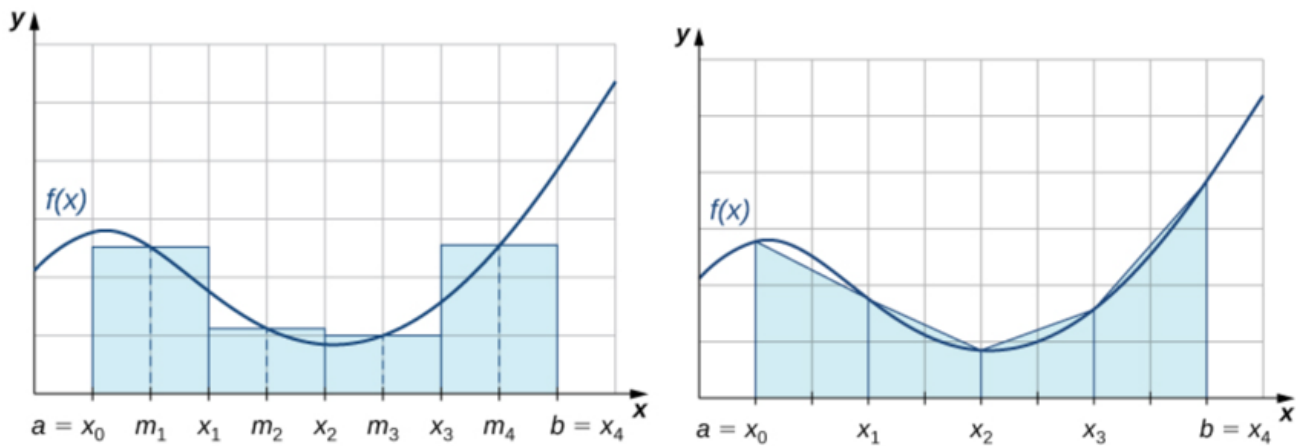


Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

В. А. Головацький

ПРОГРАМУВАННЯ ТА МАТЕМАТИЧНЕ МОДЕЛЮВАННЯ

Навчально-методичний посібник



$$\int_a^b f(x) dx \approx \frac{\Delta x}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + f(x_4))$$



Чернівці
Чернівецький національний університет
2025

Друкується за ухвалою Вченої ради
Навчально-наукового Інституту фізико-технічних та комп'ютерних наук
Чернівецького національного університету
імені Юрія Федьковича

Програмування та математичне моделювання: Навчально-методичний посібник /
В. А. Головацький. — Чернівці: Чернівецький нац. ун-т, 2025. — 158 с.

У посібнику викладено основи програмування мовою **Python** та основні числові методи, що використовуються при комп'ютерному моделюванні фізичних процесів та явищ. Розглянуто велику кількість програм реалізації стандартних алгоритмів. Крім цього, для можливості створення інтерактивних графічних демонстрацій у посібнику подано основи **Wolfram Mathematica**. Матеріал посібника відповідає дисципліні «Програмування та математичне моделювання», яка викладається для студентів спеціальності Е6 «Прикладна фізика та наноматеріали». Може використовуватися студентами та аспірантами інших спеціальностей галузі знань Е — Природничі науки, математика та статистика.

Зміст

I	ОСНОВИ PYTHON	8
1	Вступ до Python	9
1.1	Що таке Python, чому він популярний та з чого почати?	9
1.2	Змінні та типи даних	10
1.2.1	Присвоєння змінних	10
1.2.2	Основні типи даних	11
1.2.3	Складніші типи даних	12
1.2.4	Правила створення імен змінних у Python	12
1.3	Введення та виведення даних	13
1.3.1	Виведення на екран	13
1.3.2	Ввід даних від користувача	14
1.3.3	Об'єднання рядків між собою та з числами	14
1.4	Символьні рядки	15
1.4.1	Основні операції з рядками	15
1.4.2	Методи рядків	15
1.4.3	ASCII коди символів	15
1.5	Оператори	21
1.6	Умовні конструкції (if, else, elif)	23
1.6.1	Тернарний оператор if	24
1.7	Цикли (for, while)	24
1.8	Структури даних list, tuple, set, dict	26
1.8.1	Списки (list)	27
1.8.2	Кортежі (tuple)	29
1.8.3	Множини (set)	30
1.8.4	Словники (dict)	31
1.9	Списки списків (матриці)	32
1.10	Генератори списків (list comprehensions)	32
1.10.1	Використання тернарного оператора if в генераторах списків	33
1.11	Зрізи у структурах даних	34
1.12	Розпакування ітерабельних об'єктів	35
1.12.1	Присвоєння елементів ітерабельного об'єкта кільком змінним	35
1.12.2	Використання зірочки *	35
1.12.3	Розпакування в циклах	35
1.12.4	Функція enumerate()	36
1.12.5	Розпакування вкладених структур	36
1.13	Функції у Python	36
1.13.1	Аргументи за замовчуванням	37
1.13.2	Лямбда-функції та іменовані аргументи	37

1.14	Робота з файлами	38
1.14.1	Запис у файл	38
1.14.2	Читання з файлу	38
1.14.3	Читання окремих рядків або списком	39
1.14.4	Режими відкриття файлу	39
1.14.5	Робота з CSV-файлами	39
1.14.6	Зчитування даних з txt-файла розміщеного на Google диску	40
1.14.7	Обробка винятків при роботі з файлами	40
1.15	Імпорт модулів і використання бібліотек	41
1.15.1	Що таке модуль у Python	41
1.15.2	Способи імпорту модулів	41
1.15.3	Стандартна бібліотека Python	43
1.15.4	Висновки	44
1.16	Обробка винятків (помилки) у Python	45
1.16.1	Що таке виняток?	45
1.16.2	Основні типи винятків	45
1.16.3	Синтаксис інструкції <code>try</code>	45
1.16.4	Блоки <code>else</code> та <code>finally</code>	46
1.16.5	Приклад: сума чисел з файлу	46
2	Об'єктно-орієнтоване програмування	47
2.1	Клас, конструктор та екземпляр	47
2.2	Розширення класу	48
2.3	Інкапсуляція: доступ до атрибутів через методи	49
2.4	Приклад створення класу комплексних чисел	50
3	Побудова графіків за допомогою matplotlib	53
3.1	Типи графіків	53
3.2	Приклади побудови графіків функцій	57
3.3	Приклад побудови стовпчикової діаграми	59
3.4	Побудова тривимірних графіків у Matplotlib	59
3.5	Висновки	61
II	ЧИСЕЛЬНІ МЕТОДИ	62
1	Похибки	63
1.1	Класифікація похибок	63
1.2	Абсолютна та відносна похибки	64
1.3	Операції з наближеними числами	65
2	Апроксимація	69
2.1	Апроксимація даних	69
2.2	Інтерполяція	69
2.2.1	Задача інтерполяції	69
2.2.2	Локальна та глобальна інтерполяція	70
2.2.3	Кусково-лінійна інтерполяція	70
2.2.4	Кусково-квадратична інтерполяція	70
2.2.5	Многочлен Лагранжа	71
2.2.6	Многочлен Ньютона	71
2.3	Регресія	73
2.3.1	Поняття регресії	73
2.3.2	Типи регресійних моделей	73

2.3.3	Метод найменших квадратів (МНК)	73
2.3.4	Нелінійна регресія	75
2.3.5	Оцінка якості регресії	76
2.3.6	Вибір виду регресії	76
3	Чисельне розв'язування рівнянь	77
3.1	Метод Ньютона	78
3.2	Метод половинного ділення (дихотомія)	79
3.3	Метод січних	80
3.4	Метод хорд	81
4	Чисельне інтегрування	83
4.1	Формула Ньютона — Котеса	84
4.2	Метод прямокутників	86
4.3	Метод трапецій	87
4.4	Метод Сімпсона	88
4.5	Порівняння чисельних методів інтегрування	89
4.6	Обчислення багатовимірних інтегралів	90
5	Розв'язок диференціальних рівнянь	95
5.1	Чисельне диференціювання	95
5.2	Суть методу скінченних різниць (сіток)	96
6	Алгоритми сортування	98
6.1	Сортування вибором (Selection Sort)	98
6.2	Сортування вставками (Insertion Sort)	99
6.3	Бульбашкове сортування (Bubble Sort)	100
6.4	Швидке сортування (Quick Sort)	100
6.5	Сортування злиттям (Merge Sort)	102
III	ДОДАТКИ. ПРИКЛАДИ PYTHON ПРОГРАМ	103
1	Чисельне знаходження кореня рівняння	104
1.1	Метод Ньютона	104
1.2	Метод січних (Secant Method)	104
1.3	Метод хорд (Regula Falsi)	105
1.4	Метод половинного ділення (Bisection Method)	106
2	Чисельне інтегрування	107
2.1	Метод прямокутників	107
2.2	Метод трапецій	109
2.3	Метод Сімпсона	109
3	Розв'язок диференціальних рівнянь	111
3.1	Розв'язання диференціального рівняння ітераційним методом	111
3.2	Скінченно-різницевий розв'язок рівняння з неоднорідними граничними умовами	112
4	Сортування масивів	114
4.1	Програми сортування різними алгоритмами та порівняння швидкості їх роботи	114

5	Робота з матрицями	117
5.1	Створення матриць	117
5.2	Транспонування матриці	118
5.3	Додавання та множення матриць	118
5.4	Обчислення визначника матриці рекурсивним методом	119
5.5	Обчислення визначника методом Гауса	121
5.6	Обернена матриця	123
5.7	Обернена матриця (модуль NumPy)	124
5.8	Розв'язання систем лінійних рівнянь	124
5.9	Метод Крамера	125

IV ОСНОВИ Wolfram Mathematica 126

1	Синтаксис, базові конструкції, елементи програмування	127
1.1	Загальні відомості. Ядро та інтерфейс пакета <i>Mathematica</i>	127
1.2	Точні та наближені обчислення. Чисельні та аналітичні операції	128
1.2.1	Базовий синтаксис	129
1.2.2	Арифметичні операції	129
1.2.3	Коментарі	129
1.2.4	Присвоєння значень змінним	130
1.3	Робота з текстовими рядками	130
1.4	Масиви та операції з ними	130
1.4.1	Створення та візуалізація масивів	130
1.4.2	Основні операції з масивами	131
1.5	Правила підстановки	132
1.6	Логічні та умовні оператори	133
1.6.1	Умовні конструкції	133
1.7	Цикли і рекурсія	133
1.7.1	Цикл For	133
1.7.2	Цикл While	134
1.7.3	Цикл Do	134
1.7.4	Table	134
1.7.5	Рекурсивні функції	134
1.8	Операції математичного аналізу у Wolfram Mathematica	135
1.8.1	Границі	135
1.8.2	Похідні	136
1.8.3	Інтегрування	136
1.8.4	Розклад функції в ряд Тейлора	138
2	Побудова графіків функцій у Wolfram Mathematica	140
2.1	Побудова графіка за допомогою функції Plot[]	140
2.2	Графік кількох функцій за допомогою Plot з використанням опцій	141
2.3	Графік кількох функцій з використанням Plot та додаткових опцій оформлення	142
2.4	Побудова точкових графіків	143
2.5	Об'єднання графіків точок і функцій	143
2.6	Графік у тривимірному просторі	144
2.7	Графік 3D поверхні з додатковими просторовими об'єктами	145

3	Інтерактивні демонстрації у Wolfram Mathematica	147
3.1	Основний синтаксис	147
3.1.1	Приклад роботи Manipulate для зміни частоти та фази	147
3.2	Налаштування елементів керування ControlType	148
3.2.1	SetterBar	148
3.2.2	RadioButtonBar	148
3.2.3	Використання прапорців (Checkbox)	149
3.2.4	Анімація гармонічного осцилятора	149
3.2.5	Використання ControlType для різних елементів	150
3.3	Демонстрація затухаючих коливань з використанням Manipulate	151
3.4	Фігури Ліссажу	155

Частина І

ОСНОВИ PYTHON

Розділ 1

Вступ до Python

Python — це популярна мова програмування, яку легко вивчити. Вона використовується для всього: від простих скриптів до складних вебзастосунків, наукових досліджень і штучного інтелекту. У цьому підручнику ми спробуємо пояснити все простою мовою та з прикладами. Навіть якщо ти ніколи не програмував(-ла), не хвилюйся — почнемо з найпростішого.

1.1. Що таке Python, чому він популярний та з чого почати?

Python створив Гвідо ван Россум у 1991 році. Назва не має нічого спільного зі зміями — це відсилання до британського комедійного шоу “Monty Python’s Flying Circus”. Гвідо ван Россум, був фанатом їхнього гумору.

Чому Python такий популярний:

- Простий та зрозумілий синтаксис
- Безкоштовний і відкритий
- Працює майже скрізь — Windows, Mac, Linux
- Має багато готових бібліотек (для графіки, науки, вебу тощо)

Python вважається однією з найкращих мов для першого знайомства з програмуванням. Його синтаксис наближений до звичайної мови, тому навіть новачки легко розуміють, що відбувається в коді. У багатьох мовах програмування потрібно писати багато технічного коду, але Python дозволяє робити складні речі просто і читається як текст.

Python використовують у різних сферах:

- **Аналіз даних (Data Science)** — обробка великих обсягів інформації, статистика, графіки
- **Машинне навчання та штучний інтелект (AI/ML)** — одна з головних мов у цій сфері
- **Веброзробка** — створення сайтів і серверів (фреймворки Flask, Django)
- **Автоматизація** — написання скриптів для щоденних завдань
- **Ігри та візуалізація** — 2D/3D-графіка, робота з медіа
- **Освіта** — Python використовують для викладання інформатики в школах, а в університетах студенти освоюють різні числові методи.
- **Мобільні та десктоп-застосунки, робототехніка, IoT**

Як створювати програми на Python у OnlineGDB

Щоб не встановлювати нічого на комп’ютер, для початку ми будемо використовувати *онлайн середовище програмування* **OnlineGDB** (<https://www.onlinegdb.com>). Це зручний сайт, де можна писати й запускати Python-код прямо в браузері навіть з смартфона.

Як почати:

1. Перейди на сайт: https://www.onlinegdb.com/online_python_compiler
2. У верхньому меню переконайся, що вибрано Python як мову.
3. У робочому вікні напиши свій код. За замовчуванням в робочому вікні вже буде рядок коду `print("Hello, World!")`. З цієї фрази зазвичай починається вивчення нової мови програмування.
4. Натисни кнопку Run або F8, щоб запустити програму, і побачиш у нижній частині екрану результат: Hello, World!.

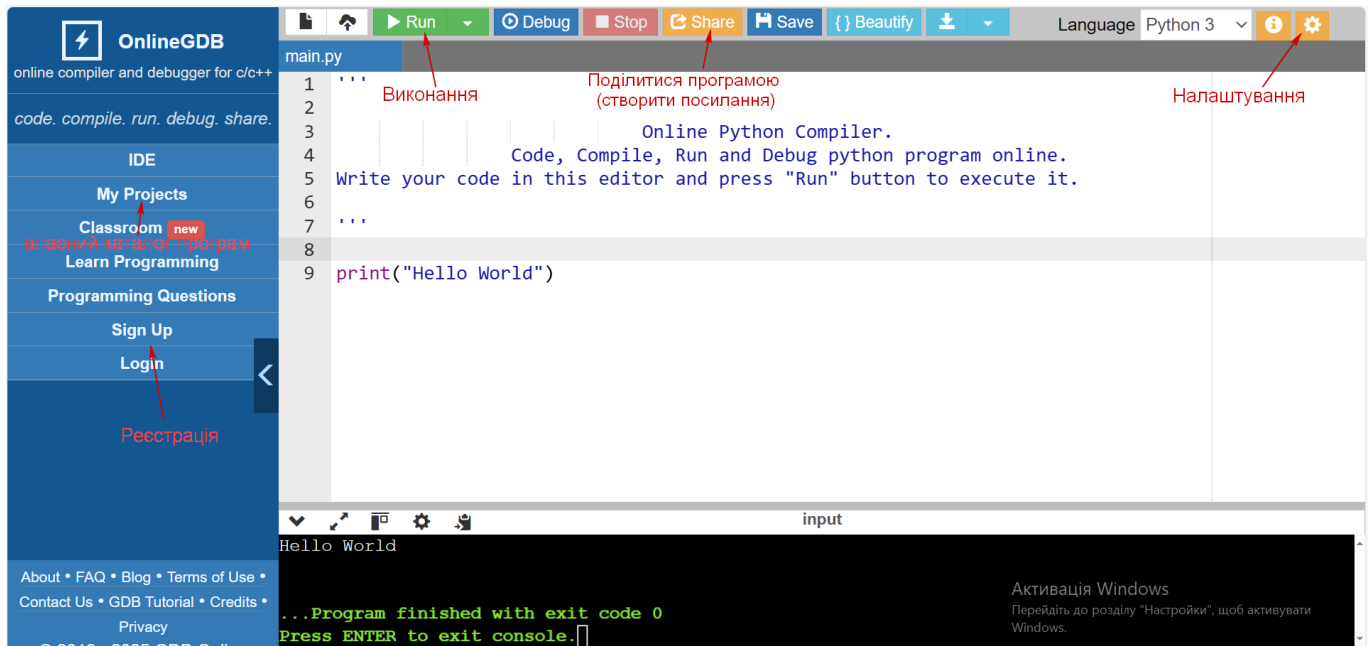


Рис. 1.1. Інтерфейс онлайн середовища onlineGDB.

Вітаю! Ти щойно написав(-ла) свою першу програму на Python.

Переваги онлайн (хмарного) середовища OnlineGDB:

- Можна працювати навіть з планшета або телефону
- Підтримує багато мов програмування, не лише Python
- Є можливість зберігати проекти та ділитися посиланням на копію твоєї програми
- Зручний інтерфейс із підсвічуванням синтаксису і зручною покроковою відладкою коду

1.2. Змінні та типи даних

Змінні — це іменовані об'єкти, що використовуються для зберігання даних. У Python немає потреби заздалегідь вказувати тип змінної — він автоматично визначається під час присвоєння значення.

Присвоєння змінних

Щоб створити змінну, просто напиши її ім'я, знак рівності й значення:

```
x = 10
name = "Аня"
pi = 3.14
is_happy = True
y = pi * x
```

Знак `=` у Python (як і у більшості інших мов програмування) не означає *дорівнює*, як у математиці. Це **оператор присвоєння**. За його допомогою значення, що стоїть праворуч від знака `=`, присвоюється змінній, ім'я якої вказано ліворуч. Праворуч може бути вказано вираз, результат обчислення якого присвоюється змінній.

Коментарі

Коментарі — це рядки, які не виконуються програмою. Вони починаються зі знака решітки `#` для однорядкового коментаря. Для багаторядкових коментарів використовують потрібний знак `'''` на початку та в кінці коментаря:

```
x = 5  #  однорядковий коментар
''' Тут можна залишити багаторядковий коментар,
що пояснює роботу програми
'''
```

Часто програмісти використовують коментарі для тимчасового вимкнення окремого рядка або цілого блоку коду з виконання програми (наприклад, під час налагодження чи тестування).

Основні типи даних

`int` — цілі числа: 0, 1, 3, -7,

`float` — числа з плаваючою крапкою: 3.14, -0.5

`str` — рядки (текст): 'Привіт', 'Python', '23' (число в лапках є текстом)

`bool` — логічні значення: `True`, `False`

У Python тип `int` підтримує цілі числа довільної довжини. Це означає, що максимальний розмір цілого числа не обмежений наперед фіксованим діапазоном, як у багатьох інших мовах програмування (наприклад, у C чи Java). У Python значення типу `int` можуть зростати до дуже великих чисел, обмежених лише обсягом оперативної пам'яті комп'ютера.

