$ – неперервна на відрізку $x \in [a,b]$) неперервно-диференційована. Тоді, розкладаючи $F(x)$ в околі вузла x_{i-1} в ряд Тейлора до другої похідної включно, і використовуючи рівність $F'(x) = f(x) = y(x)$, отримаємо:

$$\begin{aligned} R_i &= \int_{x_{i-1}}^{x_i} f(x)dx - y_{i-1}h = [F(x_i) - F(x_{i-1})] - y_{i-1}h = [F'(x_{i-1})h + F''(\xi)\frac{h^2}{2}] - y_{i-1}h = \\ &= [y_{i-1}h + y'(\xi)\frac{h^2}{2}] - y_{i-1}h = y'(\xi)\frac{h^2}{2}, \quad \xi \in (x_{i-1}, x_i). \end{aligned} \tag{8.4}$$

На всьому відрізку $[a,b]$ цю похибку необхідно підсумувати n разів ($b-a=nh$), отримаємо

$$R_{np} = R_i n = y'(\xi)\frac{(b-a)h}{2}, \quad \xi \in (a,b) \tag{8.5}$$

Оскільки місцеположення точки ξ на інтервалі (a,b) не відоме, то на основі виразу для похибки (8.5) можна виписати верхню оцінку абсолютної похибки методу прямокутників і при заданій точності ε методу виписати нерівність:

$$\left|R_{np}\right| \le \frac{(b-a)h}{2} M_1 \le \varepsilon, \quad M_1 = \max_{x\in[a,b]} \left|f'(x)\right| \tag{8.6}$$

яку можна використовувати для верхньої оцінки кроку h чисельного інтегрування по методу прямокутників:

$$h \le \frac{2\varepsilon}{(b-a)M_1}, \quad M_1 = \max_{x\in[a,b]} |f'(x)| \tag{8.7}$$

З виразу для похибки випливає, що на кожному відрізку $[x_{i-1}, x_i]$ формула прямокутників має похибку, пропорційну h^2, а на всьому відрізку $[a,b]$ - кроку чисельного інтегрування h. Тому кажуть, що *мет од прямокут ників є мет одом першого порядку т очност і (головний член похибки пропорційний кроку в першому ст епені).*

Алгоритм (рис. 8.1) запропонованого метода складається з наступних основних кроків:

1. Весь участок ***[a,b]*** ділиться на n рівних частин з кроком ***h=(b-a)/n***.
2. Визначається значення y_i підінтегральної функції ***f(x)*** в кожній частині ділення, тобто: $y_i = f(x_i), i = 0, n$.
3. В кожній частині ділення підінтегральна функція апроксимується інтерполяційним багаточленом степеня ***n = 0***, тобто прямою, що паралельна осі ***OX***. В результаті вся підінтегральна функція на ділянці ***[a,b]*** апроксимується ламаною лінією.
4. Для кожної частини ділення визначається площа S_i часткового прямокутника.

5. Сумуються усі площі. Наближене значення інтегралу ***I*** дорівнює сумі площ часткових прямокутників.

В алгоритмі введено такі позначення (рис. 8.1):

- ***a,b*** – кінці інтервалу;
- ***ε*** – задана точність;
- ***c=0*** – метод лівих прямокутників;
- ***c=1*** – метод правих прямокутників;
- S_1 – значення інтегралу на попередньому кроці;
- ***S*** – значення інтегралу на поточному кроці.

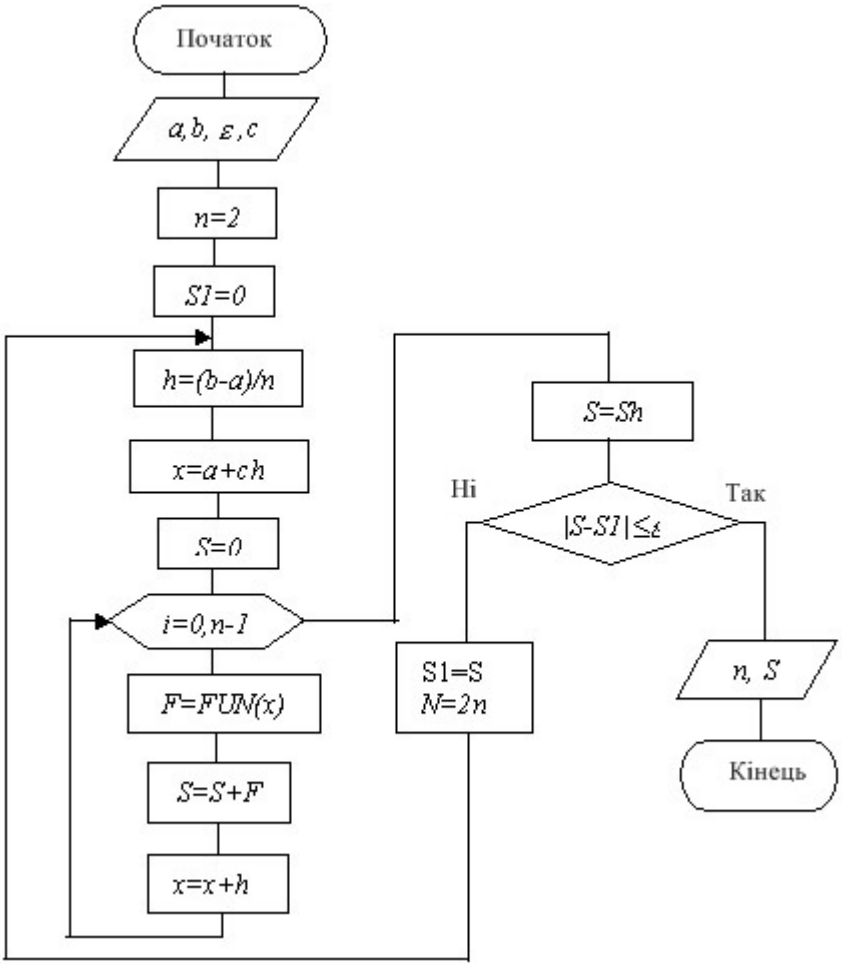


Рис. 8.1. Блок-схема алгоритму методу прямокутників

Реалізація запропонованого алгоритму представлена на рис. 8.2.

```
from math import sin,pi,fabs
from numpy import zeros


def f(x):
    return (1/x)*sin(pi*x/2)


def prjam(a,b,n=10,epsilon=1e-4,c=0):
    h=(b-a)/n
    xb=a+c*h
    s=zeros((n)); print s


    print "N-->x-->F-->s-->s1"
```

```
    for i in range(n+1):
        x=xb+i*h
        s[i]=s[i-1]+f(x)*h

        print "%d-->%.9f-->%.9f-->%.9f-->%.9f" %(i,x,f(x),s[i],s[i-1])
        if (fabs(s[i-1]-s[i])<epsilon):
            break
    return n,s

a=1.;b=2.;n=100;
N,S=prjam(a,b,n=1000)
```

Рис. 8.2. Реалізація алгоритму

8.2. Метод трапецій

Розглянемо інтеграл $I = \int_a^b f(x)dx$ на відрізку $x \in [x_{i-1}, x_i]$ і на цьому відрізку обчислюватимемо його приблизно, замінюючи підінтегральну функцію інтерполяційним многочленом Лагранжа першого степеня, отримаємо:

$$\int_{x_{i-1}}^{x_i} f(x)dx = \int_{x_{i-1}}^{x_i} L_1(x)dx + R_i \qquad (8.8)$$

де R_i – похибка, яка підлягає визначенню (на рис. 8.3 – заштрихована область), а L_1 – інтерполяційний многочлен Лагранжа першого степеня, проведений через два вузли інтерполяції x_{i-1} і x_i.

$$L_1(x) = y_{i-1}\frac{x - x_i}{x_{i-1} - x_i} + y_i\frac{x - x_{i-1}}{x_i - x_{i-1}}$$

Нехай $x_i - x_{i-1} = h = const$, де $i = \overline{1,n}$.

Позначимо $\frac{x-x_{i-1}}{h}=t$, тоді $\frac{x-x_i}{h}=\frac{(x-x_{i-1})-(x_i-x_{i-1})}{h}=t-1$, $dx=hdt$ і многочлен Лагранжа прийме вигляд $L_1(x)=L_1(x_{i-1}+ht)=-y_{i-1}(t-1)+y_it$. При $x=x_i$ верхня границя $t=1$, при $x=x_{i-1}$ нижня границя $t=0$.

Тепер інтеграл $\int_{x_{i-1}}^{x_i} L_1(x)dx$ від многочлена $L_1(x)$ можна представити у вигляді:

$$\int_{x_{i-1}}^{x_i} f(x)dx \approx \int_{x_{i-1}}^{x_i} L_1(x)dx = h\int_0^1 [-y_{i-1}(t-1)+y_it]dt = h[-y_{i-1}(\frac{t^2}{2}-t)+y_i\frac{t^2}{2}]_0^1 = \frac{h}{2}(y_{i-1}+y_i) \quad (8.9)$$

Цей вираз називають *формулою трапецій* чисельного інтегрування на відрізку $x\in[x_{i-1},x_i]$ (див. рис. 8.3).

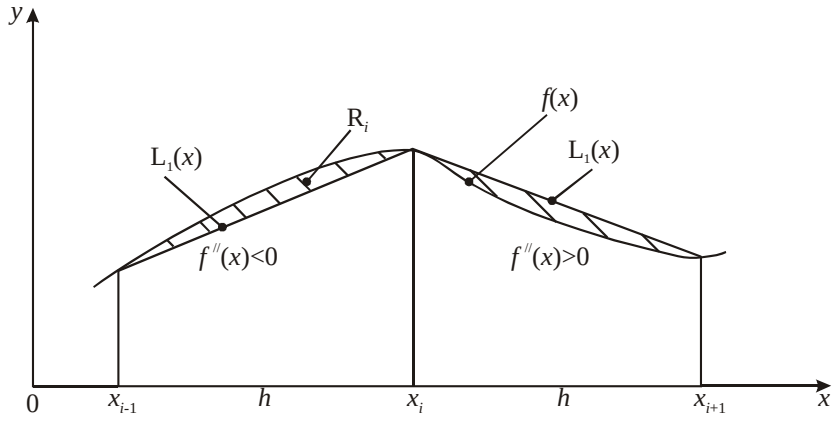


Рис. 8.3. Метод трапецій

Для всього відрізка $[a,b]$ необхідно скласти цей вираз n разів.

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(x_0)+f(x_n)+2\sum_{i=1}^{n-1} f(x_i)) = \frac{h}{2}(y_0+y_n+2\sum_{i=1}^{n-1} y_i) \quad (8.10)$$

Отриманий вираз (8.10) називають *формулою трапецій* чисельного інтегрування для всього відрізка $[a,b]$.

У разі змінного кроку метод трапецій використовується в наступному вигляді (8.11):

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{f(x_{i-1})+f(x_i)}{2}h_i, \quad h_i=x_i-x_{i-1}. \quad (8.11)$$

Похибку R_i формули трапецій на відрізку $[x_{i-1},x_i]$ можна отримати, інтегруючи похибку лінійної апроксимації:

$$R_i = -\frac{h^3}{12} y''(\xi), \quad \text{де } \xi \in (x_{i-1}, x_i). \tag{8.12}$$

Для цього позначимо $\bar{x}_i = \frac{x_i + x_{i+1}}{2}$ серединну точку відрізка $[x_{i-1}, x_i]$ і розкладемо *f(x)* по степеням $(x - \bar{x}_i)$ за формулою Тейлора, припускаючи, що вона має п'ять неперервних похідних. Маємо:

$$f(x) = f(\bar{x}_i) + (x - \bar{x}_i) f'(\bar{x}_i) + \frac{1}{2}(x - \bar{x}_i)^2 f''(\bar{x}_i) + \frac{1}{6}(x - \bar{x}_i)^3 f'''(\bar{x}_i) + \frac{1}{24}(x - \bar{x}_i)^4 f^{(4)}(\bar{x}_i) + \ldots$$

Інтегруючи *f(x)* на відрізку $[x_{i-1}, x_i]$, маємо:

$$\int_{x_i}^{x_{i+1}} f(x)dx = \int_{x_i}^{x_{i+1}} (f(\bar{x}_i) + (x - \bar{x}_i) f'(\bar{x}_i) + \frac{1}{2}(x - \bar{x}_i)^2 f''(\bar{x}_i) + \frac{1}{6}(x - \bar{x}_i)^3 f'''(\bar{x}_i) + \frac{1}{24}(x - \bar{x}_i)^4 f^{(4)}(\bar{x}_i) +)dx =$$

$$h_i f(\bar{x}_i) + \frac{1}{24} h^3 f'''(\bar{x}_i) + \frac{1}{1920} h_i^5 + \ldots$$

Інтеграли парних степенів перетворюються в нуль.

Це означає, що коли h_i мале, то похибка інтегрування на відрізку за формулою прямокутників має порядок $\frac{1}{24} h^3 f'''(\bar{x}_i)$.

Щоб оцінити порядок похибки для формули трапецій, знову використаємо розклад за формулою Тейлора. Підставляючи в нього значення $x = x_i$ і $x = x_{i+1}$, одержимо:

$$f(x_i) = f(\bar{x}_i) - \frac{1}{2} h_i f'(\bar{x}_i) + \frac{1}{8} h_i^2 f''(\bar{x}_i) - \frac{1}{48} h_i^3 f'''(\bar{x}_i) + \frac{1}{384} h_i^4 f^{(4)}(\bar{x}_i) + \ldots$$

$$f(x_{i+1}) = f(\bar{x}_i) + \frac{1}{2} h_i f'(\bar{x}_i) + \frac{1}{8} h_i^2 f''(\bar{x}_i) + \frac{1}{48} h_i^3 f'''(\bar{x}_i) + \frac{1}{384} h_i^4 f^{(4)}(\bar{x}_i) + \ldots$$

Звідси маємо:

$$\frac{f(x_i) + f(x_{i+1})}{2} = f(\bar{x}_i) + \frac{1}{8} h_i^2 f''(\bar{x}_i) + \frac{1}{384} h_i^4 f^{(4)}(\bar{x}_i) + \ldots$$

Об'єднуючи це з розкладом інтеграла, одержуємо:

$$\int_{x_i}^{x_{i+1}} f(x)dx = h_i \frac{f(x_i) + f(x_{i+1})}{2} - \frac{1}{12} h^3 f''(\bar{x}_i) - \frac{1}{480} h_i^5 f^{(4)}(\bar{x}_i) + \ldots \tag{8.13}$$

Таким чином, при малих h_i похибка інтегрування на відрізку за формулою трапецій має порядок $-\frac{1}{12} h^3 f''(\bar{x}_i)$. Порівнюючи це з оцінкою похибки для формули прямокутників, можна зауважити, як не дивно, але формула прямокутників приблизно вдвічі точніша за формулу трапецій.

