Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

ПРОГРАМУВАННЯ ЧИСЛОВИХ МЕТОДІВ МОВОЮ *PYTHON*

Навчальний посібник

За редакцією
чл.-кор. НАН України А. В. Анісімова

Рекомендовано Міністерством освіти і науки,
молоді та спорту України як навчальний посібник
для студентів вищих навчальних закладів

КИЇВСЬКИЙ УНІВЕРСИТЕТ ВІТЦ 2013

УДК 004.43(075.8)
ББК 22.183.492я73
П78

Автори:
А. Ю. Дорошенко, С. Д. Погорілий,
Я. Ю. Дорогий, Є. В. Глушко

Рецензенти:
д-р техн. наук, проф. О. А. Молчанов,
д-р техн. наук, проф. Є. О. Башков,
д-р фіз.-мат. наук, проф. М. М. Глибовець,
д-р техн. наук, проф. О. Л. Синявський

Рекомендовано до друку вченою радою факультету кібернетики
(протокол № 3 від 28 листопада 2011 року)

Ухвалено науково-методичною радою
Київського національного університету імені Тараса Шевченка
28 жовтня 2011 року

П78 Програмування числових методів мовою *Python* : навч. посіб. / А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий, Є. В. Глушко ; за ред. А. В. Анісімова. – К. : Видавничо-поліграфічний центр "Київський університет", 2013. – 463 с.

ISBN 978-966-439-693-3

Викладено основні відомості про засоби та методи програмування мовою *Python*, яка є універсальною інтерпретованою, об'єктно-орієнтованою мовою програмування високого рівня, що розвивається у відкритих початкових кодах на багатьох платформах і надається безкоштовно для загального користування.

Для науковців, інженерів, студентів вищих навчальних закладів із напрямів підготовки "Інформатика", "Прикладна математика", "Програмна інженерія", "Комп'ютерна інженерія", "Системна інженерія", а також для всіх зацікавлених у швидкому розробленні прикладних програм наукового спрямування.

УДК 004.43(075.8)
ББК 22.183.492я73

Гриф надано Міністерством освіти і науки, молоді та спорту України
(лист № 1/11-4486 від 04.04.12)

ISBN 978-966-439-693-3

ЗМІСТ

ПЕРЕДМОВА

Мова *Python* є, мабуть, однією з найпростіших у вивченні й використанні серед найбільш поширених мов програмування. Програмний код мовою *Python* легко читати й писати, і, при всій лаконічності, він не виглядає загадковим. І все це завдяки тому, що *Python* – дуже виразна мова, що дозволяє умістити застосування в меншу кількість рядків, ніж це потрібно з використанням інших мов, таких як C + + або Java. *Python* є мультиплатформною мовою: зазвичай одна й та сама програма мовою *Python* може запускатися в різних операційних системах (Windows, UNIX, Linux, BSD і Mac OS), для чого треба просто скопіювати файл або файли програми на потрібний комп'ютер; при цьому навіть не потрібно виконувати "збірку", або компіляцію програми.

Однією з переваг мови *Python* є, зокрема, наявність повної стандартної бібліотеки, що дозволяє задовольнити найбуденніші вимоги користувачів, наприклад, завантажити файл з Інтернету, розпакувати архів або створити веб-сервер за допомогою кількох рядків програмного коду. Крім того, існують тисячі додаткових бібліотек сторонніх виробників, які забезпечують складніші й більш потужні можливості, наприклад, бібліотека для організації мережних взаємодій Twisted, бібліотека для виконання обчислювальних завдань Numpy або пакет моделювання Simpy. При цьому більшість сторонніх бібліотек можна знайти в Інтернеті. Саме ці обставини і спонукали авторів зупинити свій вибір на мові програмування *Python*.

Python може використовуватися для програмування у процедурному, об'єктно-орієнтованому і, меншою мірою, у функціональному стилі програмування, хоча загалом *Python* – це об'єктно-орієнтована мова. Основна мета цього посібника – надати студентам базові відомості про мову *Python*, необхідні для програмування прикладних задач, і допомогти навчитися писати процедурні й об'єктно-орієнтовані програми для виконання лабораторних робіт із числових методів (і не тільки).

Автори вважають, що читачі посібника мають деякий досвід програмування (будь-якою мовою). Зокрема, передбачається наявність знань про типи даних (наприклад, числа і рядки), колекції даних (наприклад, множини і списки), керуючі структури (такі як інструкції **if** і **while**) і функції. Крім того, деякі приклади припускають знання основ мови розмітки HTML, а деякі більш спеціалізовані розділи передбачають наявність базових знань з обговорюваних тем, наприклад, із баз даних. Проте ці знання не виходять за рамки підготовки школярів з інформатики, а матеріал посібника є цілком доступним для студентів молодших курсів вищих навчальних закладів, що навчаються за напрямами підготовки "Інформатика", "Прикладна математика", "Програмна інженерія", "Комп'ютерна інженерія", "Системна інженерія". Видання буде корисним для науковців, інженерів, а також для всіх зацікавлених у швидкому розробленні прикладних програм наукового спрямування.

ЧАСТИНА 1
ПРОГРАМУВАННЯ МОВОЮ *PYTHON*

Розділ 1
ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ *PYTHON*

1. 1. Мова й інтерпретатор *PYTHON*

Історія створення й розвитку мови пов'язана з життєвими та службовими проблемами її автора Гвідо ван Россума. Назва мови виникла зовсім не від назви виду плазунів, а на честь популярного британського комедійного телешоу 1970-х років "Літаючий цирк Монті Пайтона". Утім, незважаючи на це, назву мови частіше асоціюють саме зі змією, аніж із передачею – піктограми файлів у KDE або в Microsoft Windows і навіть емблема на сайті python.org (до виходу версії 2.5) зображають зміїні голови.

Наявність дружелюбної, чуйної спільноти користувачів вважається поряд із дизайнерською інтуїцією Гвідо одним із факторів успіху *Python*. Розвиток мови відбувається згідно з чітко регламентованим процесом створення, обговорення, відбору та реалізації документів PEP (англ. Python Enhancement Proposal) – пропозицій щодо розвитку *Python*.

1.1.1. Ліцензійні угоди з використання програмного забезпечення мови *Python*

Не можна забувати про важливість вивчення авторських прав і ліцензійних угод, програмуючи з відкритим кодом. Авторські права на програмне забезпечення з відкритим кодом **(Open Source)** та ліцензійні угоди утворюють химерні ланцюжки документів. Не варто приділяти велику увагу заявам конкуруючих виробників комерційного програмного забезпечення (ПЗ) про економічні загрози, що несуть ліцензії на ПЗ із відкритим кодом. Ліпше поговорити із постачальником, що просуває на ринку ті інструменти, які вам сподобалися. Скоріше за все виявиться, що умови ліцензійної угоди дозволять вам заощадити гроші,

забезпечити необхідну гнучкість та мобільність, а також скористатися іншими унікальними перевагами мов та інструментальних засобів із відкритим кодом.

Приклад перекладу фрагмента ліцензійної угоди: "Дозвіл на використання, копіювання, модифікування та поширення цього програмного продукту та його документації, із будь-якою метою і без усякої оплати, цим тут підтверджується, за умови, що наведене вище зауваження про авторське право буде наявне в усіх копіях, і, що обидва зауваження, про авторське право і про цей дозвіл, будуть наявні в документації підтримки, і, що імена "Stichting Mathematish Centrum" (CWI) та "Corporation for National Research Initiatives" (CNRI), не будуть використані в рекламі, що стосується поширення програмного продукту, без особливого, письмового на те дозволу".

CWI є першоджерелом для даного програмного продукту, а Corporation for National Research Initiatives зробила доступною в Інтернеті модифіковану версію за адресою ftp://ftp.python.org/".

Версії мови *Python* і її інтерпретатора часто й радикально змінюються [див. сайт http://www.python.org/]. Тому необхідно відслідковувати та враховувати нововведення згідно з наведеним вище посиланням.

1.1.2. Порівняння мови *Python* з іншими мовами програмування. Її переваги та вади

Мови програмування часто оцінюють за рівнем: процедурні, об'єктно-орієнтовані, функціональні, логічні. Рівень мови показує, наскільки мова близька до природного для людини запису.

Процедурні мови – найнижчого рівня, функціональні – значно вищого. Логічні мови принципово могли б належати до найвищого рівня, але через високу складність теорії, що лежить у їхній основі, розробляються досить повільно. Використання об'єктів здатне принести досить скромні дивіденди: можливе збільшення продуктивності програмістів у діапазоні 20–30 %, а зовсім не вдвічі й тим більше не в 10 разів. Змінні та функції в об'єктно-орієнтованому програмуванні (ООП) групуються у так звані класи. Завдяки цьому досягається вищий рівень структу-

ризації програми. Об'єктно-орієнтований спосіб написання програм не є чимось особливим і самостійним, тому що базується на процедурній моделі програмування.

1.1.3. Переваги та вади ООП порівняно з мовами сценаріїв

У кожному разі, об'єкти не збільшують продуктивність настільки істотно, як сценарна (скриптова) технологія, а забезпечуваних ними переваг можна досягнути й за допомогою мов сценаріїв (скриптів). Сильна типізація більшості об'єктно-орієнтованих мов робить модулі досить спеціалізованими, ускладнюючи їхнє повторне використання. Інша проблема об'єктно-орієнтованих мов – їхній акцент на спадкуванні. Реалізації класів прив'язуються одна до одної, так що жоден клас не можна зрозуміти без іншого.

Мови сценаріїв фактично втілили в життя принцип повторного використання. Модель, яку вони застосовують при створенні програм, враховує те, що необхідні компоненти вже є в системі, їх можна "склеювати" за допомогою мови сценаріїв. Такий "поділ праці" забезпечує природну схему, що полегшує повторне використання коду.

Проте об'єктно-орієнтоване програмування має принаймні дві корисні властивості. Перша – інкапсуляція: об'єкти поєднують дані та код способом, що приховує деталі реалізації. Друга – спадкування інтерфейсу, при якому класи забезпечують ті самі методи й атрибути, навіть якщо вони реалізовані по-різному. Переваг об'єктів у мовах програмування систем можна досягнути й у **мовах сценаріїв**. При цьому зберігається властива мовам сценаріїв відсутність у об'єктів типу.

Python – це універсальна інтерпретована, об'єктно-орієнтована високорівнева мова програмування сценаріїв із динамічною семантикою. Розвинені вбудовані структури даних у поєднанні з динамічною типізацією та динамічним зв'язуванням роблять її дуже привабливою для швидкої розробки застосувань, а також для використання як скриптової або мови, що "склеює" разом наявні компоненти. Мова *Python* є простою, має легкий у вивченні синтаксис, забезпечує високу читабельність і через те зменшує загальну вартість експлуатації програм. *Python* підтримує модулі та пакети, які сприяють мобільності програм і по-

вторному використанню коду. Інтерпретатор *Python* та розширена стандартна бібліотека доступна як у вихідному коді, так і у двійковому форматі, причому безкоштовно і для всіх основних платформ. Його можна вільно поширювати та навіть убудовувати у власні застосування.

1.1.4. Використання інтерпретатора *Python*

Запуск інтерпретатора звичайно здійснюється просто командою **python.exe**, або вказуванням повного шляху до інтерпретатора. Для того, щоб вийти з *Python*, треба скористатися комбінацією клавіш **CTRL+D** – Unix; **CTRL+Z** або **CTRL+Break** – Dos + + Windows; якщо це не допомогло, то можна набрати у відповідь на запрошення інтерпретатора **(>>>)** такі рядки:

```
>>> import sys
sys.exit()
```

Інтерпретатор працює у двох режимах: інтерактивному та власне інтерпретатора. Вхід в інтерактивний режим здійснюється введенням **python** без параметрів, параметр **file** викликає інтерпретацію (виконання) зазначеного файла. В інтерактивному режимі *Python* пише інформацію про себе та про систему, а потім виводить своє запрошення **(>>>)**. Необхідно ввести рядок коду *Python* із клавіатури. Змусити *Python* інтерпретувати введений вами рядок можна клавішею **Enter**.

Коментарі в *Python* позначаються символом # і продовжуються до кінця рядка:

```
>>> a = "Це рядок" # це коментар
>>> b = "# це вже НЕ коментар"
```

Програма (скрипт) на *Python* – це команди, які записано в текстовому файлі з розширенням **.py**. Програмі можна передати параметри через командний рядок, розділяючи їх пробілами. Вони передаються у список **sys.argv[i]** для подальшої обробки у програмі. За необхідності треба вказати повні шляхи до файлів:

```
python.exe скрипт.py параметри
```

1.1.5. Середовище програмування для *Python*

Існує багато різних графічних середовищ програмування для *Python*. Ефективність роботи програміста істотно залежатиме від правильного вибору такого середовища.

Tkinter – це обгортка для Tcl/Tk. *Tkinter* є стандартним прикладним графічним інтерфейсом (GUI) для *Python* і багато різних застосувань написано саме на ньому. Це найкращий із кросплатформних GUI.

Idle – *Python* IDE побудований із використанням комплекту інструментальних засобів *Tkinter* GUI. Програма написана виключно на *Python* із використанням *Tkinter* GUI. Працює на ОС Windows і Unix.

WxPython. Після Tkinter, це пакет віджетів загального призначення "№ 2", але його не портовано на таку кількість платформ як *Tkinter*.

Pythonwin є обгорткою для Microsoft Foundation Classes (MFC). У ньому можна писати застосування, які дуже тісно пов'язані з Windows, використовуючи особливості Windows GUI.

1.2. Неформальний вступ до мови *Python*

Вступ ми назвали "неформальним" через застосований тут дещо нетрадиційний підхід до початку вивчення мови. Методика навчання базується на прикладах, а не на формальних визначеннях синтаксису та семантики. Таким чином, забезпечується швидкий початок практичних занять, а вивчення формальних визначень відкладається на наступні розділи.

1.2.1. Інтерпретатор *Python* як калькулятор. Числа

Інтерпретатор працює як простий калькулятор. Подібно до мови програмування С, символ рівності (**'='**) використовується для присвоювання значення змінній. Присвоєне значення при цьому не виводиться. Але можна набрати вираз й інтерпретатор виведе результат його обчислення:

```
>>> x=2*2-2^2#Це арифметичний вираз
>>> x#Послідовність виконання операцій: *,-,^,=
0
>>> x==2*2-2^2#Це логічний вираз
1
>>>_#змінна _ містить останній результат обчислень, 1
```

В арифметичних виразах можна використовувати арифметичні операції: 1) **; 2) ***,/,%***; 3) **+,–**. Операції розділено на групи з однаковими пріоритетами. Групи операцій перераховано в порядку зменшення пріоритетів. Найпростіші логічні вирази – це відношення (нерівності). Вони складаються з виразів, між якими ставиться знак операції відношення (оператор): **<, >, ==**, **<=** (менше або дорівнює), **>=** (більше або дорівнює) і **!=**, **<>** (не дорівнює). В останньому прикладі результат виявився рівним 1 (якщо не 0, то істина, 0 – хибність, а окремого логічного типу даних немає). Значення можна присвоїти одночасно декільком змінним:

>>> a=b=c=-2/4; A=B=C=float(-2/4); C=-2.0/4

>>> a, b, c, A, B, C

(-1, -1, -1, -1.0, -1.0, -0.5) #операція з цілими дає ціле число

Python **розрізняє** великі та малі літери в іменах змінних і ключових словах. Декілька інструкцій можна записати в одному рядку, ро̀зділяючи їх символом ";". Наявні функції перетворення в ціле число та в число з рухомою комою: **int(), long()** та **float().** Необмежене довге ціле число закінчується символом **l** або **L**. Наприклад: **10000000000000000L**. Послідовне присвоювання не визначає нових посилань. Утворюються різні імена одного й того самого посилання.

>>> id(a), id(b), id(c), id(A), id(B), id(C)

(7754680, 7754680, 7754680, 20057992, 20057992, 20058040)

Убудована функція **id(object)** повертає ідентифікатор об'єкта **object** – ціле число (типу **int** або **long**). Гарантується унікальність ідентифікатора протягом усього часу існування об'єкта. Насправді, поки об'єкт існує, ідентифікатор є його адресою. Тому ідентифікатор змінної, створеної пізніше (як у вищенаведеному прикладі), зазвичай матиме більше числове значення.

Щоб обчислювати за формулами, що містять елементарні математичні функції, необхідно імпортувати модуль **math**, де їх описано.

>>> import math

>>> uno=math.sin(0.3*math.pi)2 + math.cos(0.3*math.pi)**2**

>>> uno

1.0

1.2.2. Пріоритети операцій

Пріоритети операцій (за зростанням) наведено у табл. 1.1.

Таблиця 1.1. **Операції в порядку зростання пріоритетів**

Операція	Опис
Lambda	Лямбда-вираз
Or	Логічне АБО
And	Логічне І
Not	Логічне НЕ
in, not in	Перевірка належності
is, is not	Перевірка ідентичності
<, <=, >, >=, <>, !=, ==	Порівняння
|; ^; &	Побітове АБО; виключне АБО; І
<<, >>	Зсуви
+, -; *, /, %	Додавання, віднімання; множення, ділення
+x, -x	Унарні плюс і мінус
~x	Побітове НЕ
****	Піднесення до степеня
x[індекс]	Взяття елемента за індексом
x[від:до]	Виділення зрізу "від" і "до"
f(аргумент,...)	Виклик функції
(...)	Дужки або кортеж
[...]	Список або спискове включення
{ключ:дане,...}	Словник
`вираз,...`	Перетворення в рядок (' – наголос)

1.2.3. Стандартний модуль math

До модуля входять змінні **pi**, **e**. Функції (аналогічні функціям мови програмування С) наведено в табл. 1.2.

Таблиця 1.2. **Функції модуля math**

acos(x)	**cosh(x)**	**ldexp(x,y)**	**sqrt(x)**
asin(x)	**exp(x)**	**log(x)**	**tan(x)**
atan(x)	**fabs(x)**	**sinh(x)**	**frexp(x)**
atan2(x,y)	**floor(x)**	**pow(x,y)**	**modf(x)**
ceil(x)	**fmod(x,y)**	**sin(x)**	
cos(x)	**log10(x)**	**tanh(x)**	

1.2.4. Імпорт модулів

Інструкція **import ім'я_модуля** має досить важливе значення через те, що *Python* побудовано з модулів. Якщо ви збираєтесь використовувати функцію часто, можете присвоїти її локальній змінній:

```
>>> import math
>>> sn=math.sin; cs=math.cos; p=math.pi
>>> uno=sn(0.3*p)**2 + cs(0.3*p)**2; uno
1.0
```

Інший варіант інструкції **from ім'я_модуля import *** імпортує всі імена з модуля, за винятком таких, що починаються із символу **"_"**.

```
>>> from math import * # може бути перекриття існуючих імен
>>> print tan(pi/4)# краще from math import sin, tan
1.0
```

Тип значення змінної або виразу можна дістати за допомогою функції **type**:

```
>>> type(uno); type(sn); type('a')
<type 'float'>
<type 'builtin_function_or_method'>
<type 'str'>
```

type(object) – повертає тип об'єкта **object**. Значення, що повертається, є об'єктом типу. У стандартному модулі **types** визначено імена для всіх убудованих типів даних.

1.2.5. Рядки

Крім чисел, *Python* також працює з рядками, які записують різними способами. Вони можуть міститись в одинарних (апострофи) або подвійних лапках:

```
>>> a='Рядок1\n'; b="String2"
>>> a,b
('\xd1\xf2\xf0\xee\xea\xe01\n', 'String2')
>>> print a,b
Рядок1
String2
```

Інструкція **print** застосовується для виведення результатів обчислення виразів або значень змінних тощо. Замість символів кирилиці в найліпшому разі ви побачите їхні коди. Для правильного виведення в інтерпретаторі кириличних рядків слід скористатися інструкцією **print**. Символи, що керують виведенням, записуються після "\". Перетворення n на рядковий тип відбувається за допомогою виклику **str(n)** або запису **`n`** (` – наголос).

Якщо рядок занадто довгий, то можна розмістити його в декількох рядках, вказавши у кінці рядка символ \, *наприклад:*

```
>>> l = "Це дуже довгий \
рядок, що містить \
3 рядки\n"
>>> print l
Це дуже довгий рядок, що містить 3 рядки
```

Символ **\n** є так званим символом керування, що переводить виведення на наступний рядок. Кілька рядків можна розмістити між ''' або """.

Рядок – послідовність символів із довільним доступом. Ви можете отримати будь-який символ рядка за його індексом. Подібно до мови С, перший символ має індекс 0. Підрядок можна отримати за допомогою **зрізу** (slice) – двох індексів, розділених двокрапкою **[m:n]**. Символ, на який указує другий індекс, у зріз не включається.

```
>>> a = "strinG"; b = "String"
>>> print a[0:2]+b[2:]
strring
```

Індекси зрізу мають корисні значення за замовчуванням: випущений перший індекс вважається рівним 0; випущений другий індекс дає такий самий результат, як у випадку, коли б він дорівнював довжині рядка.

```
>>> len(a), len(a[1:]), len(b), len(b[0:])
(8, 7, 7, 7)
```

Корисний інваріант операції зрізу: **s[:i] + s[i:]** дорівнює **s.** Від'ємні індекси у зрізах обробляються так, наче вони дорівнюють 0. Рядки в мові *Python* неможливо змінити. Убудована функція **len()**

обчислює довжину аргументу (використовується не тільки для рядкових об'єктів, але варто звернути увагу на те, що для рядкових змінних довжина обчислюється в байтах, а не в символах).

1.2.6. Списки

Python має тип даних, що використовується для групування декількох значень, списків. Зовсім необов'язково, щоб елементи списку були одного типу. Список **(list)** можна записати як послідовність значень (елементів), розділених комами, яку вкладено у квадратні дужки.

```
>>> sp=['uno',2,'tre',4]; sp
['uno', 2, 'tre', 4]
```

Нумерація індексів списку починається з нуля. Для списку можна отримати зріз, об'єднати кілька списків тощо. Окремі елементи списку можна змінювати, можна також присвоювати нові значення зрізу.

```
>>> sp[0]=1; sp[1]='due'; sp.append(5); sp; len(sp)
[1, 'due', 'tre', 4, 5]
```

Метод **append()** додає елемент у кінець списку. Слово "метод" вжито тут не помилково, тому що змінна **sp** – це насправді об'єкт. Убудовану функцію **len()** можна також застосовувати й до списків.

Списки можуть бути вкладеними. Тобто можна створювати списки, що містять інші списки як елементи:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('вставка')
>>> p
[1, [2, 3, 'вставка'], 4]
>>> q
[2, 3, 'вставка']
```

Увага: **p[1]** та **q** посилаються на один і той самий об'єкт.

Метод **pop(i)** видаляє елемент з індексом **i** та повертає його. Якщо викликати **pop()** без параметрів, то функція поверне та видалить останній елемент списку.

1.3. Елементи програмування: інструкції while, if, for

Прості інструкції присвоювання, інструкції-вирази, інструкції виведення, імпорту й інші формують послідовне керування ходом виконання програми. У такій послідовності потрібно лише правильно визначити порядок виконання інструкцій.

Обробка складних структур даних та реалізація складних алгоритмів вимагають спеціального керування виконанням програми за допомогою відповідних інструкцій. Найпростіші інструкції, що дозволяють керувати процесом, це **while, if** та **for**.

Додаткові інструкції допомагають вирішувати завдання, які з'являються при використанні списків, кортежів та словників. Ці структури даних мають аналоги в інших мовах програмування. Винятком є словник. Крім того, *Python* не має вбудованого типу даних "масив", але одновимірний масив (клас) можна імпортувати з бібліотеки стандартних модулів **array**, багатовимірні масиви – з інших джерел.

1.3.1. Інструкція while

Наведемо приклад використання інструкції **while** для організації виведення деякої послідовності чисел. Цикл триває доти, поки відношення **i<11** повертатиме значення, що не дорівнює 0 (нагадаємо, що будь-яке ненульове значення є істиною, а 0 – це хибність).

```
>>> i,n=1,2
>>> while i<11:
...print i,n # Це блок
...i,n=n+1,i+1 #Всередині циклу
...
1 2
3 2
3 4
5 4
```

```
5 6
7 6
7 8
9 8
9 10
10
```

Перший рядок містить багаторазове присвоювання. Змінним *i* та *n* відповідно присвоюються значення 1 та 2. В останньому рядку присвоювання використовується знову, демонструючи таке: вирази в правій частині обчислюються зліва праворуч до присвоювання.

Заголовок циклу закінчується символом ":". Тіло циклу записане з відступом (чотири пробіли). Відступи використовуються в *Python* для запису групи інструкцій. Так організується блок інструкцій без використання додаткових символів (дужки "{}" у мові C) або слів (**begin** та **end** у мові Pascal).

Ознакою кінця блоку стало введення останнього, порожнього рядка:

```
>>> i=1
>>> while i<11:
...print i,
...i=i+1
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Кома в списку виведення інструкції **print** виводить пробіл, зокрема в кінці списку (замість переходу на новий рядок). Інтерпретатор сам переходить на новий рядок перед виведенням наступного запрошення, навіть якщо останній виведений рядок не завершується переходом на новий рядок.

1.3.2. Інструкція if

Слово **if** починає конструкцію вибору. Вибір між гілками (блоками) алгоритму відбувається залежно від умови, що записана за **if**. Ця інструкція виконує блок коду, що йде після нього з відступами, тільки в разі, якщо вираз у його заголовку не дорівнює 0 (тобто істина):

```
>>> x = int(raw_input("Введіть ціле число: "))  # Введення
>>> if x < 0:
...     x = 0
...     print 'Від'ємне число стало нулем'
... elif x == 0:
...     print 'Це число – нуль'
... elif x == 1:
...     print 'Це число - 1'
... else:
...     print 'Це число більше ніж одиниця'
...
```

Оператор **if**, як видно, супроводжують гілки **else** ("інакше" – блок коду виконується, якщо умова в заголовку **if** дорівнює 0, тобто є хибною) та **elif** ("інакше якщо" – блок коду виконується, якщо умова в заголовку **if** дорівнює 0, тобто є хибною, а умова в заголовку цього оператора не дорівнює 0, тобто є істинною).

У прикладі використано ще одну вбудовану функцію: **raw_input()**. Вона дозволяє вводити вихідні дані (*PythonWin* навіть пропонує діалогове вікно введення). Таким чином, інтерфейс взаємодії із користувачем став майже повним. Ви маєте функцію введення даних **raw_input()** та функцію виведення результатів **print()**. Загальний вид функції: **raw_input(запрошення)**.

Якщо вказано "запрошення", то його значення виводиться на стандартний потік виведення (на відміну від інструкції **print**, символ нового рядка у кінці не додається). Після запрошення користувач уводить значення. Функція зчитує це значення – рядок зі стандартного потоку введення– та повертає його (крім завершального символу переходу на новий рядок). Інша функція введення **input(запрошення)** повертає введене значення без перетворення його в рядок.

1.3.3. Інструкція for

Інструкція **for** має дещо незвичайний вигляд у *Python*:

for змінна in діапазон: блок

Блок коду після заголовка виконується доти, поки змінна належить діапазону (причому цей діапазон може бути списком, числовою послідовністю, рядком, іншою послідовністю яких-небудь проіндексованих значень):

```
>>> # Визначимо список:
... a = ['Linux', 'Open', 'Source']
>>> for x in a:
...     print x, len(x)

Linux 5
Open 4
Source 6
```

Не рекомендується змінювати в тілі циклу **for** значення діапазону (це може спричинити досить дивну його роботу, зациклення й ускладнити розуміння програми), крім випадків, коли в ролі діапазону виступає список. У цьому разі можна виконати просте копіювання елементів списків.

```
>>> a = ['Linux', 'Open', 'Office']
>>> for x in a[:]:# елементи списку a по черзі копіюються в
                  # змінну x
...     if len(x) >= 6: a.insert(0, x)
...
>>> a
['Office', 'Linux', 'Open', 'Office']
```

Метод **insert(0, x)** вставляє у список другий аргумент перед індексом, який визначає перший аргумент.

Для визначення діапазону як арифметичної прогресії (1 2 3 4 5 6...) зручно користуватися функцією **range()**. Вона має три форми. Розглянемо їх на прикладах.

Перша форма повертає всі цілі числа в діапазоні від 0 до числа 10, не включаючи саме число 10.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Друга форма повертає всі цілі числа в діапазоні від 5 до числа 10, не включаючи саме число 10, але включаючи початкове число 5.

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
```

Третя форма повертає всі цілі значення в діапазоні – від початкового до кінцевого (не включаючи його) із кроком, що визначається третім параметром.

```
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Причому, якщо, наприклад, ви спробуєте ввести **range (1, 100, -1)**, то чисел у цьому діапазоні не буде і, відповідно, результатом буде порожній список **[]**.

Якщо ви хочете задати діапазон через кількість елементів у списку, то варто скористатися функцією **range** у сполученні з функцією **len:**

```
>>> a = ['Linux', 'is', 'the', 'best', 'system']
>>> for i in range(len(a)):
...     print i, a[i] #звертання до елемента списку за індексом
...

0 Linux
1 is
2 the
3 best
 4 system
```

Для перебирання чисел з великого діапазону створення списку є невиправданим, а деколи просто може не вистачити пам'яті. Якщо ми не збираємося змінювати список, досить створити псевдосписок – об'єкт, для одержання значень "елементів", але не для зміни їхніх значень або порядку їхнього проходження. Для цього у мові *Python* передбачено функцію **xrange()**.

```
>>> i=range(5,10); i # Створюється список
[5, 6, 7, 8, 9]# Виведення
>>> i=xrange(5,10); i# Створюється псевдосписок
xrange(5, 10)# Виведення
>>> for i in xrange(5,10): # Псевдосписок заощаджує ресурси
... print i, # Без переходу на інший рядок
...
5 6 7 8 9
```

Псевдосписок – об'єкт, для якого ми можемо отримати значення "елементів", але не можемо змінити їх або порядок їхнього проходження.

1.3.4. Переривання та продовження циклів for і while

Для негайного виходу з циклу можна використовувати інструкцію **break**. Для продовження циклу, але з наступним значенням змінної циклу (тобто наступної ітерації) застосовують інструкцію **continue**.

Інструкція **break**, як і в мові С, виходить із найглибшого внутрішнього вкладеного циклу ***for*** або ***while***.

```
>>> for n in xrange(1, 10):
...     if n % 2 == 0:
...         print n, ' = Парне число'
...     elif n % 7 == 0:
...         print n,' = Останнє непарне число, ділиться на 7'
...         break
...     else:
...         print n, ' = Непарне число'
...
1  = Непарне число
2  = Парне число
3  = Непарне число
4  = Парне число
5  = Непарне число
6  = Парне число
7  = Останнє непарне число, ділиться на 7
```

Інструкція **continue**, яку також запозичено з мови С, продовжує виконання циклу з наступної ітерації.

```
>>> for n in xrange(1, 10):
...     if n % 2 == 0:
...         print n, ' = Парне число'
...     elif n % 3 == 0:
...         print n, ' = Це непарне число, ділиться на 3'
...         continue
...     else:               # else належить до if
...         print n, ' = Непарне число'
... else:                   # else належить до for
...     print n, ' = Останнє число'
...
```

1 = Непарне число
2 = Парне число
3 = Це непарне число, ділиться на 3
4 = Парне число
5 = Непарне число
6 = Парне число
7 = Непарне число
8 = Парне число
9 = Це непарне число, ділиться на 3
9 = Останнє число

Остання гілка **else** належить до циклу, а не до інструкції **if**. Цикли можуть мати гілку **else**, що виконується при "нормальному" виході (вичерпання послідовності в циклі **for** або порушення умови в циклі **while**), якщо цикл не було перервано інструкцією **break**.

1.3.5. Оптимальні цикли

Інструкції керування **while**, **if**, **for** та супутні їм питання розглядалися в попередньому розділі. Тут обговоримо деякі нюанси проектування циклів, що важливі для обробки складних структур даних.

Деякі принципи оптимізації циклів [10].

- Оптимізувати лише ті фрагменти, які критичні у сенсі швидкодії. Оптимізувати тільки внутрішній цикл.
- Що менше команд, то ліпше.
- Якомога більше використовувати вбудовані операції об'єктів **map()**, **filter()** та **reduce()**. Наприклад, убудований в **map()** цикл працює швидше, ніж явний цикл **for**; а цикл **while** із явним лічильником циклу буде ще повільнішим.
- Уникати викликів функції, написаних на *Python,* у внутрішньому циклі.
- Доступ до локальних змінних відбувається швидше, ніж до глобальних; якщо використовуєте глобальні константи в циклі, скопіюйте їх у локальні змінні перед початком циклу.
- Перевіряти свої алгоритми на наявність квадратичної поведінки (коли час виконання пропорційний квадрату обсягу вхідних даних). Але майте на увазі, що використання ефективнішого, але складнішого алгоритму виправдане, зазвичай, тільки для великих N.

• Збирати й аналізувати дані про швидкодію окремих частин вашої програми. Чудовий модуль **profile** допоможе швидко виявити критичні місця у коді.

1.3.6. Порожня інструкція pass

pass означає, що не треба виконувати жодних дій. Цю інструкцію доцільно використовувати тоді, коли наявність якого-небудь коду необхідна синтаксично, але непотрібна:

```
>>> while 1:
...     pass # Нескінченний цикл: чекаємо на переривання від клавіатури
...
```

Переривання від клавіатури в цьому прикладі не показано, тому виконувати такий цикл не рекомендується, тому що ви не зможете його "нормально" зупинити чи перервати.

1.4. Стиль запису програм *Python*

У більшості мов програмування використання відступів для форматування вихідного коду є цілком косметичним прийомом, що просто підвищує його читабельність. **У *Python* відступи необхідні для того, щоб програма працювала правильно!** Наприклад, для інтерпретатора BASIC код

```
FOR I = 1 TO 10
PRINT I
NEXT I
```

інтерпретуватиметься так само, що й:

```
FOR I = 1 TO 10
PRINT I
NEXT I
```

Звісно, читати перший варіант із відступами дещо легше. Для *Python* текст, що по-різному відформатовано й інтерпретуватиметься по-різному.

Якщо відступи повинні допомагати відобразити логічну структуру коду, то запис у стилі *Python:*

```
XXXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXXX
```

XXXXXXXXXXXXXXXX

виглядає краще, ніж традиційна форма, що використовується в інших мовах програмування:

XXXXXXXXXXXXXXXXXXXX
XXXXX
XXXXXXXXXXX
XXXXXXXXXXX
XXXXXXXXXXX
XXXXX

Це відбувається через те, що із запису в стилі *Python* (див. вище) чітко видно один блок. Стиль запису програм у *Python* є особливо важливим, адже саме стиль запису з відступами забезпечує дивовижну лаконічніть та прозорість текстів програм.

Контрольні запитання

1. Назвіть типи даних, які ви знаєте і які розпізнає *Python*.
2. Опишіть три варіанти використання функції **range**.
3. Яку функцію замість **range** рекомендується застосовувати у разі великого розміру діапазону? Чому це може бути важливо?
4. Чи можуть списки містити інші списки як елементи?
5. Скільки елементів міститиме зріз **a[-1:12]**?
6. Яким чином може завершитися виконання циклу?
7. Яким чином кодуються логічні значення в мові *Python*? Чи має *Python* окремий логічний тип?
8. Які функції можна використовувати для введення й виведення даних?
9. Наведіть головні рекомендації щодо оптимізації циклів.
10. Які методи для виклику об'єктів-списків ви знаєте? Опишіть їх призначення.

Розділ 2
ФУНКЦІЇ ТА СТРУКТУРИ ДАНИХ

2.1. Визначення та документування функцій

Основна перевага використання функцій – це можливість повторного застосування програмного коду, тобто їх можна викликати багато разів не тільки в тій програмі, де її було визначено, але, можливо, і в інших програмах, іншими користувачами та для інших цілей.

2.1.1. Функціональне програмування (FP)

Функціональне програмування має такі властивості.

Функції – об'єкти першого класу [2, 3, 4]. Тобто, усе, що можна робити з "даними", можна робити і з функціями.

Рекурсія як основна структура керування.

Акцент на обробці списків **(Lisp – LISt Processing)**.

Функціональні мови унеможливлюють побічні ефекти.

FP не схвалює застосування операторів **(statements)**. Замість них – обчислення виразів (тобто функцій з аргументами).

FP наголошує на тому, ***що*** має бути обчислено, а не ***як.***

У FP переважно використовуються функції вищого порядку – функції, що оперують функціями.

2.1.2. Визначення функції

У *Python* функція визначається заголовком та блоком коду. Заголовок починається ключовим словом **def**. Далі записується ім'я функції, потім в дужках **()** ідуть параметри, які розділяються комою. До цих параметрів можна звертатися всередині функції. Утім, функція може не мати параметрів, але дужки потрібно залишити. Після визначення функції до неї можна звертатися за іменем із будь-якого місця програми, передаючи їй описані у визначенні параметри:

```
>>> def fib(n):
... """Числа Фібоначчі"""#Рядок документації функції
... a, b = 0, 1
... while b < n:
... print b,
... a, b = b, a+b
...
>>> # Тепер функцію можна викликати
... fib(2000)
Числа Фібоначчі
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Функції можуть не тільки набувати параметрів, але й повертати результат своєї роботи. Повернення значення з функції у програму, звідки функцію було викликано, здійснюється за допомогою інструкції **return**.

```
>>> def fib(n):
...     result = [1] #Цей список міститиме числа Фібоначчі
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...         result.append(b) #вставка числа в список
...     return result #повернення результату
...
>>> # Тепер функцію можна викликати
... fib(2000)
```

Результат виконання:
[1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597]
Визначення функцій має деякі особливості.

2.1.3. Параметри за замовчуванням

Згадаємо, наприклад, функцію **range()**. Її можна викликати в трьох різних формах – з одним параметром, із двома та із трьома. Для організації такої поведінки своєї функції можна описати після звичайних (позиційних) ключові параметри зі значеннями за замовчуванням:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
   '''функція ask_ok з параметрами за замовчуванням'''
   while 1:
      ok = raw_input(prompt)       #введення значення
      if ok in ('т', 'так', 'yes'): return 1
      if ok in ('н', 'ні', 'no', 'nope'): return 0
      retries = retries - 1
      if retries < 0: raise IOError, 'Помилка'
      print complaint
```

Виклик **ask_ok ("Прошу ввести:", 2)** установить перший параметр – рядок, як запрошення, а другий параметр змінить кількість неправильних спроб із чотирьох на дві. Викликаючи функцію, ключові параметри можна ставити після позиційних параметрів у довільній послідовності, якщо явно вказано імена ключових параметрів.

```
>>> ask_ok('Старт!', complaint='Так чи ні англійською,
будь ласка', retries=2)
```

Розглянемо приклад функції:

```
i = 5
def f(arg=i):
   print arg

i = 6
f()     #виведе не 6, а 5; f(10) виведе 10
```

Механізм параметрів за замовчуванням діє так: якщо змінну проініціалізовано до виклику функції, то у функцію передається саме це значення, інакше у функцію передається значення за замовчуванням.

Зауваження. Тіло функції не виконується при її визначенні, а тільки компілюється. І навпаки, значення за замовчуванням обчислюються при визначенні функції та зберігаються в об'єкті-функції.

Тобто, значення за замовчуванням установлюється лише один раз. Це відіграє роль при встановленні значення за замовчуванням спискам, *наприклад:*

```
def f(a, L=[]):
   L.append(a)
   return L
```

```
print f(1)
print f(2)
print f(3)
```

Результат роботи програми:

```
[1]
[1, 2]
[1, 2, 3]# тобто елементи накопичуються в списку
```

Для передачі параметрів за замовчуванням без нагромадження необхідно використовувати таку форму:

```
def f(a, L=None):   # None – порожній об'єкт, не вказане значення
    if L is None:   # якщо параметр L не вказано
       L = []
    L.append(a)
    return L
```

2.1.4. Передавання у функцію змінної кількості аргументів

Часто використовуваним прийомом у програмуванні є передавання у функцію змінного числа аргументів. Для цього в *Python* можна скористатися символом * перед списком аргументів змінної довжини.

Попереду списку аргументів може бути (не обов'язково) один або кілька обов'язкових аргументів:

```
def fprintf(file, format, *args):
    file.write(format % args)
```

2.1.5. Використання lambda-функцій

Для створення простої безіменної функції можна скористатися оператором **lambda**:

```
lambda param_list: expr
```

Це коротка форма створення функції, що повертає значення виразу **expr**. Її поведінка аналогічна поведінці функції, створеної за допомогою інструкції **'def name(param_list): return expr'**.

Lambda-функції прийшли в *Python* із мови Lisp та можуть здатися незвичайними програмісту, який працює з мовами C або Pascal. **Lambda**-функції – це невеликі функції, які створюють

інші функції на своїй основі. Цю технологію названо функціональним програмуванням. Приклад: **lambda a, b: a+b** – обчислює суму двох своїх аргументів. На базі функції, що повертає **lambda**, можна побудувати інші функції, *наприклад:*

```
>>> def make_incrementor(n):
#x – параметр, що передається в породжену функцію f(x)
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

2.1.6. Простір імен функції

Виконання функції вводить новий простір імен, що використовується для локальних змінних. Точніше, усі присвоювання змінним у тілі функції зберігаються в локальному просторі імен. При посиланні на змінну її пошук виконується спочатку в локальному просторі імен, потім – у глобальному і в останню чергу – у просторі вбудованих імен. Так, глобальним змінним не можна прямо присвоїти значення в тілі функції (не згадавши їх перед цим в інструкції **global**), хоча на них можна посилатися.

Аргументи функції в момент виклику містяться в локальному просторі імен функції, що викликається. Таким чином, аргументи передаються за значенням (де значення є посиланням на об'єкт, а не його значенням). Для виклику функції іншою функцією також створюється новий локальний простір імен для цього виклику.

Визначення функції вводить ім'я цієї функції в поточний простір імен. Ім'я функції має тип визначеної користувачем функції. Це значення можна присвоїти іншому імені, яке потім також використовуватиметься.

2.1.7. Документування функцій

Гарним стилем є документування кожної функції. Для цього в наступному рядку відразу після заголовка необхідно помістити короткий опис функції, укладений у потрійні **'''** апострофи

або **"""** лапки. Увесь вміст усередині потрійних лапок виводиться як є (наприклад, інструкцією **print ім'я_функції.__doc__**). Такий спосіб дозволяє легко зрозуміти призначення функції, якщо прочитати початковий текст або скористатись спеціальним сервером документації *Python*.

2.1.8. Системи одержання довідкової інформації з *Python*

Велика кількість різноманітної інформації з програмування на *Python* міститься в мережі Інтернет. Існує велика кількість онлайнових підручників та різноманітних посібників із вивчення цієї мови. До найпопулярніших можна віднести такі:

- http://ru.wikibooks.org/wiki/Python,
- http://python.su,
- http://docs.python.org/

2.1.9. Документація з мови *Python*

Після встановлення *Python* стають доступними документи формату *html* у відповідній папці, наприклад: C:\Program Files\Python26\Doc\index.html. Документи містять інформацію про методи встановлення мови, бібліотеки мови, синтаксис мови тощо. Документація надає засоби пошуку потрібної інформації.

2.1.10. Убудована допомога, модуль pydoc

Приклад 1. Одержати довідкову інформацію стосовно модуля **quopri**, використовуючи вбудовану функцію **help()**, основану на модулі **pydoc**:

```
>>> help('quopri')

Help on module quopri:
NAME
    quopri - Conversions to/from quoted-printable transport encoding as per RFC 1521.
FILE
    c:\program files\python22\lib\quopri.py
FUNCTIONS
    a2b_qp(...)
```

Decode a string of qp-encoded data

b2a_qp(...)

b2a_qp(data, quotetabs=0, istext=1, header=0) -> s;

Encode a string using quoted-printable encoding.

On encoding, when istext is set, newlines are not encoded, and white

space at end of lines is. When istext is not set, \r and \n (CR/LF) are

both encoded. When quotetabs is set, space and tabs are encoded.

І т. д., опис функцій модуля

DATA

EMPTYSTRING = ''

ESCAPE = '='

HEX = '0123456789ABCDEF'

MAXLINESIZE = 76

__all__ = ['encode', 'decode', 'encodestring', 'decodestring']

__file__ = r':\PROGRAM FILES\PYTHON22\lib\quopri.py'

__name__ = 'quopri'

Приклад 2. Одержання довідки про функцію **string.maketrans**.

```
>>> import string
>>> help(string.maketrans)

Help on built-in function maketrans in string:

maketrans(...)
maketrans(frm, to) -> string

Return a translation table (a string of 256 bytes long)
suitable for use in string.translate. The strings frm and to
must be of the same length.
```

Починаючи з *Python 2.1*, з'явився зручний у використанні модуль **pydoc** для інтерактивного доступу до вбудованої документації.

```
>>> from pydoc import help
```

Допомога може працювати у спеціальному режимі підказки:

```
>>> help
Welcome to Python 2.6!| This is the online help utility.
```

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing

Python programs and using Python modules. To quit this help utility and

return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",

"keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

help>

На запрошення **help>** можна вводити ті самі дані, що й в аргументі функції **help()**. Крім того, уведення слова **modules** дасть список установлених модулів, а **modules http** – виведе список модулів, що містять в описі слово **http**. Можна одержати список ключових слів за допомогою **keywords**, теми допомоги – за **topics**, а завершується робота із системою словом **quit**.

Якщо ваші власні пакети, модулі, класи, методи та функції мають рядки документації, **help()** видасть допомогу й по них.

2.1.11. Пошук та перегляд документації у веб-браузері

Крім використання **pydoc** як модуля, можна виконати програму **pydoc.py** (основану на цьому модулі) як скрипт із параметром, наприклад із командного рядка. Із *PythonWin* відкривають програму "C:\Program Files\Python26\Lib\pydoc.py" та вводять параметри. Рядки документації програми містять документацію з її використання:

pydoc ім'я

Показати документацію з функції, модуля, пакету, класу й т. п. з указаним ім'ям.

pydoc –k ключове_слово

Знайти модулі, в описі яких зустрічається це ключове слово.

pydoc –p порт

Стартує HTTP-сервер на вказаному порті локальної машини.

pydoc -g

Викликає графічний інтерфейс у вигляді діалогового вікна для пошуку й перегляду документації через веб-браузер.

pydoc –w ім'я

Записує документацію з указаної теми (імені) у файл *ім'я.html*, що лежить у поточному каталозі.

Приклад використання.

```
>>>import sys
>>>sys.argv.insert('-k')
>>>sys.argv.insert('math')
>>>import pydoc
>>>pydoc.cli()
```

2.2. Структури даних: рядки, списки, кортежі, словники

2.2.1. Додаткові дані про списки

Раніше вже з'ясовано, що метод **append()** дозволяє додати елемент у кінець списку:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> a.append(333)
>>> a
[66.6, 333, 333, 1, 1234.5, 333]
```

Однак іноді необхідно вставити елемент у початок або в іншу позицію списку. Це дозволяє виконати метод **insert()** – можна вказати індекс елемента, перед яким новий елемент буде додано:

```
>>> a.insert(2, -1)
[66.6, 333, -1, 333, 1, 1234.5, 333]
```

Крім того, для списків визначено методи, що дозволяють аналізувати його вміст: знайти, у якій позиції міститься (перший) елемент із певним значенням (метод **index**) або підрахувати кількість таких елементів (метод **count**):

```
>>> a.index(333)
1
>>> print a.count(333), a.count(66.6), a.count('x')
3 1 0
```

Метод **remove()** дозволяє видалити зі списку (перший) елемент, що має певне значення:

```
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
```

Елементи списку можна відсортувати (метод **sort()**) та змінити порядок розташування елементів на протилежний (метод **reverse()**):

```
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> a.reverse()
>>> a
[1234.5, 333, 333, 66.6, 1, -1]
```

Докладніше ці та інші операції над списками описано далі.

2.2.2. Додаткові можливості конструювання списків

Починаючи з версії 2.0 мови *Python*, з'явилися додаткові можливості конструювання списків без використання засобів функціонального програмування. Такі визначення списків (спискові включення) записуються зі застосуванням у квадратних дужках виразу та наступних за ним блоків **for** та **if**:

```
>>> freshfruit = ['banana', 'loganberry', 'passion fruit']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit'] # без кінцевих пробілів

>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Елементи можна фільтрувати, указавши умову ключовим словом **if**:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

Вираз, що дає в результаті кортеж, записують у круглих дужках:

```
>>> [x, x**2 for x in vec]
File "<stdin>", line 1
[x, x**2 for x in vec]
^
SyntaxError: invalid syntax

>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
```

Якщо в конструкторі вказано кілька блоків **for**, елементи другої послідовності перебираються для кожного елемента першої і т. д., тобто перебираються всі комбінації елементів:

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
```

2.2.3. Інструкція del

Існує спосіб видалити не тільки елемент за його значенням, але й елемент із певним індексом, елементи з індексами в певному діапазоні (раніше ми виконували цю операцію присвоюванням порожнього списку зрізові): інструкція **del**:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

Інструкцію **del** також використовують, щоб видаляти змінні:

```
>>> del a
```

Після цього посилання на змінну **a** генеруватиме виняткову ситуацію **NameError доти**, доки їй не буде надано інше значення. Пізніше розкажемо про інші застосування інструкції **del**.

2.2.4. Кортежі

Списки та рядки мають багато спільного. Наприклад, для них можна одержати елемент за індексом або застосувати операцію зрізу. Це два приклади послідовностей. Ще одним убудованим типом послідовностей є кортеж **(tuple).** Кортеж складається з набору значень, розділених комою, *наприклад:*

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

При виведенні кортежі завжди вкладаються в круглі дужки, для того, щоб вкладені кортежі сприймалися коректно. Уводити їх можна як у круглих дужках, так і без них, хоча в деяких випадках дужки необхідні (якщо кортеж є частиною складнішого виразу).

Кортежі мають безліч застосувань: пари координат (*x*, *y*), запис у базі даних і т. д. Для кортежів, як і для рядків, неприйнятні зміни: не можна присвоїти значення елементу кортежу (проте можна імітувати таку операцію за допомогою зрізів та наступного об'єднання).

Для конструювання порожнього елемента кортежу або кортежу, що містить один елемент, доводиться йти на синтаксичні хитрування. Порожній кортеж створюється за допомогою порожньої пари дужок, кортеж з одним елементом – за допомогою значення та наступної за ним **коми** (недостатньо просто помістити значення в дужки). *Наприклад*:

```
>>> empty = ()
... singleton = 'hello',
>>> len(empty)
0
>>> empty
()
>>> len(singleton)
```

1
>>> singleton
('hello',)

Інструкція **t = 12345, 54321, 'hello!'** – приклад пакування в кортеж. Значення 12345, 54321 та **'hello!'** пакуються разом в один кортеж. Також можлива обернена операція – розпакування кортежу:

>>> x, y, z = t

Розпакування кортежу вимагає, щоб ліворуч стояло стільки змінних, скільки елементів у кортежі. Зазначимо, що багаторазове присвоювання є лише комбінацією пакування та розпакування кортежу.

Іноді є корисним розпакування списку:

>>> a = ['spam', 'eggs', 100, 1234]
>>> a1, a2, a3, a4 = a

Як зазначалося, у кортежах, як і в рядках, не допускаються зміни. Однак, на відміну від рядків, кортежі можуть містити об'єкти, які можна змінити за допомогою методів:

>>> t = 1, ['foo', 'bar']
>>> t
(1, ['foo', 'bar'])

>>> t[1] = []
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> t[1].append('baz')
>>> t
(1, ['foo', 'bar', 'baz'])

2.2.5. Словники

Ще один убудований в *Python* тип даних – словник **(dictionary)** – часто називають асоціативним масивом. На відміну від послідовностей, що індексуються діапазоном чисел, доступ до елементів словника провадиться за ключем будь-якого типу, що не дозволяє змін, – рядки та числа завжди можуть бути ключами. Кортежі використовуться як ключі, якщо вони містять рядки, числа та кор-

тежі, що задовольняють цьому правилу. Не можна використовувати списки, тому що їх можна змінити (не створюючи нового об'єкта-списку) за допомогою, наприклад, методу **append()**.

Найліпше можна уявити словник як невпорядковану множину пар **key:value** (ключ:значення), причому кожен ключ є унікальним у межах одного словника. Фігурні дужки **{}** створюють порожній словник. Поміщаючи список пар **key:value**, розділених комами, у фігурні дужки, ви вказуєте початковий вміст словника. Такий самий вигляд матиме словник при виведенні.

Основні операції над словником – збереження з указаним ключем та виведення за ним значення. Можна також видалити пари **key:value** за допомогою інструкції **del**. Під час зберігання нового значення із ключем, що вже використовується, старе значення видаляється. При спробі прочитати значення за ключем, якого немає в словнику, генерується виняткова ситуація **KeyError**.

Метод **keys()** для словника повертає список усіх використовуваних ключів у довільному порядку (якщо необхідно, щоб список був упорядкований, застосовують до нього метод **sort()**). Щоб визначити, чи використовується певний ключ, застосовують метод **has_key()**.

Наведемо простий приклад використання словника як телефонного довідника:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127 # Доповнення словника
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']# Видалення пари зі словника
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1 # Є такий ключ
```

2.2.6. Порівняння послідовностей

Об'єкти-послідовності можна порівнювати з іншими об'єктами того самого типу. Порівняння виконується лексикографічно: спочатку порівнюються перші елементи послідовностей, і, якщо вони відрізняються, то результат їх порівняння визначає результат; якщо вони рівні, порівнюються наступні елементи і т. д. доти, поки не буде досягнуто результату або якась із послідовностей не вичерпається.

Якщо два елементи самі є послідовностями одного типу, то лексикографічне порівняння виконується рекурсивно. Якщо всі елементи двох послідовностей у результаті порівняння виявляються рівними, то послідовності вважаються рівними.

Припустимо, що одна з послідовностей є початком іншої, тоді меншою вважається послідовність із меншою кількістю елементів. Лексикографічний порядок рядків визначається порядком ASCII-символів.

Розглянемо декілька прикладів порівняння послідовностей однакового типу з результатом 1 (істина):

(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) = = (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

Зазначимо, що порівняння об'єктів різного типу припустиме. Результат буде цілком визначеним, однак не треба на нього покладатися – правила порівняння об'єктів різного типу можуть змінитися в наступних версіях *Python*. Числа порівнюються відповідно до їх значень, так, наприклад, 0 дорівнює 0.0 тощо.

2.3. Деякі бібліотечні модулі

Модулі пропонують широкий діапазон "послуг", пов'язаних з інтерпретатором *Python* та його взаємодією зі своїм оточенням. Довідкова система *Python* містить докладну інформацію про бібліотеку стандартних модулів.

Модуль **sys** забезпечує доступ до системно-залежних параметрів та функцій, а також до деяких змінних, що використовуються або підтримуються інтерпретатором, і до функцій, які інтенсивно взаємодіють з інтерпретатором. Цей модуль завжди доступний.

Модуль **string** містить загальні операції над рядками.

Модуль **bisect** вміщує реалізацію методу вставки з двійковим пошуком для того, щоб підтримувати список у відсортованому стані після кожної вставки .

Модуль **array** уможливлює використання масивів однорідних числових значень.

Модуль **os** забезпечує спільне використання функціональності, залежної від операційної системи (ОС).

Модулі **profile** та **pstats** дають можливість вимірювати продуктивність програм мовою *Python*, аналізувати результати випробувань та виводити звіти.

Контрольні запитання

1. Перелічіть основні характеристики функціонального програмування. Як підтримується FP мовою *Python*?
2. Яким чином передаються параметри у функції *Python*?
3. Для чого бажано документувати функції?
4. Якими способами можна отримати доступ до документації з мови *Python*?
5. Які є можливості генерування списків у мові *Python*?
6. Яким чином визначається кортеж, що містить один елемент?
7. Охарактеризуйте структуру даних "словник".

Розділ 3
ОРГАНІЗАЦІЯ ПРОЕКТУ ЗАСТОСУВАННЯ

3.1. Модулі та пакети

Організація та структурування проекту здійснюється за допомогою модулів, зібраних у спеціальні каталоги, які називають пакетами.

Python дозволяє помістити визначення у файл та використовувати їх у програмах та в інтерактивному режимі. Такий файл називається модулем. Визначення з модуля імпортують в інші модулі та в головний модуль (набір змінних, що є доступним у програмі та в інтерактивному режимі).

3.1.1. Створення та використання модулів

Модуль – файл, що містить визначення й інші інструкції мови *Python*. Ім'я файла утворюється шляхом додавання до імені модуля суфікса (розширення) **'.py'.** У межах модуля, його ім'я доступне через глобальну змінну **__name__**. Наприклад, використовуючи улюблений текстовий редактор, створіть у поточному каталозі файл з ім'ям **'fibo.py'** і таким вмістом:

```
''' Генерація та виведення чисел Фібоначчі '''
def fib1(n):
''' Виводить послідовність чисел Фібоначчі <= n '''
a, b = 0, 1
while b < n:
print b,
a, b = b, a+b

def fib2(n):
''' Повертає список чисел Фібоначчі <= n '''
result = []
a, b = 0, 1
while b < n:
result.append(b)
a, b = b, a+b
return result
```

Тепер запустіть інтерпретатор та імпортуйте тільки що створений модуль:

>>> import fibo

Ця інструкція не вводить імена функцій, визначених у **fibo**, безпосередньо в поточний простір імен, вона вводить тільки ім'я модуля **fibo**. Використовуючи ім'я модуля, можна одержати доступ до функцій цього модуля:

>>> fibo.fib1(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'

Якщо ви збираєтесь використовувати функцію часто, можете присвоїти її локальній змінній:

>>> fib = fibo.fib1
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

Модуль може містити будь-які інструкції для його ініціалізації, а не тільки визначення функцій. Вони виконуються, коли модуль вперше імпортується.

Можна одержати доступ до глобальних змінних модуля точно так само, як і до його функцій (насправді, функції також є змінними), **modname.itemname**.

Модулі можуть імпортувати інші модулі. Зазвичай інструкцію **import** розташовують на початку модуля або програми. Імена імпортованих модулів уміщують у поточний простір імен модуля або програми, що імпортують ці модулі.

Інший варіант інструкції **import** імпортує **імена** з модуля безпосередньо в поточний простір імен:

>>> from fibo import fib1, fib2
>>> fib1(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

У цьому разі ім'я модуля не буде представлене в поточній області видимості (у наведеному прикладі, ім'я **fibo** не визначене).

Ще один варіант інструкції **import** дає можливість імпортувати всі імена, визначені в модулі, крім імен, що починаються із символу підкреслення (**'_'**):

>>> from fibo import *
>>> fib1(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

Часто необхідно імпортувати модуль або об'єкт модуля, використовуючи для нього локальне ім'я, яке відрізняється від початкового. Наприклад, наступний код дозволяє замінити ім'я **string** на **_string** (яке не буде імпортуватися інструкцією **'from my_module import *'**) при написанні модуля:

```
import string
_string = string
del string
```

Звичайну програму можна вважати модулем з іменем **__main__** і модуль можна запускати на виконання як звичайну програму. Можна навіть виконувати деякий код (наприклад, тест) тільки у разі, якщо модуль запускається як окрема програма.

```
# Якщо файл модуля інтерпретується як головна програма:
if __name__ == "__main__":
print "Тест модуля mymodule"
print __doc__
```

3.1.2. Імпорт модулів

Перелік різноманітних варіантів інструкцій підключення модулів:

- **import список_модулів** – імпортує зазначені в списку модулі;
- **from ім'я_модуля import *** – імпортує всі імена з модуля, за винятком таких , що починаються із символу "_";
- **from ім'я_модуля import список_об'єктів** – імпортує вказані об'єкти з модуля;
- **import ім'я_модуля as name** – імпортує модуль під новим іменем **name**;
- **from ім'я_модуля import identifier as name** – імпортує з модуля об'єкт **identifier** під новим іменем **name**.

3.1.3. Пошук модулів

У процесі імпорту модуля, наприклад **spam**, інтерпретатор шукає файл із іменем **'spam.py'** у поточному каталозі, потім у каталогах, указаних у змінній оточення ***PYTHONPATH***, потім у каталогах за замовчуванням, перелік яких залежить від платформи.

Змінна **sys.path** містить список рядків з іменами каталогів, у яких здійснюється пошук модулів. Вона ініціалізується значенням змінної оточення ***PYTHONPATH*** та вбудованим значенням за замовчуванням. Можна змінити її значення, використовуючи стандартні операції зі списками. Таким чином, програми мовою *Python* можуть через них змінювати шляхи пошуку модулів під час їхнього виконання.

```
>>> import sys
>>> sys.path# виведення списку шляхів пошуку модулів
['K:/Python26', 'K:\\Python26\\Lib\\site-packages\\visual\\examples',
'K:\\Python26\\Lib\\site-packages\\vidle',  'K:\\WINDOWS\\system32
\\python26.zip', 'K:\\Python26\\DLLs', 'K:\\Python26\\lib', 'K:\\Python26
\\lib\\plat-win',  'K:\\Python26\\lib\\lib-tk',  'K:\\Python26',  'K:\\Python26
\\lib\\site-packages',  'K:\\Python26\\lib\\site-packages\\PIL',  'K:\\Python26
\\lib\\site-packages\\wx-2.8-msw-unicode']
```

3.1.4. "Скомпільовані" файли

При завантаженні модуля (**import**) або при виконанні програми спочатку інтерпретатор намагається завантажити версію цього модуля у вигляді байт-коду (файл **.pyc** або **.pyo**). Якщо це не вдається, то він автоматично компілює модуль.

Для прискорення запуску програм, що використовують велику кількість модулів, якщо вже існує файл із іменем **'spam.pyc'** у тому самому каталозі, де знайдений **'spam.py'**, вважається, що він містить "байт-скомпільований" модуль **spam**. Якщо такого файла немає, то він створюється, і час останньої зміни **'spam.py'** записується у створеному **'spam.pyc'** (при наступному використанні **'.pyc'**-файл ігнорується, якщо первинний **'.py'**-файл було змінено).

Зазвичай не треба нічого робити, щоб створити **'spam.pyc'**. Як тільки **'spam.py'** буде успішно відкомпільовано, інтерпретатор спробує записати скомпільовану версію у **'spam.pyc'**. Якщо інтерпретатору з якихось причин це не вдається (наприклад, недостатньо повноважень користувача), помилка не генерується. Якщо ж файл записано не повністю, далі він розпізнається як некоректний та ігнорується. Вміст байт-скомпільованих файлів є платформонезалежним (але може відрізнятися для різних версій інтерпретатора), таким чином, каталог із модулями може спільно використовуватися машинами з різними архітектурами.

Щоб явно відтранслювати первинний файл(и), достатньо скористатися модулем **compileall:**

```
>>> import compileall
>>> compileall.compile_dir("C:\\Python\\MyPrograms\\",force=1)
```

Компілюються всі файли з указаного каталогу. Можна також вказати конкретний файл.

3.1.5. Використання стандартних модулів

Python розповсюджується разом із бібліотекою стандартних модулів. Частина модулів убудована в інтерпретатор і реалізує операції, які не входять до ядра мови, але які вбудовано або з міркувань ефективності або для забезпечення доступу до примітивів операційної системи.

Один модуль заслуговує на особливу увагу: **sys**, який завжди доступний у програмах. Змінні **sys.ps1** та **sys.ps2** роботі під час роботи в інтерактивному режимі визначають рядки, що використовуються для первинного та вторинного запрошення інтерпретатора:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

Для того, щоб отримати перелік імен, які визначено в модулі, можна використовувати вбудовану функцію **dir()**. Вона повертає відсортований список рядків:

```
>>> import fibo, sys
>>> dir(fibo)
```

Список, що повертається функцією **dir()**, не містить імена вбудованих функцій та змінних – вони визначені в стандартному модулі **__builtin__**:

```
>>> import __builtin__
>>> dir(__builtin__)
```

3.1.6. Пакети

Модулі можна організувати в пакети (packages). Це спосіб структурування просторів імен модулів із використанням "крапкового" запису. Наприклад, ім'я модуля **A.B** визначає субмодуль з іменем **B** у пакеті **A**. Так само, як застосування модулів робить

безпечним використання глобального простору імен авторами різних модулів, так і застосування "крапкового" запису робить безпечним використання імен модулів авторами багатомодульних пакетів.

Пакет зберігається в окремому каталозі, який є у переліку шляхів для пошуку модулів. Цей каталог повинен містити файл **__init__.py**, який використовується для ініціалізації модулів пакета. Якщо пакет **(package)** складається з субпакетів **(subpackage),** то кожний субпакет повинен мати свій **__init__.py**.

В **__init__.py** пакета **package** можна помістити таке присвоювання: **__all__ = [subpackage1, subpackage2, subpackage3]**.

Цей список указує інтерпретатору на те, які модулі (або субпакети) необхідно імпортувати командою **from package import ***. Рядок **import subpackage1, subpackage2** імпортує субпакети при ініціалізації самого пакета інструкцією **import package**.

Доступ до модулів пакета отримують такими способами:

1) import package.subpackage1.module1

Завантажує модуль **package.subpackage1.module1** та робить доступним клас **cls1** через звернення **package.subpackage1. module1.cls1**;

2) from package.subpackage1 import module1

Те саме, але тепер **module1** доступний як **module1**, а клас **cls1** – як **module1.cls1**.

3) from package.subpackage1.module1 import cls1

Робить доступним клас **cls1** як **cls1**.

У згаданому вище "крапковому" записі, перші елементи повинні бути пакетами, а останній елемент може бути як пакетом, так і модулем.

3.1.7. Рекомендація щодо інсталяції пакетів

Для того, щоб *Python*-програма використовувала пакет, його має якимось чином знайти оператор імпорту. Інакше кажучи, пакет повинен бути підкаталогом каталогу, який міститься в списку **sys.path**.

Традиційно найпростішим способом для певності у тому, що пакет буде доступний через **sys.path**, було б або інсталювати його в стандартну бібліотеку або додати каталог до списку **sys.path,** використовуючи змінну оточення *$PYTHONPATH*. На практиці, обидва підходи швидко приводять до хаосу.

Виділені каталоги. У *Python 1.5* уведено домовленість, що запобігатиме хаосу, шляхом надання системному адміністратору більшого контролю. Насамперед, додано ще один каталог у кінець шляху пошуку за замовчуванням:

$prefix/lib/python1.5/site-packages

Каталог **site-packages** використовують для пакетів, які ймовірно залежать від версій *Python* (наприклад, пакет, що містить розподілені бібліотеки або використовує нові особливості).

Рекомендується розміщати кожний пакет у власний підкаталог усередині каталогу **site-packages**. Підкаталог повинен мати ім'я пакета, який буде доступний як ідентифікатор *Python*. Після цього, будь-яка *Python*-програма може імпортувати модулі з пакета, указуючи їхні повні імена. Наприклад, пакет **Sound**, міг би бути інстальований у каталог **$prefix/lib/python1.5/site-packages/Sound**, що уможливило б використання операторів імпорту таких як:

import Sound.Effects.echo.

У деяких випадках бажано інсталювати пакети не в згадані каталоги, але так, щоб вони були доступні всім *Python*-програмам, що запускаються всіма користувачами. Цього можна досягти двома різними способами.

• *Використовувати символічні зв'язки.* Помістіть символічний зв'язок на каталог верхнього рівня пакета в каталог **site-packages** або **site-python**. Ім'я символічного зв'язку має збігатися з іменем пакета; наприклад, пакет **Sound** може мати символічний зв'язок **$prefix/Lib/python1.5/site-packages/Sound**, що вказує на **/usr/home/soundguru/lib/Sound-1.1/src** (приклад наведено для Unix-систем).

• *Використовувати файли конфігурації шляху.* Якщо справді необхідно додати один або більше каталогів у **sys.path** (наприклад, якщо пакет не структуровано і він не підтримує імпорт імен, розділених крапками), то "файл конфігурації шляху" з іменем **package.pth** можна помістити в каталог **site-packages**. Кожний рядок у цьому файлі (за винятком коментарів та порожніх рядків) містить ім'я каталогу. Усі ці каталоги буде додано до **sys.path**. Дозволено використовувати відносні шляхи, базовим каталогом у цьому разі буде каталог, що містить **файл .pth**.

Файли **.pth** зчитуються в алфавітному порядку, регістр літер розрізняється або не розрізняється залежно від локальної файлової системи. Варто зазначити, що типова інсталяція не повинна мати **.pth**-файлів, а якщо вони все ж є, то їх має бути якомога менше. Проте іноді в цьому виникає необхідність.

3.2. Уведення/виведення даних

Дані виводяться як рядки. *Python* надає можливість перетворювати значення будь-якого типу в рядок за допомогою функцій **str()** та **repr()**. Функція **str()** по можливості повертає подання, найбільш придатне для виведення, а функція **repr()** – вираз, який по можливості повинен створювати цей об'єкт, якщо це подання передати функції **eval()**.

Довгі цілі числа записуються мовою *Python* із суфіксом **'L'** або **'l'**. Функція **str()** його не виводить, а **repr()** – виводить.

```
>>> repr(1000000l)
'1000000L'
>>> str(10000000l)
'10000000'
>>>
```

Можна одержати подання у вигляді рядка і для інших типів даних:

```
>>> x=[3/2.45, -2.79999, 'abcd']; p=('C','Pascal','Python')
>>> str(x); str(p)
"[1.2244897959183672, -2.7999900000000002, 'abcd']"
"('C', 'Pascal', 'Python')"
```

3.2.1. Форматоване виведення

Форматувати виведення допомагають убудовані методи рядків. Вони не виводять нічого – просто повертають новий рядок. **rjust(n)** вирівнює рядок праворуч у полі вказаної ширини **n**, доповнюючи його ліворуч пробілами. Методи **ljust(n)** та **center(n)** вирівнюють рядок по лівому краю поля й по ширині поля, відповідно.

```
>>> s='ABCDEF'
>>> for i in range(1, 6):
...     print str(i).rjust(2), str(i*i).rjust(3),
...     print str(i*i*i).rjust(4),s[0:i].center(2+i),s.lower(), s[i:6].
ljust(6)
```

```
...
1   1    1  A  abcdef BCDEF
2   4    8  AB  abcdef CDEF
3   9   27  ABC  abcdef DEF
4  16   64  ABCD  abcdef EF
5  25  125  ABCDE  abcdef F
```

Зверніть увагу на використання операцій зрізу в рядку **s**. Зазвичай описані вище методи можна застосовувати до рядка безпосередньо без участі функції **str()**.

В іншому способі форматування використовується оператор %.

'рядок формату' % (список виразів)

Рядок формату записується в стилі функції **printf()** мови С, оператор повертає рядок із результатом форматування.

```
>>> for i in range(1,11):
...  print '%2d %3d %4d %10.7f' % (i, i*i, i*i*i, i*math.pi)
...
1   1    1  3.1415927
2   4    8  6.2831853
3   9   27  9.4247780
4  16   64 12.5663706
5  25  125 15.7079633
6  36  216 18.8495559
7  49  343 21.9911486
8  64  512 25.1327412
9  81  729 28.2743339
10 100 1000 31.4159265
```

Більшість специфікаторів формату працюють точно так само, як і в С. Однак, якщо типи аргументів не відповідають формату, інтерпретатор по можливості зводить їх до необхідного типу.

3.2.2. Зчитування та запис файлів

Убудована функція **open()** повертає об'єкт-файл (**file**) та звичайно використовується із двома аргументами: **'open(filename, mode)'**.

```
>>> f=open('C:\\Tmp\\Script1.py', 'r+b')
>>> print f, id(f)
<open file 'C:\Tmp\Script1.py', mode 'r+b' at 0x01351CB0>
20257968
```

Перший аргумент – рядок, що містить ім'я файла, другий аргумент – рядок, що містить кілька символів, що описують режим використання файла.

Режим може бути **'r'**, якщо файл відкривається тільки для читання, **'w'** – тільки для запису (якщо файл існує, то його буде перезаписано) та **'a'** – для дописування в кінець файла. У режимі **'r+'** файл відкривається відразу для читання та запису. Аргумент **mode** не є обов'язковим: якщо його опустити, то використовуватиметься режим **'r'**.

У Windows (а деколи й у Macintosh) файли за замовчуванням відкриваються в текстовому режимі. Для відкривання файла у двійковому режимі необхідно до рядка режиму **mode** додати **'b'**. Варто пам'ятати, що двійкові дані, такі як зображення у форматі JPEG і навіть текст в UNICODE, при читанні з файла або запису у файл, який відкрито у текстовому режимі, будуть зіпсовані! Найліпший спосіб убезпечити себе – завжди відкривати файли у двійковому режимі, навіть на тих платформах, де двійковий режим використовується за замовчуванням.

3.2.3. Методи об'єктів-файлів

f.read(size) зчитує й повертає **size** байтів із файла. Якщо аргумент відсутній, то зчитується весь файл.