Тип `float` у Python використовується для представлення дійсних чисел. Точність цих чисел в Python обмежена приблизно 15-17 значущими цифрами.

```
x = 0.1 + 0.2
print(x) # 0.30000000000000004
```

Для задання рядків у Python можна використовувати як одинарні лапки `'...'`, так і подвійні `"..."`. Обидва варіанти є еквівалентними. Вибір лапок часто зумовлений зручністю: якщо в рядку потрібно використати апостроф (одинарну лапку), зручно використовувати подвійні лапки:

```
text = "It's a sunny day"
```

А якщо потрібно вмістити подвійні лапки в рядок, можна скористатися альтернативним типом лапок або ескапуванням за допомогою знака `backslash`:

```
quote = 'He said: "Hello!"'
escaped = 'It\'s also possible to use backslashes.'
```

Складніші типи даних

Крім базових типів даних (`int`, `float`, `bool`, `str`), у Python існують складніші типи, які дозволяють працювати з колекціями, структурами даних та множинами. Вони широко використовуються у практичному програмуванні:

`list` — список: впорядкована змінювана послідовність елементів. Дозволяє дублювати.

`tuple` — кортеж: впорядкована незмінювана послідовність. Підходить для зберігання фіксованих наборів даних.

`dict` — словник: колекція пар «ключ — значення», де ключі унікальні.

`set` — множина: неупорядкована колекція унікальних елементів. Не допускає дублікатів.

`frozenset` — незмінювана множина: як `set`, але не підтримує зміну після створення.

`range` — діапазон: представляє послідовність чисел, часто використовується в циклах.

`bytes`, `bytearray` — послідовності байтів, зручні для роботи з двійковими файлами або мережевими даними.

`NoneType` — спеціальний тип, який має єдине значення `None` (позначає відсутність значення).

На основі базових числових типів даних `int` та `float` створено тип `complex`, який дозволяє використовувати комплексні числа у вигляді `a + bj`, де `a` — дійсна частина, а `b` — уявна.

Правила створення імен змінних у Python

Імена змінних у Python повинні відповідати певним правилам, щоб бути синтаксично коректними.

- Ім'я змінної повинно починатися з літери латинського алфавіту (`a-z`, `A-Z`) або символу підкреслення `_`.
- Усі наступні символи можуть бути літерами, цифрами або підкресленням: `user1`, `data_set`, `value_2`.
- Імена чутливі до регістру: `Height` і `height` — це різні змінні.
- Ім'я не може містити пробілів або спеціальних символів: `! @ # $ %`, тощо.
- Ім'я не може збігатися з ключовим словом Python (наприклад, `for`, `if`, `class`, `None`).
- Не рекомендується використовувати символи `1`, `0`, `I` — вони легко плутаються з цифрами `1` або `0`.

Рекомендації щодо стилю (PEP 8):

Python має офіційний документ зі стилю коду — **PEP 8** (*Python Enhancement Proposal №8*). Він описує загальноприйняті правила оформлення коду, зокрема:

- Імена змінних пишуться малими літерами з підкресленням між словами: `user_name`, `total_sum`.
- Імена констант — великими літерами з підкресленням: `PI = 3.14`, `MAX_VALUE = 1000`.
- Назви класів використовують стиль **CamelCase** (перша літера кожного слова — велика): `Student`, `TemperatureSensor`.

1.3. Введення та виведення даних

Виведення на екран

Команда `print()` дозволяє виводити інформацію на екран. Вона може приймати один або кілька аргументів, розділених комами. Кожен аргумент буде перетворено на рядок (якщо це потрібно) і виведено у вказаному порядку. Між аргументами автоматично вставляється пробіл.

```
name = "Олена"  
x = 10  
pi = 3.14  
  
print("Мене звати", name)  
print("Сума:", x + pi)
```

Результат виконання:

```
Мене звати Олена  
Сума: 13.14
```

Команда `print()` також має параметри для точнішого керування виведенням:

`sep` — роздільник між аргументами (за замовчуванням — пробіл):

```
print("Python", "3.12", sep="-")
```

Результат:

```
Python-3.12
```

`end` — символ або рядок, який додається наприкінці виводу (за замовчуванням — символ нового рядка `\n`, який забезпечує наступний друк з нового рядка). Якщо вказати `end=' '`, то виведення буде в один рядок:

```
print("Привіт", end=" ")  
print("світ!")
```

Результат:

```
Привіт світ!
```

Форматований вивід (f-string та `str.format()`)

У Python існує кілька способів форматowanego виводу даних — це дозволяє керувати виглядом чисел, рядків, відступів тощо.

f-рядки (f-strings) — найсучасніший та рекомендований спосіб (Python 3.6+):

```
name = "Олена"  
age = 19  
print(f"Мене звати {name}, мені {age} років.")
```

Результат:

```
Мене звати Олена, мені 19 років.
```

Можна вказувати формат виведення чисел:

```
pi = 3.1415926  
n = 11**5  
print(f"pi = {pi:.2f}")  
print(f"11**5 = {n} = {n:.1e}")
```

Результат:

```
pi = 3.14
11**5 = 161051 = 1.6e+05
```

Метод `str.format()` — універсальний спосіб для форматування:

```
print("Ім'я: {}, зріст {:.2f} м.".format("Тарас", 1.823))
```

Результат:

```
Ім'я: Тарас, зріст 1.82 м.
```

У сучасному коді рекомендовано використовувати f-рядки за їхню зрозумілість і зручність.

Ввід даних від користувача

Функція `input()` дозволяє "запитати" користувача про щось і зчитати його відповідь з клавіатури.

```
name = input("Як тебе звати? ")
print("Привіт,", name)
```

Функція `input()` завжди повертає рядок. Щоб перетворити його у число, необхідно використовувати `int()` або `float()`:

```
a = int(input("Введи ціле число: "))
b = float(input("Введи десяткове число: "))
print("Добуток:", a * b)
```

Об'єднання рядків між собою та з числами

Перед виведенням часто буває зручно створити один рядок з декількох змінних. Наприклад, рядки можна об'єднувати (конкатенувати) за допомогою знака `+`:

```
first_name = "Анна"
last_name = "Петренко"
full_name = first_name + " " + last_name
print(full_name)
```

Щоб об'єднати рядок із числом, потрібно перетворити число на рядок за допомогою `str()`:

```
age = 20
print("Мені " + str(age) + " років")
```

Завдання для практики

1. Створи змінні різного типу та виведи їх на екран.
2. Зчитай два числа, уведені користувачем з клавіатури і виведи їхню суму, різницю й добуток.
3. Зчитай дані, введені з клавіатури: ім'я, рік народження та обчисли вік користувача.
4. Створи змінну з числом і об'єднай її з рядком у виводі на екран.
5. Придумай змінні, які описують тебе (ім'я, місто, рік народження, хобі), і виведи з них коротке повідомлення на кшталт: "Привіт! Мене звати ..., я з міста ..., мені ... років і я люблю ..."

1.4. Символьні рядки

Рядки (тип `str`) — це послідовності символів, що широко використовуються для зберігання та обробки тексту, введення даних з клавіатури, виведення повідомлень тощо.

Створення рядків

Рядки можна записувати в одинарних або подвійних лапках:

```
text1 = 'Привіт!'
text2 = "Python - класний"
```

Основні операції з рядками

- + — конкатенація (об'єднання рядків)
- * — повторення рядка
- in — перевірка входження
- len() — довжина рядка

```
greet = "Привіт"
print(greet + " світ")      # Привіт світ
print("-" * 10)             # -----
print("в" in greet)         # True
print(len(greet))           # 6
```

Методи рядків

Методом називають функцію, яка викликається для певного об'єкта, використовуючи запис через крапку після імені цього об'єкта.

Рядки мають багато зручних методів:

```
text = " Привіт, світе! "
print(text.strip())          # видаляє пробіли з країв
print(text.lower())          # все в нижній регістр
print(text.upper())          # все у верхній регістр
print(text.replace("і", "и")) # замінює 'і' на 'и'
print(text.find("св"))        # знаходить індекс входження
```

ASCII коди символів

Кожен символ має свій числовий код. Для перших 128 символів використовується стандарт ASCII (American Standard Code for Information Interchange), що включає:

- керівні символи (коди 0–31 і 127);
- спеціальні та розділові знаки (коди 32–47 та 123–126);
- цифри (коди 48–57);
- латинські великі та малі літери (коди 65–122).

Множина кодів ASCII є підмножиною більш загального стандарту `**Unicode**`, який охоплює практично всі символи сучасних мов і спеціальні знаки (математичні, технічні тощо). Отже, перші 128 кодів ASCII та Unicode повністю збігаються. У Python усі рядки зберігаються у форматі Unicode, тому можуть містити як ASCII-символи, так і літери кирилиці, арабської абетки, китайські ієрогліфи та інші символи.

У таблицях символів деякі коди не відповідають видимим символам, а є **керуючими**. Вони використовуються для керування відображенням або обробкою тексту. Наприклад:

`\t` — табуляція (Tab), код 9

`\n` — кінець рядка, код 10

`\r` — повернення каретки (код клавіші Enter), код 13

ESC — код клавіші Escape, код 27

Ці символи не мають графічного зображення, але відіграють важливу роль у форматуванні тексту та взаємодії з клавіатурою або терміналом.

Для роботи з кодами символів у Python можна використовувати функції:

- `ord(c)` — повертає Unicode символу `c`;
- `chr(n)` — повертає символ за його Unicode `n`.

Наприклад:

```
print(ord('A'))    # 65
print(chr(169))    # ©
```

Далі наведено таблицю кодів ASCII для символів, цифр і деяких спеціальних знаків.

ASCII-коди символів латинського алфавіту

Великі літери						Малі літери					
СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД
A	65	J	74	S	83	a	97	j	106	s	115
B	66	K	75	T	84	b	98	k	107	t	116
C	67	L	76	U	85	c	99	l	108	u	117
D	68	M	77	V	86	d	100	m	109	v	118
E	69	N	78	W	87	e	101	n	110	w	119
F	70	O	79	X	88	f	102	o	111	x	120
G	71	P	80	Y	89	g	103	p	112	y	121
H	72	Q	81	Z	90	h	104	q	113	z	122
I	73	R	82			i	105	r	114		

ASCII-коди спецсимволів та цифр

СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД	СИМВОЛ	КОД
<code>\t</code>	9	,	44	:	58	{	123
<code>\n</code>	10	-	45	;	59	}	125
space	32	.	46	<	60	~	126
!	33	/	47	=	61	Enter	13
"	34	0	48	>	62	ESC	27
#	35	1	49	?	63		
\$	36	2	50	@	64		
%	37	3	51	[	91		
&	38	4	52	\	92		
'	39	5	53	]	93		
(	40	6	54	^	94		
)	41	7	55	_	95		
*	42	8	56				
+	43	9	57				

Unicode символів українського алфавіту

Великі літери

СИМВ.	КОД	СИМВ.	КОД	СИМВ.	КОД
А	1040	І	1030	Т	1058
Б	1041	Ї	1031	У	1059
В	1042	Й	1049	Ф	1060
Г	1043	К	1050	Х	1061
Ґ	1168	Л	1051	Ц	1062
Д	1044	М	1052	Ч	1063
Е	1045	Н	1053	Ш	1064
Є	1028	О	1054	Щ	1065
Ж	1046	П	1055	Ь	1068
З	1047	Р	1056	Ю	1070
И	1048	С	1057	Я	1071

Малі літери

СИМВ.	КОД	СИМВ.	КОД	СИМВ.	КОД
а	1072	і	1110	т	1090
б	1073	ї	1111	у	1091
в	1074	й	1081	ф	1092
г	1075	к	1082	х	1093
ґ	1169	л	1083	ц	1094
д	1076	м	1084	ч	1095
е	1077	н	1085	ш	1096
є	1108	о	1086	щ	1097
ж	1078	п	1087	ь	1100
з	1079	р	1088	ю	1102
и	1080	с	1089	я	1103

Примітка. Базовий стандарт ASCII містить лише 128 символів (коди 0–127): керівні символи, латинські літери, цифри та основні розділові знаки. Пізніше з'явилися розширені версії ASCII, які використовували 8 біт (256 позицій). Однак для діапазону 128–255 не існувало єдиного стандарту: у різних регіонах застосовувалися різні таблиці символів (так звані *кодові сторінки*).

Для кирилических мов у середовищі Windows широко використовувалася кодова сторінка Windows-1251, яка дозволяла відображати кирилическі символи.

У сучасних системах переважає стандарт Unicode (зокрема кодування UTF-8), який охоплює практично всі писемності світу та замінює множину локальних кодувань.

Перекодування у Python. Якщо потрібно прочитати рядок у старому кодуванні (наприклад, Windows-1251) і перетворити його у стандартне представлення Unicode (яке використовується в Python 3), можна скористатися методами `encode()` та `decode()`. Аналогічно можна закодувати рядок у формат UTF-8 або розкодувати бінарний рядок, закодований у UTF-8:

```
# Перетворення байтів Windows-1251 у рядок Unicode
text1 = 'Привіт' # уведення рядка символів у стандартному представленні Unicode
t_cp1251=text1.encode('cp1251')
print(t_cp1251) # b'\xcf\xf0\xe8\xe2\xb3\xf2'

text2 = t_cp1251.decode('cp1251')

print(text2) # Привіт
print(ord(text2[0])) # 1055 - Unicode символ 'П'
```

```
t_utf8 = text2.encode('utf-8')
print(t_utf8)                # b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd1\x96\xd1\x82'

text3 = t_utf8.decode('utf-8')
print(text3)                  # Прусим
```

Змінна `t_cp1251`, яка виведена бінарним рядком (про це вказує символ `b` перед рядком), містить 6 байтів. Кожний байт записується у вигляді двох шістнадцяткових цифр, які слідують за командою `\x`. Шістнадцяткові числа `cf`, `f0`, `e8` ... відповідають кодам літер "П", "р", "и", ... в кодовій таблиці `cp1251`. Легко перекодувати ці числа в десятковий формат ($cf_{16} = 207$, $f0_{16} = 240$, $e8_{16} = 232$, ...).

Перетворення Unicode у UTF-8 (і навпаки) здійснюється за однозначним алгоритмом, який ґрунтується на поданні символів у двійковій формі. Стандарт UTF-8 визначає, як саме байти формуються з бітів Unicode-коду символу. У Python ці перетворення виконуються автоматично за допомогою методів `encode('utf-8')` та `decode('utf-8')`.

Нижче наведено кодову таблицю `cp1251` в десятковому та шістнадцятковому вигляді а також Unicode для літер української абетки (верхнього та нижнього регістрів). З таблиці видно, що коди символів в декількох місцях не відповідають порядку в українській абетці для обох кодувань. Це означає, що сортування за кодами символів не відповідає алфавітному порядку. Це через символи Г, Є, Ї, І, і, г, є, ї (десяткові коди `CP1251` 165, 170, 175, 178 для верхнього регістру, та 179, 180, 186, 191 для нижнього регістру), коди яких знаходяться за межами основного діапазону кирилиці в `CP1251`.

Завдання для практики

1. Введи рядок з клавіатури. Виведи його довжину, перший і останній символи.
2. Запитай у користувача ім'я та привітайся з ним (наприклад: **Привіт, Анна!**).
3. Введи рядок. Виведи його у верхньому регістрі без пробілів на початку і в кінці.
4. Створи рядок і заміни в ньому всі голосні літери на знак `-`.
5. Перевір, чи слово "Python" міститься у введеному тексті.
6. Задано текст у вигляді масиву кодів символів в кодуванні `cp1251` потрібно розкодувати його: [194, 32, 50, 49, 32, 241, 242, 238, 235, 179, 242, 242, 179, 32, 243, 236, 179, 242, 232, 32, 239, 240, 238, 227, 240, 224, 236, 243, 226, 224, 242, 232, 32, 246, 229, 32, 255, 234, 32, 243, 32, 50, 48, 32, 241, 242, 238, 235, 179, 242, 242, 179, 32, 243, 236, 179, 242, 232, 32, 247, 232, 242, 224, 242, 232, 47, 239, 232, 241, 224, 242, 232]

Табл. 1.1. Великі літери українського алфавіту в cp1251

Літера	cp1251 Dec	cp1251 Hex	Unicode	Unicode Dec
А	192	C0	U+0410	1040
Б	193	C1	U+0411	1041
В	194	C2	U+0412	1042
Г	195	C3	U+0413	1043
Ґ	165	A5	U+0490	1168
Д	196	C4	U+0414	1044
Е	197	C5	U+0415	1045
Є	170	AA	U+0404	1028
Ж	198	C6	U+0416	1046
З	199	C7	U+0417	1047
И	200	C8	U+0418	1048
І	178	B2	U+0406	1030
Ї	175	AF	U+0407	1031
Й	201	C9	U+0419	1049
К	202	CA	U+041A	1050
Л	203	CB	U+041B	1051
М	204	CC	U+041C	1052
Н	205	CD	U+041D	1053
О	206	CE	U+041E	1054
П	207	CF	U+041F	1055
Р	208	D0	U+0420	1056
С	209	D1	U+0421	1057
Т	210	D2	U+0422	1058
У	211	D3	U+0423	1059
Ф	212	D4	U+0424	1060
Х	213	D5	U+0425	1061
Ц	214	D6	U+0426	1062
Ч	215	D7	U+0427	1063
Ш	216	D8	U+0428	1064
Щ	217	D9	U+0429	1065
Ь	218	DA	U+042C	1068
Ю	219	DB	U+042E	1070
Я	220	DC	U+042F	1071

Табл. 1.2. Малі літери українського алфавіту в cp1251

Літера	cp1251 Dec	cp1251 Hex	Unicode	Unicode Dec
а	224	E0	U+0430	1072
б	225	E1	U+0431	1073
в	226	E2	U+0432	1074
г	227	E3	U+0433	1075
ґ	180	B4	U+0491	1169
д	228	E4	U+0434	1076
е	229	E5	U+0435	1077
є	186	BA	U+0454	1108
ж	230	E6	U+0436	1078
з	231	E7	U+0437	1079
и	232	E8	U+0438	1080
і	179	B3	U+0456	1110
ї	191	BF	U+0457	1111
й	233	E9	U+0439	1081
к	234	EA	U+043A	1082
л	235	EB	U+043B	1083
м	236	EC	U+043C	1084
н	237	ED	U+043D	1085
о	238	EE	U+043E	1086
п	239	EF	U+043F	1087
р	240	F0	U+0440	1088
с	241	F1	U+0441	1089
т	242	F2	U+0442	1090
у	243	F3	U+0443	1091
ф	244	F4	U+0444	1092
х	245	F5	U+0445	1093
ц	246	F6	U+0446	1094
ч	247	F7	U+0447	1095
ш	248	F8	U+0448	1096
щ	249	F9	U+0449	1097
ь	250	FA	U+044C	1100
ю	251	FB	U+044E	1102
я	252	FC	U+044F	1103

1.5. Оператори

Оператори в Python — це спеціальні символи або ключові слова, які дозволяють виконувати дії над змінними та значеннями. Усі оператори можна умовно поділити на кілька груп.

Математичні оператори в Python

У мові програмування Python використовуються стандартні математичні оператори для виконання арифметичних обчислень. Вони працюють як з цілими числами `int`, так і з дійсними числами `float`.

Оператор	Призначення	Приклад	Результат
<code>+</code>	Додавання	<code>2 + 5</code>	7
<code>-</code>	Віднімання	<code>10 - 3</code>	7
<code>*</code>	Множення	<code>3 * 6</code>	18
<code>/</code>	Ділення (результат <code>float</code>)	<code>5 / 2</code>	2.5
<code>//</code>	Цілочисельне ділення	<code>8 // 3</code> <code>-8 // 3</code>	2 -3
<code>%</code>	Остача від ділення	<code>8 % 3</code> <code>-7 % 3</code>	2 2
<code>**</code>	Піднесення до степеня	<code>2 ** 3</code> <code>9 ** 0.5</code>	8 3.0

Примітки:

- Оператор `/` завжди повертає дійсне число типу `float`, навіть якщо ділення націло.
- Операції `//`, `%` з від'ємними числами підпорядковуються математичному правилу округлення вниз.
- Піднесення до дробового степеня, наприклад `4 ** 0.5`, дозволяє знаходити корені.

Оператор присвоєння та комбінований оператор присвоєння

Оператор присвоєння (`=`) використовується для збереження значення у змінній. Він не означає математичну рівність, а вказує інтерпретатору присвоїти праве значення лівому імені.

```
x = 5          # змінній x присвоєно значення 5
y = x + 2      # y отримає значення 7
```

Змінна зліва повинна бути допустимим ідентифікатором, а праворуч — будь-який вираз, результат якого буде обчислено й збережено.