На всьому відрізку похибку необхідно збільшити в ***n*** разів:

$$R_{mp} \approx -\frac{nh^3}{12} y''(\xi) = -\frac{(nh)h^2}{12} y''(\xi) = -\frac{b-a}{12} h^2 y''(\xi),\ \xi \in (a,b). \qquad (8.14)$$

Вираз для оцінки похибки зазвичай записується таким чином:

$$\left|R_{mp}\right| \le \max_{x\in[a,b]} \left| f''(x) \right| \frac{b-a}{12} h^2. \qquad (8.15)$$

Звідки, задаючи точність чисельного інтегрування, можна записати наступну нерівність, використовувану для визначення кроку h чисельного інтегрування:

$$h \le \sqrt{\frac{12 \cdot \varepsilon}{(b-a) M_2}}, \qquad M_2 = \max_{x\in[a,b]} \left| f''(x) \right|. \qquad (8.16)$$

Чисельне інтегрування за методом трапецій у разі заданої точності ε може бути проведене таким чином:

1) за формулою (8.16) визначається крок чисельної інтеграції h;

2) за допомогою цього кроку складається сіткова функція для підінтегральної функції $f(x)$;

3) обчислюється наближене значення інтеграла за формулою $\int_a^b f(x)dx \approx \frac{h}{2}(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i)$, крок h в якій гарантує задану точність ε.

Блок-схема алгоритму методу трапецій представлено на рис. 8.4.

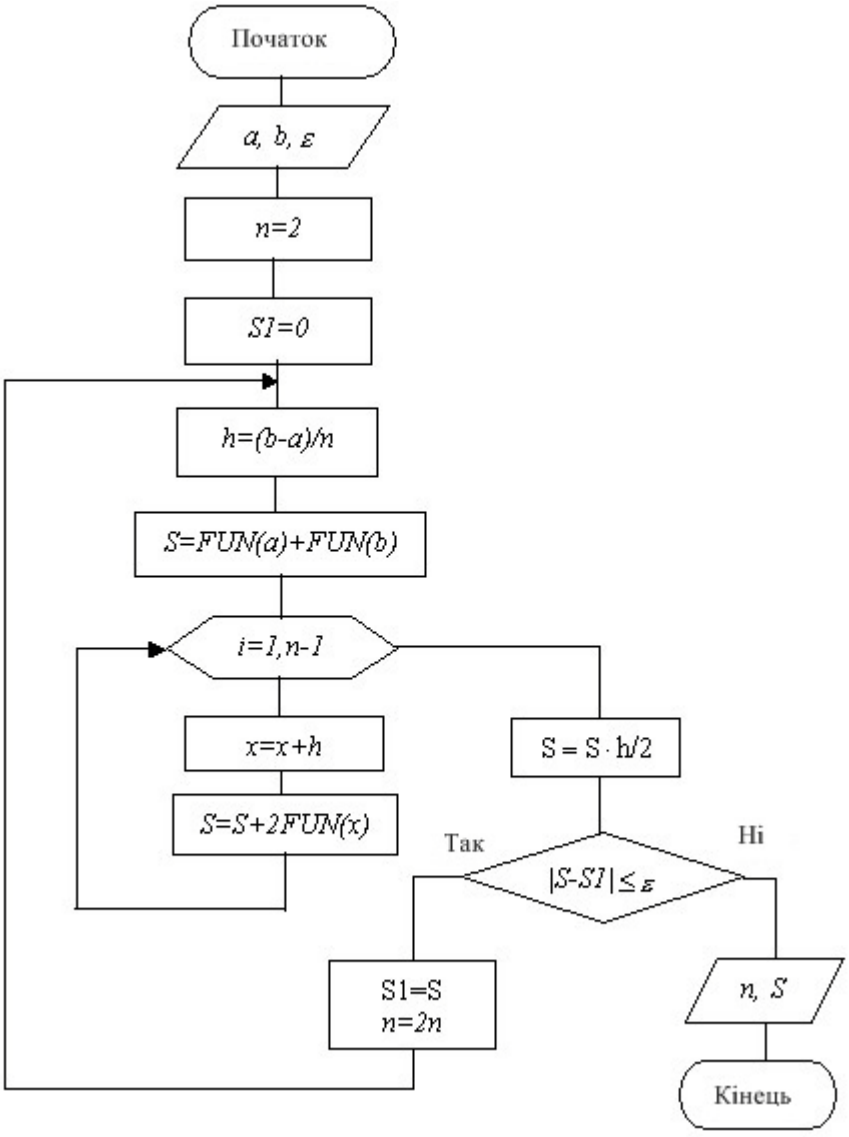


Рис. 8.4. Блок-схема алгоритму методу трапецій

Одна з можливих реалізацій методу трапецій у вигляді модуля на мові *Python* представлена на рис. 8.5.

```
# -*- coding: cp1251 -*-

## module trapezoid

''' Inew = trapezoid(f,a,b,Iold,k).

    Рекурсивна формула трапецій:

    Iold = Інтеграл f(x) від x = a до b, який визначається

    методом трапецій з 2^(k-1) ділянками.

    Inew = Той же інтеграл, але ділянок 2^k.

'''
```

```
def trapezoid(f,a,b,Iold,k):
  if k == 1:Inew = (f(a) + f(b))*(b - a)/2.0
  else:
    n = 2**(k -2 )     # Кількість нових точок
    h = (b - a)/n      # Крок між точками
    x = a + h/2.0      # Координати першої точки
    sum = 0.0
    for i in range(n):
      sum = sum + f(x)
      x = x + h
    Inew = (Iold + h*sum)/2.0
  return Inew
```

Рис. 8.5. Реалізація методу трапецій

Розглянемо застосування розробленого модуля на прикладі вирішення завдання (приклад 8.1).

Приклад 8.1. Визначити інтеграл $\int_0^{\pi} \sqrt{x} cosxdx$ з точністю до 6 знаку. Визначити, скільки потрібно ділянок розбиття для досягнення такої точності. На рис. 8.6 представлена програма, яка розв'язує поставлене завдання.

```
# -*- coding: cp1251 -*-
from math import sqrt,cos,pi
from trapezoid import *

def f(x): return sqrt(x)*cos(x)

Iold = 0.0
for k in range(1,21):
  Inew = trapezoid(f,0.0,pi,Iold,k)
  if (k > 1) and (abs(Inew - Iold)) < 1.0e-6: break
```

```
    Iold = Inew
print "Інтеграл =",Inew
print "nДілянок =",2**(k-1)
raw_input("\nНажати ввід для виходу з програми")
```

Рис. 8.6. Приклад застосування модуля

Результат роботи програми представлено на рис. 8.7.

```
>>>
Інтеграл = -0.894831664853
nДілянок = 32768

Нажати ввід для виходу з програми
```

Рис. 8.7. Результат роботи програми

Як бачимо, потрібно *32768* розбиттів, щоб досягнути поставленої точності.

8.3. Метод Сімпсона

Розіб'ємо відрізок $[a,b]$ на m пар відрізків $b-a=nh=2mh$, $h=x_i-x_{i-1}=const, i=\overline{1,n}$ і через кожні три вузли проведемо інтерполяційний многочлен Лагранжа (див. рис. 8.8).

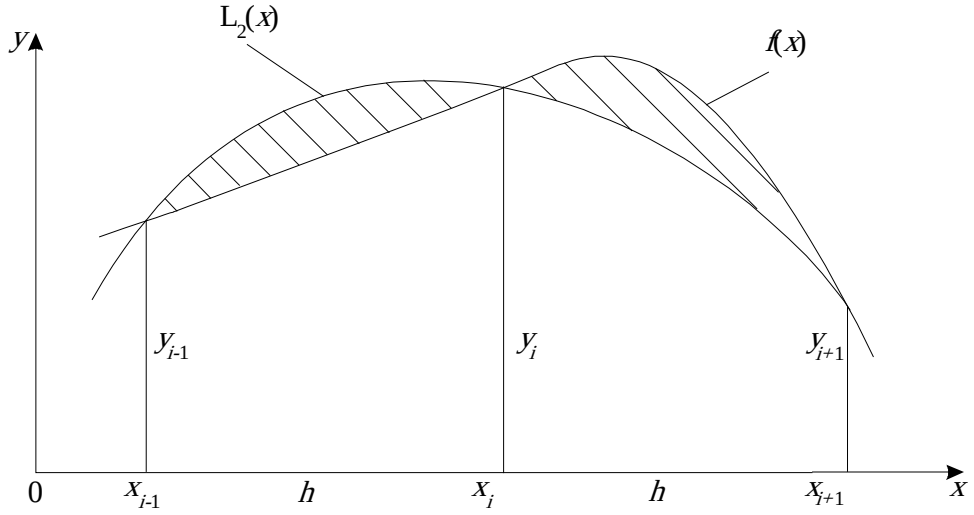


Рис. 8.8. Метод Сімпсона

Тоді $\int_{x_{i-1}}^{x_{i+1}} f(x)dx = \int_{x_{i-1}}^{x_{i+1}} L_2(x)dx + R_i$,

(8.17)

де $L_2(x) = y_{i-1}\frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} + y_i\frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} + y_{i+1}\frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)}$

Зробимо заміну $\frac{x-x_{i-1}}{h} = t,\ dx = hdt$ і тоді маємо:

$$\frac{x-x_i}{h} = \frac{(x-x_{i-1})-(x_i-x_{i-1})}{h} = t-1;\ \frac{x-x_{i+1}}{h} = \frac{(x-x_{i-1})-(x_{i+1}-x_{i-1})}{h} = t-2. \quad (8.18)$$

Доданки в $L_2(x)$ приймуть вигляд:

$$y_{i-1}\frac{(x-x_i)(x-x_{i+1})}{h\cdot 2h} = (t-1)(t-2)\frac{y_{i-1}}{2};\qquad y_i\frac{(x-x_{i-1})(x-x_{i+1})}{-h\cdot h} = -t(t-2)y_i;$$

$$y_{i+1}\frac{(x-x_{i-1})(x-x_i)}{2h\cdot h} = \frac{t}{2}(t-1)y_{i+1}.$$

При $x = x_{i-1}$: $t = 0$ $x = x_{i+1}$: $t = 2$.

Тоді $\int_{x_{i-1}}^{x_{i+1}} L_2(x)dx = h\int_0^2\left[\frac{y_{i-1}}{2}(t-1)(t-2) - y_i t(t-2) + \frac{y_{i+1}t}{2}(t-1)\right]dt = \frac{h}{3}(y_{i-1}+4y_i+y_{i+1})$

Звідки $\int_{x_{i-1}}^{x_{i+1}} f(x)dx \approx \int_{x_{i-1}}^{x_{i+1}} L_2(x)dx = \frac{h}{3}(y_{i-1}+4y_i+y_{i+1})$.

(8.19)

На всьому відрізку вираз необхідно скласти ***m*** разів, оскільки є ***m*** пар відрізків довжиною ***h***. Отримаємо *формулу Сімпсона чисельного інт егрування*:

$$\int_a^b f(x)dx \approx \frac{h}{3}(y_0 + y_n + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i}).$$

(8.20)

Похибка формули Сімпсона на подвійному кроці пропорційна *4-ій* похідній функції і п'ятому степеню кроку :

$$R_i = \int_{x_{i-1}}^{x_{i+1}} f(x)dx - \frac{h}{3}(y_{i-1}+4y_i+y_{i+1}) = -\frac{h^5}{90}f^{IV}(\xi),\quad \xi\in(x_{i-1},x_{i+1})$$

(8.21)

Це випливає з того, що формула Сімпсона *S(f)* є комбінацією формул прямокутників *R(f)* і трапецій *T(f),* а саме: $S(f) = \frac{2}{3}R(f) + \frac{1}{3}T(f)$, в результаті якої

кубічні члени скорочуються, а найстаршим залишається член п'ятого степеня.

Для всього відрізка $[a,b]$ цю похибку необхідно помножити на m пар відрізків:

$$R_c \approx -\frac{mh^5}{90} f^{IV}(\xi) = -\frac{2mh^5}{180} f^{IV}(\xi) = -\frac{nh \cdot h^4}{180} f^{IV}(\xi) = -\frac{(b-a)h^4}{180} f^{IV}(\xi), \quad \xi \in (a,b),$$

тобто у формулі Сімпсона на всьому відрізку $[a,b]$ *похибка пропорційна четвертому степеню кроку і, отже, метод Сімпсона є методом четвертого порядку точності (головний член похибки пропорційний четвертому степеню кроку h).*

Оскільки положення точки ξ на відрізку $[a,b]$ не відоме, то доцільно використовувати верхню оцінку похибки:

$$|R_c| \le \frac{(b-a)h^4}{180} M_4, \quad M_4 = \max_{x\in[a,b]} |f^{IV}(x)|,$$

звідки при заданій точності ε можна отримати

$$h \le \sqrt[4]{\frac{180\varepsilon}{(b-a)M_4}}, \quad M_4 = \max_{x\in[a,b]} |f^{IV}(x)|. \tag{8.22}$$

Блок-схема алгоритму метода Сімпсона представлено на рис. 8.9.