```
>>> f=open('C:\\Tmp\\Script1.py', 'r+b')
>>> f.read(127)
"import sys\r\nprint 'Good Python'\r\n\r\nfor x in
xrange(1,200,10):\r\nprint x,\r\n\r\nsys.exit()\r\n"
```

Шлях до файла у Windows містить '\', тому у відповідному рядку він записується двічі '\\'. Символи керування виводяться у вигляді послідовності **'\символ'**. Після виконання операцій покажчик поточного положення зміщується у файлі, тому його потрібно встановити в потрібну позицію перед виконанням наступної дії. Це можна зробити прямо або непрямо. Наприклад, при відкритті файлу для читання або для запису покажчик встановлюється на початок файла. Важливо після виконання роботи закривати файл. Для цього використовують метод **f.close()**.

f.readline() зчитує з файла один рядок.

f.readlines() зчитує весь вміст файла по рядку та повертає список рядків.

f.write(s) записує вміст рядка **s** у файл.

f.tell() повертає поточне положення покажчика у файлі в байтах, відраховане від початку файла.

Для зміни поточного положення треба використовувати **'f.seek(offset, from_what)'**. Нове положення обчислюється шляхом додавання **offset** до точки відліку. Точка відліку вибирається залежно від значення аргументу **from_what**: 0 відповідає початку файла (використовується за замовчуванням, якщо метод викликається з одним аргументом), 1 – поточному положенню і 2 – кінцю файла.

```
>>> f=open('C:\\Tmp\\Script1.py', 'r+b')
>>> f.readlines()
['import sys\r\n', "print 'Good Python'\r\n", '\r\n', 'for x in
xrange(1,200,10):\r\n', 'print x,\r\n', 'sys.exit()\r\n']
>>> f.write('#Comment to file Script1.py')
>>> f.tell()
116L
>>> f.close()
>>> f=open('C:\\Tmp\\Script1.py', 'r')
>>> f.tell()
0L
>>> f.read()
"import sys\nprint 'Good Python'\n\nfor x in xrange(1,200,10):\n
print x,\nsys.exit()\n#Comment to file Script1.py"
>>> f.close()
```

Зверніть увагу на різницю у виконанні методів **f.readlines()** та **f.read()**. А ось такий вигляд має файл **Script1.py** у текстовому редакторі:

```
import sys
print 'Good Python'

for x in xrange(1,200,10):
   print x,
sys.exit()
#Comment to file Script1.py
```

3.2.4. Модуль pickle

Модуль **pickle** дозволяє одержати подання майже будь-якого об'єкта (навіть деякі форми коду) у вигляді набору байтів (рядка), однакового для всіх платформ.

Якщо у вас є об'єкт **x** та файловий об'єкт **f**, відкритий для запису, то найпростіший спосіб зберегти об'єкт вимагає всього один рядок коду:

pickle.dump(x, f)

Так само просто можна відновити об'єкт (**f** – файловий об'єкт, відкритий для читання):

x = pickle.load(f)

3.3. Помилки та виняткові ситуації

Ураховуючи складність програмування, помилки, на жаль, неминучі. Помилки в синтаксисі програми виявляє інтерпретатор та допомагає їх виправити, тобто виконує в певному розумінні функції навчальної програми. Програма з такими помилками просто не виконується.

Помилки, що виникають під час виконання програми, викликають у найліпшому разі переривання виконання програми, а в найгіршому – перезавантаження ЕОМ із втратою не збережених даних. Деякі помилки викликаються неправильними діями користувача і такі дії треба передбачити, перехоплюючи виняткові ситуації, розпізнаючи й обробляючи їх у програмі. Такий механізм обробки виняткових ситуацій є в усіх сучасних програмних системах, включаючи, звичайно, і *Python*.

3.3.1. Синтаксичні помилки

Реакція *Python* на помилки в інструкціях дозволяє легко зрозуміти причину й місце їх виникнення. Це просто через те, що інтерпретатор відразу реагує на першу помилку, яка зустрілась, і показує, де і як вона виникла **(Traceback).**

```
>>> if 3>2 print math.sin(3)
...
Traceback (  File "<interactive input>", line 1
   if 3>2 print math.sin(3)
            ^
SyntaxError: invalid syntax
```

Пропущено символ **':'**, місце помилки прокоментовано **'line 1'** та позначено стрілкою **'^'**.

```
>>> if 3>2:
```

```
... print math.sin(3)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'math' is not defined
>>> if 3>2:
... print math.sin(3)
Traceback (IndentationError: expected an indented block (line 2)
>>> #Порушено правила відступів при записі блоку
>>> if 3>2:
...     import math
...     print math.sin(3)
...
0.14112000806
>>> if 3>2:
...     import math
...     print math.sin(x)
...
Traceback (most recent call last):
  File "<interactive input>", line 3, in?
NameError: name 'x' is not defined
```

Вище показано різні варіанти виявлення невідомих імен.

3.3.2. Виняткові ситуації

Виняткові ситуації є різного типу. Кожний тип виводиться в повідомленні. *Наприклад:* **ZeroDivisionError**, **NameError** та **TypeError**. Імена стандартних виняткових ситуацій є вбудованими ідентифікаторами. У додатку перераховано всі вбудовані виняткові ситуації та їхнє призначення.

3.3.3. Обробка виняткових ситуацій

Програми можуть реагувати на певні виняткові ситуації за допомогою інструкції **try: блок1 except ім'я_виняткової_ситуації: блок 2**.

Приклад, у якому користувачу видаватиметься запрошення "ввести ціле число". Якщо виконання переривається (звичайно **Ctrl-C**), то генерується виняткова ситуація **KeyboardInterrupt.**

```
>>> while 1:
...     try:
...         x = int(raw_input("Введіть ціле число: "))
...         break
...     except ValueError:
...         print "Неправильне число. Спробуйте знову..."
```

Інструкція **try** працює в такий спосіб.

Спочатку виконується гілка **try** (інструкції, що містяться між ключовими словами **try** та **except**).

Якщо не виникає виняткової ситуації, гілка **except** пропускається та виконання інструкції **try** завершується.

Якщо під час виконання гілки **try** генерується виняткова ситуація, то виконання гілки припиняється. Далі, якщо клас виняткової ситуації відповідає вказаному після ключового слова **except** класу, виконується гілка **except** і виконання інструкції **try** завершується.

Якщо виняткова ситуація не відповідає вказаному після ключового слова **except** класу, то воно передається зовнішньому блоку **try** або, якщо обробник не знайдено, виняткова ситуація вважається не перехопленою, виконання програми переривається та виводиться повідомлення про помилку.

3.3.4. Генерування виняткових ситуацій

Інструкція **raise** дозволяє програмістові примусово генерувати виняткові ситуації. *Наприклад:*

```
>>> raise NameError('HiThere')
Traceback (innermost last):
File "<stdin>", line 1
NameError: HiThere
```

Як аргумент **raise** використовується змінна цього класу. Клас указує на тип виняткової ситуації; аргумент, який передається конструктору, зазвичай описує подробиці виникнення виняткової ситуації.

Можна використовувати свої власні виняткові ситуації, створюючи нові класи. *Наприклад:*

```
>>> class MyError(Exception): pass
...
```

```
>>> try:
...     raise MyError(2*2)
... except MyError, e:  # обробка виняткової ситуації
...     print 'Виняткова ситуація MyError, e = ', e
...
Виняткова ситуація MyError, e =  4
>>> raise MyError(1)    # без обробки виняткової ситуації
Traceback (innermost last):
File "<stdin>", line 1
__main__.MyError: 1
```

Ще один варіант запису інструкції **try** – із визначенням "страхувальної" гілки **finally**, що виконуватиметься за будь-яких обставин. *Наприклад:*

```
>>> try:
...     raise KeyboardInterrupt()
... finally:
...     print 'Переривання!' ...
Переривання!
Traceback (innermost last):
File "<stdin>", line 2
KeyboardInterrupt
```

Гілка **finally** виконується незалежно від того, чи виникла виняткова ситуація під час виконання блоку **try** чи ні, навіть якщо вихід відбувається по інструкції **break** або **return**. Інструкція **try** може мати або одну, або кілька гілок **except**, або одну гілку **finally**, але не обидва види гілок одночасно.

3.3.5. Налагоджувач коду мовою *Python* у бібліотечному модулі pdb

Модуль **pdb** реалізує інтерактивний налагоджувач програм, написаних мовою *Python*. Він дозволяє встановлювати точки переривання, використовувати покрокове виконання коду, досліджувати кадри стеку, виводити вихідний код та виконувати інструкції мови *Python* у контексті будь-якого кадру стеку. Цей модуль також дозволяє проаналізувати програми або інтерактивні інструкції, виконання яких завершилося виникненням виняткової ситуації.

Контрольні запитання

1. Поясніть поняття "модуль" і "пакет"? Який між ними зв'язок?
2. Назвіть варіанти використання інструкції **import**?
3. Чому не варто використовувати інструкцію **from <модуль> import ***?
4. Яким чином здійснюється пошук модулів інтерпретатором?
5. Як можна зробити модулі видимими для інтерпретатора?
6. Яким чином можна отримати перелік імен, які визначено в певному модулі?
7. Опишіть можливості *Python* із форматованого виведення.
8. Чим функція **str()** відрізняється від функції **repr()**?
9. Опишіть наявні в *Python* функції роботи з файлами і методи файлових об'єктів.
10. Що таке виняткові ситуації і яким чином здійснюється їх обробка у *Python*?
11. Для чого використовується гілка **finally** в інструкції **try**? Чи можна її поєднувати з гілками **except**?
12. Які два класи виняткових ситуацій наявні в *Python*? Який із них рекомендується використовувати в програмах?

Розділ 4
КЛАСИ Й ВИЗНАЧЕННЯ КОНСТРУКЦІЙ МОВИ

4.1. Основні відомості про класи

Клас – це спеціальний об'єктний тип мови *Python*. Змінні цього типу називаються екземплярами класу або просто об'єктами. Інші типи даних теж є об'єктними, але клас можна визначити самостійно. Класи в *Python* реалізують усі основні концепції ООП: інкапсуляцію, спадкування, поліморфізм.

4.1.1. Області видимості та простори імен

При використанні класів важливу роль відіграє область видимості змінної. Ви можете звертатися до змінної тільки в межах блоку, де її було визначено (вперше використано). Якщо змінну визначено в межах функції, то поза функцією до неї немає доступу. Якщо функцію визначено в основному коді, то вона стає глобальною для даної програми (файла-модуля). Але якщо змінну або функцію визначено всередині модуля, то звертатися до неї безпосередньо ззовні за іменем неможливо (див. розд. 3.1).

Для звертання до змінної, яку визначено в межах модуля, поза модулем найчастіше використовується синтаксис **ім'я_модуля.ім'я_змінної**. Щоб звернутися до змінної всередині даної програми, можна скористатися модулем головної програми **__main__**. Усі змінні та методи, які визначено в різних модулях створюються та видаляються в певному порядку. Наприклад, коли ви імпортуєте модуль, то створюються всі об'єкти, які оголошуються в ньому. Убудовані функції інтерпретатора також містяться в особливому модулі **__builtin__**. Об'єкти цього модуля створюються при запуску та не знищуються ніколи. Тобто час існування таких об'єктів поширюється на весь час роботи програми.

Якщо ви оголошуєте змінну у функції, циклі, інструкції вибору, то вона доступна тільки в межах блоку, у якому її було оголошено. Якщо ви хочете оголосити глобальну змінну, то скористайтеся ключовим словом **global**. Змінні зберігаються в

пам'яті комп'ютера, а ім'я змінної – це фактично посилання на чарунку пам'яті. Тому, коли ви присвоюєте одній змінній значення іншої, то вони насправді вказують на один і той самий об'єкт. Коли змінюється одна змінна, то змінюється й інша, тому треба завжди бути обережним при виконанні таких операцій.

4.1.2. Опис класу

У *Python* надається підтримка парадигми об'єктно-орієнтованого програмування через механізм класів. Класи можуть містити в собі різноманітні елементи: змінні, константи, функції й інші класи. Типовий опис класу в *Python* має такий вигляд:

```
class ім'я_класу:
елемент_класу_1
...
елемент_класу_n
```

Оголошення класу схоже на використання ключового слова **def** для функції. Поки клас не оголошено, його використовувати не можна. Клас можна оголошувати в межах функції або структури **if**, але все-таки бажано описувати клас поза структурами керування, а ще ліпше описати всі класи на самому початку програми, оскільки це полегшує її читання. Клас після оголошення створює в програмі нову область видимості, тому все, що описано всередині класу, включається в область видимості класу й є недоступним іззовні. Зазвичай клас складається в основному з елементів-функцій (методів), вони визначаються всередині класу словом **def**. Методи класу мають деякі особливості щодо списку аргументів (ці відомості розглядатимуться далі).

До об'єктів класів у *Python* можна звертатися двома способами: посиланням на елемент (**ім'я_класу.ім'я_елемента**); через присвоєння змінній значення, що повертається конструктором, який викликається за допомогою функціонального запису (**змінна = ім'я_класу()**), *наприклад:*

```
class MyClass:
    "Простий клас"
    i = 12345
    def f(self):
        return 'Привіт, світе!'

x = MyClass()
```

Таке оголошення присвоює змінній **x** об'єкт класу **MyClass**, після нього до всіх елементів цього класу можна звертатися через дану змінну. Причому область видимості даного об'єкта збігається з областю видимості змінної **x**. Слід звернути увагу на особливий параметр **self**, що передається класу. Цей параметр є обов'язковим для методів, тому що містить посилання на клас, якому належить цей метод і таким чином дозволяє з нього звертатися до інших членів класу.

За замовчуванням створюється порожній об'єкт, але часто такий алгоритм поведінки виявляється неправильним та незручним. Тоді допомогти може метод **__init__()** класу. Подібно функції **__init__()** модуля, вона викликається при створенні об'єкта класу й виконує ініціалізацію полів (змінних) класу цього об'єкта. Приклад використання функції:

```
__init__():
def __init__(self):
   self.data = []
```

Функція **__init__()** може набувати скільки завгодно параметрів. Тоді при створенні екземпляра класу необхідно вказати всі параметри крім, звичайно, **self**:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> y = Complex(3.0, -4.5)
>>> y.r, y.i
(3.0,-4.5)
```

4.1.3. Доступ до елементів класів через посилання

Звертатися через посилання можна до будь-яких членів класу після того, як об'єкт класу був створений. Можна змінювати поля класу, можна читати з полів дані, що там зберігаються. Однак запис поля поза межами класу є поганим стилем програмування, тому що порушує принцип приховування інформації (інкапсуляцію), тоді як читати поля є абсолютно безпечним.

```
X.counter = 1
while X.counter < 10:
 X.counter = X.counter * 2
 print X.counter
del X.counter
```

Можна також читати методи класу та присвоювати їх змінним у програмі, щоб потім викликати. Присвоюючи змінній посилання на метод класу, як *наприклад:*

```
xf = x.f
while 1:
 print xf()
```

спостерігаємо таке: створюється змінна, яка вказує на область пам'яті, де розташована перша виконувана інструкція функції. Але це місце в пам'яті лежить усередині класу, тобто при виклику такої функції відбуватиметься те саме, що й при виклику **MyClass.f()**.

4.1.4. Спадкування

Клас може включати в себе всі елементи батьківського класу (або класів) та використовувати їх як свої власні.

Одиночне спадкування. Якщо в успадкованому класі перевизначаються деякі методи батьківського класу, то викликатися будуть методи, які перевизначені в цьому класі, а не батьківські.

Синтаксис опису класу, що успадковує від одного класу:

```
class ім'я_успадкованого_класу(ім'я_батьківського_класу):
 елемент_класу_1
 ...
 елемент_класу_n
```

При цьому батьківський клас може перебувати в іншій області видимості, наприклад, в іншому модулі. Тоді ім'я батьківського класу відокремлюється від імені модуля крапкою:

```
class ім'я_успадкованого_класу(ім'я_модуля.ім'я_батьківського_класу): елемент_класу_1
...
елемент_класу_n
```

При звертанні до елементів та методів батьківського класу використовується синтаксис:

ім'я_батьківського_класу.ім'я_поля або **ім'я_батьківського_класу.ім'я_методу(аргументи)**.

Множинне спадкування. Часто виникає необхідність звертатися до елементів багатьох класів одразу, тоді можна скористатися механізмом множинного спадкування. Із погляду програми, жодної різниці між одиночним та множинним спадкуванням немає. Тобто одиночне спадкування – це окремий випадок множинного спадкування. Щоб звернутися до елементів базових (батьківських) класів, використовується синтаксис, подібний до одиночного спадкування. Синтаксис оголошення класу, що успадковує від багатьох батьківських:

```
class ім'я_успадкованого_класу(ім'я_батьківського_класу1,
ім'я_батьківського_класу2, ... ім'я_батьківського_класуN):
елемент_класу_1
...
елемент_класу_n
```

При цьому батьківські класи можуть перебувати в різних областях видимості. Тоді необхідно вказати область видимості кожного класу *(див. вище)*.

4.1.5. Закриті (частково) атрибути

У *Python* поки що надається дуже обмежена підтримка закритих (**private**) елементів класу, тобто елементів, доступних лише членам цього класу.

Якщо ви оголошуєте будь-який елемент, починаючи його ім'я з подвійного підкреслення, то він автоматично стає закритим, спроба звернутися до нього поза класом викличе синтаксичну помилку, тоді як звертання через **self** працюватиме:

```
>>> class Test2:
...     __foo = 0
...     def set_foo(self, n):
...         if n > 1:
...             self.__foo = n
```

```
...     print self.__foo
...
>>> x = Test2()
>>> x.set_foo(5)
5
>>> x.__foo
Traceback (most recent call last):
File "<interactive input>", line 1, in ?
AttributeError: Test2 instance has no attribute '__foo'
```

Крім цього закритою є також змінна **__dict__**, що є в будь-якому модулі.

4.1.6. Приклад використання класу

Часто потрібно мати деяку структуру, що містить у собі поля різних типів, причому додавати поля потрібно динамічно, під час виконання програми. Тоді можна використовувати класи, які не містять жодних елементів, а потім довільно додавати будь-які поля до екземпляра класу:

```
class Employee:
   pass

john = Employee() # Створення порожньої структури

# Створюємо й заповнюємо поля структури
john.name = 'Іван Іванович'
john.dept = 'Програміст'
john.salary = 100000
```

4.2. Синтаксис та семантика конструкцій мови

4.2.1. Вирази, атоми й оператори (операції)

Поняття виразу, його елементів (атомів) та операторів (операцій) є спільними для різних мов програмування.

Проте існує набір характерних саме для *Python* операцій (таблиці 4.1–4.3).

Таблиця. 4.1. **Оператори для всіх типів послідовностей (списки, кортежі, рядки)**

Оператор	Опис
len(s)	Повертає довжину *s*
min(s),max(s)	Найменший і найбільший елементи *s* відповідно
x in s	Істина (1), якщо *s* містить у собі елемент, рівний *x*, інакше – хибність(0)
x not in s	Хибність, якщо *s* включає *x*, інакше істина
s + t	Злиття *s* та *t*
s * n , n * s	*n* копій *s*, що злиті разом (наприклад, '*' * 5 – це рядок '*****')
s[i]	*i*-й елемент *s*, де *i* рахується з 0
s[i:j]	Частина елементів *s*, починаючи з *i* до *j*–1 включно. Або *i*, або *j*, або обидва параметри можна опустити (*i* за замовчуванням дорівнює 0, *j* – довжині s).

Таблиця 4.2. **Оператори для списків (list)**

Оператори	Опис
s[i] = x	*i*-й елемент *s* замінюється на *x*
s[i:j] = t	Частина елементів *s* від *i* до *j*–1 замінюється на *t* (*t* може бути також списком)
del s[i:j]	Видаляє частину *s* (так само як і *s*[*i:j*] = [])
s.append(x)	Додає елемент *x* в кінець *s*
s.count(x)	Повертає кількість елементів *s*, рівних *x*
s.index(x)	Повертає найменший *i*, такий, що *s*[*i*]==*x*
s.insert(i,x)	Частина *s*, починаючи з *i*-го елемента, зсовується праворуч, і *s*[*i*] присвоюється *x*
s.remove(x)	Те саме, що й **del s[s.index(x)]** – видаляє перший елемент *s*, рівний *x*
s.reverse()	Записує рядок у зворотному порядку
s.sort()	Сортує список за зростанням

Таблиця 4.3. **Оператори для словників (dictionary)**

Оператори	Опис
len(a)	Кількість елементів *a*
a[k]	Елемент із ключем *k*
a[k] = x	Присвоює елементу з ключем *k* значення *x*
del a[k]	Видаляє *a*[*k*] зі словника
a.items()	Список кортежів пар (ключ, значення)
a.keys()	Список ключів *a*
a.values()	Список значень *a*
a.has_key(k)	Повертає 1, якщо *a* має ключ *k*, інакше повертає 0

Приклад. Цикл перегляду елементів словника (із модуля **os**)

Варіант 1

```
>>> import os
>>> for k, v in os.environ.items(): # Використовуються два параметри циклу:
    ... print "%s=%s" % (k, v)        # для ключа та значення
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim
```

Варіант 2

```
>>> print "\n".join(["%s=%s" % (k, v) for k, v in os.environ.items()])
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim
```

4.2.2. Прості інструкції

Прості інструкції не містять інших інструкцій. До простих інструкцій можна віднести вирази, інструкцію присвоювання, порожню інструкцію, команди імпорту модулів.

Вирази – це інструкції. *Наприклад:*

```
>>> 2**2; 'abc'+'bcdef'
4
'abcbcdef'
```

Інструкція присвоювання має вигляд: **змінна = вираз**.

У мові також використовується порожня інструкція: **pass.**

У *Python* немає команди переходу, але є багато спеціальних інструкцій, що фактично заміняють її в різних ситуаціях:

- **break** – перериває виконання вкладеного циклу;
- **continue** – виконує перехід на наступну ітерацію вкладеного циклу;
- **return [вираз]** – перериває виконання функції, повертаючи значення виразу;
- **global список_імен** – указує на те, що імена в поточному блоці посилаються на глобальні змінні.

4.2.3. Інші елементи мови та вбудовані функції

У табл. 4.4 наведено додаткові елементи та функції мови.

Таблиця 4.4. **Інші елементи мови та вбудовані функції**

Функція	Опис
=	Присвоювання
print [< c1 >[,<c2>]*[,]]	Виводить значення <*c*1>, <*c*2> у стандартний потік виведення. Ставить пробіл між аргументами. Якщо коми в кінці переліку аргументів немає, то переходить на новий рядок
Abs(x)	Повертає абсолютне значення *x*
apply(f, <аргументи>)	Викликає функцію (або метод) **f із <аргументами>**
Chr(i)	Повертає 1-символьний рядок з ASCII-кодом *i*
cmp(x, y)	Повертає від'ємне, нуль, або додатне значення, якщо, відповідно, *x* <, ==, або > ніж *y*
divmod(a, b)	Повертає кортеж (*a*/*b*, *a*%*b*), де *a*/*b* – це *a div b* (ціла частина результату ділення), *a*%*b* – це **a mod b** (остача)
eval(s)	Повертає об'єкт, визначений у *s* як рядок **(string). s** може містити будь-яку структуру мови. *s* також може бути кодовим об'єктом, наприклад: **x = 1**; **incr_x= eval("x+1")**
float (x)	Повертає дійсне значення, що рівне числу *x*
Hex(x)	Повертає рядок, що містить шістнадцяткове подання числа *x*
input(<рядок>)	Виводить <рядок>, зчитує й повертає значення зі стандартного потоку введення
Int(x)	Повертає ціле значення числа *x*
Len(s)	Повертає довжину (кількість елементів) об'єкта
long(x)	Повертає значення довгого цілого числа *x*

Закінчення табл. 4.4

Функція	Опис
max(s), min (s)	Повертають найбільший та найменший з елементів послідовності *s* (тобто *s* – рядок, список або кортеж)
Oct(x)	Повертає рядок, що містить подання числа *x*
open(<ім'я файлу>,<режим>='r')	Повертає файловий об'єкт, відкритий для читання. *<режим> = 'w'* – відкриття для запису
ord(c)	Повертає ASCII-код символу (рядки завдовжки 1 символ) *c*
pow(x, y)	Повертає значення *x* у степені *y*
range(<початок>,<кінець>, <крок>)	Повертає список цілих чисел, більших або рівних *<початок>* та менших ніж *<кінець>*, згенерованих із указаним *<кроком>*
raw_input([<текст>])	Виводить *<текст>* у стандартний потік виведення та зчитує рядок (**string**) зі стандартного потоку введення
round (x, n=0)	Повертає дійсне число *x*, округлене до *n*-го розряду після коми
str(<об'єкт>)	Повертає рядкове подання *<об'єкта>*
type(<об'єкт>)	Повертає тип об'єкта. Наприклад: **if type(x) ==type('): print 'це рядок '**
xrange(<початок>,<кінець>, <крок>)	Аналогічний **range**, але лише імітує список, не створюючи його. Використовується в циклі **for**

4.2.4. Складні інструкції: if, while, for, try

Інструкція **if:**

if <умова1>: <інструкція1>
[elif <умова2>: <інструкція2>]*
[else:<інструкція3>]

Інструкції, що містяться у квадратних дужках, можна випускати. Символ **"*"** після дужок означає, що вкладена в дужки частина може повторюватися. Якщо **<умова1> істинна, то**

виконуватиметься <інструкція1>, а гілки **elif** та **else** проігноруються. Якщо ж **<умова2>** істинна, то виконується **<інструкція2>**, а гілка **else** ігнорується. Якщо жодна з умов не є істинною, то виконується **<інструкція3>**.

Цикл **while**

while <умова>:
<інструкція1>
[else: <інструкція2>]

<інструкція1> виконуватиметься увесь час доти, поки <**умова>** істинна. У разі нормального завершенні циклу, тобто, якщо не було використано **break**, буде виконано **<інструкцію2>**.

Цикл **for**

for <змінна> in <список>:
<інструкція1>
[else: <інструкція2>]

<змінна> пробігає всі елементи **<списку>** і для кожного поточного значення **<змінної>** виконується **<інструкція1>**. У разі нормального завершення циклу, тобто, якщо не було використано **break**, буде виконано **<інструкцію2>**.

Обробка виняткових ситуацій виконується так:

try :
<інструкція1>
[except [<виняткова_ситуація> [,<змінна>]]:
<інструкція2>]
[else: <інструкція3>]

Виконується **<інструкція1>**, якщо при цьому виникла **<виняткова_ситуація>**, то виконується **<інструкція2>**. Якщо **<виняткова_ситуація>** має значення, то воно присвоюється **<змінній>**. У разі успішного завершення **<інструкції1>** виконується **<інструкція3>**.

try :
<інструкція1>
finally :
<інструкція2>

Виконується **<інструкція1>**. Якщо не виникло виняткових ситуацій, то виконується **<інструкція2>**. В іншому випадку виконується **<інструкція2>** і негайно генерується виняткова ситу-

ація. Інакше кажучи, **<інструкція2>** виконуватиметься завжди, незалежно від того, чи виникла виняткова ситуація під час виконання **<інструкції1>**.

raise <виняткова_ситуація> [<значення>]

Ініціює **<виняткову_ситуацію>** з параметром **<значення>**.

4.3. Визначення функцій і класів

4.3.1. Визначення функції

def <ім'я_функції>([<список_параметрів>]):<тіло_функції>

Тут **<тіло_функції>** – послідовність операторів, вирівняних по тексту праворуч від слова **"def". <список_параметрів>** у найзагальнішому вигляді можна подати так:

[<id>[,<id>]*][<id>=<v>[,<id>=<v>]*][,*<id>][,<id>]**

Тут **<id>** – ідентифікатор змінної; **<v>** – деяке значення. Параметри **<id>,** за якими має бути **"=",** отримують значення **<v>** за замовчуванням (**[,...]*** – дозволяє продовжити список елементів у дужках). Якщо список закінчується рядком **"*<id>"**, то **id** присвоюється кортеж (**tuple**) з усіх що залишилися (зайвих) позиційних аргументів, переданих функції. Якщо список_параметрів закінчується рядком **"**<id>"**, то формальному параметру **id** присвоюється словник з усіх не знайдених у списку параметрів іменованих аргументів (разом з їхніми значеннями), переданих функції.

Приклад з усіма різновидами параметрів:

```
def hairy(a, (b, c), d=4, *rest, **keywords):
 print "\na:", a, ", b:", b, ", c:", c, ", d:", d
 print "rest:", rest
 print "keywords:", keywords
 print "1 hairy():", hairy(1, (2, 3), 40, 5, 6, x=-3, y=-2, z=-1)
 print "2 hairy():", hairy(1, (2, 3), z=-1)
```

Результати виконання:

```
1 hairy():
a: 1 , b: 2 , c: 3 , d: 40
rest: (5, 6)
keywords: {'y': -2, 'x': -3, 'z': -1}
None
```

```
2 hairy():
a: 1 , b: 2 , c: 3 , d: 4
rest: ()
keywords: {'z': -1}
None
```

4.3.2. Оголошення класів

class <ім'я_класу> [(<батьківський_клас1>[,
<батьківський_клас2>]*)]:
<тіло_класу>

Тут **<тіло_класу>** може містити присвоювання змінним (ці змінні стають атрибутами, тобто полями класу) та визначення функцій (що є методами).

Першим аргументом методу завжди є екземпляр класу, для якого викликається цей метод. За угодою, цей аргумент називається **"self"**. Спеціальний метод **__init__()** – конструктор – викликається автоматично при створенні екземпляра класу.

Приклад:

```
class cMyClass:
   def __init__(self, val):        # конструктор
      self.value = val            # поле класу
   def printVal(self):             # метод класу
      print ' value = ', self.value
# кінець оголошення cMyClass
obj = cMyClass(3.14)
obj.printVal()
obj.value = " string now "
obj.printVal()
Результати:
value = 3.14
value = string now
```

4.3.3. Простори імен

Простір імен – спосіб прив'язати імена (ідентифікатори) до певних об'єктів, вони вказують область, де зберігається змінна (наприклад, в якому модулі). За функціональністю простори імен еквівалентні словникам та реалізуються як словники. У мові *Python* завжди наявні три простори імен: локальний, глобаль-

ний та простір убудованих імен. Пошук локальних імен завжди виконується спочатку в локальному просторі імен, глобальних – спочатку в глобальному, а потім у просторі вбудованих імен. Є ім'я локальним або глобальним визначається під час компіляції: за відсутності інструкції **global**. Ім'я, що додається де-небудь у блоці коду, є локальним у всьому блоці; усі інші імена вважаються глобальними. Інструкція **global** дозволяє змусити інтерпретатор вважати вказані імена глобальними.

Імена можна додавати (тільки в локальний простір імен) такими способами:

передаванням формальних аргументів функції;

використанням інструкції **import**;

визначенням класу або функції (додає в локальний простір імен ім'я класу або функції);

використанням оператора призначення;

вказуючи ім'я в другій позиції після ключового слова **except**.

Якщо глобальне ім'я не знайдено в глобальному просторі імен, його пошук виконується в просторі вбудованих імен, що, насправді, є простором імен модуля **__builtin__**. Цей модуль (або словник визначених у ньому імен) доступний під глобальним іменем **__builtins__**. Якщо ж ім'я не знайдено в просторі вбудованих імен, генерується виняткова ситуація **NameError**.

Убудовані функції **globals()** та **locals()** повертають словник, що містить глобальний та локальний простори імен відповідно.

4.3.4. Типи та класи в *Python*

У *Python 2.6* є два види класів – "старі" **class**, поведінка яких залишилась незмінною, і "нові", властивості яких ми зараз розглянемо. "Нові" класи – це ті, коренем ієрархії яких служить спеціальний тип **object**. "Старі" класи – ті, які не базуються (прямо або непрямо) на цьому типі.

Наслідування класів від убудованих типів. У корені ієрархії всіх убудованих типів *Python*, типів розширення та "нових" класів лежить тип **object**:

```
>>> type(2).__bases__ # атрибут з іменем базового класу
(<type 'object'>,)
>>> type("").__bases__
(<type 'object'>,)
```

Таким чином, тепер можна спадкувати класи від убудованих типів та типів розширення (надалі всі типи, які "написано на C" – як убудовані в інтерпретатор, так і типи розширення, називатимемо вбудованими типами). При цьому будь-який клас, прямо або непрямо успадкований від будь-якого вбудованого типу – "новий" клас.

Приклад:

```
# Об'єкт списку з контролем типу елементів,
# успадкований від типу list
# У "класичному" Python для цього довелося б використовувати
# клас-обгортку, таку як, наприклад, UserList
from types import ClassType
class StrictlyTypedList(list):
   """Список, що контролює тип своїх елементів"""
   def __init__(self, value_type):
      if not isinstance(value_type, (type, ClassType)):
         raise TypeError, 'неправильний тип параметра конструктора \
               StrictlyTypedList: %s' % type(value_type)
      self.value_type = value_type
      list.__init__(self) # виклик конструктора базового класу
   def insert(self, index, object):
      list.insert(self, index, self.__checktype(object))
   def append(self, object):
      list.append(self, self.__checktype(object))
   def __iadd__(self, other_list):
      # Зверніть увагу, що функція __iadd__ приймає список
      # об'єктів для додавання, тому ми повинні перевірити всі його елементи
      list.__iadd__(self, map(self.__checktype, other_list))
   extend = __iadd__
   def __checktype(self, object):
      if not isinstance(object, self.value_type):
         raise TypeError, '%s' % type(object)
      return object
```

Конструктор класу **StrictlyTypedList** приймає тип елемента списку. Зверніть увагу на те, що перевантажені методи успадкованого класу викликають методи базового (убудованого) типу, серед

яких є **__iadd__** (спеціальний метод, що реалізує оператор +=). Це демонструє таке:у *Python 2.6* убудовані типи отримали іменовані спеціальні методи, які раніше могли бути тільки в класах.

Подивимося на роботу нашого класу:

```
>>> type(StrictlyTypedList)
<type 'type'>
```

Типом "нових" класів є 'type', на відміну від "старих", тип яких – 'class'.

```
>>> lst = StrictlyTypedList(str)
>>> lst.append('Hello')
>>> lst += (',', 'world!')
>>> print lst
['Hello', ',', 'world!']
>>> lst.append(1000)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "f:/TEMP/python-960_ic", line 17, in append
File "f:/TEMP/python-960_ic", line 28, in __checktype
TypeError: <type 'int'>
```

Успадковані від убудованих типів класи можуть застосовуватися майже скрізь, де використовуються їхні базові типи, наприклад, як параметри вбудованих функцій.

Контрольні запитання

1. Охарактеризуйте правила, за якими визначаються області видимості імен в *Python*.
2. Які концепції ООП підтримує об'єктна модель *Python*?
3. Які обмеження є в моделі ООП, реалізованій в *Python*?
4. Опишіть механізми спадкування, реалізовані в мові *Python.*
5. Назвіть та опишіть оператори, що застосовуються до всіх об'єктів послідовностей.
6. Окремо перерахуйте та опишіть оператори, що застосовуються тільки до списків, кортежів та словників.
7. Перерахуйте та опишіть убудовані функції мови *Python*.
8. Які інструкції для організації циклів наявні в *Python*?
9. У якому разі виконується гілка **else** для циклу **while**?
10. Поясніть переваги використання механізму просторів імен.

Розділ 5
УБУДОВАНІ ТИПИ ДАНИХ

5.1. Убудовані базові типи даних

5.1.1. Базові логічні операції

Деякі операції підтримуються декількома типами об'єктів; зокрема, усі об'єкти можна порівнювати, перевіряти істинність значення і перетворювати в рядок (за допомогою запису **`...`**). Перетворення в рядок неявно використовується при виведенні об'єкта оператором **print**. Будь-який об'єкт можна перевірити на істинність для використання в умові оператора **if** чи **while** або як операнд логічних операцій. Перераховані нижче значення інтерпретуються як хибність:

None;

нуль будь-якого числового типу, наприклад, **0**, **0L**, **0.0**, **0j**;

будь-яка порожня послідовність, наприклад, **''**, **()**, **[]**;

будь-який порожній словник, наприклад, **{}**;

екземпляри визначених користувачем класів, якщо клас містить метод **__nonzero__()** або **__len__()** і якщо цей метод повертає 0.

Усі інші значення вважаються істиною. Отже, об'єкти переважної більшості типів будуть завжди мати логічне значення "істина".

Операції та вбудовані функції, що повертають логічний результат, завжди повертають значення 0 – хибність і 1 – істина, якщо не зазначено іншого. (***Важливий виняток***: логічні операції **"or"** та **"and"** завжди повертають один зі своїх операндів.)

Таблиця 5.1. **Логічні операції, упорядковані за зростанням пріоритету**

Операція	Результат	Примітка
x **or** *y*	Якщо ***x*** хибність, то ***y***, інакше ***x***	1
x **and** *y*	Якщо ***x*** хибність, то ***x***, інакше ***y***	1
not *x*	Якщо ***x*** хибність, то 1, інакше 0	2

Примітки. 1. Другий аргумент обробляється тільки тоді, коли це необхідно. 2. **"not"** має нижчий пріоритет, ніж небулеві операції, так, наприклад, **not a == b** інтерпретується як **not (a == b)**, а **a == not b** є синтаксичною помилкою.

Операції порівняння підтримуються для всіх об'єктів. Усі вони мають однаковий пріоритет (який вище ніж у логічних операцій). Порівняння можуть утворювати довільні ланцюжки, наприклад, **x < y <= z** еквівалентно **x < y and y <= z**, за винятком того, що **y** обчислюється лише один раз (але в обох випадках **z** не оцінюється зовсім, якщо твердження **x < y** виявляється помилковим). Табл. 5.2 підсумовує всі операції порівняння.

Таблиця 5.2. **Операції порівняння**

Операція	Значення	Примітка
<	Строго менше ніж	
<=	Менше або дорівнює	
>	Строго більше ніж	
>=	Більше або дорівнює	
==	Дорівнює	
<>	Не дорівнює	1
!=	Не дорівнює	1
Is	Тотожність об'єктів	
is not	Нетотожність об'єктів	

Примітка. Зверніть увагу на позначення <> та **!=** для однієї й тієї самої операції.

Об'єкти різних типів, за винятком числових, ніколи не рівні при порівнянні; такі об'єкти впорядковані, але в довільному порядку. Більше того, деякі типи (наприклад, файлові об'єкти) підтримують тільки "вироджене" порівняння, при якому будь-які два об'єкти цього типу не рівні. Причому такі об'єкти впорядковані довільним чином.

Екземпляри класу зазвичай не рівні при порівнянні, якщо клас не має методу **__cmp__()**.

Зауваження з реалізації: об'єкти різних типів, за винятком чисел, упорядковані за іменами своїх типів; об'єкти одного типу, які не підтримують коректне порівняння, упорядковані за своїми адресами.

Ще дві операції з однаковим синтаксичним пріоритетом, **"in"** та **"not in"**, підтримуються тільки типами послідовностей.

5.1.2. Числові типи: цілі, дійсні, комплексні

Існує чотири числових типи: *звичайне ціле*, *довге ціле*, *числа з рухомою крапкою* та *комплексні числа*. Звичайні цілі (так звані *цілі*) реалізовані з використанням **long** у C, що містить принаймні 32 біти. Довгі цілі мають необмежену розрядність. Числа з рухомою крапкою реалізовано як **double** в C. Про їхню розрядність не можна сказати нічого певного, якщо ви не знаєте, на якій машині працюєте.

Комплексні числа мають дійсну та уявну частини, які реалізовані на C як компоненти типу **double**. Для добування цих частин із комплексного числа **z** користуються **z.real** та **z.imag**.

Числа вказуються числовими літералами або можуть бути результатом убудованих функцій та операторів. Літерали цілих (включаючи шістнадцяткові та вісімкові числа) позначають звичайні цілі. Літерали цілих із суфіксом ***"L"*** або ***"l"*** дають довгі цілі. Числові літерали, що містять знак десяткової крапки або знак показника експоненти, дають числа з рухомою крапкою.

Додавання ***"j"*** або ***"J"*** до числового літерала позначає комплексне число. *Python* повністю підтримує змішану арифметику: коли оператор арифметики містить операнди різних числових типів, операнд із "меншим" типом зводиться до "старшого" типу іншого операнда. Звичайні цілі "менші", ніж довгі цілі, які "менші", ніж числа з рухомою крапкою, котрі у свою чергу "менші" від комплексних чисел. Порівняння між числами змішаних типів використовують таке саме правило. Функції **int()**, **long()**, **float()** та **complex()** можна використовувати для примусового зведення чисел до потрібного типу.

5.1.3. Арифметичні й бітові операції

Усі числові типи підтримують операції (табл. 5.3), упорядковані за зростанням пріоритету. Зазначимо, що операції в одній групі мають один і той самий пріоритет; усі числові операції мають вищий пріоритет, ніж операції порівняння.

Таблиця 5.3. **Арифметичні операції**

Операція	Результат	Примітка
x + y	Сума ***x*** і ***y***	
x – y	Різниця ***x*** і ***y***	
x * y	Добуток ***x*** і ***y***	
x / y	Ціле від ділення ***x*** на ***y***	1
x % y	Остача від ділення ***x*** на ***y***	
-x	***x*** із протилежним знаком	
+x	***x*** без зміни	
abs(x)	Абсолютне значення ***x***	
int(x)	***x***, перетворене на ціле	2
long(x)	***x***, перетворене на довге ціле	2
float(x)	***x***, перетворене на число з рухомою крапкою	
complex(re,im)	Комплексне число з дійсною частиною **re** й уявною частиною **im. im** за замовчуванням дорівнює 0	
c.conjugate()	Спряжене число комплексного числа ***c***	
divmod(x, y)	Пара **(*x* / *y*, *x* % *y*)**	
pow(x, y)	***x*** до степеня ***y***	
x ** y	***x*** до степеня ***y***	

Примітки. 1. При діленні цілих (звичайних або довгих) результат буде також цілим числом. Цей результат завжди округляється в сторону мінус нескінченності: ½ – це 0, (–1)/2 – це (–1), 1/(–2) – це (–1), а (–1)/(–2) дорівнює 0. Зверніть увагу на таке: якщо кожний з операндів є довгим цілим, то тип результату також буде довгим цілим, незалежно від його значення. 2. Перетворення чисел із рухомою крапкою в цілі (звичайні або довгі) може призводити до округлення або відкидання дробової частини як у мові C; див. функції **floor()** та **ceil()** у модулі **math**, які дозволяють явно вказувати необхідні перетворення.

Типи звичайних та довгих цілих підтримують додаткові операції, які мають сенс тільки для ланцюжків бітів. Від'ємні числа подаються в додатковому коді, що отримується з прямого коду інвертуванням усіх розрядів та додаванням одиниці (для довгих цілих відводиться досить велика кількість бітів, так, що не відбувається переповнення під час виконання операції).

Пріоритети бінарних побітових операцій нижчі, ніж у числових операцій, та вищі, ніж у порівнянь; унарна операція "~" має той самий пріоритет, що й інші унарні числові операції ("+" та "–").

Бітові операції, упорядковані за зростанням пріоритету, наведено в табл. 5.4.

Таблиця 5.4. **Бітові операції**

Операція	Результат	Примітка
x | y	Побітове **or** (АБО) між ***x*** і ***y***	
x ^ y	Побітове виключне АБО	
x & y	Побітове **and** (І) між ***x*** і ***y***	
x << n	***x***, зсув ліворуч на ***n*** бітів	1, 2
x >> n	***x***, зсув праворуч на ***n*** бітів	1, 3
~x	Інверсія всіх бітів ***x***	

Примітки. 1. Зсув на від'ємне число бітів не дозволяється та викликає помилку **ValueError**. 2. Зсув ліворуч на **n** бітів еквівалентний множенню на **pow(2, n)** без перевірки на переповнення. 3. Зсув праворуч на **n** бітів еквівалентний діленню на **pow(2, n)** без перевірки на переповнення.

5.1.4. Незмінювані послідовності: рядки, кортежі

Є три типи послідовностей: рядки, списки, кортежі.

Літерали рядків записуються в одинарних або подвійних лапках: **'xyzzy'**, **"frobozz"**. Кортежі створюються операцією "**кома**" з або без круглих дужок, але порожній кортеж повинен мати круглі дужки, наприклад, **a**, **b**, **c** або **()**. Одноелементний кортеж повинен мати завершальну кому, наприклад, **(d,)**.

Типи послідовностей підтримують ряд операцій. Операції **"in"** та **"not in"** мають той самий пріоритет, що й операції порівняння. Операції **"+"** та **"*"** мають такий самий пріоритет, що й відповідні числові операції.

Табл. 5.5 містить операції послідовностей, упорядковані за зростанням пріоритету (операції в одній групі мають однаковий пріоритет). У таблиці **s** та **t** є послідовностями одного типу; **n**, **i** та **j** – цілі.

Таблиця 5.5. **Операції над послідовностями, упорядковані за пріоритетом**

Операція	Результат	Примітка
x in s	1, якщо елемент із *s* дорівнює *x*, інакше 0	
x not in s	0, якщо елемент із *s* дорівнює *x*, інакше 1	
s + t	Конкатенація (з'єднання) *s* і *t*	
s * n, n * s	*n* конкатенованих копій *s*	1
s[i]	*i-й* елемент із *s*, починаючи з 0	2
s[i:j]	Зріз із *s* від *i* до *j*	2, 3
len(s)	Довжина *s*	
min(s)	Найменший елемент із *s*	
max(s)	Найбільший елемент із *s*	

Примітки. 1. Значення *n*, менше від 0, обробляється як 0 (що дає порожню послідовність такого самого типу як у *s*). 2. Якщо *i* або *j* від'ємне, то індекс відраховується від кінця рядка, тобто індекси заміняються на **len(s) + i** або **len(s) + j**. Але пам'ятатимемо, що "–0" – це все ж таки 0. 3. Зріз *s* від *i* до *j* визначається як послідовність елементів з індексами *k*, такими, що *i* <= *k* < *j*. Якщо *i* або *j* більше ніж **len(s)**, то береться **len(s)**. Якщо індексу *i* немає, то використовується 0. У випадку, коли *j* випущено, використовується **len(s)**. Якщо *i* більше або дорівнює *j*, то зріз буде порожнім.

Рядкові об'єкти мають одну унікальну вбудовану операцію: оператор **%** (модуль) із рядком. Лівий аргумент інтерпретується як рядок форматування із форматом, схожим на той, що використовує функція **sprintf()** мови С, і який застосовується до правого аргументу (список аргументів для форматування) та повертає рядок, отриманий у результаті цієї операції форматування.

Правий аргумент повинен бути кортежем з одним елементом для кожного аргументу, необхідного рядку форматування; якщо рядок вимагає одного аргументу, то правий аргумент може також бути єдиним некортежним об'єктом. Дозволено використовувати такі символи форматування: **%, c, s, i, d, u, o, x, X, e, E, f, g, G**. Ширину та точність можна вказувати як *****, це означає, що цілий

аргумент визначає дійсну ширину або точність. Дозволено наступні символи-прапорці: **-, +, пробіл, #** та **0**. Специфікатори розміру **h**, **l** або **L** можна вказувати, але їх буде проігноровано. Перетворення **%s** приймає будь-який об'єкт *Python* та конвертує його в рядок, використовуючи **str()**. ANSI-модифікатори **%p** та **%n** не підтримуються. Оскільки рядки *Python* мають явну довжину, перетворення **%s** не вважає, що нульовий символ (**'\0'**) є кінцем рядка, як це робиться в мові С.

Із міркувань безпеки значення специфікатора точності для чисел із рухомою крапкою обмежується 50 знаками; перетворення **%f** для чисел, що за модулем більші від **1e50**, заміняється перетворенням **%g**. Усі інші помилки ініціюють виняткові ситуації.

Якщо правий аргумент – це словник (або будь-який вигляд відображень), то формати в рядку мають містити ключі цього словника в дужках, які поміщаються відразу після символу **"%",** та кожний формат форматує відповідний елемент відображення.

Приклад:

```
>>> count = 2
>>> language = 'Python'
>>> print'%(language)s має %(count)03d типи лапок.' % vars()
Python має 002 типи лапок.
```

Убудована функція **vars()** повертає словник зі змінними в поточній області видимості.

У цьому разі формат не може містити специфікаторів ***** (оскільки вони потребують, щоб список параметрів був послідовний).

Додаткові рядкові операції визначено в стандартному модулі **string** та в убудованому модулі **re**.

5.1.5. Змінювані послідовності: списки

Списки створюються за допомогою квадратних дужок, елементи відокремлюються один від одного комою: **[a, b, c]**. Порожній список **[]** не містить елементів.

Рядкові об'єкти підтримують додаткові операції, які дозволяють внутрішню модифікацію об'єкта. Ці операції так само підтримуватимуться й іншими типами змінюваних послідовностей, які в май-

бутньому можливо з'являться в мові. Рядки та кортежі – типи незмінюваних послідовностей, такі об'єкти не можуть модифікуватися після створення. Для типів змінюваних послідовностей, де **x** – довільний об'єкт, визначено такі операції (табл. 5.6).

Таблиця 5.6. **Операції над змінюваними послідовностями**

Операція	Результат	Примітка
s[i] = x	*i-й* елемент *s* заміняється на *x*	
s[i:j] = t	Зріз *s* від *i* до *j* заміняється на *t*	
del s[i:j]	Те саме, що **s[i:j] = []**	
s.append(x)	Те саме, що **s[len(s):len(s)] = [x]**	
s.extend(x)	Те саме, що **s[len(s):len(s)] = x**	1
s.count(x)	Повертає кількість таких *i*, що **s[i] == x**	
s.index(x)	Повертає найменше *i*, таке, що **s[i] ==x**	2
s.insert(i, x)	Те саме, що **s[i:i] = [x] if i >= 0**	
s.pop([i])	Те саме, що **x = s[i]; del s[i]; return x**	3
s.remove(x)	Те саме, що **del s[s.index(x)]**	2
s.reverse()	Записує елементи *s* у зворотному порядку	4
s.sort([cmpfunc])	Сортує елементи *s*	4,5

Примітки. 1. Генерує виняткову ситуацію, якщо **x** не рядковий об'єкт. Метод **extend()** є експериментальним та не підтримується змінюваними типами послідовностей, окрім списків. 2. Генерує виняткову ситуацію **ValueError**, якщо **x** не знайдено в **s**. 3. Метод **pop()** є експериментальним та не підтримується іншими змінюваними типами послідовностей, окрім списків. Необов'язковий аргумент ***i*** за замовчуванням дорівнює –1, тобто видаляється та повертається останній елемент. 4. Методи **sort()** та **reverse()** модифікують безпосередньо список заради економії пам'яті при сортуванні або реверсуванні великих списків. Вони не повертають відсортований або реверсований список. 5. Метод **sort()** приймає необов'язковий аргумент, що визначає функцію порівняння двох аргументів (елементів списку), яка має повертати –1, 0 або 1, залежно від того, чи вважається перший елемент меншим, рівним або більшим ніж другий аргумент. Зазначимо, що це значно сповільнює процес сортування; наприклад, щоб відсортувати список у зворотному порядку, значно швидше це можна зробити, використовуючи виклики **sort()** та **reverse()**, ніж **sort()** із функцією порівняння, що обертає порядок елементів.

5.1.6. Словники (відображення)

Об'єкт *відображення* (*mapping*) відбиває значення одного типу (типу ключа) на довільні об'єкти. Відображення – змінювані об'єкти. У мові *Python* існує лише один стандартний тип відображень – *словник* (*dictionary*). Ключі словника – майже довільні значення. Єдиними типами значень, неприйнятними як ключ, є значення, що містять списки або словники, або інші змінювані типи, які порівнюються за їхніми значеннями, а не через тотожність об'єктів. Числові типи, використані для ключів, підлягають звичайним правилам порівняння чисел: якщо два числа визначаються як рівні (наприклад, 1 та 1.0), то їх можна використовувати для індексації одного й того самого елемента словника.

Словники створюються записом списку розділених комою пар **ключ: значення** у фігурних дужках, наприклад: **{'jack': 4098, 'sjoerd': 4127}** або **{4098: 'jack', 4127: 'sjoerd'}**.

Для відображень визначено такі операції, де *a* та *b* – відображення, *k* – ключ, а *v* та *x* – довільні об'єкти (табл. 5.7).

Таблиця 5.7. **Операції над відображеннями**

Операція	Результат	Примітка
len(a)	Кількість елементів в ***a***	
a[k]	Елемент ***a*** з ключем ***k***	1
a[k] = x	***a*[*k*]** присвоюється ***x***	
del a[k]	Видаляє ***a*[*k*]** з ***a***	1
a.clear()	Видаляє всі елементи з ***a***	
a.copy()	"Поверхнева" копія ***a***	
a.has_key(k)	1, якщо ***a*** має ключ ***k***, інакше 0	
a.items()	Копія списку (***ключ***, ***значення***) із ***a***	2
a.keys()	Копія списку ключів із ***a***	2
a.update(b)	**for k, v in b.items(): a[k] = v**	3
a.values()	Копія списку значень із ***a***	2
a.get(k[, x])	**a[k]**, якщо **a.has_key(k)**, інакше **x**	4

Примітки. 1. Генерує виняткову ситуацію **KeyError**, якщо **k** немає в словнику. 2. Ключі та значення перераховуються у випадковому порядку. Якщо **keys()** та **values()** викликано без проміжної модифікації

словника, то два списки повністю відповідатимуть один одному. Це дозволяє створювати пари (***значення***, ***ключ***), використовуючи **map(): "pairs = map(None, a.values(), a.keys())"**. 3. ***b*** повинен бути того самого типу, що й ***a***. 4. Ніколи не генерує виняткової ситуації, якщо **k** немає у словнику, замість цього повертає ***x*** (***x*** є необов'язковим); якщо його не вказано та **k** немає у словнику, вертається **None**.

5.2. Викликувані об'єкти, інші типи вбудованих об'єктів

Розглянемо вбудовані в інтерпретатор складні типи об'єктів. Складні об'єкти можуть містити інші об'єкти. Більшість із них підтримує небагато операцій порівняно з розглянутими раніше типами даних. До таких об'єктів належать: викликувані об'єкти, об'єкти-функції, вбудовані методи та функції та ін.

5.2.1. Викликувані об'єкти

Викликувані типи об'єктів, мають операцію виклику *ім'я_об'єкта* (*список_параметрів*). Убудована функція **callable()** повертає 1, якщо об'єкт можна викликати.

```
>>> from math import *
>>> callable(pi); callable(sin); callable(__doc__)
0
1
0
```

Якщо об'єкт не можна викликати, то результатом буде 0.

5.2.2. Функції

Об'єкти-функції створюються шляхом визначення функцій. Єдина операція, що існує для функцій, – це їхній виклик: ***функ*(*список_аргументів*)**. Існує два різновиди об'єктів-функцій: убудовані функції та функції, визначені користувачем. Обидва підтримують одну й ту саму операцію (виклик функції), але реалізовані вони по-різному, отже, є різними типами об'єктів. Реалізація додає два особливих атрибути, призначені тільки для читання: **f.func_code** – *кодовий об'єкт* фун-

кції, і **f.func_globals** – словник, що використовується як глобальний простір імен функції (подібно **module.__dict__**, де **module** – модуль, у якому визначено функцію **f**).

5.2.3. Методи

Методи – це функції, які викликаються з використанням синтаксису запису атрибутів ("крапковий" запис). Існує два різновиди методів: убудовані (такі як **append()** для списків) та методи екземплярів класів. Убудовані методи описані з тими типами, які їх підтримують. До методів екземплярів класів додається два особливих атрибути, призначених тільки для читання: **m.im_self** – об'єкт, яким оперує метод, і **m.im_func** – функція, що реалізує метод. Виклик **m(arg1, arg2, ..., argn)** повністю еквівалентний виклику **m.im_func(m.im_self, arg1, arg2, ..., argn)**.

5.2.4. Убудовані функції та методи

Убудовані функції та методи реалізовано в мові *Python* об'єктами одного типу – **builtin_function_or_method**.. Кількість та тип аргументів визначається реалізацією. Об'єкти, що реалізують убудовані функції та методи, є незмінними й не містять посилань на змінювані об'єкти.

Об'єкти **builtin_function_or_method** мають такі атрибути:

- **__doc__** – рядок документації (**None**, якщо рядок документації не визначено);
- **__name__** – ім'я функції/методу;
- **__self__** – посилання на об'єкт, до якого застосовується метод, або **None** для вбудованих функцій.

5.2.5. Кодові об'єкти

Кодові об'єкти використовуються реалізацією, щоб зберігати "псевдоскомпільований" виконуваний код *Python*, такий як тіло функції. Вони відрізняються від об'єктів-функцій тим, що не містять посилання на своє глобальне оточення виконання. Кодові об'єкти вертаються вбудованою функцією **compile()**,їх можна "витягнути" з об'єктів-функцій через їхній атрибут **func_code**.

Кодовий об'єкт можна виконати, передавши його (замість рядка вихідного коду) в оператор **exec** або вбудовану функцію **eval()**.

5.2.6. Класи та їхні екземпляри

Для виклику об'єкта-класу створюється та вертається новий екземпляр цього класу. При цьому викликається спеціальний метод класу **__init__** (якщо він визначений) з аргументами, із якими викликається об'єкт-клас. Якщо метод **__init__** не визначено, об'єкт-клас має викликатися без аргументів.

Якщо для класу визначити метод **__call__()**, то його екземпляри підтримуватимуть виклик. Отже, виклик ***екземпляра*** (**x(список_параметрів)**) перетвориться у виклик методу **__call__()** із тими самими аргументами (**x.__call__(список_параметрів)**).

5.2.7. Модулі

Єдиною особливою операцією для модуля є доступ до атрибута: **m.name**, де **m** – це модуль, а **name** дає доступ до імені, визначеного в символьній таблиці модуля **m**. Модульним атрибутам можна присвоювати значення. (Строго кажучи, оператор **import** не є операцією модульного об'єкта; **import foo** не вимагає, щоб існував модульний об'єкт із назвою **foo**, скоріше вимагається, щоб десь (зовні) було визначено модуль із назвою **foo**.)

Кожний модуль має особливий член **__dict__**. Це словник, що містить символьну таблицю модуля. Модифікація такого словника дійсно змінить символьну таблицю модуля, але пряме присвоювання атрибуту **__dict__** неможливе (тобто, можна написати **m.__dict__['a'] = 1**, що означало б визначення **m.a**, рівного 1, але не можна написати **m.__dict__ = {}**).

Убудовані в інтерпретатор модулі виводяться у такий спосіб: **<module 'sys' (built-in)>**. У разі, якщо їх було завантажено з файла, вони друкуються як **<module 'os' from '/usr/local/lib/python1.5/os.pyc'>**.

5.2.8. Класи

Об'єкт-клас створюється за допомогою визначення класу. Простір імен класу реалізовано у вигляді словника, доступного через спеціальний атрибут **__dict__**. Посилання на атрибут класу перетворяться на пошук відповідного імені у словнику, на-

приклад, **C.x** перетвориться на **C.__dict__['x']**. Якщо атрибут не знайдено, пошук здійснюється в базових класах. Пошук виконується спочатку на глибину, потім зліва праворуч – у порядку запису батьківських класів. Якщо атрибут позначає об'єкт **function**, то об'єкт перетвориться на не прив'язаний до екземпляра об'єкт-метод. Атрибут **im_class** цього об'єкта-методу вказує на клас, у якому знайдено відповідний йому об'єкт-функцію (він може бути одним із базових класів).

Присвоєння атрибуту класу вносить зміни у словник цього класу, але ніколи не змінює словники базових класів. Об'єкт-клас можна викликати, щоб одержати екземпляр класу. Об'єкти-класи мають такі атрибути:

- **__name__** – ім'я класу;
- **__module__**– ім'я модуля, в якому клас визначено;
- **__dict__** – словник атрибутів класу. Можна змінювати його вміст, тим самим додавати та видаляти атрибути класу і змінювати їхні значення. Можна навіть присвоїти **__dict__** нове значення, але це може спровокувати досить дивні помилки;
- **__bases__** – кортеж базових класів у порядку їхнього проходження у списку базових класів. Можна змінити значення цього атрибута, однак навряд чи знайдеться досить вагома причина для подібних маніпуляцій;
- **__doc__** – рядок документації класу (або **None**, якщо він не визначений).

5.2.9. Спеціальні методи класів у *Python*

Спеціальні методи класів – це дуже важливий засіб *Python*, що робить його незрівнянним за гнучкістю та простотою використання. У всіх класах користувача можна створювати власні реалізації цих методів, які будуть автоматично викликатися інтерпретатором *Python* у певних стандартних ситуаціях.

Приклад. Об'єкт, що опікується своїм поданням.

```
class a:
  def __repr__(x):
    return "a()"
  def __str__(x):
    return "a"
```

```
b = a()
print b, 'b', str(b), repr(b), "%s" % b

# a a() a a() a
```

5.2.10. Використання спеціальних методів

Більшість спеціальних методів класів вимагають передавання їм певних аргументів і в певному порядку, а у відповідь вони повертають значення певного типу.

Наприклад, методи: **__add__(self, other), __sub__(self, other), __mul__(self, other), __div__(self, other), __mod__(self, other), __divmod__(self, other), __pow__(self, other [, modulo]), __lshift__(self, other), __rshift__(self, other), __and__(self, other), __xor__(self, other), __or__(self, other)** викликаються (для лівого операнда) при виконанні бінарних операцій **+,-, *,/,%,** убудованої функції **divmod()**, операції ** та вбудованої функції **pow()**, операцій **<<, >>, &,** ^ та | відповідно.

Наприклад, для того, щоб обчислити вираз **x+y**, де **x** є екземпляром класу, що має метод **__add__**, викликається **x.__add__(y)**. Зауважимо таке, коли має підтримуватися виклик убудованої функції **pow()** із трьома аргументами, то метод **__pow__()** повинен бути визначений із третім необов'язковим аргументом.

Аналогічні спеціальні методи з префіксом **"r": __radd__(self, other), __rsub__(self, other), __rmul__(self, other)** та інші викликаються (для правого операнда) при реалізації перерахованих вище бінарних операцій, якщо правий операнд не має відповідного методу (без літери **'r'**). Наприклад, для обчислення виразу **x–y**, де **y** є екземпляром класу, що має метод **__rsub__**, викликається **y.__rsub__(x)**.

5.2.11. Екземпляри класів

Об'єкт-екземпляр класу повертається при виклику об'єкта-класу. Простір імен об'єкта-екземпляра реалізовано у вигляді словника, у якому пошук атрибутів виконується в першу чергу. Якщо атрибут не знайдено, то пошук триває серед атрибутів класу. При цьому атрибут класу, що є методом, стає прив'язаним

до цього екземпляра класу. За відсутності атрибута класу з таким іменем викликається спеціальний метод **__getattr__()**. Якщо метод **__getattr__()** не визначено, то генерується виняткова ситуація **AttributeError**.

Створення атрибута і його видалення вносять зміни в словник екземпляра класу, але ніколи не змінюють словник класу. Якщо визначено спеціальні методи **__setattr__()** та **__delattr__()**, то викликаються ці методи замість внесення змін у словник екземпляра безпосередньо.

Об'єкти-екземпляри мають такі атрибути:

- **__dict__** – словник атрибутів екземпляра класу. Можна змінювати його вміст, тим самим додавати та видаляти атрибути екземпляра та змінювати їхнє значення. Можна навіть присвоїти **__dict__** нове значення, однак це нерідко спровоковує досить дивні помилки (часто програми мовою *Python* розраховують на можливість додати атрибут до екземпляра класу, присвоювання ж **__dict__** нового значення може призвести до загадкових зникнень таких атрибутів);
- **__class__** – об'єкт-клас, екземпляром якого об'єкт є.

5.2.12. Файлові об'єкти. Файлові методи

Файлові об'єкти реалізовано з використанням пакета **stdio** мови С, їх можна створювати за допомогою вбудованої функції **open()**, описаної далі в підрозд. 5.3. Вони повертаються також деякими іншими функціями та методами.

Коли файлова операція завершується невдачею через помилку введення/виведення, виникає виняткова ситуація **IOError**. Наприклад, коли операцію не визначено з якоїсь причини, подібно до запису у файл, відкритий для читання.

Файли мають такі методи:

close(), flush(), isatty(), fileno(), read([size]), readline([size]), readlines([sizehint]), seek(offset[, whence]), tell(), truncate([size]), write(str), writelines(list).

До файлових об'єктів належать такі атрибути:

closed, mode, name, softspace.

5.2.13. Інші об'єкти

Порожній об'єкт None. Такий об'єкт повертається функцією, що явно не повертає значення. Він не підтримує ніяких особливих операцій. Існує рівно один нульовий об'єкт, який називається **None** (це – убудоване ім'я). Виводиться як **None**.

Об'єкти типу Type. Об'єкти-типи представляють різні типи об'єктів. Доступ до типу об'єкта можна отримати через вбудовану функцію **type()**. Цей тип не має яких-небудь особливих операцій. Стандартний модуль **types** визначає імена для всіх стандартних вбудованих типів.

Типи виводяться у вигляді, подібному до **<type 'int'>**.

5.2.14. Спеціальні атрибути

Реалізація додає кілька спеціальних атрибутів тільки для читання до деяких об'єктних типів, у яких вони важливі:

- **__dict__** – словник, що використовується для зберігання атрибутів (доступних для присвоювання) об'єкта;
- **__methods__** – список методів багатьох убудованих об'єктних типів, наприклад, **[].__methods__** дає **['append', 'count', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']**;
- **__members__** – подібний до **__methods__**, але повертає список атрибутів даних;
- **__class__** – клас, якому належить екземпляр класу;
- **__bases__** – кортеж батьківських класів класового об'єкта.

5.3. Убудовані функції та вбудовані класи виняткових ситуацій

5.3.1. Убудовані функції

Існує багато вбудованих в інтерпретатор *Python* і завжди доступних функцій. Деякі з них уже розглядалися. До основних функцій належать перераховані нижче за абеткою.

__import__(name[, globals[, locals[, fromlist]]]), abs(x), apply(function, args[,keywords]), buffer(object[, offset[, size]]), callable(object), chr(i), cmp(x, y), coerce(x, y), compile(string, filename, kind), complex(real[, imag]), delattr(object,name),

input([prompt]), int(x), isinstance(object, class), issubclass(class1, class2),len(s), list(sequence), locals(), long(x), map(function, list, ...), max(s[, args...]), min(s[, args...]), oct(x), open(filename[, mode[, bufsize]]), ord(c) pow(x, y[, z]), range([start,] stop[, step]), raw_input([prompt]), repr(object), round(x[, n]), setattr(object, name, value), slice([start,] stop[, step]), str(object), tuple (sequence),type(object), vars([object]), xrange([start,] stop[, step]).

5.3.2. Убудовані класи виняткових ситуацій

Виняткові ситуації можуть бути класовими або рядковими об'єктами. Хоч у традиційно виняткових ситуаціях переважали рядкові об'єкти, у *Python 1.5* усі стандартні виняткові ситуації було перероблено на класові об'єкти і користувачам рекомендується зробити те саме. Первинний код для цих виняткових ситуацій лежить у модулі **exceptions** стандартної бібліотеки; цей модуль немає потреби явно імпортувати.

Для класових виняткових ситуацій, в операторі **try** з рядком **except**, що використовує конкретний клас, цей пункт також оброблятиме будь-які успадковані від цього класу класи виняткових ситуацій (але не класи виняткових ситуацій, від яких він успадковує). Два класи виняткових ситуацій, які не пов'язані спадкуванням, ніколи не будуть еквівалентними, навіть якщо вони мають однакові імена.

Убудовані виняткові ситуації, які перераховано нижче, можуть генеруватися інтерпретатором або вбудованими функціями. За винятком того, коли це явно вказано, вони мають "асоційоване значення", що показує детальну причину помилки. Це може бути рядок або кортеж, що містить кілька елементів інформації (наприклад, код помилки та рядок, що пояснює код). Асоційоване значення передається як другий аргумент оператору **raise**. Для рядкових виняткових ситуацій асоційоване значення саме зберігатиметься у змінній, вказаній як другий аргумент **except** (за його наявності). Для класових виняткових ситуацій ця змінна одержує екземпляр виняткової ситуації. Якщо клас виняткової ситуації успадковує від стандартного базового класу **Exception**, асоційоване значення передається як атрибут **args** екземпляра виняткової ситуації, а також, можливо, і через інші атрибути.

Код користувача може генерувати вбудовані виняткові ситуації. Це використовується для перевірки обробника виняткової ситуації або для повідомлення про помилку у схожій ситуації, у якій інтерпретатор генеруватиме таку саму виняткову ситуацію; однак треба бути уважним тому, що можна згенерувати невідповідну виняткову ситуацію.

Наведемо класи виняткових ситуацій, які використовуються лише як батьківські класи для інших виняткових ситуацій:

Exception – базовий клас виняткових ситуацій. Усі вбудовані виняткові ситуації спадкують від нього. Усім визначеним користувачам виняткових ситуацій також варто успадковувати від нього, але це (ще) не категорична вимога. Функція **str()** при застосуванні до екземпляра цього класу (або до більшості класів, що спадкують від нього) повертає рядкове значення аргументу або аргументів, хоча може повернути й порожній рядок, якщо конструктору не було передано аргументів. У разі використання як послідовності, надає доступ до аргументів, які передано конструктору (зручно для зворотної сумісності зі старим кодом). Аргументи, подані як кортеж, також доступні через атрибут **args** екземпляра.

Продовження списку базових класів для виняткових ситуацій:

StandardError,
ArithmeticError,
LookupError,
EnvironmentError.

Розглянемо виняткові ситуації, що реально генеруються:

AssertionError, AttributeError, AssertionError, AttributeError, EOFError, FloatingPointError, IOError, ImportError, IndexError, KeyError, KeyboardInterrupt, MemoryError, NameError, NotImplementedError, OSError, OverflowError, RuntimeError, SyntaxError, SystemError, SystemExit, TypeError, ValueError, ZeroDivisionErro.

5.4. Бібліотечні модулі

Модулі надають широкий діапазон послуг, пов'язаних з інтерпретатором *Python* та його взаємодією зі своїм оточенням. Довідкова система *Python* містить докладну інформацію про бібліотеку стандартних модулів. Деякі модулі вже застосовувалися на практичних заняттях. Розглянемо деякі часто вживані модулі.

5.4.1. Службовий модуль sys – характерні для системи параметри та функції

Модуль **sys** забезпечує доступ до системно-залежних параметрів та функцій. Цей модуль надає доступ до деяких змінних, що використовуються або підтримуються інтерпретатором, і до функцій, які інтенсивно взаємодіють з інтерпретатором. Він завжди доступний. Склад модуля:

argv, builtin_module_names, copyright, exc_info(), exec prefix, exit(n), exitfunc, getrefcount(object), maxint, modules, path, platform, ps1, ps2, setcheckinterval(interval), setprofile(profilefunc), settrace(tracefunc), stdin, stdout, stderr, tracebacklimit, version.

5.4.2. Математичний апарат

Модулі, описані в цьому пункті, надають різний сервіс, доступний у всіх версіях *Python*:

math – математичні функції (**sin()** тощо);

cmath – математичні функції для комплексних чисел;

random – генератор псевдовипадкових чисел із різними розподілами;

array – ефективні масиви однорідних числових значень;

fileinput – perl-подібна ітерація рядків із декількох вхідних потоків із можливістю "збереження на місці";

calendar – функції, які емулюють стандартну Unix-програму **cal**;

cmd – убудований рядково-орієнтований командний інтерпретатор. Використовується модулем **pdb**;

bisect – зміст списку в сортованому порядку.

Цей модуль надає підтримку утриманню списку в сортованому порядку, не сортуючи його після кожної вставки. Для списків із великою кількістю елементів та дорогою операцією порівняння це може бути ліпшим вибором порівняно з більш загальними методами.

Методи **bisect(list, item[, lo[, hi]])** визначають місце елемента, що вставляється, **item** у списку **list**, не порушуючи порядок сортування. Параметри **lo** та **hi** можна використовувати для того,

щоб указати підмножину списку, у яку виконуватиметься вставка. Результат, що повертається, придатний для використання як перший параметр для **list.insert()**.

insert(list, item[, lo[, hi]]) Вставити **item** у **list** зі збереженням порядку. Це еквівалентно **list.insert(bisect.bisect(list, item, lo, hi), item)**.

Приклад.

Функція **bisect()** в основному корисна для категоризації числових даних. Цей приклад використовує **bisect()** для пошуку оцінки результату іспиту (у вигляді літери), сформованого на основі вибору числових діапазонів: 85 та вище – оцінка 'A', 75…84 – 'B' тощо.

```
>>> grades = "FEDCBA"
>>> breakpoints = [30, 44, 66, 75, 85]
>>> from bisect import bisect
>>> def grade(total):
...         return grades[bisect(breakpoints, total)]
...
>>> grade(66)
'C'
>>> map(grade, [33, 99, 77, 44, 12, 88])
['E', 'A', 'B', 'D', 'F', 'A']
```

5.4.3. Збереження та копіювання об'єктів

Модуль **copy** – операції поверхневого та глибокого копіювання. Цей модуль надає загальні (поверхнева та глибока) операції копіювання.