Python також підтримує **скорочені форми присвоєння**, які поєднують арифметичну операцію з присвоєнням:

Оператор	Еквівалент	Приклад (при <code>a = 3</code>)	Присвоєне значення <code>a</code>
<code>+=</code>	<code>x = x + y</code>	<code>a += 3</code>	6
<code>-=</code>	<code>x = x - y</code>	<code>a -= 2</code>	1
<code>*=</code>	<code>x = x * y</code>	<code>a *= 4</code>	12
<code>/=</code>	<code>x = x / y</code>	<code>a /= 3</code>	1.0
<code>//=</code>	<code>x = x // y</code>	<code>a //= 2</code>	1
<code>%=</code>	<code>x = x % y</code>	<code>a %= 4</code>	3
<code>**=</code>	<code>x = x ** y</code>	<code>a **= 2</code>	9

Оператори порівняння (логічні умови)

Оператор	Опис	Приклад при x=5	Результат
==	дорівнює	print(x == 5)	True
!=	не дорівнює	print(x != 3)	True
>	більше	print(x > 2)	True
<	менше	print(x < 10)	True
>=	більше або дорівнює	print(x >= 6)	False
<=	менше або дорівнює	print(x <= 5)	True

Порівнювати можна не лише числа, а й інші типи даних — наприклад, рядки, списки, кортежі або множини. У разі порівняння рядків Python порівнює їх за порядком символів у таблиці Unicode, починаючи з першого символу зліва.

```
'A' < 'B' > 'C'      # True
'apple' < 'banana'   # True
```

Порівнювати можна лише значення об'єктів одного типу. Водночас порівняння значень різних типів (наприклад, числа і рядка) заборонено і призведе до помилки типу:

```
5 > '2'  # TypeError: '>' not supported between instances of 'int' and 'str'
```

Можна також створювати **ланцюжкові порівняння**:

```
1 != 2 <= 2 + 1 > 1.5  # True
```

Увага! Через наближене подання дійсних чисел у пам'яті комп'ютера результати порівнянь можуть бути неочікуваними:

```
print(0.1**3 == 0.001)  # False
```

Щоб уникнути подібних ситуацій, доцільно перевіряти близькість чисел із заданою точністю:

```
x = 0.1**3
y = 0.001
print(abs(x - y) < 1e-6)  # True
```

Логічні оператори

Окрім порівняльних операторів, Python підтримує логічні оператори, що дозволяють об'єднувати кілька умов в один вираз:

Оператор	Опис	Приклад при x=5	Результат
and	логічне «і»: обидві умови мають бути істинні	x > 2 and x < 10	True
or	логічне «або»: хоча б одна умова істинна	x < 2 or x == 5	True
not	заперечення	not(x == 5)	False

Таким чином, логічні вирази — це основа умовних конструкцій у Python, які дозволяють програмі приймати рішення під час виконання.

1.6. Умовні конструкції (if, else, elif)

В умовних конструкціях ми перевіряємо певну умову і виконуємо відповідну дію. Це дозволяє програмі приймати рішення.

Конструкція if

```
if <умова>:  
    <інструкція>  # дія, якщо умова істинна
```

Приклад:

```
age = 18  
if age >= 18:  
    print("Ти повнолітній")
```

Конструкція if-else

```
if <умова>:  
    <інструкція 1># якщо умова істинна  
else:  
    <інструкція 2># якщо умова хибна
```

Приклад:

```
temp = 15  
if temp > 20:  
    print("Жарко")  
else:  
    print("Прохолодно")
```

Конструкція if-elif-else

Якщо є багато умов — використовуємо elif (від “else if”):

```
if <умова 1>:  
    <інструкція 1>          # якщо умова 1 істинна  
elif <умова 2>:  
    <інструкція 2>          # якщо умова 2 істинна  
...                          # ( можуть бути ще декілька умов )  
else:  
    <інструкція 3>          # якщо всі попередні умови хибні
```

Приклад:

```
score = 85  
if score >= 90:  
    print("Оцінка: A")  
elif score >= 80:  
    print("Оцінка: B")  
elif score >= 70:  
    print("Оцінка: C")  
else:  
    print("Оцінка: F")
```

Зверни увагу:

Після if, elif, else ставиться двокрапка ":". У Python у конструкціях після двокрапки завжди слідує відступ (4 пробіли або табуляція). Якщо має виконуватись блок команд, то всі рядки цього блоку мають бути з відступом.

Тернарний оператор if

У Python існує так званий **тернарний оператор**, який по суті представляє видозмінений оператор `if else`.

Уявімо, що у вас є код:

```
if a < b:
    min = a
else:
    min = b
```

Щоб такий простий код не займав багато місця, для нього придумали компактний запис. Він виглядає так:

```
min = a if a < b else b
```

Загальний вигляд тернарного оператора можна описати у формі:

```
змінна = значення1 if умова else значення2
```

Це означає: якщо умова істинна, то змінній присвоюється `значення1`, інакше — `значення2`.

Зверніть увагу:

- Двокрапка після умови `if` не ставиться.
- Термін “тернарний” означає “той, що має три частини”, бо конструкція включає три елементи: умову, значення при істинності та значення при хибності.
- Тернарний оператор часто використовується для спрощення умовних присвоєнь у виразах.

Завдання для практики

1. Запроси у користувача вік. Якщо йому більше або дорівнює 18 — виведи “Доступ дозволено”, інакше — “Доступ заборонено”.
2. Запроси у користувача число. Виведи, чи воно додатне, від’ємне чи дорівнює нулю.
3. Запитай у користувача оцінку (0–100) і виведи відповідну літерну оцінку (A, B, C, F) за шкалою, як у прикладі вище.
4. Створи просту гру: користувач вводить пароль, і якщо він правильний — виводиться привітання, інакше — повідомлення про помилку.

1.7. Цикли (for, while)

Цикли дозволяють багаторазово виконувати один і той самий блок коду. Вони особливо зручні, коли потрібно щось повторити кілька разів. Цикли дозволяють багаторазово виконувати один і той самий блок коду.

Одне виконання тіла циклу називається *ітерацією*, а змінну, що змінюється на кожному кроці циклу та визначає порядок перебору елементів або номер поточної ітерації називають *ітераційною змінною*, *ітератором* або *лічильником*. *Ітерабельними* називають об’єкти, які мають внутрішню структуру, складаються з окремих елементів і дозволяють перебір цих елементів у циклі. Такі об’єкти реалізують спеціальний механізм доступу до елементів один за одним (наприклад, через ітератор). У Python до ітерабельних належать текстові рядки, списки, кортежі, множини, словники тощо. Цикли поділяють на два основні типи:

- **Цикли з наперед заданою кількістю ітерацій** — коли заздалегідь відомо, скільки разів виконається тіло циклу. До цього типу належать цикли `for`.

- **Цикли з невизначеною кількістю ітерацій** — виконуються доти, доки виконується певна умова. Такі цикли застосовуються, коли кількість повторень заздалегідь невідома (цикл `while`).

Цикл `for`

Цикл `for` використовується для перебору елементів послідовностей (наприклад, списків або рядків).

Приклад 1: Виведення чисел від 1 до 5

```
for i in range(1, 6):  
    print(i)
```

Функція `range(1, 6)` повертає ітерабельний об'єкт типу `range`, елементами якого є цілі числа від 1 до 6 (верхня межа не включається). Функція `range(8)` повертає послідовність від 0 до 8 (не включаючи 8). Виключення правої межі послідовності дозволяє легко порахувати кількість ітерацій: це 5 у першому прикладі та 8 у другому.

Приклад 2: Виведення кожної літери рядка

```
for letter in "Python":  
    print(letter)
```

Цикл `while`

Цикл `while` виконується, доки умова істинна.

Приклад 1: Цикл з лічильником

```
count = 0  
while count < 5:  
    print("Лічильник:", count)  
    count += 1
```

Увага: Якщо забути збільшувати значення змінної `count` всередині циклу, цикл може стати нескінченним!

Приклад 2: Цикл з невідомою кількістю ітерацій

```
eps = 1  
while eps > 0.05:  
    print("Лічильник:", count)  
    eps = eps/2    # або eps /= 2
```

Оператор `break` і `continue`

`break` — зупиняє виконання циклу повністю.

`continue` — переходить до наступної ітерації, пропускаючи залишок коду в циклі при поточній ітерації.

Приклад 1: Зупинка на числі 3 (`break`)

```
for i in range(1, 6):  
    if i == 3:  
        break  
    print(i)
```

Приклад 2: Пропуск числа 3 (`continue`)

```
for i in range(1, 6):  
    if i == 3:  
        continue  
    print(i)
```

Приклад 3: Зупинка нескінченного циклу при правильному паролі

```
while True:  
    password = input("Введи пароль: ")  
    if password == "1234":  
        print("Доступ дозволено")  
        break  
    print("Неправильний пароль. Спробуй ще раз.")
```

Коментар: Цей цикл буде працювати вічно, доки не буде введено правильний пароль. Коли користувач введе "1234" спрацює `break`, і цикл завершиться.

Приклад 4: Пропуск непарних чисел

```
number = 0  
while number < 10:  
    number += 1  
    if number % 2 != 0:  
        continue  
    print("Парне число:", number)
```

Коментар: Якщо число непарне, виконується `continue`, і `print()` пропускається. У результаті виводяться лише парні числа.

Зверни увагу:

Блок команд, що виконуються в тілі циклу, повинен мати однаковий відступ (зазвичай 4 пробіли). Якщо в середині циклу використовуються умовні конструкції (`if`, `elif`, `else`), то їхні внутрішні команди отримують додатковий відступ. Так само відділяються блоки коду у вкладених циклах. Таким чином Python розуміє, які саме рядки належать до якого блоку коду.

Завдання для практики

1. Виведи на екран числа від 10 до 1 у зворотному порядку.
2. Попроси користувача ввести 5 чисел, обчисли їх суму.
3. Запитай у користувача пароль. Якщо пароль неправильний — повторюй запитувати, доки не введе один з наступних паролів: "12345" чи "54321".
4. Виведи всі парні числа від 1 до 20 (використовуй `continue` для пропуску непарних).

1.8. Структури даних `list`, `tuple`, `set`, `dict`

У Python існує кілька вбудованих структур даних, які дозволяють зручно зберігати та обробляти групи значень. Основні з них: списки, кортежі, множини та словники. Всі вони ітерабельні об'єкти і для їх поелементної обробки використовують цикли.

Списки (list)

Списки (list) — це змінювані колекції елементів, що зберігаються у фіксованому порядку. Це дозволяє звертатись до елементів за їхнім індексом, який приймає значення від 0 до $n - 1$, де n — кількість елементів у списку.

Найпростіший спосіб створити список — це використати оператор присвоєння `=` та вказати елементи списку у квадратних дужках, розділяючи їх комами:

```
numbers = [1, 2, 3, 4]
names = ["Анна", "Богдан", "Юля"]
mixed = [1, "Привіт", 3.14, True]
```

До елементів створених списків можна звертатись за їх індексом. Це також дозволяє змінювати значення елементів за допомогою оператора присвоєння:

```
print(numbers[0] + numbers[3])      # Виведе: 5
numbers[2] = 99                     # Зміна елемента
print(numbers)                      # [1, 2, 99, 4]
```

Списки можуть бути аргументами функцій, які виконують певні розрахунки, але не змінюють самі списки. Крім того, до списків можна застосовувати методи — вони записуються через крапку після імені списку. Більшість методів не повертають результату, проте змінюють сам список безпосередньо.

Табл. 1.3. Функції та методи для роботи зі списками

Назва	Опис	Змінює список?
Функції		
<code>len(список)</code>	Повертає кількість елементів у списку.	Ні
<code>sum(список)</code>	Обчислює суму всіх чисел у списку.	Ні
<code>min(список)</code>	Повертає найменший елемент.	Ні
<code>max(список)</code>	Повертає найбільший елемент.	Ні
<code>sorted(список)</code>	Повертає новий список із відсортованими елементами.	Ні
<code>list(об'єкт)</code>	Повертає список, створений з ітерабельного об'єкта (наприклад, рядка, кортежа, множини, об'єкта <code>range</code> та ін.).	Ні
Методи списків		
<code>append(x)</code>	Додає елемент <code>x</code> в кінець списку.	Так
<code>copy()</code>	Створює і повертає копію списку.	Ні
<code>insert(i, x)</code>	Вставляє елемент <code>x</code> на позицію <code>i</code> .	Так
<code>remove(x)</code>	Видаляє перше входження елемента <code>x</code> .	Так
<code>pop(i)</code>	Видаляє елемент за індексом <code>i</code> (останній, якщо не вказано).	Так
<code>sort()</code>	Сортує список на місці.	Так
<code>reverse()</code>	Змінює порядок елементів на протилежний.	Так
<code>clear()</code>	Видаляє всі елементи зі списку.	Так
<code>count(x)</code>	Повертає кількість входжень елемента <code>x</code> .	Ні
<code>index(x)</code>	Повертає індекс першого входження <code>x</code> .	Ні

Приклади використання функцій і методів списків

Нижче наведено приклади, які демонструють різницю між функціями (не змінюють список) та методами (змінюють сам список).

```
numbers = [3, 1, 4, 1, 5, 9]
```

```
# Функції
print(len(numbers))      # 6 - довжина списку
print(sum(numbers))      # 23 - сума всіх елементів
print(min(numbers))      # 1 - мінімальне значення
print(max(numbers))      # 9 - максимальне значення
print(sorted(numbers))   # [1, 1, 3, 4, 5, 9] - новий відсортований список
print(numbers)           # [3, 1, 4, 1, 5, 9] - оригінальний список не змінився

# Метод list() для створення списку з рядка
letters = list("привіт")
print(letters)           # ['п', 'р', 'и', 'в', 'і', 'т']

# Методи списків
numbers.append(10)        # додаємо елемент у кінець
numbers.sort()            # сортуємо список на місці
print(numbers)           # [1, 1, 3, 4, 5, 9, 10]

fruits = ["яблуко", "банан", "вишня"]
my_fruits = fruits.copy() # створюємо копію списку
my_fruits.remove("банан") # видаляємо елемент
my_fruits.reverse()       # змінюємо порядок
print(my_fruits)          # ['вишня', 'яблуко']
print(fruits)             # ['яблуко', 'банан', 'вишня']
```

Як видно з прикладів:

- Функції, як-от `len()`, `sum()`, `sorted()` — не змінюють список, а повертають результат.
- Методи, як-от `append()`, `sort()`, `remove()` — змінюють список на місці та не повертають значення.
- Метод `copy()` — створює незалежну копію списку.

Зверніть увагу! Копіювання списку за допомогою оператора присвоєння створює не нову копію, а новий ідентифікатор, що посиляється на той самий об'єкт у пам'яті. Отже, зміни в одному списку будуть автоматично відображатися і в іншому, оскільки обидва ідентифікатори вказують на один і той самий список. Це видно з прикладу:

```
a = [1, 2, 3]
b = a          # b - це не новий список, а посилання на той самий об'єкт
b[0] = 99
print(a)       # [99, 2, 3] - зміни відобразились і в списку a
print(b)       # [99, 2, 3]
```

Проблему копіювання списку вирішує метод `copy()`, але це лише для одномірних списків, а у випадку вкладених списків незалежна копія не вийде. У такому випадку потрібно робити поелементну копію списку, або користуватися функцією `deepcopy()` з модуля `copy`.

Крім функцій та методів до списків можна використовувати оператори.

Оператори для списків у Python

Оператори		
Оператор	Приклад	Дія
del	del my_list[i]	Видаляє елемент за індексом або зріз
in	if 3 in my_list:	Перевіряє, чи є елемент у списку
not in	if 10 not in my_list:	Перевіряє, чи елемент відсутній
+	[1, 2] + [3, 4]	Конкатенація (об'єднання списків)
*	[0] * 5	Повторення списку (створює копії елементів)
[]	my_list[2]	Доступ до елемента за індексом
[:]	my_list[1:4]	Зрізи. Створює підсписок (копію частини списку)

```
numbers = [1, 2, 3, 4, 5]
```

```
# Видалення елемента
```

```
del numbers[2]
```

```
print(numbers)  # [1, 2, 4, 5]
```

```
# Перевірка наявності
```

```
print(3 in numbers)      # True
```

```
print(10 not in numbers) # True
```

```
# Конкатенація
```

```
print(numbers + [6, 7]) # [1, 2, 4, 5, 6, 7]
```

```
# Повторення
```

```
print([1, 2] * 3)        # [1, 2, 1, 2, 1, 2]
```

```
# Індексція та зрізи
```

```
print(numbers[0])        # 1
```

```
print(numbers[1:3])      # [2, 4]
```

Кортежі (tuple)

Кортежі (tuple) — це незмінювані послідовності елементів. Вони схожі на списки, але після створення їх вміст не можна змінювати: не можна додавати, видаляти чи змінювати елементи. Завдяки своїй незмінності, кортежі зручні для зберігання даних, які мають залишатися сталими протягом виконання програми. Крім цього, кортежі обробляються швидше за списки, що може бути важливо для оптимізації продуктивності в великих обчисленнях.

Найпростіший спосіб створити кортеж — це перерахувати елементи в круглих дужках через кому:

```
point = (3, 4)
```

```
colors = ("червоний", "жовтий", "зелений", "синій")
```

```
mixed = (1, "текст", 3.14, True)
```

```
empty = ()          # порожній кортеж
```

```
single = (42,)      # кортеж з одним елементом
```

Зверніть увагу: для створення кортежу з одним елементом необхідно після значення ставити кому, інакше це буде просто вираз - звичайне число в дужках. До елементів кортежу можна звертатись за індексом, але змінити їх неможливо:

```
print(colors[3]+' '+colors[1])  # Виведе: "синій жовтий"
```

```
colors[1] = "білий" # Помилка! Кортеж незмінюваний об'єкт!
```

У разі потреби змін у кортежі, можна створити новий кортеж з врахуванням потрібних змін.