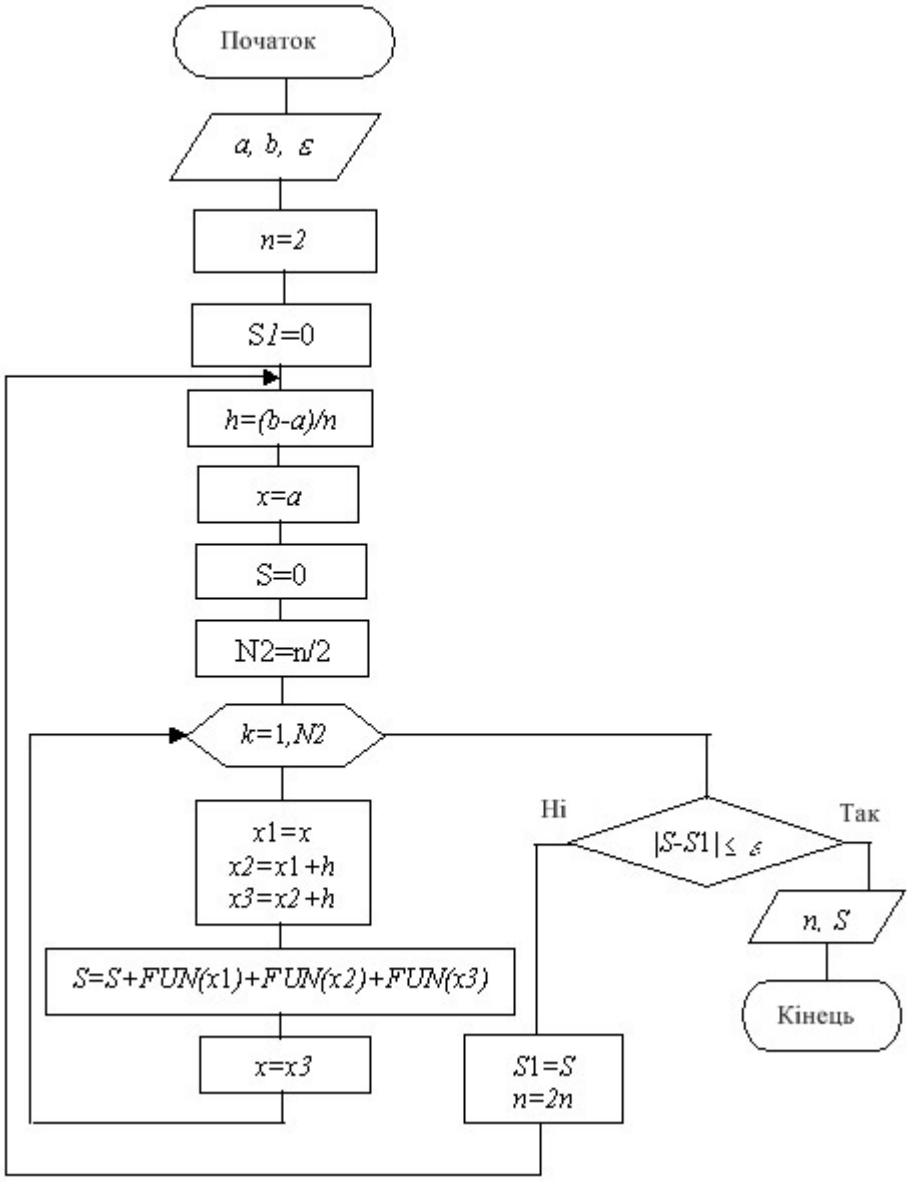


Рис. 8.9. Блок-схема алгоритму методу Сімпсона

Розглянемо ще один приклад.

Приклад 8.2. Методом трапецій з точністю $\varepsilon = 10^{-2}$ і Сімпсона з точністю $\varepsilon_1 = 10^{-4}$ обчислити визначений інтеграл (обчислюваний точно)

$$\int_0^1 \frac{dx}{1+x} = \ln|1+x|\,\Big|_0^1 = \ln 2 = 0{,}69315.$$

Розв'язок.

1) *Метод трапецій.* Виходячи із заданої точності $\varepsilon = 10^{-2}$, обчислимо крок чисельного інтегрування, для чого використовується формула (8.16)

$$h \le \sqrt{\frac{12\varepsilon}{(b-a)M_2}}, \quad M_2 = \max_{x\in[0;1]} |f''(x)| = \max_{x\in[0;1]} \left|\frac{2}{(1+x)^3}\right| = 2; \; h \le \sqrt{\frac{12\cdot 0{,}01}{(1-0)\cdot 2}} = \sqrt{6}\cdot 0{,}1 = 0{,}2449.$$

Необхідно вибрати такий крок, який задовольняє нерівності $h \le 0{,}2449$ і щоб

на відрізку інтеграції $x \in [0;1]$ він укладався ціле число разів. Приймаємо $h = 0{,}2$.

Для підінтегральної функції $f(x) = (1+x)^{-1}$ з незалежною змінною x_i, що змінюється відповідно до рівності $x_i = x_0 + ih = 0 + i \cdot 0{,}2$, $i = \overline{0,5}$, складаємо сіткову функцію з точністю до другого знаку після коми:

i	0	1	2	3	4	5
x_i	0	0,2	0,4	0,6	0,8	1,0
y_i	1,0	0,83	0,71	0,63	0,56	0,5

Далі використовується формула трапецій (8.10) для чисельного інтегрування при *n=5.*

$$\int_0^1 \frac{dx}{1+x} \approx \frac{h}{2}\left(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i \right) = \frac{0{,}2}{2}\left[1{,}0 + 0{,}5 + 2(0{,}83 + 0{,}71 + 0{,}63 + 0{,}56)\right] = 0{,}696.$$

Порівнюючи це значення з точним, бачимо, що абсолютна похибка не перевищує заданої точності ε: $|0{,}69315 - 0{,}696| < 0{,}01$.

Таким чином, за наближене значення визначеного інтеграла по методу трапецій з точністю $\varepsilon = 0{,}01$ приймається значення

$$\int_0^1 \frac{dx}{1+x} \approx 0{,}696.$$

2) *Мет од Сімпсона*. Виходячи із заданої точності обчислюється крок чисельного інтегрування для методу Сімпсона за формулою (8.22):

$$h \le \sqrt[4]{\frac{180\varepsilon}{(b-a)M_4}},\ M_4 = \max_{x\in[0;1]} \left| f^{IV}(x) \right| = \max_{x\in[0;1]} \left| \frac{24}{(1+x)^5} \right| = 24;$$

$$h \le \sqrt[4]{\frac{180 \cdot 10^{-4}}{(1-0)\cdot 24}} = 10^{-1}\sqrt[4]{7{,}5} = 0{,}165.$$

Необхідно вибрати такий крок, щоб він задовольняв нерівності $h \le 0{,}165$ і щоб на відрізку інтегрування $[0;1]$ він укладався *парне число* разів. Приймаємо $h = 0{,}1$. З цим кроком для підінтегральної функції $f(x) = (1+x)^{-1}$ формується сіткова функція з незалежною змінною x_i, що змінюється згідно

із законом $x_i = x_0 + ih = 0 + i \cdot 0{,}1,\ i = \overline{0,10}\ , n = 10,\ m = 5$, причому значення сіткової функції обчислюються з точністю до четвертого знаку після коми:

i	**0**	**1**	**2**	**3**	**4**
x_i	0	0,1	0,2	0,3	0,4
y_i	1,0	0,9091	0,8333	0,7692	0,7143
5	6	7	8	9	10
0,5	0,6	0,7	0,8	0,9	1,0
0,6667	0,625	0,5882	0,5556	0,5263	0,5

Використовується формула Сімпсона (8.20) для чисельного інтегрування (*n=10, m=5*).

$$\int_0^1 \frac{dx}{1+x} = \frac{h}{3}\left(y_0 + y_n + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i} \right) = \frac{0{,}1}{3} \cdot [1{,}0 + 0{,}5 + 4(y_1 + y_3 + y_5 + y_7 + y_9) +$$

$$+ 2(y_2 + y_4 + y_6 + y_8)] = \frac{0{,}1}{3}[1{,}5 + 4(0{,}9091 + 0{,}7692 + 0{,}6667 + 0{,}5882 + 0{,}5263) +$$

$$+ 2(0{,}8323 + 0{,}7143 + 0{,}625 + 0{,}5556)] = \frac{0{,}1}{3}(1{,}5 + 4 \cdot 3{,}4595 + 2 \cdot 2{,}7281) = \frac{0{,}1}{3} 20{,}7942 =$$

$$= 0{,}69314.$$

Порівняння цього значення з точним значенням інтеграла показує, що абсолютна похибка не перевищує заданої точності ε_1:

$|0{,}69315 - 0{,}69314| < 0{,}0001$.

Таким чином, за наближене значення визначеного інтеграла по методу Сімпсона з точністю $\varepsilon_1 = 0{,}0001$ приймається значення:

$$\int_0^1 \frac{dx}{1+x} \approx 0{,}6931.$$

8.4. Метод Монте-Карло

8.5. Використання сплайнів для чисельного інтегрування функцій

8.6. Контрольні запитання

Розділ 9. Розв’язування звичайних диференціальних рівнянь

Диференціальними називаються рівняння, що містять одну або декілька похідних. Інженерові дуже часто доводиться стикатися з ними при розробці нових виробів або технологічних процесів, оскільки велика частина законів фізики формулюється саме у вигляді диференціальних рівнянь. По суті будь-яке завдання проектування, пов'язане з розрахунком потоків енергії або руху тіл, кінець кінцем зводиться до розв’язування диференціальних рівнянь. На жаль, лише дуже небагато з них вдається вирішити без допомоги обчислювальних машин. Тому чисельні методи рішення диференціальних рівнянь грають таку важливу роль в практиці інженерних розрахунків.

Залежно від кількості незалежних змінних і типу похідних, що в них входять, диференціальні рівняння діляться на дві істотно різні категорії: *звичайні*, такі, що містять одну незалежну змінну і похідні по ній, і *рівняння в част инних похідних*, що мають декілька незалежних змінних і похідних по ним. У цьому розділі розглядаються методи рішення звичайних диференціальних рівнянь.

9.1. Задача Коші та крайова задача

Щоб розв’язати звичайне диференціальне рівняння, необхідно знати значення залежної змінної і (або) її похідних при деяких значеннях незалежної змінної. Якщо ці додаткові умови задаються при одному значенні незалежної змінної, то така задача називається задачею з початковими умовами, або *задачею Коші*. Якщо ж умови задаються при двох або більш значеннях незалежної змінної, то задача називається *крайовою*. В задачі Коші додаткові умови називають початковими, а в крайовій задачі – граничними. Часто в задачі Коші в ролі незалежної змінної виступає час. Прикладом може служити задача про вільні коливання тіла, підвішеного на пружині. Рух такого тіла описується диференціальним рівнянням, в якому незалежною змінною є час t. Якщо додаткові умови задані у вигляді значень переміщення і швидкості при $t=0$, то маємо завдання Коші. Для тієї ж механічної системи можна сформулювати і крайову задачу. В цьому випадку одна з умов

повинна полягати в завданні переміщення після закінчення деякого проміжку часу. У крайових задачах як незалежна змінна часто виступає довжина. Відомим прикладом такого роду є диференціальне рівняння, що описує деформацію пружного стрижня. В цьому випадку граничні умови зазвичай задаються на обох кінцях стрижня. Хоча обидва вказані завдання розглядаються в одному розділі, при їх рішенні застосовуються істотно різні методи і обчислювальні алгоритми. Виклад почнемо із задачі Коші.

9.1.1. Задача Коші

Задачу Коші можна сформулювати таким чином. Нехай задано диференціальне рівняння з початковою умовою:

$$\begin{aligned} y' &= f(x, y) \\ y(x_0) &= y_0 \end{aligned} \qquad (9.1)$$

Треба знайти функцію ***y(x)***, що задовольняє як вказане рівняння, так і початкову умову. Звичайно чисельний розв'язок цієї задачі отримують, обчислюючи спочатку значення похідної, а потім задаючи малий приріст ***x*** і переходячи до нової точки $x_1 = x_0 + h$. Положення нової точки визначається за нахилом кривої, обчисленому за допомогою диференціального рівняння. Таким чином, графік чисельного рішення є послідовність коротких прямолінійних відрізків, якими апроксимується дійсна крива ***y=f(x)***. Сам чисельний метод визначає порядок дій при переході від даної точки кривої до наступної.

Оскільки чисельне рішення задачі Коші широко застосовується в різних областях науки і техніки, то воно протягом багатьох років було об'єктом пильної уваги і кількість розроблених для нього методів дуже велике. Зупинимося тут на наступних двох групах методів рішення задачі Коші.

1. *Однокрокові методи*, в яких для знаходження наступної точки на кривій ***y=f(x)*** потрібна інформація лише про один попередній крок. Однокроковими є методи Ейлера і Рунге- Кутта.

2. *Методи прогнозу і корекції* (багатокрокові), в яких для відшукання наступної точки кривої ***y=f(x)*** потрібна інформація про більш ніж одну з

попередніх точок. Щоб отримати достатньо точне чисельне значення, часто вдаються до ітерації. До таких методів належать, зокрема, методи Адамса.

Для знаходження чисельного розв'язку на відрізку $[a,b]$, де $x_0 = a$ введемо на відрізку різницеву сітку $\Omega^{(k)} = \{x_k = x_0 + hk\}$, $k = 0,1,....,N$, $h = |b-a|/N$. .

Точки x_k називаються *вузлами* різницевої сітки, відстані між вузлами – *кроком* різницевої сітки (h), а сукупність значень, заданих у вузлах сітки, визначає *сіткову функцію* $y^{(h)} = \{y_k, k = 0,1,...., N\}$.

Наближений розв'язок задачі Коші (9.1) шукатимемо чисельно у вигляді сіткової функції $y^{(h)}$. Для оцінки похибки наближеного чисельного розв'язку розглядатимемо цей розв'язок як елемент ***N+1***-вимірного лінійного векторного простору з певною нормою, а похибку як норму відхилення $\delta^{(h)} = y^{(h)} - [y]^{(h)}$, де $[y]^{(h)}$ – точний розв'язок задачі (9.1) у вузлах розрахункової сітки. Таким чином $\varepsilon_h = \left\| y^{(h)} - [y]^{(h)} \right\|$.

9.2. Однокрокові методи

Всім однокроковим методам властиві певні загальні риси:

1. Щоб отримати інформацію в новій точці, треба мати дані лише в одній попередній точці.

2. У основі всіх однокрокових методів лежить розкладання функції в ряд Тейлора, в якому зберігаються члени, що містять ***h*** в степені до ***k*** включно. Ціле число ***k*** називається *порядком* методу. Похибка на кроці має порядок ***k*+1**.

3. Всі однокрокові методи не вимагають дійсного обчислення похідних, обчислюється лише сама функція, проте може бути потрібне її значення в декількох проміжних точках. Це спричиняє, звичайно, додаткові витрати часу і зусиль.

4. Властивість п.1 в залежності від інформації лише попереднього кроку дозволяє легко міняти величину кроку ***h***, що робиться автоматично в програмах обчислювальних методів.

9.2.1. Метод Ейлера

Це простий метод розв'язування задачі Коші, що дозволяє інтегрувати диференціальні рівняння першого порядку. Його точність невелика, і тому на практиці їм користуються порівняно рідко. Проте на основі цього методу легше зрозуміти алгоритм інших, ефективніших методів.