Розглянемо інтерфейс такого модуля:

```
import copy
x = copy.copy(y)# виконуємо поверхневу копію y
x = copy.deepcopy(y)# виконуємо глибоку копію y
```

Для помилок, що можуть виникнути в модулі, генерується **copy.error**.

Різниця між поверхневим та глибоким копіюванням істотна лише для складених об'єктів (таких, що містять інші об'єкти, і подібних до списків або екземплярів класів):

поверхнева копія створює новий складений об'єкт, а потім (при можливості) вставляє в нього *посилання* на об'єкти, що містяться в оригіналі;

глибока копія створює новий складений об'єкт, а потім рекурсивно вставляє в нього *копії* об'єктів, що містяться в оригіналі.

Модуль копіювання не використовує модуль реєстрації **copy_reg**.

Для визначення власної реалізації копіювання для класу можна використати спеціальні методи **__copy__()** та **__deepcopy__()**. Перший викликається для реалізації операції поверхневого копіювання; додаткові аргументи не передаються. Другий викликається для реалізації операції глибокого копіювання; йому передається один аргумент, словник **memo**. Якщо реалізація **__deepcopy__()** має потребу у створенні глибокої копії компонента, то він повинен викликати функцію **deepcopy()** із компонентом (як перший аргумент) та словником **memo** (як другий).

5.4.4. Перетворення об'єктів у послідовну форму

1. Модуль **pickle** – перетворення об'єктів *Python* у послідовну форму.

Модуль **pickle** реалізує базовий, але потужний алгоритм для "консервування (*pickling*)" – перетворення в послідовну форму (*serializing*), вибудовування (*marshalling*) або вирівнювання (*flattening*) – майже довільних об'єктів *Python*. Він полягає в перетворенні об'єктів у потік байтів (і обернена операція – "розконсервування (*unpickling*)").

2. Модуль **cPickle** – альтернативна реалізація **pickle**.

Модуль **cPickle** надає подібний інтерфейс та ідентичну функціональність, що й модуль **pickle**, але може виявитися значно (майже у 1000 разів швидшим, оскільки він реалізований на C). Ще одна важлива відмінність варта уваги полягає в тому, що **Pickler()** та **Unpickler()** є функціями, а не класами і не можуть наслідуватися.

Формат консервованих даних ідентичний отриманим із використанням модуля **pickle**, так що, **pickle** та **cPickle** є взаємозамінними.

3. Модуль **copy_reg** – реєстрація функцій підтримки **pickle**.

Модуль **copy_reg** надає підтримку для модулів **pickle** та **cPickle**.

Модуль **copy** ймовірно теж використовуватиме модуль **pickle** у майбутньому. Він надає конфігураційну інформацію щодо конструкторів об'єктів, що не є класами. Такі конструктори можуть бути фабричними функціями або екземплярами класів.

4. **shelve** – збереження об'єктів *Python.*

"Shelf (поличка)" – сталий, словниковоподібний об'єкт. Відмінність від баз даних **"dbm"** полягає в тому, що значення (не ключі!) у shelf можуть бути по суті довільними об'єктами *Python*, які може обробляти модуль **pickle**. Сюди належить більшість екземплярів класів, рекурсивні типи даних та об'єкти, що містять багато підоб'єктів. Ключі – звичайні рядки.

Приклад використання інтерфейсу (**key** – рядок, **data** – довільний об'єкт):

```
import shelve
d = shelve.open(filename) # відкриваємо файл, filename - без суфікса
d[key] = data   # зберігаємо дані з ключем key (затирає старі дані,
                # що використовують наявний ключ)
data = d[key]   # звертаємося до даних із ключем key (виникає ситуація KeyError, якщо
                # такий ключ не існує)
del d[key]      # видаляємо дані, збережені з ключем key
                # (виникає KeyError, якщо такий ключ не існує)
flag = d.has_key(key)   # істина, якщо ключ key існує
list = d.keys() # список усіх наявних ключів (повільно!)
d.close()       # закриваємо файл.
```

5. **marshal** – перетворення об'єктів *Python* у послідовну форму (із різними обмеженнями).

Цей модуль містить функції, які можуть читати та записувати величини *Python* у бінарному форматі. Такий формат специфічний для *Python*, але не залежить від архітектури машини (наприклад, ви можете записати величину *Python* у файл на комп'ютері, перенести файл на Sun та прочитати там). Деталі формату навмисно не документовано; вони можуть змінюватися від версії до версії *Python* (хоча це відбувається рідко).

Контрольні запитання

1. Як *Python* визначає логічне значення для об'єкта?
2. Перерахуйте основні логічні операції.
3. Який порядок обчислення виразів у операторі **and**?
4. Чи можна порівнювати об'єкти довільних типів?
5. Перерахуйте числові типи, які містить *Python*, і опишіть область використання кожного з них.
6. Наведіть та опишіть арифметичні операції.
7. Для чого можна використовувати бітові операції?
8. Перерахуйте незмінювані послідовності, які є в мові *Python*.
9. Які об'єкти можна використовувати як ключі до словника?
10. Перерахуйте й опишіть операції над змінюваними послідовностями. Які виняткові ситуації можуть виникати при їх використанні?
11. Опишіть викликувані об'єкти мови *Python*.
12. Поясніть переваги використання спеціальних методів класів у мові *Python*.
13. Дайте огляд бібліотечних модулів мови *Pyhton*.
14. Для чого може використовуватись переведення об'єктів у послідовну форму (серіалізація)?

Розділ 6
ГРАФІЧНИЙ ІНТЕРФЕЙС КОРИСТУВАЧА

6.1. Огляд графічних бібліотек

Будувати **графічний інтерфейс користувача G**raphical **U**ser **I**nterface (GUI) для програм мовою *Python* можна за допомогою відповідних **бібліотек компонентів графічного інтерфейсу** або, використовуючи кальку з англійської, **бібліотек віджетів**.

Наведений список бібліотек далеко не повний, але відбиває розмаїття можливих рішень:

Tkinter – багатоплатформний пакет, має добрі засоби керування розташуванням компонентів. Інтерфейс має однаковий вигляд на різних платформах (Unix, Windows, Macintosh). Входить у стандартну поставку *Python*. Як документацію можна використовувати посібник "An Introduction to *Tkinter*" ("Вступ до *Tkinter*"), написаний Фредріком Лундом. Сайт: http://www.pythonware.com/library/tkinter/introduction/.

wxPython – побудований на багатоплатформній бібліотеці *wxWidgets* (раніше називалася *wxWindows*). На кожній платформі має стандартний для неї вигляд, активно вдосконалюється, реалізовано підтримку GL. Існує для всіх основних платформ. Можливо, посяде місце *Tkinter* у майбутніх версіях *Python.* Сайт: http://www.wxpython.org/.

PyGTK – набір візуальних компонентів для GTK+ та *Gnome*. Тільки для платформи GTK.

PyQT/PyKDE – добрі пакети для тих, хто використовує *Qt* (під UNIX або Windows) або KDE.

Pythonwin – побудований навколо MFC, поставляється разом з оболонкою в пакеті *win32all*; використовується тільки для Windows.

pyFLTK – аналог *Xforms*, підтримка *OpenGL*. Є для платформ Windows та Unix. Сайт: http://pyfltk.sourceforge.net/.

AWT, JFC, Swing – поставляється разом з *Jython*, а для *Jython* доступні засоби, які використовує *Java*. Підтримує платформу *Java*.

anygui – незалежний від платформи пакет для побудови графічного інтерфейсу для програм мовою *Python*. Сайт: http://anygui.sourceforge.net/.

PythonCard – конструктор графічного інтерфейсу, подібний за ідеологією з *HyperCard/MetaCard*. Розроблений на базі *wxPython*. Сайт: http://pythoncard.sourceforge.net/.

Список актуальних посилань на різні графічні бібліотеки, які доступні з *Python*, можна знайти за адресою:

http://phaseit.net/claird/comp.lang.python/python_GUI.html

Бібліотеки можуть бути багаторівневими. *Наприклад, PythonCard* використовує *wxPython*, що, скажімо, на платформі *Linux* базується на багатоплатформній GUI-бібліотеці *wxWindows*, що, у свою чергу, базується на GTK+ або на *Motif*, а ті, у свою чергу, використовують для виведення *X Window*. Зазначимо, що для *Motif* у *Python* є свої прив'язки.

У цьому посібнику буде розглянуто пакет *Tkinter*, що по суті є обгорткою для *Tcl/Tk* – відомого графічного пакета для мови сценаріїв *Tcl*. Також наводитиметься пакет *PyQt*. На прикладі цих пакетів легко вивчити основні принципи побудови графічного інтерфейсу користувача.

Про графічний інтерфейс. Майже всі сучасні графічні інтерфейси загального призначення будуються за моделлю WIMP – *Window, Icon, Menu, Pointer* (вікно, іконка, меню, покажчик). Усередині вікон рисуються **елементи графічного інтерфейсу**, які для стислості називатимемо **віджетами** (widget – штука). Меню можуть розташовуватися в різних частинах вікна, але їхня поведінка досить однотипна: вони використовуються для вибору дії з визначеного набору. Користувач графічного інтерфейсу "пояснює" комп'ютерній програмі необхідні дії за допомогою покажчика. Звичайно покажчиком служить курсор миші або джойстика, однак є й інші "вказівні" пристрої. За допомогою іконок графічний інтерфейс здобуває незалежність від мови й у деяких випадках дозволяє швидше орієнтуватися в інтерфейсі.

Основним завданням графічного інтерфейсу є спрощення комунікації між користувачем та комп'ютером. Про це варто завжди пам'ятати при проектуванні інтерфейсу. Використання наявних у програміста (або дизайнера) засобів при створенні графічного інтерфейсу потрібно звести до мінімуму, обираючи найзручніші для користувача віджети в кожному конкретному випадку. Крім того, корисно додержуватися принципу "най-

меншого подиву": із форми інтерфейсу має бути зрозумілою його поведінка. Погано продуманий інтерфейс псує враження користувача від програми, навіть якщо за фасадом інтерфейсу ховається ефективний алгоритм.

Інтерфейс повинен бути зручний для типових дій користувача. Для багатьох програм такі дії виділені в окремі серії екранів, які називаються "майстрами" (*wizards*). Однак якщо застосування – скоріше конструктор, за допомогою якого користувач може будувати потрібні йому рішення, типовою дією є саме побудова рішення. Визначити типові дії не завжди легко, тому компромісом може бути гібрид, у якому є "майстри" та добрі можливості для власних побудов. Проте, графічний інтерфейс не є найефективнішим інтерфейсом у всіх випадках. Для багатьох предметних областей рішення простіше подати як набір декларацій деякою формальною мовою або алгоритм мовою сценаріїв.

6.2. Основи *Tk*

Основна риса будь-якої програми з графічним інтерфейсом – ***інтерактивність***. Програма не просто щось виконує (у пакетному режимі) від початку свого запуску до кінця: її дії залежать від втручання користувача. Фактично, графічне застосування виконує нескінченний цикл обробки подій. Програма, що реалізує графічний інтерфейс, **подійно-орієнтована**. Вона чекає на зміни від інтерфейсу подій, які й обробляє згідно зі своїм внутрішнім станом. Ці події виникають в елементах графічного інтерфейсу (віджетах) та обробляються прикріпленими до цих віджетів обробниками. Самі віджети мають численні властивості (колір, розмір, розташування), вишиковуються в ієрархію належності (один віджет може бути хазяїном іншого), а також мають методи для доступу до свого стану.

За розташування віджетів (у межах інших віджетів) відповідають так звані **менеджери розташування**. Віджет установлюється на місце за правилами менеджера розташування. Ці правила можуть визначати не тільки координати віджета, а і його розміри. У *Tk* є три типи менеджерів розташування: простий пакувальник (pack), сітка (grid) та довільне розташування (place).

Але цього для роботи графічної програми недостатньо. Зазначимо, що деякі віджети в графічній програмі мають бути певним чином взаємозалежними. Наприклад, смужка прокручу-

вання може залежати від текстового віджета: при використанні смужки текст у віджеті повинен рухатися, і навпаки, при переміщенні по тексту смужка має показувати поточне положення. Для зв'язку між віджетами в *Tk* використовуються змінні, через які віджети й передають один одному параметри.

6.2.1. Класи віджетів

Для побудови графічного інтерфейсу в бібліотеці *Tk* відібрано такі класи віджетів (за абеткою):

button (кнопка) – звичайна кнопка для виклику деяких дій (виконання певної команди);

canvas (малюнок) – основа для виведення графічних примітивів;

checkbutton (прапорець) – кнопка, що перемикається між двома станами при натисканні на неї;

entry (поле введення) – горизонтальне поле, у яке можна ввести рядок тексту;

frame (рамка) – віджет, що містить у собі інші візуальні компоненти;

label (напис) – віджет може показувати текст або графічне зображення;

listbox (список) – прямокутна рамка зі списком, із якого користувач може виділити один або кілька елементів;

menu (меню) – елемент, за допомогою якого можна створювати висхідне (popup) та низхідне (pulldown) меню;

menubutton (кнопка-меню) – кнопка з висхідним меню;

message (повідомлення) – аналогічно напису, але дозволяє загортати довгі рядки й міняти розмір на вимогу менеджера розташування;

radiobutton (селекторна кнопка) – кнопка для представлення одного з альтернативних значень. Такі кнопки зазвичай діють у групі. При натисканні на одну з них кнопка групи, обрана раніше, "відскакує";

scale (шкала) – служить, щоб указувати числове значення переміщенням повзунка в певному діапазоні;

scrollbar (смуга прокручування) – смуга прокручування використовується для відображення величини прокручування в інших віджетах. Може бути як вертикальною, так і горизонтальною;

text (форматований текст) – цей прямокутний віджет дозволяє редагувати й форматувати текст із використанням різних стилів, включати в текст рисунки й навіть вікна;

toplevel (вікно верхнього рівня) – відображається як окреме вікно та містить інші віджети.

Усі ці класи не мають стосунку до спадкування – вони рівноправні. У більшості випадків цей набір є достатнім, щоб побудувати інтерфейс користувача.

6.2.2. Події

У системі сучасного графічного інтерфейсу є можливість відслідковувати різні події, пов'язані з клавіатурою та мишею, які відбуваються на "території" того або іншого віджета. У *Tk* події описуються текстовим рядком – шаблоном події, що складається з трьох елементів (модифікаторів, типу події (табл. 6.1) та деталізації події).

Таблиця 6.1. **Типи подій**

Тип події	Зміст події
Activate	Активізація вікна
ButtonPress	Натискання кнопки миші
ButtonRelease	Звільнення кнопки миші
Deactivate	Деактивізація вікна
Destroy	Закриття вікна
Enter	Входження курсору в межі віджета
FocusIn	Одержання фокуса вікном
FocusOut	Втрата фокуса вікном
KeyPress	Натискання клавіші на клавіатурі
KeyRelease	Відпускання клавіші на клавіатурі
Leave	Вихід курсору за межі віджета
Motion	Рух миші в межах віджета
MouseWheel	Прокручування коліщати миші
Reparent	Зміна батька вікна
Visibility	Зміна видимості вікна

Приклади описів подій рядками та деякі назви клавіш:

"<ButtonPress-3>" або просто "<3>" – натискання правою кнопкою миші (тобто, третьою, якщо рахувати на трикнопочній миші зліва праворуч);

"<Shift-Double-Button-1>" – подвійне натискання (лівою кнопкою миші) із натиснутою клавішею **Shift**. Як модифікатори використовують такі (список неповний):

Control, Shift, Lock,
Button1-Button5 або B1-B5,
Meta, Alt, Double, Triple.

Тут символ позначає подію – натискання клавіші. Наприклад, "k" – те саме, що й ″<KeyPress-k>″. Для неалфавітно-цифрових клавіш є спеціальні назви:

Cancel, BackSpace, Tab, Return, Shift_L, Control_L,
Alt_L,Pause, Caps_Lock, Escape, Prior, Next, End, Home, Left,
Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7,
F8, F9, F10, F11, F12, Num_Lock, Scroll_Lock, space, less

Тут **<space>** позначає пробіл, а **<less>** – знак менше. **<Left>, <Right>, <Up>, <Down>** – стрілки. **<Prior>, <Next>** – це **PageUp** та **PageDown**. Інші кнопки більш-менш відповідають написам на стандартній клавіатурі.

Примітка. **Shift_L**, на відміну від **Shift**, не можна використовувати як модифікатор.

У конкретному середовищі комбінації, що означають щось особливе в системі, можуть не дістатися до графічного застосування. Наприклад, відомий всім **Ctrl-Alt-Del**.

Наступна програма дозволяє друкувати інформацію про направлені до віджета події, зокрема – **keysym**, а також аналізувати, як різні клавіші можна представити в шаблоні події (лістинг 6.1).

Лістинг 6.1 Приклад "Події"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 1

from Tkinter import *
tk = Tk()       # головне вікно застосування
```

```
txt = Text(tk)   # текстовий віджет, що належить вікну tk
txt.pack()      # розташовується менеджером pack

# функція обробки події
def event_info(event):
  txt.delete("1.0", END)   # видаляється з початку до кінця тексту
  for k in dir(event):     # цикл за атрибутами події
    if k[0] != "_":        # беруться тільки неслужбові атрибути
      # готується опис атрибута події
      ev = "%15s: %s\n" % (k, repr(getattr(event, k)))
      txt.insert(END, ev)  # додається в кінець тексту

# прив'язується до віджету txt функція event_info для обробки подій,
# що відповідають шаблону <KeyPress>
txt.bind("<KeyPress>", event_info)
tk.mainloop()   # головний цикл обробки подій
```

При натисканні клавіші **Esc** у вікні бачимо приблизно таке:

```
char: '\x1b'
 delta: 9
height: 0
 keycode: 9
keysym: 'Escape'
keysym_num: 65307
 num: 9
send_event: False
serial: 159
 state: 0
time: -1072960858
type: '2'
widget: <Tkinter.Text instance at 0x401e268c>
 width: 0
 x: 83
x_root: 448
 y: 44
y_root: 306
```

Варто пояснити деякі з цих атрибутів:

- **char** – символ, що відповідає натиснутій клавіші (для деяких подій);
- **height, width** – висота та ширина;
- **focus** – чи був у момент події фокус у вікна;
- **keycode** – код символу (скан-код клавіатури);
- **keysym** – символічне ім'я клавіші;
- **serial** – серійний номер події (збільшується у міру виникнення подій);
- **time** – час виникнення події (увесь час збільшується);
- **widget** – віджет, у якому виникла подія;
- **x, y** – координати покажчика у віджеті під час події;
- **x_root, y_root** – координати покажчика на екрані під час події.

Зовсім необов'язково, щоб події обробляв той самий віджет, що їх первинно прийняв. Наприклад, можна перенаправляти всі події всередині підпорядкованих віджетів на даний віджет за допомогою методу **grab_set()** (**grab_release()** звільняє віджет від цього обов'язку). У *Tk* існують й інші можливості керування подіями, із якими можна ознайомитись у відповідній документації.

6.2.3. Створення та конфігурування віджета

Створення віджета відбувається за допомогою виклику конструктора відповідного класу. Виклик конструктора має такий синтаксис:

Widget([master[, option=value, ...]])

Тут **Widget** – клас віджета, **master** – віджет-хазяїн, **option** та **value** – конфігураційна опція та її значення (таких пар може бути декілька).

Кожний віджет має властивості, які можна встановлювати (конфігурувати) за допомогою методів **config()** (або **configure()**) та читати, використовуючи методи, подібні до методів роботи зі словниками.

Наведемо можливий синтаксис для роботи із властивостями:

```
widget.config(option=value, ...)
widget["option"] = value
value = widget["option"]
widget.keys()
```

У разі, коли ім'я властивості збігається з ключовим словом мови *Python*, прийнято використовувати після імені одиночне підкреслення. Так, властивість **class** потрібно вказувати як **class_**, а **to** як **to_**.

Змінювати конфігурацію віджета можна в будь-який момент. Ця зміна з'явиться на екрані після повернення в цикл обробки подій або за явним викликом **update_idletasks()**.

У наступному прикладі (лістинг 6.2) показано вікно з двома віджетами – полем уведення та написом. За допомогою змінної напис прямо пов'язаний із полем уведення. У цьому прикладі навмисно використано дуже багато властивостей, щоб продемонструвати можливості з конфігурації.

Лістинг 6.2. Приклад "Два віджети"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 2

from Tkinter import *
tk = Tk()
tv = StringVar()
Label(tk,
    textvariable=tv,
    relief="groove",
    borderwidth=3,
    font=("Courier", 20, "bold"),
    justify=LEFT,
    width=50,
    padx=10,
    pady=20,
    takefocus=False,
    ).pack()
Entry(tk,
    textvariable=tv,
    takefocus=True,
    ).pack()
tv.set("123")
tk.mainloop()
```

Віджети конфігуруються прямо при створенні. Більш того, віджети не зв'язуються з іменами, їх тільки розташовують усередині віджета-вікна. У цьому прикладі використано властивості **textvariable** (текстова змінна), **relief** (рельєф), **borderwidth** (ширина границі), **justify** (вирівнювання), **width** (ширина, у знакомісцях), **padx** та **pady** (проміжок у пікселях між вмістом та границями віджета), **takefocus** (можливість прийняти фокус при натисканні клавіші **Tab**), **font** (шрифт, один зі способів його визначення). Ці властивості досить типові для багатьох віджетів, хоч іноді одиниці вимірювання можуть відрізнятися, наприклад, для віджета **Canvas** ширина задається в пікселях, а не в знакомісцях.

У наступному прикладі (лістинг 6.3) продемонстровано можливості з визначення кольорів фону, переднього плану (тексту), виділення віджета (підсвічування межі) в активному стані та за відсутності фокуса.

Лістинг 6.3. Приклад "Підсвічування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 3

from Tkinter import *
tk = Tk()
tv = StringVar()
Entry(tk,
    textvariable=tv,
    takefocus=True,
    borderwidth=10,
    ).pack()
mycolor1 = "#%02X%02X%02X" % (200, 200, 20)
Entry(tk,
    textvariable=tv,
    takefocus=True,
    borderwidth=10,
    foreground=mycolor1,            # fg, текст віджета
    background="#0000FF",           # bg, фон віджета
    highlightcolor='green',         # підсвічування при фокусі
    highlightbackground='red',      # підсвічування без фокуса
```

```
    ).pack()
tv.set("123")
tk.mainloop()
```

За бажання можна вказати стильові опції для всіх віджетів одразу: за допомогою методу **tk_setPalette()**. Крім використаних вище властивостей, у цьому методі застосовують **selectForeground** та **selectBackground** (передній план та фон виділення), **selectColor** (колір в обраному стані, наприклад, у **Checkbutton**), **insertBackground** (колір точки вставки) та деякі інші.

Примітка. Одержати значення з поля введення можна й за допомогою методу **get()**. Наприклад, якщо назвати об'єкт класу **Entry** іменем **e**, одержати значення можна так: **e.get()**. Правда, цей метод не має тієї гнучкості, яку має метод **get()** екземплярів класу для форматованого тексту **Text**. В останньому випадку можна скористатися тільки всім значенням разом.

6.2.4. Віджет форматованого тексту

Для демонстрації роботи з нетривіальним віджетом розглянемо віджет **ScrolledText** з однойменного модуля *Python*. Цей віджет аналогічний рамці з форматованим текстом та вертикальною смугою прокручування (лістинг 6.4).

Лістинг 6.4. Приклад "Текст із прокруткою"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 4

from Tkinter import *
from ScrolledText import ScrolledText

tk = Tk()               # вікно верхнього рівня
txt = ScrolledText(tk)  # віджет тексту з прокруткою
txt.pack()              # віджет розміщується

for x in range(1, 1024):    # віджет наповнюється текстовим змістом
  txt.insert(END, str(2L**x)+"\n")

tk.mainloop()
```

Зараз варто докладніше розглянути методи та властивості віджета з форматованим текстом.

Для навігації в тексті в *Tk* передбачено спеціальні індекси. Індекси типу 1.0 та END уже зустрічалися – це початок тексту (перший рядок, нульовий символ) та його кінець (у *Tk* рядки нумеруються з одиниці, а символи рядка – з нуля). Наведемо повніший список індексів:

L.C – тут **L** – номер рядка, а **C** – номер символу в рядку;

INSERT – точка вставки;

CURRENT – символ, найближчий до курсору миші;

END – позиція відразу за останнім символом у тексті;

M.first, M.last – індекси початку та кінця позначеної тегом **M** ділянки тексту;

SEL_FIRST, SEL_LAST – індекси початку та кінця виділеного тексту;

M – користувач може визначати свої іменовані позиції в тексті (аналогічно до **END**, **INSERT** або **CURRENT**). При редагуванні тексту маркери зсуватимуться за вказаними для них правилами;

@x,y – символ тексту, найближчий до точки з координатами **x**, **y**.

Наступний приклад (лістинг 6.5) показує, як збагатити форматований текст гіпертекстовими можливостями.

Лістинг 6.5. Приклад "Форматований текст"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 5

from Tkinter import *
import urllib
tk = Tk()
txt = Text(tk, width=64)  # поле з текстом
txt.grid(row=0, column=0, rowspan=2)
addr=Text(tk, background="White", width=64, height=1) # поле адреси
addr.grid(row=0, column=1)
```

```
    page=Text(tk, background="White", width=64) # поле з
html-кодом
    page.grid(row=1, column=1)

    def fetch_url(event):
      click_point = "@%s,%s" % (event.x, event.y)
      trs = txt.tag_ranges("href")  # список областей тексту, відмі-
чених як href
      url = ""
      # визначається, на яку ділянку прийшовся клік миші,
та береться
      # відповідний йому URL
      for i in range(0, len(trs), 2):
        if txt.compare(trs[i], "<=", click_point) and \
           txt.compare(click_point, "<=", trs[i+1]):
          url = txt.get(trs[i], trs[i+1])
      html_doc = urllib.urlopen(url).read()
      addr.delete("1.0", END)
      addr.insert("1.0", url)  # URL розміщується в поле адреси
      page.delete("1.0", END)
      page.insert("1.0", html_doc) # відображається HTML-
документ

    textfrags = ["Python main site: ", "http://www.python.org",
          "\nJython site: ", "http://www.jython.org",
          "\nThat is all!"]
    for frag in textfrags:
      if frag.startswith("http:"):
        txt.insert(END, frag, "href")  # URL розміщується в текст із
міткою href
      else:
        txt.insert(END, frag)        # фрагмент розміщується в текст

    # посилання позначаються підкресленням та синім кольором
    txt.tag_config("href", foreground="Blue", underline=1)
    # при натискуванні миші на тексті, що відмічений як "href",
    # необхідно викликати fetch_url()
    txt.tag_bind("href", "<1>", fetch_url)

    tk.mainloop()        # запускається цикл подій
```

Для надання деяким ділянкам тексту особливих властивостей необхідно їх позначити тегом. У нашому випадку URL позначається тегом **href**. Пізніше методом **tag_config()** задаються властивості відображення тексту, позначеного таким тегом. Методом **tag_bind()** зв'язується деяка подія (клік миші) із викликом указаної функції (**fetch_url()**).

У самій функції **fetch_url()** потрібно спочатку визначити, на яку саме ділянку тексту припав клік миші. Для цього методом **tag_ranges()** отримують всі інтервали, які позначено як **href**. Для визначення конкретного URL порівнюють (методом **compare()**) точки натискання мишею з кожним інтервалом. Так знаходять інтервал, на який припало натискання, та методом **get()** отримують текстове значення знайденого інтервалу. Визначивши URL, у поле записують адресу й отримують HTML-код, що відповідає URL.

Цей приклад показує основні принципи роботи з форматованим текстом. Застосованими методами арсенал віджета не вичерпується. Про інші методи і властивості можна довідатися з документації.

6.2.5. Менеджери розташування

Наступний приклад (лістинг 6.6) достатньо наочний, щоб зрозуміти принципи роботи менеджерів розташування, які є у *Tk*. У трьох рамках можна застосувати різні менеджери: **pack**, **grid** та **place**.

Лістинг 6.6. Приклад "Менеджери розташування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 6

from Tkinter import *
tk = Tk()

# Створюємо три рамки
frames = {}
b = {}
for fn in 1, 2, 3:
```

```
    f = Frame(tk, width=100, height=200, bg="White")
    f.pack(side=LEFT, fill=BOTH)
    frames[fn] = f
    for bn in 1, 2, 3, 4:  # Створюємо кнопки для кожної з рамок
      b[fn, bn] = Button(frames[fn], text="%s.%s" % (fn, bn))

  # Перша рамка:
  # Спочатку дві кнопки прикріплюємо до лівого краю
  b[1, 1].pack(side=LEFT, fill=BOTH, expand=1)
  b[1, 2].pack(side=LEFT, fill=BOTH, expand=1)
  # Ще дві - до нижнього
  b[1, 3].pack(side=BOTTOM, fill=Y)
  b[1, 4].pack(side=BOTTOM, fill=BOTH)

  # Друга рамка:
  # Дві кнопки зверху
  b[2, 1].grid(row=0, column=0, sticky=NW+SE)
  b[2, 2].grid(row=0, column=1, sticky=NW+SE)
  # і одна на дві колонки знизу
  b[2, 3].grid(row=1, column=0, columnspan=2, sticky=NW+SE)

  # Третя рамка:
  # Кнопки заввишки та завширшки близько 40 % розмірів рамки, якір у лівому верхньому куті.
  # Координати якоря складають 1/10 від ширини й висоти рамки
  b[3, 1].place(relx=0.1, rely=0.1, relwidth=0.4, relheight=0.4, anchor=NW)
  # Кнопка строго по центру. Якір у центрі кнопки
  b[3, 2].place(relx=0.5, rely=0.5, relwidth=0.4, relheight=0.4, anchor=CENTER)
  # Якір у центрі кнопки. Координати якоря складають 9/10 від ширини й висоти рамки
  b[3, 3].place(relx=0.9, rely=0.9, relwidth=0.4, relheight=0.4, anchor=CENTER)

  tk.mainloop()
```

Менеджер **pack** просто заповнює внутрішній простір на підставі переваги того чи іншого краю, необхідності заповнити весь простір. У деяких випадках йому доводиться змінювати розміри підпорядкованих віджетів. Цей менеджер варто використовувати лише для досить простих схем розташування віджетів.

Менеджер **grid** поміщає віджети у клітки сітки (це дуже схоже на спосіб верстання таблиць в HTML). Кожному розташовуваному віджету даються координати в одній із чарунок сітки (**row** – рядок, **column** – стовпець), а також, якщо потрібно, стільки наступних чарунок (у рядках нижче або у стовпцях праворуч), скільки він може зайняти (властивості **rowspan** або **columnspan**). Це найгнучкіший з усіх менеджерів.

Менеджер **place** дозволяє розташовувати віджети з довільними координатами та з довільними розмірами підпорядкованих віджетів. Розміри та координати можна вказувати у процентах від розміру віджета-хазяїна.

Безпосередньо всередині одного віджета не можна використовувати більше одного менеджера розташування: менеджери можуть накласти суперечливі обмеження на вкладені віджети, що призведе до унеможливлення розташування внутрішніх віджет.

6.2.6. Зображення у *Tkinter*

Засобами *Tkinter* можна виводити не тільки текст, примітивні форми (за допомогою віджета **Canvas**), але й растрові зображення. Лістинг 6.7 демонструє виведення іконки з растровим зображенням (для цього прикладу потрібно попередньо встановити пакет *Python Imaging Library*, PIL).

Лістинг 6.7. Приклад "Растрові зображення"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 7

import Tkinter, Image, ImageTk

FILENAME = "lena.jpg"  # файл із графічним зображенням

tk = Tkinter.Tk()
c = Tkinter.Canvas(tk, width=128, height=128)
```

```
src_img = Image.open(FILENAME)

img = ImageTk.PhotoImage(src_img)
c.create_image(0, 0, image=img, anchor="nw")
c.pack()
Tkinter.Label(tk, text=FILENAME).pack()

tk.mainloop()
```

Тут використано віджет-рисунок (Canvas). За допомогою функцій із пакетів **Image** та **ImageTk** з PIL отримується об'єкт-зображення, що підходить для включення в рисунок *Tkinter*. Властивість **anchor** задає кут, що прив'язується до координат (0, 0) у рисунку. У цьому прикладі це північно-західний кут (**NW** – *North-West*). Інші можливості: **n** (північ), **w** (захід), **s** (південь), **e** (схід), **ne**, **sw**, **se** та **c** (центр).

У наступному прикладі (лістинг 6.8) показано графічні примітиви, які можна використовувати на рисунку (наведені коментарі пояснюють властивості графічних об'єктів усередині віджета-рисунка).

Лістинг 6.8. Приклад "Графічні примітиви"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 8

from Tkinter import *

tk = Tk()
# Рисунок 300x300 пікселей, фон - білий
c = Canvas(tk, width=300, height=300, bg="white")

c.create_arc((5, 5, 50, 50), style=PIESLICE) # Сектор ("шматок пирога")
c.create_arc((55, 5, 100, 50), style=ARC)    # Дуга
c.create_arc((105, 5, 150, 50), style=CHORD,  # Сегмент
        start=0, extent=150, fill="blue")  # від 0 до 150 градусів
# Ламана зі стрілкою на кінці
c.create_line([(5, 55), (55, 55), (30, 95)], arrow=LAST)
# Крива (згладжена ламана)
c.create_line([(105, 55), (155, 55), (130, 95)], smooth=1)
```

```
    # Багатокутник зеленого кольору
    c.create_polygon([(205, 55), (255, 55), (230, 95)], fill="green")
    # Овал
    c.create_oval((5, 105, 50, 120), )
    # Прямокутник червоного кольору з великою сірою границею
    c.create_rectangle((105, 105, 150, 130), fill="red",
                outline="grey", width="5")
    # Текст
    c.create_text((5, 205), text=" Hello", anchor="nw")
    # Ця точка візуально позначає кут прив'язки
    c.create_oval((5, 205, 6, 206), outline="red")
    # Текст із заданим вирівнюванням
    c.create_text((105, 205), text="Hello,\nmy friend!",
              justify=LEFT, anchor="c")
    c.create_oval((105, 205, 106, 206), outline="red")
    # Ще один варіант
    c.create_text((205, 205), text="Hello,\nmy friend!",
              justify=CENTER, anchor="se")
    c.create_oval((205, 205, 206, 206), outline="red")

    c.pack()
    tk.mainloop()
```

Зазначимо, що методи **create_*** створюють об'єкти, властивості яких можна змінювати далі: перемістити в інше місце, перефарбувати, видалити, змінити порядок і т. д. У наступному прикладі (лістинг 6.9) можна намалювати кружок, що міняє колір кліком миші.

Лістинг 6.9. Приклад "Кружок"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 9

from Tkinter import *
from random import choice

colors = "Red Orange Yellow Green LightBlue Blue Violet".split()
```

```
R = 10

tk = Tk()
c = Canvas(tk, bg="White", width="4i", height=300, relief=
SUNKEN)
c.pack(expand=1, fill=BOTH)

def change_ball(event):
  c.coords(CURRENT, (event.x-R, event.y-R, event.x+R,
event.y+R))
  c.itemconfigure(CURRENT, fill=choice(colors))

oval = c.create_oval((100-R, 100-R, 100+R, 100+R), fill="Black")
c.tag_bind(oval, "<1>", change_ball)
tk.mainloop()
```

Тут намальовано кружок із радіусом *R*, із ним пов'язана функція **change_ball()**, яка виконується натисканням кнопки миші. У зазначеній функції вказуються нові координати кружка (його центр розташований у місці натискання миші), а потім змінюється колір випадковим чином за допомогою методу **itemconfigure()**. Тег **CURRENT** у *Tkinter* використаний, щоб указати об'єкт, який прийняв подію.

6.2.7. Графічне застосування на *Tkinter*

Тепер варто розглянути невелике застосування, написане з використанням *Tkinter*. У цьому застосуванні буде завантажено файл із графічним зображенням. Застосування матиме найпростіше меню **File** із пунктами **Open** та **Exit**, а також віджет **Canvas**, на якому й демонструватимуться зображення (лістинг 6.10). Знову буде потрібний пакет PIL.

Лістинг 6.10. Приклад "Просте застосування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 10

from Tkinter import *
import Image, ImageTk, tkFileDialog
global img, imgobj
```

```
    def show():
      global img, imgobj
      # Запит на ім'я файла
      filename = tkFileDialog.askopenfilename()
      if filename != ():  # Якщо ім'я файла задано користувачем
        # рисується зображення з файла
        src_img = Image.open(filename)
        img = ImageTk.PhotoImage(src_img)
        # конфігурується зображення на рисунку
        c.itemconfigure(imgobj, image=img, anchor="nw")

    tk = Tk()
    main_menu = Menu(tk)      # формується меню
    tk.config(menu=main_menu)  # меню додається до вікна
    file_menu = Menu(main_menu)  # створюється підменю
    main_menu.add_cascade(label="File", menu=file_menu)
    # Заповнюється меню File
    file_menu.add_command(label="Open", command=show)
    file_menu.add_separator()  # роздільник для відокремлення
пунктів меню
    file_menu.add_command(label="Exit", command=tk.destroy)

    c = Canvas(tk, width=300, height=300, bg="white")
    # готуємо об'єкт-зображення на рисунку
    imgobj = c.create_image(0, 0)
    c.pack()

    tk.mainloop()
```

Зазначимо, що тут довелося застосувати дві глобальні змінні. Це не дуже добре. Існує інший підхід, коли застосування створюється на основі вікна верхнього рівня. Таким чином, саме застосування стає особливим віджетом. Перероблену програму наведено в лістингу 6.11.

Лістинг 6.11. Перероблений приклад "Просте застосування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Tkinter приклад 11
```

```
from Tkinter import *
import Image, ImageTk, tkFileDialog

class App(Tk):
  def __init__(self):
    Tk.__init__(self)
    main_menu = Menu(self)
    self.config(menu=main_menu)
    file_menu = Menu(main_menu)
    main_menu.add_cascade(label="File", menu=file_menu)
    file_menu.add_command(label="Open",
command=self.show_img)
    file_menu.add_separator()
    file_menu.add_command(label="Exit",
command=self.destroy)

    self.c = Canvas(self, width=300, height=300, bg="white")
    self.imgobj = self.c.create_image(0, 0)
    self.c.pack()

  def show_img(self):
    filename = tkFileDialog.askopenfilename()
    if filename != ():
      src_img = Image.open(filename)
      self.img = ImageTk.PhotoImage(src_img)
      self.c.itemconfigure(self.imgobj, image=self.img, anchor="nw")

app = App()
app.mainloop()
```

В об'єкті наявна інформація, що до цього була глобальною, з усіма обмеженнями, що з цього випливають. Можна далі виділити в окремий метод налаштування меню (якщо застосування буде динамічно змінювати меню, то об'єкти-меню теж можна зберігати в застосуванні).

Примітка. На деяких системах нові версії *Python* погано працюють із національними кодуваннями, зокрема, із кодуваннями для кирилиці. Це пов'язано з переходом на Unicode Tcl/Tk. Проблем можна уникнути, якщо використовувати кодування UTF-8 у рядках, які повинні виводитися у віджетах.

6.3. Основні принципи роботи з *PyQt4*

6.3.1. Відомості про *PyQt*

Qt – це потужний інструментарій для створення кросплатформних графічних застосувань (для ОС: *BSD, Mac OS X, Linux, Windows, Symbian S60, Windows Mobile, QNX тощо). Бібліотеку написано мовою C++ компанією Trolltech, яку недавно купила компанія Nokia (http://qt.nokia.com). Бібліотека настільки популярна, що створено величезну кількість прив'язок до неї для мов: *Python,* Ruby, PHP, Perl, C#, Java. Розробкою *PyQt* займається компанія Riverbank.

Бібліотека *PyQt4* реалізована у вигляді *Python*-модулів та включає модулі:

- **QtCore** – містить підтримку подій (сигнали/слоти), юнікоду, регулярних виразів, багатопоточності тощо;
- **QtGui** – містить у собі класи GUI-віджетів: таблиці, списки, мітки тощо;
- **QtHelp** – модуль для створення довідки в застосуванні;
- **QtNetwork** – модуль із класами для роботи з мережею Інтернет (підтримка протоколів: HTTP, HTTPS, FTP);
- **QtOpenGL** – підтримка 3D-графіки;
- **QtScript** – підтримка в програмі ActionScript-подібної мови;
- **QtSql** – підтримка СКБД;
- **QtSvg** – підтримка SVG-графіки;
- **QtTest** – проведення юніт-тестів;
- **QtWebKit** – містить віджет-браузер на основі движка WebKit;
- **QtXml** – підтримка формату XML (робота за допомогою SAX- або DOM-інтерфейсу);
- **QtXmlPatterns** – підтримка XQuery та XPath;
- **Phonon** – підтримка мультимедіа;
- **QtAssistant** – інтерфейс до QtAssistant для інтеграції зі своїм застосуванням;
- **QtDesigner** – створення розширень для QtDesigner;
- **QAxContainer** (тільки Windows) – модуль для роботи з ActiveX- та COM-об'єктами.

6.3.2. Приклади програм на *PyQt4*

Перш ніж перейти до теорії, спочатку розглянемо декілька прикладів програм (лістинги 6.12–6.16).

Лістинг 6.12. Приклад "Звичайне вікно"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 1
import sys
from PyQt4 import QtGui

app = QtGui.QApplication(sys.argv)
widget = QtGui.QWidget()
widget.resize(250, 150)
widget.setWindowTitle('Приклад PyQt')
widget.show()
sys.exit(app.exec_())
```

У рядках 4–5 включається модуль **sys**, щоб мати можливість використовувати аргументи командного рядка, і модуль **QtGui** – для роботи з віджетами. Кожне застосування, написане на *PyQt*4, має створювати об'єкт **QApplication**. У 8-му рядку створюється об'єкт застосування й у конструкторі йому передаються аргументи командного рядка. Ці аргументи специфічні лише для застосувань, написаних на *PyQt*4. Клас **QApplication** міститься в модулі **QtGui**. У 9-му рядку створюється звичайний порожній віджет. **QWidget** – це основний GUI-клас в *PyQt*4. Через те, що створюється віджет без параметрів (без батька), то він не є нащадком, а це значить, що він стане незалежним вікном. У 10-му рядку встановлюється розмір віджета (вікна) 250×150, де 250 – ширина, а 150 – висота.

Далі в 11-му рядку вказується заголовок для вікна. У 12-му рядку команда робить віджет видимим. Останній рядок є викликом головного циклу програми (**MainLoop**), тобто з цього рядка програма почне обробляти події (сигнал закриття вікна, клік мишею і т. д.). Після завершення циклу функція **exec_** поверне числовий код завершення у функцію **exit**. Наприклад, якщо застосування завершилося нормально, то код дорівнюватиме 0, інакше функція поверне код помилки.

Лістинг 6.13. Приклад "Вікно з іконкою"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 2

import sys
from PyQt4 import QtGui, QtCore

class WithIcon(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(10, 300, 250, 150)
        self.setWindowTitle(self.trUtf8('Застосування з іконкою'))
        self.setWindowIcon(QtGui.QIcon('1_2.png'))

app = QtGui.QApplication(sys.argv)
with_icon = WithIcon()
with_icon.show()
sys.exit(app.exec_())
```

У 6-му рядку імпортується модуль **QtCore** для того, щоб можна було використовувати функцію **trUtf8**. Для виклику функції **trUtf8** потрібно, щоб кодування вихідного файла було в UTF-8. За допомогою цієї функції можна працювати мовами, що послуговуються кирилицею (і не тільки), тобто, якщо не використовувати **trUtf8**, то кириличний текст виводитиметься у вигляді карлючок.

У 8-му рядку створюється клас **WithImage**, успадкований від **QWidget**; у рядках 9-10 ініціалізується об'єкт **QWidget**.

У 12-му рядку встановлюються позиція та розмір віджета (вікна в цьому випадку), де 10×300 – це позиція віджета на екрані, а 250×150 – ширина й висота відповідно. У 13-му рядку вказується заголовок вікна російською мовою через функцію **trUtf8**. У 14-му рядку встановлюється іконка для вікна. У конструкторі **QIcon** потрібно передати шлях до рисунка (іконки). Після цього створений об'єкт з типом **QIcon** передається у функцію **setWindowIcon**.

Лістинг 6.14. Приклад "Вікно із спливаючою підказкою"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 3
```

```
import sys
from PyQt4 import QtGui, QtCore

class WithTooltip(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle(self.trUtf8('Підказка'))

        self.setToolTip(self.trUtf8('Це <b>QWidget</b>.'))
        QtGui.QToolTip.setFont(QtGui.QFont('Verdana', 11))

app = QtGui.QApplication(sys.argv)
tooltip = WithTooltip()
tooltip.show()
sys.exit(app.exec_())
```

У цьому прикладі додано функцію **setToolTip**, що встановлює для віджета текст спливаючої підказки. У *PyQt*4 можна використовувати HTML-теги для форматування тексту.

У 16-му рядку встановлюється шрифт виведення та його розмір.

Лістинг 6.15. Приклад "Вікно з кнопкою виходу"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 4

import sys
from PyQt4 import QtGui, QtCore

class WithQuitButton(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 200, 100)
        self.setWindowTitle(self.trUtf8('Вікно з кнопкою'))

        quit = QtGui.QPushButton(self.trUtf8('Вихід!'), self)
        quit.setGeometry(10, 10, 60, 35)

        self.connect(quit, QtCore.SIGNAL('clicked()'),
            QtGui.qApp, QtCore.SLOT('quit()'))
```

```
app = QtGui.QApplication(sys.argv)
qb = WithQuitButton()
qb.show()
sys.exit(app.exec_())
```

У цьому прикладі додано кнопку виходу з програми.

У 15-му рядку створюється кнопка, де перший параметр у конструкторі – це текст кнопки, а другий – об'єкт-батько. Якщо не вказувати в конструкторі об'єкт-батько, то кнопка матиме власне вікно. Далі в 16-му рядку встановлюються координати кнопки на формі (віджеті або екрані, якщо кнопка – окремий об'єкт), де 10×10 – позиція, а 60×35 – розмір.

У *PyQt*4 всі події (натискання кнопки, введення тексту тощо) реалізовано через концепцію сигналів та слотів. Тобто, якщо потрібно обробити натиснення кнопки миші, то слід з'єднати сигнал "натиснення кнопки" із необхідним слотом. Слотом може бути звичайна або вбудована функція (наприклад, функція виходу **"quit"**). З'єднання відбувається через функцію **QtCore.QObject.connect**, а роз'єднання через **QtCore.QObject.disconnect**. У конструкторі **connect()** або **disconnect()** потрібно передати ім'я об'єкта та сигнал об'єкта, який треба обробити, а також 3-й та 4-й параметри – ім'я об'єкта та його слот. Іменем об'єкта також може бути й сам клас **self**, а слот – функцією класу **self._func**.

У 18–19 рядках відбувається з'єднання сигналу натискання кнопки та слота виходу із застосування.

Лістинг 6.16. Приклад "Вікно з повідомленням"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 5

import sys
from PyQt4 import QtGui

class MessageBox(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle(self.trUtf8('Повідомлення'))
```

```
    def closeEvent(self, event):
        reply = QtGui.QMessageBox.question(self, self.trUtf8
('Повідомлення'),
            self.trUtf8("Ви впевнені, що хочете вийти?"),
            QtGui.QMessageBox.Yes, QtGui.QMessageBox.No)

        if reply == QtGui.QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()

app = QtGui.QApplication(sys.argv)
qb = MessageBox()
qb.show()
sys.exit(app.exec_())
```

При закритті **QWidget** генерується подія **closeEvent** тому, що створений клас **MessageBox** є успадкованим від **QtGui.Qwidget**. У ньому модифікується функція **closeEvent**, 15-й рядок.

Команди в 16–18 рядках відображають повідомлення. За допомогою конструкторів передаються заголовок повідомлення, текст повідомлення та дві кнопки ("Так", "Ні"). Як тільки в повідомленні буде натиснуто одну з кнопок, функція **QtGui.QMessageBox.question** поверне значення цієї кнопки у змінну **reply**.

Потім у 20–23 рядках перевіряється, яку з кнопок було натиснуто. Якщо це кнопка "Так", то посилається сигнал далі (**event.accept**) і програма завершується, якщо "Ні" – сигнал відкидається і програма працює далі.

6.3.3. Меню і панель інструментів у *PyQt4*

Головне вікно. За допомогою класу **QMainWindow** можна створити основне (головне) вікно застосування. У цьому основному вікні можна створити рядок стану, панель інструментів та панель меню.

Рядок стану. Цей віджет потрібний для відображення додаткової інформації, що розташовується в нижній частині вікна.

Код програми "Рядок стану" наведено в лістингу 6.17.

Лістинг 6.17. Приклад "Рядок стану"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 6
```

```
import sys
from PyQt4 import QtGui

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(250, 150)
        self.setWindowTitle(self.trUtf8('Рядок стану'))
        button, label = QtGui.QPushButton(self.trUtf8("Кнопка")),
QtGui.QLabel(self.trUtf8("текст"))
        self.statusBar().addWidget(button)
        self.statusBar().addWidget(label)
        button.show()
        label.show()

        self.statusBar().showMessage(self.trUtf8('Текст'))

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Функція **statusBar()** у рядку 15 класу **QMainWindow** повертає посилання на об'єкт **QStatusBar** (рядок стану), тільки якщо цей віджет існує на формі, інакше функція поверне значення **None**.

QStatusBar має слот **showMessage**, що виводить повідомлення на рядку стану, якщо на ньому немає сторонніх віджетів.

У рядок стану можна помістити будь-який віджет, для цього потрібно використати функцію **addWidget**.

Наприклад, потрібно додати кнопку та мітку з текстом:

```
...
button, label = QtGui.QPushButton(self.trUtf8("Кнопка")),
QtGui.QLabel(self.trUtf8("текст"))
self.statusBar().addWidget(button)
self.statusBar().addWidget(label)
button.show()
label.show()
...
```

Панель меню. Панель меню – це головний віджет (лістинг 6.18), на який часто звертають увагу користувачі, тому що тут міститься більшість згрупованих команд.

Лістинг 6.18. Приклад "Панель меню"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 7

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(250, 150)
        self.setWindowTitle(self.trUtf8('Панель меню'))

        exit = QtGui.QAction(QtGui.QIcon('exit.png'), self.trUtf8
('Вихід'), self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip(self.trUtf8('Вихід із застосування'))
        self.connect(exit, QtCore.SIGNAL('triggered()'), QtCore.
SLOT('close()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu(self.trUtf8('Файл'))
        file.addAction(exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Меню, контекстне меню та панель інструментів у *PyQt*4 використовують об'єкти **QAction** замість звичайних кнопок. У 15-му рядку конструктору передається об'єкт **QIcon** для створення іконки (якщо іконка не потрібна, то слід передати в конструктор порожній об'єкт **QtGui.QIcon("")**), другий параметр –

це заголовок, а третій параметр – об'єкт-батько (у нашому випадку – це вікно). У 16-му рядку об'єкту **QAction** призначається гаряча клавіша **Ctrl+Q.**

setStatusTip() в 17-му рядку задає повідомлення, яке буде відображатися в рядку стану при наведенні курсору.

Після цього сигнал **triggered()** з'єднується зі слотом закриття вікна **close()** у 18-му рядку. Сигнал **triggered()** відсилається при виборі об'єкта **QAction**. У 22–24 рядках створюється меню із заголовком "Файл" і додається об'єкт **QAction** (тобто дія) – для закриття вікна.

Панель інструментів. Панель інструментів (лістинг 6.19) дозволяє швидше звертатися до функцій програми порівняно з меню.

Лістинг 6.19. Приклад "Панель інструментів"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 8

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
  def __init__(self):
    QtGui.QMainWindow.__init__(self)

    self.resize(250, 150)
    self.setWindowTitle(self.trUtf8('Панель інструментів'))

    self.exit = QtGui.QAction(QtGui.QIcon('exit.png'), self.trUtf8
('Вихід'), self)
    self.exit.setShortcut('Ctrl+Q')
     self.connect(self.exit,  QtCore.SIGNAL('triggered()'),
QtCore.SLOT('close()'))

    self.toolbar = self.addToolBar('Exit')
    self.toolbar.addAction(self.exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

У 15–17 рядках створюється кнопка (дія **QAction**) для закриття вікна. За допомогою функції **addToolBar('Exit')** створюється панель інструментів із заголовком (його можна побачити, якщо "відірвати" панель від вікна) **"Exit"**. Далі панель розміщується в основному вікні програми.

У 20-му рядку додається кнопка закриття через функцію **toolbar.addAction(self.exit)**.

Центральний віджет. Наостанок розглянемо функцію встановлення центрального віджета **setCentralWidget()** у класі **QMainWindow**. Центральний віджет займає весь вільний простір на формі (лістинг 6.20).

Лістинг 6.20. Приклад "Центральний віджет"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 9

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(350, 250)
        self.setWindowTitle(self.trUtf8('Головне вікно'))

        textEdit = QtGui.QTextEdit()
        self.setCentralWidget(textEdit)

        exit = QtGui.QAction(QtGui.QIcon('exit.gif'), self.trUtf8
('Вихід'), self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip(self.trUtf8('Вихід із застосовання'))
        self.connect(exit, QtCore.SIGNAL('triggered()'), QtCore.
SLOT('close()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu(self.trUtf8('Файл'))
        file.addAction(exit)
```

```
        toolbar = self.addToolBar('Exit')
        toolbar.addAction(exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

У 15-му рядку створюється текстовий віджет та в 16-му рядку цей віджет призначається центральним.

6.3.4. Сигнали та слоти в *PyQt4*

Події (events). За допомогою сигналів та слотів у бібліотеці *PyQt*4 відбувається взаємодія між об'єктами. Об'єктами можуть виступати, наприклад, віджети.

Після запуску головного циклу **app.exec_()** програма починає обробляти події. Наприклад, коли буде натиснуто кнопку (**QPushButton**), то відбудеться подія (сигнал) **clicked()**. Події бувають системними або є результатом дій користувача (наприклад, натискання кнопки).

Для обробки події (сигналу) об'єкта (наприклад, **QPushButton**) необхідно її зв'язати з потрібним слотом: бібліотечною функцією або функцією користувача.

У лістингу 6.21 наведено код програми "Сигнали і слоти".

Лістинг 6.21. Приклад "Сигнали і слоти"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 10

import sys
from PyQt4 import QtGui, QtCore

class SigSlot(QtGui.QWidget):
   def __init__(self, parent=None):
      QtGui.QWidget.__init__(self, parent)

      self.setWindowTitle(self.trUtf8('Сигнали і слоти'))

      lcd = QtGui.QLCDNumber(self)
      slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
```

```
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(lcd)
        vbox.addWidget(slider)

        self.setLayout(vbox)
        self.connect(slider,    QtCore.SIGNAL('valueChanged(int)'),
lcd, QtCore.SLOT('display(int)'))
        self.resize(250, 150)

    app = QtGui.QApplication(sys.argv)
    sigslot = SigSlot()
    sigslot.show()
    sys.exit(app.exec_())
```

У прикладі використовується два віджети, LCD-індикатор та повзунок. Повзунок (**QSlider**) має подію **valueChanged(int)**, що відбувається при зміні положення повзунка. Сигнал також несе в собі значення поточної позиції повзунка: **valueChanged(int)**, де **int** – ціле значення для слота.

Метод **connect** має чотири параметри: **connect** (джерело, сигнал, приймач, слот), де **джерело** – це об'єкт, що генерує подію; **сигнал** – подія об'єкта-джерела; **приймач** – об'єкт, який прийматиме сигнал; **слот** – функція або ім'я функції в **QtCore.SLOT**, що оброблятиме сигнал. Для з'єднання сигналу зі слотом кількість та тип їхніх аргументів мають бути **однакові!** Хоча фактично сигнал може мати кількість аргументів більшу, ніж слот. При цьому, коли сигнал буде відправлено слоту, зайві аргументи ігноруватимуться.

У 21-му рядку з'єднується сигнал повзунка (**valueChanged(int)**) зі слотом (**display(int)**) LCD-індикатора. Слот **display(int)** індикатора служить для виведення числового значення. Його можна використовувати як звичайну функцію: **lcd.display(200)** виводить на індикатор число 200.

Перевизначення подій. У *PyQt*4 можна перевизначати події (лістинг 6.22).

У 16-му рядку перевизначається подія *keyPressEvent*(*QKeyEvent*). Тепер при натисканні клавіші **Esc** вікно програми закривається.

Лістинг 6.22. Приклад "Перевизначення подій"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 11

import sys
from PyQt4 import QtGui, QtCore

class Escape(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Перевизначення подій'))
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'), QtCore.
SLOT('close()'))

    def keyPressEvent(self, event):
        if event.key() == QtCore.Qt.Key_Escape:
            self.close()

app = QtGui.QApplication(sys.argv)
sigslot = Escape()
sigslot.show()
sys.exit(app.exec_())
```

Відправлення сигналів (генерація подій). Об'єкти, що успадковані від **QtCore.Qobject,** можуть відправляти сигнали. Наприклад, потрібно створити програму, в якій за кліком миші в області вікна буде відправлено сигнал **ourSignal(),** та з'єднати цей сигнал зі слотом закриття вікна.

У лістингу 6.23 наведено код програми "Генерація подій".

Лістинг 6.23. Приклад "Генерація подій"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 12

import sys
from PyQt4 import QtGui, QtCore
```

```
class Emit(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Власний сигнал'))
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('ourSignal()'), QtCore.
SLOT('close()'))

    def mousePressEvent(self, event):
        self.emit(QtCore.SIGNAL('ourSignal()'))

app = QtGui.QApplication(sys.argv)
qb = Emit()
qb.show()
sys.exit(app.exec_())
```

У 14-му рядку сигнал **ourSignal()** з'єднується зі слотом закриття вікна програми.

У 16-му рядку перевизначається подія **mousePressEvent (QMouseEvent)**. Тепер за кліком миші в будь-якій області вікна буде відправлений (згенерований) сигнал **ourSignal()**.

Якщо потрібно передати сигнал із заздалегідь визначеним значенням, то це можна зробити так:

```
...
# на основі прикладу 3.1
self.emit(QtCore.SIGNAL('valueChanged(int)'), 767) # де
"767"– це значення, яке буде передано разом із сигналом
...
```

Створення своїх слотів. Нехай потрібно створити застосування зі своїм слотом (лістинг 6.24).

Лістинг 6.24. Приклад "Свій слот у *PyQt*"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 13

import sys, random
from PyQt4 import QtGui, QtCore
```

```
class withSlot(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Свій слот у PyQt'))
        self.resize(300, 50)
        self.button=QtGui.QPushButton('Get!', self)
        self.button.show()

        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button)
        self.setLayout(vbox)
        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.getIt)

    def getIt(self):
        self.button.setText(str(random.randint(1, 10)))

app = QtGui.QApplication(sys.argv)
qb = withSlot()
qb.show()
sys.exit(app.exec_())
```

У 20-му рядку сигнал натискання кнопки з'єднується з нашим слотом **getIt()**, що випадковим чином змінює текст на кнопці.

6.3.5. Розміщення віджетів у *PyQt4*

Важливою частиною у створенні будь-якого GUI-застосування є розміщення віджетів. У *PyQt4* віджети можна розміщувати двома способами: абсолютного позиціонування (лістинг 6.25) або за допомогою схем розташування (layout).

Абсолютне позиціонування. При використанні абсолютного позиціонування необхідно вказувати розміри та позицію кожного віджета вручну, а також враховувати таке:

- розміри та позиція віджетів не змінюються, якщо розмір вікна змінюється;
- програма може мати різний вигляд на інших платформах;
- зміна шрифтів у програмі може позначитися на її дизайні;
- при зміні дизайну програмі доведеться знову встановлювати розміри та позицію віджетів.

Лістинг 6.25. Приклад
"Абсолютне позиціонування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 14

import sys
from PyQt4 import QtGui

class Absolute(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('*nix')

        label = QtGui.QLabel('Gentoo', self)
        label.move(15, 10)
        label = QtGui.QLabel('Ubuntu', self)
        label.move(35, 40)
        label = QtGui.QLabel('Debian', self)
        label.move(55, 65)
        self.resize(250, 100)

app = QtGui.QApplication(sys.argv)
qb = Absolute()
qb.show()
sys.exit(app.exec_())
```

У цьому прикладі використовується метод **move(x, y)** для визначення позиції віджета, де **x** – це аргумент, що визначає відступ ліворуч, а **y** – відступ згори.

Вертикальні та горизонтальні схеми розташування. У *PyQt4* існують класи для автоматичного розміщення віджетів, тобто за зміну розміру та позиції відповідають схеми розташування, а не програміст. Розглянемо схеми розташування **QHBoxLayout** та **QVBoxLayout** (лістинг 6.26)**.** Вони вишиковують віджети горизонтально та вертикально. Для створення вільного місця між віджетами використовують розтягування (**stretch**).

Наприклад, потрібно створити програму, що має у вікні дві кнопки в нижньому правому кутку. Для цього використовують вертикальну та горизонтальну схеми розташування.

Лістинг 6.26. Приклад "Схеми розташування"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 15

import sys
from PyQt4 import QtGui

class BoxLayout(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Використання схем розташувань'))

        ok = QtGui.QPushButton(self.trUtf8("Добре"))
        cancel = QtGui.QPushButton(self.trUtf8("Відміна"))

        hbox = QtGui.QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(ok)
        hbox.addWidget(cancel)

        vbox = QtGui.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)

        self.setLayout(vbox)
        self.resize(300, 150)

app = QtGui.QApplication(sys.argv)
qb = BoxLayout()
qb.show()
sys.exit(app.exec_())
```

У 17-му рядку створюється горизонтальна схема розташування, далі у 18-му рядку – вільний простір, а в 19–20 рядках додаються кнопки. Тепер перед кнопками буде порожня область.

У 22–24 рядках створюється вертикальна схема розташування та додається вільна область перед горизонтальноюсхемою.

У 26-му рядку встановлюється вертикальна схема як головна схема розташування вікна.

Розміщення елементів у вигляді таблиці. Таблична схема розташування **QGridLayout** (лістинг 6.27)– найбільш універсальна. Вона ділить простір на рядки та стовпці.

Лістинг 6.27. Приклад
"Таблична схема розташування-1"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 16

import sys
from PyQt4 import QtGui

class GridLayout(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Таблична схема розташування'))

        names = [self.trUtf8('Очищення'), self.trUtf8('Видалити'), '', self.trUtf8('Закрити'), '7', '8', '9', '/',
                 '4', '5', '6', '*', '1', '2', '3', '-',
                 '0', '.', '=', '+']

        grid = QtGui.QGridLayout()

        j = 0
        pos = [(0, 0), (0, 1), (0, 2), (0, 3),
               (1, 0), (1, 1), (1, 2), (1, 3),
               (2, 0), (2, 1), (2, 2), (2, 3),
               (3, 0), (3, 1), (3, 2), (3, 3 ),
               (4, 0), (4, 1), (4, 2), (4, 3)]

        for i in names:
            button = QtGui.QPushButton(i)
            if j == 2:
                grid.addWidget(QtGui.QLabel(''), 0, 2)
            else: grid.addWidget(button, pos[j][0], pos[j][1])
            j = j + 1

        self.setLayout(grid)
```

```
app = QtGui.QApplication(sys.argv)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

У цьому прикладі створюється таблиця з кнопок. Для виконання одного пробілу замість кнопки додається порожній об'єкт **QLabel**. У 18-му рядку створюється таблична схема розташування.

У 29–31 рядках додаються віджети в таблицю за допомогою методу **addWidget (віджет, номер рядка, номер стовпця)**.

Віджети можуть займати кілька рядків або стовпців у таблиці, як у прикладі з табличним віджетом у лістингу 6.28.

Лістинг 6.28. Приклад "Таблична схема розташування-2"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 17

import sys
from PyQt4 import QtGui

class GridLayout2(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle(self.trUtf8('Таблична схема розташування'))

        title = QtGui.QLabel(self.trUtf8('Назва'))
        author = QtGui.QLabel(self.trUtf8('Автор'))
        review = QtGui.QLabel(self.trUtf8('Огляд'))

        titleEdit = QtGui.QLineEdit()
        authorEdit = QtGui.QLineEdit()
        reviewEdit = QtGui.QTextEdit()

        grid = QtGui.QGridLayout()
        grid.setSpacing(10)

        grid.addWidget(title, 1, 0)
        grid.addWidget(titleEdit, 1, 1)
```

```
        grid.addWidget(author, 2, 0)
        grid.addWidget(authorEdit, 2, 1)

        grid.addWidget(review, 3, 0)
        grid.addWidget(reviewEdit, 3, 1, 5, 1)

        self.setLayout(grid)
        self.resize(350, 300)

app = QtGui.QApplication(sys.argv)
qb = GridLayout2()
qb.show()
sys.exit(app.exec_())
```

У 22–23 рядках створюється таблична схема та встановлюється відступ у 10 пікселів між віджетами.

При додаванні віджета в таблицю (крім аргументів "номер стовпця" та "номер рядка") можна вказати кількість зайнятих стовпців та рядків. Таким чином, у 32-му рядку вказується, що текстовий віджет займатиме п'ять рядків та один стовпець.

6.3.6. Діалогові вікна в *PyQt4*

Діалогові вікна є невід'ємною частиною більшості сучасних GUI-застосувань і використовуються для введення даних, зміни даних, збереження або відкриття файлів тощо. Діалогові вікна бувають двох типів: стандартні й користувальницькі.

Діалогове вікно для введення даних. Клас **QInputDialog** забезпечує простий та зручний діалог для введення одного значення. Уведене значення може бути рядком, числом або значенням зі списку.

У лістингу 6.29 наведено код програми "Діалогове вікно для введення даних".

Лістинг 6.29. Приклад
"Діалогове вікно для введення даних"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 18

import sys
from PyQt4 import QtGui, QtCore
```

```
    class InputDialog(QtGui.QWidget):
        def __init__(self, parent=None):
            QtGui.QWidget.__init__(self, parent)

            self.setGeometry(300, 300, 350, 80)
            self.setWindowTitle(self.trUtf8('Діалог вводу'))

            self.button = QtGui.QPushButton(self.trUtf8('Діалог'), self)
            self.button.setFocusPolicy(QtCore.Qt.NoFocus)

            self.button.move(20, 20)
            self.connect(self.button, QtCore.SIGNAL('clicked()'), self.
showDialog)
            self.setFocus()

            self.label = QtGui.QLineEdit(self)
            self.label.move(130, 22)

        def showDialog(self):
            text, ok = QtGui.QInputDialog.getText(self, 'Input Dialog',
self.trUtf8('Уведіть своє ім\'я:'))

            if ok:
                self.label.setText(unicode(text))

    app = QtGui.QApplication(sys.argv)
    icon = InputDialog()
    icon.show()
    app.exec_()
```

У прикладі створюється кнопка і текстове поле. При натисканні кнопки з'являється діалог для введення будь-якого значення, яке виводитиметься в текстове поле.

У 26-му рядку викликається діалог. Перший параметр функції – це назва вікна, а наступний параметр – повідомлення в діалоговому вікні. Функція **getText** повертає введений текст та логічне значення. Якщо в діалозі було натиснуто кнопку **Ok**, то логічне значення дорівнює **True**, інакше – **False**.

Діалогове вікно для вибору кольору. Клас **QColorDialog** дозволяє вибрати колір із палітри. У лістингу 6.30 наведено приклад програми "Діалог вибору кольору".

Лістинг 6.30. Приклад "Діалог вибору кольору"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 19

import sys
from PyQt4 import QtGui, QtCore

class ColorDialog(QtGui.QWidget):
    def __init__(self, parent=None):

        QtGui.QWidget.__init__(self, parent)

        color = QtGui.QColor(0, 0, 0)

        self.setGeometry(300, 300, 250, 180)
        self.setWindowTitle(self.trUtf8('Діалог для вибору кольору'))

        self.button = QtGui.QPushButton(self.trUtf8('Діалог'), self)
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)
        self.button.move(20, 20)

        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.
showDialog)
        self.setFocus()

        self.widget = QtGui.QWidget(self)
        self.widget.setStyleSheet("QWidget { background-color: %s }"
            % color.name())
        self.widget.setGeometry(130, 22, 100, 100)

    def showDialog(self):
        col = QtGui.QColorDialog.getColor()

        if col.isValid():
            self.widget.setStyleSheet("QWidget { background-
color: %s }"
                % col.name())

app = QtGui.QApplication(sys.argv)
cd = ColorDialog()
cd.show()
app.exec_()
```

У цьому прикладі при натисканні кнопки з'являтиметься діалогове вікно для вибору кольору. Після закриття діалогового вікна об'єкт **QWidget** буде залито обраним кольором.

У 31-му рядку викликається діалогове вікно.

Далі в 33–35 рядках значення кольору перевіряється на коректність та встановлюється як фон об'єкта **QWidget**.

Під час призначення стилю віджетам у *PyQt4* використовується CSS (каскадні таблиці стилів).

Діалогове вікно для вибору шрифту. Клас *QFontDialog* служить для вибору шрифту. У лістингу 6.31 наведено код програми "Діалог вибору шрифту".

Лістинг 6.31. Приклад "Діалог вибору шрифту"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 20

import sys
from PyQt4 import QtGui, QtCore

class FontDialog(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        hbox = QtGui.QHBoxLayout()

        self.setGeometry(300, 300, 250, 110)
        self.setWindowTitle(self.trUtf8('Діалог вибору шрифту'))

        button = QtGui.QPushButton(self.trUtf8('Діалог'), self)
        button.setFocusPolicy(QtCore.Qt.NoFocus)
        button.move(20, 20)

        hbox.addWidget(button)

        self.connect(button, QtCore.SIGNAL('clicked()'), self.
showDialog)

        self.label = QtGui.QLabel(self.trUtf8('Тест вибраного
шрифту'), self)
        self.label.move(130, 20)
```

```
        hbox.addWidget(self.label, 1)
        self.setLayout(hbox)

    def showDialog(self):
        font, ok = QtGui.QFontDialog.getFont()
        if ok:
            self.label.setFont(font)

app = QtGui.QApplication(sys.argv)
cd = FontDialog()
cd.show()
app.exec_()
```

У прикладі використано горизонтальну схему розташування, у яку поміщається кнопка та текстова мітка. При виборі шрифту текстова мітка може вийти за межі вікна. Щоб цього не відбулося, використано горизонтальну схему, яка автоматично вирівнює розміри вікна під розміри текстової мітки. У 32-му рядку викликається діалог для вибору шрифту. Функція повертає стиль шрифту та логічне значення (чи було натиснуто кнопку *Ok*).

Якщо кнопку *Ok* було натиснуто, то програма встановлює для текстової мітки обраний шрифт за допомогою функції **setFont**.

Діалогове вікно для вибору файлів або каталогів. Клас *QFileDialog* дозволяє користувачам вибрати файли або каталоги (лістинг 6.32).

Лістинг 6.32. Приклад "Діалог вибору файлів"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 21

import sys
from PyQt4 import QtGui, QtCore

class OpenFile(QtGui.QMainWindow):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)

        self.setGeometry(300, 300, 350, 300)
        self.setWindowTitle(self.trUtf8('Діалог вибору файлів'))
```

```
        self.textEdit = QtGui.QTextEdit()
        self.setCentralWidget(self.textEdit)
        self.statusBar()
        self.setFocus()

        exit = QtGui.QAction(QtGui.QIcon('open.png'), self.trUtf8
('Відкрити'), self)
        exit.setShortcut('Ctrl+O')
        exit.setStatusTip(self.trUtf8('Відкрити новий файл'))
         self.connect(exit, QtCore.SIGNAL('triggered()'), self.
showDialog)

        menubar = self.menuBar()
        file = menubar.addMenu(self.trUtf8('Файл'))
        file.addAction(exit)

      def showDialog(self):
        filename = QtGui.QFileDialog.getOpenFileName(self,
                self.trUtf8('Відкриття файла'), '/home')
         self.setWindowTitle(self.trUtf8("Проглядання файла")
+filename)
        file=open(filename)
        data = file.read()
        self.textEdit.setText(data)

    app = QtGui.QApplication(sys.argv)
    cd = OpenFile()
    cd.show()
    app.exec_()
```

Приклад містить рядок меню, статусний рядок та текстовий віджет, у який виводитиметься текст з обраного файла. У 30-му рядку викликається функція **QFileDialog.getOpenFileName**, яка відображає діалогове вікно. Перший параметр функції – це назва вікна діалогу, а другий параметр – робочий каталог.

Далі в 33–35 рядках відкривається обраний файл та виводиться його вміст у текстовий віджет.

6.3.7. Віджети в *PyQt4*

Віджети є основними блоками при побудові GUI-застосувань. Бібліотека *PyQt4* має широкий спектр різних віджетів: кнопки, прапорці, повзунки, комбіновані списки, таблиці тощо.

Прапорець (QCheckBox). Віджет **QCheckBox** за замовчуванням має два стани: вимкнений та увімкнений. Але є і третій стан – "частково увімкнений". Щоб додати стан "частково увімкнений" використовується функція **setTristate**, якій передається логічний аргумент, що вимикає/вмикає цей стан. При зміні стану прапорець (лістинг 6.33) посилає сигнал **stateChanged(int)**.

Лістинг 6.33. Приклад "Прапорець"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 22

import sys
from PyQt4 import QtGui, QtCore

class CheckBox(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle(self.trUtf8('Прапорець'))

        self.cb = QtGui.QCheckBox(self.trUtf8('Показати заголовки'), self)
        self.cb.setFocusPolicy(QtCore.Qt.NoFocus)
        self.cb.move(10, 10)
        self.cb.toggle();
        self.cb.setTristate(True)
        self.connect(self.cb, QtCore.SIGNAL('stateChanged(int)'), self.changeTitle)

    def changeTitle(self, value):
        if value==QtCore.Qt.Checked:
            self.setWindowTitle(self.trUtf8('Прапорець'))
        elif value==QtCore.Qt.PartiallyChecked:
            self.setWindowTitle(self.trUtf8('Частково'))
```

```
        else:
            self.setWindowTitle('')

    app = QtGui.QApplication(sys.argv)
    icon = CheckBox()
    icon.show()
    app.exec_()
```

У 16-му рядку вимикається фокус для прапорця. У 18–19 рядках установлюється стан прапорця "увімкнений" та додається "частково увімкнений" стан.