Табл. 1.4. Функції та методи для роботи з кортежами

Функції	
<code>len(кортеж)</code>	Повертає кількість елементів.
<code>min(кортеж)</code>	Повертає мінімальний елемент (якщо елементи можна порівняти).
<code>max(кортеж)</code>	Повертає максимальний елемент.
<code>sum(кортеж)</code>	Обчислює суму всіх числових елементів.
<code>sorted(кортеж)</code>	Повертає відсортований список, не змінюючи кортеж.
<code>tuple(об'єкт)</code>	Створює кортеж з ітерабельного об'єкта (наприклад, рядка або списку).
Методи кортежів	
<code>count(x)</code>	Повертає кількість входжень значення <code>x</code> у кортежі.
<code>index(x)</code>	Повертає індекс першого входження значення <code>x</code> .

```
coordinates = (1, 2, 3, 2)
print(len(coordinates)) # 4
print(coordinates.count(2)) # 2
print(coordinates.index(3)) # 2
```

Множини (set)

Множини (`set`) — це неупорядковані колекції унікальних елементів. На відміну від списків і кортежів, елементи множини не мають фіксованої позиції, тому звертатися до них за індексом не можна. Множини не допускають повторів, що робить їх зручними для перевірки унікальності, а також для виконання типових операцій теорії множин — об'єднання, перетину, різниці тощо.

```
a = {1, 2, 3}
b = set([2, 3, 4])
print(a | b) # об'єднання: {1, 2, 3, 4}
print(a & b) # перетин: {2, 3}
print(a - b) # різниця: {1}
```

Табл. 1.5. Функції та методи для роботи з множинами

Назва	Опис	Змінює множину?
Функції		
<code>len(множина)</code>	Повертає кількість елементів у множині.	Ні
<code>set(об'єкт)</code>	Створює множину з ітерабельного об'єкта.	Ні
Методи		
<code>add(x)</code>	Додає елемент <code>x</code> до множини.	Так
<code>remove(x)</code>	Видаляє елемент <code>x</code> (викликає помилку, якщо елемент відсутній).	Так
<code>discard(x)</code>	Видаляє елемент <code>x</code> , якщо він є (без помилки, якщо відсутній).	Так
<code>clear()</code>	Видаляє всі елементи з множини.	Так
<code>union(інша_множина)</code>	Повертає нову множину — об'єднання двох множин.	Ні
<code>intersection(інша_множина)</code>	Повертає нову множину — спільні елементи.	Ні
<code>difference(інша_множина)</code>	Повертає нову множину — елементи, які є в першій, але відсутні в другій.	Ні

```

friends = {"Оля", "Іван", "Марія"}
friends.add("Тарас")
friends.discard("Іван")
print(friends)      # {'Оля', 'Тарас', 'Марія'}

```

Словники (dict)

Словники (dict) — це колекції пар *ключ: значення*, де ключі унікальні. Вони дозволяють швидко знаходити значення за ключем.

```

student = {"ім'я": "Анна", "курс": 2, "вік": 20}
print(student["курс"])      # 2
student["курс"] = 3         # зміна значення
student["спеціальність"] = "CS" # додавання пари

```

Табл. 1.6. Методи словників

Назва	Опис	Змінює словник?
<code>dict[ключ]</code>	Доступ до значення за ключем або присвоєння нового значення.	Так
<code>get(ключ)</code>	Повертає значення за ключем або <code>None</code> , якщо ключ відсутній.	Ні
<code>keys()</code>	Повертає об'єкт-колекцію всіх ключів словника.	Ні
<code>values()</code>	Повертає об'єкт-колекцію всіх значень словника.	Ні
<code>items()</code>	Повертає об'єкт-колекцію пар ключ:значення.	Ні
<code>pop(ключ)</code>	Видаляє пару за ключем і повертає її значення.	Так
<code>update(інший)</code>	Додає або оновлює пари зі словника або з ітерованого набору пар.	Так
<code>clear()</code>	Видаляє всі елементи словника.	Так

```
info = {"автор": "Бойченко І.В.", "назва": "Книга", "рік": 2020}
print(info.get("рік")) # 2020
for key, value in info.items():
    print(key, ":", value) # виводить усі пари ключ:значення
```

1.9. Списки списків (матриці)

Списки можуть містити інші списки як свої елементи. Таку структуру часто називають матрицею або двовимірним списком. Вона зручна для представлення таблиць, координат, сіток тощо. Для доступу до елементів матриці використовують подвійний індекс: спочатку індекс рядка, потім — стовпчика.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[0][1]) # 2 - елемент з 1-го рядка, 2-го стовпчика
```

Матриці зручно обробляти за допомогою вкладених циклів.

```
for row in matrix:
    for element in row:
        print(element, end=" ") # виведення елементів рядка, розділених пробілом
    print()                    # перехід на наступний рядок
```

Копіювання матриць

Оскільки матриця є списком списків, важливо правильно створювати її копії. Звичайне копіювання (наприклад, через оператор `=`, метод `copy()` або зріз `[:]`) створює лише поверхневу копію:

```
original = [[1, 2], [3, 4]]
shallow = original.copy()
shallow[0][0] = 99
print(original) # [[99, 2], [3, 4]] - оригінал змінився!
```

Щоб створити незалежну копію матриці, потрібно скористатись модулем `copy` та методом `deepcopy()`:

```
import copy
original = [[1, 2], [3, 4]]
deep = copy.deepcopy(original)
deep[0][0] = 99
print(original) # [[1, 2], [3, 4]] - оригінал не змінився
```

1.10. Генератори списків (list comprehensions)

Генератори списків — це зручний і компактний спосіб створити список за допомогою циклу `for` у одному рядку.

Синтаксис:

[вираз `for` елемент `in` колекція `if` умова]

Приклади:

1. Квадрати чисел від 0 до 9:

```
squares = [x**2 for x in range(10)]  
print(squares) # [0, 1, 4, 9, ..., 81]
```

2. Тільки парні числа:

```
evens = [x for x in range(10) if x % 2 == 0]  
print(evens) # [0, 2, 4, 6, 8]
```

3. Довжини слів у списку:

```
words = ["яблуко", "банан", "ківі"]  
lengths = [len(word) for word in words]  
print(lengths) # [6, 5, 4]
```

4. Транспонування матриці:

```
matrix = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]  
transposed = [[matrix[j][i] for j in range(len(matrix))] for i in range(len(matrix[0]))]
```

Використання тернарного оператора if в генераторах списків

Тернарний оператор часто використовують у **генераторах списків**, коли потрібно динамічно вибирати значення залежно від умови.

Наприклад, маємо список чисел, і хочемо створити новий список, у якому всі від'ємні числа замінено на нуль, а інші залишено без змін:

```
numbers = [-5, 3, 0, -2, 8]  
processed = [x if x > 0 else 0 for x in numbers]  
print(processed)
```

Результат виконання:

```
[0, 3, 0, 0, 8]
```

Пояснення. Вираз `x if x > 0 else 0` повертає:

- `x`, якщо `x > 0` — тобто число додатне;
- `0`, якщо умова хибна (тобто `x` нуль або від'ємне).

Це особливо зручно для створення компактного та виразного коду при обробці великих масивів даних.

Ще приклад. Перетворимо список чисел у список рядків: ‘парне’ або ‘непарне’.

```
numbers = [1, 2, 3, 4, 5]  
labels = ["парне" if x % 2 == 0 else "непарне" for x in numbers]  
print(labels)
```

Результат:

```
['непарне', 'парне', 'непарне', 'парне', 'непарне']
```

Завдання для практики

1. Створи список чисел від 1 до 20, які діляться на 3.
2. Створи список квадратів парних чисел від 1 до 10.
3. Є список слів. Створи новий список, де будуть тільки слова довжиною понад 4 символи.
4. Створи список з перших літер кожного слова у реченні.

1.11. Зрізи у структурах даних

Зрізи (slicing) — це спосіб отримати частину послідовності (списку, рядка, кортежа) за допомогою індексів.

послідовність[початок:кінець:крок]

Загальні правила:

- початок — індекс першого елемента, включно.
- кінець — індекс елемента, до якого робиться зріз (НЕ включається).
- крок — крок ітерації (може бути від'ємним).

Зрізи списків:

```
numbers = [10, 20, 30, 40, 50, 60]
print(numbers[1:4])    # [20, 30, 40]
print(numbers[:3])     # [10, 20, 30]
print(numbers[::2])    # [10, 30, 50]
print(numbers[::-1])   # [60, 50, 40, 30, 20, 10]
```

Зрізи рядків:

Рядки — це послідовності символів, тому підтримують зрізи аналогічно до списків.

```
s = "Програмування"
print(s[0:5])    # Progr
print(s[-1])     # я
print(s[::-1])   # янвевнаммаргорП
```

Зрізи кортежів:

Кортежі також підтримують зрізи, але результатом буде новий кортеж.

```
t = (1, 2, 3, 4, 5)
print(t[1:4])    # (2, 3, 4)
```

Зрізи в матрицях (списки списків):

Зрізи потрібно застосовувати на кожному рівні окремо.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(matrix[0][1:])    # [2, 3] - зріз другого та третього елемента 1-го рядка
print([row[0] for row in matrix]) # [1, 4, 7] - перший стовпчик
```

Завдання для практики

1. Створи список з п'яти улюблених страв. Виведи перший та останній елемент.
2. Створи кортеж з координатами точки (x, y). Виведи значення.
3. Об'єднай дві множини з різними числами та знайди перетин.
4. Створи словник з інформацією про книгу (назва, автор, рік). Зміни рік на 2025.
5. Пройдись циклом по словнику і виведи інформацію в форматі "ключ: значення".

1.12. Розпакування ітерабельних об'єктів

Python дозволяє зручно розпаковувати ітерабельні об'єкти — тобто одразу призначати значення кільком змінним з послідовності (наприклад, списку чи кортежу).

Присвоєння елементів ітерабельного об'єкта кільком змінним

```
a, b = [10, 20]
print(a, b)    # 10 20
```

```
name, surname = 'Олег', 'Іванов'
print(name, surname)
```

Використання зірочки *

Символ * дозволяє зібрати кілька значень у список при розпакуванні. Це зручно, коли кількість елементів може бути змінною:

```
a, *middle, b = [1, 2, 3, 4, 5]
print(a)          # 1
print(middle)     # [2, 3, 4]
print(b)          # 5
```

Можна також збирати «хвіст» списку:

```
main, *other = ['Олег', 'Марія', 'Іван']
print(main)    # Олег
print(other)   # ['Марія', 'Іван']
```

Розпакування в циклах

Якщо ітерабельний об'єкт містить кортежі, то їх можна розпаковувати безпосередньо в тілі циклу:

```
pair_list = [(1, 'Олег'), (2, 'Марія'), (3, 'Іван')]

for number, name in pair_list:
    print(number, name)
print(*pair_list[::-1])
```

Результат:

```
1 Олег
2 Марія
3 Іван
(3, 'Іван') (2, 'Марія') (1, 'Олег')
```

В цьому прикладі показано розпакування кортежів в циклі, а також розпакування списку в оберненому порядку за допомогою зрізу.

Функція enumerate()

Функція `enumerate()` повертає індекси та значення з послідовності — це теж приклад розпакування:

```
names = ['Олег', 'Марія', 'Іван']

for i, name in enumerate(names):
    print(i, name)
```

Результат:

```
0 Олег
1 Марія
2 Іван
```

За замовченням індексація починається з 0. Якщо потрібно змінити початкове значення індексу, використовують аргумент `start`:

```
for i, name in enumerate(names, start=1):
    print(i, name)
```

Розпакування вкладених структур

```
students = [(1, ('Олег', 'Іванов')), (2, ('Марія', 'Петренко'))]

for i, (name, surname) in students:
    print(i, name, surname)
```

Висновок: розпакування в Python — потужний та зручний механізм, який робить код компактнішим і зрозумілішим.

1.13. Функції у Python

Функції — це іменовані блоки коду, які можна викликати багаторазово. Вони допомагають робити програми компактнішими, зручнішими для читання і тестування.

Функція оголошується за допомогою ключового слова `def`, після чого йде її назва, дужки (в яких можуть бути аргументи) і двокрапка. Тіло функції має відступ.

```
def назва_функції(аргументи):
    # тіло функції
return результат
```

Приклад: функція для обчислення квадрата числа:

```
def square(x):
    return x * x

print(square(5)) # Виведе 25
```

Можна створювати функції без аргументів:

```
def hello():
    print("Привіт!")

hello() # Виклик функції
```

Зверни увагу:

- Імена функцій мають бути зрозумілими та писатися в нижньому регістрі.
- Команда `return` повертає результат із функції. Якщо її немає — функція повертає `None`.
- Після `def ...:` обов'язково ставиться відступ у тілі функції.

Аргументи за замовчуванням

Можна задавати значення параметрів за замовчуванням, які використовуються, якщо при виклику функції не вказано відповідний аргумент:

```
def greet(name="Гість"):
    print("Привіт,", name)

greet() # Виведе: Привіт, Гість
greet("Аліна") # Виведе: Привіт, Аліна
```

Лямбда-функції та іменовані аргументи

Лямбда-функції — це короткі анонімні функції, які створюються за допомогою ключового слова `lambda`. Вони зручні для простих обчислень, які не потребують багато коду.

Синтаксис:

<назва> = `lambda` <аргументи>: <вираз>

Приклади:

```
square = lambda x: x * x
print(square(4)) # Виведе 16

add = lambda a, b: a + b
print(add(2, 3)) # Виведе 5
```

Лямбда-функції часто використовуються в поєднанні з функціями `map()`, `filter()`, `sorted()` тощо.

1. Застосування лямбда-функції до кожного елементу масиву за допомогою функції `map()`

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared) # [1, 4, 9, 16, 25]
```

2. Фільтрування масиву з допомогою лямбда-функції

```
numbers = [1, 2, 3, 4, 5, 6]
even = list(filter(lambda x: x % 3 == 0, numbers))
print(even) # [3, 6]
```

3. Сортування масиву з допомогою лямбда-функції

```
words = ["яблуко", "груша", "ананас", "ківі"]
sorted_words = sorted(words, key=lambda word: len(word))
print(sorted_words) # ['ківі', 'груша', 'яблуко', 'ананас']
```

4. Сортування пар чисел за другим елементом

```
pairs = [(1, 3), (2, 2), (4, 1)]
sorted_pairs = sorted(pairs, key=lambda pair: pair[1])
print(sorted_pairs) # [(4, 1), (2, 2), (1, 3)]
```

Іменовані аргументи (ключові слова при виклику) дозволяють передавати значення аргументів у функцію явно за назвою параметра. Це покращує читабельність коду та дозволяє не дотримуватись порядку аргументів.

Приклад:

```
def greet(name, age):
    print(f"Привіт, {name}. Тобі {age} років.")

greet(age=20, name="Олег") # Порядок не має значення
```

Іменовані аргументи особливо корисні, коли в функції багато параметрів, і не всі мають значення за замовчуванням.

Завдання для практики

1. Напиши функцію `add(a, b)`, яка повертає суму двох чисел.
2. Напиши функцію, яка перевіряє, чи число є парним.
3. Створи функцію, яка приймає ім'я користувача і вітає його.
4. Напиши функцію, яка обчислює факторіал числа.
5. Створи функцію, яка підносить число до квадрату, але значення аргументу за замовчуванням дорівнює 2.

1.14. Робота з файлами

Python дозволяє зручно зчитувати дані з файлів, а також зберігати їх у файлах.

Запис у файл

Щоб записати дані у файл, його відкривають в одному із режимів запису: "w" (write — запис), "a" (append — додати в кінець) або "x" (створити новий файл). Наприклад:

```
with open("output.txt", "w", encoding="utf-8") as file:
    file.write("Це перший рядок.\n")
    file.write("Це другий рядок.\n")
```

Ключова конструкція `with ... as ...` гарантує автоматичне закриття файлу після завершення виконання блоку команд, що розташовані з відступом після цього рядка.

Читання з файлу

Щоб зчитати весь вміст файлу як рядок:

```
with open("output.txt", "r", encoding="utf-8") as file:
    text = file.read()
    print(text)
```

Читання рядків по черзі:

```
with open("output.txt", "r", encoding="utf-8") as file:
    for line in file:
        print("Рядок:", line.strip())
```

Читання окремих рядків або списком

Зчитування першого та другого рядка:

```
with open("output_surnames.txt", "r", encoding="utf-8") as file:
    first_line = file.readline()
    print("Перший рядок:", first_line)

    second_line = file.readline()
    print("Другий рядок:", second_line)
```

Метод `readline()` читає один рядок і переміщує "вказівник читання" на початок наступного. Отже, кілька послідовних викликів `readline()` зчитують наступні рядки один за одним. Метод `readlines()` дозволяє зчитувати всі рядки у список:

```
with open("output.txt", "r", encoding="utf-8") as file:
    lines = file.readlines()
    print(lines)
```

Примітка: Зверни увагу, що назви методів `readlines()` та `readline()` відрізняються лише однією літерою.

Режими відкриття файлу

Режим	Опис
"r"	Читання. Файл має існувати.
"w"	Запис. Якщо файл існує — його вміст буде стерто.
"a"	Дозапис у кінець файлу без стирання наявних даних.
"x"	Створення нового файлу. Якщо файл вже існує — виникне помилка.
"rb"	Читання у двійковому (бінарному) режимі.
"wb"	Запис у двійковому режимі.

Робота з CSV-файлами

CSV (Comma-Separated Values) — це текстовий файл, у якому дані зберігаються у вигляді таблиці, а значення в кожному рядку розділяються комами (або іншими роздільниками, наприклад, крапкою з комою). Цей формат часто використовується для збереження табличних даних. Перший рядок зазвичай містить заголовки стовпців. Для зручної роботи з такими файлами Python пропонує стандартний модуль `csv`.

```
import csv
ff = "students.csv"
with open(ff, "w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerow(["Ім'я", "Оцінка"])
    writer.writerow(["Олег", 95])
    writer.writerow(["Марія", 87])
print(f'Дані записано у файл {ff} ')

with open(ff, "r", encoding="utf-8") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
print(f'Файл {ff} зчитано')
```

Результат:

```
Дані записано у файл students.csv
["Ім'я", 'Оцінка']
['Олег', '95']
['Марія', '87']
Файл students.csv зчитано
```

Зчитування даних з txt-файла розміщеного на Google диску

Для прикладу на Google диску виставлено txt файл з куплетом Енеїди Котляревського

Зчитування txt файлу з Google Disk Текстовий файл відкритий для читання за посиланням

<https://drive.google.com/file/d/1UTa9-mCE4oQMP5SDE0KrG0-fM09XbXUZ/view?usp=sharing>

У цьому посиланні міститься file_ID, яке знаходиться між file/d/ та /view тобто

```
file_id = "1UTa9-mCE4oQMP5SDE0KrG0-fM09XbXUZ"
```

Для зчитування файлу з Інтернету за URL-адресою застосовується модуль `requests`.

```
import requests

file_id = "1UTa9-mCE4oQMP5SDE0KrG0-fM09XbXUZ"
url = f"https://drive.google.com/uc?export=download&id={file_id}"
response = requests.get(url)

if response.status_code == 200:
    text1 = response.content
    # text2 = response.content.decode('utf-8')
    text2 = response.text
else:
    print("Помилка завантаження:", response.status_code)

print(text1[:10])
print(text2[:25])
```

Результат:

```
b'\xd0\x95\xd0\xbd\xd0\xb5\xd0\xb9 \xd0'
Еней був парубок моторний
```

В результаті запиту за URL-адресою отримано об'єкт `response` до якого можна застосовувати методи для перевірки статусу запиту та доступу до завантажених даних у вигляді бінарного чи текстового рядка.

Обробка винятків при роботі з файлами

Під час відкриття файлів за допомогою стандартної функції `open` можлива ситуація, коли файл з вказаним іменем не існує або не може бути відкритий. Щоб уникнути виведення системного повідомлення про помилку, її можна перехопити та обробити самостійно за допомогою конструкції `try-except`:

```
try:
    with open("data.txt", "r", encoding="utf-8") as file:
        content = file.read()
except FileNotFoundError:
    print("Файл не знайдено.")
```

1.15. Імпорт модулів і використання бібліотек

Що таке модуль у Python

Програми на Python можуть складатися з одного файлу, але у більшості випадків великі проекти поділяють на частини — модулі. Модуль — це окремий файл з розширенням `.py`, який містить змінні, функції або класи, що можуть використовуватися в інших частинах програми.

Коли ви імпортуєте модуль, ви отримуєте доступ до його функцій та змінних. Звернення до елементів модуля здійснюється через **точкову нотацію**, за допомогою синтаксису:

```
<ім'я_модуля>.<ім'я_атрибута>
```

Приклад. Припустимо, у вас є файл `messages.py`, який містить прості текстові повідомлення:

```
welcome = "Ласкаво просимо"  
goodbye = "До побачення!"
```

а також функцію:

```
def greet(name):  
    return f"{welcome}, {name}!"
```

Тепер, у вашій основній програмі ви можете використати змінні та функцію з використанням імені модуля:

```
import messages  
  
print(messages.welcome)  
print(messages.goodbye)  
print(messages.greet("Олена"))
```

Якщо звернутися до `messages.welcome` без попереднього імпорту модуля, виникне помилка:

```
NameError: name 'messages' is not defined
```

Якщо ж спробувати звернутися до змінної `welcome` без вказівки назви модуля `messages`, також виникне помилка:

```
NameError: name 'welcome' is not defined
```

Способи імпорту модулів

Python підтримує кілька способів імпорту, кожен з яких має свої переваги та особливості.

1. Імпорт усього модуля

Найбільш стандартний підхід. Модуль імпортується повністю, і доступ до його вмісту здійснюється через точкову нотацію (як і в попередньому прикладі):

```
import math  
  
r = 5  
area = math.pi * r ** 2  
print("Площа круга:", area)
```

2. Імпорт конкретних об'єктів з модуля

Цей спосіб дозволяє отримати прямий доступ до вибраних атрибутів без вказування імені модуля:

```
from math import sqrt, pi, sin

print(sqrt(49))          # 7
print(f"{sin(pi):.3f}")  # 0.000
```

3. Імпорт усіх об'єктів модуля (не рекомендовано)

Команда