Метод Ейлера заснований на розкладанні у в ряд Тейлора в околі точки x_0:

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2}h^2 y''(x_0) + \ldots \qquad (9.2)$$

Якщо h мале, то члени, що містять h в другій або вищих ступенях, є малими вищих порядків і ними можна нехтувати. Тоді $y(x_0+h)=y(x_0)+hy'(x_0)$. Величину $y'(x_0)$ знаходимо з диференціального рівняння, підставивши в нього початкову умову. Враховуючи, що $\Delta y = y_1 - y_0$ і замінюючи похідну на праву частину диференціального рівняння, отримуємо співвідношення $y_1 = y_0 + hf(x_0, y_0)$. Вважаючи тепер точку (x_1, y_1) початковою і повторюючи всі попередні міркування, можна знайти наступне наближене значення залежної змінної в точці (x_2, y_2). Цей процес можна продовжити, використовуючи співвідношення (що і є розрахунковою формулою методу Ейлера):

$$y_{n+1} = y_n + hf(x_n, y_n) \qquad (9.3)$$

і роблячи скільки завгодно багато кроків. Графічно метод Ейлера показаний на рис. 9.1.

9.2.2. Похибка методу Ейлера

В ітеративних чисельних методах розрізняють два види похибок:

1. *локальная помилка* – це сума похибок, що вносяться до обчислювального процесу на кожному кроці обчислень;

2. *глобальна помилка* – різниця між обчисленим і точним значенням величини на кожному етапі реалізації чисельного алгоритму, що визначає сумарну похибку, що накопичилася з моменту початку обчислень.

Локальна похибка методу Ейлера визначається $\varepsilon_k^h = \frac{y''(\xi)}{2}h^2$, де $\xi \in [x_{k-1}, x_k]$ і має порядок h^2, оскільки члени, що містять h в другій і вищих ступенях, відкидаються. Для одержання глобальної похибки треба локальну похибку підсумувати на всьому відрізку $[a,b]$ n разів (b-a=nh), отримаємо: $\varepsilon_k^h = \frac{y''(\xi)}{2}nh^2 = \frac{y''(\xi)}{2}(b-a)h$, і отже метод Ейлера має перший порядок точності відносно величини кроку h.

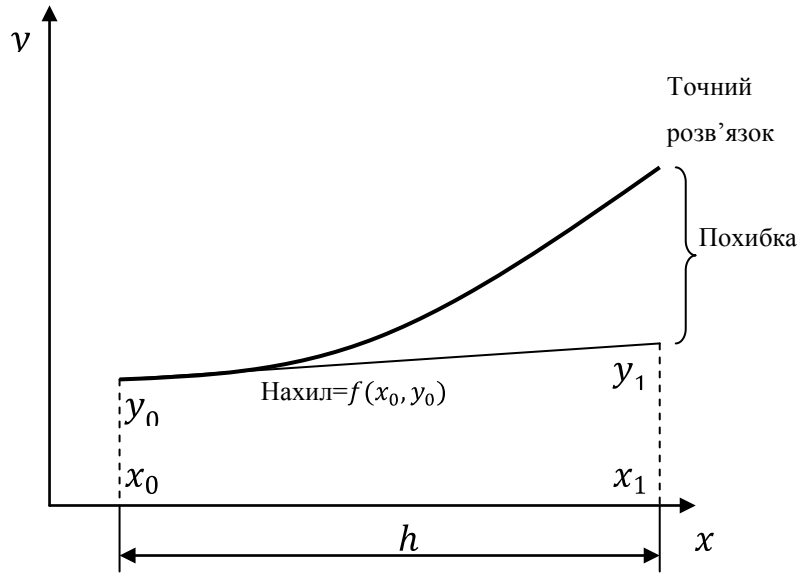


Рис. 9.1. Метод Ейлера

Блок-схема алгоритму диференціювання за методом Ейлера представлено на рис. 9.2.

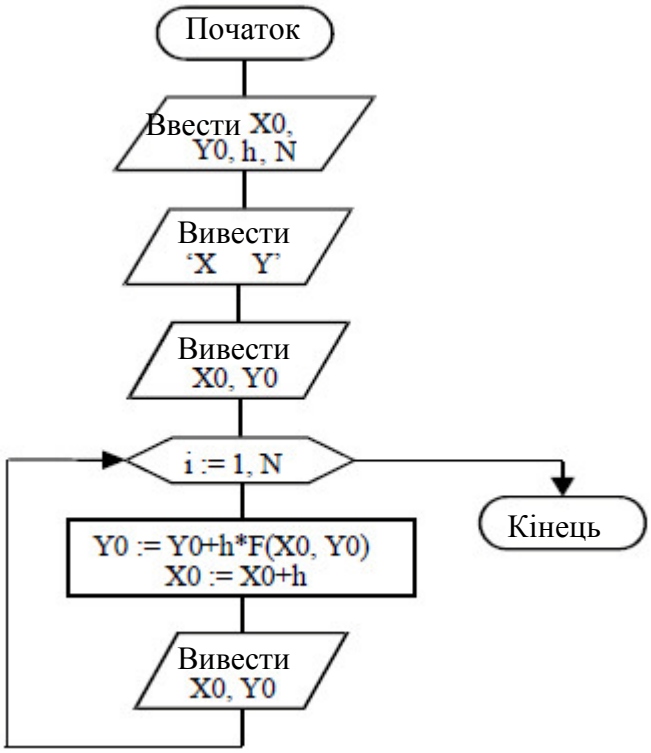


Рис. 9.2. Блок-схема алгоритму диференціювання
за методом Ейлера

На рис. 9.3 представлена реалізація алгоритму Ейлера на мові *Python*.

```
import numpy as np
from math import tan

def F(X,Y):
    return (Y+X)**2

def Euler(x,x1,y,n=100):
    h=(x1-x)/n

    for i in xrange(1,n+1):
        F1=F(x,y)
        y+=F1*h
        x+=h
```

```
        print "x%d=%.8f-->y%d=%.8f-->y_exact=%.8f" %(i,x,i,y,(tan(x)-x))


x=0.

x1=0.5

n=10

y=0


print "Euler method"

Euler(x,x1,y,n)
```

Рис. 9.3. Реалізація алгоритму диференціювання
за методом Ейлера

Дана реалізація вирішує завдання з прикладу 9.1. Головним елементом програми є функція Euler, яка має 4 параметри: x,x1 – інтервал пошуку, y – початкова умова, n – кількість точок розрахунку. Результат роботи програми представлено на рис. 9.4.

```
>>>
Euler method
x1=0.05000000-->y1=0.00000000-->y_exact=0.00004171
x2=0.10000000-->y2=0.00012500-->y_exact=0.00033467
x3=0.15000000-->y3=0.00062625-->y_exact=0.00113522
x4=0.20000000-->y4=0.00176066-->y_exact=0.00271004
x5=0.25000000-->y5=0.00379603-->y_exact=0.00534192
x6=0.30000000-->y6=0.00701665-->y_exact=0.00933625
x7=0.35000000-->y7=0.01172962-->y_exact=0.01502849
x8=0.40000000-->y8=0.01827203-->y_exact=0.02279322
x9=0.45000000-->y9=0.02701961-->y_exact=0.03305507
x10=0.50000000-->y10=0.03839699-->y_exact=0.04630249
```

Рис. 9.4. Результат роботи програми

Як видно з результатів, метод Ейлера дуже не точний, тому бажано його не використовувати.

9.2.3. Модифікований метод Ейлера-Коші

Хоча тангенс кута нахилу дотичної до дійсної кривої в початковій точці відомий і рівний $\mathbf{y'(x_0)}$, він змінюється відповідно до зміни незалежної змінної. Тому в точці $\boldsymbol{x_o+h}$ нахил дотичною вже не такий, яким він був в точці $\boldsymbol{x_0}$. Отже, при збереженні початкового нахилу дотичною на всьому інтервалі $\boldsymbol{h}$ до результатів обчислень вноситься певна похибка. Точність методу Ейлера можна істотно підвищити, поліпшивши апроксимацію похідної. Це можна зробити, наприклад, використовуючи середнє значення похідної на початку і кінці інтервалу. У модифікованому методі Ейлера спочатку обчислюється значення функції в наступній точці по методу Ейлера $y^*_{n+1} = y_n + hf(x_n, y_n)$, яке використовується потім для обчислення наближеного значення похідною в кінці інтервалу $\boldsymbol{f(x_{n+1}, y^*_{n+1})}$. Обчисливши середнє між цим значенням похідної і її значенням на початку інтервалу, знайдемо точніше значення $\boldsymbol{y_{n+1}}$ (9.4)**:**

$$y_{n+1} = y_n + \frac{1}{2}h[f(x_n, y_n) + f(x_{n+1}, y^*_{n+1})] \qquad (9.4)$$

Цей прийом ілюструється нижче на рис. 9.5. Принцип, на якому заснований модифікований метод Ейлера, можна пояснити і інакше.

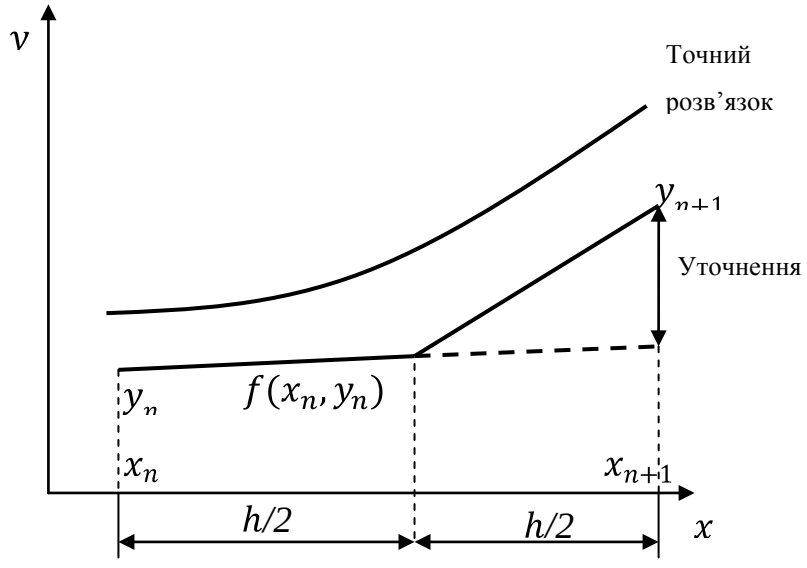


Рис. 9.5. Модифікований метод Ейлера

Для цього повернемося до розкладання функції в ряд Тейлора:

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2}h^2 y''(x_0) + \ldots \tag{9.5}$$

Здається очевидним, що, зберігши член з $\boldsymbol{h^2}$ і відкинувши члени вищих порядків, можна підвищити точність. Проте, щоб зберегти член з $\boldsymbol{h^2}$, треба знати другу похідну $\boldsymbol{y''(x_0)}$. Її можна апроксимувати кінцевою різницею (9.6):

$$y''(x_0) = \frac{\Delta y'}{\Delta x} = \frac{y'(x_0 + h) - y'(x_0)}{h} \tag{9.6}$$

Підставивши цей вираз в формулу Тейлора (9.5) з відкинутими членами, вищими за другий порядок, знайдемо:

$$y(x_0 + h) = y(x_0) + \frac{1}{2}h[y'(x_0 + h) + y'(x_0)] \tag{9.7}$$

що практично співпадає з раніше отриманим виразом. Позначивши $x_1 = x_0 + h$, маємо $y(x_1) = y(x_0) + \frac{1}{2}h[y'(x_1) + y'(x_0)]$ і, в загальному випадку,

$$y(x_{n+1}) = y(x_n) + \frac{1}{2}h[y'(x_{n+1}) + y'(x_n)]. \tag{9.8}$$

Оскільки наступне значення (функції і похідної) входить по обидва боки рівності, то таку формулу називають *неявною*. Якщо функція $\boldsymbol{f(x,y)}$ лінійна по $\boldsymbol{y}$, то тоді неявне рівняння можна розв'язати відносно $\boldsymbol{y_{n+1}}$. Але, в більшості випадків таке розв'язання неможливе і тоді використовують інші схеми, зокрема *ітераційні*, для знаходження похідних в більш, ніж одній точці.

Цей метод є методом другого порядку точності, оскільки в ньому використовується член ряду Тейлора, що містить $\boldsymbol{h^2}$. Помилка на кожному кроці при використанні цього методу, має порядок $\boldsymbol{h^3}$. За підвищення точності доводиться розплачуватися додатковими витратами машинного часу, необхідними для обчислення $\boldsymbol{y^*_{n+1}}$. Вища точність може бути досягнута, якщо користувач готовий згаяти додатковий машинний час на кращу апроксимацію похідної шляхом збереження більшого числа членів ряду Тейлора. Ця ж ідея лежить в основі методів Рунге-Кутта.

Блок-схема модифікованого алгоритму Ейлера представлена на рис. 9.6.