У функції **changeTitle** перевіряється стан прапорця та залежно від його стану змінюється заголовок вікна.

Стани прапорця:

- **Qt.Unchecked** – вимкнений;
- **Qt.PartiallyChecked** – частково увімкнений;
- **Qt.Checked** – увімкнений.

Кнопка-перемикач (QPushButton). Створимо кнопку (перемикач), що встановлюватиме колір віджета. Кнопка-перемикач (лістинг 6.34) має два стани: "натиснута" і "ненатиснута".

Лістинг 6.34. Приклад "Кнопка-перемикач"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 23

import sys
from PyQt4 import QtGui, QtCore

class ToggleButton(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.color = QtGui.QColor(0, 0, 0)

        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle(self.trUtf8('Кнопка-перемикач'))

        self.red = QtGui.QPushButton(self.trUtf8('Червоний'), self)
        self.red.setCheckable(True)
        self.red.move(10, 10)
        self.connect(self.red, QtCore.SIGNAL('clicked()'), self.setRed)
```

```
        self.green = QtGui.QPushButton(self.trUtf8('Зелений'), self)
        self.green.setCheckable(True)
        self.green.move(10, 60)
        self.connect(self.green, QtCore.SIGNAL('clicked()'), self.
setGreen)

        self.blue = QtGui.QPushButton(self.trUtf8('Синій'), self)
        self.blue.setCheckable(True)
        self.blue.move(10, 110)
        self.connect(self.blue, QtCore.SIGNAL('clicked()'), self.setBlue)

        self.square = QtGui.QWidget(self)
        self.square.setGeometry(150, 20, 100, 100)
        self.square.setStyleSheet("QWidget { background-color: %s }"
% self.color.name())
        QtGui.QApplication.setStyle(QtGui.QStyleFactory.create
('cleanlooks'))

    def setRed(self):
        if self.red.isChecked():
            self.color.setRed(255)
        else: self.color.setRed(0)

        self.square.setStyleSheet("QWidget { background-color: %s }"
% self.color.name())

    def setGreen(self):
        if self.green.isChecked():
            self.color.setGreen(255)
        else: self.color.setGreen(0)

        self.square.setStyleSheet("QWidget { background-color: %s }"
% self.color.name())

    def setBlue(self):
        if self.blue.isChecked():
            self.color.setBlue(255)
        else: self.color.setBlue(0)

        self.square.setStyleSheet("QWidget { background-color: %s }"
% self.color.name())
```

```
app = QtGui.QApplication(sys.argv)
tb = ToggleButton()
tb.show()
app.exec_()
```

За допомогою кнопок-перемикачів змішуються кольори: червоний, зелений, синій. Отриманий колір установлюється як колір фону віджета **QWidget**.

У 12-му рядку створюється змінна **color** для зберігання кольору. Конструктору передаються три аргументи, які визначають червоний, зелений і синій кольори.

У 18, 23 та 28 рядках лишаємо кнопку в стані "натиснута". Далі з'єднуються створені кнопки зі слотами, в яких встановлюватиметься колір.

У 32-му рядку створюється віджет, для якого буде встановлюватися колір фону. Для присвоєння кольору змінній **self.color** використовуються функції **setRed(int)** – червоний, **setGreen(int)** – зелений, **setBlue(int)** – синій.

Функція **isChecked** дозволяє довідатися, чи натиснуто кнопку. Колір фону встановлюється за допомогою CSS-стилів.

Повзунок (QSlider) та напис (QLabel). Повзунок дуже зручний при швидкому виборі значення, тобто, коли не потрібно вводити значення вручну. А напис використовується для виведення тексту або зображення. У лістингу 6.35 наведено код програми "Імітація регулювання звуку".

Лістинг 6.35. Приклад
"Імітація регулювання звуку"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 24

import sys
from PyQt4 import QtGui, QtCore

class SliderLabel(QtGui.QWidget):
   def __init__(self, parent=None):
      QtGui.QWidget.__init__(self, parent)
```

```
        self.setGeometry(300, 300, 200, 100)
        self.setWindowTitle(self.trUtf8('Регулювання звуку'))

        self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
        self.slider.setFocusPolicy(QtCore.Qt.NoFocus)
        self.slider.setGeometry(30, 40, 100, 30)
       self.connect(self.slider,  QtCore.SIGNAL('valueChanged(int)'),
self.changeValue)

        self.label = QtGui.QLabel(self)
        self.label.setPixmap(QtGui.QPixmap('./muted.png'))
        self.label.setGeometry(160, 40, 80, 30)

    def changeValue(self, value):
        if value == 0:
            self.label.setPixmap(QtGui.QPixmap('./muted.png'))
        elif value > 0 and value <= 30:
            self.label.setPixmap(QtGui.QPixmap('./low.png'))
        elif value > 30 and value < 80:
            self.label.setPixmap(QtGui.QPixmap('./medium.png'))
        else:
            self.label.setPixmap(QtGui.QPixmap('./high.png'))

app = QtGui.QApplication(sys.argv)
icon = SliderLabel()
icon.show()
app.exec_()
```

У прикладі імітується регулювання звуку. При зміні положення повзунка міняється іконка в написі.

У 15-му рядку створюється горизонтальний повзунок. Далі з'єднується сигнал зміни повзунка зі слотом **changeValue**.

За допомогою функції **setPixmap** в 21-му рядку встановлюється початкове зображення (вимкнений звук) напису.

У функції **changeValue** перевіряється позиція повзунка та залежно від цього встановлюється відповідна іконка напису.

Для визначення позиції використовується значення з сигналу **valueChanged(int)**. Хоча можна застосовувати метод **value**:

```
... pos = self.slider.value()
```

Індикатор виконання (QProgressBar). Віджет використовується для візуалізації виконання довгої операції. Індикатор виконання (лістинг 6.36) може бути горизонтальним та вертикальним. Віджету можна вказати мінімальне та максимальне значення (за замовчуванням віджет має значення від 0 до 99).

Лістинг 6.36. Приклад "Індикатор виконання"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 25

import sys
from PyQt4 import QtGui, QtCore

class ProgressBar(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle(self.trUtf8('Індикатор виконання'))

        self.vbox=QtGui.QVBoxLayout(self)
        self.pbar = QtGui.QProgressBar()
        self.vbox.addWidget(self.pbar)
        self.button = QtGui.QPushButton(self.trUtf8('Старт'))
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)
        self.vbox.addWidget(self.button)
        self.connect(self.button, QtCore.SIGNAL('clicked()'),
self.onStart)

        self.timer = QtCore.QBasicTimer()
        self.step = 0;

    def timerEvent(self, event):
        if self.step >= 100:
            self.timer.stop()
            return
        self.step += 1
        self.pbar.setValue(self.step)
```

```
    def onStart(self):
        if self.timer.isActive():
            self.timer.stop()
            self.button.setText(self.trUtf8('Старт'))
        else:
            self.timer.start(100, self)
            self.button.setText(self.trUtf8('Стоп'))

app = QtGui.QApplication(sys.argv)
icon = ProgressBar()
icon.show()
app.exec_()
```

У цьому прикладі використовується два віджети, це індикатор виконання та кнопка. Кнопка запускає або зупиняє рух індикатора.

У 15-му рядку створюється віджет "індикатор". Далі у 20-му рядку з'єднується сигнал натискання кнопки зі слотом **onStart**. У цьому прикладі використовується таймер для зміни "індикатора виконання". За допомогою класу **QBasicTimer** створюється об'єкт "таймер" у батьківському віджеті. Він ініціалізується у 22-му рядку.

У 25-му рядку перезаписується подія **timerEvent**, що викликається, коли таймер спрацьовує. Тепер у випадку, якщо таймер буде спрацьовувати, збільшуватиметься значення індикатора за допомогою функції **setValue(int)**. А коли значення індикатора дорівнюватиме 100, таймер вимикатиметься функцією **stop**. Функція **onStart** служить для увімкнення або вимикання заповнення індикатора. У 33-му рядку перевіряється, чи працює таймер. Якщо так, то вимикається таймер та на кнопці формується текст "Старт", якщо ні, то запускається таймер функцією **start** та формується текст "Стоп".

У функцію старту таймера **start(int, QObject)** потрібно передати час у мілісекундах, через який спрацьовуватиме таймер та об'єкт, у якому буде запущений таймер.

Календар (QCalendarWidget). За допомогою цього віджета можна легко й інтуїтивно вибрати потрібну дату. У лістингу 6.37 наведено код програми "Календар".

Лістинг 6.37. Приклад "Календар"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 26

import sys
from PyQt4 import QtGui, QtCore

class Calendar(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 130)
        self.setWindowTitle(self.trUtf8('Календар'))
        self.vbox = QtGui.QVBoxLayout(self)
        self.cal = QtGui.QCalendarWidget()
        self.cal.setGridVisible(True)
        self.cal.move(20, 20)
        self.connect(self.cal, QtCore.SIGNAL('selectionChanged()'),
self.showDate)
        self.vbox.addWidget(self.cal)

        self.label = QtGui.QLabel()
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))
        self.label.move(130, 260)
        self.vbox.addWidget(self.label)

    def showDate(self):
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))

app = QtGui.QApplication(sys.argv)
icon = Calendar()
icon.show()
app.exec_()
```

У прикладі використовуються віджети "календар" та "напис". При виборі дати в календарі вона відображатиметься в написі.

У 18-му рядку з'єднується сигнал вибору дати **selectionChanged** зі слотом **showDate**.

Щоб довідатися, яку дату вибрано в календарі, потрібно використовувати функцію **selectedDate**, що повертає значення вибраної дати. Тип дати буде **QDate**.

Для переведення значення дати з типу **QDate** у тип, із яким працює *Python*, використовується функція **toPyDate**.

Контрольні запитання

1. Перерахуйте основні пакети для створення графічного інтерфейсу користувача, що застосовуються з *Python*.
2. Чому пакети графічного інтерфейсу *Python* – це в основному обгортки для бібліотек, що написані іншими мовами програмування?
3. Перерахуйте основні віджети пакета *Tkinter*.
4. Опишіть особливості обробки подій у пакеті *Tkinter*.
5. Поясніть концепцію слотів і сигналів, яка використовується в бібліотеці *PyQt4*.
6. Перерахуйте основні віджети бібліотеки *Qt*.
7. Як буде відображатися віджет, для якого при створенні не вказано батька?
8. Перерахуйте схеми розташування, що використовуються в пакеті *PyQt4*, та поясніть особливості їхнього застосування.
9. Екземпляр якого класу має обов'язково бути в програмі, що використовує бібліотеку *PyQt*4?

Розділ 7
ОБРОБКА ТЕКСТУ

Під **обробкою текстів** маємо на увазі аналіз, перетворення, пошук, створення текстової інформації. Здебільшого робота з текстами не розглядатиметься надто докладно й глибоко. Крім того, тут нема відомостей щодо обробки текстів за допомогою текстових процесорів та редакторів, хоча деякі з них (наприклад, *Cooledit*) дають можливість писати макрокоманди на *Python*.

Зазначимо, що для *Python* створено також модулі для роботи з природними мовами, а також для лінгвістичних досліджень. Добрим навчальним прикладом може служити *nltk* (*Natural Language Toolkit*).

Рекомендуємо для використання проект *PyParsing* (сайт: http://pyparsing.sourceforge.net), за допомогою якого можна організувати обробку тексту згідно з визначеною граматикою.

7.1. Основні операції

Python дуже багатий на операції з рядковими об'єктами. Рядки можна визначити у програмі за допомогою **рядкових літералів.** Літерали записуються з використанням апострофів ', лапок " або цих самих символів, узятих тричі. Усередині літералів зворотна коса риска має спеціальне значення. Вона служить для введення спеціальних символів та для визначення символів через коди. Якщо перед рядковим літералом поставлено *r*, зворотна коса риска не має спеціального значення (*r* від англійського слова *raw*, рядок вказуєтья "як є"). Unicode-літерали вказуються із префіксом **u**. *Наведемо кілька прикладів:*

```
s1 = "рядок 1"
s2 = r'\1\2'
s3 = """apple\ntree"""# \n - символ переведення рядка
s4 = """apple
tree"""# рядок у потроєних лапках може мати всередині переведення рядків
s5 = '\x73\65'
u1 = u"Unicode literal"
u2 = u'\u0410\u0434\u0440\u0435\u0441'
```

Примітка. Зворотна коса риска не повинна бути останнім символом у літералі, тобто, **"str\"** викличе синтаксичну помилку.

Вказування кодування дозволяє використовувати в Unicode-літералах вказаний на початку програми тип кодування. Якщо тип кодування не вказано, можна користуватися тільки кодами символів, визначеними через зворотну косу риску.

7.1.1. Виділення підрядків

Найперше розглянемо виділення підрядків за допомогою зрізів. Вираз **s[i:j]** подібно до зрізу, що використовується для списків, вирізає символи з рядка **s**, починаючи з **i** по **j-1** включно:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s[8:] # з індексу 8 до кінця рядка
'18.4 C at 4 pm'
>>> s[8:12] # символи з індексами 8, 9, 10 і 11
'18.4'
```

7.1.2. Пошук підрядків

Виклик **s.find(s1)** у разі виявлення в рядку **s** підрядка **s1** повертає індекс, із якого починається підрядок. У разі, якщо підрядок не знайдено, повертається значення –1:

```
>>> s.find('Berlin') # де починається 'Berlin'?
0
>>> s.find('pm')
20
>>> s.find('Oslo') # не знайде
-1
```

Іноді потрібно лише перевірити наявність такого підрядка в рядку, для цього, як і раніше, використовується оператор **in**:

```
>>> 'Berlin' in s
True
>>> 'Oslo' in s
False
```

Типовий приклад використання такої конструкції:

```
>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
```

...
C found

Є ще два інші, специфічні для рядків, методи: **startswith** (починається з) та **endswith** (закінчується), поведінка яких зрозуміла з назви:

```
>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False
```

7.1.3. Заміщення

Виклик **s.replace(s1, s2)** заміняє в рядку **s** підрядок **s1** на **s2** скрізь, де зустрінеться підрядок **s1**:

```
>>> s.replace(' ', '_')
'Berlin:_18.4_C_at_4_pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

Варіант попереднього прикладу, у якому поєднується кілька операцій, ілюструє заміну назви міста перед двокрапкою:

```
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

7.1.4. Розбиття рядків

Виклик **s.split()** розбиває рядок *s* на слова, розділені пробілами, табуляціями та символами нового рядка:

```
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

Розбиття рядка на слова, розділені символом або текстом **t**, здійснюється передаванням цього тексту як аргументу **s.split(t)**:

```
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
```

За допомогою **s.splitlines()** текст можна розбити на рядки, що дуже зручно, коли було завантажено весь вміст файла у вигляді одного рядка і потрібно розкласти його у список рядків:

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
```

2nd line
3rd line
>>> t.splitlines()
['1st line', '2nd line', '3rd line']

7.1.5. Верхній та нижній регістри

Перевести всі символи в нижній регістр дозволяє **s.lower()**, у верхній регістр – **s.upper()**.

>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'

7.1.6. Рядки – це константи

Хоча багато методів рядків та індексація схожі на роботу зі списками, рядки не можна змінювати. Тобто, всі вище перераховані активні методи дають новий рядок, не змінюючи той, із яким вони працюють. Тому будь-які дії, що намагаються змінити рядок або символи в ньому, зазнають фіаско:

>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment

Якщо ж справді необхідно замінити, *наприклад,* символ **s[18]**, то доведеться створити новий рядок, приміром за допомогою зрізів:

>>> s[:18] + '5' + s[19:]
'Berlin: 18.4 C at 5 pm'

7.1.7. Перевірка на цифри

Для визначення наявності в рядку тільки символів цифр, існує простий метод **isdigit**:

>>> '214'.isdigit()
True
>>> ' 214 '.isdigit()
False
>>> '2.14'.isdigit()
False

7.1.8. Перевірка на пробільні символи

Можна також перевірити чи містяться в рядку пробіли, символи табуляції та порожні рядки (все це й називається пробільним символами) за допомогою **isspace**:

```
>>> ' '.isspace()# пробіли
True
>>> '\n'.isspace() # символ нового рядка
True
>>> '\t'.isspace() # табуляція
True
>>> ''.isspace() # порожній рядок
False
```

Метод **isspace** зручний для виявлення у файлі порожніх рядків. Альтернативний спосіб – використати метод **strip**, що відрізає зайві пробільні символи по краях рядка:

```
>>> line = ' \n'
>>> line.strip() == ''# чи залишився тільки порожній рядок
True
>>> s = 'text with leading/trailing space \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip()# left strip
'text with leading/trailing space \n'
>>> s.rstrip()# right strip
'text with leading/trailing space'
```

7.1.9. Об'єднання рядків

Протилежний методу **split** метод **join** об'єднує рядки зі списку в один рядок. Використання методу зрозуміле з прикладу й без пояснень:

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> t = ', '.join(strings)
>>> t
'Newton, Secant, Bisection'
```

Хоча методи **split** та **join** прямо протилежні, разом вони можуть працювати дуже злагоджено. *Наприклад,* потрібно вирізати з рядка перші два слова:

```
>>> line = 'This is a line of words separated by space'
```

```
>>> words = line.split()
>>> line2 = ' '.join(words[2:])
>>> line2
'a line of words separated by space'
```

7.1.10. Основні операції з рядками

У табл. 7.1 наведено деякі найбільш уживані методи об'єктів-рядків та Unicode-об'єктів.

Таблиця 7.1. **Методи об'єктів-рядків**

Метод	Опис
center(w)	Центрує рядок у поле завдовжки *w*
count(sub)	Повертає число входжень рядка *sub* у рядок
encode([enc[, errors]])	Повертає рядок у кодуванні *enc*. Параметр *errors* може набувати значення "*strict*" (за замовчуванням), "*ignore*", "*replace*" або "*xmlcharrefreplace*"
endswith(suffix)	Чи закінчується рядок на *suffix*?
expandtabs([tabsize])	Заміняє символи табуляції на пробіли. За замовчуванням *tabsize*=8
find(sub [,start [,end]])	Повертає найменший індекс, із якого починається входження підрядка *sub* у рядок. Параметри *start* та *end* обмежують пошук вікном *start:end*, але повертається індекс, що відповідає вихідному рядку. Якщо підрядок не знайдено, повертається –1
index(sub[, start[, end]])	Аналогічно **find()**, але генерує виняткову ситуацію **ValueError** у разі невдачі
alnum()	Повертає *True*, якщо рядок містить тільки літери та цифри, і має ненульову довжину. Інакше – *False*
Isalpha()	Повертає *True*, якщо рядок містить тільки букви та має ненульову довжину
isdecimal()	Повертає *True*, якщо рядок містить тільки десяткові знаки (тільки для рядків Unicode) та має ненульову довжину
isdigit()	Повертає *True*, якщо містить тільки цифри та має ненульову довжину

Закінчення табл. 7.1

Метод	Опис
islower()	Повертає *True*, якщо всі букви малі (і їх більше однієї), інакше – *False*
isnumeric()	Повертає *True*, якщо в рядку тільки числові знаки (тільки для Unicode)
isspace()	Повертає *True*, якщо рядок складається тільки із символів пробілу. **Увага!** Для порожнього рядка повертається *False*
join(seq)	З'єднання рядків із послідовності *seq* через роздільник, указаний рядком
lower()	Робить усі літери в рядку малими
lstrip()	Видаляє символи пробілу ліворуч
replace(old, new[, n])	Повертає копію рядка, у якому підрядки *old* замінено на *new*. Якщо вказано параметр *n*, то заміняються тільки перші *n* входжень
rstrip()	Видаляє символи пробілу праворуч
split([sep[, n]])	Повертає список підрядків, що отримуються розбиттям рядка *a* роздільником *sep*. Параметр *n* визначає максимальну кількість розбиттів (ліворуч)
startswith(prefix)	Чи починається рядок із підрядка *prefix*?
strip()	Видаляє пробільні символи на початку й у кінці рядка
translate(table)	Виконує перетворення за допомогою таблиці перекодування *table*, що містить словник для перекладу кодів у коди (або в *None*, щоб видалити символ). Для Unicode-рядків
translate(table[, dc])	Те саме, для звичайних рядків. Замість словника – рядок перекодування на 256 символів, який можна сформувати за допомогою функції **string.maketrans()**. Необов'язковий параметр *dc* задає рядок із символами, які потрібно видалити
upper()	Робить усі літери в рядку великими

У наступному прикладі використовуються методи **split()** та **join()** для розбиття рядка в список (по роздільниках) та повторне об'єднання списку рядків у рядок:

```
>>> s = "This is an example."
>>> lst = s.split(" ")
>>> print lst
['This', 'is', 'an', 'example.']
>>> s2 = "\n".join(lst)
>>> print s2
This
is
an
example.
```

Щоб перевірити чи закінчується рядок певним сполученням літер, можна використати метод **endswith()**:

```
>>> filenames = ["file.txt", "image.jpg", "str.txt"]
>>> for fn in filenames:
...   if fn.lower().endswith(".txt"):
...     print fn
...
file.txt
str.txt
```

Шукати в рядку можна за методом **find()**. Наступна програма виводить всі функції, визначені в модулі оператором **def**:

```
import string
text = open(string.__file__[:-1]).read()
start = 0
while 1:
  found = text.find("def ", start)
  if found == -1:
    break
  print text[found:found + 60].split("(")[0]
  start = found + 1
```

Важливим для перетворення текстової інформації є метод **replace()**:

```
>>> a = "Це текст , у якому є неправильно поставлені коми"
>>> b = a.replace(" ,", ",")
>>> print b
Це текст, у якому є неправильно поставлені коми.
```

7.1.11. Рекомендації з ефективності

При роботі з дуже довгими рядками або великою кількістю рядків використані операції можуть по-різному впливати на швидкодію програми.

Наприклад, не рекомендується багаторазово використовувати операцію конкатенації для склеювання великої кількості рядків в один. Ліпше накопичити рядки в списку, а потім за допомогою **join()** об'єднати в один рядок:

```
>>> a = ""
>>> for i in xrange(1000):
...   a += str(i)       # неефективно!
...
>>> a = "".join([str(i) for i in xrange(1000)])  # ефективніше
```

Звичайно, якщо рядок потім обробляється, можна застосовувати ітератори, які дозволять звести використання пам'яті до мінімуму.

7.1.12. Приклади розв'язування реальних задач

Приклад 7.1. Читання пар чисел

Нехай є файл, що складається з пар дійсних чисел, записаних у форматі (*a*, *b*). Це подання часто використовується, щоб указувати на площині точки, вектори, комплексні числа. Нехай файл **read_pairs1.dat** містить такі пари:

```
(1.3,0) (-1,2)(3,-1.5)
(0,1) (1,0) (1,1)
(0,-0.01) (10.5,-1) (2.5,-2.5)
```

Потрібно зчитати ці пари чисел у вкладений список, що складається із поданих як кортежі пар дійсних чисел. Для виконання такого завдання необхідно рядок за рядком прочитати файл, кожний рядок розбити на слова по пробільних символах. Далі в словах треба позбутися дужок, розбити слово по символу комою та конвертувати слова в числа. Наведемо реалізацію цього алгоритму мовою *Python*:

```
lines = open('read_pairs1.dat', 'r').readlines()
```

```
pairs = []   # список для пар чисел (n1, n2)
for line in lines:
   words = line.split()
   for word in words:
      word = word[1:-1]  # відрізаємо дужки
      n1, n2 = word.split(',')
      n1 = float(n1);  n2 = float(n2)
      pair = (n1, n2)
      pairs.append(pair)  # додати кортеж у список

infile.close()
import pprint
pprint.pprint(pairs)
```

Після запуску програми маємо список:

```
[(1.3, 0.0),
 (-1.0, 2.0),
 (3.0, -1.5),
 (0.0, 1.0),
 (1.0, 0.0),
 (1.0, 1.0),
 (0.0, -0.01),
 (10.5, -1.0),
 (2.5, -2.5)]
```

Але це завдання так вирішується у випадку, якщо між дужками немає пробілів. Якщо ж пробіли є, то **split**, розбиваючи рядок по пробілах, розіб'є й наші пари на *'(a'* та *'b)'*. Що потрібно тоді зробити?

Можна спочатку видалити в кожному рядку пробіли, а потім звернути увагу, що наші пари розділено сполученнями дужок)(, крім яких у рядку залишаються тільки перша й остання дужки, які можна легко обрізати. Таким чином, друга версія програми матиме такий вигляд:

```
infile = open('read_pairs2.dat', 'r')   # вже файл із пробілами
lines = infile.readlines()

pairs = []
for line in lines:
```

```
    line = line.strip()  # видаляємо пробільні символи на кінцях рядків
    line = line.replace(' ', '')  # видаляємо всі пробіли
    words = line.split(')(')      # розбиваємо по )(
    # обрізаємо першу та останню дужки на кінцях:
    words[0] = words[0][1:]      # (-1,3  ->  -1,3
    words[-1] = words[-1][:-1]   # 8.5,9) ->  8.5,9
    for word in words:
      n1, n2 = word.split(',')
      n1 = float(n1);  n2 = float(n2)
      pair = (n1, n2)
      pairs.append(pair)

infile.close()
import pprint
pprint.pprint(pairs)
```

Третій можливий варіант – запис у вигляді пар чисел, розділених комами:

```
(1, 3.0),(-1,2), (3, -1.5)
(0, 1),(1,0),(1, 1)
```

Зверніть увагу на те, що таке подання тексту файла вже достатньо близьке до запису списку кортежів. Залишається тільки додати квадратні дужки та кому в кінці кожного рядка:

```
[(1, 3.0),(-1,2), (3, -1.5),
 (0, 1),(1,0),(1, 1)]
```

А коли запис чимось схожий на код *Python*, варто подумати про чарівну функцію **eval**, ту саму, що приймає рядковий аргумент та виконує код. Цей запис матиме такий вигляд:

```
infile = open('read_pairs3.dat', 'r')
listtext = '['
for line in infile:
  # додати рядок, без переходу на новий (line[:-1]), плюс кома:
  listtext += line[:-1] + ', '
infile.close()
listtext = listtext + ']'
pairs = eval(listtext)
```

infile.close()
import pprint
pprint.pprint(pairs)

Як видно зі стислості та простоти коду, це взагалі досить непогана ідея – використовувати форматування, найближче до синтаксису *Python*, що дозволяє обробляти такі дані за допомогою **eval** або **exec** прямо "на льоту".

Приклад 7.2. Зчитування координат

Нехай є файл, у якому записано координати у тривимірному просторі, формат такий:

x=-1.345y= 0.1112z= 9.1928
x=-1.231y=-0.1251z= 1001.2
x= 0.100y= 1.4344E+6 z=-1.0100
x= 0.200y= 0.0012z=-1.3423E+4
x= 1.5E+5 y=-0.7666z= 1027

Потрібно зчитати ці координати у вкладений список потрійних кортежів, де кожний кортеж відповідає за свою координату. Після цього перетворити такий список у масив, придатний для обчислень у **NumPy**.

Зазначимо, що форматування файла містить таку особливість, як пробіл після знака рівності у разі додатних чисел. Для від'ємних чисел його немає і це накладає на користувача певні обов'язки. Нижче розглянуто на цьому прикладі 7.2 різні підходи до розв'язання одного й того самого завдання.

Розв'язання 1. Вирізання підрядків

Форматування файла достатньо одноманітне: кожний стовпець виділяється тим, що містить *x*=, *y*=, *z*=. При цьому стовпці вирівняно по лівому краю, значить текст, що їм відповідає, у кожному рядку починається з тих самих індексів:

x_start = 2
y_start = 16
z_start = 31

Далі підрядки, що стосуються самих чисел, можна одержати за допомогою рядкових зрізів:

x = line[x_start+2:y_start]

```
y = line[y_start+2:z_start]
z = line[z_start+2:]
```

А далі потрібно лише конвертувати рядки в числа, додати у список та перетворити його в масив:

```
infile = open('xyz.dat', 'r')
coor = []  # список кортежів (x,y,z)
for line in infile:
   x_start = 2
   y_start = 16
   z_start = 31
   x = line[x_start+2:y_start]
   y = line[y_start+2:z_start]
   z = line[z_start+2:]
   print 'debug: x="%s", y="%s", z="%s"' % (x,y,z)
   coor.append((float(x), float(y), float(z)))
infile.close()

from numpy import *
coor = array(coor)
print coor.shape, coor
```

Розв'язання 2. Пошук замість підрахунку

Проблема попереднього прикладу в його конкретиці. Варто змінити позицію тексту про координату, як програма перестане працювати зовсім, або, що ще гірше, неправильно вирізатиме числа. Крім того, узагалі підраховувати індекс символів важка справа для програмістів, тому краще їх просто знайти. Звідси, поліпшений код у цьому місці стане таким:

```
x_start = line.find('x=')
y_start = line.find('y=')
z_start = line.find('z=')
```

Розв'язання 3. Розбиття підрядків

Розбиття рядків – це потужний інструмент, який можна використати і в наведеному випадку. Розбивка рядків за символом рівності дасть у першому рядку слова:

```
['x', '-1.345 y',' 0.1112 z',' 9.1928']
```

Далі, потрібно відкинути перше слово, у наступних – відкинути крайній символ – літеру. Останнє слово залишається таким, як є. І в такий спосіб необхідно обробити всі рядки. Як результат – дуже короткий зрозумілий код:

```
infile = open('xyz.dat', 'r')
coor = []
for line in infile:
   words = line.split('=')
   x = float(words[1][:-1])
   y = float(words[2][:-1])
   z = float(words[3])
   coor.append((x, y, z))
infile.close()

from numpy import *
coor = array(coor)
print coor.shape, coor
```

7.2. Кодування *Python*-програми

Для того, щоб Unicode-літерали в *Python*-програмі сприймалися інтерпретатором правильно, необхідно вказати кодування на початку програми, записавши в першому або другому рядку приблизно таке (для Unix/Linux):

```
# -*- coding: koi8-r -*-
```

або (під Windows):

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

Можуть бути й інші варіанти:

```
# -*- coding: latin-1 -*-
# -*- coding: utf-8 -*-
# -*- coding: mac-cyrillic -*-
# -*- coding: iso8859-5 -*-
```

Повний перелік кодувань (та їхніх псевдонімів):

```
>>> import encodings.aliases
>>> print encodings.aliases.aliases
{'iso_ir_6': 'ascii', 'maccyrillic': 'mac_cyrillic',
'iso_celtic': 'iso8859_14', 'ebcdic_cp_wt': 'cp037',
'ibm500': 'cp500', ...
```

Якщо кодування не вказано, то вважається, що використовується **us-ascii**. При цьому інтерпретатор *Python* видаватиме попередження при запуску модуля:

sys:1: DeprecationWarning: Non-ASCII character '\xf0' in file example.py
on line 2, but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details

7.3. Модулі для роботи з рядками

7.3.1. Модуль string

Перш, ніж у рядків з'явилися методи, для операцій над рядками використовувався модуль **string**. Наведений приклад демонструє, як замість функції зі **string** використовувати метод (до речі, другий спосіб є ефективнішим):

```
>>> import string
>>> s = "one,two,three"
>>> print string.split(s, ",")
['one', 'two', 'three']
>>> print s.split(",")
['one', 'two', 'three']
```

У версії *Python 3.0* функції, які доступні через методи, більше не дублюватимуться в модулі **string**.

У *Python 2.4* з'явилася альтернатива використанню операції форматування: клас **Template.**

Приклад:

```
>>> import string
>>> tpl = string.Template("$a + $b = ${c}")
>>> a = 2
>>> b = 3
>>> c = a + b
>>> print tpl.substitute(vars())
2 + 3 = 5
>>> del c# видаляється ім'я c
>>> print tpl.safe_substitute(vars())
2 + 3 = $c
```

>>> print tpl.substitute(vars(), c=a+b)
2 + 3 = 5
>>> print tpl.substitute(vars())
Traceback (most recent call last):
File "/home/rnd/tmp/Python-2.4b2/Lib/string.py", line 172, in substitute
return self.pattern.sub(convert, self.template)
File "/home/rnd/tmp/Python-2.4b2/Lib/string.py", line 162, in convert
val = mapping[named]
KeyError: 'c'

Об'єкт-шаблон має два основних методи: **substitute()** та **safe_substitute()**. Значення для підстановки в шаблон беруться зі словника (**vars()** містить словник зі значеннями змінних) або з іменованих фактичних параметрів. Якщо є неоднозначність у визначенні ключа, можна використовувати фігурні дужки при написанні ключа в шаблоні.

7.3.2. Модуль StringIO

У деяких випадках бажано працювати з рядком як із файлом. Модуль **StringIO** надає саме таку можливість.

Відкриття *"файла"* виконується викликом **StringIO()**. При виклику без аргументу створюється новий *"файл"*, якщо вказати рядок як аргумент – *"файл"* відкривається для читання:

import StringIO
my_string = "1234567890"
f1 = StringIO.StringIO()
f2 = StringIO.StringIO(my_string)

Далі з файлами **f1** та **f2** можна працювати як зі звичайними файловими об'єктами.

Для одержання вмісту такого файла у вигляді рядка застосовується метод **getvalue()**:

f1.getvalue()

Протилежний варіант (подання файла на диску як рядка) можна реалізувати на платформах Unix та Windows із використанням модуля **mmap**. Тут цей модуль розглядатися не буде.

7.3.3. Модуль difflib

Для приблизного порівняння двох рядків у стандартній бібліотеці передбачено модуль **difflib**.

Функція **difflib.get_close_matches()** дозволяє виділити *n* близьких рядків до вказаного рядка:

get_close_matches(word, possibilities, n=3, cutoff=0.6)

тут **word** – рядок, до якого шукаються близькі рядки, **possibilities** – список можливих варіантів, **n** – необхідна кількість найближчих рядків, **cutoff** – коефіцієнт (із діапазону [0, 1]) необхідного рівня збігу рядків. Рядки, які при порівнянні з **word** дають менше значення, ігноруються.

Наступний приклад показує функцію **difflib.get_close_matches()** у дії:

```
>>> import unicodedata
>>> names = [unicodedata.name(unicode(chr(i))) for i in range(40, 127)]
>>> print difflib.get_close_matches("LEFT BRACKET", names)
['LEFT CURLY BRACKET', 'LEFT SQUARE BRACKET']
```

У списку names – назви Unicode-символів з ASCII-кодами від 40 до 127.

7.4. Регулярні вирази

Розглянутих стандартних можливостей для роботи з текстом часто не вистачає для розв'язання певних задач. Наприклад, у методах **find()** та **replace()** указується всього один рядок. У реальних завданнях така однозначність зустрічається досить рідко, частіше потрібно знайти або замінити рядки, що відповідають деякому шаблону.

Регулярні вирази (regular expressions) описують безліч рядків, використовуючи спеціальну мову, яку зараз розглянемо детальніше. Рядок, у якому зберігається регулярний вираз, називатимемо шаблоном.

Для роботи з регулярними виразами у *Python* використовується модуль **re**. У наступному прикладі регулярний вираз допомагає виділити з тексту всі числа:

```
>>> import re
>>> pattern = r"[0-9]+"
>>> number_re = re.compile(pattern)
>>> number_re.findall("122 234 65435")
['122', '234', '65435']
```

У цьому прикладі шаблон **pattern** описує безліч рядків, які складаються з одного або більше символів з набору "0", "1",..., "9". Функція **re.compile()** компілює шаблон у спеціальний Regex-об'єкт, що має кілька методів, у тому числі метод **findall()** для отримання списку всіх входжень рядків, що задовольняють шаблон, у вказаний рядок.

Те саме можна зробити й так:

```
>>> import re
>>> re.findall(r"[0-9]+", "122 234 65435")
['122', '234', '65435']
```

Попередня компіляція шаблона є бажанішою при його частому використанні, особливо в циклі.

Примітка. Щоб визначити шаблон використано неопрацьований рядок. У наведеному прикладі він не потрібен, але загалом ліпше записувати рядкові літерали саме так, щоб виключити вплив спеціальних послідовностей, записуваних через зворотну косу риску.

7.4.1. Синтаксис регулярного виразу

Синтаксис регулярних виразів у *Python* майже такий, як у *Perl*, *grep* та деяких інших інструментах. Частина символів (в основному букви та цифри) позначають самі себе. Рядок **задовольняє (відповідає)** шаблону, якщо він входить у множину рядків, які цей шаблон описує.

Тут також зазначимо, що різні операції використовують шаблон по-різному. Так, **search()** шукає перше входження рядка, що задовольняє шаблону, у вказаному рядку, а **match()** вимагає, щоб рядок задовольняв шаблон із самого початку.

Символи, які мають спеціальне значення в записі регулярних виразів, наведено в табл. 7.2.

Таблиця 7.2. **Символи спеціального призначення в записі регулярних виразів**

Символ	Значення в регулярному виразі
"."	Будь-який символ
"^"	Початок рядка
"$"	Кінець рядка
"*"	Повторення фрагмента нуль або більше раз (жадібний варіант)
"+"	Повторення фрагмента один або більше раз (жадібний варіант)
"?"	Повторення фрагмента нуль або один раз
"{*m,n*}"	Повторення попереднього фрагмента від *m* до *n* раз включно (жадібний варіант)
"[...]"	Будь-який символ з набору в дужках. Можна визначати діапазони символів із кодами, що йдуть підряд, наприклад: *a–z*
"[^...]"	Будь-який символ не з набору в дужках
"\"	Зворотна коса риска скасовує спеціальне значення наступного за нею символу
"|"	Фрагмент праворуч або фрагмент ліворуч
"*?"	Повторення фрагмента нуль або більше раз (нежадібний варіант)
"+?"	Повторення фрагмента один або більше раз (нежадібний варіант)
"{*m,n*}?"	Повторення попереднього фрагмента від *m* до *n* раз включно (нежадібний варіант)

Якщо *A* та *B* – регулярні вирази, то їхня конкатенація *AB* є новим регулярним виразом, причому конкатенація рядків *a* та *b* задовольнятиме *AB*, якщо *a* задовольняє *A* і *b* задовольняє *B*. Можна вважати, що конкатенація – основний спосіб складання регулярних виразів.

Дужки, описані нижче, використовуються для визначення пріоритетів та виділення груп (фрагментів тексту, які потім можна отримати за номером із словника, і навіть послатися в тому самому регулярному виразі).

Алгоритм, що зіставляє рядки з регулярним виразом, перевіряє відповідність того або іншого фрагмента рядка регулярному виразу. Наприклад, рядок *"a"* відповідає регулярному виразу *"*[*a*–*z*]*"*, рядок *"fruit"* відповідає *"fruit*|*vegetable"*, а от рядок *"apple"* не відповідає шаблону *"pineapple"*.

У табл. 7.3 замість *регвир* можна записати регулярний вираз, замість ім'я – ідентифікатор, а *прапорці* буде розглянуто нижче.

Таблиця 7.3. **Дужки для визначення пріоритетів та виділення груп**

Позначення	Опис
"(регвир)"	Виокремлює регулярний вираз у дужках та виділяє групу
"(?:регвир)"	Виокремлює регулярний вираз у дужках без виділення групи
"(?=регвир)"	*Погляд вперед*: рядок має відповідати вказаному регулярному виразу, але подальше зіставлення з шаблоном почнеться з того самого місця
"(?!регвир)"	Те саме, але із запереченням відповідності
"(?<=регвир)"	*Погляд назад*: рядок перед поточною позицією має відповідати *регвир*. Параметр *регвир* повинен бути фіксованої довжини (тобто, без "+" та "*")
"(?<!регвир)"	Те саме, але із запереченням відповідності
"(?P<ім'я>регвир)"	Виділяє іменовану групу з іменем *ім'я*
"(?P=ім'я)"	Точно відповідає виділеній раніше іменованій групі з іменем *ім'я*
"(?#регвир)"	Коментар (ігнорується)
"(?(ім'я)рв1|рв2)"	Якщо група з номером або іменем *ім'я* виявилася визначеною, результатом буде зіставлення із *рв1*, інакше – із *рв2*. Частини *|рв2* може не бути
"(?прапор)"	Визначає прапорець для всього цього регулярного виразу. Прапорці необхідно визначати на початку шаблона

У табл. 7.4 описано спеціальні послідовності, що використовують зворотну косу риску.

Таблиця 7.4. **Спеціальні послідовності, що використовують зворотну косу риску**

Послідовність	Опис
"\1" – "\9"	Група з визначеним номером. Групи нумеруються, починаючи з 1
"\A"	Проміжок перед початком усього рядка (майже аналогічно *"^"*)
"\Z"	Проміжок перед кінцем усього рядка (майже аналогічно *"$"*)
"\b"	Проміжок між символами перед словом або після нього
"\B"	Навпаки, не відповідає проміжку між символами на границі слова
"\d"	Цифра. Аналогічно "[0–9]"
"\s"	Будь-який пробільний символ. Аналогічно *"[\t\n\r\f\v]"*
"\S"	Будь-який символ крім пробільних. Аналогічно *"[^\t\n\r\f\v]*
"\w"	Будь-яка цифра або буква (залежить від прапорця LOCALE)
"\W"	Будь-який символ, що не є цифрою або буквою (залежить від прапорця LOCALE)

Прапорці, що використовуються з регулярними виразами:

- **"(?i)", re.I, re.IGNORECASE** – зіставлення проводиться без урахування регістра букв;
- **"(?L)", re.L, re.LOCALE** – впливає на визначення букви у *"\w", "\W", "\b", "\B"* залежно від поточного середовища (*locale*);
- **"(?m)", re.M, re.MULTILINE** – якщо цей прапорець указано, **"^"** та **"$"** відповідають початку та кінцю будь-якого рядка;
- **"(?s)", re.S, re.DOTALL** – якщо вказано, **"."** відповідає також і символу кінця рядка **"\n";**
- **"(?x)", re.X, re.VERBOSE** – якщо вказано, то пробільні символи, які не екрановано в шаблоні зворотною косою рискою, не є частиною шаблона, а все, що розташовано після символу **"#",** – коментарі. Дозволяє записувати регулярний вираз у кілька рядків для поліпшення його читабельності та запису коментарів;
- **"(?u)", re.U, re.UNICODE** – у шаблоні й у рядку використовується Unicode.

7.4.2. Методи шаблона-об'єкта

У результаті успішної компіляції шаблона функцією **re.compile()** створюється шаблон-об'єкт (він називається SRE_Pattern), що має кілька методів, деякі з них буде розглянуто. Подробиці й інформація про додаткові аргументи міститься в документації з *Python*:

- **match(s)**

зіставляє рядок *s* із шаблоном, повертаючи в разі вдалого зіставлення об'єкт із результатом порівняння (об'єкт SRE_Match). У разі невдачі повертає **None**. Зіставлення починається від початку рядка;

search(s)

аналогічний виклику **match(s)**, але шукає підхідний підрядок в усьому рядку *s*;

- **split(s[, maxsplit=0])**

розбиває рядок на підрядки, розділені підрядками, які визначено шаблоном. Якщо в шаблоні виділено групи, вони потраплять у вихідний список, перемежовуючись із підрядками між роздільниками. Якщо вказано **maxsplit**, буде зроблено не більше **maxsplit** розбиттів;

- **findall(s)**

шукає всі підрядки *s*, що не перекриваються, які задовольняють шаблон;

- **finditer(s)**

повертає за об'єктами ітератор із результатами порівняння для всіх підрядків, що не перекриваються, які задовольняють шаблон;

- **sub(repl, s)**

заміняє в рядку *s* усі (або тільки **count**, якщо його вказано) входження підрядків, що не перекриваються, які задовольняють шаблон, на рядок, указаний параметром **repl**. Як **repl** може виступати рядок або функція. Повертає рядок із виконаними замінами. У першому випадку рядок **repl** підставляється не просто так, а інтерпретується із заміною входжень **"\номер"** на групу з відповідним номером, також входжень **"\g<ім'я>"** на групу з номером або іменем **ім'я.** У разі, коли **repl** – функція, їй передається об'єкт із результатом кожного успішного зіставлення, а з неї повертається рядок для заміни;

- **subn(repl, s)**

Заміняє в рядку *s* усі (або тільки **count**, якщо його вказано) входження підрядків, що не перекриваються, які задовольняють шаблон, на рядок, указаний параметром **repl**. Як **repl** може

виступати рядок або функція. Повертає рядок із виконаними замінами. У першому випадку рядок **repl** підставляється не просто так, а інтерпретується із заміною входжень **"\номер"** на групу з відповідним номером, також входжень **"\g<ім'я>"** на групу з номером або іменем *ім'я*. У разі, коли **repl** – функція, їй передається об'єкт із результатом кожного успішного зіставлення, а з неї повертається рядок для заміни;

subn(repl, s)

аналогічний **sub()**, але повертає кортеж із рядка з виконаними замінами та кількістю замін.

У наступному прикладі рядок розбивається на підрядки за вказаним шаблоном:

```
>>> import re
>>> delim_re = re.compile(r"[:,;]")
>>> text = "This,is;example"
>>> print delim_re.split(text)
['This', 'is', 'example']
```

А тепер можна довідатися, чим саме розбито рядки:

```
>>> delim_re = re.compile(r"([:,;])")
>>> print delim_re.split(text)
['This', ',', 'is', ';', 'example']
```

7.4.3. Приклади шаблонів

Уміння використовувати регулярні вирази може істотно прискорити побудову алгоритмів для обробки даних. Найліпше знайомитися із шаблонами *на конкретних прикладах:*

- **r"\b\w+\b"**

відповідає слову з букв та знаків підкреслення;

- **r"[+-]?\d+"**

відповідає цілому числу. Можливо, зі знаком;

- **r"\([+-]?\d+\)"**

означає число, що стоїть у дужках. Дужки використовуються в регулярних виразах, тому вони екрануються **"\"**;

- **r"[a-cA-C]{2}"**

відповідає рядку з двох літер **"a"**, **"b"** або **"c"**. *Наприклад*, **"Ac"**, **"CC"**, **"bc"**;

- **r"aa|bb|cc|AA|BB|CC"**

рядок із двох однакових букв;

- **r"([a-cA-C])\1"**

рядок із двох однакових букв, але шаблон визначено з використанням груп;

- **r"aa|bb".**

відповідає **"aa"** або **"bb";**

- **r"a(a|b)b"**

відповідає **"aab"** або **"abb";**

- **r"^(?:\d{8}|\d{4}):\s*(.*)$"**

відповідає рядку, що починається набором із восьми або чотирьох цифр та двокрапки. Усе, що йде після двокрапки та після наступних за нею пробілів, виділяється в групу з номером 1, тоді як набір цифр у групу не виділений;

- **r"(\w+)=.*\b\1\b"**

слова ліворуч та праворуч від знака рівності наявні. Операнд **"\1"** відповідає групі з номером 1, виділеній дужками;

- **r"(?P<var>\w+)=.*\b(?P=var)\b"**

те саме, але тепер використовується іменована група **var**;

- **r"\bregular(?=\s+expression)".**

відповідає слову **"regular"** тільки тоді, якщо за ним після пробілів йде **"expression"**;

- **r"(?<=regular)expression"**

відповідає слову **"expression"**, перед яким стоїть **"regular"** та один пробіл.

Зазначимо, що приклади з поглядом назад можуть значно впливати на продуктивність, тому їх не варто використовувати без особливої потреби.

7.4.4. Налагоджування регулярних виразів

Наступний невеликий сценарій дозволяє налагоджувати регулярний вираз, за умови, що є приклад рядка, який шаблон повинен задовольняти. Узято фрагмент логу iptables, його необхідно розібрати для отримання полів. Цікаві рядки, в яких після **kernel:** стоїть **PAY:**, а в цих рядках потрібно отримати дату, значення **DST**, **LEN** та **DPT** (лістинг 7.1).

Лістинг 7.1. Приклад
із налагоджуванням регулярних виразів

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# PyQt приклад 7.1

import re

def debug_regex(regex, example):
    """Налагодження рег. вираз. Перед налагодженням краще забрати зайві дужки """
    last_good = ""
    for i in range(1,len(regex)):
        try:
            if re.compile(regex[:i]).match(example):
                last_good = regex[:i]
        except:
            continue
    return last_good

example = """Nov 27 15:57:59 lap kernel: PAY: IN=eth0 OUT=
    MAC=00:50:da:d9:df:a2:00:00:00:1c:b0:c9:db:08:00
SRC=192.168.1.200 DSR=192.168.1.1
    LEN=1500 TOS=0x00 PREC=0x00 TTL=64 ID=31324 DF
PROTO=TCP SPT=8080 DPT=1039
    WINDOW=17520 RES=0x00 ACK PSH URGP=0"""

log_re = r"""[A-Za-z]{3}\s+\d+\s+\d\d\d:\d\d} \S+ kernel: PAY: .+
    DST=(?P<dst>\S+).* LEN=(?P<len>\d+).* DPT=(?P<dpt>\d+)
"""

print debug_regex(log_re,example)
```

Функція **debug_regex()** пробує зіставляти приклад із порціями, що збільшуються, регулярного виразу й повертає останнє вдале зіставлення:

```
[A-Za-z]{3}\s+\d+\s+\d\d
```

Відразу видно, що не поставлено символ ":".

7.5. Робота з Unicode

До появи Unicode символи в комп'ютері кодувалися одним байтом (а то й лише 7 бітами). Один байт охоплює діапазон кодів від 0 до 255 включно, а це значить, що більше двох алфавітів, цифр, знаків пунктуації та деякого набору спеціальних символів в одному байті не вміститься. Кожний виробник використовував своє кодування для того самого алфавіту. *Наприклад,* дотепер дожили цілих п'ять кодувань літер кирилиці, і кожний користувач не раз бачив у своєму браузері або електронному листі приклад невідповідності кодувань.

Стандарт Unicode – єдине кодування для символів усіх мов світу. Його введення принесло велике полегшення та деяку незручність одночасно. Перевага Unicode полягає в тому, що в одному Unicode-рядку містяться символи зовсім різних мов. Мінус – у тому, що користувачі звикли застосовувати однобайтові кодування, більшість застосувань орієнтована на них, у багатьох системах підтримка Unicode здійснюється лише частково, тому що вимагає серйозної роботи з розробки шрифтів. Щоправда, символи одного кодування можна перекодувати в Unicode і назад.

Тут же зазначимо, що файли, як і раніше, прийнято вважати послідовністю байтів, тому для зберігання тексту у файлі в Unicode потрібно використовувати одне із транспортних кодувань Unicode (*utf*-7*, utf*-8*, utf*-16,...). У деяких таких кодуваннях має значення прийнятий на цій платформі порядок байтів (*big-endian* – старші розряди в кінці (або *little-endian* – молодші у кінці)). Довідатися про порядок байтів можна, прочитавши атрибут із модуля **sys**. На платформі Intel це має такий вигляд:

```
>>> sys.byteorder
'little'
```

Для уникнення неоднозначності можна на самому початку Unicode-документа розмістити BOM (byte-order mark – мітка порядку байтів) – Unicode-символ із кодом **0xfeff**. Для платформи Intel рядок байтів для BOM буде таким:

```
>>> codecs.BOM_LE
'\xff\xfe'
```

Для перекодування рядка в Unicode необхідно знати, в якому кодуванні закодовано вихідний текст. Припустимо, що це *cp*1251. Тоді перетворити текст в Unicode можна в такий спосіб:

>>> s = "Рядок в cp1251"

>>> s.decode("cp1251")

u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 cp1251'

Те саме за допомогою вбудованої функції **unicode()**:

>>> unicode(s, 'cp1251')

u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 cp1251'

Однією з корисних функцій цього модуля є функція **codecs.open()**, що дозволяє відкрити файл в іншому кодуванні:

codecs.open(filename, mode[, enc[, errors[, buffer]]])

Тут: **filename** – ім'я файла, **mode** – режим відкриття файла, **enc** – кодування, **errors** – режим реагування на помилки кодування (**'strict'** – генерувати виняткову ситуацію, **'replace'** – заміняти відсутні символи, **'ignore'** – ігнорувати помилки), **buffer** – режим буферизації (0 – без буферизації, 1 – порядково, **n** – байт буфера).

Контрольні запитання

1. Перерахуйте й опишіть основні операції з рядками, наявні в мові *Python*.
2. Яким чином можна змінювати рядки, якщо вони є незмінюваним типом даних?
3. Як указати кодування, що використовується для тексту *Python*-програми?
4. Яким чином із рядком можна працювати як із файлом?
5. Що таке регулярні вирази?
6. Перерахуйте основні символи загального призначення, що використовуються в регулярних виразах, та охарактеризуйте їх.
7. Опишіть основні методи об'єкта-шаблона.
8. Яким чином можна перекодувати деякий файл із кодуванням *koi-8u* у файл із кодуванням *cp866*?

ЧАСТИНА 2
ЧИСЛОВІ МЕТОДИ НА *PYTHON*

Розділ 1
ВСТУП

1.1. Автоматизація розв'язування задач проектування

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

Проектування починається з ретельного вивчення можливих рішень. Будується *фізична модель або простий прототип* розроблюваного об'єкта. Потім збирається інформація, що дозволяє побудувати модель розроблюваного виробу чи процесу для оцінювання та перевірки правильності прийнятого рішення. Необхідність цього етапу обумовлена економічними міркуваннями, тому що практична перевірка рішення майже завжди обходиться дуже дорого, забирає багато часу, вимагає занадто великих матеріальних та енергетичних витрат. Для цього, як правило, розробляється *математична модель* або використовується поєднання простого прототипу та математичної моделі. Побудувавши модель, вивчають її властивості, прагнучи з'ясувати, якою мірою розроблюваний виріб відповідає своєму призначенню. При цьому для знайдення задовільного рішення будують *обчислювальну модель* проектованого об'єкта. Побудова такої моделі та вивчення її властивостей може повторюватись доти, доки не з'явиться впевненість, що знайдено найліпше можливе рішення.

У процесі проектування доводиться виконувати найрізноманітніші обчислення, характер і обсяг яких можуть змінюватись. Наведемо області обчислень, для виконання яких доводиться використовувати комп'ютерні програми та системи автоматизації:

1) обчислення великого обсягу через їхній *ітераційний характер*;

2) громіздкі обчислення, які вимагають забезпечення необхідної *точності*;

3) *візуалізація* графічного подання даних.

Характер роботи інженера визначає багаторазовість, повторюваність розв'язуваних ним 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 тощо.

У цьому посібнику з числових методів розглядаються такі задачі, а також роз'яснюються основні поняття та терміни, пов'язані з методами їхнього розв'язування. Ці методи доводяться до обчислювальних алгоритмів розв'язування інженерних задач, що можуть бути реалізовані різними мовами програмування (у цьому посібнику використано мову *Python*) на різних обчислювальних пристроях, причому наголошується навиборі 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. Зауважимо, що в прикладних роботах користувача, як правило, цікавлять кількісні значення величин, тобто відповідь необхідно довести "до числа". При цьому всі розрахунки проводяться з числами, записаними як скінченні десяткові дроби, і тому результати обчислень завжди принципово носять *наближений характер*. Важливо тільки домогтися, щоб похибки укладалися в рамки необхідної точності.

1.2.1. Поняття алгоритму та граф-схеми алгоритму

У більшості математичних задач відповідь дається у вигляді *формули.* Формула, з огляду на встановлені математичні правила, 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 тощо.

Однак існує багато відомих задач, для яких відповідь можна легко знайти, хоча її не можна записати у вигляді 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 розв'язує поставлену задачу: воно визначає послідовність математичних операцій, яку потрібно виконати, щоб обчислити шукану величину.

Числовий метод розв'язування задачі – це визначена послідовність операцій в обчислювальній моделі задачі, тобто обчислювальний алгоритм, мовою якого врешті-решт є числа й арифметичні дії. Така примітивність мови дозволяє реалізувати числові методи на обчислювальних машинах, що робить ці методи потужним універсальним інструментом дослідження. Однак задачі формулюються, як правило, звичайною математичною мовою (рівнянь, функцій, диференціальних операторів тощо). Тому потрібно розробити числовий метод, що 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) *подання алгоритмів* розв'язування задач у цьому посібнику користуватимемося деякою абстрактною моделлю обчислювача й уніфікованих засобів опису алгоритмів у вигляді *граф-схем алгоритмів*.

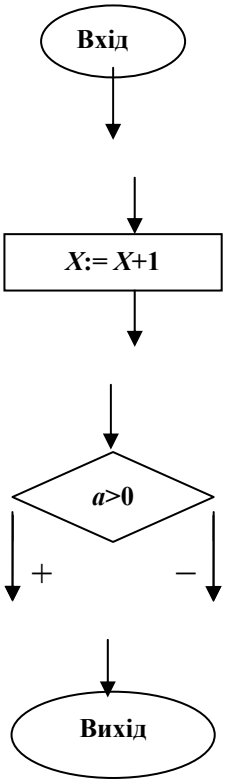
Вважатимемо, що *абстрактний обчислювач* із пам'яттю з довільним доступом складається з таких компонентів:

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

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

3) нескінченна в обидва боки стрічка введення-виведення (довготермінова пам'ять), з якої 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 такі графічні конструкції програмування:



Вхідна вершина; єдина точка входу для алгоритму

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 числових методів

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

Алгоритми розв'язування багатьох математичних задач, для яких не вдається одержати відповідь у вигляді формули, основані на такій процедурі: будується нескінченний процес, що збігається до шуканого розв'язку. Він переривається на певному 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, при якому похибка у визначенні розв'язку задачі не перевищить ε. Вираз "наближений розв'язок" не означає "розв'язок другого сорту". Якщо ми маємо відповідь для розв'язку певної задачі у вигляді формули і нам потрібно обчислити по ній значення, то, через подання чисел при обчисленнях скінченними десятковими дробами, можна одержати тільки наближений результат із певною бажаною точністю. Проблема застосування алгоритмів, що використовують нескінченний збіжний процес, не в наближеному характері відповіді, а у великому обсязі необхідних обчислень. Отже, такі алгоритми заведено називати обчислювальними алгоритмами, а основані на них методи розв'язування математичних задач – *числовими методами.* Широке застосування обчислювальних алгоритмів стало можливим тільки завдяки ЕОМ. До їхньої появи числові методи використовувалися рідко й лише в порівняно простих випадках через надзвичайну трудомісткість обчислень, що виконувалися вручну.

Поняття числового 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 різноманітність 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 $\sqrt{\ }$ нам не допоможе, бо не дає способу обчислення величини x.

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

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

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

Правомірність такого припущення залежить, очевидно, від вимог, які висуваються до точності розв'язування, від величини a та від параметра N. Зважаючи на будь-які вимоги, потрібно довести, що для будь-якого $a \geq 0$ відповідним вибором N можна досягти того, що обчислене значення x буде як завгодно близьке до точного значення $\sqrt{a}$.

Доведемо, що наш алгоритм (1.2) задовольняє цю умову. Покладемо

$$\frac{x_n}{\sqrt{a}} = 1 + \varepsilon_n. \tag{1.3}$$

Поділимо рівність (1.2) на $\sqrt{a}$. Підставивши в неї (1.3), одержимо

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

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, додатні. Використовуючи це, одержуємо з (1.4)

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

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 $\frac{1}{2}$. Отже, $x_N \to \sqrt{a}$ при $N \to \infty$. Твердження доведено.

На рис. 1.1 проілюстровано ітераційний процес (1.2). Тут зображено два графіки: лівої $y_{\text{л}}(x)$ та правої $y_{\text{п}}(x)$ частин (1.2). Оскільки, очевидно, що $y_{\text{л}}(\sqrt{a}) = y_{\text{п}}(\sqrt{a})$, то ці графіки перетинаються в точці $x = \sqrt{a}$. Проведення ітерацій за формулою (1.2) еквівалентно руху по зображеній на рисунку ламаній лінії, затиснутій між $y_{\text{л}}(x)$ та $y_{\text{п}}(x)$. Це ще раз переконує нас у збіжності ітерацій до $\sqrt{a}$ при $N \to \infty$.

При дослідженні збіжності було прийнято деяку ідеалізацію алгоритму, ми мовчки припустили, що можливо реалізувати точні обчислення 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 перевищувати точність розрахунків. Важливо встановити, у якому відно-

шенні ці точності перебувають, і чи не будуть похибки, що допускаються при округленні, накопичуватись, позбавляючи результат будь-якої цінності.

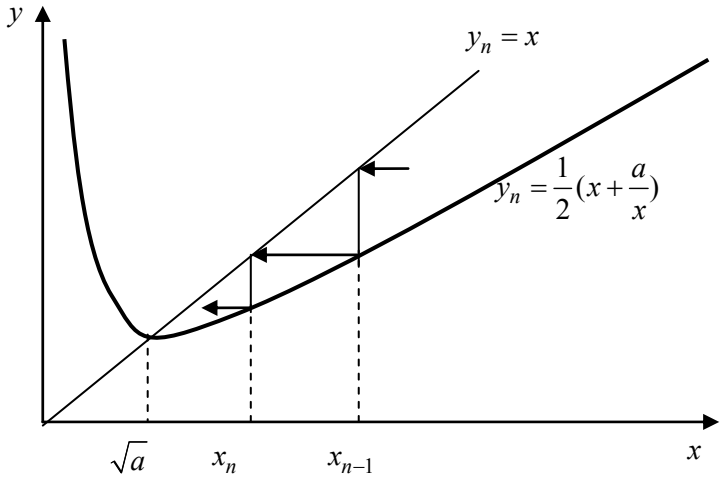


Рис. 1.1. Графічна інтерпретація збіжності

Перевіримо формально вплив зазначеного фактора. Роль округлень зводиться до того, що фактично замість формули (1.2) 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 цьому n-му кроці розрахунку, а $x_n^{'}$ − фактично одержувана послідовність. Величина $\partial_n << 1$ характеризує точність обчислень, Замінюючи $x_n^{'}$ на $\sqrt{a}(1 + \varepsilon_n)$, одержимо замість (1.6)

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

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

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

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

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 загальні принципи для всіх числових методів:

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

– задача (1.2) містить параметр N, якого немає у початковій задачі;

– вибором цього параметра можна домогтися будь-якої близькості розв'язку другої задачі до розв'язку першої;

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

Контрольні запитання

1. Назвіть типи моделей.
2. Що таке абстрактний обчислювач?
3. Дайте визначення граф-схеми алгоритму?
4. Які основні елементи граф-схеми алгоритму?
5. У чому суть числових методів?
6. Що таке збіжність?
7. Назвіть особливості числових методів.

Розділ 2
НАБЛИЖЕНІ ЧИСЛА Й ОЦІНЮВАННЯ ПОХИБОК ОБЧИСЛЕНЬ

2.1. Наближені числа. Класифікація похибок

У процесі обчислень часто доводиться працювати 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* і дає наближений результат;

– при введенні, обчисленнях та виведенні 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 називатимемо 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 цього наближення користуються поняттям похибки або помилки.

Похибкою Δa наближеного числа a називається різниця виду

$$\Delta a = A - a, \tag{2.1}$$

де A – відповідне точне число.

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

$$\Delta = |A - a|. \tag{2.2}$$

Через те, що точне число A зазвичай не відоме, то користуються поняттям граничної абсолютної похибки. Граничною абсолютною похибкою Δ_a наближеного числа a називається число, не менше від абсолютної похибки цього числа, тобто

$$\Delta_a \geq \Delta. \tag{2.3}$$

Із (2.3) маємо

$$\Delta_a \geq |A - a|, \tag{2.4}$$

отже,

$$a - \Delta_a \leq A \leq a + \Delta_a. \tag{2.5}$$

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 вимірювань звичайно розуміють граничну абсолютну похибку. *Наприклад,* якщо відстань між двома пунктами $S = 900$ м отримано з точністю до 0,5 м, то точне значення величини S лежить у межах $899{,}5 \leq S \leq 900{,}5$ м.

Уведення абсолютної чи граничної абсолютної похибки звичайно недостатньо для характеристики ступеня точності наближених чисел. Важливим 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

$$\delta = \frac{\Delta}{|A|}. \tag{2.6}$$

Граничною відносною похибкою наближеного числа a називається число δ_a, яке не менше за відносну похибку цього числа, тобто

$$\delta_a \geq \delta. \tag{2.7}$$

Із (2.7) маємо

$$\Delta \le |A| \delta_a . \qquad (2.8)$$

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

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

Якщо прийняти $A \approx a$, то формула (2.9) набуде вигляду

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

Отже, точне число A лежить у межах

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

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 можна подати у вигляді скінченної або 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 можна подати як

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

При реальних обчисленнях усяке число, що має вигляд нескінченної суми доданків, замінюється сумою скінченного числа доданків, тобто замість суми (2.12) записують суму у вигляді

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

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

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 цьому числі 0,073040 остання цифра не є значущою, то це число має бути записане у вигляді 0,07304.

Із вигляду 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 записують у вигляді 8,27 106, якщо ж воно має чотири значущі цифри, то у вигляді 8,270·105.

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

Наприклад, для точного числа $A = 14,298$ число $a = 14,300$ є наближеним числом із 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 – кількість вірних значущих цифр. Звідси, зв'язок кількості 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 – кількість вірних десяткових знаків числа a; a_m — перша значуща цифра числа a. Як граничну відносну похибку 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 цьому разі $a_m = 1,\ n = 4$. Із формули (2.15) маємо $\delta_a = 5\cdot 10^{-4}$ і навпаки, знаючи граничну відносну похибку, можна визначати кількість вірних десяткових знаків.

Приклад 2.2. Знайти кількість вірних десяткових знаків числа $\sqrt{5}$ при відносній похибці $\delta = 0,001$.

Очевидно, що $a_m = 2$. Далі $0,25\cdot 10^{1-n} \le 0,001$. Звідси $1-n \le -3+\lg 4,\ n \ge 4$.

2.3. Правила округлення чисел

Округленням числа a (точного чи наближеного) називається заміна його числом b із 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 усі його цифри праворуч, починаючи з $(n+1)$-ї, або, якщо необхідно зберегти розряди, заміняють їх нулями. Зазначене відкидання цифр здійснюється за такими правилами округлення:

1) якщо $(n+1)$-ша цифра більша 5, то до n-ї цифри додається одиниця;

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

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

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

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

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

2.4. Обчислення похибки функції від *n* аргументів

Нехай задано деяку функцію $y = f(x_1, x_2, ..., x_n)$, від n аргументів і нехай значення кожного з аргументів *xi* визначено з деякими похибками $|\Delta x_i|, i = 1, 2, ..., n$. Потрібно знайти похибку функції.

Щоб розв'язати цю задачу, припустимо, що функція $y = f(x_1, x_2, ..., x_n)$ є диференційовною у деякій області D. Знайдемо абсолютну похибку $|\Delta y|$ функції при заданих абсолютних похибках $|\Delta x_1|, |\Delta x_2|, ..., |\Delta x_n|$ аргументів

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

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

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

Отже, гранична абсолютна похибка Δy функції y

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

де Δx_i – гранична абсолютна похибка аргументу x_i.

Оцінку для відносної похибки функції отримуємо діленням обох частин нерівності (2.16) на $|y|$:

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

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

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

Розглянемо окремі приклади на обчислення похибок різних функціональних співвідношень. Припустимо, що в кожному прикладі задано певні види похибок аргументів.

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

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

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

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:

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

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

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

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

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 $\delta_y = \frac{\delta_x}{n}$.

Контрольні запитання

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

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

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

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) $a_1 = \pi \pm 0,001$; $a_2 = \cos 25^o \pm 0,001$; $y = a_1 - a_2$; $y = a_1 \cdot a_2$.

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

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

6) $a_1 = 3,85 \pm 0,01$; $a_2 = 2,043 \pm 0,0004$; $a_3 = 962,6 \pm 0,1$;

$$y = \frac{a_1 \cdot a_2}{a_3}.$$

7) $a_1 = 7,27 \pm 0,01$; $a_2 = 5,205 \pm 0,002$; $a_3 = 87,32 \pm 0,03$;

$$y = \frac{a_1}{a_2 + a_3}.$$

3. Кут, виміряний теодолітом, виявився рівним $22^{O}20'30"\pm30"$. 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 (із точністю до 0,01 см^2). Із якою відносною похибкою та зі скількома вірними знаками можна визначити сторону квадрата?

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

9. Знайти граничну відносну похибку при обчисленні повної поверхні зрізаного конуса, якщо радіуси його основ $R = 23,64 \pm 0,01$ (см), $r = 17,31 \pm 0,01$ (см), твірна $l = 10,21 \pm 0,01$ (см); число $\pi = 3,14$.

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

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

12. Перерахуйте види похибок.
13. Назвіть причини виникнення похибок.
14. Чи може похибка бути від'ємним числом?
15. Що таке неусувна похибка?
16. У чому суть машинної похибки?
17. Розкрийте поняття абсолютної похибки.
18. Що таке відносна похибка. Як вона вимірюється?
19. Дайте визначення поняття "значущі цифри".
20. Перерахуйте правила округлення чисел.
21. Як обчислити похибку від *n* аргументів?

Розділ 3
ЕЛЕМЕНТИ ВЕКТОРНОЇ І МАТРИЧНОЇ АЛГЕБРИ

3.1. Вектори

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

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

Операція *транспонування,* яка міняє стовпці на рядки і навпаки, позначається верхнім індексом T. Вектори зазвичай виражаються через базис; для цього вибирається стандартна сукупність векторів $b_1, b_2, ..., b_n$, а всі інші вектори виражаються через цей базис:

$$y = y_1 b_1 + y_2 b_2 + ... + y_n b_n .$$

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

$$y = (y_1, y_2, ..., y_n)^T .$$

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:

$$x + y = y + x = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n)^T ,$$
$$x - y = -(y - x) , \ (x + y) + z = x + (y + z) ;$$

• множення на скаляр:

$$ax = (ax_1, ax_2, ..., ax_n).$$

Сукупність векторів $x_1, x_2, ..., x_m$ – *лінійно незалежна*, якщо жодна їх лінійна комбінація не дорівнює нулю, за винятком нульової комбінації.

Простір називається *натягнутим* на сукупність векторів $x_1, x_2, ..., x_m$, якщо кожен вектор простору може бути виражений через лінійну комбінацію цих векторів. Розмірністю векторного простору є мінімальна кількість цих векторів, потрібних, щоб отримати простір; кожен базис N-мірного простору повинен мати у своєму складі N векторів. N-мірні векторні простори скорочено часто називаються N-просторами, і N-вектори є векторами в N-просторі. *Скалярний*, або внутрішній, добуток двох векторів x і y має вигляд

$$x^m y = xy = \sum_{i=1}^{m} x_i y_i ,$$

де x_i, y_i – компоненти векторів x та y. Два вектори називаються *ортогональними* (*перпендикулярними*), якщо $x^m y = 0$. Розмір вектора можна виміряти нормою $\|x\|$ так, що

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

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

$$\|x\|_\infty = \max |x_i| ,$$

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

Кут α між двома векторами визначається співвідношенням

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

Формат запису векторів має узгоджуватися з форматом запису матриць, тому вектори зазвичай записуються як вектористовпці. Це ускладнює виклад тексту, тому вектори записуються горизонтально зі знаком транспонування T, якщо використовувати формат стовпця не обов'язково.

3.2. Матриці

Абстрактні об'єкти, які приводять до матриць, є лінійними відображеннями, перетвореннями, або функціями, визначеними над векторами. Оскільки для векторів введено координати, то ці лінійні функції можна подати як двовимірну таблицю чисел, тобто як деяку матрицю $A = (a_{ij})$.

Якщо y є лінійна функція від x, то кожна компонента y_k вектора y є лінійна функція компоненти x_j вектора x. Таким чином, для кожного k маємо

$$y_k = a_{k1}x_1 + a_{k2}x_2 + ... + a_{kn}x_n.$$

Коефіцієнти збираються в матрицю A, яка і є лінійною функцією (відображення, перетворення, або співвідношення) між векторами x і y. Лінійна функція позначається Ax.

Дії над матрицями визначаються правилами, що виконуються для лінійних відображень. Так, $A + B$ є поданням суми двох лінійних функцій, які подано у свою чергу матрицями A та B. Маємо $A + B = C$, де $c_{ij} = a_{ij} + b_{ij}$. Добуток AB – це наслідок застосування відображення B та подальшого застосування відображення A. Покажемо, що $AB = C$, де $c_{ij} = \sum_k a_{ik} b_{kj}$ (*). Для цього позначимо $y = Bx$ та $z = Ay$. Отже, необхідно визначити матрицю таку, що $z = Cx$. Виразимо це співвідношення мовою компонент:

$$y_k = \sum_{j=1}^{n} b_{kj} x_j \,, \qquad z_i = \sum_{k=1}^{n} a_{ik} y_k \,.$$

Таким чином

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

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

$$a_{i1}(b_{11} x_1 + b_{12} x_2 + ... + b_{1n} x_n) + a_{i2}(b_{21} x_1 + b_{22} x_2 + ... + b_{2n} x_n) + ...$$
$$... + a_{in}(b_{n1} x_1 + b_{n2} x_2 + ... + b_{nn} x_n) =$$
$$= x_1(a_{i1} b_{11} + a_{i2} b_{21} + ... + a_{in} b_{n1}) + x_2(a_{i1} b_{12} + a_{i2} b_{22} + ... + a_{in} b_{n2}) + ...$$
$$... + x_n(a_{i1} b_{1n} + a_{i2} b_{2n} + ... + a_{in} b_{nn}) =$$
$$= \sum_{j=1}^{n} (\sum_{k=1}^{n} a_{ik} b_{kj}) x_j \,.$$

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.

Вірні такі арифметичні правила:

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

Функції додавання та множення матриць мовою *Python* мають вигляд як у лістингу 3.1.

Лістинг 3.1. Множення
та додавання матриць мовою *Python*

```
>>> 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
[[ 375037953840388539303975402040654 1104155]
 [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
[[ 20222426283032343638]
 [ 40424446485052545658]
 [ 60626466687072747678]
 [ 80828486889092949698]
 [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 117654333]
 [ 3222222222]
 [ 2111111111]
 [ 1111111111]
 [ 1111111111]
 [ 1111111111]
 [ 1111111111]
 [ 1111111111]
 [ 1111111111]
 [ 1111111111]]
```

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

Транспоновану матрицю за допомогою *Python* можна отримати таким чином (лістинг 3.2).

Лістинг 3.2. Транспонування матриць мовою *Python*

```
>>> 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]])
>>>

Одинична матриця обов'язково квадратна, тобто має однакову кількість рядків і стовпців. Очевидно, що $EA = AE = A$. Обернена матриця A–1 є така матриця, для якої виконується рівність $A^{-1}A = E$. Не всяка матриця має обернену. Справді, добуток AB може дорівнювати 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 висловлювання еквівалентні:

- матриця A невироджена;
- обернена матриця A^{-1} існує;
- стовпці матриці A лінійно незалежні;
- рядки матриці A лінійно незалежні;
- рівність $Ax = 0$ означає, що $x = 0$.