```
from math import *
```

імпортує всі доступні об'єкти з модуля `math` безпосередньо в простір імен програми. Це дозволяє використовувати функції та константи без вказання імені модуля, наприклад: `sqrt(16)` замість `math.sqrt(16)`.

Однак такий спосіб імпорту **не рекомендується**. Він може призвести до конфліктів імен, якщо у програмі або в інших модулях є змінні чи функції з такими самими назвами. Це ускладнює читання та відлагодження коду, особливо в великих проектах.

4. Імпорт з перейменуванням назви модуля

Використання короткого псевдоніма для довгих імен модулів дозволяє спростити назви функцій та констант при точковій нотації. Наприклад, модуль `NumPy` часто називають як `np`:

```
import numpy as np

ar = np.array([2, 4, 6, 8])  # створення масиву NumPy для використання швидких методів

print("Середнє значення:", np.mean(ar))    # 5
print("Сума елементів:", np.sum(ar))       # 20
```

5. Уникнення конфлікту імен

Уявімо, що у вас є два різні модулі `circle.py` та `square.py`, які обидва мають функцію `area()`.

Щоб уникнути плутанини, можна зробити так:

```
from circle import area as circle_area
from square import area as square_area

print(circle_area(5))  # площа круга
print(square_area(4))  # площа квадрата
```

Або через точкову нотацію:

```
import circle
import square

print(circle.area(5))
print(square.area(4))
```

Стандартна бібліотека Python

Python поставляється з великою кількістю стандартних модулів, які не потребують додаткової інсталяції. Ці модулі охоплюють широкий спектр завдань — від обробки дат і часу до роботи з файлами, числами, інтернетом та криптографією.

Щоб дізнатися, які об'єкти містить модуль, можна скористатися функціями:

```
import math

print(dir(math))    # список імен у модулі
help(math)          # документація
```

Нижче в таблиці наведемо список найуживаніших модулів стандартної бібліотеки Python.

Табл. 1.7. Поширені стандартні модулі Python

Модуль	Призначення
<code>math</code>	Математичні функції: корені, логарифми, тригонометрія, константи <code>pi</code> , <code>e</code> .
<code>random</code>	Генерація випадкових чисел, випадковий вибір елементів зі списку, перемішування послідовностей.
<code>datetime</code>	Дата і час: поточний час, різниця між датами, форматування.
<code>time</code>	Робота з часовими мітками, вимірювання тривалості, паузи у виконанні.
<code>os</code>	Доступ до операційної системи: робота з файлами, шляхами, середовищем.
<code>sys</code>	Інформація про систему: аргументи командного рядка, завершення програми, версія Python.
<code>statistics</code>	Статистичні функції: середнє, медіана, мода, дисперсія.
<code>json</code>	Обробка JSON: перетворення між Python-об'єктами та JSON-рядками.
<code>re</code>	Робота з регулярними виразами: пошук, заміна, перевірка шаблонів.
<code>csv</code>	Читання і запис CSV-файлів. Зручно для обміну табличними даними.
<code>pathlib</code>	Робота з файлами й шляхами в об'єктно-орієнтованому стилі.
<code>itertools</code>	Інструменти для ефективної роботи з ітераторами, комбінації, перестановки тощо.
<code>enum</code>	Створення переліків — обмежених наборів іменованих констант.
<code>shutil</code>	Операції над файлами й папками: копіювання, видалення, архівація.
<code>glob</code>	Пошук файлів за шаблоном, наприклад <code>*.txt</code> .

Розглянемо найуживаніші функції модуля `math`:

Основні функції модуля math

Функція	Опис
<code>sqrt(x)</code>	Квадратний корінь з x
<code>pow(x, y)</code>	Піднесення x до степеня y
<code>exp(x)</code>	Обчислює експоненту e^x
<code>log(x)</code>	Натуральний логарифм $\ln(x)$
<code>log10(x)</code>	Десятковий логарифм $\log_{10}(x)$
<code>log2(x)</code>	Двійковий логарифм $\log_2(x)$
<code>sin(x)</code>	Синус (аргумент у радіанах)
<code>cos(x)</code>	Косинус
<code>tan(x)</code>	Тангенс
<code>asin(x)</code>	Арксинус
<code>acos(x)</code>	Арккосинус
<code>atan(x)</code>	Арктангенс
<code>degrees(x)</code>	Переводить з радіан у градуси
<code>radians(x)</code>	Переводить з градусів у радіани
<code>pi</code>	Константа $\pi \approx 3.14159$
<code>e</code>	Константа $e \approx 2.71828$
<code>factorial(n)</code>	Факторіал натурального числа n
<code>fabs(x)</code>	Абсолютне значення (для дійсних чисел)
<code>ceil(x)</code>	Округлення вгору до найближчого цілого
<code>floor(x)</code>	Округлення вниз до меншого цілого
<code>trunc(x)</code>	Усікання дробової частини числа

Основні функції модуля random

Функція	Опис
<code>random()</code>	Генерує випадкове число з інтервалу $[0.0, 1.0)$
<code>randint(a, b)</code>	Генерує випадкове ціле число в інтервалі $[a, b]$ (включно з обома межами)
<code>uniform(a, b)</code>	Генерує випадкове число з плаваючою комою в інтервалі $[a, b]$
<code>choice(seq)</code>	Повертає випадковий елемент з послідовності <code>seq</code> (список, рядок, тощо)
<code>choices(seq, k=n)</code>	Повертає список з n випадкових елементів з <code>seq</code> (з повтореннями)
<code>sample(seq, k=n)</code>	Повертає список з n унікальних випадкових елементів з <code>seq</code> (без повторень)
<code>shuffle(seq)</code>	Перемішує елементи послідовності <code>seq</code> на місці
<code>seed(n)</code>	Встановлює початкове значення для генератора випадкових чисел (для відтворюваності результатів)

Висновки

Модулі — це основа масштабованості у Python. Вони дозволяють розділяти код на частини та повторно використовувати функції, які вже були реалізовані. Завдяки імпорту модулів із стандартної бібліотеки, Python надає потужні інструменти без необхідності встановлення додаткового програмного забезпечення. Модулі є важливим механізмом для організації коду в Python. Вони дають змогу структурувати програми, поділяючи їх на логічні частини, а також повторно використовувати готові функції й інструменти. Стандартна бібліотека Python містить багато корисних модулів, які можна підключити без додаткового встановлення — це значно розширює можливості мови.

1.16. Обробка винятків (помилки) у Python

Якісна програма повинна передбачати можливість виникнення помилок під час виконання та правильно реагувати на них, а не завершуватись аварійно з типовими повідомленнями інтерпретатора. Для реалізації цього у Python передбачено вбудований механізм обробки винятків (*exception handling*).

Що таке виняток?

Виняток — це подія, яка виникає під час виконання програми й порушує її нормальний хід. Якщо програма синтаксично правильна, але при виконанні трапляється помилка (наприклад, ділення на нуль або спроба доступу до неіснуючого файлу), Python генерує виняток. Якщо виняток не буде оброблений, програма завершиться з повідомленням про помилку.

Крім помилок, винятки використовують для:

- забезпечення завершення важливих дій (наприклад, закриття файлу);
- альтернативного розгалуження логіки;
- обробки подій, що можуть або не можуть трапитися.

Основні типи винятків

Інтерпретатор Python підтримує багато вбудованих типів винятків. Ось деякі з найпоширеніших:

- `FileNotFoundError` — файл не знайдено;
- `ImportError` — неможливо імпортувати модуль;
- `IndexError` — вихід за межі списку;
- `KeyError` — доступ до неіснуючого ключа у словнику;
- `NameError` — звернення до неоголошеної змінної;
- `TypeError` — операція між об'єктами несумісних типів;
- `ZeroDivisionError` — спроба ділення на нуль.

Синтаксис інструкції try

Для обробки винятків застосовують конструкцію `try-except`. Найпростіший синтаксис:

```
try:
    ... # код, який потенційно викликає виняток
except:
    ... # код, що виконується при виникненні винятку
```

Щоб обробляти винятки певного типу:

```
try:
    ... # код, який потенційно викликає виняток
except ValueError:
    print("Помилка перетворення типів.")
except ZeroDivisionError:
    print("Спроба ділення на нуль.")
except:
    print("Інша помилка.")
```

Блоки `else` та `finally`

Блок `else` виконується, якщо у блоці `try` не виникло винятку:

```
try:
    x = int(input("Введіть число: "))
except ValueError:
    print("Це не число.")
else:
    print(f"Ваше число: {x}")
```

Блок `finally` виконується завжди — незалежно від наявності винятку:

```
try:
    file = open("data.txt")
    data = file.read()
except FileNotFoundError:
    print("Файл не знайдено.")
finally:
    file.close()
```

Приклад: сума чисел з файлу

Нижче наведено приклад програми, яка читає числа з файлу й обчислює їхню суму. Програма обробляє помилки, пов'язані з відсутністю файлу та неправильними даними:

```
fileName = input("Введіть ім'я файлу: ")

try:
    with open(fileName) as f:
        lines = f.readlines()
        numbers = list(map(float, lines))
except FileNotFoundError:
    print('Файл не знайдено!')
except ValueError:
    print('Файл містить нечислові значення!')
else:
    print("Сума чисел у файлі:", sum(numbers))
```

Питання для самоперевірки

1. Що таке виняток і які його типи ви знаєте?
2. Що відбувається з програмою, якщо виняток не оброблено?
3. Для чого призначений блок `finally`?
4. Напишіть програму, яка читає з файлу два числа і виводить їхню частку, обробляючи всі можливі помилки.
5. Поясніть роль контекстного менеджера `with` у роботі з файлами. Як переписати код з `with open(...)` без використання контекстного менеджера?

Розділ 2

Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування (ООП) – одна з парадигм програмування, яка розглядає програму як множину "об'єктів що взаємодіють між собою. Основна ідея ООП полягає в моделюванні програмної системи, яка відображає структуру і поведінку об'єктів реального світу. Об'єкти в ООП мають стан (дані, атрибути) та поведінку (методи, функції), і взаємодіють між собою через обмін повідомленнями (виклики методів).

Об'єктно-орієнтований підхід особливо корисний у великих проєктах, де важливо забезпечити повторне використання коду, зручність супроводу, масштабованість і модульність. Саме тому ООП широко використовується в сучасних мовах програмування, включаючи Python.

У Python, як і в інших ООП-мовах, основними поняттями є:

- **Клас** – шаблон (або формальний опис) для створення об'єктів. Він визначає структуру і поведінку майбутніх екземплярів (об'єктів).
- **Екземпляр класу (об'єкт)** – конкретне втілення класу, з власними даними і доступом до методів класу.
- **Інкапсуляція** – приховування внутрішньої реалізації об'єкта від зовнішнього світу.
- **Успадкування** – механізм створення нових класів на основі вже існуючих.
- **Поліморфізм** – здатність об'єктів з різними типами поводитись однаково при використанні одного інтерфейсу.
- **Композиція** – побудова складних об'єктів шляхом включення об'єктів інших класів.

2.1. Клас, конструктор та екземпляр

Розглянемо створення простого класу, який моделює фізичне тіло:

```
class Body:
    def __init__(self, mass):
        self.mass = mass  # маса тіла, кг

    def print_info(self):
        print(f"Маса тіла: {self.mass} кг")
```

У цьому прикладі ми створюємо клас `Body`. Він містить спеціальний метод `__init__`, який є так званим **конструктором**.

Конструктор — це спеціальний метод, який автоматично викликається щоразу, коли створюється новий об'єкт цього класу. Його основне завдання — ініціалізувати (тобто задати початкові значення) атрибути об'єкта. В нашому випадку конструктор приймає один параметр `mass`

і присвоює його змінній `self.mass`. Команда `self.mass = mass` означає: "запам'ятай значення `mass` як атрибут цього об'єкта".

`self` — це посилання на сам об'єкт, що створюється. Усі методи класу мають як перший параметр `self`, який дозволяє отримати доступ до атрибутів та інших методів цього ж об'єкта.

Метод `print_info` — це звичайна функція, яка належить класу, і вона виводить інформацію про масу тіла. Її можна викликати лише для конкретного об'єкта цього класу (екземпляра).

Метод — це функція, яка є частиною класу. Вона може отримувати доступ до атрибутів об'єкта через `self`, і викликається через об'єкт (екземпляр класу).

Створимо екземпляр:

```
b = Body(10)
b.print_info()  # Маса тіла: 10 кг
```

У цьому фрагменті коду створюється новий об'єкт класу `Body` з масою 10 кг. Python викликає конструктор `__init__` з аргументом 10, і значення 10 зберігається у `b.mass`. Далі викликається метод `print_info()`, який друкує: Маса тіла: 10 кг.

Інший приклад:

```
a = Body(5)
a.print_info()  # Маса тіла: 5 кг
```

Тут створено інший об'єкт `a` з масою 5 кг. Він також має метод `print_info()`, який виведе відповідний текст.

2.2. Розширення класу

Розглянемо приклад успадкування: створимо новий клас `ChargedBody`, який розширює клас `Body`, додаючи до нього властивість електричного заряду:

```
class ChargedBody(Body):
    def __init__(self, mass, charge):
        super().__init__(mass)
        self.charge = charge  # електричний заряд, Кл

    def print_info(self):
        super().print_info()
        print(f"Заряд: {self.charge} Кл")
```

Тут ми створили клас `ChargedBody`, який успадковує всі властивості та методи класу `Body`. Для цього в дужках після назви класу вказано (`Body`). Це означає: "цей клас базується на класі `Body`".

Метод `__init__` нового класу викликає конструктор батьківського класу за допомогою `super().__init__(mass)`. Після цього додається новий атрибут `charge` — електричний заряд.

Метод `print_info` у цьому класі перевизначено: він спочатку викликає метод `print_info` з класу `Body`, а потім додає виведення заряду.

Приклад використання:

```
c = ChargedBody(3, -1)
c.print_info()
# Маса тіла: 3 кг
# Заряд: -1 Кл
```

2.3. Інкапсуляція: доступ до атрибутів через методи

Тепер створимо клас `Particle`, який інкапсулює заряд як приватний атрибут і надає методи для доступу до нього:

```
class Particle:
    def __init__(self, charge):
        self.__charge = charge  # Приватний атрибут

    def get_charge(self):
        return self.__charge

    def set_charge(self, new_charge):
        if isinstance(new_charge, (int, float)):
            self.__charge = new_charge
        else:
            print("Помилка: заряд повинен бути числом")
    def print_info(self):
        print(f"Заряд: {self.__charge} Кл")

p = Particle(1)
p.set_charge(-3)  # зміна атрибута через метод
p.charge = 7      # проба зміни напряму (ігнорується)
print(p.get_charge())  # Виведе: -3

p.set_charge("електрон")  # Виведе: Помилка: заряд повинен бути числом
```

В цій програмі рядок `isinstance(new_charge, (int, float))` перевіряє чи чи об'єкт `new_charge` належить до одного з класів `int` або `float`. Інкапсуляція є одним із ключових понять об'єктно-орієнтованого програмування і забезпечує стабільну, захищену і гнучку структуру програм.

У цьому прикладі атрибут `__charge` є приватним: два підкреслення перед назвою вказують, що його не слід змінювати напряму поза межами класу.

Інкапсуляція — це принцип ООП, який дозволяє приховати внутрішні змінні та дозволити доступ до них лише через спеціальні методи (геттери і сеттери).

`get_charge()` — геттер: метод, який повертає значення заряду.

`set_charge()` — сеттер: метод, який змінює значення заряду.

Композиція та поліморфізм

Розглянемо ці поняття на основі вже створених класів

Композиція. Створимо новий клас `System`, який об'єднує об'єкти типу `Body`, `ChargedBody`, `Particle`. Це і є приклад композиції — коли складніші об'єкти формуються з простіших.

```
class System:
    def __init__(self):
        self.elements = []  # список тіл у системі

    def add_element(self, body):
        self.elements.append(body)  # додаємо тіло будь-якого типу

    def print_all(self):
        for elem in self.elements:
            elem.print_info()  # викликається відповідна реалізація
```

```
b = Body(5)
cb = ChargedBody(2, -1)
p = Particle(1)

s = System()
s.add_element(b)
s.add_element(cb)
s.add_element(p)

s.print_all()
```

Результат виконання програми:

```
Маса тіла: 5 кг
Маса тіла: 2 кг
Заряд: -1 Кл
Заряд: 1 Кл
```

Пояснення: Ми створили об'єкт `System`, який включає в себе інші об'єкти — це приклад **композиції**, тобто ситуації, коли один об'єкт складається з інших. Метод `print_all()` викликає метод `print_info()` для кожного об'єкта. Оскільки різні класи реалізують цей метод по-своєму, ми бачимо різну поведінку. Це — **поліморфізм**: єдина інтерфейсна функція працює з об'єктами різних типів. Такий підхід дозволяє створювати гнучкі та розширювані програми. Ми можемо додавати нові типи тіл, які реалізують метод `print_info()`, і клас `System` автоматично з ними працюватиме без змін у своєму коді. Саме так працює **поліморфізм**.

2.4. Приклад створення класу комплексних чисел

Наведемо приклад створення класу `ComplexNumber`, який моделює комплексне число. Кожне таке число має дійсну та уявну частини. Ми реалізуємо методи для доступу до частин числа, обчислення модуля, знаходження спряженого числа, а також операції додавання та множення.

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real      # дійсна частина
        self.imag = imag      # уявна частина

    def get_real(self):
        return self.real

    def get_imag(self):
        return self.imag

    def abs(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5

    def conjugate(self):
        return ComplexNumber(self.real, -self.imag)

    def add(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)
```

```
def multiply(self, other):
    real_part = self.real * other.real - self.imag * other.imag
    imag_part = self.real * other.imag + self.imag * other.real
    return ComplexNumber(real_part, imag_part)

def print_info(self):
    sign = '+' if self.imag >= 0 else '-'
    print(f"{self.real} {sign} {abs(self.imag)}i")
```

Пояснення кожного методу

- `__init__(real, imag)` — конструктор. Ініціалізує дійсну та уявну частини.
- `get_real()` — повертає дійсну частину.
- `get_imag()` — повертає уявну частину.
- `abs()` — обчислює модуль (абсолютне значення) числа за формулою $\sqrt{a^2 + b^2}$.
- `conjugate()` — повертає новий об'єкт, який є спряженим до поточного: змінюється знак уявної частини.
- `add(other)` — додає інше комплексне число до поточного.
- `multiply(other)` — перемножує два комплексних числа за відомою формулою:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

- `print_info()` — виводить комплексне число у формі $a + bi$ або $a - bi$.

Використання створеного класу `ComplexNumber`

```
z1 = ComplexNumber(3, 4)
z2 = ComplexNumber(1, -2)

z1.print_info()          # 3 + 4i
z2.print_info()          # 1 - 2i

z_sum = z1.add(z2)
z_sum.print_info()       # 4 + 2i

z_prod = z1.multiply(z2)
z_prod.print_info()      # 11 - 2i

print(z1.abs())          # 5.0

z_conj = z1.conjugate()
z_conj.print_info()      # 3 - 4i
```

Аналогічно можна увести методи для ділення комплексних чисел та для піднесення до степені. Крім цього, при операціях з комплексними числами зручно користуватися операторами `+`, `-`, `*`, `.`, тому необхідно у методах внутрішнього представлення цих операторів додати можливість операцій з комплексними числами (поліморфізм). Приклад модифікації внутрішніх методів, які використовуються для роботи оператора `+`.

```
def __add__(self, other):    # працюватиме у випадку "ComplexNumber + int" та "ComplexNumber + float"
    if isinstance(other, ComplexNumber):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)
    elif isinstance(other, (int, float)):
        return ComplexNumber(self.real + other, self.imag)
    else:
        raise TypeError("Додавання можливе тільки з комплексним або дійсним числом")
```

Цей метод потрібно додати до списку методів при описі класу `ComplexNumber`. Він забезпечить виконання виразів `ComplexNumber + int` та `ComplexNumber + float`. У випадку `int + ComplexNumber` та `float + ComplexNumber` Python спробує використати стандартний метод `__add__` до об'єктів типу `int` чи `float`, але ці типи змінних нічого не знають про створений нами клас `ComplexNumber`. Після невдачі Python спробує використати метод `__radd__` до об'єкта, який справа, тобто до об'єкта `ComplexNumber`. Отже, в класі `ComplexNumber` потрібно описати дії для цього методу. Наприклад, можна перенаправити цей метод на вже описаний нами метод `__add__`, в якому нами передбачено операції з типами `int` та `float`.

```
def __radd__(self, other):    # застосовується до правого операнда в операції: \texttt{число} + ComplexNumber
    return self.__add__(other)
```

Подібним способом можна описати дію інших операторів. Назви внутрішніх методів, що застосовуються для функціонування інших операторів наведено в таблиці.