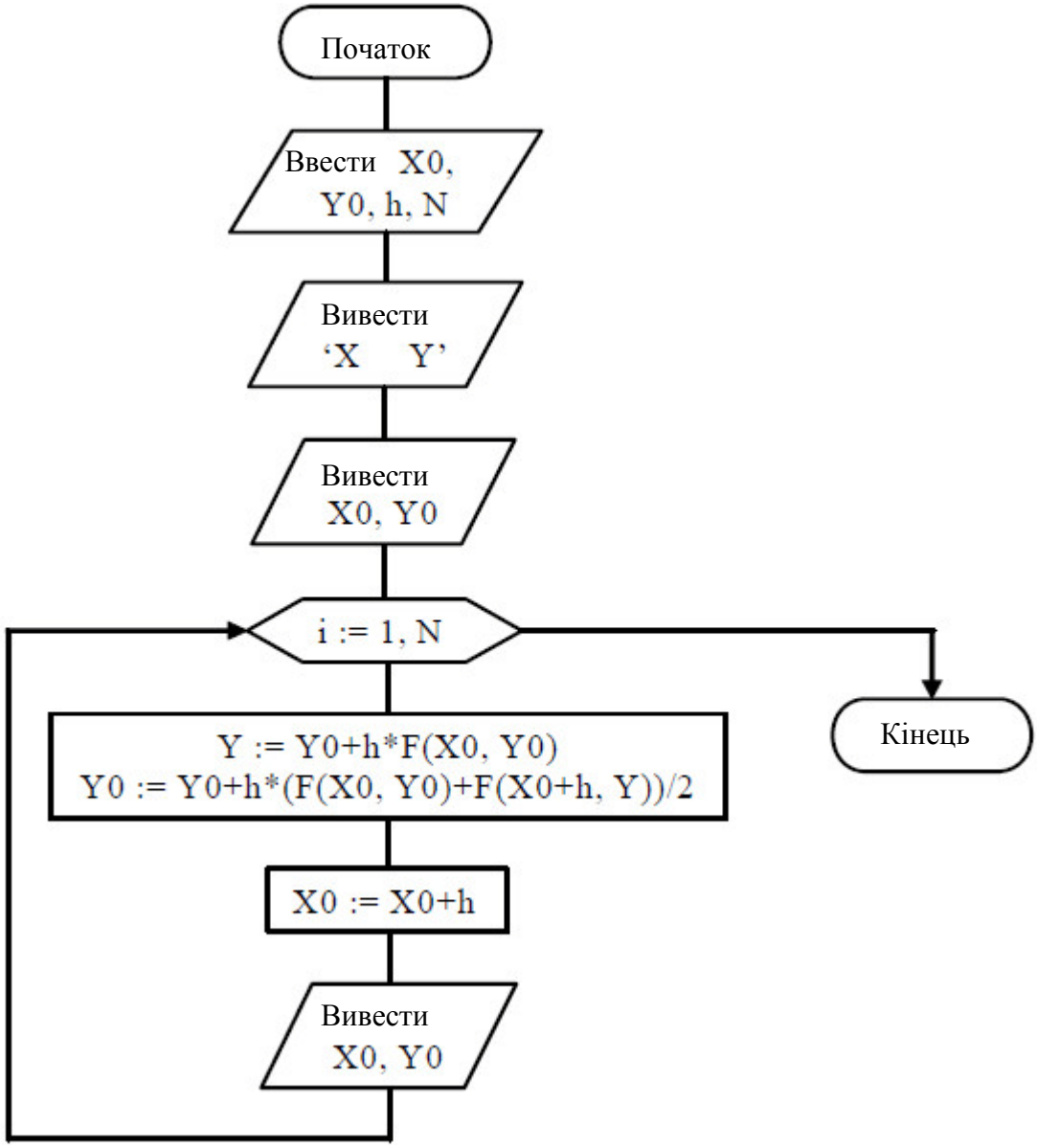


Рис. 9.6. Блок-схема алгоритму диференціювання
за модифікованим методом Ейлера

Реалізація запропонованого алгоритму представлена на рис. 9.7.

```
import numpy as np

from math import tan


def F(X,Y):
    return (Y+X)**2


def Euler_mod(x,x1,y,n=100):
    h=(x1-x)/n
```

```
    y1=y


    for i in xrange(1,n+1):
        F1=F(x,y)
        x+=h
        y+=y*F1*h
        y=y1+h*(F1+F(x,y))/2
        print "x%d=%.8f-->y%d=%.8f-->y_exact=%.8f" %(i,x,i,y,(tan(x)-x))
        y1=y


x=0.
x1=0.5
n=10
y=0
Euler(x,x1,y,n)
```

Рис. 9.7. Реалізація алгоритму диференціювання
за модифікованим методом Ейлера

Дана реалізація вирішує завдання з прикладу 9.1. Головним елементом програми є функція Euler_mod, яка має 4 параметри: x,x1 – інтервал пошуку, y – початкова умова, n – кількість точок розрахунку. Результат роботи програми представлено на рис. 9.8.

```
>>>
x1=0.05000000-->y1=0.00006250-->y_exact=0.00004171
x2=0.10000000-->y2=0.00037547-->y_exact=0.00033467
x3=0.15000000-->y3=0.00119267-->y_exact=0.00113522
x4=0.20000000-->y4=0.00277613-->y_exact=0.00271004
x5=0.25000000-->y5=0.00540155-->y_exact=0.00534192
x6=0.30000000-->y6=0.00936432-->y_exact=0.00933625
```

x7=0.35000000-->y7=0.01498635-->y_exact=0.01502849

x8=0.40000000-->y8=0.02262414-->y_exact=0.02279322

x9=0.45000000-->y9=0.03267853-->y_exact=0.03305507

x10=0.50000000-->y10=0.04560680-->y_exact=0.04630249

Рис. 9.8. Результат роботи програми

Як видно з результатів, модифікований метод Ейлера досить точний та має дуже просту реалізацію.

Розглянемо декілька прикладів.

Приклад 9.1. Явним методом Ейлера з кроком ***h=0.1*** отримати чисельний розв'язок диференціального рівняння $y' = (y + x)^2$ з початковими умовами $y(0) = 0$ на інтервалі ***[0, 0.5]*** . Чисельне рішення порівняти з точним рішенням $y = tg(x) - x$.

Розв'язок. Отже, виходячи з початкової точки, $x_0 = 0$, $y_0 = 0$ розрахуємо значення y_1 у вузлі x_1=0.1 за формулою $y_1 = y_0 + hf(x_0, y_0) = 0 + 0.1(0 + 0)^2 = 0$. Аналогічно отримаємо розв'язок у наступному вузлі x_2=0.2; $y_2 = y_1 + hf(x_1, y_1) = 0 + 0.1(0 + 0.1)^2 = 0.001$. Продовжимо обчислення і, ввівши позначення $\Delta y_k = hf(x_0, y_0)$ і, $\varepsilon_k = |y_{точн}(x_k) - y_k|$, де $y_{точн}(x_k)$ – точний розв'язок у вузлових точках, отримувані результати занесемо в табл. 9.1.

Табл. 9.1. Отримані результати

k	**x**	**y**	Δy_k	$y_{точн}$	ε_k
0	0.000000000	0.000000000	0.000000000	0.000000000	0.0000
1	0.100000000	0.000000000	0.001000000	0.000334672	0.3347E-03
2	0.200000000	0.001000000	0.004040100	0.002710036	0.1710E-02
3	0.300000000	0.005040100	0.009304946	0.009336250	0.4296E-02
4	0.400000000	0.014345046	0.017168182	0.022793219	0.8448E-02
5	0.500000000	0.031513228		0.046302490	0.1479E-01

Рішенням задачі є таблична функція (табл. 9.2) (залишено *5* значущих цифр в кожному числі).

Табл. 9.2. Таблична функція

k	**0**	**1**	**2**	**3**	**4**	**5**

x_k	0.00000	0.10000	0.200000	0.3000000	0.400000	0.500000
y_k	0.00000	0.00000	0.001000	0.0050401	0.014345	0.031513

Приклад 9.2. Розв'язати попередню задачу з прикладу 9.1 методом Ейлера-Коші.

Виходячи з початкової точки, $x_0 = 0$, $y_0 = 0$ розрахуємо значення y_1 у вузлі x_1=0.1 за формулами:

$$\tilde{y}_{k+1} = y_k + hf(x_k, y_k)$$

$$y_{k+1} = y_k + \frac{h(f(x_k, y_k) + f(x_{k+1}, \tilde{y}_{k+1})}{2}$$

$$x_{k+1} = x_k + h.$$

Маємо $\tilde{y}_1 = y_0 + hf(x_0, y_0) = 0 + 0.1(0+0)^2 = 0$. $f(x_1, \tilde{y}_1) = (0+0.1)^2 = 0.01$

$$y_1 = y_0 + 0.5h(f(x_0, y_0) + f(x_1, \tilde{y}_1)) = 0 + 0.5 * 0.1 * (0 + 0.01) = 0.0005$$

Аналогічно отримаємо розв'язок в решті вузлів. Продовжуючи обчислення і вводячи позначення $\Delta y_k = 0.5h(f(x_k, y_k) + f(x_{k+1}, \tilde{y}_{k+1}))$, отримувані результати занесемо в табл. 9.3.

Табл. 9.3. Отримані результати

k	x_k	y_k	$\tilde{y}_k$	Δy_k	$y_{точн}$	ε_k
0	0.0	0.000000000		0.000500000	0.000000000	0.000000000
1	0.1	0.000500000	0.00000	0.002535327	0.000334672	0.1653E-03
2	0.2	0.003035327	1.510025E-003	0.006778459	0.002710036	0.3253E-03
3	0.3	0.009813786	7.157661E-003	0.013594561	0.009336250	0.4775E-03
4	0.4	0.023408346	1.941224E-002	0.023615954	0.022793219	0.6151E-03
5	0.5	0.047024301	4.133581E-002		0.046302490	0.7218E-03

Рішенням задачі є таблична функція (табл. 9.4) (залишено 5 значущих цифр в кожному числі)

Табл. 9.4. Таблична функція

k	0	1	2	3	4	5
x_k	0.00000	0.100000	0.2000000	0.3000000	0.4000000	0.500000
y_k	0.00000	0.000500	0.0030353	0.0098138	0.0234083	0.047024

9.2.4. Методи Рунге —Кутта

Щоб отримати у ряді Тейлора член ***n***-го порядку, необхідно якимось чином обчислювати ***n***-у похідну залежної змінної. При використанні модифікованого методу Ейлера для отримання другої похідної в кінцево-різницевій формі достатньо було знати нахили кривої на кінцях даного інтервалу. Щоб обчислити третю похідну в кінцево-різницевому вигляді, необхідно мати значення другої похідної щонайменше в двох точках. Для цього необхідно додатково визначити нахил кривої в деякій проміжній точці інтервалу ***h***, тобто між $\boldsymbol{x_n}$ і $\boldsymbol{x_{n+1}}$. Очевидно, чим вище порядок обчислюваної похідної, тим більше додаткових обчислень буде потрібно усередині інтервалу.

Метод Рунге-Кутта дає набір формул для розрахунку координат внутрішніх точок, потрібних для реалізації цієї ідеї. Оскільки існує декілька способів розташування внутрішніх точок і вибору відносної ваги для знайдених похідних, то метод Рунге-Кутта по суті об'єднує ціле сімейство методів вирішення диференціальних рівнянь першого порядку.

Ми не станемо детально виводити повні формули, а розглянемо натомість більш простий метод другого порядку. Цей метод використовує лише два обчислення функції за крок.

Перше з них – $k_1 = hf(x, y)$, потім робиться дробовий крок на основі $\boldsymbol{k_1}$. Введемо два поки невизначених коефіцієнта ***a*** і ***b***. Покладемо $k_2 = hf(x + ah, y + bk_1)$ і будемо обчислювати наступне значення функції у вигляді комбінації $y(x + h) = y(x) + c_1k_1 + c_2k_2$.

Щоб визначити коефіцієнти, розкладемо одержаний вираз через формулу Тейлора відносно точки x_n, розглядаючи k_2 як функцію двох змінних: $k_2 = hf(x, y) + h(bk_1 f_y + ahf_x)$, де f_x і f_y є частинними похідними $f(x,y)$. Підставляючи це значення у вираз для $y(x_n+h)= y_{n+1}$, і позначаючи $y(x_n)=y_n$ маємо: $y(x_n + h) = y(x_n) + h(c_1 + c_2) f(x_n, y_n) + c_2bh^2 f_y + c_2ah^2 f_x$. Порівнюючи тепер одержаний вираз із розкладом точного розв’язку:

$$y(x_n+h)=y(x_n)+hy'(x_n)+\frac{1}{2}h^2y''(x_n) = y(x_n+h)=y(x_n)+hf(x_n,y_n)+\frac{h^2}{2}(f_y f(x_n,y_n)+f_x) \quad (9.9)$$

і порівнюючи коефіцієнти при однакових степенях ***h***, приходимо до системи рівнянь відносно коефіцієнтів:

$$\begin{cases} c_1+c_2=1, \\ c_2 b=\frac{1}{2}, \\ c_2 a=\frac{1}{2} \end{cases}$$

Вибравши один з цих коефіцієнтів, наприклад ***a***, за параметр, приходимо до однопараметричного сімейства методів Рунге –Кутта:

$$k_1 = hf(x,y);$$
$$k_2 = hf(x+ah, y+ak_1);$$
$$y_{n+1} = y_n + (1-\frac{1}{2a})k_1 + \frac{1}{2a}k_2. \quad (9.10)$$

Два очевидних вибори для ***a*** – це *1/2* і *1*. Вони визначають методи, тісно пов'язані з формулами квадратури відповідно прямокутників і трапецій. Дослідження перших членів, відкинутих у вищезгаданих розкладаннях, показує, що ні при якому виборі ***a*** не вдається виключити ці члени для всіх функцій ***f(x,y)*** і таким чином, метод має другий порядок точності при будь-якому ***a.***

Розглянутий метод Рунге-Кутта є узагальненням методів Ейлера другого порядку точності. Для одержання більш точних методів вдаються до більш складних схем, що утримують в розкладі Тейлора похідні старших порядків.

Реалізація методу Рунге-Кутта *2*-го на прикладі рівняння Бесселя на інтервалі ***[0,5;1]*** з параметром ***P=0*** та початкових умовах $y_1(0)=0,9384698$ та $y_2(0)=-0,2422685$ представлена на рис. 9.9.

```
import numpy as np


def RP(X,Y,F):
   F[1]=Y[2]
   F[2]=((P/X)**2-1)*Y[1]-(Y[2]/X)
```

```
    return Y,F


def Runge2(X,h,Y,F,n):
    H2=h/2
    RP(X,Y,F)
    for i in xrange(1,n+1):
        y1[i]=Y[i]+(H2*F[i])
    RP(X+H2,y1,F)
    for i in xrange(1,n+1):
        Y[i]+=h*F[i]
    return Y,F


P=0.
X=0.5
X9=0.99
h=0.1
y1=np.zeros((4))
Y=np.array([0.,0.9384698,-0.2422685,0.])
F=np.array([0.,0.,0.,0.])


while (X<X9) and (h>0.0):
    Runge2(X,h,Y,F,2)
    X+=h
    print "X=%.8f-->Y[1]=%.8f-->Y[2]=%.8f" %(X,Y[1],Y[2])
```

Рис. 9.9. Реалізація метода Рунге-Кутта 2-го порядку

Результат виконання програми представлено на рис. 9.10.

```
>>>
```

X=0.60000000-->Y[1]=0.91197329-->Y[2]=-0.28672866

X=0.70000000-->Y[1]=0.88112996-->Y[2]=-0.32904107

X=0.80000000-->Y[1]=0.84617050-->Y[2]=-0.36889624

X=0.90000000-->Y[1]=0.80735562-->Y[2]=-0.40600425

X=1.00000000-->Y[1]=0.76497400-->Y[2]=-0.44009756

Рис. 9.10. Результат виконання програми

Найбільш поширеним з методів Рунге-Кутта є метод, при якому одержуються всі члени, включаючи $\boldsymbol{h^4}$. Це метод четвертого порядку точності, для якого помилка на кроці має порядок $\boldsymbol{h^5}$. Розрахунки при використанні цього класичного методу проводяться за формулою:

$$y_{n+1} = y_n + \frac{K_1 + 2K_2 + 2K_3 + K_4}{6}$$

де $K_1 = hf(x_n, y_n),$ $K_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{K_1}{2}),$ $K_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2),$

$K_4 = hf(x_n + h, y_n + K_3).$ (9.11)