Одиничну та нульову матриці можна отримати за допомогою мови *Python* (лістинг 3.3).

Лістинг 3.3. Створення нульових
та одиничних матриць мовою *Python*

>>> 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]])
```

Розв'язування лінійних рівнянь полягає у знаходженні такого вектора x за даною матрицею A і даним вектором b, для якого виконується рівність $Ax = b$.

Якщо матриця невироджена (тим самим A – квадратна), то це завдання завжди має єдиний розв'язок для будь-якого b. Якщо A має більше рядків, ніж стовпців (тобто існує більше рівнянь, ніж змінних), то зазвичай така задача не має розв'язку. Якщо A має більше стовпців, ніж рядків, то зазвичай існує нескінченно багато розв'язків. Найдавнішим і стандартним методом розв'язування цієї задачі є метод виключення Гаусса (не будемо розглядати правило Крамера через його вкрай низьку ефективність).

Система рівнянь називається *однорідною*, якщо права частина дорівнює нулю, наприклад, $Ax = 0$.

Матриця U ортогональна, якщо її стовпці $u_1, u_2, ..., u_n$ ортогональні, як вектори, і їхня довжина дорівнює 1. Інакше кажучи, $u_i^T u_j = 0$, якщо $i \neq j$ і $u_i^T u_i = 1$.

Системи $Ux = b$ з ортогональними матрицями легко розв'язуються, оскільки $U^T U$ – одинична матриця (рядки U^T є стовпцями матриці U). Якщо $Ux = b$, то $U^T Ux = x = U^T b$ і x обчислюється безпосередньо. Таким чином, якщо відомий метод перетворення матриці A в ортогональну матрицю, то цей метод можна застосувати для розв'язування системи $Ax = b$.

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 ліворуч або праворуч на матрицю перестановок має результатом перестановку рядків або стовпців початкової матриці. Ця властивість і дала їм назву. Матриці перестановок застосовуються у формулах, щоб задавати перестановки рядків і стовпців, і рідко використовуються в обчисленнях як таких.

Власні значення матриці A – це такі числа λ, для яких виконується рівність $Ax = \lambda x$ для деякого ненульового вектора x; вектор 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 зазвичай, але не завжди, n власних векторів. *Спектральний радіус* $\rho(A)$ матриці A – це найбільше за абсолютною величиною власне значення A. Спектральний радіус грає фундаментальну роль у дослідженні збіжності ітераційних процесів, пов'язаних із матричними обчисленнями.

Існують різні способи вимірювання величини, або норми, матриці. Ми використовуватимемо лише 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, визначеного матрицею A. Така норма визначається в термінах векторної норми, і, отже, різні векторні норми дають різні норми для матриці A. Норма може виражатись явно через три раніше введені векторні норми:

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

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2 = \text{(найбільше власне значення } A^T A^{\frac{1}{2}}\text{)},$$

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

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 радіуса матриці.

Для будь-якої норми матриці A вірні такі властивості:

1) $\|A\| > 0$, причому $\|A\| = 0$, коли A – нульова матриця;

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

3) $\|A + B\| \le \|A\| + \|B\|$ – нерівність трикутника;

4) $\|A \cdot B\| \le \|A\| \cdot \|B\|$ – нерівність Коші – Буняковського.

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

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

Нехай λ – власне значення матриці A. Тоді для будь-якого $x \ne 0$ маємо $Ax = \lambda x$. Відомо, що однакові вектори мають рівні норми: $\|Ax\| = \|\lambda x\|$. За властивістю норми маємо $\|\lambda x\| = |\lambda| \cdot \|x\| \le \|A\| \cdot \|x\|$. Звідси $|\lambda| \le \|A\|$.

Цікавість до матриць виникає через їхній зв'язок із лінійними функціями декількох 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 так само, як і вектори "існують" як абстрактні об'єкти. Для векторів визначено поняття, якими можна маніпулювати і проводити обчислення за допомогою введення систем координат; те саме відбувається і для функцій із багатьма змінними. Якщо виразити x та y в позначеннях деякої системи координат, то має існувати формула для обчислення координат $(y_1, y_2, ..., y_n)^T$ через координати $x = (x_1, x_2, ..., x_n)^T$. Ця формула включає двовимірну таблицю чисел, скажімо $A = (a_{ij})$, і має вигляд

$$y = Ax .$$

Таким чином, матриця є конкретним поданням лінійної функції з багатьма змінними, отримане уведенням системи координат для векторів. Різні варіанти вибору систем координат дають різні матричні подання для однієї й тієї самої функції.

Якщо $B = C^{-1}AC$, то A і B називаються *подібними* матрицями; ця формула показує 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 змінюються 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, коли як базисні вибрано власні вектори (припустимо, що їх вистачить для цього). У цьому разі матричне подання лінійної функції – просто діагональна матриця. Компоненти вектора Ax є компонентами вектора 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 (детермінантом) матриці 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 замінюють довільною перестановкою чисел $1, 2, ..., n$.

Оскільки кількість перестановок із n чисел дорівнює $n!$, то кількість доданків у (3.1) дорівнює $n!$. Знак кожного доданка визначається k-числом "безладу" в послідовності $a_1, a_2, ..., a_n$. "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 від "1". Число "безладу" тут дорівнює одиниці і весь добуток у визначнику 3-го порядку беруть зі знаком "мінус". У добутку $a_{31}a_{12}a_{23}$ перші індекси утворюють послідовність 3, 1, 2. Оскільки число "3" розташовано раніше від "1" і "2", то число "безладу" дорівнює "2" і добуток підсумовується зі знаком "плюс".

Для обчислення визначника порядку n за формулою (3.1) треба додати $n!$ доданків, тобто виконати $(n!-1)$ операцій додавання. Для обчислення кожного доданка треба виконати $(n-1)$ операцій множення. Тобто загальна кількість операцій для обчислення визначника за формулою (3.1)

$$N = n! \cdot (n-1) + n! - 1 = n \cdot n! - 1 \approx n \cdot n!.$$

Із зростанням порядку n матриці кількість потрібних операцій збільшується дуже швидко. Наприклад, при $n = 3$ кількість операцій $N = 17$, при $n = 10$ $N = 3{,}6 \cdot 10^7$, при $n = 20$ $N = 5 \cdot 10^{19}$. ЕОМ із середньою швидкодією 1 млн операцій за секунду обчислить за формулою (3.1) визначник 10-го порядку за 36 с, а 20-го порядку – за 5×108 діб [1]. Зрозуміло, що за необхідності обчислити визначник, користуватися формулою (3.1) недоцільно, тому застосовують інші методи.

Нагадаємо деякі визначення.

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

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

Алгебраїчним доповненням A_{ij} елемента a_{ij} називають його мінор, узятий 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 рядки рівні або пропорційні, або якщо один із рядків є лінійна комбінація яких-небудь інших рядків.

3. Множник, спільний 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, i-й рядок якого дорівнює сумі відповідних елементів i-х рядків доданків:

$$\begin{vmatrix} \overset{C}{} \\ a_{i1}\dots a_{in} \\ \underset{D}{} \end{vmatrix} + \begin{vmatrix} \overset{C}{} \\ b_{i1}\dots b_{in} \\ \underset{D}{} \end{vmatrix} = \begin{vmatrix} \overset{C}{} \\ a_{i1} + b_{i1}\dots a_{in} + b_{in} \\ \underset{D}{} \end{vmatrix}.$$

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:

$$|A| = a_{i1}A_{i1} + a_{i2}A_{i2} + ... + a_{in}A_{in},$$

де A_{ij} – алгебраїчне доповнення.

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

$$\sum_{j=1}^{n} a_{ij}A_{kj} = 0 \quad \text{ï ðè } i \neq k.$$

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

9. Із теореми (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 Гаусса або його модифікацію, а потім обчислюють визначник одержаної трикутної або діагональної матриці. При цьому знак не змінюється, якщо кількість перестановок рядків при виборі головного елемента парна, і змінюється на протилежний, якщо кількість таких перестановок непарна.

Щоб обчислити детермінант матриці мовою *Python*, потрібно виконати дії, наведені в лістингу 3.4.

Лістинг 3.4. Визначення детермінанта матриці мовою *Python*

```
>>> 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.]])
>>> 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]])
```

3.2.2. Обернення матриць

Обчислення елементів оберненої матриці називається *оберненням* матриці.

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

Визначення 3.1. Квадратна матриця називається верхньою (нижньою) трикутною, якщо елементи, що стоять вище (нижче) від головної діагоналі, дорівнюють 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. У дисципліні лінійної алгебри доводиться таке:

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

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

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

$$
\begin{aligned}
&1)\ u_{ij} = 0 \ \text{ при } \ i > j ; \\
&2)\ u_{ij} = \frac{1}{a_{ii}} \ \text{ при } \ i = j ; \\
&3)\ u_{ij} = -\frac{1}{a_{jj}} \sum_{k=1}^{j-1} u_{ik} a_{rj} \ \text{ при } \ i < j .
\end{aligned}
\qquad (3.2)
$$

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

$$
\begin{aligned}
&1)\ l_{ij} = 0 \ \text{ при } \ i < j ; \\
&2)\ l_{ij} = \frac{1}{a_{ii}} \ \text{ при } \ i = j ; \\
&3)\ l_{ij} = -\frac{1}{a_{ii}} \sum_{k=1}^{i-1} l_{kj} a_{ik} \ \text{ при } \ i > j .
\end{aligned}
\qquad (3.3)
$$

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 так до n-го рядка.

Загальна кількість дій множення та ділення, потрібних для обернення трикутних матриць, визначається з виразу

$$\frac{n^3 + 3n^2 + 2n}{6}.$$

3.2.3. Матричні рівняння

У системі лінійних алгебраїчних рівнянь

$$Ax = B \tag{3.4}$$

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 розглянути різні зовнішні впливи, тобто розв'язати декілька 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*:

$$Ax = B\,. \tag{3.6}$$

Тут A – квадратна $n \times n$ матриця; x – матриця $n \times m$, яка складається з векторів невідомих $x_1, x_2, ..., x_m$; B – матриця $n \times m$, яка складається 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 записі матричного рівняння (3.6) верхній індекс означає номер однієї із систем (3.5), які можуть виникати послідовно одна за одною при розв'язуванні більш складної задачі. Чи можна в цьому разі скоротити кількість потрібних операцій

для їх розв'язування порівняно з $\frac{2}{3}n^3m$, використовуючи незмінність їхньої матриці? Очевидно, розклавши один раз матрицю A на добуток двох трикутних матриць і витративши при цьому $\frac{4n^3-3n^2-n}{6}$ операцій, можна потім для кожної з m систем виконувати тільки прямий і зворотний ходи LU-алгоритму (цей алгоритм розглянуто в п. 6.2.5). Прямий хід потребує $n(n-1)$ операцій, зворотний – n, всього $2n^2-n$ операцій. Використовуючи такий підхід, розв'яжемо системи (3.5) за $\frac{4n^3-3n^2-n}{6}+m(2n^2-n)$ операцій, що становить близько $\frac{2}{3}n^3+2mn$. Одержана кількість операцій приблизно в m разів менша, ніж потрібно ($\frac{2}{3}n^3m$) при незалежному розв'язуванні цієї серії систем.

Системи (3.5) можуть бути також одночасно відомі. У цьому разі їх можна об'єднати в одну матричну систему (3.7) і розв'язувати не лише за допомогою LU-алгоритму, але й методом Гаусса. В іншому разі при відніманні рядків матриці A віднімаються й ті самі рядки матриці B. Тим самим одночасно розв'язуються m систем. Із погляду кількості операцій метод Гаусса і LU-алгоритм еквівалентні.

Контрольні запитання

1. Обчислити $\|x\|_1$, $\|x\|_2$ та $\|x\|_\infty$ для кожного з 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 кут α між 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 векторних 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 $x_1,x_2,...,x_m$ є лінійно незалежні, а $x_1,x_2,...,x_m,x_{m+1}$ лінійно залежні. Показати, що x_{m+1} є лінійною комбінацією $x_1,x_2,...,x_m$.

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-якої з трьох норм (1,2 або ∞) виконується нерівність

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

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 й 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 і ∞). *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 ненульовий розв'язок системи:

$$\begin{cases} x_1 - 2x_2 + x_3 = 0, \\ x_1 + x_2 - 2x_3 = 0. \end{cases}$$

14. Нехай

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

а) обчислити AB та BA і показати, що $AB \ne BA$,

б) знайти $(A+B)+C$ та $A+(B+C)$,

в) показати, що $(AB)C = A(BC)$,

г) показати, що $(AB)^T = B^T A^T$.

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

16. Для матриці A із 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 4-го порядку переставляє її другий та четвертий стовпці. Знайти P_2, яка переставляє перший та третій рядки матриці A.

18. Перевірити системи рівнянь на спільність (тобто показати існування принаймні одного спільного розв'язку):

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 кількість операцій додавання та множення, необхідних для множення $(n\times n)$-матриці на n-вектор та перемноження 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 3×3-матрицю A та ненульовий вектор x такі, що $Ax=0$.

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

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

25. Нехай x – вектор-рядок. Показати, що $xx^T=xx=\|x\|^2$. Чому дорівнює x^Tx?

26. Нехай для будь-якого вектора x функцію F визначено одним із таких 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 із цих способів визначає F як лінійну функцію. Знайти для кожної з лінійних функцій матрицю, що її подає.

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

28. Показати, що $\|AB\| \le \|A\| \cdot \|B\|$ для матричної норми, визначеної в цьому розділі. Довести правильність цієї формули для норми $\|A\|_\infty \|A\|_1$.

29. Показати для ($n \times n$)-матриці A, що $\max\limits_{i,j} |a_{ij}| \le \|A\| \le n \cdot \max\limits_{i,j} |a_{ij}|$.

30. Що таке вектор?
31. У чому суть операції транспонування?
32. Яка сукупність векторів називається лінійно незалежною?
33. Який простір вважається натягнутим на сукупність векторів?
34. Які вектори є ортогональними?
35. Дайте визначення поняття матриці.
36. Перерахуйте властивості одиничної матриці.
37. Яка матриця є виродженою?
38. Що таке ортогональність матриці?
39. Розкрийте поняття матриці перестановок.
40. Що таке власні значення матриці?
41. Що таке спектральний радіус матриці?
42. Які норми має матриця?
43. Які властивості мають норми матриці?
44. Як обчислити визначник матриці?
45. Перерахуйте основні властивості визначника матриці.
46. Дайте визначення мінора матриці.
47. Дайте визначення алгебраїчного доповнення.
48. Що таке трикутна матриця? Які вони бувають?
49. Розкрийте поняття матричного рівняння.

Розділ 4
ЧИСЛОВЕ РОЗВ'ЯЗУВАННЯ АЛГЕБРАЇЧНИХ ТА ТРАНСЦЕНДЕНТНИХ РІВНЯНЬ

4.1. Вступ

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

Вибір відповідного алгоритму для розв'язування рівнянь залежить від характеру завдання. Завдання, що зводяться до розв'язування алгебраїчних та трансцендентних рівнянь, можна класифікувати за кількістю рівнянь, передбачуваному характеру і кількістю розв'язків (рис. 4.1).

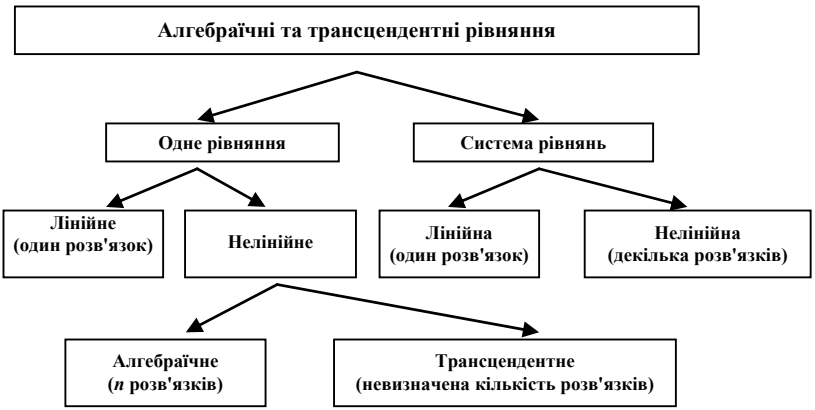


Рис. 4.1. Схема класифікації рівнянь

Рівняння називатимемо *лінійним, алгебраїчним* або *трансцендентним* залежно від того, чи має воно один розв'язок, *n* розв'язків або невизначену кількість розв'язків. Систему рівнянь називатимемо *лінійною* або *нелінійною* залежно від математичної природи рівнянь, що входять до неї. Розв'язування лінійного

рівняння з одним невідомим виконується досить просто і тут не розглядається. Мета цього розділу – виклад різних методів розв'язування рівнянь, що належать до решти типів.

4.2. Корені нелінійного рівняння

Зазвичай нелінійні рівняння поділяють на трансцендентні та алгебраїчні. Хоча вони часто розв'язуються одними й тими самими методами, розглянемо методи їх розв'язування окремо, оскільки розв'язування алгебраїчних рівнянь має особливі властивості. Цей підрозділ присвячений трансцендентним рівнянням; алгебраїчні 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 (відокремлення коренів) і 2) уточнення наближених значень коренів.

Загальних методів відокремлення коренів не існує. Методи уточнення наближених значень коренів при розв'язуванні нелінійних рівнянь такого типу є *прямі* й *ітераційні*. Перші дозволяють знайти рішення безпосередньо за допомогою формул і завжди забезпечують отримання точного розв'язку. Відомим прикладом прямих методів є формула для визначення коренів квадратного рівняння. В ітераційних методах задається процедура розв'язування у вигляді багаторазового застосування певного 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 підрозділі. 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 алгебраїчних рівнянь.

4.2.1. Метод половинного ділення

Блок-схему алгоритму методу половинного ділення зображено на рис. 4.2.

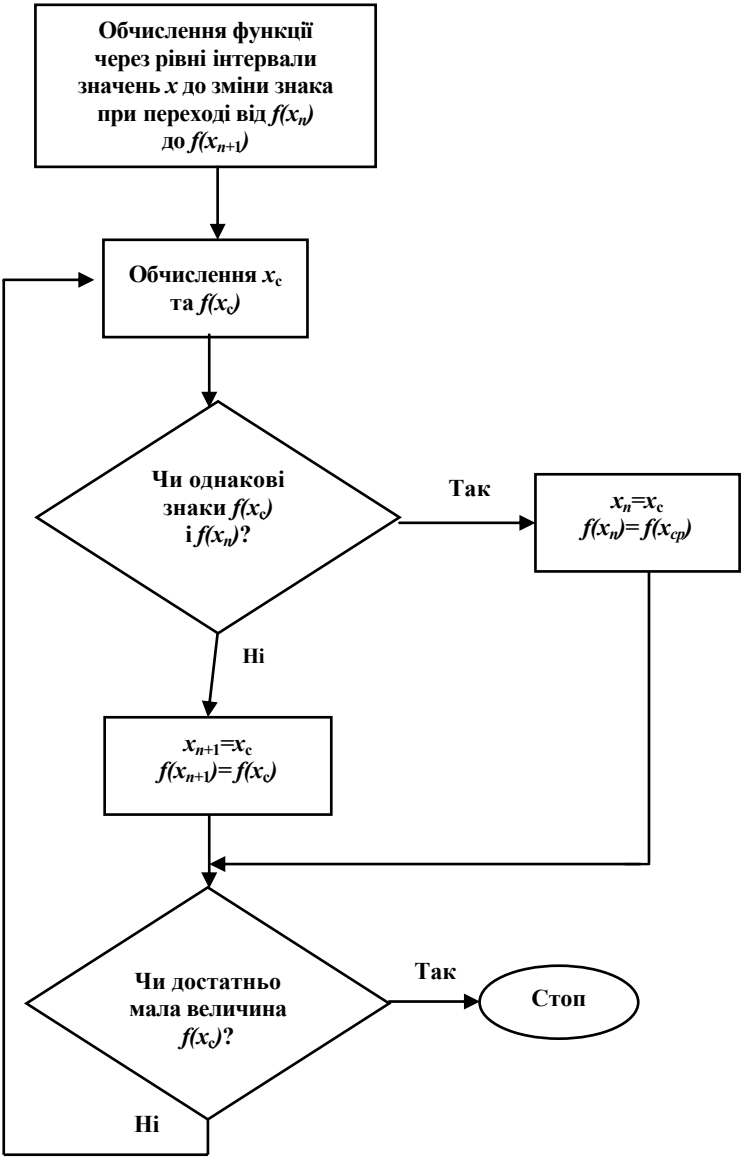


Рис. 4.2. Блок-схема алгоритму методу половинного ділення

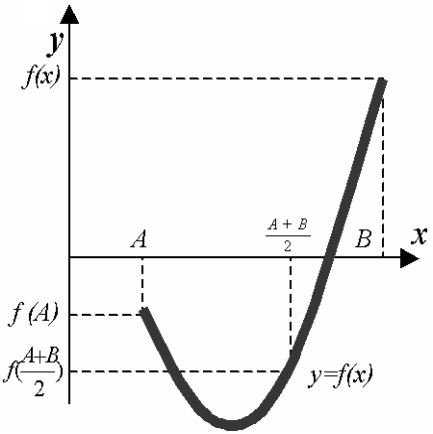


Рис. 4.3. Графічна інтерпретація алгоритму методу половинного ділення

Алгоритм складається з таких операцій. Спочатку обчислюються значення функцій у точках, розташованих через рівні інтервали на осі x. Це робиться доти, поки не буде 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 відшукується значення функції $f(x_c)$. Якщо знак $f(x_c)$ збігається зі знаком $f(x_n)$, то надалі замість $f(x_n)$ використовується $f(x_c)$. Якщо ж $f(x_c)$ має знак, протилежний знаку $f(x_n)$, тобто її знак збігається зі знаком $f(x_{n+1})$, то на $f(x_c)$ замінюється це значення функції. У результаті інтервал, у якому міститься значення кореня, звужується. Якщо $f(x_c)$ достатньо близьке до нуля, процес закінчується, в іншому випадку він продовжується. На рис. 4.3. ця процедура показана графічно. Хоча метод половинного ділення не має високої обчислювальної ефективності, із збільшенням кількості

ітерацій він забезпечує отримання все більш точного наближеного значення кореня. Після того, як уперше знайдено інтервал, у якому міститься корінь, його ширина після N ітерацій зменшується у $2N$ разів.

Реалізацію наведеного алгоритму мовою *Python* подано в лістингу 4.1.

Лістинг 4.1. Реалізація алгоритму методу половинного ділення мовою *Python*

```
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
```

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

4.2.2. Метод хорд

В основі цього методу лежить лінійна інтерполяція по двох значеннях функції, що мають протилежні знаки. При відшуканні кореня цей метод часто забезпечує швидшу збіжність, ніж попередній. Метод виконується таким чином. Спочатку знаходять значення функції в точках, розташованих на осі x через рівні інтервали. Це робиться доти, поки не буде знайдено пару послідовних значень функції $f(x_n)$ та $f(x_{n+1})$, що мають протилежні знаки. На рис. 4.4 процес розв'язування показано графічно.

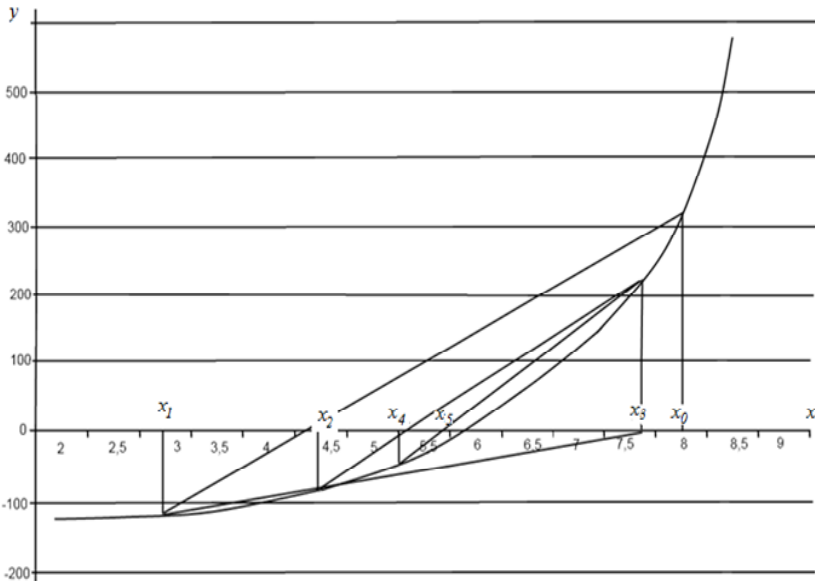


Рис. 4.4. Метод хорд

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 обчислення значення функції $f(x^*)$, яке порівнюється зі значеннями функцій $f(x_n)$ та $f(x_{n+1})$ і надалі використовується замість того з них, з яким воно збігається за знаком. Якщо значення $f(x^*)$ є недостатньо близьким до нуля, то вся процедура повторюється доти, поки не буде досягнуто необхідного ступеня збіжності. Блок-схему алгоритму методу хорд показано на рис. 4.5.

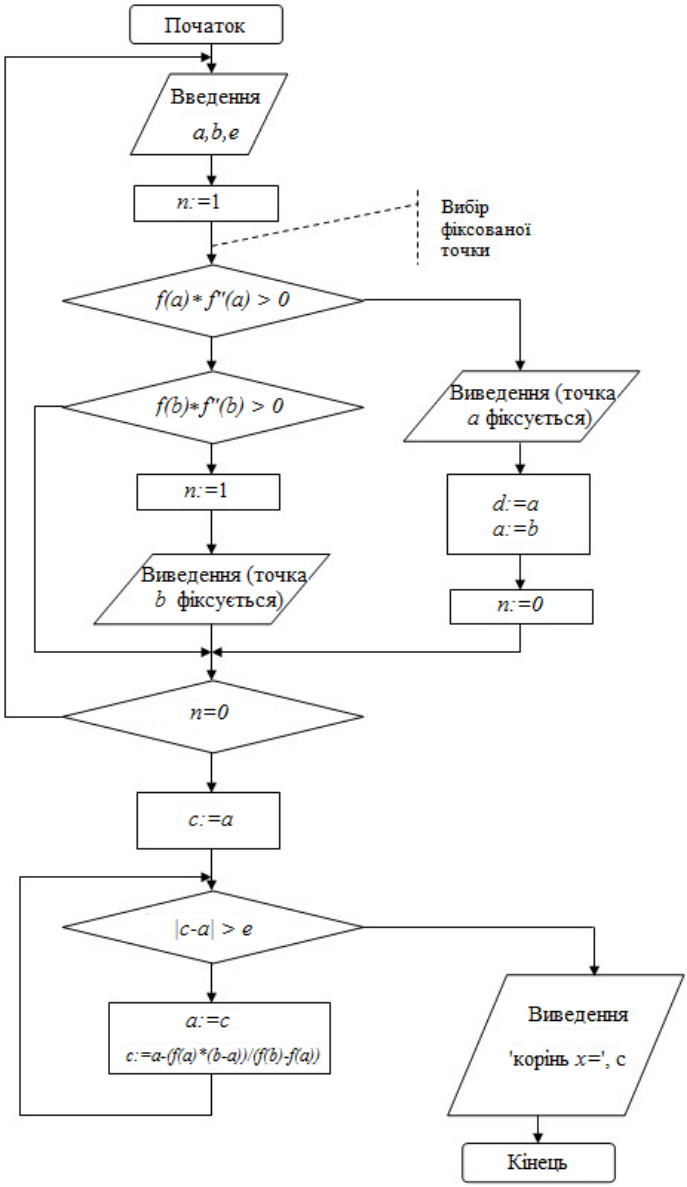


Рис. 4.5. Блок-схема алгоритму методу хорд

Оптимізовану під мову *Python* реалізацію алгоритму методу хорд наведено в лістингу 4.2.

Лістинг 4.2. Реалізація алгоритму методу хорд мовою *Python*

```
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
```

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

4.2.3. Метод Ньютона

Метод послідовних наближень, розроблений Ньютоном, широко використовується при побудові ітераційних алгоритмів. Його популярність зумовлена тим, що на відміну від двох попередніх методів для визначення інтервалу, в якому розташовується 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 у даній точці. На рис. 4.6 наведено блок-схему алгоритму цього методу, в основі якого лежить розвинення функції $f(x)$ у ряд Тейлора:

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

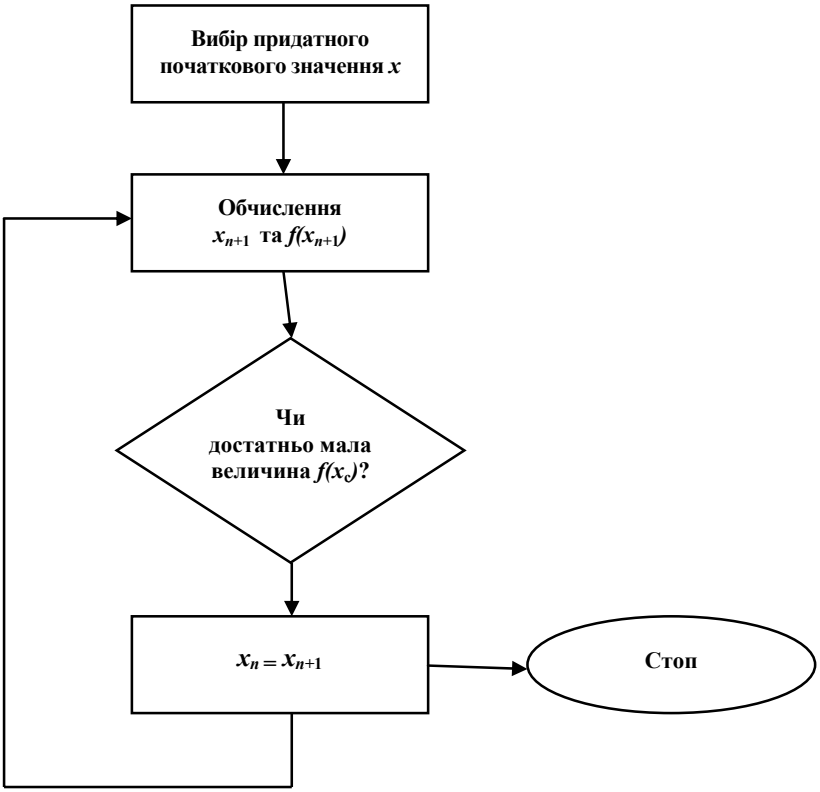


Рис. 4.6. Блок-схема алгоритму методу Ньютона

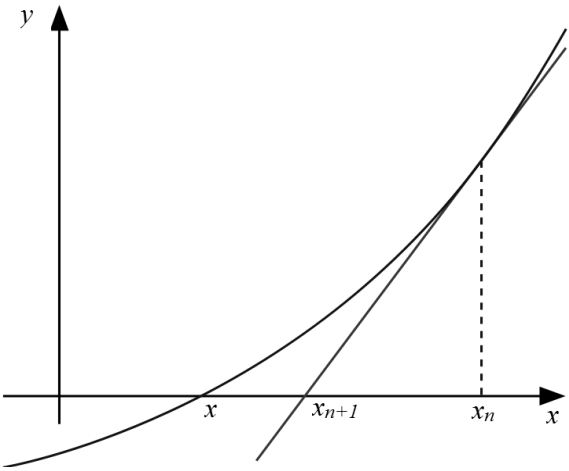


Рис. 4.7. Метод Ньютона

Члени, що містять h, при другій та вищих степенях похідних відкидаються; використовується 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 $f(x)$ відрізняється 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. 4.7 процес розв'язування рівняння методом Ньютона показано графічно. Ясно, що швидкість збіжності великою мірою залежить від вдалого вибору початкової точки. Якщо в процесі ітерацій тангенс кута нахилу дотичної $f(x)$ перетворюється на

нуль, то застосування методу ускладнюється. Можна також показати, що в разі нескінченного великого $f''(x)$ метод також не буде достатньо ефективним. Оскільки умова кратності кореня має вигляд $f(x) = f'(x) = 0$, то в цьому разі метод Ньютона не забезпечить збіжності. Варіант реалізації алгоритму мовою *Python* наведено в лістингу 4.3.

Лістинг 4.3. Реалізація алгоритму методу Ньютона мовою *Python*

```
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
```

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

4.2.4. Метод простої ітерації

Щоб застосувати цей метод, рівняння $f(x) = 0$ подається у вигляді $x = g(x)$. Відповідна ітераційна формула має вигляд

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

Блок-схему алгоритму методу показано на рис. 4.8, реалізацію алгоритму мовою *Python* – у лістингу 4.4. Простота методу простої ітерації приваблює, проте не слід забувати, що і цей метод має вади, оскільки він не завжди забезпечує збіжність. Тому для будь-якої програми, в якій використовується цей алгоритм, необхідно передбачати контроль збіжності та припиняти розрахунок, якщо збіжність не забезпечується.

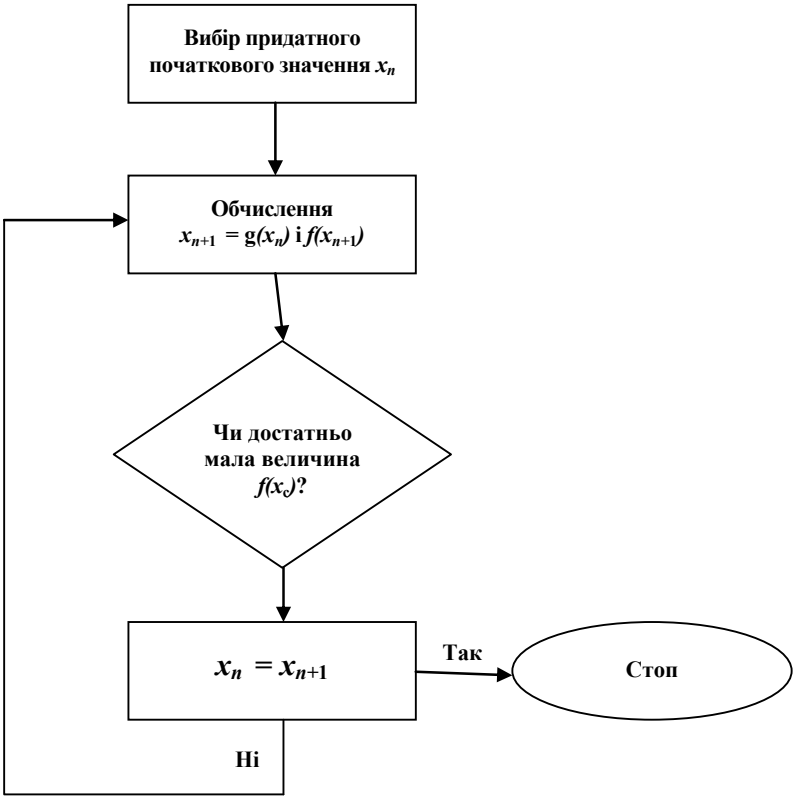


Рис. 4.8. Блок-схема алгоритму методу простої ітерації

Лістинг 4.4. Реалізація алгоритму методу простої ітерації

```
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
```

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

Функція $f(x)$ – це подання рівняння у вигляді $x = f(x)$, наприклад, для рівняння $e^{2x} + 3x - 4 = 0$ функція $f(x)$ має вигляд $\dfrac{\ln(4-3x)}{2}$.

Як приклад пригадаємо раніше розглянуте рівняння $x^2 - a = 0$. Його коренями є числа $X = \pm\sqrt{a}$. Щоб подати рівняння у вигляді $x = g(x)$, яке може бути неоднозначним, слід запропонувати, наприклад, рівняння $x = \dfrac{a}{x}$ або $x = 2x - \dfrac{a}{x}$ із такими самими коренями. Неважко переконатися, що жодна формула не годиться для ітерацій. Перша дає незбіжну послідовність $x_1 = \dfrac{a}{x_0},\ x_2 = x_0,\ x_3 = \dfrac{a}{x_0}$, а друга породжує послідовність x_n, що прямує до нескінченності. Проте рівняння $x = \dfrac{1}{2}(x - \dfrac{a}{x})$ із такими самими коренями, як було показано вище, уже дає збіжну послідовність. Таким чином, далеко не будь-який запис рівняння у вигляді (4.4) приводить до мети. З'ясуємо, які властивості $g(x)$ впливають на збіжність процесу. Оскільки збіжність означає, що $x_n - X \to 0$ при $n \to \infty$, то потрібно, щоб величина $|x_n - X|$ спадала зі 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 дещо підсилити її. Вище припущено, що корінь локалізовано всередині певного інтервалу. Якщо для будь-якої пари точок x', x'' із цього інтервалу виконується умова

$$\left|g(x') - g(x'')\right| \le c\left|x' - x''\right|, c < 1, \tag{4.7}$$

то тим більше виконується і (4.6). Відображення $x = g(x)$, що задовольняє (4.7), називають *стискаючим відображенням* (оскільки воно стискає відрізки $x'' - x'$). Як бачимо, 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 у цьому сенсі природно вважати той, для якого коефіцієнт c буде найменшим.

Приклад 4.1. Методом хорд визначити корінь рівняння

$$f(x) = x^3 + x - 12 = 0. \tag{4.8}$$

Для знаходження наближеного кореня рівняння (4.8) слід намалювати ескізи графіків функцій $y = x^3$ та $y = 12 - x$. Приблизно можна визначити, що шуканий корінь, тобто абсциса точки перетину графіків, лежить у інтервалі (2,3). Справді

$$f(2) = 2^3 + 2 - 12 = -2,$$
$$f(3) = 3^3 + 3 - 12 = +18.$$

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

Застосовуючи формулу (4.1), отримаємо наближене значення кореня:

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

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

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

Тут слід дотримуватися правила: *якщо друга похідна функції* $f''(x)$ *зберігає сталий знак в інтервалі* (a,b), *то дотичну слід проводити в тій кінцевій точці дуги* AB, *для якої знак функції збігається зі 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) приймаємо $x_n = 3$. 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 зазначимо, 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 самою точкою $x_n = 3$ для дотичної.

Приклад 4.3. Розв'язати рівняння

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

із точністю $\varepsilon = 10^{-3}$.

Для локалізації коренів застосуємо графічний спосіб. Зведемо початкове рівняння до такого еквівалентного вигляду:

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

Побудувавши графіки функцій $f_1(x) = e^{2x}$ та $f_2(x) = 4 - 3x$ (рис. 4.9), визначаємо, що розв'язуване рівняння має тільки один корінь, який міститься в інтервалі $0,4 < x^{(*)} < 0,6$.

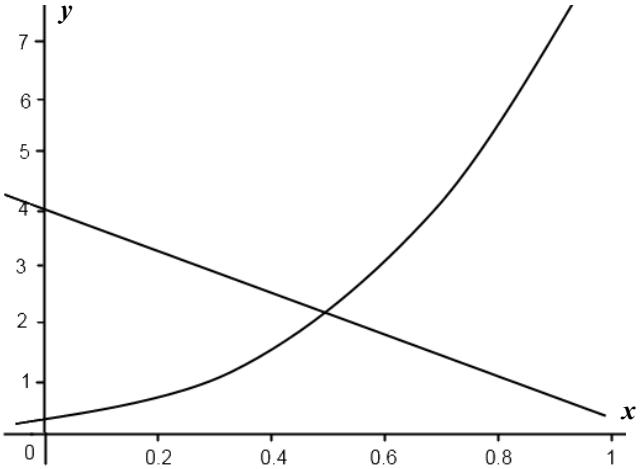


Рис. 4.9. Графіки функцій

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

Першим застосуємо *метод половинного ділення.* Як початковий відрізок виберемо $[0.4, 0.6]$. Результати подальших обчислень відповідно до наведеного вище алгоритму занесемо в табл. 4.1.

Таблиця 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]	

$$x^{(*)} \approx 0,474.$$

Далі застосуємо *метод Ньютона.* Для коректного використання цього 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 $f'(x) = 2e^{2x} + 3$ та $f''(x) = 4e^{2x}$ – додатні в усій 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 проводять за формулою (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 заносяться в табл. 4.2.

Таблиця 4.2. **Результати обчислень методом Ньютона**

k	$x^{(k)}$	$f(x^{(k)})$	$f'(x^{(k)})$	$-f(x^{(k)}) / f'(x^{(k)})$
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]			

$$x^{(*)} \approx 0,474.$$

Далі використаємо *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 прийнятний варіант (4.11), тому що, узявши за основний інтервал $(0.4, 0.55)$ та поклавши $g(x) = \frac{\ln(4-3x)}{2}$, матимемо

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 $(0.4, 0.55)$ $\left|g'(x)\right| < 0,64 = q$.

Як початкове наближення покладемо $x_0 = (0,4 + 0,55)/2 = 0,475$.

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

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

Результати обчислень наведено в табл. 4.3.

Таблиця 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 $x \in X$ уведено 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 a з властивостями:

$$a(x_1 + x_2) = ax_1 + ax_2;$$
$$(a_1 + a_2)x = a_1x + a_2x;$$
$$a_1(a_2x) = (a_1a_2)x.$$

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, $y \in X$, виконуються

$$\rho(x, y) = \rho(y, x) \text{ для } x \neq y;$$
$$\rho(x, x) = 0;$$
$$\rho(x, y) \leq \rho(x, z) + \rho(z, y); x < z < y.$$

Визначення 4.4. Дійсний лінійний простір називається *нормованим*, якщо кожному елементу $x \in X$ поставлено у відповідність від'ємне число $\|x\|$, що називається нормою і задовольняє такі умови:

$$\|x\| \geq 0,$$
$$a\|x\| = |a|\|x\|,$$
$$\|x + y\| \leq \|x\| + \|y\|.$$

У лінійному нормованому просторі метрику вводять як норму різниці між елементами:

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

Найбільш уживаними є такі види норм:

L^1-норма – середнє значення на інтервалі $[0,T]$ має вигляд

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

L2-норма – ефективне значення похибки на $[0,T]$ запишеться як

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

M- норма – мажоранта для всіх значень абсолютної похибки має вигляд

$$\|f(x)\|_M = \max_x |f(x)|.$$

Послідовність $\{x_n\}$ елементів простору X називається збіжною до елемента $x_0 \in X$, якщо з $\|x_m - x_n\| \to 0$ 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 $f(x) = 0$ зводиться до вигляду $x = g(x)$, ітерації проводяться за формулою $x^{n+1} = g(x^n)$.

Із більш загальних позицій цей метод можна подати таким чином. Нехай H – повний метричний простір, а оператор $y = g(x)$ відображає H у себе. Розглядається ітераційний процес $x^{n+1} = g(x^n)$ для розв'язування рівняння $x = g(x)$. Якщо при деякому $q < 1$ відображення $y = g(x)$ задовольняє умову

$$\rho(g(x_1), g(x_2)) \le q \cdot \rho(x_1, x_2) \qquad (4.12)$$

при всіх x_1, x_2, то таке відображення називають стискаючим.

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

Доведення. Із визначення стискаючого відображення (4.12) маємо

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

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

$$\begin{aligned} \rho(x^l, x^n) &\le \rho(x^l, x^{l-1}) + \ldots + \rho(x^{n+1}, x^n) \le \\ &\le q^{l-1}\rho(x^1, x^0) + \ldots + q^n \rho(x^1, x^0) \le \\ &\le q^n \rho(x^1, x^0) \sum_{i=0}^{\infty} q^i = \frac{q^n \rho(x^1, x^0)}{1-q}. \end{aligned} \tag{4.13}$$

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

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

З іншого боку, виконується ланцюжок співвідношень

$$\begin{aligned} \rho(X, g(X)) &\le \rho(X, x^{n+1}) + \rho(x^{n+1}, g(X)) = \rho(X, x^{n+1}) + \rho(g(x^n),\ g(X)) \le \\ &\le \rho(X, x^{n+1}) + q\rho(x^n, X) \le 2\frac{q^{n+1}\rho(x^1, x^0)}{1-q}. \end{aligned} \tag{4.14}$$

Переходячи у (4.14) до границі, маємо $\rho(X, g(X)) = 0$, отже, $X = g(X)$. Якби рівняння мало два розв'язки X_1 і X_2, то $\rho(X_1, X_2) = \rho(g(X_1), g(X_2)) \le q\rho(X_1, X_2) < \rho(X_1, X_2)$, приходимо до суперечності. Теорему доведено.

Контрольні запитання

1. Знайти розв'язки трансцендентних рівнянь:

1) $0,25X - \sin X = 0;$

2) $\sin X - 1 / X = 0;$

3) $\ln Z - \sin Z = 0.$

2. Знайти хоча б один із дійсних коренів алгебраїчних рівнянь:

1) $x^4 + 7x^3 + 3x^2 + 4x + 1 = 0;$

2) $7x^4 + 5x^3 + 2x^2 + 4x + 1 = 0;$

3) $x^4 + 5x^3 + 5x^2 - 5x + 6 = 0;$

4) $x^5 + x^4 + 2x^2 - x - 2 = 0.$

3. У чому відмінність алгебраїчного і трансцендентного рівнянь?

4. Чи має метод половинного ділення гарантовану збіжність?

5. Яким чином вибирається початкове наближення в методі Ньютона?

6. Для яких функцій не рекомендується застосовувати метод Ньютона?

7. Чи може інтервал у методі хорд розміщуватися з однієї сторони від кореня?

8. У чому суть методу половинного ділення?

9. Розкрийте алгоритм методу простої ітерації.

10. Що таке стискаюче відображення?

11. Розкрийте поняття дійсного лінійного простору.

12. Який простір називається метричним?

13. Який простір називається нормованим?

14. 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 є лише таблиця її значень у вигляді сукупності точок, одержана, наприклад, експериментальним шляхом. Нам же можуть знадобитися значення y в інших точках x, відмінних від вузлів x_i. Замість того, щоб проводити надто дорогий експеримент для розширення таблиці, використовують відносно нескладні формули апроксимації функції для наближеного знаходження її значень у проміжних точках.

Зустрічається й інша ситуація, коли залежність $y = f(x)$ відома, але дуже складна (містить важко обчислювані вирази, складні інтеграли та ін.) і обчислення кожного значення вимагає великих зусиль. У цьому разі 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 у будь-якій точці цього відрізка. Для цієї мети використовують задачу апроксимації (наближення) функції $f(x)$: цю функцію треба замінити наближенням (апроксимувати) деякою функцією $\varphi(x)$ так, щоб відхилення $\varphi(x)$ від $f(x)$ були найменшими. Функцію $\varphi(x)$ називають апроксимуючою. Якщо одержано цю функцію,

є можливість для будь-якого значення x обчислити відповідне значення y, приблизно таке, що дорівнює $f(x)$. Як функцію $\varphi(x)$ беруть узагальнений поліном:

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

де $\psi_i(x)$ – система деяких функцій (тригонометричних, степеневих тощо). Частіше функцію $y = f(x)$ апроксимують звичайним поліномом

$$\varphi(x) = a_0 + a_1x + ... + a_nx^n = \sum_{i=0}^{n} a_ix^i, \qquad (5.1)$$

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

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

Зрозуміло, що коефіцієнти a_i полінома (5.1) підбирають так, щоб $\psi(x)$ мало відрізнявся від $f(x)$, тобто відхилення полінома 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 побудові полінома $\psi(x)$, який набирає у заданих точках x, тих самих значень y_i, що і функція $f(x)$, тобто

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

При цьому вважається, що серед значень x_i немає однакових, тобто $x_i \neq x_j$ при $i \neq j$. Точки x_i називають вузлами інтерполяції, поліном $\varphi(x)$ – інтерполяційним поліномом.

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

Якщо один і той самий поліном (його степінь дорівнює n) інтерполює $f(x)$ на всьому відрізку $[a,b]$, то говорять про глобальну інтерполяцію.

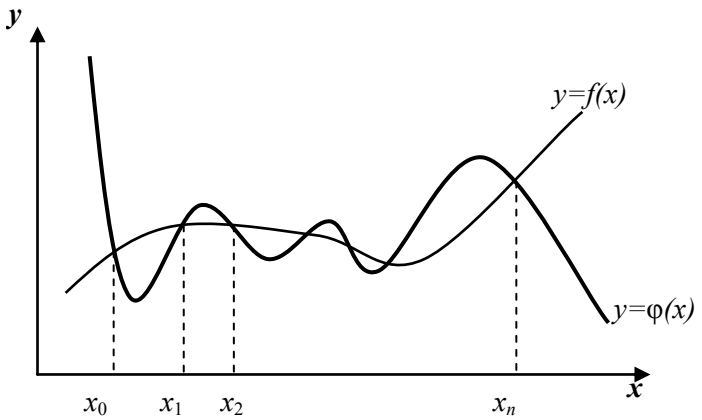


Рис. 5.1. Інтерполяція функції

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

Як правило, інтерполяційний поліном $\varphi(x)$ використовують для апроксимації функції в точках, які належать відрізку $[a,b]$. Але інколи його застосовують 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 коливний характер зі зростаючою амплітудою, а поза відрізком швидко зростає.

Обов'язкове проходження інтерполяційного полінома через табличні точки зумовлює деякі вади цього підходу. Оскільки степінь інтерполяційного полінома пов'язаний із кількістю вузлів (при $n+1$ вузлі степінь дорівнює n), то за великої кількості вузлів степінь полінома буде високим. Це збільшує час обчислення значень цього полінома 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 із ними.

Такий підхід використовують при середньоквадратичному наближенні, коли мірою відхилення полінома $\varphi(x)$ від функції $f(x)$ є значення

$$S = \sum_{i=0}^{n} \left[\varphi(x_i) - y_i \right]^2, \qquad (5.2)$$

тобто сума квадратів різниць значень полінома та функції у вузлових точках (рис. 5.2). Степінь m апроксимуючого полінома $\varphi(x)$ беруть, як правило, невисоким ($m = 1, 2, 3$), а коефіцієнти цього полінома приймають такими, щоб міра відхилення S була мінімальною. У цьому полягає метод найменших квадратів.

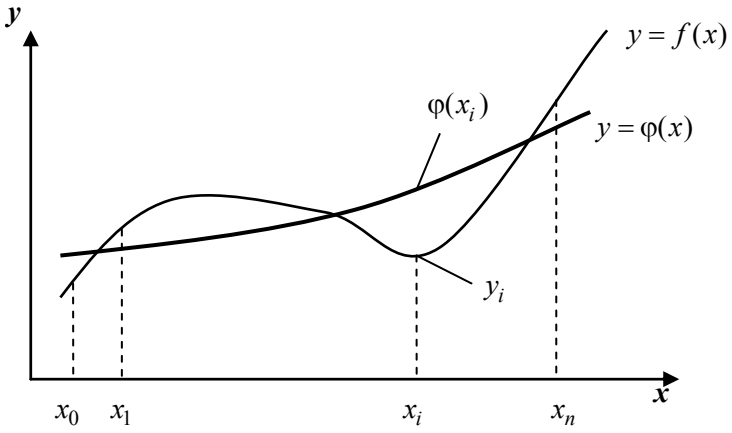


Рис. 5.2. Середньоквадратичне наближення (1-й варіант)

Поліном $\varphi(x)$, одержаний при середньоквадратичному наближенні, може в деяких вузлах значно відрізнятися від функції $f(x)$, як наприклад, у вузлі x_k на рис. 5.3.

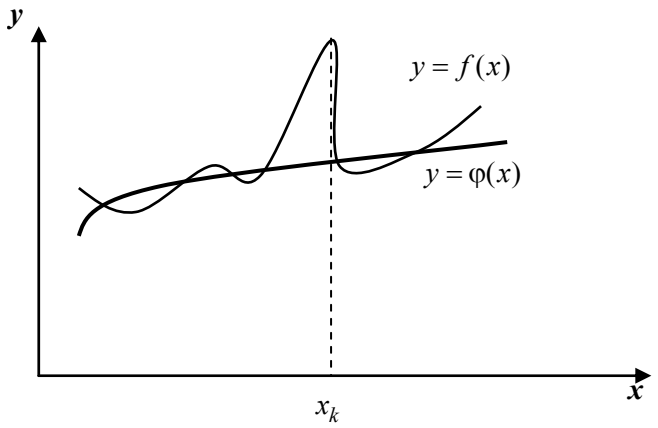


Рис. 5.3. Середньоквадратичне наближення (2-й варіант)

Тому інколи ставлять більш жорстку вимогу: поліном має відхилятися від функції $f(x)$ не більше, ніж на $\varepsilon > 0$, тобто

$$\left|f(x) - \varphi(x)\right| < \varepsilon, \quad a \le x \le b. \tag{5.3}$$

У цьому полягає задача рівномірного наближення. Тоді говорять, що поліном $\varphi(x)$ рівномірно апроксимує функцію $f(x)$ на відрізку $[a,b]$**.** Графік функції (рис. 5.4) у такому разі лежить у "трубі" завширшки 2ε відносно полінома $\varphi(x)$.

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

$$\Delta = \max\left|f(x) - \varphi(x)\right|,$$
$$a \le x \le b.$$

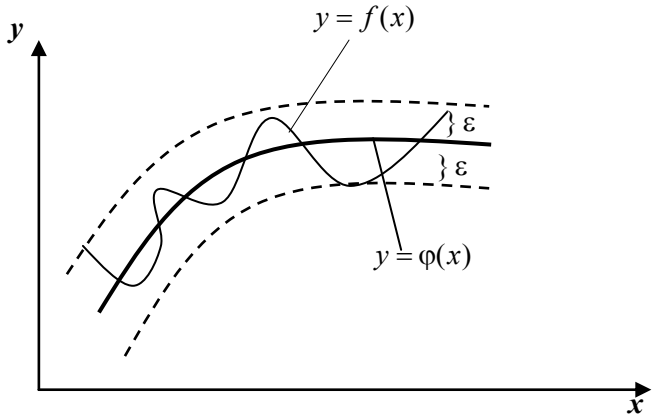


Рис. 5.4. Рівномірне наближення

Теорема 5.1. Якщо функція $f(x)$ неперервна на відрізку $[a,b]$, то для будь-якого $\varepsilon > 0$ існує поліном $\varphi(x)$ степеня $m = m(\varepsilon)$, абсолютне відхилення якого від функції $f(x)$ на відрізку $[a,b]$ менше за ε.

З іншого боку існує поняття найліпшого наближення функції $f(x)$ поліномом $\varphi(x)$ при фіксованому степені. Тут серед поліномів степеня m необхідно знайти такий, при якому абсолютне відхилення буде найменшим. Такий поліном називають поліномом найліпшого рівномірного наближення. Про його існування свідчить така теорема [3].

Теорема 5.2. Для будь-якої функції $f(x)$, неперервної на відрізку $[a,b]$, та будь-якого натурального m існує поліном $\varphi(x)$ степеня не вище за m, абсолютне відхилення якого від функції $f(x)$ мінімальне, причому такий поліном 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 якої 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 поліном $y = \varphi(x)$ такий, що $\varphi(x) \approx f(x)$.

Для розв'язання цієї задачі треба провести пряму через дві точки (рис. 5.5) та скласти її рівняння.

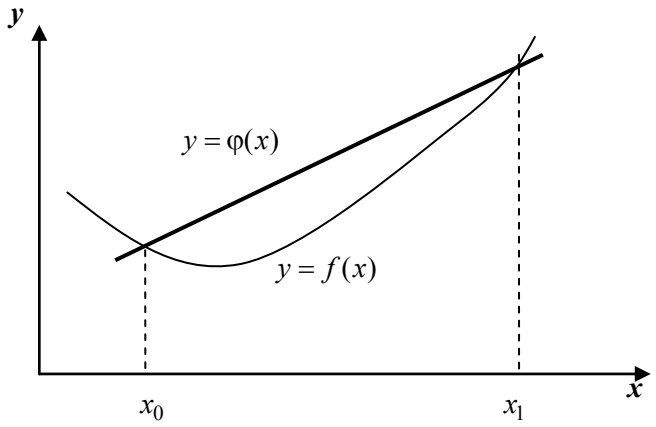


Рис. 5.5. Лінійна інтерполяція

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

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

Позначивши

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

одержуємо

$$y = a_0 + a_1 x = \varphi(x), \tag{5.6}$$

тобто рівняння зведено до стандартного вигляду запису поліномів.

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

Таким чином, для двох вузлів x_0, x_1 $(n = 1)$ можна побудувати єдиний інтерполяційний поліном $\varphi(x)$ степеня не вище першого й нескінченну множину поліномів більш високих степенів.

Це саме рівняння полінома першого степеня можна одержати інакше. Шукатимемо функцію вигляду $y = a_0 + a_1 x$, 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 від другого рівняння перше та знайдемо невідомі

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

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

які збігаються з (5.5) та (5.6).

Реалізацію цього алгоритму наведено в лістингу 5.1.

Лістинг 5.1. Реалізація алгоритму лінійної інтерполяції

```
# -*- 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
```

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. Шукатимемо інтерполяційний поліном 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 із 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

$$\begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} \begin{pmatrix} \grave{a}_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}.$$

Після її розв'язування поліном (5.7) буде визначено і ним можна скористатися для обчислення функції $y = f(x) \approx \varphi(x)$ у проміжних точках (рис. 5.6).

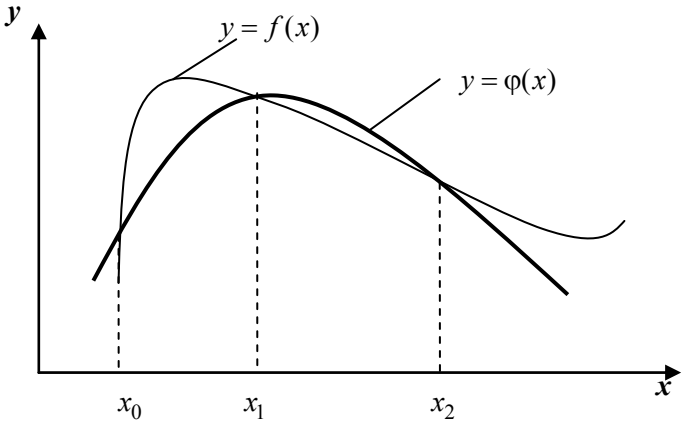


Рис. 5.6. Квадратична інтерполяція

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 нескінченну множину поліномів третього, четвертого і більших степенів.

5.2.2. Інтерполяційна формула Лагранжа

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

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

Однак чи можна розв'язати цю задачу? 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

$$\varphi(x_i) = y_i, \; i = 0,1,2,..., n. \tag{5.9}$$

Для побудови такого полінома $\varphi(x)$ достатньо підходу, який називають *методом невизначених коефіцієнтів*.

Із умови (5.9) рівності значень полінома $\varphi(x)$ у вузлах x_i відповідним табличним значенням y_i можна одержати систему $(n+1)$ рівнянь відносно $(n+1)$ невідомої $a_0, ..., a_n$:

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

аналогічну простим системам п. 5.2.1. Матрицю цієї системи

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

називають матрицею Вандермонда. У літературі показано, що її визначник не дорівнює нулю, якщо серед вузлів 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 a_i інтерполяційного полінома.

Такий підхід інколи корисний для теоретичних досліджень, але для практичного використання він мало придатний унаслідок своєї трудомісткості та поганої зумовленості системи при $n \geq 5$.

Єдиний інтерполяційний поліном, про який ішлося в теоремі 5.3, можна побудувати кількома способами. Розглянемо лише одну форму запису цього полінома – *формулу Лагранжа.*

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

$$l_i(x_j) = \begin{cases} 1 & \text{при} \quad i = j, \\ 0 & \text{при} \quad i \neq j. \end{cases} \tag{5.12}$$

Тобто поліном $l_j(x)$ із номером i дорівнює одиниці тільки у вузлі з таким самим номером x_j, а в інших вузлах він дорівнює нулю (рис. 5.7). Для деякої нерівномірної сітки $x_i \quad (i = 0,1,...,n)$ кожний із $(n+1)$-го поліномів $l_i(x)$ можна подати у вигляді

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

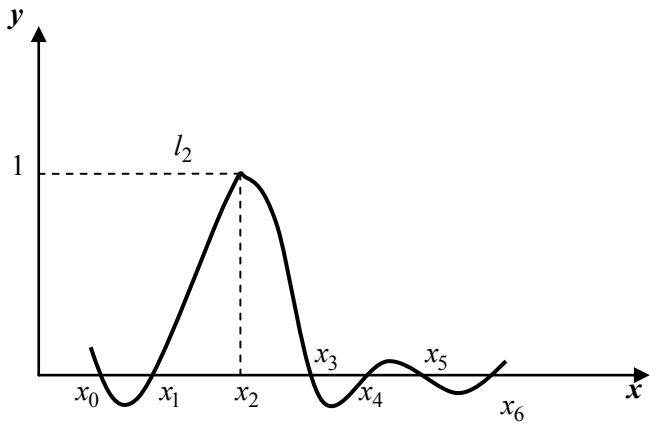


Рис. 5.7. Поліном Лагранжа

Перевіряючи, можна впевнитися, що такий поліном справді дорівнює нулю в усіх вузлах, окрім 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 $x = x_i$ в (5.13):

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

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):

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

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

$$\varphi(x) = y_0 l_0(x) + y_1 l_1(x) + ... + y_n l_n(x) = \sum_{i=0}^{n} y_i l_i(x) =$$

$$= \sum_{i=0}^{n} y_i \frac{(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)} = L_n(x). \quad (5.15)$$

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

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

У цій сумі відповідно до (5.12) усі 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 й уся 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

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

і 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 $y = f(x)$, яку задано таблицею із трьох точок:

x	0	1	3
y	2	0	1

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

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

У другому разі підставимо в (5.16) тільки табличне значення й 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. 5.8 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 разі, коли функція $f(x)$ є поліномом n-го степеня, одержаний за допомогою формули (5.15) інтерполяційний поліном $L_n(x)$ за відсутності помилок округлення збігається з $f(x)$ на всій числовій осі.

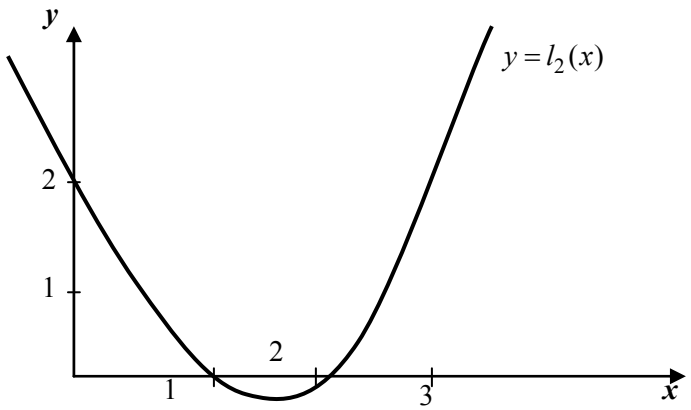


Рис. 5.8. Інтерполяція за трьома точками

У випадку довільної функції $f(x)$ збіг буде тільки у вузлах інтерполяції, а в проміжних точках різниця між $f(x)$ та $L_n(x)$ не 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 $a = x_0, x_1, ..., x_n$ – різні вузли.

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

$$L_n(x_i) = f(x_i), \;\; i = 0,1,...,n,$$

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

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

де z – деяка точка відрізка $[a,b]$.

Позначимо $M = \max\left|f^{(n+1)}(z)\right|$.

Тоді

$$a \le z \le b,$$
$$\left|R_n(x)\right| \le \frac{\left|(x-x_0)(x-x_1)...(x-x_n)\right|}{(n+1)!} M. \quad (5.19)$$

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

Розглянемо на прикладі використання оцінки похибки (5.19).

Приклад 5.2. З якою точністю можна обчислити $\sqrt{117}$ за допомогою інтерполяційної формули Лагранжа для функції y:

$$a_2(A_1x_2+B_1)+b_2x_2+c_2x_3=d_2,$$
$$x_2(a_2A_1+b_2)+c_2x_3+a_2B_1=d_2,$$
$$x_2=\frac{d_2-c_2x_3-a_2B_1}{a_2A_1+b_2}=-\frac{c_2}{a_2A_1+b_2}\cdot x_3+\frac{d_2-a_2B_1}{a_2A_1+b_2}=A_2x_3+B_2,$$

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

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

Знайдемо максимальне значення y''' на відрізку $[100,144]$. Це спадна функція, тому вона досягає максимуму 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

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

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

Складемо програму мовою *Python* для наближення функції інтерполяційною формулою Лагранжа (лістинг 5.2). Тут можливі два підходи.

Перший – це підстановка, як і в прикладі 5.1, у формулу (5.15) значення аргументу та табличних значень одночасно. Він простий з погляду програмування, але тут не формується поліном $\varphi(x)$ у канонічному вигляді (5.8) і тому кількість арифметичних операцій для обчислення $L_n(x)$ велика. У наведеному далі фрагменті програми (лістинг 5.2) масиви X та Y містять таблицю значень функції, $(N+1)$ – довжина таблиці, XR – масив робочих точок, для яких треба обчислити значення полінома. Початкові дані накопичуються в масиві YR.

Лістинг 5.2. Фрагмент програми
для інтерполяції методом Лагранжа

```
…
for K in range(1,MR+1):
    Z=XR[K]
    F=0
    for l 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
…
```

Підрахуємо кількість арифметичних операцій, потрібних для обчислення $L_n(x)$ за формулою (5.15) на підставі таблиці, яка складається з $(n+1)$ точок. 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 у формулі Лагранжа є $(n+1)$ таких дробів, то для їх обчислення та накопичення суми потрібно $4n(n+1)+n=4n^2+5n$ операцій.

Якщо цю таблицю буде використано 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 $\varphi(x)$ 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 поліномів.

5.2.3. Обчислення значень поліномів

Необхідність обчислення значень поліномів

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

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

i	1	2	3	...	$n+1$
$A =$	a_0	a_1	a_2	...	a_n

Послідовність розміщення в ньому коефіцієнтів може бути 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[l+1]*X**l
…
```

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 в алгоритмі

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

У цьому разі слід виконати лише $3n$ арифметичних операцій.

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

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

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 полінома n-го степеня із застосуванням менше, ніж $2n$ арифметичних операцій. Тому далі використовуватимемо 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 полінома за формулою Лагранжа

Якщо передбачається багаторазове використання 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 поліном 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 поліноми другого, третього і більших степенів, кожен із яких має бути помноженим на двочлен, розглянемо таку допоміжну задачу: складемо програму множення полінома степеня l

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

на двочлен $(x - c)$.

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

i	1	2	3		$l+1$	
$A =$	a_0	a_1	a_2	...	a_l	...

Коефіцієнти полінома $(x - c)P(x)$, тобто результат, накопичуватимемо в масиві B:

i	1	2	3		$l+1$	$l+2$	
$B =$	b_0	b_1	b_2	...	b_l	b_{l+1}	...

Оскільки

$$\left(\begin{array}{c} \left(a_0 + a_1 + ... + a_l x^l\right)(x - c) = \\ = x(a_0 + a_1 + ... + a_l x^l) - c\left(a_0 + a_1 + ... + a_l x^l\right) = \\ = a_0 x + a_1 x_2 + ... + a_l x^{l+1} - c\left(a_0 + a_1 + ... + a_l x^l\right) \end{array} \right),$$

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 зображено на рис. 5.9.

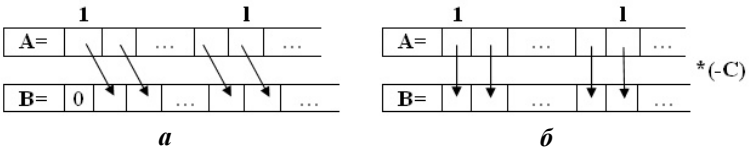


Рис. 5.9. Схема алгоритму: *а* – етап 1; *б* – етап 2

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 поліном степеня n у вигляді масиву його коефіцієнтів будемо додавати до аналогічних поліномів, одержаних для інших дробів, та накопичувати коефіцієнти результуючого полінома $L_n(x)$ 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 з робочих точок, які зберігаються в масиві **XR**). Нехай початкова таблиця значень невідомої функції складається з M точок. Запишемо програму мовою *Python* (лістинг 5.3).

Лістинг 5.3. Реалізація алгоритму

```
# -*- 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 по-
лінома"""
        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
```

Перевірка алгоритму на даних із прикладу 5.1 дала ті самі результати. Як видно з останнього рядка, 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 наведено в лістингу 5.4. Масиви X та Y репрезентують функцію, яка задана таблично. Параметр **argx** задає точку, в якій потрібно знайти значення функції.

Лістинг 5.4. Реалізація методу Лагранжа

```
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
```

Перевірка алгоритму на тих самих даних дала ті самі результати. Як видно з останнього рядка, 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. Скінченні різниці різних порядків

Ми припускатимемо зараз, що початкова таблиця значень функції має рівновіддалені вузли, тобто $x_{i+1} - x_i = \Delta x_i = h = \text{const}$ $(i = 0,1,2,...,n-1)$. Константа h називається кроком таблиці.

При розв'язуванні поставлених задач інтерполяції велике значення мають так звані *скінченні різниці* даної функції $f(x)$. Скінченними різницями 1-го порядку називаються величини:

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

Якщо кількість вузлів інтерполяції дорівнює $n+1$, то маємо n скінченних різниць 1-го порядку. З огляду на них складають $n-1$ скінченну різницю 2-го порядку:

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

Узагалі скінченна 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 $(n+1-k)$. 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 скінченних різниць, тобто оператора Δ:

1. Якщо $C = \text{const}$, то $\Delta C = 0$.
2. $\Delta\left[f_1(x) + f_2(x)\right] = \Delta f_1(x) + \Delta f_2(x)$.
3. $\Delta\left[Cf(x)\right] = C\Delta f(x)$.

Скінченні різниці дозволяють мати інформацію про важливу для інтерполяції "ступінь гладкості" цієї функції $f(x)$. При цьому велику роль відіграють наступні теореми.

Теорема 5.4. Якщо $y = f(x)$ – поліном степеня n зі старшим членом $a_0 x^n$, то для будь-якого $k \le n$ скінченна різниця $\Delta^k y$ є поліном від x степеня $n-k$ зі старшим членом: $n(n-1)(n-2)\ldots(n-k+1)a_0 h^k x^{n-k}$, де h – крок таблиці.

Справді, нехай $y = f(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1}x + a_n$.