Відповідність операторів і спеціальних методів у Python

Оператор	Спеціальний метод
+	<code>__add__()</code>
-	<code>__sub__()</code>
*	<code>__mul__()</code>
/	<code>__truediv__()</code>
//	<code>__floordiv__()</code>
%	<code>__mod__()</code>
**	<code>__pow__()</code>
==	<code>__eq__()</code>
!=	<code>__ne__()</code>
<	<code>__lt__()</code>
<=	<code>__le__()</code>
>	<code>__gt__()</code>
>=	<code>__ge__()</code>
<code>abs()</code>	<code>__abs__()</code>
<code>str()</code>	<code>__str__()</code>
<code>len()</code>	<code>__len__()</code>
<code>in</code>	<code>__contains__()</code>
<code>[]</code>	<code>__getitem__()</code>
<code>+=</code>	<code>__iadd__()</code>
<code>-=</code>	<code>__isub__()</code>
<code>*=</code>	<code>__imul__()</code>

Для самостійного опрацювання рекомендується додати до класу `ComplexNumber` методи `__sub__()` та `__rsub__()` а також множення та піднесення до степеня.

Розділ 3

Побудова графіків за допомогою matplotlib

Для генерування графіків мовою Python необхідна підтримка **GUI** (*Graphical User Interface* — графічний інтерфейс користувача). Якщо середовище не підтримує GUI, то графіки не можуть відображатися на екрані, і їх можна лише зберігати у файл.

Онлайн-середовища, такі як <https://www.onlinegdb.com>, не підтримують графічний інтерфейс, а також часто мають обмеження на збереження або перегляд згенерованих зображень. Через це побудова графіків у таких сервісах є проблематичною.

Для повноцінної роботи з графікою в Python слід використовувати середовище, встановлене на персональному комп'ютері (наприклад, IDLE, VS Code, PyCharm), або онлайн-сервіси, що підтримують візуалізацію графіки.

Одним із найзручніших таких сервісів є **Google Colab** — безкоштовне середовище виконання Python-коду у браузері, доступне за адресою <https://colab.research.google.com>. Воно дозволяє запускати графічні програми та зберігати результати роботи безпосередньо на Google Диску у власному акаунті.

Вступ до matplotlib

Бібліотека `matplotlib` є однією з найпоширеніших у Python для побудови графіків. Вона дозволяє візуалізувати дані у вигляді лінійних графіків, стовпчикових та кругових діаграм, гістограм, графіків розсіювання та багатьох інших типів.

`matplotlib` складається з кількох рівнів API. Найпростіший спосіб використання — через модуль `pyplot`, який має синтаксис, схожий на **MATLAB**.

```
from matplotlib import pyplot as plt
```

Бібліотека `matplotlib` у Python дозволяє створювати різноманітні типи графіків, які можна умовно поділити на кілька основних категорій:

3.1. Типи графіків

Бібліотека `matplotlib` у мові Python дозволяє створювати різноманітні типи графіків для візуалізації числових даних. Нижче подано короткий опис основних типів графіків. У дужках зазначено назви функцій, які реалізують побудову відповідних графіків.

- **Графік функції** (`plot`) — для візуалізації математичних функцій.
- **Стовпчикова діаграма** (`bar`) — вертикальна діаграма для порівняння значень.
- **Горизонтальна стовпчикова діаграма** (`barh`) — альтернатива вертикальній.
- **Гістограма** (`hist`) — відображає частоти значень у інтервалах.
- **Кругова діаграма** (`pie`) — показує частки кожної категорії у загальному обсязі.

- Область під кривою (fill_between) — підкреслює площу під функцією.
- Теплова карта (imshow) — відображає матрицю числових значень як зображення.
- Графік із похибками (errorbar) — показує числові значення з відповідними похибками.
- Векторне поле (quiver) — ілюструє напрямки та величини векторів.

Нижче подано Python-програму, яка створює зразки графіків різного типу.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 # Дані
4 x = [-7.5, -2, 0, 4, 10]
5 y = [-1, 0.4, -0.9, 0.3, -0.5]
6 categories = ['A', 'B', 'C']
7 values = [5, 3, 7]
8 data_hist = [1, 2, 2, 3, 3, 3, 4, 4, 5]
9 sizes = [30, 45, 25]
10 labels = ['A', 'B', 'C']
11 errors = [0.2, 0.3, 0.1, 0.3, 0.2]
12 # x_func = np.linspace(-10, 10, 100) # 100 точок в інтервалі (-10, 10)
13 x_func = np.arange(-10, 10, 0.1)
14 y_func = np.sin(x_func)
15 matrix = np.random.rand(5, 5)
16 # Створення області графіків
17 plt.figure(figsize=(15, 10))
18
19 # 1. Графік функції + точки
20 plt.subplot(3, 3, 1)
21 plt.plot(x_func, y_func, label='sin(x)')
22 plt.plot(x, y, 'o', label='Дискретні точки')
23 plt.title("Графік функції sin(x)")
24 plt.legend()
25 # 2. Область під графіком
26 plt.subplot(3, 3, 2)
27 plt.fill_between(x_func, y_func, alpha=0.4)
28 plt.title("Область під графіком sin(x)")
29 # 3. Графік із похибками
30 plt.subplot(3, 3, 3)
31 plt.errorbar(x, y, yerr=errors, fmt='o')
32 plt.title("Графік із похибками")
33 # 4. Стовпчикова діаграма (вертикальна)
34 plt.subplot(3, 3, 4)
35 plt.bar(categories, values)
36 plt.title("Стовпчикова діаграма")
37 # 5. Горизонтальна стовпчикова діаграма
38 plt.subplot(3, 3, 5)
39 plt.barh(categories, values)
40 plt.title("Горизонтальна діаграма")
41 # 6. Гістограма
42 plt.subplot(3, 3, 6)
43 plt.hist(data_hist, bins=5)
44 plt.title("Гістограма")
45 # 7. Кругова діаграма
46 plt.subplot(3, 3, 7)
47 plt.pie(sizes, labels=labels, autopct='%1.1f%%')
48 plt.title("Кругова діаграма")

```

```

49 # 8. Теплова карта
50 plt.subplot(3, 3, 8)
51 plt.imshow(matrix, cmap='hot', interpolation='nearest')
52 plt.title("Теплова карта")
53 plt.colorbar()
54 # 9. Векторне поле
55 plt.subplot(3, 3, 9)
56 X, Y = np.meshgrid(np.arange(0, 2, 0.2),
57                    np.arange(0, 2, 0.2))
58 U = np.cos(X)
59 V = np.sin(Y)
60 plt.quiver(X, Y, U, V)
61 plt.title("Векторне поле")
62
63 plt.tight_layout()
64 plt.show()

```

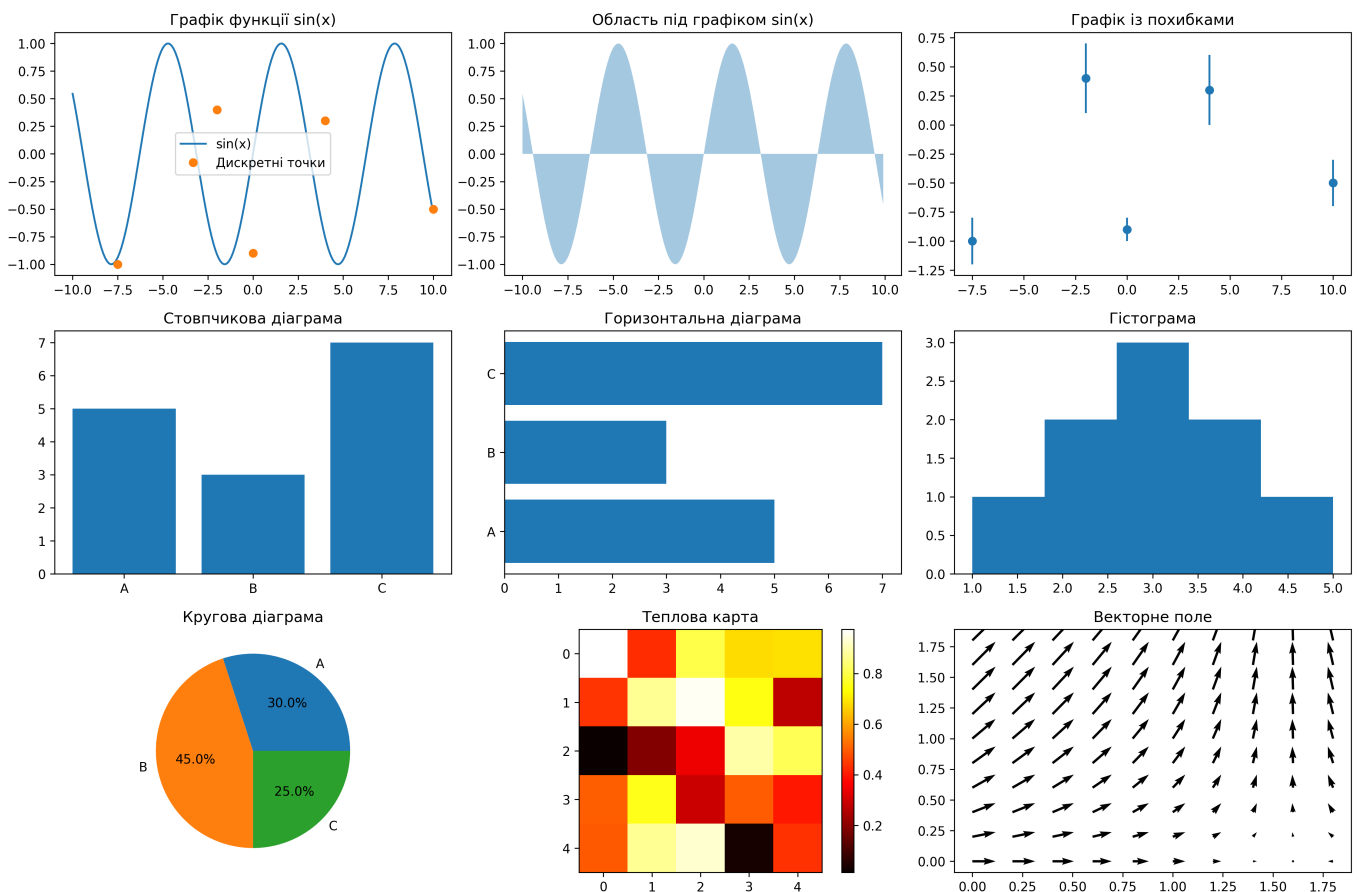


Рис. 3.1. Типи графіків в Matplotlib

Перед побудовою графіка створюється область для, в якій вказується розміри графіка в дюймах.

```
plt.figure(figsize=(15, 10))
```

Якщо не вказати значення параметра `figsize`, то буде встановлено значення за замовченням (9, 6). Далі в залежності від типу графіка використовується відповідна функція пакету `pyplot` бібліотеки `matplotlib`.

Лінійний графік (plot())

Функція `plot()` використовується для побудови графіків лінійних залежностей. Вона має низку параметрів:

- `x`, `y` — обов'язкові позиційні аргументи: списки значень координат.
- `label=` — мітка для графіка, яка відображається в легенді (після виклику `plt.legend()`).
- `color=` — колір лінії, наприклад: `'blue'`, `'#00FF00'`, `'r'`.
- `linestyle=` — стиль лінії: `'-'` (суцільна), `'--'` (штрихова), `'.'` (пунктирна), `'-.'` (штрих-пунктир).
- `linewidth=` — товщина лінії у пікселях (наприклад, 2.0).
- `marker=` — маркер (символ точки): `'o'`, `'s'`, `'D'`, `'x'`, `'*'` тощо.
- `markersize=` або `ms=` — розмір маркерів.
- `alpha=` — прозорість лінії або маркерів (від 0 до 1).

Після перших двох обов'язкових параметрів можна використати короткий запис стилю у вигляді текстової строки, яка містить інформацію про колір, стиль маркерів та тип лінії: `'ro--'`, що означає: червоні кружечки, штрихова лінія, яка їх з'єднує.

Таблиця кольорів

Кольори	Опис
<code>'b'</code>	синій (blue)
<code>'g'</code>	зелений (green)
<code>'r'</code>	червоний (red)
<code>'c'</code>	блакитний (cyan)
<code>'m'</code>	пурпуровий (magenta)
<code>'y'</code>	жовтий (yellow)
<code>'k'</code>	чорний (black)
<code>'w'</code>	білий (white)

Таблиця стилів лінії на графіку

Стиль лінії	Опис
<code>'-'</code>	суцільна лінія
<code>'--'</code>	штрихована лінія
<code>'-.'</code>	штрих-пунктирна лінія
<code>'.'</code>	пунктирна лінія
<code>''</code>	без лінії (тільки маркери)

Таблиця маркерів

Маркери	Опис
<code>'.'</code>	піксель
<code>','</code>	дуже малий піксель
<code>'o'</code>	коло
<code>'s'</code>	квадрат (square)
<code>'D'</code>	ромб (diamond)
<code>'x'</code>	хрест (diagonal)
<code>'+'</code>	плюс
<code>'*'</code>	зірочка
<code>'^'</code>	трикутник вгору
<code>'v'</code>	трикутник вниз
<code>'<'</code>	трикутник вліво
<code>'>'</code>	трикутник вправо
<code>''</code>	без маркерів

Заголовки та підписи осей створюються відповідними методами

```
plt.title("Графік функції  $y = x^2$ ")
plt.xlabel("x")
plt.ylabel("y")
```

Сітка, легенда, масштабування

```
plt.grid(True)           # вмикає сітку
plt.legend()             # виводить легенду
plt.xlim(-10, 10)        # обмеження осі x
plt.ylim(0, 100)         # обмеження осі y
plt.xticks([i for i in range(-10, 11, 2)]) # власні мітки осі x
```

3.2. Приклади побудови графіків функцій

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-3, 3, 0.1) # Створення масиву значень x

y1 = x**2 # Обчислення значень функції y1
y2 = np.sin(x) # Обчислення значень функції y2
x3 = np.arange(-3, 3, 1) # масив точок на графіку
y3 = x3**2 # Обчислення значень функції для точок на графіку

plt.plot(x, y1, label="y = x2", color='blue', linestyle='-')
plt.plot(x, y2, label="y = sin(x)", color='red', linestyle='--')
plt.plot(x3, y3, 'bo') # b - blue o - circles
plt.grid(True)
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Графіки функцій y = x2 та y = sin(x)")
plt.show()
```

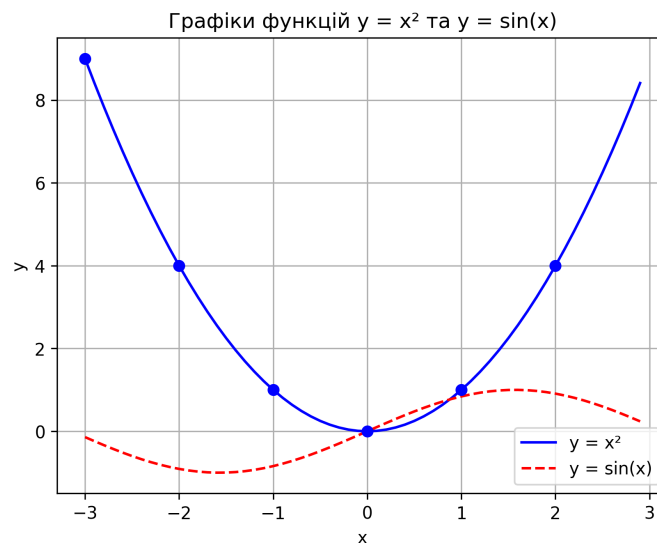


Рис. 3.2. Лінійний графік в Matplotlib з базовими налаштуваннями

У наступному прикладі побудуємо графіки тригонометричних функцій $y = \cos(x)$ та $y = \sin(x)$ на інтервалі від $-\pi$ до π з кроком 0.1 з використанням розширеного налаштування легенди, кольорів, положення та форматування текстових вставок. Встановимо зовнішні відступи між кривими та межами графіка за допомогою методу `plt.margins()`, щоб уникнути обрізання частин ліній.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 dx = 0.1
4 x = np.arange(-np.pi, np.pi, dx)
5 y1 = np.cos(x)
6 y2 = np.sin(x)
7
```

```

8 fig = plt.figure(facecolor='#f5f5f5') # світлосірий колір області figure
9 ax = fig.add_subplot() # виділення області графіка для зміни кольору
10 ax.set_facecolor('lavender') # Фон поля побудови
11 plt.plot(x, y1, label='cos(x)')
12 line2, = plt.plot(x, y2, label='sin(x)')
13 color_of_y2 = line2.get_color()
14 plt.title('Приклад побудови графіків в Matplotlib')
15 plt.ylabel('y(x)', fontsize=16)
16 plt.xlabel('x', fontsize=16)
17 plt.text(np.pi-1, 0, 'sin(x)', fontsize=16,
18         color = color_of_y2, bbox=dict(facecolor='w', edgecolor='r'))
19 plt.text(-1.4, 0.5, 'cos(x)', fontsize=16, rotation=90)
20 plt.margins(x=0.02, y=0.02) # зазори між границями графіка та лініями кривих
21 plt.grid(True)
22 # Легенда з параметрами
23 plt.legend(
24     loc='upper left',
25     fontsize=10,
26     title='Функції',
27     title_fontsize='11',
28     shadow=True,
29     frameon=True,
30     facecolor='lightyellow',
31     edgecolor='black',
32     fancybox=True,
33     ncol=1,
34     borderpad=1,
35     labelspacing=0.8,
36     handlelength=2,
37     handletextpad=0.5,
38     borderaxespad=1)
39 plt.show()

```



Рис. 3.3. Приклад побудови лінійного графіка з налаштуваннями стилю

3.3. Приклад побудови стовпчикової діаграми

```

1 import matplotlib.pyplot as plt
2
3 languages = ['Python', 'C++', 'C', 'Java', 'C#']
4 popularity = [25.87, 10.68, 9.47, 8.84, 4.69]
5
6 plt.figure(figsize=(8, 5))
7 bars = plt.bar(languages, popularity, color='lightgreen')
8
9 plt.title('TIOBE Index - популярність мов програмування (травень 2025)')
10 plt.xlabel('Мови програмування')
11 plt.ylabel('Рейтинг TIOBE (%)')
12 plt.grid(axis='y', linestyle='--', alpha=0.5)
13
14 # Додаємо числові значення на стовпчики
15 plt.bar_label(bars, fmt='%0.0f%%', fontsize=12, label_type='center')
16 plt.show()

```

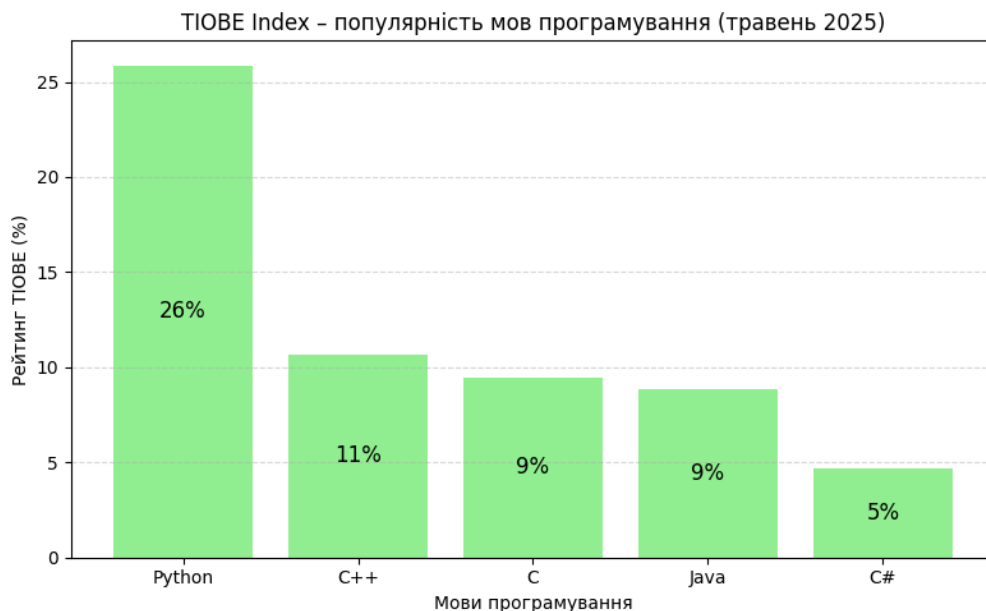


Рис. 3.4. Стовпчикова діаграма

3.4. Побудова тривимірних графіків у Matplotlib

Бібліотека Matplotlib також дозволяє будувати тривимірні (3D) графіки за допомогою модуля `mpl_toolkits.mplot3d`. Для цього потрібно створити координатну систему з проекцією 3d.

Приклад: графік поверхні функції $z = \sin(\sqrt{x^2 + y^2})$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4
5 x = np.linspace(-6, 6, 100)
6 y = np.linspace(-6, 6, 100)