Реалізація методу Рунге-Кутта *4*-го порядку представлена на рис. 9.11.

```
# -*- coding: cp1251 -*-
import numpy as np
from math import exp,fabs

def RP44(X,Y):
   return 2*(X**2+Y)

def RealFunc(X):
   return 1.5*exp(2*X)-(X*X)-X-0.5

def Runge44(X,h,Y,n):
   print "Метод Рунге-Кутта 4-го порядку"
   for i in xrange(0,n):
```

```
        X[i+1]=X[i]+h
        F1=RP44(X[i],Y[i])
        F2=RP44(X[i]+h/2.,Y[i]+h/2.*F1)
        F3=RP44(X[i]+h/2.,Y[i]+h/2.*F2)
        F4=RP44(X[i+1],Y[i]+h*F3)
        Y[i+1]=Y[i]+(F1+2*F2+2*F3+F4)*h/6.
        er=fabs(RealFunc(X[i+1])-Y[i+1])
        if er>maxi:
           max=er
        print "X[%d]=%.8f-->Y[%d]=%.8f-->RealY=%.8f-->Error=%.8f"
%(i+1,X[i+1],i+1,Y[i+1],RealFunc(X[i+1]),er)

maxi=0
n=10
h=1./n

X=np.array([0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.])
Y=np.array([1.,0.,-0.,0.,0.,0.,0.,0.,0.,0.,0.])
Runge44(X,h,Y,n)
```

Рис. 9.11. Реалізація методу Рунге-Кутта 4-го порядку

Для цієї реалізації (рис. 9.11) взяті дані з прикладу 9.3. Результат роботи програми представлено на рис. 9.12. Основну нагрузку в програмі виконує функція Runge44, на вхід якої подають наступні параметри: X – робоча точка, h – крок, Y – вектор початкових умов, який після виконання буде утримувати розраховані дані, n – кількість точок. Функція RP44 розраховує похідну в необхідній точці, функція RealFunc розраховує дійсне значення функції в точці X.

```
>>>
```

Метод Рунге-Кутта 4-го порядку

X[1]=0.10000000-->Y[1]=1.22210167-->RealY=1.22210414-->Error=0.00000247

X[2]=0.20000000-->Y[2]=1.49773064-->RealY=1.49773705-->Error=0.00000640

X[3]=0.30000000-->Y[3]=1.84316587-->RealY=1.84317820-->Error=0.00001233

X[4]=0.40000000-->Y[4]=2.27829046-->RealY=2.27831139-->Error=0.00002093

X[5]=0.50000000-->Y[5]=2.82738964-->RealY=2.82742274-->Error=0.00003310

X[6]=0.60000000-->Y[6]=3.52012537-->RealY=3.52017538-->Error=0.00005001

X[7]=0.70000000-->Y[7]=4.39272680-->RealY=4.39279995-->Error=0.00007315

X[8]=0.80000000-->Y[8]=5.48944418-->RealY=5.48954864-->Error=0.00010446

X[9]=0.90000000-->Y[9]=6.86432478-->RealY=6.86447120-->Error=0.00014641

X[10]=1.00000000-->Y[10]=8.58338196-->RealY=8.58358415-->Error=0.00020219

Рис. 9.12. Результат роботи програми

Як бачимо, результати повністю відповідають дійсності (табл. 9.5), а метод Рунге-Кутта *4*-го порядку має незначну похибку.

Метод Ейлера і його модифікація по суті справи є методами Рунге-Кутта першого і другого порядку відповідно. В порівнянні з ними метод Рунге-Кутта має важливу перевагу, оскільки забезпечує вищу точність, яка з лишком виправдовує додаткове збільшення обсягу обчислень. Вища точність методу Рунге-Кутта часто дозволяє збільшити крок інтеграції ***h***. Допустима похибка на кроці визначає його максимальну величину. Щоб забезпечити високу ефективність обчислювального процесу, величину ***h*** слід вибирати саме з міркувань максимальної допустимої помилки на кроці. Такий вибір часто здійснюється автоматично і включається як складова частина в алгоритм, побудований за методом Рунге-Кутта.

Відносну точність однокрокових методів продемонструємо на наступному прикладі.

Приклад 9.3. Нехай потрібно розв’язати рівняння:

$$\frac{dy}{dx} = 2x^2 + 2y$$

за початкової умови ***y(0)= 1, 0≤x≤1*** і ***h=0,1***. Це – лінійне рівняння першого порядку, що має наступне точне рішення

$$y = 1{,}5e^{2x} - x^2 - x - 0{,}5,$$

яке допоможе нам порівняти відносну точність, що забезпечується різними методами. Результати розрахунку представлені в приведеній нижче табл. 9.5, з якої добре видно переваги методу Рунге-Кутта в порівнянні із звичайним і модифікованим методами Ейлера.

Табл. 9.5. Порівняння точності методів

x_n	Метод Ейлера	Модифікований метод Ейлера	Метод Рунге-Кутта	Точний розв'язок
0,0	1,0000	1,0000	1,0000	1,0000
0,1	1,2000	1,2210	1,2221	1,2221
0,2	1,4420	1,4923	1,4977	1,4977
0,3	1,7384	1,8284	1,8432	1,8432
0,4	2,1041	2,2466	2,2783	2,2783
0,5	2,5569	2,7680	2,8274	2,8274
0,6	3,1183	3,4176	3,5201	3,5202
0,7	3,8139	4,2257	4,3927	4,3928
0,8	4,6747	5,2288	5,4894	5,4895
0,9	5,7376	6,4704	6,8643	6,8645
1,0	7,0472	8,0032	8,5834	8,5836

9.2.5. Метод Рунге-Кутта-Мерсона

Мерсон запропонував модифікацію метода Рунге-Кутта *4*-го порядку, яка дозволяє оцінювати похибку на кожному кроці та приймати рішення про зміну кроку. Схему Мерсона за допомогою еквівалентних перетворень приведемо до виду, що є зручним для програмування:

$$y(x_0 + h) = y_0 + \frac{k_4 + k_5}{2} + O(h^5), \qquad (9.12)$$

де

$$k_1 = h_3 f(x_0, y_0), h_3 = \frac{h}{3},$$

$$k_2 = h_3 f(x_0 + h_3, y_0 + k_1),$$

$$k_3 = h_3 f\left(x_0 + h_3, y_0 + \frac{k_1 + k_2}{2}\right),$$

$$k_4 = k_1 + 4h_3 f\left(x_0 + \frac{h}{2}, y_0 + 0{,}375(k_1 + k_3)\right),$$

$$k_5 = h_3 f\left(x_0 + h, y_0 + 1{,}5(k_4 - k_3)\right).$$

Схема Мерсона вимагає обчислювати на кожному кроці праву частину ЗДУ в п’яти точках, але за рахунок тільки одного додаткового коефіцієнта k_1 в порівнянні з класичною схемою Рунге-Кутти на кожному кроці можна визначити похибку рішення R за формулою

$$10R = 2k_4 - 3k_3 - k_5. \quad (9.13)$$

Для автоматичного вибору кроку інтегрування рекомендується наступний критерій. Якщо абсолютне значення величини R, обчислене за формулою (9.13), стане більше допустимої заданої похибки ε,

$$|R| < \varepsilon, \quad (9.14)$$

то крок h зменшується в два рази і обчислення за схемою (9.12) повторюються з точки (x_0, y_0). При виконанні умови

$$32|R| < \varepsilon, \quad (9.15)$$

крок h можна подвоїти.

Реалізація методу Рунге-Кутта-Мерсона у вигляді модуля представлена на рис. 9.13.

```
import numpy as np
from math import fabs


def sign(X,Y):
    if Y>=0:
        return fabs(X)
    else:
        return -fabs(X)


def RP(P,X,Y,F):
    F[1]=Y[2]
    F[2]=((P/X)**2-1.)*Y[1]-Y[2]/X
```

```
    return F


def RKM(P,X,h,Y,n,epsilon=1e-3):
    F0=np.zeros((4)); F=np.zeros((4))
    k1=np.zeros((4)); k3=np.zeros((4))
    RP(P,X,Y,F0)
    Z=Y[::]; R=0.
    while True:
        H3=h/3; H4=4*H3
        for i in xrange(1,n+1):
            k1[i]=H3*F0[i]
            Y[i]=Z[i]+k1[i]
        RP(P,X+H3,Y,F)
        for i in xrange(1,n+1):
            Y[i]=Z[i]+(k1[i]+H3*F[i])/2
        RP(P,X+H3,Y,F)
        for i in xrange(1,n+1):
            k3[i]=h*F[i]
            Y[i]=Z[i]+0.375*(k1[i]+k3[i])
        RP(P,X+h/2,Y,F)
        for i in xrange(1,n+1):
            k1[i]+=H4*F[i]
            Y[i]=Z[i]+1.5*(k1[i]-k3[i])
        RP(P,X+h,Y,F)
        for i in xrange(1,n+1):
            a=H3*F[i]
            Y[i]=Z[i]+(k1[i]+a)/2
```

```
        a=2*k1[i]-3.*k3[i]-a
        if Y[i]!=0.:
            a/=Y[i]
        if fabs(a)>R:
            R=fabs(a)
    h/=2
    if R>epsilon:
        break
h*=2; X+=h
if 32*R<epsilon:
    h*=2
return X,Y,h
```

Рис. 9.13. Реалізація методу Рунге-Кутта-Мерсона

Написання програми для застосування запропонованого модуля пропонується як самостійна вправа.

9.3. Багатокрокові методи

У методах, що розглядалися досі, значення y_{n+1} обчислювалось за допомогою функції, яка залежить лише від x_n, y_n і довжини кроку h_n. Мабуть, логічно припустити, що можна було б отримати більшу точність, використовуючи інформацію в попередніх точках, а саме: y_n , y_{n-1} , y_{n-2}, … і $f(x_n,y_n)$, $f(x_{n-1},y_{n-1})$, $f(x_{n-2},y_{n-2})$,… . Багатокрокові методи, засновані на цій ідеї, вельми ефективні. Якщо потрібна висока точність, то вони зазвичай економічніші, ніж однокрокові методи, і часто можна тривіально отримати оцінку похибки. Запрограмовані відповідним чином, багатокрокові методи можуть ефективно видавати значення чисельного рішення в довільних точках, не змінюючи значення h. Порядок методу може вибиратися автоматично і динамічно змінюватися, тим самим одержуються методи, що працюють для дуже широкого кола задач.

Лінійні багатокрокові методи можна розглядати як спеціальні випадки

формули: $y_{n+1} = \sum_{i=1}^{k} a_i y_{n-i+1} + h\sum_{i=0}^{k} b_i f(x_{n-i+1}, y_{n-i+1})$, де $\boldsymbol{a_k}$ чи $\boldsymbol{b_k}$ – відмінні від нуля.

Метод називається *лінійним*, тому що кожне *f*() входить у формулу лінійно; при цьому сама *f* може бути, а може і не бути лінійною функцією своїх аргументів.

Після того, як метод «стартував», кожен крок вимагає обчислення $\boldsymbol{y_{n+1}}$ з відомих значень: $\boldsymbol{y_n}$, $\boldsymbol{y_{n-1}}$, $\boldsymbol{y_{n-2}}$, … і $\boldsymbol{f(x_n,y_n)}$, $\boldsymbol{f(x_{n-1},y_{n-1})}$, $\boldsymbol{f(x_{n-2},y_{n-2})}$,…. Якщо $\boldsymbol{b_0=0}$, то метод називається *явним* і обчислення проводиться очевидним чином. Якщо ж $\boldsymbol{b_0\neq 0}$, то метод називається *неявним*, тому що для знаходження $\boldsymbol{y_{n+1}}$ потрібне значення $\boldsymbol{f(x_{n+1},y_{n+1})}$. Труднощі при використанні неявних методів компенсуються їх іншими якостями.

Багатокрокові методи різного порядку точності можна конструювати за допомогою способу квадратури (тобто з використанням еквівалентного інтегрального рівняння).

Розв’язок диференціального рівняння задовольняє інтегральному співвідношенню:

$$y_{k+1} = y_k + \int_{x_k}^{x_{k+1}} f(x, y(x))dx \qquad (9.16)$$

Якщо розв’язок задачі Коші отримано у вузлах аж до $\boldsymbol{k}$-*го*, то можна апроксимувати підінтегральну функцію, наприклад: інтерполяційним многочленом якого-небудь ступеня. Обчисливши інтеграл від побудованого многочлена на відрізку $[x_k, x_{k+1}]$ отримаємо ту чи іншу *формулу Адамса*. Зокрема, якщо використовувати многочлен нульового ступеня (тобто замінити підінтегральну функцію її значенням на лівому кінці відрізка в точці x_k), то отримаємо явний метод Ейлера. Якщо виконати те ж саме, але підінтегральну функцію апроксимувати значенням на правому кінці в точці $\boldsymbol{x_{k+1}}$, то отримаємо неявний метод Ейлера.

При використанні інтерполяційного многочлена *3*-ого степеня побудованого за значеннями підінтегральної функції в останніх чотирьох вузлах отримаємо *метод Адамса четвертого порядку* точності:

$$y_{k+1} = y_k + \frac{h}{24}(55 f_k - 59 f_{k-1} + 37 f_{k-2} - 9 f_{k-3}) \qquad (9.17)$$

де f_k є значенням підінтегральної функції у вузлі x_k.

9.3.1. Різницевий вигляд методу Адамса

Цей спосіб одержання формули Адамса ґрунтується на наближенні похідної за допомогою інтерполяційної формули Ньютона, що використовує кінцеві різниці. Нехай потрібно проінтегрувати рівняння $y'=f(x,y)$, $y(x_0)=y_0$. Різницевий вигляд методу Адамса для наближеного рішення цієї задачі полягає в наступному. Задавшись деяким кроком зміни аргументу h, знаходять яким-небудь чином, виходячи з початкових даних $y(x_0)=y_0$ наступні три значення шуканої функції $y(x)$:

$$y_1 = y(x_1) = y(x_0 + h), y_2 = y(x_0 + 2h), y_3 = y(x_0 + 3h), \qquad (9.18)$$

(ці три значення можна одержати будь-яким методом, що забезпечує потрібну точність, наприклад, за допомогою розкладання розв'язку в степеневий ряд, методом Рунге-Кутта і т. д., але не методом Ейлера зважаючи на його недостатню точність).