Тоді згідно з властивостями оператора Δ маємо

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

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

Із цих виразів отримаємо поліном, старший член якого має вигляд $a_0 nhx^{n-1}$. Таким чином, $(1-a)$ – різниця полінома степеня n зі старшим членом a_0 є поліномом степеня $n-1$ зі старшим членом $a_0 nhx^{n-1}$**.** Продовжуючи такі обчислення послідовно для різниць будь-якого порядку, можна переконатися в правильності теореми.

Наслідок 5.1. Для полінома $y = f(x)$ степеня n скінченна різниця n-го порядку є стала величина, рівна $\Delta^n y = n! a_0 h^n$, а всі різниці вищого порядку, ніж n, тотожно рівні нулю.

Цікавою з погляду зв'язку дискретного і безперервного аналізів є залежність між скінченними різницями і похідними функції. Оскільки

$$f'(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} = \lim_{h\to 0}\frac{\Delta y}{h}, \text{ то } f'(x) \approx \frac{\Delta y}{h}.$$

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

5.2.6. Поняття про поділені різниці

Нехай вузли інтерполяції $x_0, x_1, ..., x_n$ не обов'язково рівновіддалені. У цьому разі аналогічну скінченним різницям роль грають так звані поділені різниці, або "підйоми" функції.

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

$$[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0},\ [x_1, x_2] = \frac{y_2 - y_1}{x_2 - x_1},\ \ldots,$$

взагалі

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

Поділеними різницями 2-го порядку називаються величини

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

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

Поділені різниці k-го порядку визначаються через поділені різниці $(k-1)$-го порядку за допомогою рекурентного співвідношення

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

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

Надалі корисно мати вираз для значення функції $f(x_k)$ у довільному вузлі інтерполяції x_k через значення функції $f(x_0)$ в початковому вузлі x_0 і початкові значення поділених різниць $[x_0, x_1], [x_0, x_1, x_2], [x_0, x_2, x_3], \ldots$ Ці значення виводяться за методом індукції і мають вигляд

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

Якщо таблиця значень функції має сталий крок h, то за методом індукції легко встановити зв'язок між поділеними і скінченними різницями одного й того самого порядку k:

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

Для поділених різниць виконуються теореми, аналогічні теоремам про скінченні різниці і відповідним наслідкам із них. Сформулюємо найважливіше твердження.

Теорема 5.5. Якщо $f(x)$ – поліном степеня *n,* то поділені різниці *n*-го порядку є константами, не залежними від вузлів $x_0, x_1, ..., x_n$ і рівними коефіцієнту при старшому степені x, що у поліномі $f(x)$. Усі поділені різниці більшого, ніж n, порядку рівні 0.

Ця теорема лежить в основі практичного правила, що дозволяє вибирати степінь інтерполяційного полінома в разі нерівновіддалених вузлів так, щоб він збігався з порядком практично сталих поділених різниць функції.

Для осмислення зв'язку дискретного і безперервного аналізів треба мати на увазі таке. В інтервалі (x_0, x_k) існує точка ξ така, 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 оцінювань можна використовувати співвідношення: $f^{(k)}(x) \approx k![x_0, x_1, ..., x_k]$.

5.2.7. Інтерполяційна формула Ньютона

Істотною вадою поліномів Лагранжа, як було зазначено, є те, що вони часто значно ускладнюють практичні обчислення. Інше подання, основане на використанні поділених різниць,

дає можливість послідовно уточнювати результати інтерполяції і часто не вимагає попереднього знання степеня полінома.

Теорема 5.6. Інтерполяційний поліном $P_k(x)$ можна записати у вигляді, який називається формулою Ньютона:

$$P_k(x) = y_0 + [x_0, x_1](x - x_0) + [x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots$$
$$\ldots + [x_0 x_1, \ldots, x_k](x - x_0)(x - x_1)\ldots(x - x_{k-1}).$$

Доведення. По-перше, ясно, що вказаний поліном має степінь не вищий k. По-друге, він набуває значень функції у вузлах інтерполяції. Справді, $P(x_0) = y_0$, оскільки при $x = x_0$ всі члени полінома, починаючи з другого, перетворюються на нуль. При $x = x_1$ із таких самих причин на підставі виразу

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

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

Формули Лагранжа і Ньютона дають лише різні форми запису одного і того самого інтерполяційного полінома. Проте формула Ньютона зручна тим, що при додаванні до вузлів $x_0, x_1, \ldots, x_k$ нового вузла $xk+1$ усі раніше знайдені члени залишаються без зміни й у поліном лише додається один додатковий член вигляду $[x_0 x_1, \ldots, x_{k+1}](x - x_0)(x - x_1)\cdots(x - x_k)$. Це дозволяє, послідовно додаючи поодинці додаткові вузли, поступово нарощувати точність результату інтерполяції.

Оскільки табличні різниці швидко зменшуються зі збільшенням порядку, то найближчі до цієї точки x вузли інтерполяції дадуть основний внесок у шукану величину, а інші даватимуть лише невеликі поправки. У цьому разі легше уникнути помилок і встановити, на якій різниці слід закінчити обчислення.

Через тотожність поліномів Лагранжа і Ньютона залишковий член $R_k(x)$ у них однаковий. Із погляду зв'язку між аналізами дискретним і безперервним цікаво відзначити аналогію подання функції $f(x) = P_k(x) + R_k(x)$, де права частина визначається формулами Лагранжа і Ньютона, із відомою формулою Тейлора. Добуток різниць $(x - x_0)(x - x_1)\cdots(x - x_k)$ є узагальненням степеня бінома, а поділені різниці виступають як узагальнені похідні. Якщо вузли $x_0, x_1, \ldots, x_k$ 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 на формулу Тейлора. Таким чином, формулу Ньютона можна розглядати як узагальнення формули Тейлора на випадок дискретного аналізу.

Реалізацію методу інтерполяції за формулою Ньютона у вигляді окремого модуля *Python* наведено в лістингу 5.5.

Лістинг 5.5. Реалізація методу інтерполяції за формулою Ньютона

```
# -*- 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
```

Розглянемо використання створеного модуля (лістинг 5.5) на прикладі.

Приклад 5.3.

Функцію $f(x) = 4{,}8\cos\dfrac{\pi x}{20}$ задано 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 методом Ньютона значення функції в точках $x = 0;\ 0{,}5;\ 1{,}0;\ \ldots;\ 8{,}0$ і порівняти результати з точними даними $y_i = f(x_i)$. Програму (написану з використанням раніше розробленого модуля), яка розв'язує поставлену задачу, наведено в лістингу 5.6.

Лістинг 5.6. Використання методу Ньютона

```
# -*- 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 enter для виходу")
```

Наведемо результат роботи програми.

```
>>>
 xyInterp 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 Enter для виходу
```

Як бачимо, результати майже не відрізняються від точних.

5.2.8. Інтерполяція для рівновіддалених вузлів

Розглянемо важливий окремий випадок, коли $x_{i+1} - x_i = h = \text{const},\ \ i = 0,1,...,k-1$. Якщо ввести нову змінну $x = x_0 + t_h$, то при будь-якому h вузли інтерполяції завжди набувають стандартних значень відповідно до $t_0 = 0,\ t_1 = 1,\ t_2 = 2,...,\ t_k = k$.

Змінна t має сенс кількості кроків h від x_0 до x. Це дозволяє уніфікувати відповідні формули інтерполяції. Так, коефіцієнти Лагранжа вдається подати у вигляді

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

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 набуває вигляду

$$P_k(t) = \sum_{i=1}^{k} y_i L_i(t).$$

Формула Ньютона також спрощується до вигляду

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

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

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 поділених різниць. За початковими даними 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. Кількість використовуваних різниць вищих порядків може бути будь-якою, але чим вона більша, тим вища точність.

Одне з достоїнств цього методу полягає в тому, що він дозволяє уточнювати результат, використовуючи додаткові різниці, причому немає необхідності починати обчислення спочатку. Тому в разі, якщо невідомо, скільки членів слід узяти, їх кількість можна збільшувати доти, поки їхнім внеском не можна буде знехтувати. У цьому разі $h = 10°$. Використовуючи тільки першу різницю, знайдемо

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

Увівши 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 близьке до табличного (точного) значення, рівного $0{,}39073$.

5.3. Багатоінтервальна інтерполяція

5.3.1. Властивості багатоінтервальної інтерполяції

Як указувалося 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. Із цими труднощами стикаються під час інтерполяції на великих відрізках: при великих відстанях між вузлами точність дуже мала, а в разі зменшення цих відстаней збільшується кількість вузлів і відповідно степінь полінома.

У таких випадках доцільно інтерполювати 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. 5.10). Усі 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 поліном $g_k(x)$. Оскільки на кожному підінтервалі кількість вузлів невелика, то степінь полінома буде невисоким.

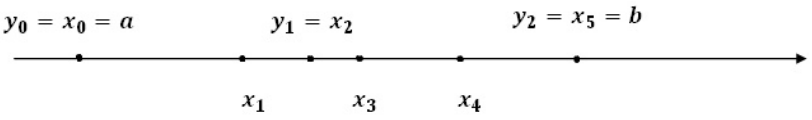


Рис. 5.10. Розбиття відрізка при кусковій інтерполяції

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 подати у такому 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 стикування підінтервалів), за допомогою якої-небудь інтерполяційної формули обчислити $y = g_k(x) \approx f(x)$.

Багатоінтервальна інтерполяція має такі властивості.

1. Степінь інтерполяційного полінома не залежить від кількості вузлів. Справді, зі зростанням кількості вузлів можна збільшити кількість m точок розбиття γ_k, тобто кількість підінтервалів, із тим, щоб кількість вузлових точок xk на кожному підінтервалі не збільшувалась.

2. При незмінному відрізку $[a,b]$ помилка інтерполяції зі зростанням кількості вузлів прямує до нуля, що зумовлено зменшенням відстані між точками.

3. Час обчислень невеликий унаслідок низького степеня полінома. Зазначимо, що хоча функція $g_k(x)$ є неперервною, у точках γ_k стикування підінтервалів, як правило, має розрив уже 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 (рис. 5.11),

яка й апроксимує функцію $y = f(x)$. Таким чином, при кусково-лінійній інтерполяції точки розбиття γ_k збігаються з вузлами x_i. Функції $g_k(x)$ для кожного з n відрізків $\{x_i, y_i\}$ є поліномами першого степеня і можуть бути одержані з рівняння прямої (див. п. 5.2.1), яка проходить через точки (x_k, y_k) та (x_{k+1}, y_{k+1}):

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

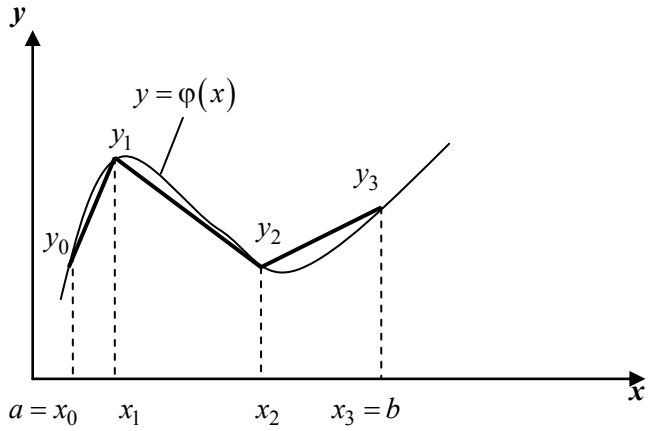


Рис. 5.11. Кусково-лінійна інтерполяція

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 формулами (5.24) обчислити $y = g_k(x) \approx f(x)$.

При програмуванні цього алгоритму слід урахувати випадок, коли x міститься за межами табличних значень, тобто $x < x_0$ або $x > x_n$. Залежно від постановки задачі вищого рівня тут може передбачатися:

• виведення діагностичного повідомлення про некоректне значення x;

• продовження ламаної поза відрізком $[a,b]$ горизонтальними прямими $y = y_0$ та $y = y_n$;

• продовження першого й останнього відрізків ламаної при $x < a$ або $x > b$.

Наведемо приклад програми, де передбачено третій варіант, тобто при $x < a$ значення y обчислюють за тією самою формулою, що і для x: $A, B, x^{(k-1)}, x^{(k-2)}, ..., x^{(k-r)}$ $[x_0, x_1]$, а при $x > b$ за тією самою формулою, що і для x: $F_k = Hx^{(k-1)} + V$ $[x_{n-1}, x_n]$.

У цій програмі (лістинг 5.7) початковими даними є:

• таблиця значень функції у вигляді масивів X та Y;

• робоча точка XR.

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

Лістинг 5.7. Фрагмент реалізації алгоритму

```
…
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;
…
```

5.3.3. Кусково-нелінійна інтерполяція

Як зазначалося в п. 5.3.1, увесь відрізок $[a,b]$, на якому задано таблицю значень функції $y = f(x)$, може бути розбитий на частинні відрізки, кожний з яких має невелику кількість то-

чок x_i. На кожному частинному 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 частинний відрізок містить тільки дві точки і тут для апроксимації використовувались поліноми першого степеня, яким відповідають прямолінійні відрізки.

Але часто трапляється, що потрібно збільшити точність апроксимації. Це забезпечують використанням поліномів хоча б другого степеня, тобто замість відрізків прямих з'єднують табличні точки відрізками парабол. Якщо кожен частинний 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 розглядатимуться в п. 5.3.4. Якщо ж ніяких додаткових умов не накладати, тоді для інтерполяції функції кусковим поліномом другого степеня в кожен частинний відрізок слід об'єднувати три сусідні точки (дві крайні та одну внутрішню). Аналогічні дії виконують при побудові інтерполяційної кусково-нелінійної функції більш високого степеня: у частинний відрізок включають чотири точки при використанні полінома третього степеня і т. д.

Для ілюстрації такого підходу розглянемо приклад.

Приклад 5.5. Невідому функцію $y = f(x)$ задано на відрізку $[0,1]$ таблицею із семи точок:

x	0	1/6	1/3	1/2	2/3	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 частинні відрізки по три точки в кожному. На кожному з цих частинних відрізків $[0, \frac{1}{3}]$, $[\frac{1}{3}, \frac{2}{3}]$, $[\frac{2}{3}, 1]$ за формулою 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 $l_2 = [\frac{1}{3}, \frac{2}{3}]$ скористаємося цією самою функцією, але як величину x_0, x_1, x_2 тепер використовуватимемо чергові точки таблиці $x_0 = \frac{1}{3}$, $x_1 = \frac{1}{2}$, $x_2 = \frac{1}{2}$, $y_0 = 1$, $y_1 = 1$, $y_2 = 0$:

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

Одержимо поліном першого, а не другого степеня. Це пов'язано з тим, що на відрізку l_2 три точки лежать на одній прямій. На третьому частинному відрізку $l_3 = [\frac{2}{3}, 1]$ як вузли інтерполяції використовуватимуться 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 l_3, одержимо в результаті кусково-параболічну функцію (рис. 5.12):

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

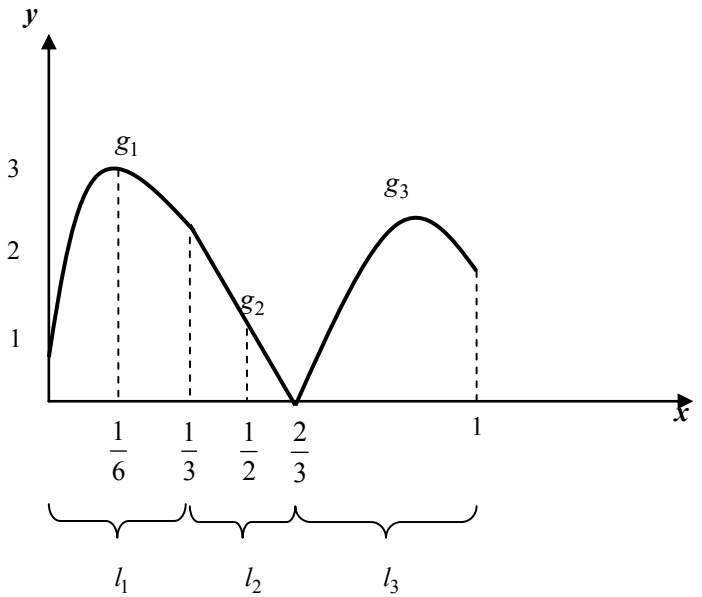


Рис. 5.12. Кусково-параболічна інтерполяція

Виникає природне запитання про точність апроксимації функції $f(x)$ одержаною функцією $g(x)$. Як відомо (див. п. 5.2.2), помилку апроксимації поліномом Лагранжа при рівновіддалених вузлах визначають із співвідношення

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

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

$$R = \left| f(x) - g(x) \leq Mh^3 = \frac{M}{6^3} \right|,$$

$$\text{де } M = \max_{0 \leq z \leq 1} \left| f'''(z) \right|.$$

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 розмір кроку. Якщо зменшимо крок удвічі, тобто розіб'ємо відрізок $[0,1]$ не на три, а на шість підінтервалів, то помилка апроксимації зміниться:

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

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

Із рис. 5.12 видно, що в точках стикування поліномів $g_k(x)$ функція $g(x)$ не є диференційовною. У ряді випадків це може бути суттєвим і тоді використовують спеціальну форму кускової апроксимації – сплайни, які розглядаються далі.

5.3.4. Параболічні сплайни

Сплайном називають задану кусково функцію, яка разом із декількома похідними неперервна на всьому відрізку $[a,b]$, а на кожному частинному відрізку $\left[x_j, x_{i+1} \right]$ окремо є деяким алгебраїчним поліномом.

Максимальний за всіма частинними відрізками степінь поліномів 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* у цьому вузлі. Назва такого роду кривих походить від англійського слова *spline* – гнучка лінійка, що використовується креслярами для проведення ліній. 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 потенціальну енергію. Одержана крива математично є природним кубічним сплайном, який розглядається в п. 5.3.5. Виконаємо побудову простих сплайнів – сплайнів другого степеня з дефектом, що дорівнює одиниці.

У п. 5.3.3 при кусково-параболічній інтерполяції ми розбивали відрізок $[a,b]$, на якому задано таблицю $\{x_i, y_i\}$, на частинні відрізки по три вузли в кожному. Ці три вузли однозначно визначають параболу. Набір таких парабол і є інтерполяційною функцією. Оскільки правий вузол чергової трійки збігався з лівим вузлом наступної трійки вузлів, то внаслідок вимоги інтерполяції набір парабол у цілому виявився неперервною функцією. Однак, як видно з рис. 5.12, параболи не переходять плавно одна в одну, а утворюють у точках стикування злом, тобто кускова функція не є диференційовною в цих точках функцією. На кожному частинному відрізку три умови інтерполяції $\left(g_k(x_i) = y_i\right)$ повністю вичерпують можливості накладання обмежень на параболу, оскільки вона характеризується своїми трьома коефіцієнтами. Тому, якщо потрібно побудувати диференційовну 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. 5.13). Як частинні відрізки виберемо $l_i = [x_i, x_{i+1}]$, $i = 1,2,3$.

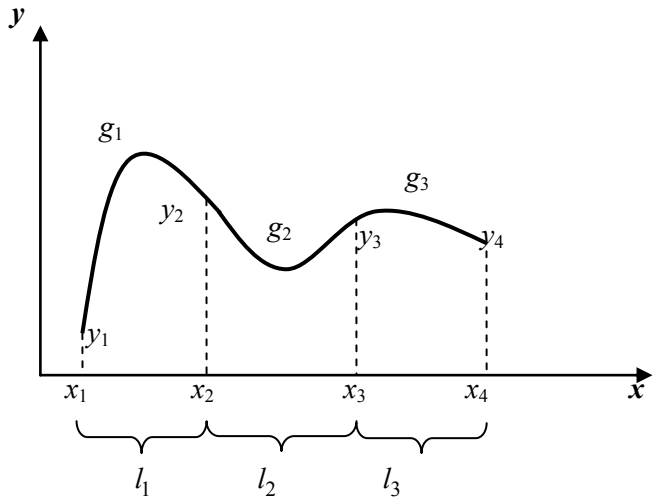


Рис. 5.13. Параболічний сплайн

На кожному частинному відрізку маємо лише два вузли і для визначення трьох коефіцієнтів параболи можна скласти тільки два рівняння. Тому з'являється 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:

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

Щоб кускова функція була диференційована в точках стикування частинних відрізків, мають виконуватися також такі дві умови:

$$g_1^l(x_2) = g_2^l(x_2),$$
$$g_2^l(x_3) = g_3^l(x_3). \tag{5.27}$$

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

$$g_1^l(x_1) = d, \tag{5.28}$$

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

$$g(x) = g_i(x) \text{ при } x \in l_i.$$

У загальному випадку, коли функція $y = f(x)$ задана таблицею в n точках, сплайн будують аналогічно. Множину вузлів розбиваємо на $(n-1)$ частинний 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:

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

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

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

У результаті маємо $2(n-1)+(n-2)=3n-4$ рівнянь із $3n-3$ 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., то для докладного запису рівнянь зручно використати степеневу форму. Перші $2(n-1)$ рівнянь (5.29), що забезпечують вимоги інтерполяції, мають вигляд

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

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

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 сплайна, 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:

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

Незаповнені позиції матриці дорівнюють нулю. Наведемо програму мовою *Python* для побудови та використання параболічного сплайна.

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

Розв'язання. 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 і B, розв'язує СЛАР та формує вектор коефіцієнтів сплайна у вигляді масиву AS. Функція **PRES** обчислює $g(z_i)$ та друкує результати. Зважаючи на те, що для обчислення значень сплайна зручніше користуватися двовимірним, а не одновимірним масивом, на початку цього модуля масив AS перетворюється на масив SP такої структури (рис. 5.14):

SP=	a_{12}	a_{11}	a_{10}	Коефіцієнти 1-ї параболи
	a_{22}	a_{21}	a_{20}	Коефіцієнти 2-ї параболи
	…	…	…	…
	a_{92}	a_{91}	a_{90}	Коефіцієнти 9-ї параболи

Рис. 5.14. Структура масиву

Реалізацію програми подано в лістингу 5.8.

Лістинг 5.8. Реалізація фрагмента програми мовою *Python*

```
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)
```

5.3.5. Кубічні сплайни

При апроксимації розв'язків диференціальних рівнянь, а також в інших випадках потрібно, щоб апроксимуюча функція була принаймні двічі безперервно диференційовна. Досягти цього за допомогою квадратичного сплайна, поданого в п. 5.3.4, у загальному випадку неможливо. Тому розглянемо кусково-кубічний поліном $S(x)$ – кубічний сплайн, який має такі властивості:

- двічі безперервно диференційовний;
- на кожному частинному відрізку є кубічним поліномом.

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

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

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 $(n-1)$ частинних відрізки $l_i i = 1,...,n-1$. На кожному з них маємо 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 $S_i(x)$ має проходити через табличні точки свого частинного 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 сплайна в точках сполучення частинних відрізків (таких точок $n-2$) породжує ще $2(n-2)$ рівняння:

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

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

Якщо вимагати, щоб кривизна сплайна 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 сплайна. Однак матриця цієї системи має велику розмірність, хоча і є розрідженою.

Розглянемо інший спосіб побудови кубічного сплайна, в якому треба розв'язати СЛАР розмірності лише $(n-2)$ відносно других похідних сплайна у внутрішніх вузлах таблиці.

Кубічні поліноми, що утворюють сплайн, шукатимемо у вигляді

$$S_i(x) = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3,\quad x \in [x_i, x_{i+1}],\ i = 1,2,...,n-1. \qquad (5.36)$$

Похідні сплайна:

$$S_i^l(x) = b_i + 2c_i(x-x_i) + 3d_i(x-x_i)^2,\ S_i^{ll}(x) = 2c_i + 6d_i(x-x_i).$$

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 $h_i = x_{i+1} - x_i$:

$$a_i = y_i, \qquad i = 1,...,n-1, \tag{5.37}$$

$$a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = y_{i+1}, \qquad i = 1,...,n-1, \tag{5.38}$$

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1}, \qquad i = 1,...,n-2, \tag{5.39}$$

$$c_i + 6d_i h_i = 2c_{i+1}, \qquad i = 1,...,n-2, \tag{5.40}$$

$$c_i = 0, \tag{5.41}$$

$$2c_{n-1} + 6d_{n-1}h_{n-1} = 0. \tag{5.42}$$

Знайдемо із (5.40) та (5.42) величини d_i:

$$d_i = \frac{2c_{i+1} - 2c_i}{6h_j} = \frac{c_{i+1} - c_i}{h_j},\ i = 1,...,n-2,\ d_{n-1} = \frac{-c_{n-1}}{3h_{n-1}}; \tag{5.43}$$

підставимо d_i та a_i в (5.38):

$$y_i + b_i h_i + c_i h_i^2 + \frac{c_{i+1}c_i}{3h_i} h_i^3 = y_{i+1},\ i = 1,...,n-2;$$

$$y_{n-1} + b_{n-1}h_{n-1} + c_{n-1}h_{n-1}^2 + \frac{c_{n-1}}{3h_{n-1}} h_{n-1}^3 = y_m.$$

Із 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, у (5.39) підставляємо одержані вирази для b_i та d_i:

$$\frac{y_{i+1}-y_i}{h_i}-\frac{c_{i+1}+2c_i}{3}h_i+2c_ih_i+3\frac{c_{i+1}-c_i}{3h_i}h_i^2=$$

$$=\frac{y_{i+2}-y_{i+1}}{h_{i+1}}-\frac{c_{i+2}+2c_{i+1}}{3}h_{i+1},\ i=1,\ldots,n-3,$$

$$\frac{y_{n-1}-y_{n-2}}{h_{n-2}}-\frac{c_{n-1}+2c_{n-2}}{3}h_{n-2}+2c_{n-2}h_{n-2}+3\frac{c_{n-1}-c_{n-2}}{3h_{n-2}}h^2{}_{n-2}=$$

$$=\frac{y_n-y_{n-1}}{h_{n-1}}-\frac{2}{3}c_{n-1}h_{n-1}.$$

Одержали систему $(n-2)$ рівнянь відносно величин c_i $\left(i=1,:..,n-1\right)$. Оскільки з (5.41) $c_i=0$, то кількість невідомих збігається з кількістю рівнянь. Після нескладних перетворень система набирає вигляду

$$\begin{bmatrix} 2(h_1+h_2) & h_2 & & \\ h_2 & 2(h_1+h_2) & h_3 & \\ & & \ldots & \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix}\begin{bmatrix} c_2 \\ c_3 \\ \ldots \\ c_{n-1} \end{bmatrix}=\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \ldots \\ \gamma_{n-2} \end{bmatrix},\quad (5.45)$$

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 сплайна у вигляді (5.36) необхідно виконати такі дії.

1. Розв'язавши систему (5.45), знайти величини $c_2,\ldots,c_{n-1}$. Узяти $c_1=0$.

2. За формулами (5.43) обчислити $d_1,...,d_{n-1}$.

3. За формулами (5.44) обчислити $b_1,...,b_{n-1}$.

4. Узяти $a_1=y_1\,|,\ i=1,...,n-1$.

Відмітимо, що за коефіцієнтами сплайна, побудованого у такому вигляді, легко оцінити його похідні у вузлах таблиці:

$$b_i = S_i^l(x_i), c_i = \frac{S_i^{ll}(x_i)}{2}, i = 1,...,n-1$$

Розглянемо ще один підхід до формування інтерполяційних кубічних сплайнів, відповідно до якого на кожному частинному відрізку $[x_i, x_{i+1}]$ $(i = 1,...,n-1)$ кубічний поліном 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 ($S^l(x_i) = m_i$; h – відстань між вузлами x_i), які в цьому підході беруть рівновіддаленими.

Перевіркою можна переконатися, що сплайн у формі (5.46) справді є інтерполяційним, тобто $S_j(x_i) = y_i$, $S_j(x_{i+1}) = y_{i+1}$, а його похідні у вузлах дорівнюють нахилам: $S^l{}_i(x_i) = m_i$, $S^l{}_{i+1}(x_{i+1}) = m_{i+1}$.

Звідси випливає, що при заданій таблиці $\{x_i, y_i\}$ та нахилах набір кубічних поліномів вигляду (5.46) є неперервною та диференційовною кусковою функцією, тобто сплайном третього степеня з дефектом не більше ніж два.

Існують такі способи задання нахилів.

1. Виходячи із заданої таблиці, обчислюють нахили за формулами числового диференціювання 2-го 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 $y^l{}_i - = f^l(x_i)$, то беруть $m_i = y^l{}_i,\ i = 1,...,n$.

Два наведені способи задання нахилів називають локальними, бо на кожному частинному відрізку сплайн будується окремо. При цьому неперервність другої похідної не гарантується і дефект такого сплайна звичайно дорівнює двом.

3. Формується СЛАР відносно нахилів. Її рівняння одержують з умов неперервності другої похідної у внутрішніх вузлах таблиці:

$$S^{ll}{}_{i-1}(x_i) = S^{ll}{}_i(x_i), \quad l = 2,...,n-1.$$

У 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 зазвичай задаються значення m_1 і m_n нахилів на кінцях відрізка $[a,b]$. Тоді система набуває вигляду

$$\begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & \ldots & & & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{bmatrix} \begin{bmatrix} m_2 \\ m_3 \\ \ldots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = \begin{bmatrix} \gamma_2 - m_1 \\ \gamma_3 \\ \ldots \\ \gamma_{n-2} \\ \gamma_{n-1} - m_n \end{bmatrix},$$

де $\gamma_i = \dfrac{3(y_{i+1} - y_{i-1})}{h}$.

Крайові умови можуть бути задані такими способами:

1) якщо відомі $y^l{}_i = f^l(x_1)$ і $y^l{}_{n=} f^l(x_n)$, то беруть $m_1 = y^l{}_i$, $m_n = y^l{}_n$;

2) значення похідних $f'(x)$ на кінцях відрізка апроксимуємо за формулами числового диференціювання 3-го порядку:

$$m_1 = \frac{1}{6h}(-11y_1 + 18y_2 - 9y_2 + 2y_4),$$

$$m_n = \frac{1}{6h}(-11y_n + 18yn - 1_2 - 9y_{n-2} + 2y_{n-3});$$

3) можуть бути відомі значення другої похідної на кінцях відрізка $[a,b]$: $y^{ll}{}_1 = f^{ll}(x_1), y^{ll}{}_n = f^{ll}(x_n)$. У цьому випадку із співвідношень $S^{ll}{}_1(x_1) = y^{ll}{}_1$ і $S^{ll}{}_{m-1}(x_n) = y^{ll}{}_n$ одержимо такі крайові умови:

4)

$$m_1 = -\frac{m_2}{2} + \frac{3}{2} \cdot \frac{y_2 - y_1}{h} - \frac{h}{4} y^{ll}{}_1,$$

$$m_n = -\frac{m_{n-1}}{2} + \frac{3}{2} \cdot \frac{y_n - y_{n-1}}{h} - \frac{h}{4} y^{ll}{}_n.$$

Одну з реалізацій алгоритму інтерполяції за допомогою кубічного сплайна (у вигляді окремого модуля) наведено в лістингу 5.9.

Лістинг 5.9. Реалізація алгоритму інтерполяції за допомогою кубічного сплайна мовою *Python*

```
## module cubicSpline
''' k = curvatures(xData,yData).
    Повертає значення нахилів сплайна в точках.
```

```
    y = evalSpline(xData,yData,k,x).
    Вираховує значення сплайна 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
```

Ця реалізація використовує *LU*-метод розв'язання СЛАР, який описано в розд. 6. У даній реалізації прийнято, що значення нахилів для крайових умов дорівнюють нулю. Функція **curvatures** визначає нахили сплайна в точках. На вхід цієї функції подаються табличні значення **xData, yData** . Функція **evalSpline** визначає сплайн у заданій точці x. На вхід цієї функції подаються: табличні значення **xData, yData**, а також визначені попередньою функцією коефіцієнти нахилу в точках k. Програму використання створеного модуля наведено у лістингу 5.10.

Лістинг 5.10. Приклад 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")
```

Розглянемо декілька прикладів знаходження значень функції методом інтерполяції за допомогою кубічного сплайна.

Приклад 5.7. Використати натуральний кубічний сплайн для визначення функції в точці $x = 1,5$. Функцію задано таблицею:

x	1	2	3	4	5
y	0	1	0	1	0

Ураховуючи, що друга похідна натурального сплайна 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, маємо систему лінійних рівнянь

$$\begin{cases} 0 + 4m_1 + m_2 = 6[0 - 2(1) + 0] = -12, \\ m_1 + 4m_2 + m_3 = 6[1 - 2(0) + 1] = 12, \\ m_2 + 4m_3 + 0 = 6[0 - 2(1) + 0] = -12. \end{cases}$$

Розв'язком 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. Відповідне значення інтерполяційного полінома можна отримати з (5.46), підставивши $i = 0$. Після підстановки $x_i - x_{i+1} = -h = -1$ дістаємо

$$S_0(x) = -\frac{m_0}{6}[(x - x_1)^3 - (x - x_1)] + \frac{m_1}{6}[(x - x_0)^3 - (x - x_0)] - \\ - [y_0(x - x_1) - y_1(x - x_0)].$$

Тому

$$y(1,5) = S_0(1,5) = 0 + \frac{1}{6}(-\frac{30}{7})[(1,5 - 1)^3 - (1,5 - 1)] - \\ - [0 - 1(1,5 - 1)] = 0,7679.$$

Приклад 5.8. Іноді потрібно змінити одну або обидві крайові умови для сплайна. Використати крайову умову у вигляді $S_0' = 0$ замість $S_0'' = 0$ та знайти значення функції в точці $x = 2,6$. Функцію задано 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 здиференціювавши, отримаємо

$$S_0'(x) = \frac{m_0}{6}\left[3\frac{(x-x_1)^2}{x_0-x_1} - (x_0-x_1)\right] - \frac{m_1}{6}\left[3\frac{(x-x_1)^2}{x-x_1} - (x_0-x_1)\right] +$$
$$+ \frac{y_0-y_1}{x_0-x_1}.$$

Ураховуючи те, що $S_0'(x_0) = 0$, маємо

$$\frac{m_0}{3}(x_0-x_1) + \frac{m_1}{6}(x_0-x_1) + \frac{y_0-y_1}{x_0-x_1} = 0,$$

або

$$2m_0 + m_1 = -6\frac{y_0-y_1}{(x_0-x_1)^2}.$$

Із заданої умови видно, що $y_0 = y_1 = 1$. Тоді останнє рівняння перетворюється на $2m_0 + m_1 = 0$ (a). Інші рівняння з (5.47) залишаються без змін. Ураховуючи, що $m_3 = 0$, маємо

$$m_0 + 4m_1 + m_2 = 6[1 - 2(1) + 0,5] = -3 \quad (b),$$
$$m_1 + 4m_2 = 6[1 - 2(0,5) + 0] = 0 \quad (c).$$

Знайдемо розв'язок СЛАР *(a), (b), (c)*:

$$m_0 = 0,4615; m_1 = -0,9231; m_2 = 0,2308.$$

Тепер можна одержати значення інтерполяційного полінома, застосувавши (5.46). Підставляючи $i = 2$ та $x_i - x_{i+1} = -h = -1$, отримаємо

$$S_2(x) = \frac{m_2}{6}\left[-(x-x_3)^3 - (x-x_3)\right] - \frac{m_3}{6}\left[-(x-x_2)^3 - (x-x_{2)}\right] -$$
$$- y_2(x-x_3) + y_3(x-x_2),$$

$$y(2,6) = S_2(2,6) = \frac{0,2308}{6}\left[-(-0,4)^3 + (-0,4)\right] - 0 - 0,5(-0,4) + \\ + 0 = 0,1871.$$

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 виконати цю умову важко, а інколи це 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, а інтерполяційний поліном може повторити ці похибки. Тому іноді краще, щоб графік полінома не збігався, а проходив поблизу табличних точок. Степінь такого полінома беруть невисоким $(1...4)$ і меншим за n. Нехай функцію $f(x)$ задано в m точках $x_1 \ldots x_m$:

$$f(x_1) = y_1, ..., f(x_m) = y_m.$$

У 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 із них.

Величина g є функцією $n+1$ аргументу a_i:

$$g(a_0, ..., a_n) = \sum (a_0 + a_1 x_1 + ... + a_n x_i^{\,n} - y_i)^2. \qquad (5.48)$$

Потрібно знайти точку її мінімуму, необхідною умовою чого є рівність нулю частинних 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: $p(x) = a_0$. Наприклад, масу деякого предмета вимірювали на m різних важелях і одержали значення $y_1, ..., y_m$. Що вважати масою предмета? Функція g стає функцією одного аргументу a_0: $g(a_0) = \sum (a_0 - y_i)^2$.

Із 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 (додавання проводять у межах від 1 до m):

$$g'(a_0) = \sum_{i=1}^{m} 2(a_0 - y_i) = 2(\sum a_0 - \sum y_i) = 2(m a_0 - \sum y_i) = 0.$$

Звідси випливає, що $g''(a_0) > 0$, тобто МНК дає середнє арифметичне. Зазначимо, що в будь-якій точці $g''(a_0) = 2m > 0$.

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:

$$\frac{\partial g}{\partial a_0}=\sum 2(a_0+a_1x_1-y_i)=2(\sum a_0+\sum a_1x_1-\sum y_i)=$$

$$=2\Big[ma_0+\Big(\sum x_1\Big)a_1-\sum y_i\Big]=0,$$

$$\frac{\partial g}{\partial a_1}=\sum 2(a_0+a_1x_1-y_i)x_i=2(\sum a_0x_i+\sum a_1{x_1}^2-\sum x_iy_i)=$$

$$=2\Big[(\sum x_i)a_0+\Big(\sum {x_1}^2\Big)a_1-\sum x_iy_i\Big]=0.$$

Одержали систему двох лінійних рівнянь із двома невідомими a_0 та a_1:

$$\begin{cases} ma_0 \quad\quad +(\sum x_i)a_i \;=\sum y_i, \\ (\sum x_i)a_0 \;+(\sum {x_i}^2)a_i=\sum x_iy_i \end{cases}$$

або в матричній формі:

$$\begin{pmatrix} m & \sum x_i \\ \sum x_i & \sum {x_i}^2 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}=\begin{pmatrix} \sum y_i \\ \sum x_iy_i \end{pmatrix}.$$

За формулою Крамера одержимо розв'язок:

$$a_0=\frac{\sum x_i^2\cdot\sum y_i-\sum x_iy_i\cdot\sum x_i}{m\sum x_i^2-(\sum x_i)^2};\;\; a_1=\frac{m\sum x_iy_i-\sum y_i\cdot\sum x_i}{m\sum x_i^2-(\sum x_i)^2}.$$

У разі довільного n аналогічно прирівняємо до нуля $\dfrac{\partial g}{\partial a_j}$ $\big(j=0,...,n\big)$ й одержимо СЛАР порядку $n+1$ відносно $\big(n+1\big)$-ї невідомої $a_0,...,a_n$:

$$\frac{\partial g}{\partial a_0} = 2\sum (a_0 + a_1 x_1 + ... + a_n x_i^n - y_i) = 0,$$

$$\frac{\partial g}{\partial a_1} = 2\sum (a_0 + a_1 x_1 + ... + a_n x_i^n - y_i) x_1 = 0,$$

...

$$\frac{\partial g}{\partial a_n} = 2\sum (a_0 + a_1 x_1 + ... + a_n x_i^n - y_i) x^n{}_1 = 0.$$

Ці рівняння називають *нормальними*. Запишемо цю систему лінійних алгебраїчних рівнянь, групуючи члени, що містять a_j :

$$\begin{cases} ma_0 + \left(\sum x_i\right) a_i + \left(\sum x^2{}_i\right) a_2 + ... \\ ... + \left(\sum x^n{}_i\right) a_n = \sum y_i, \\ \\ \left(\sum x_i\right) a_0 + \left(\sum x^2{}_i\right) a_i + \left(\sum x^3{}_i\right) a_2 + ... \\ ... + \left(\sum x^n{}_i\right) a_n = \sum x_i y_i, \\ ... \\ \left(\sum x^n{}_i\right) a_0 + \left(\sum x^{n+1}{}_i\right) a_i + \left(\sum x^{n+2}{}_i\right) a_2 + ... \\ ... + \left(\sum x^{2n}{}_i\right) a_n = \sum x^n{}_i y_i \end{cases} \quad (5.49)$$

або в матричній формі:

$$\begin{pmatrix} m & \sum x_i & \sum x_i^2 & ... & \sum x_i^n \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & ... & \sum x_i^{n+1} \\ & & & ... & \\ \sum x^n{}_i & \sum x_i^{n+1} & \sum x_i^{n+2} & ... & \sum x_i^{2n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_m \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ ... \\ \sum x_i^n y_i \end{pmatrix}. \quad (5.50)$$

Доведено, що ця система має єдиний розв'язок, отже, задача апроксимації за методом найменших квадратів також має єдиний розв'язок.

Розглянемо більш загальну постановку задачі середньоквадратичного наближення функцій.

Зазначимо, що апроксимуюча функція не обов'язково повинна бути алгебраїчним поліномом. Нехай φ_0, φ_1, …, φ_n – задані функції однієї змінної, а $w_1, \ldots, w_n$ – задані додатні числа (ваги або вагові коефіцієнти). Тоді загальна постановка задачі апроксимації за методом найменших квадратів полягає у знаходженні чисел $a_0, a_1, \ldots, a_n$, які мінімізують функцію:

$$g(a_0, \ldots, a_n) = \\ = \sum_{i=1}^{m} w_i \left[a_0 \phi_0(x_i) + a_1 \phi_1(x_i) + \ldots + a_n \phi_n(x_i) - y_i \right]^2. \quad (5.51)$$

Якщо $w_1 = \ldots = w_m = 1$, $a\phi_j(x) = x^j$, то одержимо постановку (5.48). Окрім степеневої іноді використовують такі базисні функції:

$$\phi_j(x) = \sin j\pi x, \; \phi_j(x) = e^{\alpha_j x},$$

де $j = 0, 1, \ldots, n$; α_j – задані числа.

Ваги w_i використовують, щоб надати більшу або меншу роль вузлам сітки. Наприклад, якщо значення $y_1, \ldots, y_{10}$ виміряні більш точно, то можна взяти $w_1 = \ldots = w_{10} = 5$, а інші ваги взяти меншими: $w_{11} = \ldots = w_m = 1$.

Для функції (5.51), як і раніше, можна побудувати нормальні рівняння. Частинні похідні цієї функції мають вигляд

$$\frac{\partial g}{\partial a_j} = 2\sum_{i=1}^{m} w_i \phi_i(x_i) \left[a_0 \phi_0(x_i) + a_1 \phi_1(x_i) + \ldots + a_n \phi_n(x_i) - y_i \right].$$

Покладемо їх такими, що дорівнюють нулю, і об'єднаємо коефіцієнти при a_i. Одержимо СЛАР:

$$\begin{pmatrix} \sum w_i\phi_0(x_i)\phi_0(x_i) & \sum w_i\phi_1(x_i)\phi_0(x_i) & \dots & \sum w_i\phi_n(x_i)\phi_0(x_i) \\ \sum w_i\phi_0(x_i)\phi_1(x_i) & \dots & & \\ & \dots & & \\ \sum w_i\phi_0(x_i)\phi_n(x_i) & \sum w_i\phi_1(x_i)\phi_n(x_i) & \dots & \sum w_i\phi_n(x_i)\phi_n(x_i) \end{pmatrix} \times$$

$$\times \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum w_i\phi_0(x_i)y_i \\ \sum w_i\phi_1(x_i)y_i \\ \dots \\ \sum w_i\phi_n(x_i)y_i \end{pmatrix}. \qquad (5.52)$$

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 зумовлені, 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, $q_1,\dots,q_n$ – поліноми степенів $0,1,\dots,n$ відповідно, їх називають *взаємно ортогональними* на множині точок $x_1,\dots,x_n$, якщо

$$\sum_{i=1}^{m} q_k(x_i)\cdot q_j(x_i) = 0 \quad \text{при } k, j = 0,1,\dots,n \quad \text{і} \quad k \neq j.$$

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

$$\sum_{i-1}^{m}\left[q_k(x_i)\right]^2 a_k = \sum_{i-1}^{m} q_k(x_i) y_i,\ k = 0,1,...,n.$$

Із такої системи легко знаходять невідомі a_k :

$$a_k = \frac{\sum_{i-1}^{m} q_k(x_i) y_i}{\sum_{i-1}^{m}\left[q_k(x_i)\right]^2}. \qquad (5.53)$$

Тобто одержуватимуть потрібний апроксимуючий поліном:

$$q(x) = \sum_{k=0}^{n} a_k q_k(x). \qquad (5.54)$$

Цей поліном є іншим поданням того самого полінома, який отримується 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 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 до вигляду

$$\sum_{i=1}^{m} x_i q_1(x_i) - m\beta_1 = 0,$$

$$\sum_{i=1}^{m} x_i \left[q_1(x_i)\right]^2 - \alpha_2 \sum_{i=1}^{m}\left[q_1(x_i)\right]^2 = 0.$$

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 інші поліноми. Нехай поліноми $q_0, q_1, ..., q_j$ вже знайдено. 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 всіх попередніх поліномів $q_k(x), k < j-1$. Справді,

$$\sum_{i=1}^{m} q_{j+1}(x_i)q_k(x_i) = \sum\left[x_i q_j(x_i) - \alpha_{j+1}q_j(x_i) - \beta_j q_{j-1}(x_i)\right]q_k(x_i) =$$

$$= \sum x_i q_j(x_i)q_k(x_i) - \alpha_{j+1}\sum q_j(x_i)q_k(x_i) - \beta_j q_{j-1}(x_i)q_k(x_i).$$

Оскільки поліноми $q_0,...,q_j$ ортогональні за припущенням, то два останні доданки рівні нулю. Поліном $xq_k(x)$ має степінь $(k+1)$ і, отже, може подаватися як лінійна комбінація поліномів $q_0,...,q_k$. Але $k+1<j-1$, тому перший доданок теж дорівнює нулю. Отже, $q_{j+1}(x)$ ортогональна до всіх поліномів $q_0,...,q_j$.

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} й одержимо

$$\alpha_{j+1} = \frac{\sum_{i=1}^{m} x_i\left[q_j(x_i)\right]^2}{\sum_{i=1}^{m}\left[q_j(x_i)\right]^2}, \qquad (5.56)$$

$$\beta_j = \frac{\sum_{i=1}^{m} x_i q_j(x_i)\cdot q_{j-1}(x_i)}{\sum_{i=1}^{m}\left[q_{j-1}(x_i)\right]^2}. \qquad (5.57)$$

Із викладеного випливає алгоритм середньоквадратичного наближення функцій за допомогою ортогональних поліномів.

1. Покласти $q_{-1}(x)=0,\ \ q_0(x)=1,\ \ j=0,\ a_0=\dfrac{\sum_{i=1}^{m} y_i}{m}$.

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 $a_{j+1} = \dfrac{\sum_{i=1}^{m} q_{j+1}(x_i) y_i}{\sum_{i=1}^{m} \left[q_{j+1}(x_i) \right]^2}$.

6. Покласти $j = j + 1$. Якщо $j \le n + 1$, то перейти до п. 2.

7. Побудувати поліном

$$P_n(x) = a_0 q_0(x) + \ldots + a_n q_n(x) = c_0 + c_1 x + \ldots + c_n x^n.$$

Переваги цього підходу полягають у тому, що тут 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 із першого степеня, потім побудувати поліном другого степеня і т. д. доти, поки не одержимо потрібний поліном із невеликим g.

Реалізацію алгоритму апроксимації за допомогою ортогональних поліномів (у вигляді модуля) наведено в лістингу 5.11.

Лістинг 5.11. Реалізація алгоритму апроксимації ортогональними поліномами

```
# -*- 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
```

Модуль складається з двох функцій: **polyFit** та **stdDev**. Перша призначена для пошуку коефіцієнтів ортогонального полінома та має на вході такі параметри: **xData, yData –** вхідна функція, яка задана таблично; **m** – степінь шуканого полінома. Друга функція визначає стандартне відхилення отриманого полінома від заданої функції. Для розв'язування СЛАР, яку потрібно знайти при пошуку полінома, у функції **polyFit** використано метод Гаусса з обертанням (**gaussPivot**). Методи розв'язування СЛАР розглядаються в розд. 6.

Приклад використання створеного модуля наведено в лістингу 5.12.

Лістинг 5.12. Приклад використання створеного модуля

```
#!/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")
```

Контрольні запитання

1. У чому суть задачі апроксимації?
2. Що таке інтерполяція?
3. Що таке інтерполяційний поліном?
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. Загальні положення

Виконання багатьох прикладних, економічних та інших інженерно-технічних завдань безпосередньо пов'язано з обчислювальними методами лінійної алгебри. Важливість освоєння цих методів визначається й тим, що саме до них зводиться багато завдань числового аналізу. У цьому розділі розглядаються питання про числове розв'язування систем лінійних рівнянь алгебри, про обернення матриць і знаходження власних значень та власних векторів матриць.

Числові методи розв'язування систем лінійних рівнянь і обернення матриць ділять на три групи: 1) *точні* (прямі) методи; 2) *ітераційні* (послідовних наближень); 3) *комбіновані.* Точні методи характеризуються можливістю отримання дійсних значень невідомих величин. При цьому, звичайно, припускається, що всі обчислення здійснюються над дійсними значеннями величин. Ітераційні методи (послідовних наближень) характеризуються тим, що із самого початку задаються наближеними значеннями невідомих, за якими отримують нові наближені значення і це продовжується доти, поки не отримують наближені значення невідомих із заданою точністю. Комбіновані методи є поєднанням 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 подати у вигляді одного матричного рівняння

$$Ax = b\,, \qquad (6.2)$$

де

$$A = \begin{Bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{Bmatrix}$$ – матриця системи;

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix}$$ – матриця-стовпець вільних членів;

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix}$$ – матриця-стовпець невідомих.

За умови, що $m = n$ і визначник матриці A не дорівнює нулю, розв'язок системи (6.2) єдиний і має вигляд

$$x = A^{-1}b\,, \qquad (6.3)$$

де A^{-1} – матриця, обернена до квадратної матриці A.

За визначенням матриця A^{-1} є оберненою 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 самого порядку, що і матриця A.

Як відомо з курсу алгебри, розв'язок системи (6.2) визначається за формулами Крамера

$$x_i = \frac{\det A_i}{\det A}, \tag{6.5}$$

де $\det A_i$ – визначник матриці A_i, отриманої шляхом заміни *i*-го стовпця матриці A стовпцем вільних членів.

Формула Крамера має досить простий вигляд, проте при її використанні необхідно проводити великий обсяг обчислень (порядку $n!\left(n^2+1\right)+n$), що робить її практично непридатною при розв'язуванні 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 розв'язок. Система називається *визначеною* у випадку єдиного розв'язку.

6.2. Прямі методи

6.2.1. Метод Гаусса

Найбільш давнім і відомим прямим методом розв'язування систем лінійних рівнянь є метод виключення невідомих, який пов'язують з ім'ям Гаусса. Застосування цього методу основане на такій теоремі.

Теорема 6.1. Якщо A – квадратна матриця порядку *n*, то існують нижня матриця L і верхня матриця U того самого порядку такі, що $PA = LU$, де P – деяка матриця перестановок порядку n.

Якщо матриця A має відмінні від нуля головні (діагональні) мінори

$$\Delta_1 = a_{11} \neq 0;\ \Delta_2 = \neq 0,\ \ldots,\ \Delta_n = \det A \neq 0,$$

то матриця P може вибиратися одиничною, тобто $A = LU$. Причому це розкладання буде єдиним, якщо наперед зафіксувати діагональні елементи однієї з трикутних матриць.

Ідея алгоритму Гаусса при розв'язуванні системи (6.2) полягає в тому, що дана система зводиться до еквівалентної системи з верхньою трикутною матрицею (прямий хід). Із 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

$$Ux = y, \tag{6.6}$$

то розв'язок одержується з

$$Ly = b. \tag{6.7}$$

Перетворення (6.6) називається *прямим ходом* методу Гаусса, а перетворення (6.7) – *зворотним.*

На підставі цієї теореми матрицю A, головний діагональний мінор якої відмінний від нуля, можна подати у вигляді добутку двох трикутних матриць:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}. \tag{6.8}$$

Прямий хід розв'язування можна здійснювати таким чином. Вважаючи, що коефіцієнт $a_{11} \neq 0$ (це не обмежує загальних міркувань, оскільки 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 $(i-1)$ 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 у вигляді

$$u_{11} = 1;\ u_{1i} = a_{1i} / a_{11};\ l_{1i} = a_{1i};\ i = 1,...,n;$$

$$y_1 = b_1 / a_{11};\ l_{ki} = a_{ki} - \sum_{j=1}^{i-1} l_{kj} u_{ji};\ k = i,...,n;\ i = 2,\ldots,n;$$

$$u_{ik} = (a_{ik} - \sum_{j=1}^{i-1} l_{ij} u_{jk}) / l_{ii};\ k = i+1,...,n;$$

$$y_i = (b_i - \sum_{j=1}^{i-1} l_{ij} y_j) / l_{ii};$$

$$x_n = y_n;\ x_{n-i} = y_{n-i} - \sum_{j=n-i+1}^{n} u_{n-1,j} x_j;\ i = 2,\ldots,n-1.$$

Розглянемо як приклад просту систему

$$\begin{cases} 0,000100\, x_1 + x_2 = 1, \\ x_1 + x_2 = 2, \end{cases}$$

яка має точний розв'язок $x_1 = 1,00010,\ x_2 = 0,99990$.

Якщо задача розв'язується, наприклад, із використанням трьох десяткових цифр, то це означає, що залишаються тільки три старші значущі десяткові цифри будь-якого результату арифметичних операцій. Припустимо, що результат округлюється. Згідно з виключенням Гаусса помножимо перше рівняння на –10 000 і додамо до другого. Маємо

$$\begin{cases} 0,000100\, x_1 + x_2 = 1; \\ -10000\, x_2 = -10000 \end{cases}$$

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. Справді, спробуємо переставити рівняння (виконавши вибір провідного елемента). Бачимо, що виключення Гаусса дає розв'язок (при тих самих припущеннях про точність обчислень) $x_1 = 1.00,\ x_2 = 1.00$. Причому цей розв'язок обчислений із тією точністю, яку можна отримати для трьох десяткових знаків.

На підставі цього прикладу доходимо висновку – недостатньо уникати тільки нульових провідних елементів, необхідно також уникати вибору малих провідних елементів при зведенні матриць до трикутного вигляду.

Таким чином, приходимо до модифікованого методу *Гаусса – Жордано*, який може використовувати дві стандартні стратегії вибору провідних елементів. Перша полягає в *частковому виборі* провідного елемента: на *k*-му кроці прямого ходу як провідний береться найбільший (за абсолютною величиною) елемент у незведеній частині *k*-го стовпця, тобто

$$\left|a_{kk}\right| = \max\left|a_{lk}\right|,\ i = k,\ k+1,\ ...,n\,.$$

Друга стратегія полягає в *повному виборі* провідного елемента: на *k*-му кроці прямого ходу за провідний береться найбільший (за абсолютною величиною) елемент у незведеній частині матриці, тобто

$$\left|a_{kk}\right| = \max\left|a_{ij}\right|,\ i = k,\ k+1,\ \ldots,\ n;\ j\ =\ k,\ k+1,\ \ldots,\ n\,.$$

Хоча стратегія повного вибору надійніша, частіше застосовується все-таки перша стратегія, особливо для розв'язування великих систем, оскільки обчислювальні витрати при цьому значно менші. Детальніше цей метод буде розглянуто пізніше.

Реалізацію методу Гаусса у вигляді модуля мовою *Python* наведено в лістингу 6.1.

Лістинг 6.1. Модуль із реалізацію алгоритму Гаусса

```
# -*- coding: cp1251 -*-
## module gaussElimin
''' x = gaussElimin(a,b).
 Вирішує [a]{b} = {x} методом Гаусса.
'''
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]
 # Зворотний 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
```

На вхід функції gaussElimin подаються матриця A та вектор-стовпець B.

6.2.2. Число зумовленості методу Гаусса

Число зумовленості також грає фундаментальну роль в аналізі помилок округлень, здійснюваних у процесі гауссового виключення. Припустимо, що всі елементи A і b точно подаються числами з рухомою точкою, і нехай $x*$ – вектор розв'язку, який складено з чисел із рухомою точкою. Припустимо також, що точна виродженість матриці, якщо це має місце, не була виявлена і що не було ні машинних нулів, ні переповнювань. Тоді виконуються нерівності

$$\frac{\left\|b - Ax^*\right\|}{\|A\| \cdot \left\|x^*\right\|} \le \rho \cdot 2^{-t} \text{ і } \frac{\left\|x - x^*\right\|}{\left\|x^*\right\|} \le \rho \cdot \mathrm{cond}(A) 2^{-t}.$$

Тут 2 – основа системи для подання чисел із рухомою точкою; t – кількість розрядів дробової частини зі значенням молодшого розряду 2^{-t} має значення "машинного епсилон". Величина ρ нижче визначається точніше, але звичайне її значення не перевищує 2.

Перша нерівність свідчить про таке: як правило, можна розраховувати на те, що відносний відхил матиме величину, порівнянну з похибкою округлення, незалежно від того, на скільки погано зумовлена матриця.

Друга нерівність вимагає, щоб A була невироджена, бо в неї входить точний розв'язок x. Ця нерівність випливає безпосередньо з першої і визначення $\mathrm{cond}(A)$; її сенс – відносна похибка буде мала, якщо мале число $\mathrm{cond}(A)$, 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-таки зберігає силу, тоді як друга не має сенсу.

Щоб точніше визначити величину ρ, необхідно використовувати поняття матричної норми і встановити деякі допоміжні нерівності:

$$\|A\| = \max_x \frac{\|Ax\|}{\|x\|} = \max_j \|a_j\|,$$

де a_j – стовпці матриці A.

Основний результат у дослідженні помилок округлень в гауссовому виключенні полягає в тому, що обчислений розв'язок x^* точно задовольняє систему

$$(A + E)x^* = b,$$

де E – матриця, елементи якої мають величину порядку помилок округлень в елементах матриці A. Звідси можна негайно вивести нерівності, що визначають відхил і похибку в обчисленому розв'язку. Відхил знаходиться як $b - Ax^* = Ex^*$. Отже,

$$\| b - Ax^* \| = \| Ex^* \| \le \|E\| \cdot \|x^*\|.$$

У визначенні відхилу бере участь добуток $Ax*$ так, що доречно розглядати відносний відхил, який порівнює норму вектора $b - Ax^*$ із нормами A та $x*$. Оскільки $\frac{\|E\|}{\|A\|} = \rho \cdot 2^{-t}$, то з наведеної вище нерівності відразу випливає, 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 $x - x^* = A^{-1}\left(b - Ax^*\right)$ і тоді $\| x - x^* \| = \| Ex^* \| \le \| A^{-1} \| \cdot \| E \| \cdot \| x^* \|$.

Найпростіше порівнювати норму похибки із нормою обчисленого розв'язку. Таким чином, відносна похибка задовольняє нерівність $x^{(k)}$ $\frac{\|x - x^*\|}{\|x^*\|} \le \rho \cdot \|A\| \cdot \|A^{-1}\| 2^{-t}$.

Виявляється, що $\|A^{-1}\| = 1/m$, тоді $\mathrm{cond}(A) = \|A\| \, \|A^{-1}\|$ і, таким чином

$$\frac{\|x - x^*\|}{\|x^*\|} \le \rho \cdot \mathrm{cond}(A) \cdot 2^{-t}.$$

6.2.3. Метод Гаусса з вибором головного елемента

У викладеному вище методі Гаусса ми припускали, що діагональні елементи a_{kk}, на які проводиться 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 має передбачатися.

Нехай ми виконали $(k-1)$ крок алгоритму Гаусса та звели систему до вигляду, в якому піддіагональні елементи перших $(k-1)$-го стовпців дорівнюють нулю (рис. 6.1, б), та нехай $a_{kk}=0$.

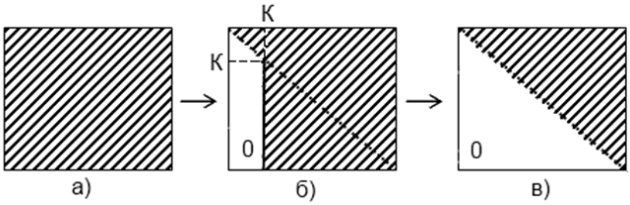


Рис. 6.1. Зміна ненульової структури матриці за процедури Гаусса: *а* – вихідна матриця; *б* – нульові стовпці; *в* – остаточна матриця

Тоді серед нижче розташованих елементів *k*-го стовпця є хоча б один, що не дорівнює нулю, інакше матриця A була б виродженa. Нехай він знаходиться в рядку з номером h. Переставивши місцями k-те та h-те рівняння системи, що не змінює її розв'язку, помістимо на діагональ ненульовий елемент і зможемо продовжити процедуру Гаусса. Зазначимо, що в *k*-му та *h*-му рядках матриці A ліворуч від *k*-го стовпця розташовані нулі, тому перестановка цих рядків не змінює структуру матриці, зображеної на рис. 6.1, *б*. Саме з цієї причини пошук ненульового елемента проводимо тільки нижче від 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, а в результаті матриця системи не набула б верхньої трикутної форми.

Виявляється, що перестановка рівнянь бажана, якщо навіть діагональний елемент не дорівнює нулю. Метод Гаусса 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 менша, чим мен-

ший множник $R = a_{ik} / a_{kk}$. Щоб зробити множник якомога меншим, треба щоб *akk* було якнайбільшим. 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. Інакше кажучи, при перестановці рівнянь необхідно досягти того, щоб $|a_{kk}| \geq |a_{ik}|$. Тоді множник $|R| \leq 1$.

Описаний спосіб розв'язування СЛАР називають *методом Гаусса з вибором головного елемента* за стовпцем, або з частковим упорядкуванням.

Алгоритм цього методу складається з таких кроків:

1) установлюємо $k = 1$;

2) серед піддіагональних елементів *k*-го стовпця та a_{kk} знайти максимальний за модулем елемент $a_{hk} = \max |\grave{a}_{ik}| \quad (i \geq k)$. Якщо $k \neq h$, 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) установити $k = k+1$. Якщо $k < n-1$, перейти до п. 2. Якщо $k = n$, 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.

Як же впливає вибір головного елемента на точність результатів? Проілюструємо це на прикладі системи двох рівнянь із двома невідомими [2]:

$$\begin{pmatrix} -10^{-5} & 1 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{6.9}$$

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 із чотирма десятковими розрядами, тобто на ЕОМ, числа якої зберігаються у формі $0.nnnn\cdot 10^{P}$, де n – цифри мантиси, P – порядок.

Процедуру Гаусса 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 від другого рівняння перше, помножене на R, перетворимо на нуль елемент $a21$, а нове значення $a22$ 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 обчисленні нового значення b_2 похибок не виникає:

$$b_2=0-(-0,2\cdot 10^{6})(0,1\cdot 10^{1})=0,2\cdot 10^{6}.$$

У результаті система (6.9) зводиться до вигляду

$$\begin{pmatrix}-10^{-5} & 1\\ 0 & 0,2\cdot 10^{6}\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}1\\ 0,2\cdot 10^{6}\end{pmatrix}.$$

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 знаку призвести до катастрофічно неправильного розв'язку? Скористаємося принципом зворотного аналізу похибок, який полягає не у з'ясуванні того, яка допущена похибка, а в пошуку задачі, яка в дійсності розв'язувалась.

Відкинута при обчислюванні $a_{22}^{(1)}$ величина 0,000·00110 є значенням a_{22} початкової 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 сподіватись, що застосувавши процедуру вибору головного елемента за стовпцем, ми одержимо більш точний розв'язок.

Поміняємо в системі (6.9) рівняння місцями. У 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 Гаусса, зазначивши, 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 Гаусса зі 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 Гаусса з вибором головного елемента за стовпцем. У першому стовпці максимальний елемент міститься 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 із цієї ситуації є повне упорядкування, тобто вибір головного елемента по всьому полю. Для реалізації запропонованого алгоритму спочатку створимо допоміжний модуль, який матиме дві функції: **swapRows** – для переміщення рядків матриці та **swapCols** – для переміщення стовпців. Реалізацію цього допоміжного модуля наведено в лістингу 6.2.

Лістинг 6.2. Реалізація допоміжного модуля

```
# -*- coding: cp1251 -*-
## module swap
''' swapRows(v,i,j).
 Міняє місцями рядки і та j вектора або матриці [v].

 swapCols(v,i,j).
 Міняє місцями стовпці 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
```

У лістингу 6.3 наведено реалізацію запропонованого алгоритму у вигляді модуля мовою *Python* із використанням функцій допоміжного модуля.

Лістинг 6.3. Реалізація модуля алгоритму

```
# -*- coding: cp1251 -*-
## module gaussPivot
''' x = gaussPivot(a,b,tol=1.0e-9).
 Вирішує [a]{x} = {b} методом Гаусса
```

```
 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')

 # Зворотний 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
```

Створений модуль складається з однієї функції, яка і реалізує алгоритм. На вхід функції **gaussPivot** подаються матриця A та вектор-стовпець B.

6.2.4. Метод Гаусса з вибором головного елемента по всьому полю

У п. 6.2.3 ми показали, що часткове упорядкування не забезпечило прийнятних результатів для системи

$$\begin{pmatrix} 10 & -10^6 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -10^6 \\ 0 \end{pmatrix}, \tag{6.10}$$

еквівалентної системі (6.9) із розв'язком $(-0,5;1)$.

Перевіримо, що дає нам повне упорядкування. Для цього шукаємо головний елемент по всьому полю матриці. Очевидно, максимальним за модулем є елемент $a_{12} = -10^6$. Розмістимо його на діагоналі. Для цього переставимо місцями стовпці матриці. Оскільки кожен стовпець відповідає конкретному *xj*, то треба переставити й компоненти вектора *X*. Одержимо систему

$$\begin{pmatrix} 10 & -10^6 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} -10^6 \\ 0 \end{pmatrix}. \tag{6.11}$$

Розв'яжемо її методом Гаусса:

$$R = \frac{1}{-10^6} = -10^{-6};\ \ a_{22}^{(1)} = 2 - 10 \cdot (-10^{-6}) = 2 + 10^{-5} \approx 2;$$

$$b_2^{(1)} = 0 - (-10^{-6}) = -1. \tag{6.12}$$

$$x_1 = \frac{b_2^{(1)}}{a_{22}^{(1)}} = \frac{-1}{2} = -0,5;$$

$$x_2 = \frac{1}{a_{11}}(b_1 - a_{12} \cdot x_2) = \frac{1}{-10^6}(-10^6 - 10 \cdot (-0,5)) = \frac{-10^6 + 5}{-10^6} \approx 1.$$

Одержано розв'язок, майже такий, що збігається з точним. Проте стійкість цього алгоритму до похибок округлення призводить до ускладнення програми та збільшення часу обчислення. Такий алгоритм відрізняється від попереднього тим, що на *k*-му кроці (перед перетворенням на нуль елементів *k*-го стовпця) головний елемент відшукується на правій нижній субматриці матриці *A*, яку на рис. 6.2 показано подвійним штрихуванням.

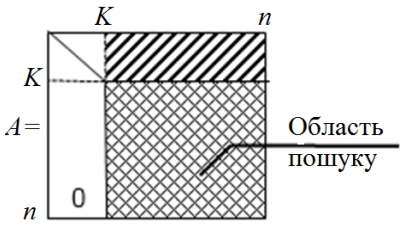


Рис. 6.2. Область пошуку головного елемента
в алгоритмі повного упорядкування

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 Гаусса з вибором головного елемента по всьому полю наведено нижче.

1. Установити $k = 1$; сформувати масив перестановок $MP(i) = i, \quad i = 1,...,n$.

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. Установити $k = k + 1$. Якщо $k \leq n - 1$, перейти до п. 2.

7. Виконати зворотний хід й одержати вектор невідомих Z.

8. За допомогою масиву перестановок MP сформувати за вектором Z масив X.

9. Обчислити вектор відхилів r та надрукувати вектори r і X.

Навіщо потрібен масив MP? Розглянемо це на прикладі розв'язування СЛАР із чотирма невідомими, точний розв'язок якої

$$x_1 = 7,\ x_2 = 5,\ x_3 = -9,\ x_4 = -2.$$

Нехай у процесі виконання процедури Гаусса 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:

i	1	2	3	4
Z=	7	–2	–9	5

в якому порівняно з вектором невідомих X переставлено 2-й і 4-й елементи. Тобто, одержимо неправильну відповідь. Саме для запам'ятовування проведених перестановок стовпців матриці і потрібен масив MP. У цьому випадку нашої системи з чотирьох рівнянь він спочатку має вигляд

i	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:

i	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 містяться в масиві MP, перед друкуванням кінцевого результату потрібна зворотна перестановка 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:

$$X\left(MP(1)\right) = X(1) \leftarrow Z(1) = 7,$$
$$X\left(MP)2)\right) = X(4) \leftarrow Z(2) = -2,$$
$$X\left(MP(3)\right) = X(3) \leftarrow Z(3) = -9,$$
$$X\left(MP(4)\right) = X(2) \leftarrow Z(4) = 5,$$

та, оскільки елементи масиву X друкуються в порядку зростання номерів елементів, отримаємо правильну відповідь.

Таким чином, у процесі зворотного ходу спочатку визначають масив Z, потім операторами (6.13) знаходять масив X.

Зазначимо, що час роботи програми можна суттєво скоротити, якщо не робити насправді перестановку рядків та стовпців матриці, а запам'ятовувати ці перестановки за допомогою масиву MP і ще одного аналогічного масиву для рядків. Хоча такий підхід ще більше ускладнюватиме програму.

6.2.5. *LU*-алгоритм

Розглянуті вище реалізації методу Гаусса (виключення за стовпцями та рядками) не вичерпують усіх можливих варіантів. Існує 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 Гаусса.

Після класичного методу Гаусса 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 $(l_{ii} = 1)$, і верхньої трикутної U, тобто

$$A = LU. \tag{6.14}$$

Тоді початкову систему $Ax = B$ записують у вигляді

$$LUx = B. \tag{6.15}$$

Якщо позначити вектор Ux через $y\left(Ux = y\right)$, остання система набере вигляду

$$Ly = B\,. \tag{6.16}$$

Оскільки її матриця є нижньою трикутною, то розв'язують таку СЛАР просто (лістинг 6.4).

Лістинг 6.4. Фрагмент програми розв'язування СЛАР

```
…
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]
…
```

Коли ми вже знайшли внаслідок розв'язування цієї системи вектор *y*, тепер можемо розв'язати систему

$$Ux = y \tag{6.17}$$

із верхньою трикутною матрицею U (лістинг 6.5) і отримати шуканий вектор X.

Лістинг 6.5. Фрагмент програми розв'язування СЛАР