```

```

7 X, Y = np.meshgrid(x, y)
8 Z = np.sin(np.sqrt(X**2 + Y**2))
9 fig = plt.figure(figsize=(10, 7))
10 ax = fig.add_subplot(111, projection='3d')
11 ax.plot_surface(X, Y, Z, cmap='viridis')
12 ax.set_xlabel('X')
13 ax.set_ylabel('Y')
14 ax.set_zlabel('Z')
15 plt.title('3D графік: $z = \sin(\sqrt{x^2 + y^2})$')
16 plt.show()

```

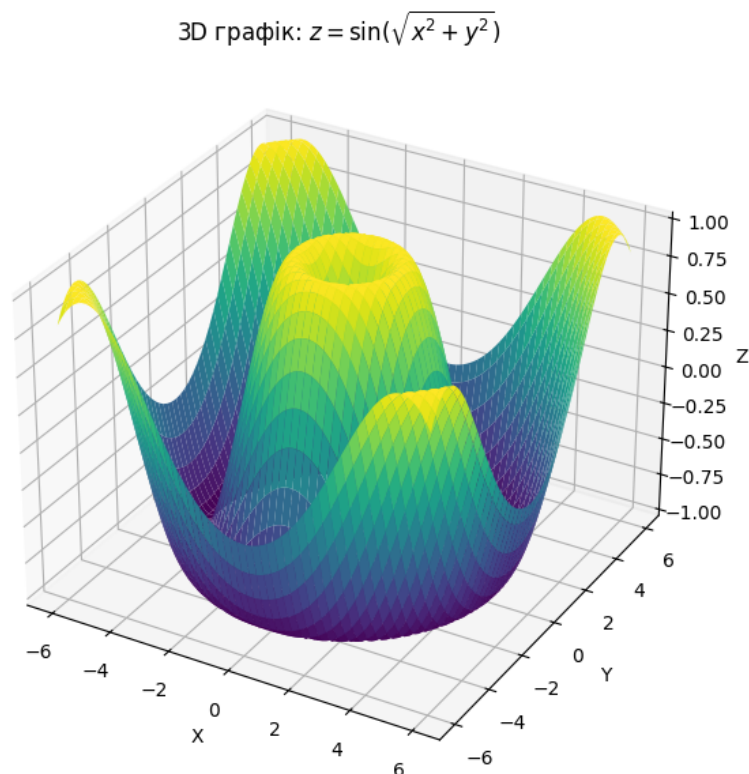


Рис. 3.5. 3D графік в Matplotlib

Основні методи побудови

- `plot_surface(X, Y, Z)` — побудова кольорової гладкої поверхні.
- `plot_wireframe(X, Y, Z)` — побудова сітчастого графіка (каркаса).
- `scatter(x, y, z)` — тривимірна діаграма розсіювання.
- `contour3D(X, Y, Z)` — тривимірні контурні лінії.

Корисні параметри та функції

- `ax.view_init(elev=30, azimuth=45)` — встановлює кут огляду.
- `cmap='plasma'` — кольорова палітра: 'viridis', 'inferno', 'coolwarm' тощо.
- `plt.savefig(...)` — збереження зображення графіка у файл.

3.5. Висновки

`matplotlib` — це універсальний інструмент візуалізації даних у Python. Його можливості охоплюють прості й складні графіки, повну кастомізацію зовнішнього вигляду, а також інтеграцію з іншими бібліотеками, такими як `pandas` та `numpy`.

Цей інструмент є необхідним для інженерів, науковців та аналітиків, які працюють з числовими даними.

Частина II

ЧИСЕЛЬНІ МЕТОДИ

Розділ 1

Похибки

1.1. Класифікація похибок

Чисельні методи, як правило, застосовуються для знаходження *наближеного* розв’язання задач, які не піддаються точному аналітичному обчисленню. Такому підходу завжди притаманна певна похибка. Розглянемо основні джерела виникнення похибок.

1. **Похибка моделі.** Природні явища часто є надто складними й багатофакторними, щоб їх можна було описати точно. Усі науки про природу оперують не самими явищами, а їх спрощеними *моделями*. Модель — це математичне або аналітичне наближення, яке виділяє головні закономірності та ігнорує другорядні властивості. Успішна модель — це не точне відтворення дійсності, а її спрощена, ідеалізована репрезентація. Однак саме через цей процес спрощення в моделі закладається системна похибка.

У математиці створення моделі полягає в побудові рівнянь або систем рівнянь, що описують фізичні або інші процеси. Похибка моделі залежить від глибини аналізу, якості експериментальних даних та знань про досліджуване явище.

2. **Похибка вихідних даних.** У численних прикладних задачах модель містить параметри, що отримуються експериментально або розрахунково. Такі параметри завжди містять неточності — як випадкові, так і систематичні. Будь-яка невизначеність у вхідних даних неминуче передається результату, незалежно від точності обчислювального алгоритму. Така похибка часто є домінуючою у практичних задачах.

Похибки, зумовлені самою моделлю або початковими даними, прийнято називати *зовнішніми похибками*. Вони не залежать від обраного чисельного методу й існують навіть тоді, коли розв’язання поставленої математичної задачі знайдено точно.

3. **Похибка методу.** Після побудови математичної моделі її потрібно розв’язувати. Часто складну задачу замінюють наближеним чисельним методом. Наприклад, замість точного аналітичного обчислення певного інтеграла застосовується чисельне обчислення інтегральної суми. Це неминуче призводить до появи похибки, яка пов’язана з обраним методом. Ми розглянемо вплив таких похибок у наступних розділах, де буде детально проаналізовано чисельні методи.

4. **Похибка округлення.** Будь-які обчислення, які виконуються вручну або за допомогою комп’ютера, оперують обмеженою кількістю цифр. Через це проміжні та кінцеві результати заокруглюються. Таким чином виникає похибка округлення. Вона може накопичуватися в процесі обчислень (особливо при довгих ітераційних методах), що негативно впливає на точність кінцевого результату. Навіть за точних вихідних даних і коректної математичної моделі результат може бути наближеним через обмежену точність обчислень.

Отже, повна похибка є результатом взаємодії всіх вищезазначених похибок і не може бути меншою за найбільшу з них.

1.2. Абсолютна та відносна похибки

Для оцінки точності результатів вводяться поняття *абсолютної* та *відносної похибки*.

Нехай x — точне значення певної величини (хоча на практиці ми його не знаємо), а a — наближене значення. Тоді **абсолютна похибка** дорівнює:

$$\Delta_a = |x - a|.$$

Звісно, оскільки x нам не відоме, то й абсолютну похибку на пряму ми визначити не можемо. Тому вводиться поняття **граничної абсолютної похибки** Δ_a^* , таке значення, що гарантує обмеження реальної похибки:

$$|x - a| \leq \Delta_a^*. \quad (1.1)$$

З цього випливає, що $x \in [a - \Delta_a^*, a + \Delta_a^*]$, тобто ми знаємо лише інтервал, у якому точно міститься істинне значення. Чим менше Δ_a^* , тим кращою є наша оцінка точності.

У технічних специфікаціях це записується як $x = a \pm \Delta_a^*$, де Δ_a^* — допустиме відхилення. Жоден вимір чи обчислення не може бути абсолютно точним, тому виробничі чи вимірювальні норми задають допустимі межі похибок (допуски).

Для оцінки якості результатів зручно використовувати **відносну похибку**, яка показує, наскільки велика абсолютна похибка порівняно з самою величиною. Визначається як:

$$\delta_a = \frac{|\Delta_a|}{|a|}, \quad \delta_a^* = \frac{\Delta_a^*}{|a|}.$$

Ця похибка є безрозмірною і часто виражається у відсотках, що дозволяє порівнювати точність вимірювань або обчислень у різних задачах.

Згідно з чинним (2015 р.) визначенням Міжнародного комітету з констант для науки і техніки, гравітаційна стала має значення:

$$\gamma = (6.67259 \pm 0.00085) \cdot 10^{-11} \text{ м}^3 \cdot \text{кг}^{-1} \cdot \text{с}^{-2},$$

а заряд електрона:

$$e = (1.602\,177\,33 \pm 0.000\,000\,49) \cdot 10^{-19} \text{ Кл.}$$

Оцінимо точність визначення цих фундаментальних фізичних сталих. Для гравітаційної сталої відносна похибка становить:

$$\delta_\gamma^* = \frac{0.00085}{6.67259} \approx 1.27 \cdot 10^{-4},$$

а для заряду електрона:

$$\delta_e^* = \frac{0.00000049}{1.60217733} \approx 3.1 \cdot 10^{-7}.$$

Як бачимо, у другому випадку похибка майже на три порядки менша — тобто значно точніше вимірювана величина. Це свідчить про вищу експериментальну точність визначення заряду електрона, ніж гравітаційної сталої.

Поняття абсолютної та відносної похибки тісно пов'язане з поняттям **вірних** (або *точних*) і **значущих цифр**.

Якщо абсолютна похибка наближеного числа не перевищує одиницю останнього (тобто крайнього правого) розряду десяткового запису, то всі цифри в цьому записі вважаються *вірними*.

Зазвичай десятковий запис наближеного числа повинен містити лише вірні цифри. У такому випадку, якщо відома гранична абсолютна похибка, її можна вказати однозначно.

Цифри, які не є вірними, називаються **сумнівними**.

Приклад. Нехай задано наближені числа:

$$a = 8.6, \quad b = 8.60, \quad c = 3200, \quad d = 3.2 \cdot 10^3.$$

Потрібно вказати граничну абсолютну похибку Δ_a^* для кожного з чисел.

Розв’язання. Для a маємо $\Delta_a^* \leq 0.1$, для b — $\Delta_b^* \leq 0.01$, для c — $\Delta_c^* \leq 1$, для d — $\Delta_d^* \leq 0.1 \cdot 10^3 = 100$.

Таким чином, числа a, b, c і d , хоча й рівні з погляду “звичайної” математики, у чисельних розрахунках відрізняються за кількістю вірних цифр, тобто за точністю.

Крім того, нулі, що розташовані праворуч після значущих цифр у десятковому або цілому числі, є *важливими* і не можуть бути опущені, якщо ми хочемо зберегти правильне уявлення про точність числа.

У всіх наближених числах значущими вважаються всі цифри, починаючи з першої ненульової цифри включно. *Початкові нулі, що стоять ліворуч від першої ненульової цифри, не є значущими. Усі інші цифри — вважаються значущими.*

Приклад. Числа 0.001 307 і 6.0400 мають відповідно чотири і п’ять значущих цифр. Отже, нулі, що стоять на початку дробу, *не значущі*, а нулі, що завершують десятковий запис, *завжди вважаються значущими цифрами*.

1.3. Операції з наближеними числами

Теорема 1. Абсолютна похибка алгебраїчної суми кількох наближених чисел не перевищує суми абсолютних похибок цих чисел.

Зокрема, для суми двох чисел a і b будь-якого знака маємо:

$$\Delta_{a \pm b} \leq \Delta_a + \Delta_b.$$

З цієї теореми випливає, що абсолютна похибка алгебраїчної суми не менша за похибку *найменш точного* із складових чисел. Уточнити точність всієї суми за рахунок точніших доданків — неможливо. Тому безглуздо зберігати зайві десяткові знаки, що не мають жодної надійності. З цього випливають такі правила:

Правило округлення суми наближених чисел:

1. Виділити число з найменшою точністю (тобто з найменшою кількістю вірних десяткових знаків у записі);
2. Округлити решту чисел так, щоб кожне з них мало на один знак більше, ніж виділене;
3. Виконати додавання або віднімання з урахуванням лише значущих знаків;
4. Отриманий результат округлити до точності найменш точного з доданків.

Нагадаємо **правила округлення** числа, тобто його заміни числом із меншою кількістю значущих цифр:

1. Якщо перша з відкинутих цифр менше 5 — останню збережену цифру залишають без змін;
2. Якщо перша з відкинутих цифр більша за 5 — останню збережену цифру збільшують на 1;
3. Якщо перша з відкинутих цифр дорівнює 5, а після неї йдуть будь-які цифри, відмінні від нулів — округлюють вгору (тобто додають 1);

4. Якщо перша з відкинутих цифр дорівнює 5, а після неї — лише нулі або нічого немає, то діють за **правилом парної цифри**:

- якщо остання збережена цифра — непарна, її збільшують на 1;
- якщо парна — залишають без змін.

Таке округлення ще називають *банківським*, воно не накопичує похибку, так як це відбувається при *класичному* округленні, в якому при відкиданні цифри 5 завжди збільшують на 1 останню збережену цифру.

Приклад. Знайти суму наближених чисел:

$$a = 414.8, \quad b = 0.025, \quad c = 24.17, \quad d = 0.000\,326.$$

За умовчанням усі цифри в числах є вірними.

Розв'язання. Найменш точним є число d , оскільки в ньому лише один вірний десятковий знак. Округлюємо решту доданків до двох знаків після коми:

$$b = 0.03, \quad c = 24.17, \quad d = 0.0003.$$

Додаємо округлені значення:

$$414.8 + 0.03 + 24.17 + 0.0003 = 439.0003.$$

Остаточний результат округлюємо до одного знака після коми (точність найменш точного доданка): 439.0.

Теорема 2. Відносна похибка добутку (або частки) наближених чисел не перевищує суми відносних похибок цих чисел:

$$\delta_{a \cdot b \cdot c}^* \leq \delta_a^* + \delta_b^* + \delta_c^*.$$

Унаслідок цього правило для множення та ділення наближених чисел таке:

Правило множення і ділення наближених чисел:

1. Серед чисел, які підлягають множенню або діленню, виділити найменш точне — те, яке має найменшу кількість вірних значущих цифр;
2. Округлити решту чисел так, щоб кожне з них мало на одну значущу цифру більше, ніж найменш точне;
3. Виконати множення або ділення округлених чисел з урахуванням збережених значущих цифр;
4. Залишити у відповіді стільки значущих цифр, скільки їх було в числі з найменшою кількістю значущих цифр.

Приклад. Знайти добуток наближених чисел:

$$a = 3.5, \quad b = 83.368.$$

Розв'язання. У першому числі дві значущі цифри, у другому — п'ять. Округлюємо b до трьох цифр: 83.4.

Виконуємо множення:

$$a \cdot b = 3.5 \cdot 83.4 = 291.9 \approx 2.9 \cdot 10^2.$$

Залишено дві значущі цифри — стільки, скільки було в найменш точному множнику.

Індивідуальні завдання

- а) Визначити, яке рівняння є точнішим.
- б) Округлити сумнівні цифри числа, залишивши лише вірні. Обчислити абсолютну похибку результату.
- в) Знайти граничну абсолютну та відносну похибки наближеного числа, у якому всі цифри за умовчанням вірні.

Варіанти:

1. а) $14/17 = 0.824$, $\sqrt{53} = 7.28$; б) 23.3748 , $\delta = 0.27\%$; в) 0.645 .
2. а) $7/3 = 2.33$, $\sqrt{58} = 7.62$; б) 13.5726 ± 0.0072 ; в) 4.8556 .
3. а) $27/31 = 0.871$, $\sqrt{42} = 6.48$; б) 0.088748 , $\delta = 0.56\%$; в) 71.385 .
4. а) $23/9 = 2.56$, $\sqrt{87} = 9.33$; б) $4.576 \pm 0.000\ 42$; в) 6.8346 .
5. а) $6/7 = 0.857$, $\sqrt{41} = 6.40$; б) 46.7843 , $\delta = 0.32\%$; в) 7.340 .
6. а) $12/7 = 1.71$, $\sqrt{46} = 6.78$; б) 0.38725 ± 0.0011 ; в) 6.006 .
7. а) $21/13 = 1.62$, $\sqrt{63} = 7.94$; б) 45.7832 , $\delta = 0.18\%$; в) 3.6765 .
8. а) $16/7 = 2.29$, $\sqrt{11} = 3.32$; б) $0.752\ 44 \pm 0.000\ 13$; в) 5.374 .
9. а) $18/7 = 2.57$, $\sqrt{22} = 4.69$; б) 46.453 , $\delta = 0.15\%$; в) 6.125 .
10. а) $17/9 = 1.89$, $\sqrt{17} = 4.12$; б) $0.663\ 85 \pm 0.000\ 42$; в) 24.6 .
11. а) $11/9 = 1.64$, $\sqrt{35} = 5.92$; б) $0.668\ 85$, $\delta = 0.34\%$; в) 0.543 .
12. а) $19/12 = 1.58$, $\sqrt{15} = 3.92$; б) $0.488\ 34 \pm 0.000\ 86$; в) 8.773 .
13. а) $13/7 = 1.857$, $\sqrt{7} = 2.65$; б) 2.8867 , $\delta = 0.43\%$; в) 63.749 .
14. а) $49/13 = 3.77$, $\sqrt{14} = 3.74$; б) 6.5483 ± 0.0017 ; в) 0.389 .
15. а) $5/3 = 1.667$, $\sqrt{38} = 6.16$; б) 3.7542 , $\delta = 0.32\%$; в) 0.389 .
16. а) $17/11 = 1.545$, $\sqrt{18} = 4.243$; б) 0.8647 ± 0.0013 ; в) 0.878 .
17. а) $7/22 = 0.318$, $\sqrt{13} = 3.61$; б) 0.3944 , $\delta = 0.18\%$; в) 21.7 .
18. а) $13/17 = 0.765$, $\sqrt{31} = 5.57$; б) 3.6878 ± 0.0013 ; в) 8.74 .
19. а) $19/23 = 0.826$, $\sqrt{29} = 5.38$; б) 3.6843 , $\delta = 0.01\%$; в) 7.44 .
20. а) $21/29 = 0.724$, $\sqrt{4} = 2$; б) 13.6853 ± 0.0023 ; в) 10.25 .
21. а) $17/19 = 0.895$, $\sqrt{52} = 7.21$; б) 7.521 , $\delta = 0.12\%$; в) 0.5748 .
22. а) $11/20 = 0.545$, $\sqrt{83} = 9.11$; б) 3.7832 ± 0.0043 ; в) 2.678 .

Приклад розв'язання одного варіанта

а) $\frac{9}{19} = 0.474$, $\sqrt{103} = 10.149$. Знаходимо значення з більшою кількістю знаків:

$$a = \frac{9}{19} = 0.47368\dots, \quad b = \sqrt{103} = 10.14889\dots$$

Округлюємо їх з надлишком:

$$\Delta_a = |0.47368 - 0.474| = 0.00032 < 0.0004, \quad \Delta_b = |10.14889 - 10.149| = 0.00011 < 0.0002.$$

Відносні похибки:

$$\delta_a = \frac{0.0004}{0.474} \approx 8.4 \cdot 10^{-4} = 0.08\%, \quad \delta_b = \frac{0.0002}{10.149} \approx 2 \cdot 10^{-5} = 0.002\%.$$

Оскільки δ_b значно менше, ніж δ_a , значення $\sqrt{103} = 10.149$ є значно точнішим.