За допомогою чисел x_0, x_1, x_2, x_3 і y_0, y_1, y_2, y_3 обчислюють величини:

$$q_0 = hy'_0 = hf(x_0, y_0), q_1 = hf(x_1, y_1), q_2 = hf(x_2, y_2), q_3 = hf(x_3, y_3), \qquad (9.19)$$

Далі, складають таблицю кінцевих різниць величин y і q (табл. 9.6):

Табл. 9.6. Таблиця кінцевих різниць

x	y	Δy	q	Δq	$\Delta^2 q$	$\Delta^3 q$
x_0	y_0		q_0			
		Δy_0		Δq_0		
x_1	y_1		q_1		$\Delta^2 q_0$	
		Δy_1		Δq_1		$\Delta^3 q_0$
x_2	y_2		q_2		$\Delta^2 q_1$	
		Δy_2		Δq_2		
x_3	y_3		q_3			
…	…	…	…	…	…	…

Знаючи значення в нижньому косому рядку, значення y_{i+1} знаходиться за формулою:

$$y_{i+1} = y_i + q_i + \frac{1}{2}\Delta q_i + \frac{5}{12}\Delta^2 q_{i-2} + \frac{3}{8}\Delta^3 q_{i-3}$$

Приклад 9.4. Використовуючи метод Адамса, знайти значення $y(0,4)$ з точністю до *0,01* для диференціального рівняння $y'=x^2+y^2$. Початкова умова – $y(0)=-1$.

Розв'язок. Знайдемо перші чотири члени розкладу розв'язку даного рівняння в ряд Тейлора в околі точки $x=0$:

$$y(x) = y(0) + y'(0)x + \frac{1}{2}y''(0)x^2 + \frac{1}{6}y'''(0)x^3 + \ldots$$

Згідно умові, $y(0)=-1$; значення похідних в точці *0* знаходимо, послідовно диференціюючи дане рівняння:

$$y' = x^2 + y^2; y'(0) = 0^2 + (-1)^2 = 1$$

$$y'' = 2x + 2yy; y''(0) = 0 + 2(-1) = -2;$$

$$y''' = 2 + 2y'^2 + 2yy''; y'''(0) = 2 + 2(-1)^2 + 2(-1)(-2) = 8;$$

Таким чином,

$$y(x) = -1 + x - x^2 + \frac{4}{3}x^3 + \ldots$$

Обчислюємо $y(x)$ в точках $x_1= 0,1$, $x_2 = 0,2$, $x_3 = 0,3$ з одним запасним (третім) знаком $y_1 =-0,909$, $y_2 = -0,829$, $y_3 = -0,754$. Складемо таблицю (табл. 9.7).

Табл. 9.7. Обчислені дані

x	y	Δy	q	Δq	$\Delta^2 q$	$\Delta^3 q$
0	*-1*		*0,1*			
		0,091		*-0,017*		
0,1	*-0,909*		*0,083*		*0,06*	
		0,080		*-0,011*		*-0,002*
0,2	*-0,829*		*0,072*		*0,04*	
		0,075		*-0,007*		
0,3	*-0,754*		*0,065*			

0,4						

Тоді: $$\Delta y_3 = q_3 + \frac{1}{2}\Delta q_2 + \frac{5}{12}\Delta^2 q_1 + \frac{3}{8}\Delta^3 q_0 =$$
$$= 0{,}065 + \frac{1}{2}(-0{,}007) + \frac{5}{12}0{,}004 + \frac{3}{8}(-0{,}002) = 0{,}062$$

Звідси: $y_4 = y_3 + \Delta y_3 = -0{,}754 + 0{,}068 = -0{,}69.$

9.3.2. Метод Адамса-Бешфортса

Метод Адамса, як і всі багатокрокові методи не є «само починаючим». Тобто, для того, що б використовувати метод Адамса, необхідно мати розв'язок у перших чотирьох вузлах. У вузлі $\boldsymbol{x_0}$ розв'язок відомий з початкових умов, а в інших трьох вузлах розв'язок можна отримати за допомогою відповідного однокрокового методу, наприклад: методу Рунге-Кутта четвертого порядку.

Зазвичай на кожному кроці розв'язування разом використовуються два багатокрокові методи. Явний метод, званий *предиктором*, супроводжується одним або більш застосуваннями неявного методу, званого *коректором*, – звідси назва «*методи предиктор-коректор*».

Методи типу предиктор-коректор дозволяють підвищити точність обчислень методу Адамса за рахунок подвійного обчислення значення функції $f(x, y)$ при визначенні y_{k+1} на кожному новому кроці.

На етапі предиктора згідно з методом Адамса по значеннях у вузлах $x_{k-2}, x_{k-1}, x_k, x_{k+1}$ розраховується "попереднє" значення розв'язку у вузлі x_{k+1}.

$$\hat{y}_{k+1} = y_k + \frac{h}{24}(55 f_k - 59 f_{k-1} + 37 f_{k-2} - 9 f_{k-3}) \qquad (9.20)$$

За допомогою цього значення розраховується "попереднє" значення функції $f_{k+1} = f(x_{k+1}, \hat{y}_{k+1})$ в новій точці.

На етапі коректора за методу Адамса 4-го порядку по значеннях у вузлах $x_{k-2}, x_{k-1}, x_k, x_{k+1}$ розраховується "остаточне" значення рішення у вузлі x_{k+1}.

$$y_{k+1} = y_k + \frac{h}{24}(9 f_{k+1} + 19 f_k - 5 f_{k-1} + f_{k-2}) \qquad (9.21)$$

Схема предиктор-коректор для обчислення y_{n+1} така:

1. Використати предиктор для обчислення $y_{n+1}^{(0)}$ – початкового наближення до y_{n+1}. Покласти $i=0$.

2. Обчислити функцію, поклавши $f_{n+1}^{(i)} = f(x_{n+1}, y_{n+1}^{(i)})$.

3. Обчислити краще наближення $y_{n+1}^{(i)}$ за формулою коректора, вважаючи $f_{n+1} = f_{n+1}^{(i)}$.

4. Якщо $\left|y_{n+1}^{(i+1)} - y_{n+1}^{(i)}\right| >$ заданого допуску, то збільшити i на *1* і перейти до кроку *2*; інакше покласти $y_{n+1} = y_{n+1}^{(i+1)}$.

Реалізація алгоритму метода Адамса-Бешфортса представлена на рис. 9.14.

```
# -*- coding: cp1251 -*-
import numpy as np
from math import exp,fabs


def RP44(X,Y):
   return 2*(X**2+Y)


def RealFunc(X):
   return 1.5*exp(2*X)-(X*X)-X-0.5


def Runge44(X,h,Y,n):
   print "Метод Рунге-Кутта 4-го порядку"
   for i in xrange(0,n):
      X[i+1]=X[i]+h
      F1=RP44(X[i],Y[i])
      F2=RP44(X[i]+h/2.,Y[i]+h/2.*F1)
      F3=RP44(X[i]+h/2.,Y[i]+h/2.*F2)
```

```
        F4=RP44(X[i+1],Y[i]+h*F3)
        Y[i+1]=Y[i]+(F1+2*F2+2*F3+F4)*h/6.
        er=fabs(RealFunc(X[i+1])-Y[i+1])
        if er>maxi:
            max=er
        print "X[%d]=%.8f-->Y[%d]=%.8f-->RealY=%.8f-->Error=%.8f"
%(i+1,X[i+1],i+1,Y[i+1],RealFunc(X[i+1]),er)

def Adams_Bashforts(X,h,Y,n):
    print "Метод Адамса-Бешфорта"
    for i in xrange(3,n):
        Y[i+1]=Y[i]+(h/24.)*(55*RP44(X[i],Y[i])-59*RP44(X[i-1],Y[i-1])+\
                        37*RP44(X[i-2],Y[i-2])-9*RP44(X[i-3],Y[i-3]))
        er=fabs(RealFunc(X[i+1])-Y[i+1])
        if er>maxi:
            max=er
        print "X[%d]=%.8f-->Y[%d]=%.8f-->RealY=%.8f-->Error=%.8f"
%(i+1,X[i+1],i+1,Y[i+1],RealFunc(X[i+1]),er)

maxi=0
n=10
h=1./n

X=np.array([0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.])
Y=np.array([1.,0.,-0.,0.,0.,0.,0.,0.,0.,0.,0.])
Runge44(X,h,Y,n)
Adams_Bashfort(X,h,Y,n)
```

Рис. 9.14. Реалізація алгоритму Адамса-Бешфортса

Для цієї реалізації (рис. 9.14) взяті дані з прикладу 9.3. Результат роботи програми представлено на рис. 9.15. Основну нагрузку в програмі виконує функція Adams_Bashforts, на вхід якої подаються дані, розраховані на першому кроці багатокрокового алгоритму за допомогою функції Runge44. Обидві функції мають однакові вхідні параметри: X – робоча точка, h – крок, Y – вектор початкових умов, який після виконання буде утримувати розраховані дані, n – кількість точок. Функція RP44 розраховує похідну в необхідній точці, функція RealFunc розраховує дійсне значення функції в точці X.

```
>>>
Метод Рунге-Кутта 4-го порядку
X[1]=0.10000000-->Y[1]=1.22210167-->RealY=1.22210414-->Error=0.00000247
X[2]=0.20000000-->Y[2]=1.49773064-->RealY=1.49773705-->Error=0.00000640
X[3]=0.30000000-->Y[3]=1.84316587-->RealY=1.84317820-->Error=0.00001233
X[4]=0.40000000-->Y[4]=2.27829046-->RealY=2.27831139-->Error=0.00002093
X[5]=0.50000000-->Y[5]=2.82738964-->RealY=2.82742274-->Error=0.00003310
X[6]=0.60000000-->Y[6]=3.52012537-->RealY=3.52017538-->Error=0.00005001
X[7]=0.70000000-->Y[7]=4.39272680-->RealY=4.39279995-->Error=0.00007315
X[8]=0.80000000-->Y[8]=5.48944418-->RealY=5.48954864-->Error=0.00010446
X[9]=0.90000000-->Y[9]=6.86432478-->RealY=6.86447120-->Error=0.00014641
X[10]=1.00000000-->Y[10]=8.58338196-->RealY=8.58358415-->Error=0.00020219
Метод Адамса-Бешфорта
X[4]=0.40000000-->Y[4]=2.27804735-->RealY=2.27831139-->Error=0.00026405
X[5]=0.50000000-->Y[5]=2.82673848-->RealY=2.82742274-->Error=0.00068426
X[6]=0.60000000-->Y[6]=3.51893335-->RealY=3.52017538-->Error=0.00124203
X[7]=0.70000000-->Y[7]=4.39079188-->RealY=4.39279995-->Error=0.00200807
X[8]=0.80000000-->Y[8]=5.48648674-->RealY=5.48954864-->Error=0.00306190
X[9]=0.90000000-->Y[9]=6.85998622-->RealY=6.86447120-->Error=0.00448498
X[10]=1.00000000-->Y[10]=8.57719808-->RealY=8.58358415-->Error=0.00638607
```

Рис. 9.15. Результати роботи програми

Приклад 9.5. Методом Адамса з кроком ***h=0.1*** отримати чисельний розв'язок диференціального рівняння $y' = (y + x)^2$ з початковими умовами

$y(0) = 0$ на інтервалі ***[0, 1.0]*** . Чисельне рішення порівняти з точним рішенням $y = tg(x) - x$.

Розв'язок. Дане завдання на першій половині інтервалу співпадає із завданням з прикладу 9.1. Тому для знаходження рішення в перших вузлах будемо використовувати результати розв'язку тієї задачі методом Рунге-Кутти четвертого порядку (табл. 9.8).

Табл. 9.8. Результати розрахунків прикладу

k	x_k	y_k	$f(x_k,y_k)$	$y_{точн}$	ε_k
0	0,0	0,0000000	0,00000000	0,000000	0,0000000
1	0,1	0,000334589	0,010067030	0,00033467	0,8301E-07
2	0,2	0,002709878	0,041091295	0,002710036	0,1573E-06
3	0,3	0,009336039	0,095688785	0,009336250	0,2103E-06
4	0,4	0,022715110	0,178688064	0,022793219	0,781090E-04
5	0,5	0,046098359	0,298223418	0,046302490	0,204131E-03
6	0,6	0,083724841	0,467479658	0,084136808	0,411968E-03
7	0,7	0,141501753	0,708125200	0,142288380	0,786628E-03
8	0,8	0,228133669	1,057058842	0,229638557	0,150489E-02
9	0,9	0,357181945	1,580506443	0,360158218	0,297627E-02
10	1,0	0,551159854	2,406096892	0,557407725	0,624787E-02

9.3.3. Метод Мілна

В цьому методі на етапі прогнозу використовується формула Мілна:

$$y_{i+1} = y_{i-3} + \frac{4}{3}h(2y'_i - y'_{i-1} + 2y'_{i-2}) + O(h^5), \qquad (9.22)$$

де $O(h^5) = \frac{28}{90}h^5 y^{(5)}$ – похибка формули прогнозу, а на етапі корекції – формула Сімпсона:

$$y_{i+1} = y_{i-1} + \frac{1}{3}h(y'_{i+1} + 4y'_i + y'_{i-1}) + O(h^5), \qquad (9.23)$$

де $O(h^5) = -\frac{1}{90}h^5 y^{(5)}$ – похибка формули корекції.