```
…
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]
…
```

Таким чином, *LU*-алгоритм має вигляд.

1. Розкласти матрицю A на добуток верхньої та нижньої матриць *L* та ***U*** (*LU*-факторизація).

2. Розв'язати систему (6.16) із нижньою трикутною матрицею (прямий хід).

3. Розв'язати систему (6.17) із 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 разі $A = LU$ розкладається тільки один раз і для кожної системи потрібно виконати тільки прямий і зворотний ходи. Кількість необхідних арифметичних операцій для всього *LU*-алгоритму є такою самою, як і для методу Гаусса, 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 Гаусса, а коефіцієнти R цього методу утворюють матрицю L. І тому для розкладу матриці A достатньо запам'ятати ці коефіцієнти й, оскільки елементи нижньої лівої частини матриці A стають нульовими і не використовуються, саме сюди можна вписувати ці коефіцієнти.

Таким чином, для проведення *LU*-факторизації не обов'язково використовувати додаткові масиви, а матриці L та U зберігати на полі матриці A, причому діагональні одиниці матриці L узагалі не зберігаються.

Наведемо, як приклад, *LU*-факторизацію такої матриці:

$$A = \begin{pmatrix} 2 & 7 & 5 \\ 4 & 4 & 3 \\ 6 & 8 & 9 \end{pmatrix}.$$

Використовуватимемо процедуру Гаусса з реалізацією за стовпцями й без вибору головного елемента.

Спочатку розглянемо піддіагональні елементи першого стовпця. Для елемента a_{21} знаходимо множник R:

$$R = \frac{a_{21}}{a_{11}} = \frac{4}{2} = 2 = l_{21}.$$

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 від другого рядка, як і в звичайному алгоритмі Гаусса:

$$a_{22}^{(1)} = a_{22} - a_{12} \cdot R = 4 - 7 \cdot 2 = -10,$$
$$a_{23}^{(1)} = a_{23} - a_{13} \cdot R = 3 - 5 \cdot 2 = -7.$$

Ці обидва числа, як і завжди, вносять на місце елементів a_{22}, a_{23}. Тепер обчислюють множник для елемента a_{31}:

$$R = \frac{a_{31}}{a_{11}} = \frac{6}{2} = 3 = l_{31}$$

і вносять його замість елемента a_{31}.

Перший рядок, починаючи з другого елемента, множимо на цей множник і віднімаємо від третього рядка, замість якого і вносяться

$$a_{32}^{(1)} = a_{32} - a_{12} \cdot R = 8 - 7 \cdot 3 = -13,$$
$$a_{33}^{(1)} = a_{33} - a_{12} 3 \cdot R = 9 - 5 \cdot 3 = -6.$$

У загальному випадку всі ці перетворення виконуються для другого та всіх наступних стовпців матриці, крім останнього. Для нашої матриці, яка тепер набула 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 від третього рядка:

$$a_{33}^{/(2)} = a_{33}^{/(1)} - a_{23}^{/(1)} \cdot R = -6 - (-7) \cdot \frac{3}{10} = \frac{31}{10}.$$

У результаті на полі початкового 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, чи наведена рівність виконується. Для цього помножимо дві отримані матриці:

$$a_{11} = l_{11}U_{11} + l_{12}U_{21} + l_{13}U_{31} = 1 \cdot 2 + 0 \cdot 0 + 0 \cdot 0 = 2,$$

$$a_{12} = l_{11}U_{12} + l_{12}U_{22} + l_{13}U_{32} = 1 \cdot 7 + 0 \cdot (-10) + 0 \cdot 0 = 7,$$

$$a_{13} = l_{11}U_{13} + l_{12}U_{23} + l_{13}U_{33} = 1 \cdot 5 + 0 \cdot (-7) + 0 = 5,$$

$$a_{21} = I_{21}U_{11} + l_{22}U_{21} + l_{23}U_{31} = 2 \cdot 2 + 1 \cdot 0 + 0 \cdot 0 = 4,$$

$$a_{22} = l_{21}U_{12} + I_{22}U_{22} + l_{23}U_{32} = 2 \cdot 7 + 1 \cdot (-10) + 0 \cdot 0 = 4,$$

$$a_{23} = I_{21}U_{13} + l_{22}U_{23} + l_{23}U_{33} = 2 \cdot 5 + 1 \cdot (-7) + 0 = 3,$$

$$a_{31} = l_{31}U_{11} + l_{32}U_{21} + l_{33}U_{31} = 32 + 0 + 1 \cdot 0 = 6,$$

$$a_{32} = l_{32}U_{12} + I_{32}U_{22} + I_{33}U_{32} = 3 \cdot 7 + \frac{13}{10} \cdot (-10) + 1 \cdot 0 = 8,$$

$$a_{33} = \mathrm{l}_{31}\mathrm{U}_{13} + \mathrm{l}_{32}\mathrm{U}_{23} + \mathrm{l}_{33}\mathrm{U}_{33} = 3 \cdot 5 + \frac{13}{10} \cdot (-7) + 1 \cdot \frac{31}{10} = 9.$$

Справді, ми отримали елементи початкової матриці. Алгоритм LU-факторизації можна сформулювати так:

1) установити $k = 1$;

2) установити $i = k + 1$;

3) знайти $R = a_{ik} / a_{kk}$ та занести його замість елемента a_{ik};

4) установити $j = k + 1$;

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 Гаусса, тут необхідно вибрати головний елемент у тій або іншій формі. Реалізацію алгоритму у вигляді модуля мовою *Python* наведено в лістингу 6.6.

Лістинг 6.6. Фрагмент програми розв'язування СЛАР

```
# -*- coding: cp1251 -*-
## module LUdecomp
''' a = LUdecomp(a).
 LU факторизація: [L][U] = [a]. Матриця, яка повертається [a] = [L\U]
 містить [U] у верхньому трикутнику і недіагональні елементи
 з [L] у 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
```

Модуль складається з двох функцій: **LUdecomp** – функції, яка виконує *LU*-факторизацію, та **LUsolve** – функції, що розв'язує СЛАР. На вхід першої функції подається тільки один параметр – матриця *A*. На вхід другої функції подаються матриця *A* та вектор-стовпець *B*.

6.2.6. Метод Жордано

Пропонована модифікація методу Гаусса зводить матрицю системи не до трикутної, а до діагональної форми (рис. 6.3).

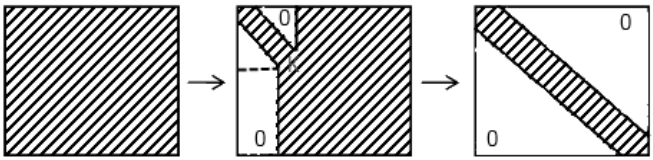


Рис. 6.3. Зміна ненульової структури матриці методом Жордано

При цьому перед виконанням *k*-го кроку (обробкою *k*-го стовпця матриці) перші $(k-1)$ стовпці матриці є нульовими, окрім діагональних елементів. У процесі *k*-го кроку перетворюються на нуль не тільки піддіагональні елементи *k*-го стовпця, але і його елементи, розташовані над діагоналлю. Виконують все це тими самими способами, що й у методі Гаусса – відніманням *k*-го рядка від інших рядків. Обчислювальна схема методу Жордано для розв'язування СЛАР така:

1) установити $k = 1$;
2) установити $i = 1$;
3) якщо $i = k$, перехід до п. 8;
4) знайти $R = a_{ik} / a_{kk}$;

5) установити $j = k$;

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 $x_з = b_j / a_{ij}$, $з = 1, 2, \ldots, n$.

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 Гаусса і становить близько n^3. Через це метод Жордано використовують, в основному, лише для обернення матриць та у задачах лінійного програмування. У лістингу 6.7 наведено реалізацію методу Жордано як функції, яка дозволяє знайти обернену матрицю.

Лістинг 6.7. Реалізація методу Жордано

```
# -*- 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
```

Модуль складається з однієї функції **gaussJordan**, яка виконує пошук оберненої матриці методом Жордано. Функції передається тільки один параметр – матриця *A*.

6.2.7. Метод оптимального виключення

Як і метод Жордано, цей метод зводить матрицю системи до діагональної форми, але послідовність виключення невідомих тут інша. Його особливістю є те, що в оперативній пам'яті можна зберігати лише частину матриці, тобто вводити її поступово із зовнішньої пам'яті. Метод використовувався на ранніх етапах розвитку обчислювальної техніки, а зараз втратив свою актуальність. Обробку матриці цим методом показано на рис. 6.4.

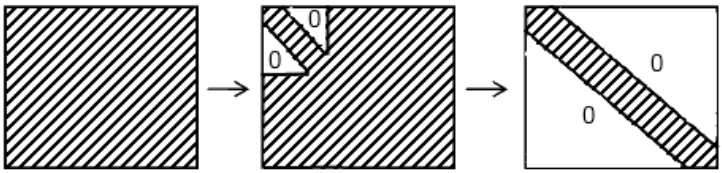


Рис. 6.4. Зміна ненульової структури матриці методом оптимального виключення

6.2.8. Метод прогонки

На практиці часто виникає необхідність розв'язування СЛАР із тридіагональною матрицею. Запишемо таку систему у вигляді

$$\begin{aligned}
b_1x_1 + c_1x_2 &= d_1,\\
a_2x_1 + b_2x_2 + c_2x_3 &= d_2,\\
a_3x_2 + b_3x_3 + c_3x_4 &= d_3,\\
\dots & \qquad a_k = f'(x_k). \quad (6.18)\\
a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n &= d_{n-1},\\
a_nx_{n-1} + b_nx_n &= d_n;
\end{aligned}$$

До розв'язування таких систем може приводити, наприклад, метод скінченних різниць при розв'язуванні крайових задач для диференціальних рівнянь.

Систему вигляду (6.18) можна розв'язати звичайним методом Гаусса, враховуючи структуру її матриці. У процесі прямого ходу для $k = 1,...,(n-1)$ виконуються такі дії:

$$\phi_k(a_k, x) = f(x_k),\ R = \frac{a_{k+1}}{b_k},\ b_{k+1}^{(1)} = b_{k+1} - c_k R,\ d_{k+1}^{(1)} = d_{k+1} - d_k R,$$

що вимагає $3(n-1)$ арифметичних операцій. При зворотному ході обчислюють вектор невідомих x за формулами:

$$x = \frac{d_n^{(1)}}{b_n^{(1)}},\ \ x_i = \left(d_i^{(1)} - c_i x_{i-1}\right) \cdot \frac{1}{b_i^{(1)}},\ \ \ i = n-1,...,1,$$

що вимагає $3(n-1)+1$ арифметичні операції. Усього для розв'язування методом Гаусса такої СЛАР необхідно $(8n-7)$ арифметичних операцій на відміну від $2/3n^3$ операцій, потрібних для розв'язування довільної системи цим методом.

У практиці обчислень часто використовують для розв'язування СЛАР вигляду (6.18) не метод Гаусса, а його модифікацію, яку називають методом прогонки [1]. За кількістю необхідних операцій цей метод еквівалентний методу Гаусса. Метод прогонки виконують у два етапи, які аналогічні прямому та зворотному ходам методу Гаусса.

На першому етапі (пряма прогонка) елементи $x_i (i = 1,\ldots,n-1)$ вектора невідомих виражають через x_{i+1} за допомогою прогонних коефіцієнтів A_i, B_i:

$$x_i \ = A_i x_{i+1} + B_i,\ i \ = \ 1,2,...,n-1\ . \qquad (6.19)$$

Мета першого етапу – обчислення цих коефіцієнтів. Із першого рівняння системи (6.19) знайдемо x_1:

$$x_1 = -\frac{c_1}{b_1} \cdot x_2 + \frac{d_1}{b_1}\ R_1,$$

тобто

$$A_1 = -\frac{c_1}{b_1},\ B_1 = \frac{d_1}{b_1}.$$

Підставимо ці співвідношення в друге рівняння й виразимо x_2 через 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:

$$A_i = -\frac{c_i}{e_i},\quad B_i = -\frac{d_i - a_iB_{i-1}}{e_i},\quad e_i = a_iA_{i-1} + b_i,\quad i = 2,3,...,n-1.$$

Останнє з цих співвідношень одержують із передостаннього рівняння. Справді,

$$a_{n-1}(A_{n-2}x_{n-1} - B_{n-2}) + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1},$$
$$x_{n-1}(a_{n-1}A_{n-2} + b_{n-1}) + c_{n-1}x_n + a_{n-1}B_{n-2} - d_{n-1},$$
$$x_{n-1} = -\frac{c_{n-1}}{a_{n-1}A_{n-2} + b_{n-1}}x_n + \frac{d_{n-1} - a_{n-1}B_{n-2}}{a_{n-1}A_{n-2} + b_{n-1}} = A_{n-1}x_n + B_{n-1}.$$

Продовжуючи цей процес, підставимо $x\ B_k$ в останнє рівняння:

$$a_n(A_{n-1}x_n + B_{n-1}) + b_nx_n = d_n,\ \ x_n(a_nA_{n-1} + b_n) = d_n - a_nB_{n-1}.$$

Одержаний вираз дає можливість знайти x_n. При цьому починається другий етап – зворотна прогонка:

$$x_n = \frac{d_n - a_nB_{n-1}}{a_nA_{n-1} + b_n}.$$

Інші елементи вектора невідомих обчислюються в процесі зворотної прогонки за формулою (6.19). Таким чином, алгоритм методу прогонки для розв'язування СЛАР із тридіагональною матрицею набуває вигляду.

1. Реалізувати пряму прогонку:

а) знайти $A_1 = -c_{-1} / b_1,\ B_1 = d_1 / b_1$;

б) для $i = 2,\ 3,...,n-1$ 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 $|b_i| \ge |a_i| + |c_i|$, причому хоча б для одного значення *i* має 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, коли діагональних елементів більше. Зазначимо, що в більшості практичних випадків ці умови виконуються, і метод прогонки (лістинг 6.8) можна реалізувати без вибору головного елемента.

Лістинг 6.8. Реалізація методу прогонки

```
# -*- 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])

#зворотний хід
for i in xrange(N-1,-1,-1):
 b[i]+=b[i+1]*a[i][i+1]
```

Модуль складається з однієї функції **progonka**, яка вирішує СЛАР методом прогонки. На вхід функції подаються два параметри: матриця A та вектор-стовпець B.

6.2.9. Погано зумовлені 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, зі зростанням розміру системи зростає й кількість необхідних операцій. При цьому помилки округлення стають більш суттєвими. Але і для систем однакового розміру отримуватимемо різні похибки.

Існують так звані погано зумовлені СЛАР, розв'язувати які важко будь-яким методом, а в більшості випадків немає сенсу шукати розв'язок [2].

Систему лінійних рівнянь називають погано зумовленою, якщо незначні зміни елементів матриці коефіцієнтів або правих частин призводять до надто значних змін у розв'язку.

Наприклад, розглянемо систему тільки двох рівнянь:

$$\begin{cases} 0,832 \cdot x_1 + 0,448 \cdot x_2 = 1, \\ 0,784 \cdot x_1 + 0,421 \cdot x_2 = 0, \end{cases} \tag{6.20}$$

точним розв'язком якої з точністю до трьох знаків є

$$x_1 = -439,\ \ x_2 = 817.$$

Розв'яжемо її методом Гаусса з вибором головного елемента по всьому полю. Цей метод, як ми встановили в п. 6.2.4, є достатньо надійним. Припустимо, що задачу обчислюють за допомогою гіпотетичної ЕОМ із трьома десятковими розрядами.

Оскільки a_{11} є максимальним за модулем серед елементів матриці, то ніяких перестановок рядків і стовпців не проводимо, а зразу виконуємо процедуру Гаусса:

$$R = \frac{a_{21}}{a_{11}} = \frac{0,784}{0,832} \approx 0,942308 \approx 0,942;$$

$$a_{22}^{(1)} = a_{22} - a_{12}R = 0,421 - 0,448 \cdot 0,942 \approx 0,421 - 0,422016 \approx$$

$$\approx 0,421 - 0,422 = -0,001;$$

$$b_2^{(1)} = b_2 - b_1 R = 0 - 1 \cdot 0,942 = -0,942.$$

У 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

$$x_1 = -506,\ \ x_2 = 942 \quad B_k\,. \tag{6.21}$$

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). Як зазначено в літературі [2], насправді ми розв'язали систему

$$f_{k-1} \begin{cases} 0,832 \cdot x_1 + 0,447974... \cdot x_2 = 1, \\ 0,783744... \cdot x_1 + 0,420992... \cdot x_2 = 0, \end{cases} \tag{6.22}$$

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 відрізняються на 15 %, тобто помилки початкових 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. 6.5). Визначник цієї матриці наближається до нуля, її рядки майже пропорційні.

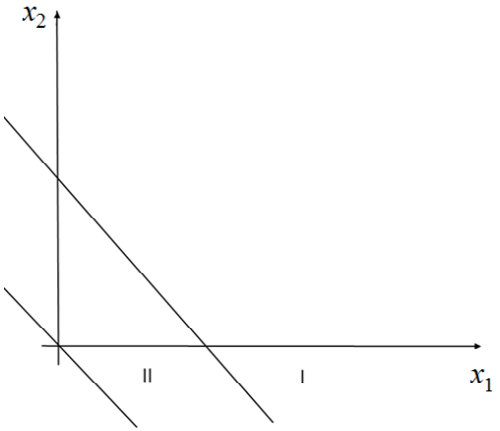


Рис. 6.5. Прямі, що визначаються погано зумовленою системою (6.20)

Розглянемо систему рівнянь, отриману із системи (6.20):

$$x - x^{(k-1)} \begin{cases} 0,832 \cdot x_1 + 0,447974... \cdot x_2 = 1, \\ 0,784 \cdot x_1 + (0,421 + \varepsilon) \cdot x_2 = 0, \end{cases} \tag{6.23}$$

її друге рівняння визначає сім'ю прямих, яка залежить від параметра ε.

При $\varepsilon = 0$ ця система збігається із системою (6.20). Якщо ε збільшується від нуля до 0,012, пряма II на рис. 6.5 повертається проти годинникової стрілки до такого положення, в якому вона

паралельна прямій I. При цьому система (6.23) не матиме розв'язку, визначник дорівнюватиме нулю. У процесі обертання точка перетину двох прямих віддалятиметься в нескінченність. За такого зменшення параметра є до значення, при якому система вироджується, навіть невеликі зміни коефіцієнтів системи призводять до все більших змін у розв'язку.

Очевидно, із погляду стійкості розв'язку система двох рівнянь буде "ідеально зумовленою", якщо визначені нею прямі взаємно перпендикулярні.

Відомо, що коли система вироджена, її визначник дорівнює нулю. Але чи випливає з його малого значення погана зумовленість 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 зумовлені". Проте, якщо матриця системи масштабована, як це вказано далі, визначник може слугувати мірою зумовленості системи.

Очевидно, для системи двох рівнянь

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = 0, \\ a_{21}x_1 + a_{22}x_2 = 0 \end{cases}$$

доброю мірою "паралельності" двох відповідних прямих може бути кут між ними або площа ромба (з одиничною стороною), побудованого на цих прямих (рис. 6.6).

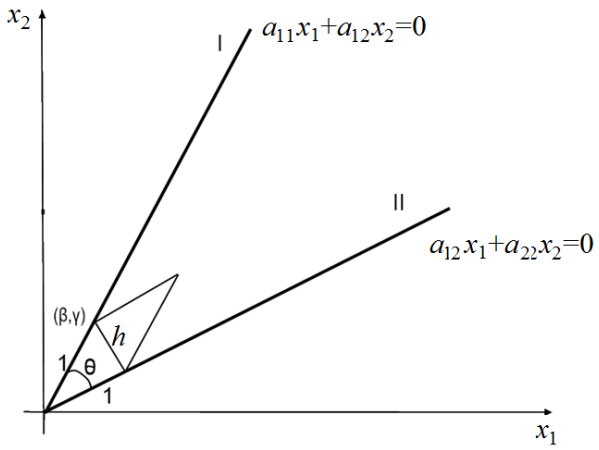


Рис. 6.6. Оцінка зумовленості системи двох рівнянь

Позначимо висоту ромба через h. Тоді його площа також дорівнюватиме h. Оскільки $h=\sin\theta$ (θ – кут між прямими), то площа цього ромба змінюватиметься 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 оцінювання зумовленості системи. Аналогічна формула є і для системи n рівнянь:

$$V = \frac{\left| \det A \right|}{\alpha_1\alpha_2 \cdots \alpha_n}, \qquad (6.25)$$

де $\alpha_1 = \sqrt{a_{i1}^2 + a_{i2}^2 + \ldots + a_{in}^2}$.

Залежність (6.25) можна записати інакше:

$$V_1 = \det\begin{pmatrix} \dfrac{a_{11}}{\alpha_1} & \dfrac{a_{12}}{\alpha_1} & \ldots & \dfrac{a_{1n}}{\alpha_1} \\ & \ldots & & \\ \dfrac{a_{n1}}{\alpha_n} & \dfrac{a_{n2}}{\alpha_n} & \ldots & \dfrac{a_{nn}}{\alpha_n} \end{pmatrix}; \qquad V = \left| V_1 \right|,$$

тобто для оцінювання зумовленості 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 початкової системи. Як ми переконалися на прикладі систем (6.20) та (6.22) незначні зміни в коефіцієнтах погано зумовленої 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 приладом, який забезпечує три правильні цифри числа. Тому замість системи (6.22) ми почали б розв'язувати систему (6.20) і навіть, як-

що б розв'язали її абсолютно точно, то одержали б відповідь $(-439; 817)$. Тобто для погано зумовлених систем необхідно не лише вміти знаходити точний розв'язок, але треба й достатньо точно вимірювати коефіцієнти системи.

Класичним прикладом погано зумовленої матриці є матриця Гільберта:

$$H_n = \begin{pmatrix} & & & \\ 1 & \frac{1}{2} & \ldots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \ldots & \frac{1}{n+1} \\ \ldots & & & \\ \frac{1}{n} & \frac{1}{2} & \ldots & \frac{1}{2n-1} \end{pmatrix}.$$

Зі збільшенням n вона стає все більш погано зумовленою. Наприклад, якщо при $n=8$ занесемо в пам'ять ЕОМ таку матрицю у вигляді восьмизначних десяткових чисел й обернемо її точно, то отримана обернена матриця відрізнятиметься від точної оберненої до H_8 матриці вже в першому знаку.

Для погано зумовлених систем характерна оманливість вектора відхилів: $r = Ax - B$.

Якщо ми одержали результат, що збігається з точним, то вектор відхилів дорівнюватиме нулю. Можна припустити, що коли ми одержали добре наближення до точного розв'язку, то вектор відхилів буде малим. І навпаки, якщо вектор відхилів малий, то ми одержали добре наближення. Але для погано зумовлених систем це не так. Наприклад, розглянемо систему

$$\begin{aligned} 0,780x_1 + 0,563x_2 &= 0,217, \\ 0,913x_1 + 0,659x_2 &= 0,254 \end{aligned}$$

із точним розв'язком $(1, -1)$.

Якби ми одержали результат, який помітно відрізняється від точного, $x = \begin{pmatrix} 0,341 \\ -0,087 \end{pmatrix}$, то відхил був би таким: $r = \begin{pmatrix} 10^{-6} \\ 0 \end{pmatrix}$.

Інший наближений розв'язок $x = \begin{pmatrix} 0,999 \\ -1,001 \end{pmatrix}$, що майже збігається з точним, дає відхил $r = \begin{pmatrix} -0,0013 \\ 0,0015 \end{pmatrix}$.

Таке значення гірше від попереднього, хоча вектор розв'язку ближче до точного. Таким чином, розмір відхилу для погано зумовлених систем може "збивати з пантелику".

Погану зумовленість матриці можна виявити не тільки за допомогою формули (6.25), але й використовуючи поняття норми вектора й матриці. Нагадаємо деякі види норм вектора x і матриці A:

$$F'(x^{(1)})\ \|x\|_1 = \sum_{i=1}^{n} |x_1|;\ \|x\|_2 = \sqrt{n \sum_{i=1}^{} x_i^2};\ \|x\|_\infty = \max_i |x_i|; \quad (6.26)$$

$$\|A\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|;\ \|A\|_\infty = \max_j \sum_{i=1}^{n} |a_{ij}|. \quad (6.27)$$

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 виконуються рівності

$$\begin{aligned} Ax^* &= B, \\ A(x^* + \Delta x) &= B + \Delta B. \end{aligned} \quad (6.28)$$

Підставивши перший вираз у другий, одержимо

$$A \cdot \Delta x = \Delta B$$

або

$$\begin{aligned} \Delta x &= A^{-1} \cdot \Delta B, \\ \|\Delta x\| &\le \left\|A^{-1}\right\| \cdot \|\Delta B\|. \end{aligned} \tag{6.29}$$

Звідси випливає, що коли $\left\|A^{-1}\right\|$ велика, то невеликі зміни вектора B можуть призвести до значних змін у розв'язку. Перейдемо до відносних величин. Із (6.28) видно, що

$$\|B\| \le \|A\| \cdot \|x^*\|. \tag{6.30}$$

Помножимо нерівності (6.29) і (6.30):

$$\|\Delta x\| \cdot \|B\| \le \left\|A^{-1}\right\| \cdot \|\Delta B\| \cdot \|A\| \cdot \|x^*\|$$

та поділимо отриманий результат на $\|x^*\| \cdot \|B\|$:

$$\frac{\|\Delta x\|}{\|x^*\|} \le \|A\| \cdot \left\|A^{-1}\right\| \cdot \frac{\|\Delta B\|}{\|B\|}. \tag{6.31}$$

Звідси випливає, що відносна зміна x^*, зумовлена зміною вектора правої частини B, обмежена відносною змінною B, помноженою на $\|A\| \cdot \left\|A^{-1}\right\|$. Останню величину називають *числом зумовленості* матриці A й позначають $\operatorname{cond}(A)$:

$$\operatorname{cond}(A) = \|A\| \cdot \left\|A^{-1}\right\|.$$

Зазначимо, що $\operatorname{cond}(A) \ge 1$. Матриці, в яких $\operatorname{cond}(A)$ велике, називають *погано зумовленими*.

Розглянемо тепер, як змінюється вектор розв'язку в разі зміни матриці A. Нехай $x^* + \delta x$ – розв'язок системи $(A + \delta A)x = B$, тобто

$$(A + \delta A) \cdot (x^* + \delta x) = B.$$

Використавши (6.28), отримаємо

$$Ax^* + a \cdot \delta A \cdot x^* + \delta A \cdot \delta x = B,$$

$$A \cdot \delta x + \delta A \cdot x^* + A \cdot \delta x = 0,$$

$$A \cdot \delta x = -\delta A(x^* + \delta x),$$

$$\delta x = -A^{-1} \cdot \delta A(x^* + \delta x),$$

$$\|\delta x\| \le \left\|A^{-1}\right\| \cdot \|\delta A\| \cdot \|x^* + \delta x\| = \|A\| \cdot \left\|A^{-1}\right\| \cdot \|\delta A\| \cdot \|x^* + \delta x\| \cdot \frac{1}{\|A\|} =$$

$$= \mathrm{cond}(A) \cdot \frac{\|\delta A\|}{\|A\|} \cdot \|x^* + \delta x\|.$$

Поділимо це співвідношення на $\|x^* + dx\|$:

$$\frac{\|\delta x\|}{\|x^* + \delta x\|} \le \mathrm{cond}(A) \cdot \frac{\|\delta A\|}{\|A\|}. \qquad (6.32)$$

В одержаній оцінці зміни вектора розв'язку знову, як і в (6.31), бере участь $\mathrm{cond}(A)$. Нерівності (6.31), (6.32) необхідно інтерпретувати так: якщо $\mathrm{cond}(A)$ мале (не набагато перевищує одиницю), то незначні зміни в СЛАР ведуть до незначних змін у розв'язку. Якщо ж $\mathrm{cond}(A)$ велике, то не обов'язково незначні зміни системи призведуть до великих змін у розв'язку.

Підсумовуючи, зазначимо, що деякі погано зумовлені системи не потрібно навіть намагатися розв'язати, а потрібно або переформулювати задачу, або точніше виміряти необхідні дані.

6.3. Ітераційні методи

До сновних переваг ітераційних методів належать такі.

1. Якщо процес ітерації збігається швидко, тобто кількість наближень менша, ніж порядок системи, то отримаємо виграш у часі розв'язування системи.

2. Метод ітерацій є таким, що самокоригується, тобто окрема помилка обчислення не впливає на остаточний результат розв'язування.

3. Процес ітерацій легко програмується на ЕОМ.

4. Деякі методи ітерацій стають особливо вигідними при розв'язуванні систем, в яких значна кількість коефіцієнтів, розташованих підряд, дорівнює нулю (розріджені матриці).

Таким чином, ітераційні методи особливо ефективні для СЛАР високої розмірності і СЛАР із розрідженою матрицею. Для погано зумовлених 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-й номер ітерації) залежить від A, B, $x^{(k-1)}$, $x^{(k-2)}$,..., $x^{(k-r)}$.

Для економії пам'яті ЕОМ звичайно беруть $r=1$ так, що $x=F_k\left(A,B,x^{(k-1)}\right)$, і такі методи називають *однокроковими* на відміну від *багатокрокових*, в 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

$$F_k = Hx^{(k-1)} + Va,$$

де H – деяка матриця; V – вектор.

Природно, 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

$$x^{(k)} = Hx^{(k-1)} + V,\ k=1,2,...,. \qquad (6.33)$$

Умови збіжності таких методів визначаються теоремами 6.2 і 6.3.

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 й достатні умови сформульовано в теоремі (6.3), яка є основним теоретичним результатом для методів (6.33).

Теорема 6.3. Для збіжності ітераційного процесу (6.33) при будь-якому початковому наближенні необхідно й достатньо, щоб усі власні значення матриці H були за модулем меншими за одиницю, тобто $\lambda_i < 1$.

Інакше кажучи, спектральний радіус $r(H)$ матриці H має 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:

$$\left\| x^* - x^{(k)} \right\| \le \frac{\|H\|}{1 - \|H\|} \cdot \left\| x^{(k)} - x^{(k-1)} \right\|.$$

У конкретних ітераційних методах використовують матрицю H і вектор V, коли зводять 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 із 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]:

$$\begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{pmatrix} = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ -a_{21} & 1-a_{22} & -a_{23} \\ -a_{31} & -a_{32} & 1-a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ x_3^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad (6.37)$$

або

$$x^{(k)} = \left(E - A\right)x^{(k-1)} + B. \quad (6.38)$$

Такий запис методу використовують тільки для теоретичних цілей, а при фактичних обчислюваннях застосовують покомпонентні формули.

При підготовці задачі до розв'язку на ЕОМ не треба вручну перетворювати системи 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

$$x_i^{(k)} = x_i^{(k-1)} - \sum_{j=1}^{n} a_{ij}x_j^{k-1} + b_i \quad (6.39)$$

або

$$x_i^{(k)} = (1-a_{ii})x_i^{(k-1)} - \sum_{j=f,\, j\neq i}^{n} a_{ij}x_j^{k-1} + b_i, \quad (6.40)$$

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 слід звести вручну еквівалентними перетвореннями системи перед уведенням 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 $x = 2,5$ таким методом, перетворимо це рівняння:

$$0 = -0,4x + 1,$$
$$x = x - 0,4x + 1,$$
$$x = 0,6x + 1.$$

Очевидно, розв'язком цього рівняння, еквівалентного початковому, є абсциса точки перетину двох прямих $y = x$ та $y = 0,6x + 1$ (рис. 6.7, *а*).

Візьмемо деяке початкове наближення $x^{(0)}$ й 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 *x*, до перетину з прямою $y = x$ у точці B. Ця точка має ту саму ординату, що й 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 самі дії, тобто з точки x на осі x установи-

ти перпендикуляр до його перетину з прямою $y = 0,6x + 1$ у точці C, через неї провести горизонтальну пряму до перетину з прямою $y = x$ у точці D, яка має абсцису x, і т. д.

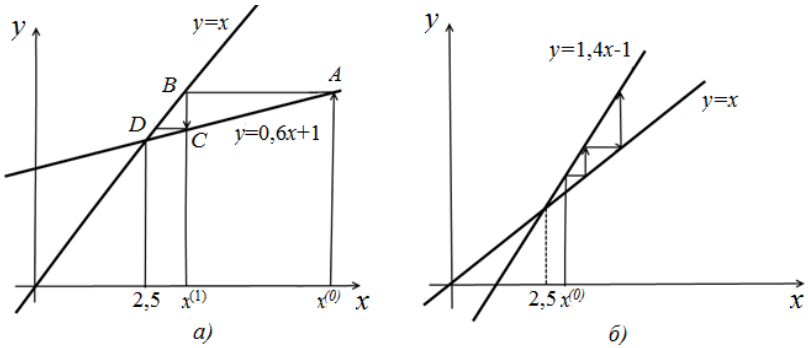


Рис. 6.7. Геометрична інтерпретація методу послідовних наближень

Побудуємо ітераційний процес

$$x^{(k)} = 0,6x^{(k-1)} + 1,\ k = 1,2,...$$

Як матриця H тут виступає кутовий коефіцієнт, який дорівнює $0,6$. Процес на рис. 6.7, *а* збігається, оскільки виконується умова $\|H\| < 1$. На рис. 6.7, *б* зображено розбіжний ітераційний процес, побудований для того самого рівняння $0,4x = 1$. Справді,

$$0 = 0,4x - 1,$$
$$x = x + 0,4x - 1,$$
$$x = 1,4x - 1,$$
$$x^{(k)} = 1,4x^{(k-1)}.$$

Тут $\|H\| = 1,4 > 1$ і не виконується достатня умова збіжності. Крім того, значення $r(H)$ теж більше за одиницю і процес повинен розбігатися. Таким чином, для однієї СЛАР $Ax = B$ ітера-

ційний 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 в 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

$$\|H\|_1 = \|E - A\|_1 = \max_j \sum_{i=1}^{n} \left| h_{ij} \right| = 0{,}7 < 1.$$

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, доки два послідовних наближення не стануть достатньо близькими. Одержана послідовність векторів

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \to \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} \to \begin{pmatrix} 0,9 \\ 2,8 \\ 1,7 \end{pmatrix} \to \begin{pmatrix} 0,96 \\ 2,85 \\ 1,76 \end{pmatrix} \to ...$$

збігатиметься до точного розв'язку системи. Результати наведені далі в табл. 6.1. При програмуванні методу не потрібно зберігати в пам'яті ЕОМ усі вектори $x^{(k)}$, які одержуємо, достатньо двох останніх для оцінюваннння різниці між ними. У загальному випадку ітераційний процес закінчується, якщо $\left\| x^{(k)} - x^{(k-1)} \right\| < \varepsilon$, тобто для всіх елементів векторів виконується умова $g_i < \varepsilon$, де

$$g_i = \begin{cases} \left| x_i^{(k)} - x_i^{(k-1)} \right| & \text{при } \left| x_i^{(k)} \right| \le 1, \\ \dfrac{\left| x_i^{(k)} - x_i^{(k-1)} \right|}{\left| x_i^{(k)} \right|} & \text{при } \left| x_i^{(k)} \right| > 1. \end{cases} \qquad (6.43)$$

Тут i – номер елемента вектора; k – номер ітерації; ε – допустима похибка (наприклад, $\varepsilon = 0,001$).

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:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1,$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2,$$
$$a_{31}x_1 + a_{32}x_2 + a_{23}x_3 = b_3.$$

Припустимо, що діагональні коефіцієнти a_{ii} відмінні від нуля (інакше можна переставити рівняння). Виразимо з першого рівняння x_1, із другого x_2, із третього x_3:

$$x_1 = \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 - a_{13}x_3\right),$$
$$x_2 = \frac{1}{a_{22}}\left(b_2 - a_{21}x_1 - a_{23}x_3\right),$$
$$x_3 = \frac{1}{a_{33}}\left(b_3 - a_{31}x_1 - a_{32}x_2\right),$$

а потім побудуємо ітераційний процес

$$x_1^{(k)} = \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^{(k-1)} - a_{13}x_3^{(k-1)}\right),$$
$$x_2^{(k-1)} = \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{(k-1)} - a_{23}x_3^{(k-1)}\right), \qquad (6.44)$$
$$x_3^{(k-1)} = \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{(k-1)} - a_{32}x_2^{(k-1)}\right),$$

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 $Ax = B$:

$$x^{(k)} = (E - D^{-1}A)x^{(k-1)} + D^{-1}B. \tag{6.46}$$

Із загальної умови $\|H\| < 1$ – збіжності лінійних ітераційних методів, випливає, що метод простої ітерації збігається, якщо виконується хоча б одна з двох умов:

$$\begin{aligned} |a_{ii}| &> \sum_{j=1, j\neq i}^{n} |a_{ij}|,\ i = 1,2,...,n; \\ |a_{jj}| &> \sum_{i=1, i\neq i}^{n} |a_{ij}|,\ j = 1,2,...,n. \end{aligned} \tag{6.47}$$

Інакше кажучи, 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. Такі умови є достатніми, але не необхідними, тобто для деяких систем ітерації збігаються і при порушенні умов (6.47).

Оцінювання кількості потрібних ітерацій для одержання розв'язку з точністю ε визначають формулою

$$k \approx \frac{\ln \dfrac{p\varepsilon}{pq+u}}{\ln(1-p)},$$

де

$$p = \max_i \sum_{j=1, j\neq i}^{n} \left|\frac{a_{ij}}{a_{ii}}\right|, \quad q = \max_i \left|x_{ij}^{(0)}\right|, \quad u = \max_i \left|\frac{b_i}{a_{ii}}\right|.$$

Для виконання однієї ітерації потрібно виконати в цьому методі n операцій ділення, ($n^2 - n$) операцій множення, n операцій додавання (n – порядок системи).

Наведемо обчислювальну схему методу простої ітерації. До початку ітераційного процесу задаємо точність і початкове наближення $x^{(0)}$. Потім виконуємо такі дії:

1. Обчислюємо чергове наближення до вектора розв'язку

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k-1)}\right), \quad i = 1, 2, ..., n.$$

2. Перевіряємо умову закінчення процесу

$$\int_a^b f(x)dx = \sum_{i=1}^{n} f(x_i)h_i + R_{\text{пр}}, \quad g_i < \varepsilon, \ i = 1, 2, ..., n,$$

$$\text{де } g_i = \begin{cases} \left| x_i^{(k)} - x_i^{(k-1)} \right| & \text{при } \left| x_i^{(k)} \right| \le 1, \\ \dfrac{\left| x_i^{(k)} - x_i^{(k-1)} \right|}{\left| x_i^{(k)} \right|} & \text{при } \left| x_i^{(k)} \right| > 1. \end{cases}$$

Якщо умова виконується для всіх i, то $x^{(k)}$ вважатимемо розв'язком, в іншому разі виконуємо чергову ітерацію з п. 1.

Приклад 6.2. Розв'язати методом простої ітерації з точністю до $\varepsilon = 0,01$ систему трьох рівнянь

$$\begin{aligned} 12x_1 - 3x_2 + x_3 &= 9, \\ x_1 + 5x_2 - x_3 &= 8, \\ x_1 - x_2 + 3x_3 &= 8. \end{aligned} \tag{6.48}$$

Як легко переконатися, виконавши перевірку, точним розв'язком цієї системи є $x_1 = 1,\ x_2 = 2,\ x_3 = 3$. Перевіримо, чи виконуються достатні умови збіжності методу простої ітерації:

$$\begin{aligned} |12| &> |-3| + |1|, \\ |5| &> |1| + |-1|, \\ |3| &> |1| + |-1|. \end{aligned}$$

Умови виконуються, тому використовуючи формулу (6.44), одержимо результати, наведені в п. 6.33. У лістингу 6.9 наведено одну з можливих реалізацій методу простої ітерації.

Лістинг 6.9. Реалізація 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
```

Ця реалізація використовує методику з частковим упорядкуванням (функція **diagonal_prevalence**). На даних з прикладу 6.2 ця програма видає таке:

```
>>>
Метод Якобі
Діагональна перевага виконується
Вектор початкового наближення
[ 0.0.0.]
[ 1.000802472.000382723.0013786 ]
Вектор відхилів: [[ 0.00986008 -0.99866255 -0.99544444]
 [ 1.009860080.001337450.00455556]
 [ 1.009860080.001337450.00455556]]
>>>
```

Як бачимо, отримано ті самі результати.

6.3.3. Метод Зейделя

За способом зведення системи $Ax = B$ до вигляду $x = Hx + V$ цей метод аналогічний методу послідовних наближень, однак тільки що обчислена компонента $x_i^{(k)}$ вектора невідомих одразу використовується для обчислення наступної компоненти $x_{i+1}^{(k)}$. Інакше кажучи, для обчислення $x_{i+1}^{(k)}$ використовуються нові значення $x_1^{(k)}, x_2^{(k)}, ..., x_i^{(k)}$ і старі значення $x_{i+1}^{(k-1)}, \ldots, x_n^{(k-1)}$.

Проілюструємо метод на прикладі системи трьох рівнянь

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2, \\ a_{31}x_1 + a_{32}x_2 + a_{23}x_3 &= b_3. \end{aligned} \tag{6.49}$$

Як і в методі послідовних наближень, додамо до обох частин *i*-го рівняння x_i й 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 $x_3^{(0)}$ із 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 закінчується перша ітерація. Наступні ітерації виконують аналогічно. Наближення з номером k можна записати у вигляді

$$
\begin{aligned}
x_1^{(k)} &= x_1^{(k-1)} - (a_{11}x_1^{(k-1)} + a_{12}x_2^{(k-1)} + a_{13}x_3^{(k-1)}) + b_1,\\
x_2^{(k)} &= x_2^{(k-1)} - (a_{21}x_1^{(k)} + a_{22}x_2^{(k-1)} + a_{23}x_3^{(k-1)}) + b_2,\\
x_3^{(k)} &= x_3^{(k-1)} - (a_{31}x_1^{(k)} + a_{32}x_2^{(k)} + a_{33}x_3^{(k-1)}) + b_3.
\end{aligned}
$$

Ітераційний процес продовжується доти, доки значення $x_1^{(k)}$, $x_2^{(k)}$, $x_3^{(k)}$ не стануть близькими (із заданою похибкою) до значень $x_1^{(k-1)}$, $x_2^{(k-1)}$, $x_3^{(k-1)}$.

У системі з n рівнянь обчислення виконують за формулою

$$x_i^{(k)} = x_i^{(k-1)} - \left(\sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} + \sum_{j=1}^{n} a_{ij} x_j^{(k-1)}\right) + b_i. \quad (6.51)$$

Умови збіжності методу Зейделя такі самі, як і для методу послідовних наближень, і в більшості випадків він збігається швидше методу послідовних наближень. У табл. 6.1. наведено результати розв'язання системи (6.41) методами послідовних наближень і Зейделя.

Таблиця 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 подати так:

$$x = Hx + V,$$

де

$$H = \begin{pmatrix} 1-a_{11} & -a_{12} & -a_{13} \\ -a_{21} & 1-a_{22} & -a_{23} \\ -a_{31} & -a_{32} & 1-a_{33} \end{pmatrix} = E - A, \quad V = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

Матрицю H представимо як суму 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 запишемо ітераційний процес у 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 методу 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 Гаусса – 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 невідоме x_i, доведемо до вигляду

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3), \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3), \\ x_3 &= \frac{1}{a_{33}}(b_3 - a_{31}x_1 - a_{32}x_2). \end{aligned} \tag{6.52}$$

Цей метод полягає в тому, що визначивши деяке початкове наближення $x_1^{(0)}$, $x_2^{(0)}$, $x_3^{(0)}$ і використовуючи щойно обчислені компоненти вектора x (для знаходження чергових компонент), обчислення виконують за такими формулами:

$$x_1^{(k)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k-1)} - a_{13}x_3^{(k-1)}),$$
$$x_2^{(k)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k-1)}), \qquad (6.53)$$
$$x_3^{(k)} = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)}).$$

У загальному випадку системи n рівнянь i-та компонента вектора розв'язку на k-й ітерації обчислюється як

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{\grave{o}} a_{ij}x_j^{(k-1)}\right). \qquad (6.54)$$

Систему (6.52) у матричній формі можна записати так:

$$x = Hx + V,$$

де

$$H = -D^{-1}(L+R), \qquad V = D^{-1}B, \qquad A = L + D + R,$$
$$L = \begin{pmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{31} & a_{32} & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 0 & a_{12} & a_{13} \\ 0 & 0 & a_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad D = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}.$$

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 в цих методах і достатні умови збіжності (діагональне домінування матриці A). Найчастіше метод Некрасова збігається швидше, ніж метод простої ітерації. Як ілюстрація в табл. 6.2 наведено результати розв'язання системи (6.48) методами простої ітерації і Некрасова.

Таблиця 6.2. **Порівняння двох ітераційних методів**

	Метод простої інтерації			**Метод Некрасова**		
k	x_1	x_2	x_3	x_1	x_2	x_3
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			

Реалізацію методу Гаусса – Зейделя у вигляді окремого модуля наведено в лістингу 6.10.

Лістинг 6.10. Реалізація методу Гаусса – Зейделя

```
# -*- 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 має 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 фактора 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'
```

Розглянемо приклад застосування наведеного модуля.

Приклад 6.3. Нехай потрібно розв'язати систему рівнянь методом Зейделя. Потрібно написати програму, яка працюватиме з будь-яким числом рівнянь n.

$$\begin{bmatrix} 2 & -1 & 0 & 0 & & 0 & 0 & 0 & 1 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & & 0 & 0 & 0 & 0 \\ & & & & \dots & & & & \\ 0 & 0 & 0 & 0 & & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 & -1 \\ 1 & 0 & 0 & 0 & & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Розв'язати систему для $n = 20$. Можна показати, що точний розв'язок системи такий: $x_i = -\frac{n}{4} + \frac{i}{2}$.

Розв'язання. Згідно з (6.55)запишемо

$$x_1 = \frac{w(x_2 - x_n)}{2} + (1-w)x_1,$$

$$x_i = \frac{w(x_{i-1} + x_{i+1})}{2} + (1-w)x_i, \quad i = 2,3,\dots,n-1,$$

$$x_n = \frac{w(1 - x_1 + x_{n-1})}{2} + (1-w)x_n.$$

Ці формули визначаються у функції **iterEqs**. Програму наведено в лістингу 6.11.

Лістинг 6.11. Програма застосування модуля

```
# -*- 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 enter для виходу")
```

Наведемо результат виконання програми:

```
>>>
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 enter для виходу
```

Контрольні запитання

1. Чим відрізняються прямі методи від ітераційних?
2. До якого вигляду зводиться матриця коефіцієнтів прямого ходу методу Гаусса.
3. Коли не можна застосувати метод Гаусса?
4. Який елемент є провідним у стовпці матриці?
5. У чому перевага методу Гаусса з вибором головного елемента у стовпці?
6. До якого вигляду зводиться матриця в методі Гаусса – Жордано?
7. Чи потрібен зворотний хід у методі Некрасова?
8. Яка умова завершення ітерації в ітераційних методах?
9. Головні переваги методу Гаусса – Зейделя порівняно з методом простих ітерацій.
10. Як перевірити істинність чи хибність знайдених коренів?

Розділ 7

РОЗВ'ЯЗУВАННЯ СИСТЕМ НЕЛІНІЙНИХ РІВНЯНЬ

Нехай потрібно розв'язати 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 $f_1, f_2, ..., f_n$ – задані, узагалі кажучи, нелінійні (серед них можуть бути й лінійні) функції (із дійсними значеннями) n дійсних змінних. Позначивши

$$\overline{x} := \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}, \quad F(x) := \begin{pmatrix} f_1(\overline{x}) \\ f_2(\overline{x}) \\ ... \\ f_n(\overline{x}) \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, ..., x_n) \\ f_2(x_1, x_2, ..., x_n) \\ ... \\ f_n(x_1, x_2, ..., x_n) \end{pmatrix}, \quad \overline{0} := \begin{pmatrix} 0 \\ 0 \\ ... \\ 0 \end{pmatrix},$$

систему (7.1) можна записати одним рівнянням

$$F(\overline{x}) = 0 \tag{7.2}$$

відносно векторної функції F векторного аргументу x. Таким чином, початкове завдання можна розглядати як задачу про нулі нелінійного відображення. У цій постановці вона є прямим узагальненням основного завдання побудови методів знаходження нулів одновимірних нелінійних відображень. Фактично це те саме завдання, лише в просторах більшої розмірності. Тому можна як заново будувати методи її розв'язування на основі розроблених вище підходів, так і здійснювати формальне перенесення виведених для скалярного випадку розрахункових формул. У 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 розв'язування скалярних рівнянь. Проте не всі результати та не всі методи можна перенести з випадку $n=1$ на випадок $n\geq 2$. Наприклад, тут уже не працюватимуть методи дихотомії, оскільки безліч векторів не впорядкована. Водночас перехід від $n=1$ до $n\geq 2$ вносить до задачі знаходження нулів нелінійного відображення свою специфіку, облік якої призводить до появи нових методів і до різних модифікацій тих, що вже є. Зокрема, велика варіативність методів розв'язування нелінійних систем пов'язана з різноманітністю способів, якими можна розв'язувати 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*-матриць. Тоді, очевидно, послідовність завдань

$$x = x - A_k F(x), \;\; k = 0,1,2,\ldots,$$

має ті самі розв'язки, що й початкове рівняння (7.2), і для наближеного знаходження цих розв'язків можна формально записати ітераційний процес

$$x^{(k+1)} = x^{(k)} - A_k F(x^{(k)}),\; k \;=\; 0,1,2,\ldots, \qquad (7.3)$$

що має вигляд методу простих ітерацій (п. 7.2.2 формула (7.17)) при $\Phi(x) := \Phi_k(x) := x - A_k F(x)$. У випадку $A_k = A$ – це справді метод простої ітерації з лінійною збіжністю послідовності

($x = x_{i-1}$). Якщо ж A_k різні при різних k, то формула (7.3) визначає велику сім'ю ітераційних методів із матричними параметрами. Розглянемо деякі з методів цієї сім'ї.

Нехай $A_k := \left[F'(x^{(k)}) \right]^{-1}$, де

$$F'(x) = J(x) = \begin{pmatrix} \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 & \dots & \partial f_1 / \partial x_n \\ \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 & \dots & \partial f_2 / \partial x_n \\ \dots & \dots & \dots & \dots \\ \partial f_n / \partial x_1 & \partial f_n / \partial x_2 & \dots & \partial f_n / \partial x_n \end{pmatrix} \qquad (7.4)$$

матриця Якобі вектор-функції $F(x)$. Підставивши 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 скалярного методу Ньютона (див. формули (4.2, 4.3)). Формулу (7.5), що вимагає обернення матриць на кожній ітерації, можна переписати в неявному вигляді

$$F'(x^{(k)})(x^{(k+1)} - (x^{(k)}) = -F(x^{(k)}). \qquad (7.6)$$

Використання (7.6) передбачає при кожному $k = 0,1,2,...$ розв'язування лінійної алгебраїчної системи

$$F'(x^{(k)} p^{(k)}) = -F(x^{(k)}) \qquad (7.7)$$

відносно векторної поправки, а потім збільшення цієї поправки до поточного наближення для отримання

$$x^{(k+1)} = x^k + p^{(k)}. \qquad (7.8)$$

До розв'язування таких лінійних систем залучають найрізноманітніші методи – як прямі, так і ітераційні, залежно від розмірності n розв'язуваної задачі та специфіки матриць Якобі $J(x^{(k)})$. Наприклад, можна враховувати їх симетричність, розрідженість тощо.

Порівнюючи (7.6) із формальним розвиненням $F(x)$ в ряд Тейлора

$$F(x) = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) + \frac{1}{2!}F''(x^{(k)})(x - x^{(k)})^2 + \dots,$$

бачимо, що послідовність (x_k) у методі Ньютона одержуємо в результаті заміни при кожному $k = 0,1,2,\dots$ нелінійного рівняння $F(x) = 0$ або, що те саме (при достатній гладкості $F(x)$), рівняння

$$F(x) = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) + \frac{1}{2!}F''(x^{(k)})(x - x^{(k)})^2 + \dots = 0$$

лінійним рівнянням

$$F(x) = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) = 0,$$

тобто з покроковою лінеаризацією. Унаслідок цього, можна розраховувати, що при достатній гладкості $F(x)$ і досить доброму 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 певних припущеннях, зокрема, теорема 7.1.

Новим, порівняно зі скалярним випадком, чинником, що ускладнює застосування методу Ньютона для розв'язування n-вимірних систем, є необхідність розв'язування n-вимірних лінійних завдань на кожній ітерації (обернення матриць в (7.5) або розв'язування СЛАР в (7.6)), обчислювальні витра-

ти на які зростають зі збільшенням n, узагалі кажучи, непропорційно швидко. Зменшення таких витрат – один із напрямків модифікації методу Ньютона.

7.1.2. Модифікований метод Ньютона

Якщо матрицю Якобі $F'(x)$ обчислити й обернути лише один раз – у 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 порівняно з основним методом Ньютона (7.5), оскільки, як окремий випадок методу послідовних ітерацій (МПІ), $A := \left[F'(x^{(0)})\right]^{-1}$, він має лише швидкість збіжності геометричної прогресії.

Компромісний варіант – це обчислення і обернення 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 приводить 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 двоступінчастого 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 – лише одним кроком процесу 2-го 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 обернених матриць:

$$\begin{cases} x^{(k+1)} = x^{(k)} - A_k F(x^{(k)}), \\ \psi_k = E - F'(x^{(k+1)})A_k, A_{k+1} = A_k + A_k\psi_k, \end{cases} \tag{7.12}$$

де $k = 0,1,2,...$, а $x^{(0)}$ і $A^{(0)}$ – початковий вектор і матриця ($\left[\approx F'(x^{(0)})\right]^{-1}$). Цей метод (називатимемо його коротше ААМН – апроксимаційний аналог методу Ньютона) має просту схему обчислень – почергове виконання векторних у першому рядку і матричних у другому рядку його запису (7.12) операцій. Швидкість його збіжності майже така сама висока, як і в методі Ньютона. Послідовність ($x^{(k)}$) може квадратично збігатися до розв'язку $x*$ рівняння $F(x) = 0$ (при цьому матрична послідовність (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 самої послідовної апроксимації обернених матриць до простого рекурсивного методу Ньютона (7.10) або, що те саме, до двоступінчастого процесу (7.11) визначає його апроксимаційний аналог

$$\begin{cases} z^{(k)} = x^{(k)} - A_k F(x^{(k)}),\ x^{(k+1)} = z^{(k)} - A_k F(z^{(k)}), \\ \psi_k = E - F'(x^{(k+1)})A_k,\ A_{k+1} = A_k + A_k\psi_k, \end{cases} \tag{7.13}$$

який як і (7.10) можна віднести до методів 3-го порядку. Доказ кубічної збіжності цього методу вимагає вже жорсткіших обмежень на властивості $F(x)$ і близькість $x^{(0)}$ до x^*, A_0 до $[F'(x^{(0)})]^{-1}$, ніж у попередньому методі. Зазначимо, що до поліпшення збіжності тут може привести підвищення порядку апроксимації обернених матриць, наприклад, за рахунок додавання ще одного доданка у формулі для обчислення A_{k+1}:

$$A_{k+1} = A_k + A_k\Psi_k + A_k\Psi_k^2. \tag{7.14}$$

Розглянемо приклад.

Приклад 7.1. Розв'язати 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 (табл. 7.1) із точністю $\varepsilon = 0,001$:

$$Z^{(k+1)} = Z^{(k)} - \left[F'(Z^{(k)}) \right]^{-1} * F(Z^{(k)}).$$

Таблиця 7.1. **Результати обчислень**

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>ε
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> ε
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> ε
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= ε
4	–0,616 –0,788	0,000522 0,0004	–1,434 0,166 –1,232 –1,576	–0,639 –0,067 0,5 –0,582	–0,616 –0,788	0< ε

Відповідь: $Z = \begin{bmatrix} -0,616 \\ -0,788 \end{bmatrix}$.

Блок-схему алгоритму розв'язування систем нелінійних рівнянь методом Ньютона наведено на рис. 7.1.

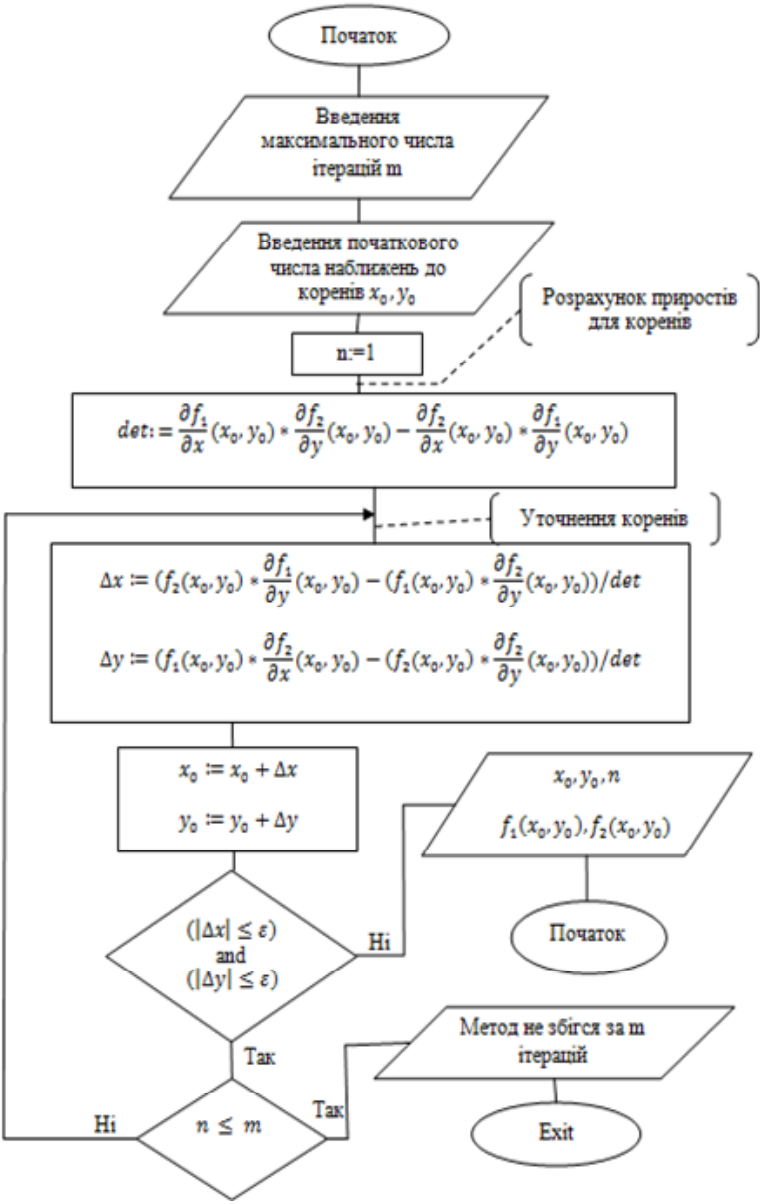


Рис. 7.1. Блок-схема алгоритму

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 заданні послідовності малих векторів $h^{(k)} = (h_1^{(k)},...,h_n^{(k)})^T$ (сталою або такою, що збігається до нуля) отриманий таким шляхом різницевий (або дискретний) метод Ньютона має надлінійну, аж до квадратичної, швидкість збіжності й узагальнює метод на багатовимірний випадок. При заданні векторного параметра h – кроку дискретизації – слід враховувати точність машинних обчислень, точність обчислення значень функцій, середні значення отримуваних наближень.

7.2. Інші методи розв'язування систем нелінійних рівнянь

7.2.1. Метод простих січних

Можна пов'язати задання послідовності ($h^{(k)}$) з якою-небудь векторною послідовністю, що збігається до нуля, наприклад, з послідовністю відхилів ($F(x^{(k)})$) або поправок ($p^{(k)}$). Так, вважаючи $h_j^{(k)} := x_j^{(k-1)} - x_j^{(k)}$ де $j = 1,\ldots,n$, $k = 1,2$, 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

$$B(x^{(k)}, x^{(k-1)}) :=$$

$$= \left(\frac{f_i(x_1^{(k)}, ..., x_j^{(k-1)}, ..., x_n^{(k)}) - f_i(x_1^{(k)}, ..., x_j^{(k)}, ..., x_n^{(k)})}{x_j^{(k-1)} - x_j^{(k)}} \right)_{i,j=1}^{n},$$

$$k = 1, 2, 3, ...$$

Цей метод є двокроковим і вимагає задання 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 саму швидкість і в багатовимірному випадку.

До методу січних так само, як і до методу Ньютона, можна застосувати покрокову апроксимацію обернених 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 $B(x^{(k+1)}, x^{(k)})$ із (7.14).

7.2.2. Метод простих ітерацій

Нехай систему (7.1) зведено до вигляду

$$\begin{cases} x_1 = \phi_1(x_1, x_2, ..., x_n) = 0, \\ x_2 = \phi_2(x_1, x_2, ..., x_n) = 0, \\ \quad ..... \\ x_n = \phi_n(x_1, x_2, ..., x_n) = 0, \end{cases} \tag{7.15}$$

або інакше, в компактному записі

$$\overline{x} = \Phi(\overline{x}), \tag{7.16}$$

де

$$\Phi(\overline{x}) := \begin{pmatrix} \phi_1(\overline{x}) \\ \phi_2(\overline{x}) \\ ... \\ \phi_n(\overline{x}) \end{pmatrix} = \begin{pmatrix} \phi_1(x_1, x_2, ..., x_n) \\ \phi_2(x_1, x_2, ..., x_n) \\ ... \\ \phi_n(x_1, x_2, ..., x_n) \end{pmatrix}.$$

Для цієї *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 побудову послідовності ($x^{(k)}$) із 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 зі швидкістю геометричної прогресії наближатиметься до вектора $x*$ – нерухомої точки відображення $\Phi(x)$. А саме, виконуватиметься така теорема.

Теорема 7.1. Нехай функція $\Phi(x)$ і замкнена множина $M \subseteq D(\Phi) \subseteq R_n$ 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 $\Phi(x)$ 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 виконуються 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 $\Phi(x)$ диференційовна 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 із $M = s$.

Якщо вимагати безперервну диференційовність $\Phi(x)$, то простіше перейти від теореми 7.1 до теореми 7.2, застосувавши таке твердження.

Лема 4.1. Нехай функція $\Phi: R_n \to R_n$ *безперервна і диференційовна на множині* $M \subseteq D(\Phi)$ *і нехай* $\|\Phi'(x)\| \le L \quad \forall x \in M$ *. Тоді* $\Phi(x)$ *задовольняє на множині M умові Ліпшіця*:

$$\|\Phi(x) - \Phi(\tilde{x})\| \le L\|x - \tilde{x}\|.$$

Запис МПІ (7.17) у розгорнутому вигляді, тобто сукупність рекурентної рівності

$$\begin{cases} x_1^{(k+1)} = \phi_1(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}), \\ x_2^{(k+1)} = \phi_2(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{(k+1)} = \phi_n(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}) \end{cases} \tag{7.18}$$

нагадує МПІ для СЛАР, який укладається в цю схему, якщо всі функції – лінійні. Ураховуючи, що в лінійному випадку, як правило, порівняно з МПІ ефективніший метод Зейделя, тут теж може виявитися корисною модифікація. А саме, замість (7.17) можна реалізувати такий метод ітерацій:

$$\begin{cases} x_1^{(k+1)} = \phi_1(x_1^{(k)}, x_2^{(k)}, ..., x_{n-1}^{(k)}, x_n^{(k)}), \\ x_2^{(k+1)} = \phi_2(x_1^{(k+1)}, x_2^{(k)}, ..., x_{n-1}^{(k)}, x_n^{(k)}), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{(k+1)} = \phi_n(x_1^{(k+1)}, x_2^{(k+1)}, ..., x_{n-1}^{(k)}, x_n^{(k)}). \end{cases} \tag{7.19}$$

Зазначимо, що як і для лінійних систем, окремі рівняння в методі (7.19) нерівноправні, тобто переставляння рівнянь системи (7.15) може змінити в якихось межах кількість ітерацій і взагалі ситуацію зі збіжністю послідовності ітерацій.

Щоб застосувати метод простих ітерації або його модифікацію Зейделя (7.19) до початкової системи (7.1), потрібно, як і в скалярному випадку, спочатку тим або іншим способом звести її до вигляду (7.15). Це можна зробити, наприклад, помноживши (7.2) на деяку неособливу $(n \times n)$-матрицю A і додавши до обох частин рівняння $AF(x) = 0$ вектор невідомих x. Отримана система

$$x = x - AF(x) \tag{7.20}$$

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 $\Phi(x) := x - AF(x)$ мала б потрібні властивості.

Реалізацію запропонованого алгоритму у вигляді модуля *Python* наведено в лістингу 7.1.

Лістинг 7.1. Реалізація алгоритму

```
# -*- 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)
```

Модуль складається з двох функцій: **System** та **MPI**. Перша функція задає систему нелінійних рівнянь, друга – реалізацію методу простої ітерації.

Програму використання розробленого модуля наведено в лістингу 7.2. Як приклад використано систему

$$\begin{cases} f_1(x_1, x_2) = 0,1x_1^2 + x_1 + 0,2x_2^2 - 0,3 = 0, \\ f_2(x_1, x_2) = 0,1x_1^2 + x_2 - 0,1x_1x_2 - 0,7 = 0 \end{cases}$$

із початковим наближенням $x_1^{(0)} = 0,25,\ x_2^{(0)} = 0,75$.

Лістинг 7.2. Використання модуля

```
# -*- 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)
```

Результат виконання програми:

```
>>>
Solution is[ 0.0.195324850.71005434]
number of iteration= 3
```

7.2.3. Метод Брауна

На відміну від покрокової лінеаризації векторної функції $F(x)$, що привела до методу Ньютона (7.5), Брауном у 1966 р. запропоновано на кожному ітераційному кроці почергово лінеаризувати компоненти вектор-функції $F(x)$, тобто лінеаризувати в системі (7.1) спочатку функцію f_1, потім f_2 і т. д. А потім послідовно розв'язувати отримувані таким чином рівняння. Аби не затінювати цю ідею громіздкими викладеннями і зайвими індексами, розглянемо виведення розрахункових формул методу Брауна у двовимірному випадку.

Нехай потрібно знайти розв'язок системи

$$\begin{cases} f(x,y)=0, \\ g(x,y)=0 \end{cases} \tag{7.20}$$

і нехай вже отримано наближення x_k, y_k.

Замінимо перше рівняння системи (7.20) лінійним, отриманим по формулі Тейлора для функції з двома змінними:

$$f(x,y) \approx f(x_k,y_k) + f_k^{'}(x_k,y_k)(x-x_k) + f_y^{'}(x_k,x_k)(y-y_k) = 0.$$

Звідси виразимо x (позначимо цей результат через $\tilde{x}$):

$$\tilde{x} = x_k - \frac{1}{f_x^{'}(x_k,y_k)}\Big[f(x_k,y_k) + f_y^{'}(x_k,y_k)(y-y_k) \Big]. \tag{7.21}$$

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 вважатимемо лише проміжним наближенням (тобто не x_{k+1}), оскільки воно не враховує друге рівняння системи (7.20).

Підставивши в $g(x,y)$ замість x змінну $\tilde{x} = \tilde{x}(y)$, прийдемо до деякої функції $G(y) := g(,y)$ лише однієї змінної y. Це дозволяє лінеаризувати друге рівняння системи (7.20) формулою Тейлора для функції однієї змінної:

$$g(x,y) \approx G(y_k) + G'(y_k)(y - y_k) = 0. \qquad (7.22)$$

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

$$G'(y_k) = -g'_x(\tilde{x}_k, y_k) * \frac{f'_y(x_k, y_k)}{f'_x(x_k, y_k)} + g'_y(\overline{x}_k, y_k).$$

При відомих значеннях $G(y_k) = g(\tilde{x}_k, y_k)$ і $G'(y_k)$ можна розв'язати лінійне рівняння (7.22) відносно y (запишемо отримані значення як

$$h \le \sqrt[4]{\frac{180\varepsilon}{(b-a)M_4}}, \quad M_4 = \max_{x\in[0;1]} \left| f^{(4)}(x) \right| = \max_{x\in[0;1]} \left| \frac{24}{(1+x)^5} \right| = 24)$$

$$y_{k+1} = y_k - \frac{G(y_k)}{G'(y_k)} =$$

$$= y_k - \frac{g(\tilde{x}_k, y_k) f'_x(x_k, y_k)}{f'_x(x_k, y_k) g'(\tilde{x}_k, y_k) - f'_y(x_k, y_k) g'_x(\tilde{x}_k, y_k)}.$$

Замінюючи в (7.21) змінну y знайденим значенням y_{k+1}, приходимо до

$$x_{k+1} = \tilde{x}(y_{k+1}) = x_k - \frac{1}{f'_x(x_k, y_k)}\Big[f(x_k, y_k) + f'_y(x_k, y_k)(y_{k+1} - y_k) \Big].$$

Таким чином, реалізація методу Брауна розв'язування двовимірних нелінійних систем вигляду (7.20) зводиться до такої послідовності дій.

При вибраних початкових значеннях x_0, y_0 подальше наближення за методом Брауна знаходиться при $k = 0,1,2,...$ із сукупності формул:

$$\tilde{x}_k = x_k - \frac{f(x_k, y_k)}{f'_x(x_k, y_k)},$$

$$q_k = \frac{g(\tilde{x}_k, y_k) f'_x(x_k, y_k)}{f'_x(x_k, y_k) g'(\tilde{x}_k, y_k) - f'_y(x_k, y_k) g'_x(\tilde{x}_k, y_k)}, \qquad (7.23)$$

$$p_k = \frac{f(x_k, y_k) - q_k f'_o(x_k, y_k)}{f'_x(x_k, y_k)},$$

$$x_{k+1} = x_k - p_k, \quad y_{k+1} = y_k - p_k,$$

розрахунок за якими має виконуватись у тій послідовності, в якій вони записані.

Обчислення в методі Брауна закінчують, коли виконається нерівність $\max\{|p_{k-1}|, |q_{k-1}|\} < \varepsilon$, (із результатом $(x^*, y^*) := (x_k, y_k)$). Під час обчислень слід контролювати величину знаменників розрахункових формул. Зазначимо, що функції f і g в цьому методі нерівноправні і зміна їх ролей може змінити ситуацію зі збіжністю.

Указуючи на наявність квадратичної збіжності методу Брауна, зазначають, що розраховувати на його велику ефективність порівняно 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

Щоб наблизитися до розуміння ідей, які лежать в основі пропонованого методу, повернемося спочатку до одновимірного випадку.

У процесі побудови методів Ньютона і січних при розв'язанні нелінійного скалярного рівняння

$$f(x, y) = 0 \tag{7.24}$$

функція $f(x)$ в околі поточної точки x_k замінюється лінійною функцією (афінною моделлю)

$$\phi_k(a_k, x) \coloneqq f(x_k) + a_k(x - x_k). \tag{7.25}$$

Прирівнювання до нуля співвідношення (7.25), 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 кореня рівняння (7.24).

Якщо зажадати, щоб афінна модель $\phi_k(a_k, x)$, яка замінює функцію $f(x)$ поблизу точки x_k, мала в цій точці однакову з нею похідну, слід диференціюванням (7.25) отримати значення коефіцієнта

$$a_k = f'(x_k),$$

підстановка якого в (7.26) приводить до відомого методу Ньютона. Якщо ж урахувати, що разом із рівністю $\phi_k(a_k, x) = f(x_k)$ повинен мати місце збіг функцій $f(x)$ і в попередній до x_k точці x_{k-1}, то з рівності

$$\phi_k(a_k, x_{k-1}) = f(x_{k-1}),$$

або відповідно до (7.25)

$$f(x_k) + a_k(x_{k-1} - x_k) = f(x_{k-1}), \tag{7.27}$$

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), переписану 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*-вимірний випадок і лежить в основі виведення методу Бройдена. Опишемо це виведення.

У *n*-вимірному векторному просторі R_n співвідношення січних подається рівністю

$$B_k(x^{(k-1)} - x^{(k)}) = F(x^{(k-1)}) - F(x^{(k)}), \tag{7.28}$$

де $x^{(k-1)}, x^{(k)}$ – відомі *n*-вимірні вектори; $F : R_n \to R_n$ – дане нелінійне відображення; B_k – деяка матриця лінійного перетворення в R_n. Із позначеннями

$$s^{(k)} := x^{(k)} - x^{(k-1)}, \quad y^{(k)} := F(x^{(k)}) - F(x^{(k-1)}) \tag{7.29}$$

співвідношення січних в R_n записується коротше

$$B_k s^{(k)} = y^{(k)}. \tag{7.30}$$

Аналогічно одновимірному випадку, а саме, формулі (7.26), шукатимемо наближення до розв'язання векторного рівняння (7.2) за формулою

$$x^{(k+1)} = x^{(k)} - B_k^{-1} F(x^{(k)}). \tag{7.31}$$

Бажаючи, щоб ця формула узагальнювала метод січних, обернену *nxn*-матрицю B_k в ній потрібно підібрати так, щоб вона задовольняла співвідношення 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, які перетворюють заданий *n*-вимірний вектор $S^{(k)}$ в інший заданий вектор $y^{(k)}$ (звідси – розуміння того, що можуть виникати різні узагальнення одновимірного методу січних).

При формуванні матриці B_k міркуватимемо таким чином.

Переходячи від наявної в точці $x^{(k-1)}$ афінної моделі функції $F(x)$

$$\Phi_{k-1} := F(x^{(k-1)}) + B_{k-1}(x - x^{(k-1)}) \tag{7.32}$$

до такої самої 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 від рівності (7.33) визначальну рівність Φ_{k-1} (7.32) і перетворимо результат, залучаючи співвідношення січних (7.28):

$$\begin{aligned}\Phi_k - \Phi_{k-1} &:= F(x^{(k)}) - F(x^{(k-1)}) + B_k(x - x^{(k)}) - B_{k-1}(x - x^{(k)}) = \\ &= B_k(x^{(k)} - x^{(k-1)}) - B_k x^{(k)} + B_{k-1}x^{(k-1)} + (B_k - B_{k-1})x = \\ &= (B_k - B_{k-1})(x - x^{(k-1)}).\end{aligned}$$

Представимо вектор $(x - x^{(k-1)})$ у вигляді лінійної комбінації фіксованого вектора $S^{(k)}$, визначеного в (7.29), і деякого вектора *t*, ортогонального до нього:

$$x - x^{(k-1)} = \alpha s^{(k)} + t, \qquad \alpha \in R_1,\ t \in R_n : (t, s^{(k)}) = 0.$$

Підстановкою цього подання вектора $(x - x^{(k-1)})$ в різницю $\Phi_k - \Phi_{k-1}$ отримуємо інший її вигляд

$$\Phi_k - \Phi_{k-1} = \alpha(B_k - B_{k-1})s^{(k)} + (B_k - B_{k-1})t\,. \qquad (7.34)$$

Аналізуючи вираз (7.34), помічаємо, що перший доданок в ньому не може бути змінений, оскільки

$$(B_k - B_{k-1})s^{(k)} = B_k s^{(k)} - B_{k-1}s^{(k)} = y^{(k)} - B_{k-1}s^{(k)},$$

тобто ця різниця є фіксований вектор при фіксованому *k*. Тому мінімальній зміні афінної моделі Φ_{k-1} відповідатиме випадок, коли другий доданок у (7.34) буде нуль-вектором при будь-яких векторах *t*, ортогональних векторам $s^{(k)}$, тобто B_k слід знаходити з умови

$$(B_k - B_{k-1})t = 0, \forall t : (t, s^{(k)}) = 0\,. \qquad (7.35)$$

Безпосередньою перевіркою переконуємося, що умова (7.35) буде виконана, якщо матричну поправку $B_k - B_{k-1}$ узяти 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 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, при якій виконувалося б співвідношення січних (7.30) в новій точці і при цьому зміни в афінній моделі (7.32) були б мінімальними.

Сукупність формул (7.31), (7.36) разом із позначеннями (7.29) називають *методом січних Бройдена* або просто методом Бройдена розв'язання систем нелінійних числових рівнянь.

У методах січних звичайним є задання двох початкових векторів ($x^{(0)}$ і $x^{(1)}$), проте для методу Бройдена характерний інший початок ітераційного процесу. Тут потрібно задати **один** початковий вектор $x^{(0)}$, початкову матрицю B_0 і далі в циклі за змінною $k = 0,1,2,...$ послідовно виконувати такі операції.

1. Розв'язати лінійну систему

$$B_k s^{(k+1)} = -F(x^{(k)}) \tag{7.37}$$

відносно вектора $s^{(k+1)}$.

2. Знайти вектори $x^{(k+1)}$ і $y^{(k+1)}$:

$$x^{(k+1)} = x^{(k)} + s^{(k+1)},\ y^{(k+1)} = F(x^{(k+1)}) - F(x^{(k)}). \tag{7.38}$$

3. Виконати перевірку на зупинку (наприклад, за допомогою перевірки на мализну величин $\left\|s^{(k+1)}\right\|$ і $\left\|y^{(k+1)}\right\|$, і якщо потрібна точність не досягнута, обчислити нову матрицю B_k за формулою перерахунку (див. формулу (7.36))

$$B_{k+1} = B_k + \frac{y^{(k+1)} - B_k s^{(k+1)} (s^{(k+1)})^T}{(s^{(k+1)})^T s^{(k+1)}}. \tag{7.39}$$

За матрицю B_0, якої потребує рівність (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 бути корисним при виродженні матриць). Але водночас, видно, що за певних вимог до матриць Якобі $F'(x^{(1)})$ матриці $F'(x^{(2)})$ мають "властивість обмеженого погіршення". Це означає, що якщо і відбувається збільшення $\left\|B_k - F'(x^{(k)})\right\|$ зі збільшенням номера ітерації k, то досить повільне. За допомогою цієї властивості доводяться

твердження про лінійну збіжність ($x^{(k)}$) до x^* при достатній близькості $x^{(0)}$ до 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 за методом Бройдена*.

Як і у випадках застосування інших методів розв'язування нелінійних систем, перевірка можливості здійснення будь-яких умов збіжності ітераційного процесу Бройдена досить важка.

Формулі перерахунку (7.39) в ітераційному процесі надамо більш простого вигляду.

Оскільки, з огляду на (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

$$(s^{(k+1)})^T s^{(k+1)} = (s^{(k+1)}, s^{(k+1)}) = \left\| s^{(k+1)} \right\|_2^2 = \left\| x^{(k+1)} - x^{(k)} \right\|_2^2,$$

то з формули (7.39) отримуємо формально еквівалентну до неї формулу перерахунку

$$B_{k+1} = B_k + \frac{F(x^{(k+1)})(x^{(k+1)} - x^{(k)})^T}{\left\| x^{(k+1)} - x^{(k)} \right\|_2^2}, \qquad (7.40)$$

яку можна використовувати замість (7.39) разом із формулою (7.31) або з (7.37), (7.38), без обчислення вектора $y^{(k+1)}$. Таке перетворення ітераційного процесу Бройдена трохи скорочує обсяг 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, у випадках, коли для здобуття розв'язку з потрібною точністю потрібно багато ітерацій за методом Бройдена, тобто коли й застосовувати його не варто.

Реалізація цього методу також пропонується як самостійна вправа.

7.3. Про розв'язування нелінійних систем методами спуску

Загальний недолік усіх розглянутих вище методів розв'язування систем нелінійних рівнянь – це суто локальний характер збіжності, що ускладнює використання цих систем у випадках, коли є проблеми з вибором добрих початкових наближень. Допомогти тут можуть числові методи оптимізації – підрозділи обчислювальної математики, які зазвичай виділяються 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 з двох рівнянь із двома невідомими (7.41).

Із функцій f і 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 невід'ємна, то знайдеться точка x^*, y^* така, що

$$\Phi(x,y) \geq \Phi(x^*,y^*) \geq 0 \forall (x,y) \in R_2 ,$$

тобто $(x^*, y^*) = \arg\min \Phi(x,y)$. Отже, якщо тим або іншим способом вдається отримати точку x^*, y^*, що мінімізує функцію $\Phi(x,y)$, і якщо при цьому виявиться, що $\min \Phi(x,y) = \Phi(x^*,y^*) = 0$, 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 $\Phi(x,y)$ – зазвичай отримують з рекурентної формули

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + a_k \begin{pmatrix} p_k \\ q_k \end{pmatrix}, k = 0,1,2,..., \qquad (7.42)$$

де $(p_k, q_k)^T$ – вектор, що визначає напрямок мінімізації, a_k – скалярна величина, що характеризує розмір кроку мінімізації (крокoвий множник). Ураховуючи геометричний сенс (рис. 7.2) завдання мінімізації функції двох змінних $\Phi(x, y)$ – "спуск на дно" поверхні $z = \Phi(x, y)$, ітераційний метод (7.42) можна назвати ***методом спуску***, якщо вектор $(p_k, q_k)^T$ при кожному k є напрямком 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 числового методу вигляду (7.42) мінімізації функції $\Phi(x, y)$ слід відповісти на два головні питання: як вибирати напрямок спуску a_k і як регулювати довжину кроку у вибраному напрямку за допомогою скалярного параметра – крокового множника a_k. Наведемо найбільш прості міркування з цього приводу.

При виборі напрямку спуску природним є вибір такого напрямку, в якому функція, що мінімізується, спадає найшвидше. Як відомо з математичного аналізу функцій декількох змінних, напрямок найбільшого зростання функції в даній точці показує її градієнт у цій точці. Тому приймемо за напрямок спуску вектор

$$\begin{pmatrix} p_k \\ q_k \end{pmatrix} := -\operatorname{grad}\Phi(x_k, y_k) = -\begin{pmatrix} \Phi'_x(x_k, y_k) \\ \Phi'_y(x_k, y_k) \end{pmatrix}.$$

Отримаємо антиградієнт функції $\Phi(x, y)$. Таким чином, із сім'ї 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 напрямку антиградієнта – це такий крок, при якому значення $\widetilde{y}_k$ – найменше серед усіх інших значень $\Phi(x,y)$ у цьому фіксованому напрямку, тобто, коли точка (x_{k+1}, y_{k+1}) є точкою умовного мінімуму. Отже, можна розраховувати на найбільш швидку збіжність методу (7.43), якщо вибрати в ньому такий кроковий множник, який називається *вичерпним спуском*. Такий вибір кроку разом із формулою (7.43) визначає *метод найшвидшого спуску*.

Геометричну інтерпретацію цього методу добре видно з рисунків 7.2 і 7.3. Характерні дев'яностоградусні злами траєкторії найшвидшого спуску, що пояснюється вичерпністю спуску та властивістю градієнта (а значить, й антиградієнта) – перпендикулярність дотичній до лінії рівня у відповідній точці.

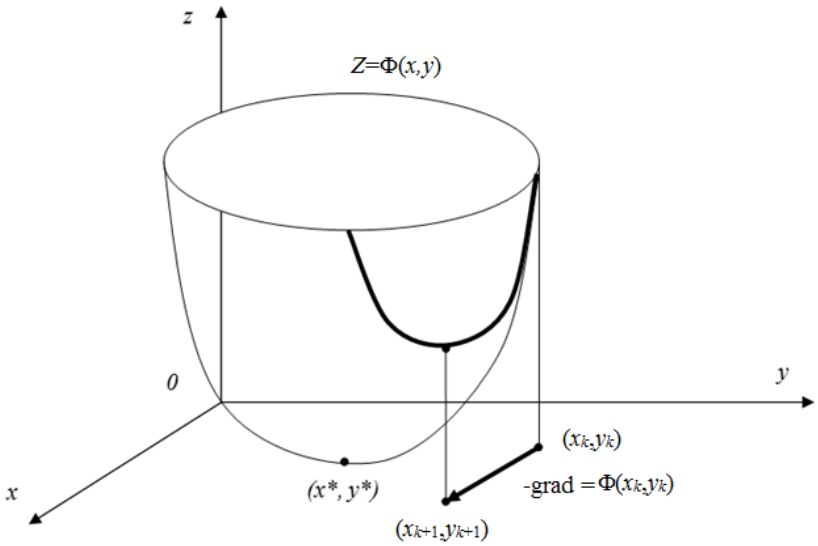


Рис. 7.2. Просторова інтерпретація методу найшвидшого спуску для функції (7.41)

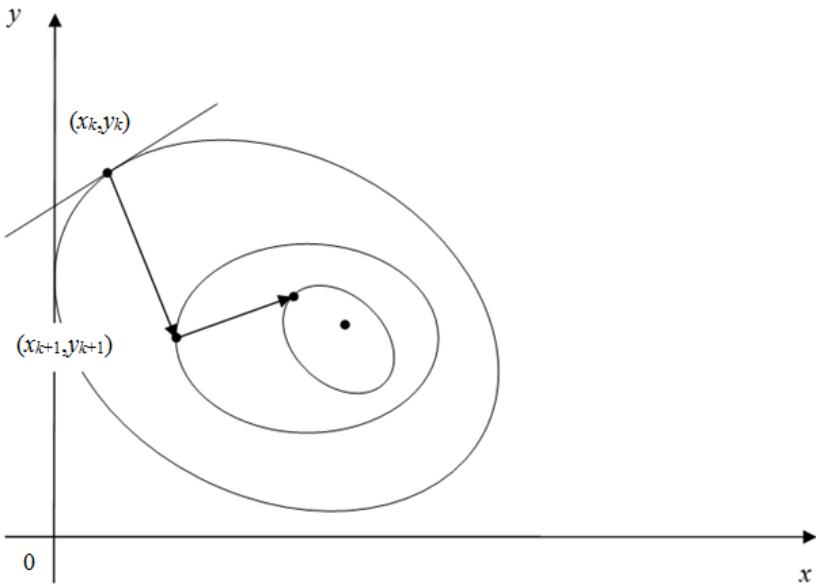


Рис. 7.3. Траєкторія найшвидшого спуску
для функції (7.41)

Найбільш типовою є ситуація, коли знайти (аналітичними методами) точне оптимальне значення a_k не вдається. Отже, доводиться використовувати які-небудь числові методи одновимірної мінімізації і знаходити a_k лише приблизно.

Незважаючи на те, що завдання знаходження мінімуму функції однієї змінної $\phi_k(\alpha) = \Phi(x_k - \alpha\Phi'_x(x_k, y_k), y_k - \alpha\Phi'_x(x_k, y_k))$ набагато простіше, ніж виконуване завдання, використання тих або інших числових методів знаходження значень $\alpha_k = \arg\min \phi_k(\alpha)$ із тією або іншою точністю вимагає обчислення декількох значень функції, що мінімізується. Оскільки це потрібно робити на кожному ітераційному кроці, то при великій кількості кроків реалізація методу найшвидшого спуску в чистому вигляді є досить високовитратною. Існують ефективні схеми наближеного обчислення квазіоптимальних a_k, в яких ураховується специфіка функцій, що мінімізуються (на зразок сум квадратів функцій).

Часто успішною є така стратегія градієнтного методу, при якій кроковий множник a_k в (7.43) береться або відразу досить малим сталим, або передбачається його зменшення, наприклад, діленням навпіл для виконання умови релаксації на черговому кроці. Хоча кожен окремий крок градієнтного методу при цьому, узагалі кажучи, далекий від оптимального, такий процес за кількістю обчислень функції може виявитися ефективнішим, ніж метод найшвидшого спуску.

Головна перевага градієнтних методів розв'язування 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 розв'язком початкової нелінійної системи.

Головна вада – повільна збіжність. Доведено, що збіжність таких методів – лише лінійна, причому, якщо для багатьох методів, таких як метод Ньютона, характерне прискорення збіжності при наближенні до розв'язку, то тут має місце швидше зворотне. Тому є сенс у побудові гібридних алгоритмів, які починали б пошук шуканої точки, – вирішення даної нелінійної системи – градієнтним методом, що глобально сходиться, а потім уточнювали б якимсь швидкозбіжним методом, наприклад, методом Ньютона (зрозуміло, якщо ці функції мають потрібні властивості).

Розроблено ряд методів розв'язування екстремальних задач, які поєднують у собі низьку вимогливість до вибору початкової точки й 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 виконуваному завданні слід відмовитися від використання похідних або їх апроксимацій і вдатися до так званих методів прямого пошуку (циклічного покоординатного спуску, Хука і Дживса, Розенброка та ін.).

Контрольні запитання

1. Які є модифікації методу Ньютона?
2. У чому суть різницевого методу Ньютона?
3. У чому полягає метод простих січних?
4. Назвіть методи спуску.

Розділ 8
ЧИСЛОВЕ ІНТЕГРУВАННЯ ФУНКЦІЙ

Відомо, що для переважної більшості функцій не вдається обчислити первісні функції, унаслідок чого доводиться вдаватися до методів наближеного і числового інтегрування функцій.

При числовому 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 поліномом. Інтеграл від неї добре обчислюється і порівняно легко оцінюється похибка.

Нехай на відрізку $x \in [a,b]$ задано неперервну функцію $y = f(x)$, і потрібно на цьому відрізку обчислити визначений інтеграл

$$I = \int_a^b f(x)dx . \qquad (8.1)$$

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:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(\xi_i)\Delta x_i,\ \xi_i \in [x_i, x_{i-1}],\ \Delta x_i = x_i, x_{i-1} . \qquad (8.2)$$

Якщо в цьому визначенні зняти знак границі і покласти $\Delta x_i = h_i,\ i = \overline{1,n}$, то з'явиться похибка $R_{\text{пр}}$ (за ξ_i можна прийняти лівий або правий кінець відрізка Δx_i), тобто

$$\int_a^b f(x)dx = \sum_{i=1}^{n} f(x_{i-1})h_i + R_{\text{пр}}$$

або

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(x_i)h_i + R_{\text{пр}}. \qquad (8.3)$$

Одержані формули (8.3) називаються, відповідно, *формулами лівих і правих прямокутників* числового інтегрування.

Розглянемо похибку R_i формули лівих прямокутників на одному кроці $[x_i,\ x_{i-1}]$ числового інтегрування.

Для цього припустимо, що первісна функція $F(x)$ для підінтегральної функції $f(x)$, яка існує (оскільки $f(x)$ – неперервна на відрізку $x \in [a,b]$), є неперервно диференційована. Тоді, розвиваючи $F(x)$ в околі вузла x_{i-1} в ряд Тейлора до другої похідної включно і використовуючи рівність $F'(x) = f(x) = y(x)$, маємо

$$R_i = \int_{x_{i-1}}^{x_i} f(x)dx - y_{i-1}h = \left[F(x_i) - F(x_{i-1}) - y_{i-1}h\right] =$$

$$= \left[F'(x_{i-1})h - F'(\xi)\frac{h^2}{2}\right] - y_{i-1}h =$$

$$= \left[-y_{i-1}h + y'(\xi)\frac{h^2}{2}\right] - y_{i-1}h = y'(\xi)\frac{h^2}{2},\ \xi \in (x_{i-1}, x_i). \qquad (8.4)$$

На всьому відрізку $[a,b]$ цю похибку необхідно підсумувати n разів ($b–a=nh$), отримаємо

$$R_{\text{пр}} = R_i n = y'(\xi)\frac{(b-a)h}{2},\ \xi \in (a,b). \qquad (8.5)$$

Оскільки місцеположення точки ξ в інтервалі *(a,b)* не відоме, то згідно з виразом для похибки (8.5) можна виписати верхню оцінку абсолютної похибки методу прямокутників і при заданій точності ε методу виписати нерівність

$$\left|R_{ï\,ð}\right| \le \frac{(b-a)h}{2} M_1 \le \varepsilon, \quad M_1 = \max_{x\in[a,b]} \left|f'(x)\right|, \tag{8.6}$$

яку можна використовувати для верхньої оцінки кроку *h* числового інтегрування методом прямокутників:

$$h \le \frac{2\varepsilon}{(b-a)M_1}, \quad M_1 = \max_{x\in[a,b]} \left|f'(x)\right|. \tag{8.7}$$

Із виразу для похибки випливає, що на кожному відрізку $[x_i, x_{i-1}]$ формула прямокутників має похибку, пропорційну h^2, а на всьому відрізку $[a,b]$ – пропорційну кроку числового інтегрування h. Тому кажуть, що *метод прямокутників є методом 1-го порядку точності (головний член похибки пропорційний кроку в першому степені)*.

Алгоритм (рис. 8.1) запропонованого методу складається з таких основних кроків.

1. Весь відрізок $[a,\ b]$ ділиться на *n* рівних частин із кроком $h = (b-a)/n$.

2. Знаходиться значення *yi* підінтегральної функції $f(x)$ у кожній частині, тобто $y_i = f(x_i),\ i = 0, n$.

3. У кожній частині підінтегральна функція апроксимується інтерполяційним поліномом степеня $n = 0$, тобто прямою, що паралельна осі *Ox*. У результаті вся підінтегральна функція на ділянці $[a,b]$ апроксимується ламаною лінією.

4. Для кожної частини визначається площа *Si* часткового прямокутника.

5. Сумуються всі площі. Наближене значення інтеграла *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;
- $S1$ – значення інтеграла на попередньому кроці;
- S – значення інтеграла на поточному кроці.

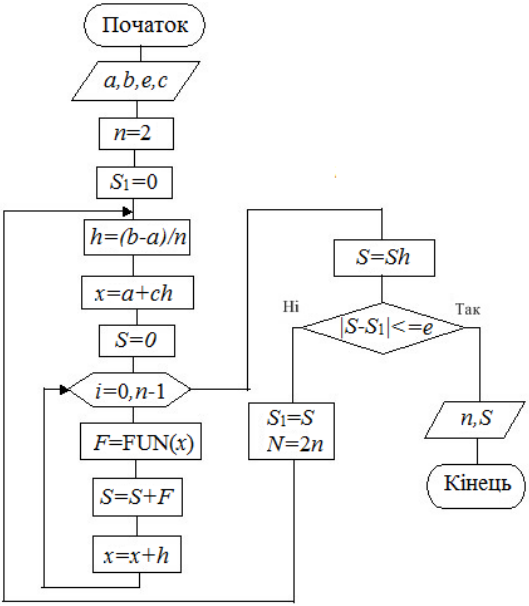


Рис. 8.1. Блок-схема алгоритму методу прямокутників

Реалізація запропонованого алгоритму подана в лістингу 8.1.

Лістинг 8.1. Реалізація алгоритму методу прямокутників

```
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)
```

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 поліномом 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); L_1 – інтерполяційний поліном Лагранжа першого степеня, проведений через два вузли інтерполяції x_{i-1} і x_i:

$$L_1(x) = y_{i-1}\frac{x - x_i}{x_{i-1} - x_i} + y_{i-1}\frac{x - x_i}{x_i - x_{i-1}}.$$

Нехай $x_i - x_{i-1} = h = \text{const}$, де $i = \overline{1,n}$.

Позначимо $\dfrac{x_i - x_{i-1}}{h} = t$, тоді $\dfrac{x - x_i}{h} = \dfrac{\left(x - x_{i-1}\right) - \left(x_i - x_{i-1}\right)}{h} =$ $= t - 1$, $dx = hdt$ і поліном Лагранжа набуде вигляду $L_1(x) = L_1(x_{i-1} + ht) = -y_{i-1}(t-1) + y_i t$. При $x = x_i$ верхня границя $t = 1$, при $x - x_{i-1}$ нижня границя $t = 0$.

Тепер інтеграл $\int\limits_{x_{i-1}}^{x_i} L_1(x)dx$ від полінома $L_1(x)$ можна подати у вигляді

$$\int\limits_{x_{i-1}}^{x_i} f(x)dx \approx \int\limits_{x_{i-1}}^{x_i} L_1(x)dx = h\int\limits_0^1 \left[-y_{i-1}(t-1) + y_i t\right]dt = \\ = h\left[-y_{i-1}\left(\frac{t^2}{2} - t\right) + y_i \frac{t^2}{2}\right]_0^1 = \frac{h}{2}(y_{i-1} + y_i). \tag{8.9}$$

Цей вираз називають *формулою трапецій* числового інтегрування (рис. 8.2) на відрізку $x \in \left[x_{i-1}, x_i\right]$.

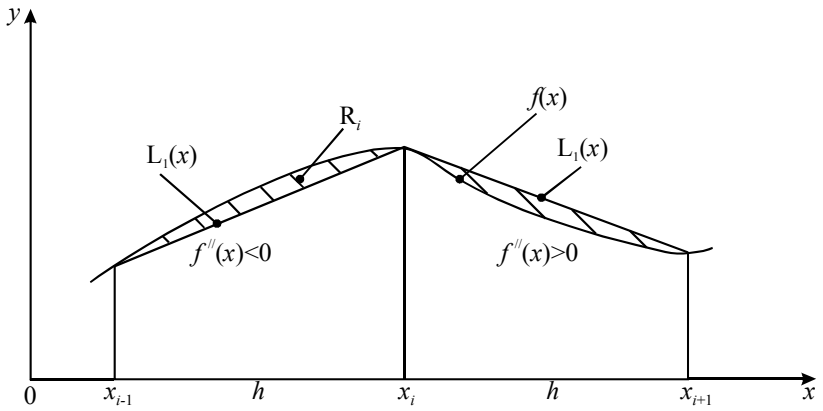


Рис. 8.2. Метод трапецій

Для всього відрізка $[a,b]$ необхідно додати цей вираз *n* разів.

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(x_0)+f(x_n)+2\sum_{i=1}^{n-1} f(x_i)) = \\ = \frac{h}{2}(y_0+y_n+2\sum_{i=1}^{n-1} y_i). \tag{8.10}$$

Отриманий вираз (8.10) називають *формулою трапецій* числового інтегрування для всього відрізка $[a,b]$.

У разі змінного кроку метод трапецій використовується у вигляді

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{f(x_{i-1})+f(x_i)}{2} h_i, \quad h_i = x_i - x_{i-1}. \tag{8.11}$$

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 $\overline{x}_i = \frac{x_i + x_{i+1}}{2}$ серединну точку відрізка $[x_{i-1},x_i]$ і розвинемо *f*(*x*) по степенях $(x-\overline{x}_i)$ за формулою Тейлора, припускаючи, що вона має п'ять неперервних похідних. Отримаємо

$$\int_0^1 \frac{dx}{1+x} \approx \frac{h}{2}\left(y_0+y_n+2\sum_{i=1}^{n-1} y_i\right) = \\ = \frac{0,2}{2}\left[1,0+0,5+2\left(0,83+0,71+0,63+0,56\right)\right] = 0,696.$$

Інтегруючи $f(x)$ на відрізку $[x_{i-1}, x_i]$, маємо

$$\int_{x_i}^{x_{i+1}} f(x)dx = \int_{x_i}^{x_{i+1}} (f(\overline{x}_i) + (x-\overline{x}_i)f'(\overline{x}_i) + \frac{1}{2}(x-\overline{x}_i)^2 f''(\overline{x}_i) +$$

$$+\frac{1}{6}(x-\overline{x}_i)^3 f'''(\overline{x}_i) + \frac{1}{24}(x-\overline{x}_i)^4 f^{(4)}(\overline{x}_i)+)dx =$$

$$= h_i f(\overline{x}_i) + \frac{1}{24}h^3 f'''(\overline{x}_i) + \frac{1}{1920}h_i^5 + \ldots$$

Інтеграли парних степенів перетворюються на нуль.

Це означає, що коли hi мале, то похибка інтегрування на відрізку *за формулою прямокутників* має порядок $\frac{1}{24}h^3 f'''(\overline{x}_i)$.

Щоб оцінити порядок похибки для формули трапецій, знову використаємо розвинення за формулою Тейлора. Підставляючи в нього значення $x = x$і та $x = xi+1$, одержимо

$$f(x_{i+1}) = f(\overline{x}_i) + \frac{1}{2}h_i f'(\overline{x}_i) + \frac{1}{8}h_i^2 f''(\overline{x}_i) +$$

$$+\frac{1}{48}h_i^3 f'''(\overline{x}_i) + \frac{1}{384}h_i^4 f^{(4)}(\overline{x}_i) + \ldots$$

Звідси маємо

$$\frac{f(x_i)+f(x_{i+1})}{2} = f(\overline{x}_i) + \frac{1}{8}h_i^2 f''(\overline{x}_i) + \frac{1}{384}h_i^4 f^{(4)}(\overline{x}_i) + \ldots$$

Об'єднуючи це з розкладом інтеграла, одержуємо

$$\int_{x_i}^{x_{i+1}} f(x)dx =$$

$$= h_i \frac{f(x_i)+f(x_{i+1})}{2} - \frac{1}{12}h^3 f''(\overline{x}_i) - \frac{1}{480}h_i^5 f^{(4)}(\overline{x}_i) + \ldots \qquad (8.13)$$

Таким чином, при малих hi похибка інтегрування на відрізку *за формулою трапецій* має порядок $-\frac{1}{12}h^3 f''(\overline{x}_i)$. Порівнюючи це з оцінкою похибки для формули прямокутників, зауважимо, що як не дивно, але формула прямокутників приблизно вдвічі точніша за формулу трапецій.

На всьому відрізку похибку необхідно збільшити в *n* разів:

$$R_{\text{тр}} \approx -\frac{nh^3}{12} y''(\xi) = -\frac{(nh)h^2}{12} y''(\xi) = -\frac{b-a}{12} h^2 y''(\xi),$$

$$\xi \in (a,b). \tag{8.14}$$

Вираз для оцінки похибки зазвичай записується таким чином:

$$\left|R_{\text{тр}}\right| \le \max_{x\in[a,b]} \left|f''(x)\right| \frac{b-a}{12} h^2. \tag{8.15}$$

Звідки, задаючи точність числового інтегрування, можна записати наступну нерівність, використовувану для визначення кроку *h* числового інтегрування:

$$h \le \sqrt{\frac{12\varepsilon}{(b-a)M_2}}, \quad M_2 = \max_{x\in[a,b]} \left|f''(x)\right|. \tag{8.16}$$

Числове інтегрування за методом трапецій у разі заданої точності ε можна провести так:

1) за формулою (8.16) визначається крок числового інтегрування *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-схему алгоритму методу трапецій наведено на рис. 8.3.

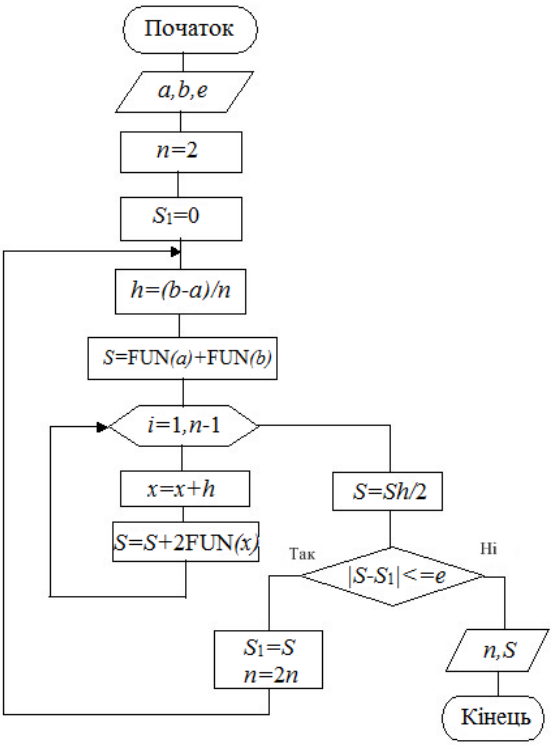


Рис. 8.3. Блок-схема алгоритму методу трапецій

Одну з можливих реалізацій методу трапецій у вигляді модуля мовою *Python* наведено в лістингу 8.2.

Лістинг 8.2. Реалізація методу трапецій

```
# -*- 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 самий 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
```

Розглянемо застосування розробленого модуля для виконання завдання (приклад 8.1).

Приклад 8.1. Визначити інтеграл $\int_0^{\pi} \sqrt{x}\cos x dx$ із точністю до 6-го знака. Визначити, скільки потрібно ділянок розбиття для досягнення такої точності. У лістингу 8.3 наведено код програми, яка розв'язує поставлену завдачу.

Лістинг 8.3. Приклад застосування модуля

```
# -*- 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("\nНатиснути enter для виходу з програми")
```

Результат роботи програми представлено нижче:

>>>

Інтеграл = -0.894831664853

nДілянок = 32768

Натиснути enter для виходу з програми

Як бачимо, потрібно 32768 розбиттів, щоб досягнути поставленої точності.

8.3. Метод Симпсона

Розіб'ємо відрізок $[a,b]$ на m пар відрізків $b-a=nh=2mh$,

$\frac{x_1}{x_2}$ і через кожні три вузли проведемо інтерполяційний поліном Лагранжа (рис. 8.4).

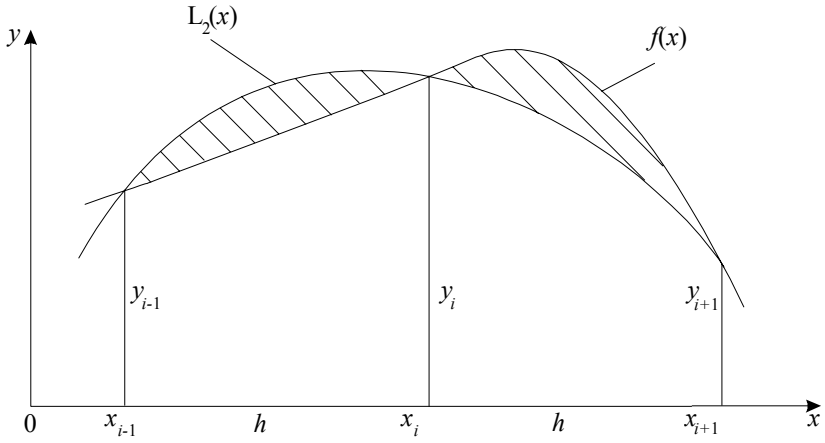


Рис. 8.4. Метод Симпсона

Тоді

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx = \int_{x_{i-1}}^{x_{i+1}} L_2(x)dx + R_i, \qquad (8.17)$$

де

$$L_2(x) = y_{i-1}\frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} + y_i\frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} +$$
$$+ y_{i+1}\frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)}.$$

Виконаємо заміну $\frac{x-x_{i-1}}{h} = t,\ \ dx = hdt$ і тоді маємо

$$\frac{x-x_i}{h} = \frac{(x-x_{i-1})-(x_i-x_{i-1})}{h} = t-1;$$
$$\frac{x-x_{i+1}}{h} = \frac{(x-x_{i-1})-(x_{i+1}-x_{i-1})}{h} = t-2. \qquad (8.18)$$

Доданки в $L_2(x)$ набудуть вигляду

$$y_{i-1}\frac{(x-x_i)(x-x_{i+1})}{h\cdot 2h} = (t-1)(t-2)\frac{y_{i-1}}{2};$$
$$y_i\frac{(x-x_{i-1})(x-x_{i+1})}{-h\cdot h} = -t(t-2)y_i;$$
$$y_{i+1}\frac{(x-x_{i-1})(x-x_i)}{2h\cdot h} = \frac{t}{2}(t-1)y_{i+1}.$$

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

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx \approx \int_{x_{i-1}}^{x_{i+1}} L_2(x)dx = \frac{h}{3}(y_{i-1} + 4y_i + y_{i+1}). \qquad (8.19)$$

На всьому відрізку вираз необхідно скласти m разів, оскільки є m пар відрізків завдовжки h. Отримаємо *формулу Симпсона числового інтегрування*:

$$\int_a^b f(x)dx \approx \frac{h}{3}(y_0 + y_n + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i}). \qquad (8.20)$$

Похибка формули Симпсона на подвійному кроці пропорційна четвертій похідній функції і п'ятому степеню кроку:

$$R_i = \int_{x_{i-1}}^{x_{i+1}} f(x)dx - \frac{h}{3}(y_{i-1} + 4y_i + y_{i+1}) =$$

$$= -\frac{h^5}{90} f^{(4)}(\xi), \quad \xi \in (x_{i-1}, x_{i+1}). \qquad (8.21)$$

Це випливає з того, що формула Симпсона $S(f)$ є комбінацією формул прямокутників $R(f)$ і трапецій $T(f)$, а саме: $S(f) = \frac{2}{3}R(f) + \frac{1}{3}T(f)$, у 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:

$$R_C \approx -\frac{mh^5}{90} f^{(4)}(\xi) = -\frac{2mh^5}{180} f^{(4)}(\xi) =$$

$$= -\frac{nh \cdot h^4}{180} f^{(4)}(\xi) = -\frac{(b-a)h^4}{180} f^{(4)}(\xi),$$

$$\xi \in (a,b),$$

тобто у формулі Симпсона на всьому відрізку $[a,b]$ похибка пропорційна четвертому степеню кроку і, отже, метод Симпсона є методом 4-го 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:

$$|R_C| \le \frac{(b-a)h^4}{180} M_4, \quad M_4 = \max_{x\in[a,b]} \left| f^{(4)}(x) \right|,$$

звідки при заданій точності $\|A\|_2 =$ можна отримати

$$h \le \sqrt[4]{\frac{180\varepsilon}{(b-a)M_4}}, \quad M_4 = \max_{x\in[a,b]} \left| f^{(4)}(x) \right|. \qquad (8.22)$$

Блок-схему алгоритму методу Симпсона наведено на рис. 8.5.

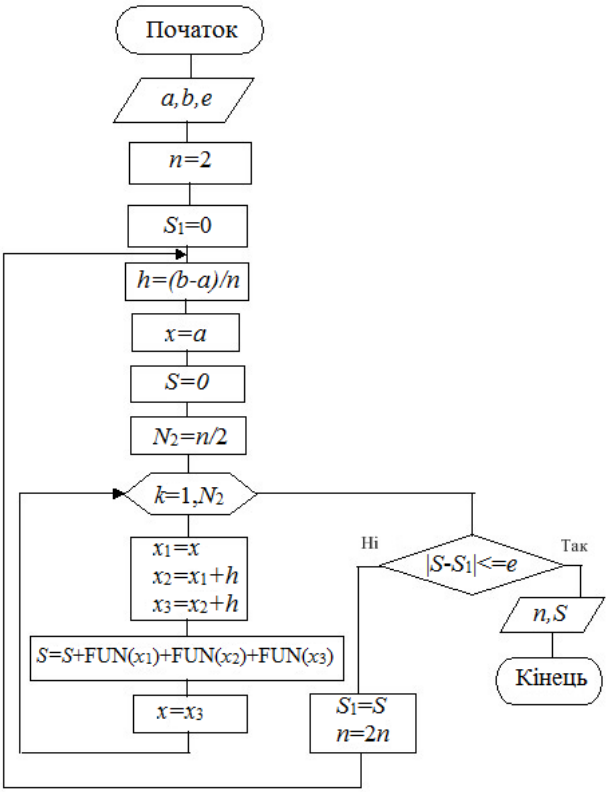


Рис. 8.5. Блок-схема алгоритму методу Симпсона

Розглянемо ще один приклад.

Приклад 8.2. Методом трапецій із точністю $\varepsilon = 10^{-2}$ і Симпсона 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'язання

1. *Метод трапецій.* Виходячи із заданої точності $\varepsilon = 10^{-2}$, обчислимо крок числового інтегрування за формулою (8.16)

$$h \le \sqrt{\frac{12\varepsilon}{(b-a)M_2}}, \quad M_2 = \max_{x\in[0;1]} \left|f''(x)\right| = \max_{x\in[0;1]} \left|\frac{2}{(1+x)^3}\right| = 2;$$

$$h \le \sqrt{\frac{12 \cdot 0,01}{(1-0)\cdot 2}} = \sqrt{6} \cdot 0,1 = 0,2449.$$

Необхідно вибрати такий крок, який задовольняє нерівність $h \le 0,2449$ і який на відрізку інтеграції $x \in [0;1]$ укладається ціле число разів. Приймаємо $h = 0,2$.

Для підінтегральної функції $f(x) = (1+x)^{-1}$ із 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 знака 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 числового інтегрування при n=5.

$$\int_0^1 \frac{dx}{1+x} \approx \frac{h}{2}\left(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i \right) =$$

$$= \frac{0,2}{2}\left[1,0 + 0,5 + 2\left(0,83 + 0,71 + 0,63 + 0,56\right)\right] = 0,696.$$

Порівнюючи це значення з точним, бачимо, що абсолютна похибка не перевищує заданої точності ε:

$$\left|0,69315 - 0,696\right| < 0,01.$$

Таким чином, за наближене значення визначеного інтеграла методом трапецій із точністю $\varepsilon = 0,01$ береться значення

$$\int_0^1 \frac{dx}{1+x} \approx 0,696.$$

2. *Метод Симпсона*. Виходячи із заданої точності обчислюється крок числового інтегрування для методу Симпсона за формулою (8.22):

$$h \le \sqrt[4]{\frac{180\varepsilon}{(b-a)M_4}},\ M_4 = \max_{x\in[0;1]}\left|f^{(4)}(x)\right| = \max_{x\in[0;1]}\left|\frac{24}{(1+x)^5}\right| = 24;$$

$$h \le \sqrt[4]{\frac{180\cdot 10^{-4}}{(1-0)\cdot 24}} = 10^{-1}\sqrt[4]{7,5} = 0,165.$$

Необхідно вибрати такий крок, щоб він задовольняв нерівність $h \le 0,165$ і щоб на відрізку інтегрування $[0;1]$ він укладався *парне число* разів. Приймаємо $h = 0,1$. Із цим кроком для пі-

дінтегральної функції $f(x)=(1+x)^{-1}$ формується сіткова функція з незалежною змінною x_i, що змінюється згідно з 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 знака 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

Далі використовується формула Симпсона (8.20) для числового інтегрування ($n=10$, $m=5$).

$$\int_0^1 \frac{dx}{1+x} = \frac{h}{3}\left(y_0 + y_n + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i} \right) =$$

$$= \frac{0{,}1}{3}\cdot\left[1{,}0 + 0{,}5 + 4\left(y_1 + y_3 + y_5 + y_7 + y_9\right) + \right.$$

$$\left. + 2\left(y_2 + y_4 + y_6 + y_8\right)\right] = \frac{0{,}1}{3}\left[1{,}5 + 4\left(\begin{array}{c} 0{,}9091 + 0{,}7692 + 0{,}6667 + \\ + 0{,}5882 + 0{,}5263 \end{array}\right) + \right.$$

$$\left. + 2\left(0{,}8323 + 0{,}7143 + 0{,}625 + 0{,}5556\right)\right] =$$

$$= \frac{0{,}1}{3}\left(1{,}5 + 4\cdot 3{,}4595 + 2\cdot 2{,}7281\right) =$$

$$= \frac{0{,}1}{3}\,20{,}7942 = 0{,}69314.$$

Порівняння цього значення з точним значенням інтеграла показує, що абсолютна похибка не перевищує заданої точності ε_1 :

$$\left|0,69315-0,69314\right|<0,0001\,.$$

Таким чином, за наближене значення визначеного інтеграла методом Симпсона з точністю $\varepsilon_1=0,0001$ береться таке:

$$\int_0^1 \frac{dx}{1+x}\approx 0,6931.$$

Контрольні запитання

1. Назвіть методи числового інтегрування?
2. У чому суть методу прямокутників?
3. За допомогою якого алгоритму інтерполяції виводиться основна формула методу Симпсона?

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 багато законів фізики формулюється саме у вигляді диференціальних рівнянь. По суті будь-яке завдання проектування, пов'язане з розрахунком потоків енергії або руху тіл, врешті-решт зводиться до розв'язування диференціальних рівнянь. На жаль, лише дуже небагато з них вдається виконати без допомоги обчислювальних машин. Тому числові методи розв'язування диференціальних рівнянь грають таку важливу роль в інженерних розрахунках.

Залежно від кількості незалежних змінних і типу похідних, що до них входять, диференціальні рівняння ділять на дві істотно різні категорії: *звичайні*, які містять одну незалежну змінну і похідні по ній, та *рівняння в частинних похідних*, що мають декілька незалежних змінних і похідних по них. У цьому розділі розглядаються методи розв'язування звичайних диференціальних рівнянь.

9.1. Задача Коші та крайова задача

Для розв'язання звичайного диференціального рівняння необхідно мати 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 більше значеннях незалежної змінної, то задача називається *крайовою.* У задачі Коші додаткові умови називають початковими, а в крайовій задачі – граничними. Часто в задачі Коші в ролі незалежної змінної виступає час. Прикладом є задача про вільні коливання тіла,

підвішеного на пружині. Рух такого тіла описується диференціальним рівнянням, в якому незалежною змінною є час t. Якщо додаткові умови задано у вигляді значень переміщення і швидкості при $t = 0$, то маємо задачу Коші. Для тієї самої механічної системи можна сформулювати і крайову задачу. У цьому випадку однією з умов є задання 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. У цьому разі граничні умови зазвичай задаються на обох кінцях стрижня. Хоча обидві вказані задачі розглядаються в одному розділі, при їх розв'язуванні застосовуються істотно різні методи й обчислювальні алгоритми. Виклад почнемо із задачі Коші.

Задача Коші. Задачу Коші можна сформулювати таким чином. Нехай задано диференціальне рівняння з початковою умовою

$$\begin{aligned} y' &= f(x, y), \\ y(x_0) &= y_0. \end{aligned} \tag{9.1}$$

Треба знайти функцію $y(x)$, що задовольняє як указане рівняння, так і початкову умову. Зазвичай числовий 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 нахилом кривої, обчисленим за допомогою диференціального рівняння. Таким чином, графік числового рішення є послідовність коротких прямолінійних відрізків, якими апроксимується дійсна крива $y = f(x)$. Сам числовий метод визначає порядок дій при переході від даної точки кривої до наступної.

Оскільки числове розв'язування задачі Коші широко застосовується в різних областях науки і техніки, то воно протягом багатьох років було об'єктом пильної уваги і має велику кількість розроблених для нього методів. Зупинимося тут на двох групах методів розв'язування задачі Коші.

1. *Однокрокові методи* – такі, 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) – такі, в яких для відшукання наступної точки кривої $y=f(x)$ потрібна інформація про більш ніж одну з попередніх точок. Щоб отримати достатньо точне числове значення, часто вдаються до ітерації. До таких методів належать, зокрема, методи Адамса.

Для знаходження числового розв'язку на відрізку $[a, b]$, де $x_0 = a,$ уведемо 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 у числовому вигляді сіткової функції $y^{(h)}$. Для оцінювання похибки наближеного числового розв'язку розглядатимемо цей розв'язок як елемент (N+1)-вимірного лінійного векторного простору з певною нормою, а похибку як норму відхилу $\delta^{(h)} = y^{(h)} - [y]^{(h)}$, 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

Усім однокроковим методам властиві певні загальні риси.

1. Щоб отримати інформацію в новій точці, треба мати дані лише в одній попередній точці..

2. В основі всіх однокрокових методів лежить розвинення функції в ряд Тейлора, в якому зберігаються члени, що містять h у степені до k включно. Ціле число k називається *порядком* методу. Похибка на кроці має порядок k+1.

3. Усі 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 1 залежно від інформації лише попереднього кроку дозволяє легко міняти значення 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 1-го порядку. Його точність невелика, тому на практиці ним користуються порівняно рідко. Проте цей метод дає змогу легше зрозуміти алгоритм інших, ефективніших методів.

Метод Ейлера оснований на розвиненні y 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 у другому або вищих степенях, 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. Ураховуючи, 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 виконуючи скільки завгодно багато кроків. Графічно метод Ейлера показано на рис. 9.1.

9.2.2. Похибка методу Ейлера

В ітеративних числових методах є 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 значеннями величини на кожному етапі реалізації числового алгоритму, що визначає сумарну похибку, яка накопичилася з моменту початку обчислень.

Локальна похибка методу Ейлера визначається за формулою $\varepsilon_k^h = \frac{y''(\xi)}{2} h^2$, де $\xi \in [x_{k-1}, x_k]$, і має порядок h^2, оскільки члени, що містять h у другому і вищих степенях, 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 1-й порядок точності відносно величини кроку h.

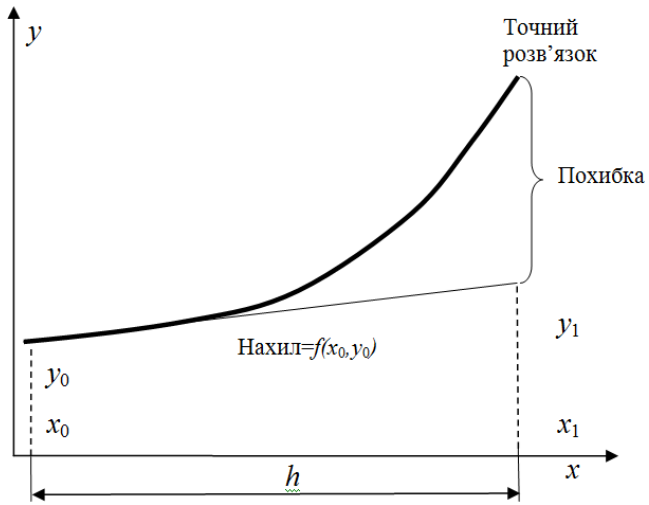


Рис. 9.1. Метод Ейлера

Блок-схему алгоритму диференціювання за методом Ейлера представлено на рис. 9.2.

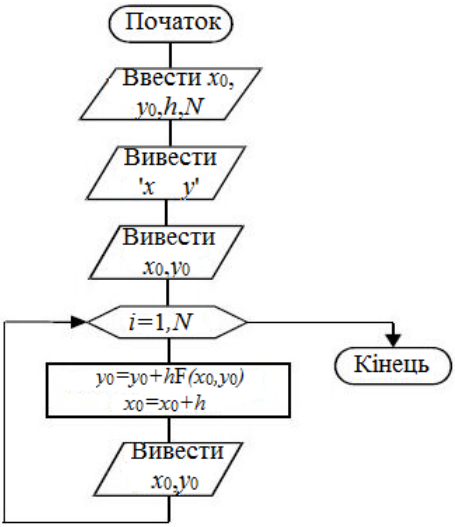


Рис. 9.2. Блок-схема алгоритму диференціювання методом Ейлера

У лістингу 9.1 подано реалізацію алгоритму методом Ейлера мовою *Python*.

Лістинг 9.1. Реалізація алгоритму диференціювання за методом Ейлера

```
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)
```

Ця реалізація розв'язує задачу 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 наведено нижче.

```
>>>
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
```

Як видно з результатів, метод Ейлера дуже неточний, 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 $y'(x_0)$, він змінюється відповідно до зміни незалежної x. Тому в точці x_0+h нахил дотичної вже не такий, яким він був у точці x_0. Отже, при збереженні початкового нахилу дотичною на всьому інтервалі h до результатів обчис-

лень 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 методом Ейлера $y^*_{n+1} = y_n + hf(x_n, y_n)$. Це значення використовується потім для обчислення наближеного значення похідної в кінці інтервалу $f(x_{n+1}, y^*_{n+i})$. Обчисливши середнє між цим значенням похідної і її значенням на початку інтервалу, знайдемо точніше значення y_{n+1}:

$$y_{n+1} = y_n + \frac{1}{2}h[f(x_n, y_n) + f(x_{n+1}, y^*_{n+1})]. \qquad (9.4)$$

Цей прийом ілюструється рис. 9.3. Принцип, на якому основано модифікований метод Ейлера, можна пояснити й інакше.

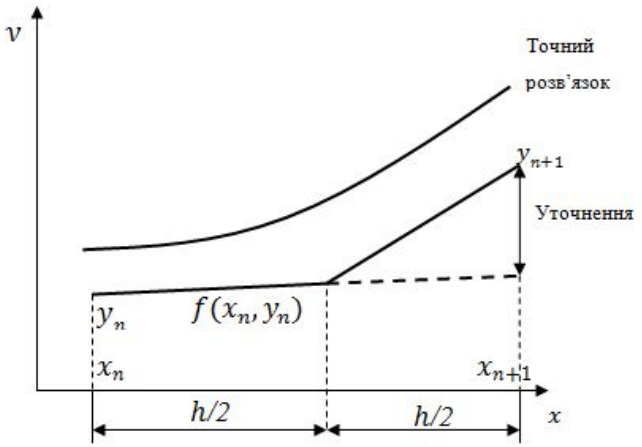


Рис. 9.3. Модифікований метод Ейлера

Для цього повернемося до розвинення функції в ряд Тейлора:

$$\left| x^{(k+1)} - x^{(k)} \right| < \varepsilon. \qquad (9.5)$$

Здається очевидним, що, зберігши член з h^2 і відкинувши члени вищих порядків, можна підвищити точність. Проте, щоб зберегти член з h^2, треба знати другу похідну $y''(x_0)$. Її можна апроксимувати скінченною різницею:

$$y''(x_0) = \frac{\Delta y'}{\Delta x} = \frac{y'(x_0 + h) - y'(x_0)}{h}. \qquad (9.6)$$

Підставивши цей вираз у формулу Тейлора (9.5) з відкинутими членами, вищими за 2-й порядок, знайдемо

$$y(x_0 + h) = y(x_0) + \frac{1}{2}h[y'(x_0 + h) + y'(x_0)], \qquad (9.7)$$

що практично збігається 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, у загальному випадку

$$y(x_{n+1}) = y(x_n) + \frac{1}{2}h[y'(x_{n+1}) + y'(x_n)]. \qquad (9.8)$$

Оскільки наступне значення (функції і похідної) входить до лівої і правої частин рівності, то таку формулу називають *неявною*. Якщо функція $f(x, y)$ лінійна по y, то тоді неявне рівняння можна розв'язати відносно y_{n+1}. Але, у більшості випадків таке розв'язання неможливе і тоді використовують інші схеми, зокрема *ітераційні*, для знаходження похідних у більш, ніж одній точці.

Цей метод є методом 2-го порядку точності, оскільки в ньому використовується член ряду Тейлора, що містить h^2. Помилка на кожному кроці при використанні цього методу, має порядок h^3. За підвищення точності доводиться розплачуватися додатковими витратами машинного часу, необхідними для обчислення 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 сама ідея лежить в основі методів Рунге – Кутта.

Блок-схему модифікованого алгоритму Ейлера представлено на рис. 9.4.

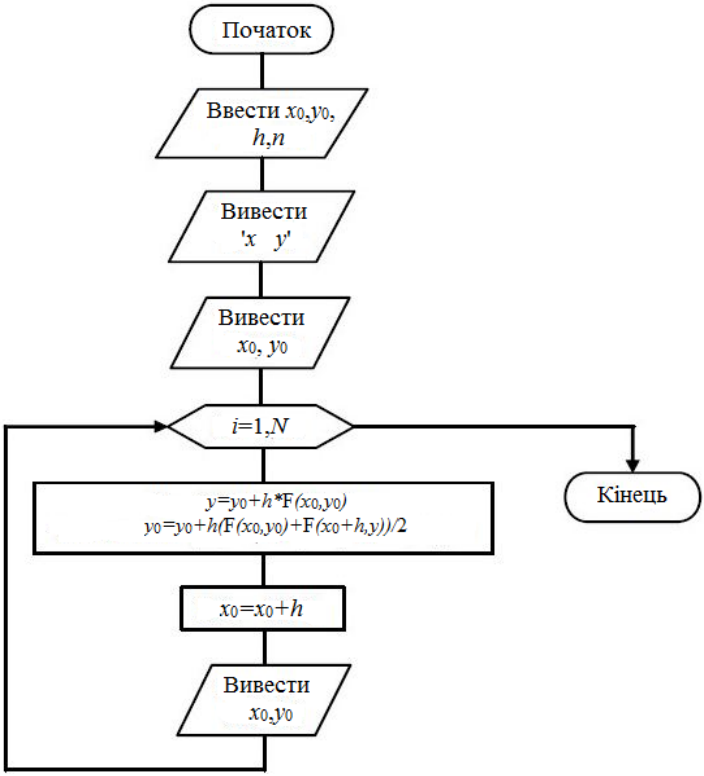


Рис. 9.4. Блок-схема алгоритму диференціювання за модифікованим методом Ейлера

Реалізацію запропонованого алгоритму наведено в лістингу 9.2.

Лістинг 9.2. Реалізація алгоритму диференціювання за модифікованим методом Ейлера

```
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)
```

Дана реалізація розв'язує задачу 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 наведено нижче.

```
>>>
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
```

Як видно з результатів, модифікований метод Ейлера досить точний і має дуже просту реалізацію.

Розглянемо декілька прикладів.

Приклад 9.1. Явним методом Ейлера з кроком $h = 0{,}1$ отримати числовий розв'язок диференціального рівняння $y' = (y + x)^2$ з початковими умовами $y(0) = 0$ в інтервалі [0, 0.5]. Числовий розв'язок порівняти з точним розв'язком $y = \mathrm{tg}(x) - x$.

Розв'язання. 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 $\varepsilon_k = \left| y_{\text{точн}.}(x_k) - y_k \right|$, де $y_{\text{точн}.}(x_k)$ – точний розв'язок у вузлових точках, отримані результати занесемо в табл. 9.1.

Таблиця 9.1. **Отримані результати**

k	x	y	Δy_k	$y_{\text{точн}}$	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

Розв'язком задачі є таблична функція (табл. 9.2), тут залишено 5 значущих цифр у кожному числі.

Таблиця 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 у 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.

Таблиця 9.3. **Отримані результати**

k	x_k	y_k	$\tilde{y}_k$	Δy_k	$y_{\text{точн.}}$	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

Розв'язком задачі є таблична функція (табл. 9.4), тут залишено 5 значущих цифр у кожному числі.

Таблиця 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 $(n-y)$ похідну залежної змінної. При використанні модифікованого методу Ейлера для отримання другої похідної в скінченно-різницевій формі достатньо було знати нахили кривої на кінцях даного інтервалу. Щоб обчислити третю похідну в скінченно-різницевому вигляді, необхідно мати значення другої похідної щонайменше у двох точках. Для цього треба додатково визначити нахил кривої в деякій проміжній точці інтервалу *h*, тобто між x_n та x_{n+1}. Очевидно, чим вищий порядок обчислюваної похідної, тим більше додаткових обчислень буде потрібно всередині 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 сім'я методів розв'язування диференціальних рівнянь 1-го порядку.

Не станемо детально виводити повні формули, а розглянемо натомість більш простий метод 2-го порядку. Цей метод використовує лише два обчислення функції за крок.

Перше з них – $k_1 = hf(x,y)$, потім робиться дробовий крок на основі k_1. Уведемо два поки невизначених коефіцієнти *a* та *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, розвинемо одержаний вираз через формулу Тейлора відносно точки x_n, розглядаючи k_2 як функцію двох змінних: $k_2 = hf(x,y) + h(bk_1f_y + ahf_x)$, де f_x і f_y є частинними похідними $f(x, y)$. Підставляючи це значення у вираз для $y(xn+h) = y_{n+1}$ 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 розвиненням точного розв'язку

$$y(x_n + h) = y(x_n) + hy'(x_n) + \frac{1}{2}h^2 y''(x_n) =$$
$$= y(x_n + h) = y(x_n) + hf(x_n, y_n) + \frac{h^2}{2}(f_y f(x_n, y_n) + f_x) \quad (9.9)$$

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 із цих коефіцієнтів, наприклад a, за параметр, приходимо до однопараметричної сім'ї методів Рунге – Кутта:

$$k_1 = hf(x, y);$$
$$k_2 = hf(x + ah, y + ak_1); \quad (9.10)$$
$$y_{n+1} = y_n + (1 - \frac{1}{2a})k_1 + \frac{1}{2a}k_2.$$

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 вище згаданих розвиненнях, показує, що ні при якому виборі a не вдається виключити ці члени для всіх функцій $f(x, y)$ і таким чином, метод має 2-й порядок точності при будь-якому a.

Розглянутий метод Рунге – Кутта є узагальненням методів Ейлера 2-го порядку точності. Для одержання більш точних методів вдаються до більш складних схем, що утримують у розвиненні Тейлора похідні старших порядків.

Реалізацію методу Рунге – Кутта 2-го порядку на прикладі рівняння Бесселя в інтервалі [0,5;1] із параметром $P = 0$ і початкових умовах $y_1(0) = 0{,}9384698$ та $y_2(0) = -0{,}2422685$ наведено в лістингу 9.3.

Лістинг 9.3. Реалізація
методу Рунге – Кутта 2-го порядку

```
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])
```

Результат виконання програми:

```
>>>
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
```

Найбільш поширеним із методів Рунге – Кутта є метод, при якому одержуються всі члени, включаючи h_4. Це метод 4-го порядку точності, для якого помилка на кроці має порядок 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

$$\begin{aligned} K_1 &= hf(x_n, y_n), \\ K_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{K_1}{2}), \\ K_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2), \\ K_4 &= hf(x_n + h, y_n + K_3). \end{aligned} \qquad (9.11)$$

Реалізацію методу Рунге – Кутта 4-го порядку наведено в лістингу 9.4.

Лістинг 9.4. Реалізація методу Рунге – Кутта 4-го порядку

```
# -*- 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)
```

Для цієї реалізації (лістинг 9.4) узято дані з прикладу 9.3. Результат роботи програми наведено нижче. Основне навантаження в програмі виконує функція **Runge44**, на вхід якої подають такі параметри: **X** – робоча точка, **h** – крок, **Y** – вектор початкових умов, який після виконання утримуватиме розраховані дані, **n** – кількість точок. Функція **RP44** обчислює 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

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 є методами Рунге – Кутта 1-го і 2-го порядку відповідно. Порівняно з ними метод Рунге – Кутта має важливу перевагу, оскільки забезпечує вищу точність, яка виправдовує додаткове збільшення обсягу обчислень. Вища точність методу Рунге – Кутта часто дозволяє збільшити крок інтеграції h. Допустима похибка на кроці визначає його максимальне значення. 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 прикладі.

Приклад 9.3. Нехай потрібно розв'язати рівняння

$$\frac{dy}{dx} = 2x^2 + 2y$$

за початкової умови $y(0)= 1$, $0 \le x \le 1$ і $h = 0{,}1$. Це – лінійне рівняння 1-го порядку, що має точний розв'язок

$$y = 1{,}5e^{2x} - x^2 - x - 0{,}5,$$

який допоможе порівняти відносну точність, що забезпечується різними методами. Результати розрахунку наведено в табл. 9.5, з якої добре видно переваги методу Рунге – Кутта порівняно зі звичайним і модифікованим методами Ейлера.

Таблиця 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 зведемо до виду, що є зручним для програмування:

$$y(x_0 + h) = y_0 + \frac{k_4 + k_5}{2} 0(h^5), \tag{9.12}$$

де

$$k_1 = h_3 f(x_0, y_0), h_3 = \frac{h}{3};$$

$$k_2 = h_3 f(x_0 + h_3, y_0 + k_1);$$

$$k_3 = h_3 f(x_0 + h_3, y_0 + \frac{k_1 + k_2}{2});$$

$$k_4 = k_1 + 4h_3 f(x_0 + \frac{h}{2}, y_0 + 0,375(k_1 + k_3);$$

$$k_5 = h_3 f(x_0 + h, y_0 + 1,5(k_4 - k_3)).$$

Схема Мерсона вимагає обчислювати на кожному кроці праву частину звичайного диференціального рівняння в п'яти точках, але за рахунок тільки одного додаткового коефіцієнта k_1 порівняно з класичною схемою Рунге – Кутта на кожному кроці можна визначити похибку розв'язку R за формулою

$$10R = 2k_4 - 3k_3 - k_5 . \qquad (9.13)$$

Для автоматичного вибору кроку інтегрування рекомендується критерій (9.14). Якщо абсолютне значення величини R, обчислене за формулою (9.13), стане більшим допустимої заданої похибки ε,

$$|R| < \varepsilon , \qquad (9.14)$$

то крок h зменшується вдвічі й обчислення за схемою (9.12) повторюються з точки (x_0, y_0). При виконанні умови

$$32|R| < \varepsilon \qquad (9.15)$$

крок h можна подвоїти.

Реалізацію методу Рунге – Кутта – Мерсона у вигляді модуля наведено в лістингу 9.5.

Лістинг 9.5. Реалізація методу Рунге – Кутта – Мерсона

```
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
```

Написання програми для застосування запропонованого модуля пропонується як самостійна вправа.

Контрольні запитання

1. Від чого залежить точність отримуваного результата?
2. Що таке якість "самостартування"?
3. У чому відмінність однокрокових методів від багатокрокових?
4. Наскільки є точнішим модифікований метод Ейлера порівняно зі звичайним?
5. Яким чином визначається порядок точності методу?
6. Метод Рунге–Кутта 2-го порядку.
7. Метод Рунге–Кутта 4-го порядку.
8. Метод Рунге–Кутта–Мерсона.

Список літератури

1. *Дорогий Я. Ю.* Вступ до мови програмування Python : навч. посіб. для студ. напрямку "Системна інженерія" [Електронний ресурс] / Я. Ю. Дорогий, Є. В. Глушко, А. Ю. Дорошенко. – К. : НТУУ "КПІ", 2010. – 319 с. Режим доступу : http://cisco-academy.com.ua/uploads/book1.pdf.

2. *Бизли Дэвид М.* Язык программирования Python : справочник : [пер. с англ.] / Дэвид М. Бизли. – К. : ДиаСофт, 2000. – 336 с.

3. *Бахвалов Н. С.* Численные методы / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельков. – М. : Лаб. базовых знаний, 2002. – 632 с. : ил.

4. *Шуп Т.* Решение инженерных задач на ЭВМ : практическое руководство :[пер. с англ.] / Т. Шуп. – М. : Мир, 1982. – 238 с. : ил.

5. *Мудров А. Е.* Численные методы для ПЭВМ на языках Бейсик, Фортран и Паскаль / А. Е. Мудров. – Томск : МП "Раско", 1991. – 272 с. : ил.

6. http://python.org.

Додаток
КЛАСИ ВИНЯТКОВИХ СИТУАЦІЙ

Нижче наведено необхідну для професійного програмування мовою *Python* довідкову інформацію про визначені виняткові ситуації.

Код користувача може генерувати вбудовані виняткові ситуації. Цю властивість можна використовувати для перевірки обробки виняткової ситуації, або щоб повідомити про помилку в "просто схожій" ситуації, в якій інтерпретатор генеруватиме таку саму виняткову ситуацію; однак будьте уважні – немає нічого, що може захистити код користувача від генерування помилки.

Наступні виняткові ситуації використовуються тільки як базові класи для інших.

Exception

є базовим класом для виняткових ситуацій. Усі вбудовані виняткові ситуації успадковують від нього. Усім визначеним користувачем винятковим ситуаціям також варто успадковувати від нього, але це (ще) не категорична вимога. Функція **str()** при застосуванні до екземпляра цього класу (або до більшості успадкованих класів) повертає рядкове значення аргументу або аргументів, хоча може повернути й порожній рядок, якщо конструктору не було передано аргументи. При використанні як послідовності, надає доступ до аргументів, які передано конструктору (зручно для зворотної сумісності зі старим кодом). Аргументи як кортеж також доступні в атрибуті **args** екземпляра.

StandardError

є базовим класом для всіх убудованих виняткових ситуацій, крім **SystemExit. StandardError** у свою чергу успадковує від кореневого класу **Exception**.

ArithmeticError

являє собою базовий клас для тих убудованих виняткових ситуацій, які виникають при різних арифметичних помилках: **OverflowError**, **ZeroDivisionError**, **FloatingPointError**.

LookupError

є базовим класом для виняткових ситуацій, які виникають, коли ключ або індекс при застосуванні до відображення або послідовності виявляється некоректним: **IndexError**, **KeyError**.

EnvironmentError

є базовий клас для виняткових ситуацій, які можуть виникнути за межами *Python*-системи: **IOError**, **OSError**. Коли виняткова ситуація цього типу створюється подвійним кортежем, перший елемент доступний як атрибут **errno** екземпляра (передбачається, що це номер помилки), а другий елемент через атрибут **strerror** (зазвичай, це повідомлення про помилку, асоційоване з кодом помилки). Сам кортеж доступний через атрибут **args**.

Коли породжується екземпляр виняткової ситуації **EnvironmentError** із потрійним кортежем, перші два елементи доступні, як вказано вище, а третій – через атрибут **filename**. Однак, для зворотної сумісності, атрибут **args** містить тільки подвійний кортеж перших двох аргументів конструктора.

Якщо цю виняткову ситуацію створено не з трьома аргументами, то атрибут **filename** дорівнює **None**. Атрибути **errno** та **strerror** також рівні **None**, коли екземпляр було створено не з двома або трьома аргументами. У цьому разі **args** містить дослівно аргументи конструктора як кортеж.

Наведемо виняткові ситуації, які реально можуть виникати і є об'єктами.

AssertionError

виникає, коли оператор **assert** зазнає невдачі.

AttributeError

з'являється, коли посилання або присвоювання атрибуту зазнає невдачі. (Якщо об'єкт зовсім не підтримує посилання на атрибут або присвоювання атрибуту, маємо виняткову ситуацію **TypeError.**)

EOFError

виникає, коли одна з убудованих функцій **input()** або **raw_input()** зустрічає кінець файла (EOF), не прочитавши жодних даних. (Але методи **read()** і **readline()** файлових об'єктів повертають порожній рядок, коли вони зустрічають EOF.)

FloatingPointError

з'являється, коли операція над числами з рухомою крапкою зазнає невдачі. Ця виняткова ситуація завжди визначена, але може виникати тільки, коли *Python* сконфігуровано з опцією

-with-fpectl або із символом **WANT_SIGFPE_HANDLER**, визначеним у файлі **config.h**.

IOError

виникає, коли операція введення/виведення (за допомогою оператора **print**, убудованої функції **open()** або методу файлового об'єкта) зазнає невдачі з причини, пов'язаної з уведенням/виведенням, наприклад, *"файл не знайдено"* або *"диск переповнений"*. Цей клас породжений від **EnvironmentError**. Дивіться обговорення, наведене вище, щоб одержати додаткову інформацію про атрибути екземпляра виняткової ситуації.

ImportError

з'являється, коли оператор **import** зазнає невдачі в пошуку визначення модуля, або коли **from ... import** зазнає невдачі в пошуку імпортованого імені.

IndexError

виникає, коли індекс послідовності виявляється за межами діапазону. (Індекси зрізів мовчки обрізаються так, щоб потрапити в допустиму область; якщо індекс не є звичайним цілим, маємо виняткову ситуацію **TypeError.**)

KeyError

з'являється, коли ключ відображення (словника) не знайдено в наборі наявних ключів.

KeyboardInterrupt

виникає, коли користувач натискає клавіші переривання (звичайно, **Ctrl-C** або **DEL**). При виконанні перевірка переривання здійснюється регулярно. Натискання перериваючих клавіш у той час, коли вбудовані функції **input()** або **raw_input()** очікують уведення, також генерує цю виняткову ситуацію.

MemoryError

виникає, коли операція вичерпує пам'ять, але ситуацію можна все ще врятувати (наприклад, видаливши з пам'яті кілька об'єктів). Асоційоване значення – це рядок, що показує, яка (внутрішня) операція вичерпала пам'ять.

Зазначимо, що через архітектуру керування пам'яттю, що лежить в основі (функції **malloc()** мови C), інтерпретатор не завжди може повністю відновитися в такій ситуації; проте , він генерує виняткові ситуації для того, щоб можна було надрукувати стан стеку в разі, коли причина була у виконуваній програмі.

NameError

виникає, коли не знайдено глобальне або локальне ім'я. Це стосується тільки невизначених імен. Асоційоване значення є іменем, яке не було виявлено.

NotImplementedError

є винятковою ситуацією, яка успадковує від **RuntimeError**. Її повинні генерувати абстрактні методи визначених користувачем базових класів, коли вони вимагають перевизначення цих методів в успадкованих класах.

OSError

є класом, який успадковує від **EnvironmentError** та використовується в основному як виняткова ситуація **os.error** модуля **os**. Дивіться вище **EnvironmentError** стосовно можливих асоційованих значень.

OverflowError

виникає, коли результат арифметичної операції занадто великий. Не виникає для довгих цілих (при обробці яких у разі проблеми радше генеруватиметься **MemoryError**). Через відсут-

ність стандартизованої обробки виняткових ситуацій для чисел із рухомою крапкою в мові С, більшість операцій з такими числами не перевіряються. Для звичайних цілих всі операції, які можуть викликати переповнення, перевіряються, за винятком побітового зсуву ліворуч, для якого зникнення лівого біта зазвичай ігнорується.

RuntimeError

з'являється, коли виникає помилка, що не потрапляє в жодну з інших категорій. Асоційованим значенням є рядок, що точно вказує, у чому річ. (Ця виняткова ситуація є реліктом попередньої версії інтерпретатора та використовується не надто часто.)

SyntaxError

виникає, коли інтерпретатор зустрічає синтаксичну помилку. Це може трапитися в операторі **import**, **exec**, у виклику вбудованої функції **eval()** або **input()**, а також при зчитуванні скрипта зі стандартного потоку введення(а також в інтерактивному режимі).

Коли використовуються класові виняткові ситуації, екземпляри цих класів мають атрибути **filename**, **lineno**, **offset** та **text** для зручного доступу до деталей; для рядкових виняткових ситуацій асоційоване значення зазвичай є кортежем у вигляді (**message**, (**filename**, **lineno**, **offset**, **text**)). Для класових виняткових ситуацій, **str()** повертає тільки повідомлення.

SystemError

виникає, коли інтерпретатор виявляє зовнішню помилку, але ситуація не виглядає зовсім безнадійною. Асоційоване значення вказує, що відбулося (у низькорівневих термінах).

SystemExit

є винятковою ситуацією, яка ініціюється функцією **sys.exit()**. Якщо її не обробити, інтерпретатор *Python* завершує роботу; при цьому не виводиться стан стеку. Якщо асоційоване значення є звичайним цілим, воно визначає системний статус виходу (який передається у функцію **exit()** мови С); якщо воно дорівнює

None, статус виходу буде нульовим, якщо воно має інший тип (наприклад, рядок), значення об'єкта друкується та статус виходу встановлюється рівним одиниці. Коли використовуються класові виняткові ситуації, їх екземпляр має атрибут **code**, у якому міститься бажаний статус виходу або повідомлення про помилку (за замовчуванням дорівнює **None**). Також ця виняткова ситуація успадковує прямо від **Exception**, а не від **StandardError**, оскільки насправді це не помилка. Виклик **sys.exit()** транслюється у виняткову ситуацію для того, щоб можна було виконати завершальні оператори обробки (пункт **finally** оператора **try**), після чого налагоджувач може виконувати скрипт без небезпеки втратити керування. Якщо абсолютно необхідно вийти негайно, то можна скористатися функцією **os._exit()** (наприклад, після **fork()** у дочірньому процесі).

TypeError

виникає, коли вбудована операція або функція застосовується до об'єкта невідповідного типу. Асоційованим значенням є рядок, що пояснює деталі про несумісність типів.

ValueError

з'являється, коли вбудована операція або функція одержує аргумент коректного типу, але невідповідного значення, і ситуацію не можна описати докладнішою винятковою ситуацією, такою як **IndexError**.

ZeroDivisionError

виникає, коли другий аргумент операції ділення або взяття остачі дорівнює нулю. Асоційоване значення – рядок, що описує тип операндів та операції.

Навчальне видання

ДОРОШЕНКО Анатолій Юхимович
ПОГОРІЛИЙ Сергій Дем'янович
ДОРОГИЙ Ярослав Юрійович
ГЛУШКО Євгеній Васильович

ПРОГРАМУВАННЯ ЧИСЛОВИХ МЕТОДІВ МОВОЮ *PYTHON*

Навчальний посібник

Редактор *Л. В. Магда*
Технічний редактор *Л. П. Шевченко*

Оригінал-макет виготовлено Видавничо-поліграфічним центром "Київський університет"

Формат 60x84$^{1/16}$. Ум. друк. арк. 26,97. Наклад 300. Зам. № 213-6798. Вид. № К2.
Гарнітура Times New Roman. Папір офсетний. Друк офсетний.
Підписано до друку 16.12.13

Видавець і виготовлювач
Видавничо-поліграфічний центр "Київський університет",
б-р Т. Шевченка, 14, м. Київ, 01601
☎ (38044) 239 32 22; (38044) 239 31 72; тел./факс (38044) 239 31 28
e-mail:vpc@univ.kiev.ua
http: vpc.univ.kiev.ua
Свідоцтво суб'єкта видавничої справи ДК № 1103 від 31.10.02