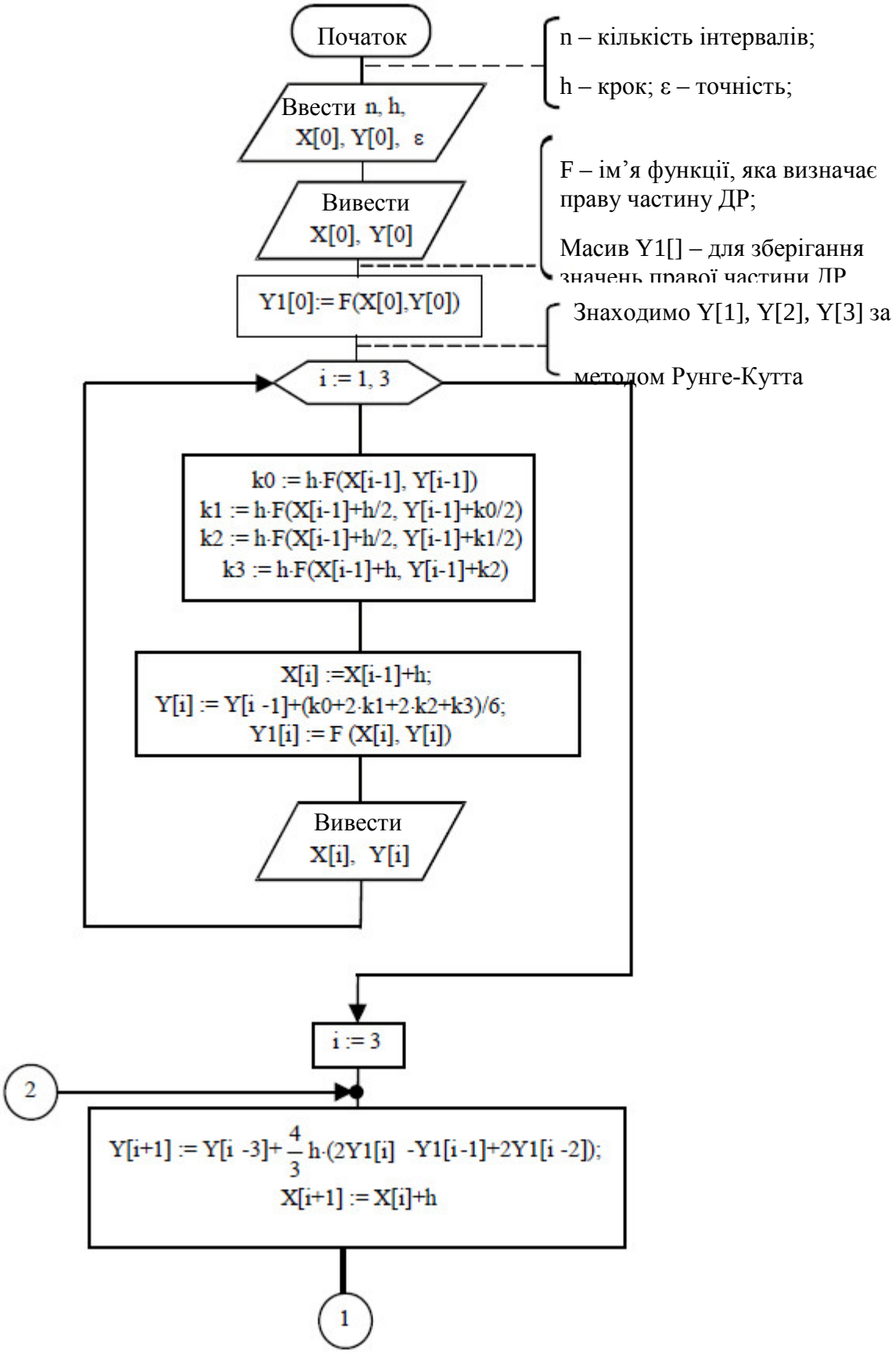
Вказані похибки в обох формулах в дійсності в ітераційному процесі не використовуються, а потрібні лише для оцінки похибки методу. Значення похідних в формулах приймаються рівними значенню правої частини диференційного рівняння.

Метод Мілна відносять до методів четвертого порядку точності, так як в ньому відкидаються члени, що мають $\boldsymbol{h}$ в п’ятій та більших степенях. Може виникнути питання: навіщо потрібно корекція, якщо прогноз вже має четвертий порядок точності?

Відповідь на це питання дає оцінка відносної величини членів, що визначають похибку. В даному випадку похибка усікання при корекції в 28 раз менше і тому представляє великий інтерес.

Взагалі формули корекції набагато точніші, ніж формули прогнозу, і тому їх використання виправдане, хоч і пов’язане з додатковими складностями.

Нижче наведена блок-схема даного методу (рис. 9.16). Ця ж блок-схема є справедливою і для метода Хемінга (п. 9.3.4) – потрібно лише замінити формули прогнозу та корекції.


Початок
n – кількість інтервалів;
h – крок; ε – точність;
Ввести n, h, X[0], Y[0], ε
F – ім'я функції, яка визначає праву частину ДР;
Масив Y1[] – для зберігання
Вивести X[0], Y[0]
Y1[0]:= F(X[0],Y[0])
Знаходимо Y[1], Y[2], Y[3] за
i := 1, 3
методом Рунге-Кутта
k0 := h·F(X[i-1], Y[i-1])
k1 := h·F(X[i-1]+h/2, Y[i-1]+k0/2)
k2 := h·F(X[i-1]+h/2, Y[i-1]+k1/2)
k3 := h·F(X[i-1]+h, Y[i-1]+k2)
X[i] :=X[i-1]+h;
Y[i] := Y[i -1]+(k0+2·k1+2·k2+k3)/6;
Y1[i] := F (X[i], Y[i])
Вивести X[i], Y[i]
i := 3
2
Y[i+1] := Y[i -3]+ 4/3 h·(2Y1[i] -Y1[i-1]+2Y1[i -2]);
X[i+1] := X[i]+h
1

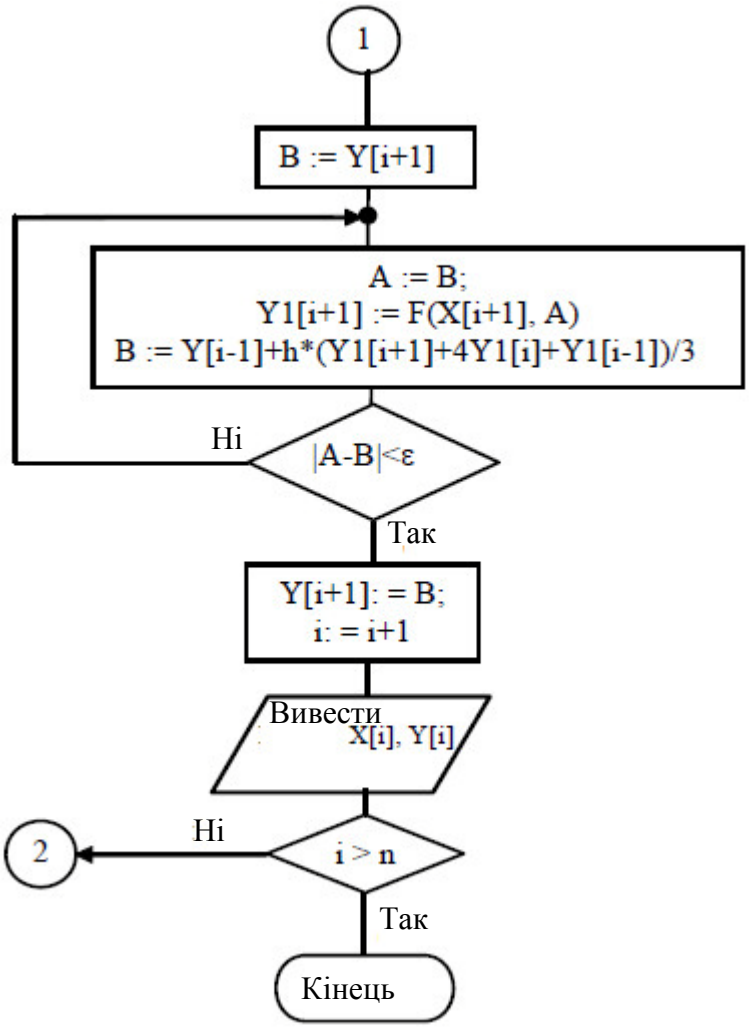


Рис. 9.16. Блок-схема алгоритму методу Мілна

Реалізація запропонованого алгоритму пропонується як самостійна вправа.

9.3.4. Метод Хемінга

Це стійкий метод четвертого порядку точності, в основі якого лежать наступні формули прогнозу:

$$y_{n+1} = y_{n-3} + \frac{4}{3}h(2y'_n - y'_{n-1} + 2y'_{n-2}) + O(h^5), \qquad (9.24)$$

де $O(h^5) = \frac{28}{90}h^5 y^{(5)}$ – похибка формули прогнозу, а на етапі корекції – формула:

$$y_{n+1} = \frac{1}{8}[9y_n - y_{n-2} + 3h(y'_{n+1} + 2y'_n - y'_{n-1})] + O(h^5), \qquad (9.25)$$

де $O(h^5) = -\frac{1}{40}h^5 y^{(5)}$ – похибка формули корекції.

Особливістю метода Хемінга є те, що він дозволяє оцінювати похибки, що вносяться на стадіях прогнозу та корекції та ліквідовувати їх. Завдяки

простоті та стійкості цей метод є одним з найбільш поширених методів прогнозу та корекції.

Блок-схема методу представлена на рис. 9.16. Потрібно лише замінити в ній формули прогнозу та корекції.

9.3.5. Метод Гіра

Одним з методів Рунге-Кутта отримаємо розв'зок y_1, y_2, y_3 задачі Коші

$$y' = f(x, y),\ y(x_0) = y_0 \tag{9.26}$$

в точках x_1, x_2, x_3. В околі вузлів $x_0,\ldots,x_4$ шуканий розв'язок $y(x)$ наближено заміним інтерполяційним поліномом Ньютона четвертого степеня:

$$y(x) = y_0 + y_{01}(x - x_0) + y_{012}(x - x_0)(x - x_1) + y_{0123}(x - x_0)(x - x_1)(x - x_2) + y_{01234}(x - x_0)(x - x_1)(x - x_2)(x - x_3), \tag{9.27}$$

де $y_{01},\ldots,y_{01234}$ – розділені різниці першого-четвертого порядків.

Ліву частину рівняння (9.26) наближено знайдемо шляхом диференціювання по x полінома (9.27)

$$\begin{aligned} y'(x) = {} & y_{01} + y_{012}(x - x_0 + x - x_1) \\ & + y_{0123}[(x - x_0)(x - x_1) + (x - x_0)(x - x_2) + (x - x_1)(x - x_2)] \\ & + y_{01234}[(x - x_0)(x - x_1)(x - x_2) + (x - x_0)(x - x_1)(x - x_3) \\ & + (x - x_0)(x - x_2)(x - x_3) + (x - x_1)(x - x_2)(x - x_3)] \end{aligned} \tag{9.28}$$

Розділені різниці для рівновіддалених вузлів виражаються через вузлові значення апроксимуючої функції

$$\begin{aligned} & y_{01} = (y_1 - y_0)/h, \\ & y_{012} = (y_2 - 2y_1 + y_0)/(2h^2), \\ & y_{0123} = (y_3 - 3y_2 + 3y_1 - y_0)/(6h^3), \\ & y_{01234} = (y_4 - 4y_3 + 6y_2 - 4y_1 + y_0)/(24h), \end{aligned} \tag{9.29}$$

де $h = x_{i+1} - x_i$.

Приймаючи в виразі для похідної (9.28) значення аргумента $x=x_4$ і враховуючи значення розділених різниць (9.29), отримаємо

$$y'(x_4) = (3y_0 - 16y_1 + 36y_2 - 48y_3 + 25y_1)/(12h). \quad (9.30)$$

З іншого боку, рівняння (9.26) при приймає $x=x_4$ вигляд

$$y'(x_4) = f(x_4, y_4). \quad (9.31)$$

Прирівняємо праві частини співвідношень (9.30) та (9.31) і знайдемо

$$y_4 = [3(4hf(x_4, y_4) - y_0) + 16y_1 - 36y_2 + 48y_3]/25. \quad (9.32)$$

Формула (9.32) представляє собою неявну схему Гіра четвертого порядку для розв'язування задачі Коші. Змінюючи кількість вузлів x_j можна аналогічним способом отримати формули Гіра як більш низьких, так і більш високих порядків.

Неявні алгоритми Гіра найбільш ефективні для розв'язування так званих жорстких рівнянь, особливістю яких є повільна зміна їх розв'язків за наявності швидко затухаючих збурень. Жорсткими рівняннями моделюються перехідні процеси в нелінійних електронних схемах, і застосування неявних методів прискорює на декілька порядків інтегрування в порівнянні з явними методами.

Для знаходження значення y_4 з рівняння (9.32) можна застосувати метод простих ітерацій. Але, для реалізації переваг неявного методу в відношення вибору кроку при інтегруванні жорстких рівнянь рекомендується використовувати метод Ньютона. Для будь-якого з вибраних методів потрібно знати початкове наближене до шуканої величини y_4. Приймаючи в виразі для похідної (9.28) значення аргумента $x=x_3$, отримаємо

$$y'(x_3) = (-y_0 + 6y_1 - 18y_2 + 10y_3 + 3y_4)/(12h). \quad (9.33)$$

Прирівнюючи праві частини вихідного рівняння (9.26) при $x=x_3$ та виразу (9.33), отримаємо схему прогнозу, за допомогою якої можна знайти початкове наближення для розв'язку рівняння (9.32)

$$y_4 = 4hf(x_3, y_3) + \frac{y_0 - 10y_3}{3} - 2y_1 + 6y_2. \quad (9.34)$$

Реалізація методу Гіра у вигляді модуля *Python* представлена на рис. 9.17.

```
import numpy as np
from math import fabs
```

```
def RP(P,X,Y,F):
    F[1]=Y[2]
    F[2]=((P/X)**2-1.)*Y[1]-Y[2]/X
    return F


def Gir(P,X,h,Y,n):
    F=np.zeros((4)); Y1=np.zeros((4))
    D=np.zeros((3,9))
    H4=4*h
    for k in xrange(3):
        for i in xrange(1,n+1):
            D[k,i]=Y[i]
        RK4(N,X,h,Y)
        X+=h
        print "X=%.8f-->Y[1]=%.8f-->Y[2]=%.8f" %(X,Y[1],Y[2])
    while True:
        RP(P,X,Y,F)
        for i in xrange(1,n+1):
            Y1[i]=Y[i]
            Y[i]=(D[0,i]-10.*Y[i])/3.-2*D[1,i]+6.*D[2,i]+H4*F[i]
        X+=h
        RP(P,X,Y,F)
        for i in xrange(1,n+1):
            Y[i]=(48.*Y1[i]-36.*D[2,i]+16*D[1,i]-\
                3.*(D[0,i]-H4*F[i]))/25.
```

```
            D[0,i]=D[1,i]
            D[1,i]=D[2,i]
            D[2,i]=Y1[i]
        print "X=%.8f-->Y[1]=%.8f-->Y[2]=%.8f" %(X,Y[1],Y[2])

        if (X>=X9) == (h>0.):
            break

    return X,Y,h
```

Рис. 9.17. Реалізація методу Гіра

Написання програми застосування запропонованого модуля реалізації методу Гіра (рис. 9.17) пропонується як самостійна вправа.

9.4. Рішення систем звичайних диференційних рівнянь

9.5. Крайові задачі

9.6. Контрольні запитання

1. Від чого залежить точність отримуваного результата?

2. Що таке якість «самостартування»?

3. У чому відмінність одно крокових методів від багатокрокових?

4. Наскільки є точнішим модифікований метод Ейлера від звичайного?

5. Яким чином визначається порядок точності метода?

6. Метод Рунге-Кутта 2-го порядку.

7. Метод Рунге-Кутта 4-го порядку.

8. Метод Рунге-Кутта-Мерсона.

9. Який головний недолік багатокрокових методів.

10. Від чого залежить точність багатокрокових методів?

11. Перерахувати переваги методів прогнозу та корекції.

12. Метод Адамса.

13. Метод Гіра.

14. Метод Хеммінга.

15. Чи є можливим застосування однокрокових та багатокрокових методів для розв’язку систем звичайних диференційних рівнянь?

16.Скільки початкових умов повинно бути задано для розв’язування крайової задачі?

17. Різнецевий метод розв’язку крайової задачі.

18. Метод стрільби.