б) Нехай наближене число $a = 72.354 \pm 0.021$. Похибку округлюємо до сотих: $\Delta_a = 0.02$.
Остаточне округлене значення: 72.4.

в) Нехай задано $a = -5.272$ з відносною похибкою $\delta_a^* = 0.001$. Знайдемо граничну абсолютну похибку:

$$\Delta_a^* = |a| \cdot \delta_a^* = 5.272 \cdot 0.001 = 0.005272 \approx 0.01.$$

Результат округлюємо до сотих: -5.27 .

г) Нехай $b = 14.278$, усі цифри вірні. При $\Delta_b \leq 0.001$,

$$\delta_b^* = \frac{0.001}{14.278} \approx 7 \cdot 10^{-5} \approx 0.01\%.$$

Розділ 2

Апроксимація

2.1. Апроксимація даних

Апроксимація (approximation) — це наближене подання однієї функції (апроксимуючої) у вигляді іншої функції (апроксимованої), яка задана в будь-якій формі (наприклад, у вигляді масиву даних). У даному контексті апроксимація розглядається як заміна однієї функції іншою, у певному сенсі більш зручною або доцільною.

Залежно від мети, апроксимація даних передбачає розв’язання таких задач:

- **інтерполяція** — побудова безперервної функції $f(x_i)$, що точно проходить через усі експериментальні точки $y_i(x_i)$;
- **екстраполяція** — продовження функції $f(x_{i+k})$ за межі інтервалу відомих значень (прогнозування);
- **регресія** — побудова наближеної (усередненої) функції $f(x_i)$ до $y_i(x_i)$;
- **фільтрація** — побудова згладженої апроксимації $f^*(x_i)$, що зменшує вплив систематичних похибок експериментальних даних.

Існують два основні підходи до апроксимації:

1. Перший підхід вимагає, щоб апроксимуюча крива (зазвичай кусково-гладка) проходила точно через усі задані точки. Такий підхід називається **інтерполяційним**.
2. Другий підхід дозволяє функції не проходити точно через кожну точку, а лише приблизно описувати залежність. У цьому випадку шукається така функція, яка найкраще узгоджується з усією сукупністю експериментальних даних — така апроксимація називається **регресійною**.

Як правило, для регресійної апроксимації використовують **метод найменших квадратів (МНК)**. Він полягає у мінімізації суми квадратів відхилень між значеннями апроксимуючої функції та табличними даними.

2.2. Інтерполяція

Задача інтерполяції

Нехай функція $f(x)$ задана набором точок (x_i, y_i) на інтервалі $[a, b]$:

$$y_i = f(x_i), \quad i = 0, 1, \dots, n, \quad a \leq x_i \leq b \quad (2.1)$$

Задача інтерполяції — знайти функцію $F(x)$, яка в точках x_i приймає ті ж значення, що й $f(x)$, тобто:

$$F(x_i) = y_i \quad (2.2)$$

Припускається, що серед значень x_i немає однакових. Точки x_i називаються **вузлами інтерполяції**.

Якщо $F(x)$ шукається лише на відрізку $[a, b]$, то маємо справу з інтерполяцією, а якщо поза межами відрізка — з екстраполяцією.

Існує багато функцій $F(x)$, які задовольняють умову інтерполяції. Однак на практиці зазвичай будують поліном:

$$F(x) = P_m(x) = a_0 + a_1x + a_2x^2 + \dots + a_mx^m, \quad m \geq 0 \quad (2.3)$$

Такий поліном називається **інтерполяційним поліномом**.

Для побудови одного полінома на всьому інтервалі $[a, b]$ потрібно розв'язати систему $n + 1$ рівнянь з $n + 1$ невідомими (коефіцієнтами полінома). У такому разі степінь полінома дорівнює n , і записуємо:

$$P_n(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n \quad (2.4)$$

Локальна та глобальна інтерполяція

Якщо задано $n + 1$ вузол, то можна побудувати:

- один поліном n -го степеня;
- $n - 1$ поліном першого степеня;
- або набір поліномів нижчого степеня, що охоплюють лише частину вузлів.

Максимальну точність забезпечують поліноми вищого степеня, але на практиці використовують поліноми невисоких степенів, щоби уникнути похибок.

Інтерполяція називається:

- **глобальною**, якщо функцію апроксимують єдиним поліномом $P_m(x)$ на $[a, b]$;
- **локальною**, якщо на кожному підінтервалі $[x_i, x_{i+1}]$ будується свій поліном.

Кусково-лінійна інтерполяція

Найпростішою локальною інтерполяцією є лінійна. Вона з'єднає сусідні вузли прямими лініями:

$$F(x) = b_i + a_ix, \quad x_{i-1} \leq x \leq x_i \quad (2.5)$$

Коефіцієнти a_i і b_i різні для кожного підінтервалу та визначаються з системи:

$$\begin{cases} f_{i-1} = b_i + a_ix_{i-1} \\ f_i = b_i + a_ix_i \end{cases} \quad (2.6)$$

Розв'язок:

$$b_i = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{x_i - x_{i-1}}, \quad a_i = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \quad (2.7)$$

Кусково-квадратична інтерполяція

Для кожних трьох точок (x_{i-1}, y_{i-1}) , (x_i, y_i) , (x_{i+1}, y_{i+1}) будується парабола:

$$F(x) = a_0 + a_1x + a_2x^2, \quad x_{i-1} \leq x \leq x_{i+1} \quad (2.8)$$

Коефіцієнти параболи різні для кожного інтервалу, який містить по три точки, але для спрощення індекс " i " біля коефіцієнтів пропущений.

Для знаходження коефіцієнтів параболи розв'яжемо систему рівнянь, які випливають з того, що три точки лежать на параболі:

$$\begin{cases} f_{i-1} = a_0 + a_1 x_{i-1} + a_2 x_{i-1}^2 \\ f_i = a_0 + a_1 x_i + a_2 x_i^2 \\ f_{i+1} = a_0 + a_1 x_{i+1} + a_2 x_{i+1}^2 \end{cases} \quad (2.9)$$

отримуємо:

$$a_2 = \frac{\frac{f_{i+1}-f_i}{x_{i+1}-x_i} - \frac{f_i-f_{i-1}}{x_i-x_{i-1}}}{x_{i+1} - x_{i-1}} \quad (2.10)$$

$$a_1 = \frac{f_i - f_{i-1}}{x_i - x_{i-1}} - a_2(x_i + x_{i-1}) \quad (2.11)$$

$$a_0 = f(x_i) - a_1 x_i - a_2 x_i^2 \quad (2.12)$$

Многочлен Лагранжа

Многочлен Лагранжа для глобальної інтерполяції:

$$L_n(x) = \sum_{i=0}^n y_i \cdot l_i(x), \quad l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j} \quad (2.13)$$

Або альтернативно:

$$L_n(x) = \sum_{i=0}^n y_i \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j} \quad (2.14)$$

Має властивість $l_i(x_j) = \delta_{ij}$. Степінь многочлена — n .

Метод Лагранжа добре працює при малому числі вузлів ($n < 20$), але при збільшенні вузлів похибка зростає.

Многочлен Ньютона

Многочлен Ньютона використовується для побудови інтерполяційного полінома за заданими табличними значеннями функції $f(x)$ у точках $x_0, x_1, \dots, x_n$. Його особливість — застосування **розділених різниць**, які дозволяють зручно розраховувати коефіцієнти інтерполяційного полінома без потреби розв'язувати систему рівнянь.

Розділена різниця першого порядку:

$$f(x_i, x_{i+1}) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \quad (2.15)$$

Розділена різниця другого порядку:

$$f(x_i, x_{i+1}, x_{i+2}) = \frac{f(x_{i+1}, x_{i+2}) - f(x_i, x_{i+1})}{x_{i+2} - x_i} \quad (2.16)$$

Загальна формула для розділеної різниці k -го порядку:

$$f(x_i, x_{i+1}, \dots, x_{i+k}) = \frac{f(x_{i+1}, \dots, x_{i+k}) - f(x_i, \dots, x_{i+k-1})}{x_{i+k} - x_i} \quad (2.17)$$

Ці значення можна організувати у вигляді таблиці розділених різниць, яка поступово заповнюється звернувши вниз.

Формула Ньютона (прямий запис)

Інтерполяційний многочлен Ньютона має вигляд:

$$P_n(x) = f(x_0) + f(x_0, x_1)(x - x_0) + f(x_0, x_1, x_2)(x - x_0)(x - x_1) + \dots + f(x_0, \dots, x_n)(x - x_0) \dots (x - x_{n-1}) \quad (2.18)$$

Кожен доданок у цьому поліномі є добутком розділеної різниці та відповідних множників $(x - x_i)$.

Поліном Ньютона побудований таким чином, що він **задовільняє всі задані точки** $(x_i, f(x_i))$. Розглянемо, як це відбувається:

- При $x = x_0$: усі множники $(x - x_i)$ у кожному доданку, починаючи з другого, дорівнюють нулю. Тому всі доданки, крім першого, зникають, і маємо:

$$P_n(x_0) = f(x_0)$$

- При $x = x_1$: у другому доданку маємо множник $(x - x_0) = (x_1 - x_0)$, отже цей доданок не обнуляється. Але починаючи з третього доданку, з'являється множник $(x - x_1) = 0$, тому всі вони дорівнюють нулю. Отже:

$$P_n(x_1) = f(x_0) + f(x_0, x_1)(x_1 - x_0)$$

Згідно з означенням розділеної різниці:

$$f(x_0, x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \Rightarrow f(x_0, x_1)(x_1 - x_0) = f(x_1) - f(x_0)$$

Тоді:

$$P_n(x_1) = f(x_0) + (f(x_1) - f(x_0)) = f(x_1)$$

- Аналогічно, при $x = x_2$: перші два доданки не обнуляються, третій — активний, а починаючи з четвертого, всі доданки дорівнюють нулю (через множник $(x - x_2) = 0$). В результаті отримаємо:

$$P_n(x_2) = f(x_0) + f(x_0, x_1)(x_2 - x_0) + f(x_0, x_1, x_2)(x_2 - x_0)(x_2 - x_1)$$

а після підстановки розділених різниць $P_n(x_2) = f(x_2)$.

Таким чином, при кожному $x = x_i$ обнуляються всі доданки, крім тих, які необхідні для точного відтворення значення $f(x_i)$. Тому інтерполяційний поліном Ньютона точно проходить через усі задані точки. Набір точок $(x_i, f(x_i))$ не обов'язково мають бути послідовними, і при додаванні нових точок до таблиці значень не потрібно перераховувати весь поліном заново — достатньо додати новий доданок.

Похибку можна оцінити за наближенням:

$$\varepsilon_n = |P_{n+1}(x) - P_n(x)| \quad (2.19)$$

Переваги методу Ньютона:

- Можна легко додавати нові вузли без зміни попередніх обчислень.
- Формула зручна для рекурсивного обчислення.
- Добре підходить для нерівновіддалених вузлів.

2.3. Регресія

Поняття регресії

Регресія — це метод побудови функції $f(x)$, яка наближено описує дискретний набір експериментальних точок, кожна з яких отримана з деякою похибкою. На відміну від інтерполяції, побудована функція не обов'язково проходить через усі точки, а лише прагне найкраще відобразити загальну тенденцію даних, мінімізуючи вплив похибок.

Наприклад, великий набір експериментальних точок, які насправді мали б відображати лінійну залежність, будуть мати багато осциляцій, які випадкові і не відображають реальної залежності. В цьому випадку потрібно використовувати лінійну регресію з параметрами, які забезпечать мінімальну сумарну похибку.

Типи регресійних моделей

Існують різні типи регресій в залежності від форми апроксимуючої функції:

- **Лінійна регресія** — $f(x) = ax + b$;
- **Поліноміальна регресія** — $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$;
- **Нелінійна регресія** — функція має інший аналітичний вигляд, наприклад $f(x) = a \sin(bx) + c$ або $f(x) = ae^{-bx}$.

Метод найменших квадратів (МНК)

Основна ідея

Нехай задано набір експериментальних точок $\{(x_i, y_i)\}_{i=1}^n$. Ми хочемо знайти функцію $f(x)$ з параметрами, яка мінімізує суму квадратів відхилень між y_i і $f(x_i)$:

$$S = \sum_{i=1}^n (y_i - f(x_i))^2 \rightarrow \min$$

Це означає, що ми шукаємо таку функцію, яка найменше "віддаляється" у середньому від експериментальних значень.

Виведення формул для лінійної регресії

Нехай задано n експериментальних точок (x_i, y_i) , і потрібно знайти функцію вигляду:

$$f(x) = ax + b \quad (2.20)$$

яка найкраще апроксимує ці дані у сенсі найменших квадратів.

Мінімізуємо суму квадратів відхилень:

$$S(a, b) = \sum_{i=1}^n (y_i - ax_i - b)^2 \quad (2.21)$$

Для мінімуму (2.21) прирівнюємо до нуля часткові похідні:

$$\frac{\partial S}{\partial a} = -2 \sum_{i=1}^n x_i (y_i - ax_i - b) = 0 \quad (2.22)$$

$$\frac{\partial S}{\partial b} = -2 \sum_{i=1}^n (y_i - ax_i - b) = 0 \quad (2.23)$$

Після спрощення отримаємо систему:

$$\sum x_i y_i = a \sum x_i^2 + b \sum x_i \quad (2.24)$$

$$\sum y_i = a \sum x_i + nb \quad (2.25)$$

Розв'яжемо систему (2.24)–(2.25). Спочатку з (2.25) виразимо b :

$$b = \frac{1}{n} \left(\sum y_i - a \sum x_i \right) \quad (2.26)$$

Підставимо (2.26) у (2.24) та спростимо. У результаті отримаємо:

$$a = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2} \quad (2.27)$$

$$b = \frac{\sum y_i - a \sum x_i}{n} \quad (2.28)$$

Формули через середні значення

Позначимо:

$$\bar{x} = \frac{1}{n} \sum x_i, \quad \bar{y} = \frac{1}{n} \sum y_i$$

Тоді формули (2.27)–(2.28) переписуються у вигляді:

$$a = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \quad (2.29)$$

$$b = \bar{y} - a \bar{x} \quad (2.30)$$

Формули (2.29)–(2.30) зручніші для інтерпретації результатів та статистичного аналізу, оскільки напряму пов'язані з дисперсією та коваріацією вибірки.

Лінійна регресія: приклад

Припустимо, маємо таблицю вимірюваних значень: Нехай маємо 5 точок:

x_i	1	2	3	4	5
y_i	1.2	1.9	3.0	3.9	5.1

Побудуємо апроксимацію у вигляді прямої $f(x) = ax + b$. Метод найменших квадратів дозволяє обчислити коефіцієнти a і b зі співвідношень (2.27) та (2.28).

Обчислимо проміжні величини: x_i^2 , $x_i y_i$ та відповідні суми:

i	x_i	y_i	$x_i y_i$	x_i^2
1	1	1.2	1.2	1
2	2	1.9	3.8	4
3	3	3.0	9.0	9
4	4	3.9	15.6	16
5	5	5.1	25.5	25
$\sum$	15	15.1	55.1	55

Тепер підставимо у формули (2.27) та (2.28):

$$a = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2} = \frac{5 \cdot 55.1 - 15 \cdot 15.1}{5 \cdot 55 - 15^2} = \frac{275.5 - 226.5}{275 - 225} = \frac{49}{50} = 0.98 \quad (2.31)$$

$$b = \frac{\sum y_i - a \sum x_i}{n} = \frac{15.1 - 0.98 \cdot 15}{5} = \frac{15.1 - 14.7}{5} = \frac{0.4}{5} = 0.08 \quad (2.32)$$

Отже, рівняння прямої регресії має вигляд:

$$f(x) = 0.98x + 0.08 \quad (2.33)$$

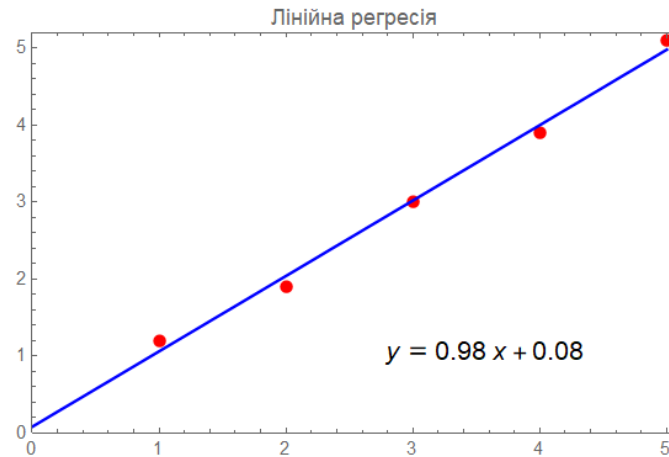


Рис. 2.1. Апроксимація набору дискретних точок методом лінійної регресії

Поліноміальна регресія: приклад

У випадку, якщо залежність має криволінійний характер, варто використати поліном другого або вищого степеня. Наприклад, побудуємо поліном другого степеня:

$$f(x) = a_0 + a_1x + a_2x^2$$

Ця функція дозволяє гнучкіше описати вигин у даних. Коефіцієнти a_0 , a_1 , a_2 визначаються розв'язанням системи рівнянь, аналогічно до лінійного випадку, але з більшою кількістю змінних.

Приклад: Припустимо, маємо дані:

x_i	1	2	3	4
y_i	1.1	4.1	8.8	16.3

Можна помітити, що залежність наближено квадратична ($y \approx x^2$), отже доцільно використати поліном ступеня 2.

Нелінійна регресія

У деяких випадках природа процесу підказує використати спеціальні функції. Наприклад:

$$f(x) = ae^{-bx} + c \quad \text{або} \quad f(x) = a \sin(bx)$$

Коефіцієнти у таких моделях зазвичай підбираються чисельними методами, наприклад, методом найменших квадратів за допомогою ітераційних алгоритмів.

Оцінка якості регресії

Після побудови моделі важливо оцінити її якість. Для цього розраховують **Середньоквадратичну похибку** :

$$D = \frac{1}{n} \sum_{i=1}^n (y_i - f(x_i))^2$$

Чим менше значення D тим краща апроксимація.

Вибір виду регресії

При виборі моделі слід враховувати:

- форму графіка даних — лінійна, квадратична, експоненціальна тощо;
- фізичний або емпіричний зміст залежності;
- мету побудови — прогноз, пояснення чи згладжування;
- простоту інтерпретації та кількість параметрів.

Розділ 3

Чисельне розв'язування рівнянь

Коренем рівняння $f(x) = 0$ називається таке значення $x = x^*$, підстановка якого у рівняння перетворює його на правильну числову рівність. Наприклад, якщо у рівняння

$$x^2 + 5x + 4 = 0$$

підставити $x = -1$, то отримаємо $0 = 0$, що є правильною рівністю.

Розв'язати рівняння — означає знайти всі його корені.

Однак далеко не кожне рівняння має аналітичне розв'язання:

- а) Трансцендентні рівняння, як правило, не розв'язуються аналітично, за винятком окремих «шкільних» випадків, коли рівняння вдається вдалою підстановкою звести до алгебраїчного. Наприклад:

$$e^{2x} - 10e^x + 25 = 0$$

- б) Навіть для алгебраїчного рівняння вигляду

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$$

степеня вище четвертого не існує формули, яка б виражала корені через коефіцієнти рівняння за допомогою скінченного числа арифметичних операцій і добування коренів. У деяких окремих випадках, наприклад, для рівняння

$$x^{42} - 5x^{21} + 4 = 0$$

такі формули можуть існувати, але в загальному випадку — ні.

Неможливість загального аналітичного розв'язання рівнянь п'ятого і вищих степенів була доведена працями Абеля (1802–1829) та Галуа (1811–1832).

Отже, завдання наближеного, чисельного знаходження коренів рівнянь набуває особливої важливості. З цією метою:

- визначають кількість коренів рівняння та ізолюють кожен з них. *Відрізком ізоляції (локалізації) кореня* називають інтервал, на якому знаходиться лише один корінь рівняння;
- обчислюють кожен корінь з заданою точністю.

Для ізоляції коренів рівняння $f(x) = 0$ використовують **графічні** та **аналітичні** методи.

У першому з них будують графік функції $y = f(x)$ і наближено знаходять точки його перетину з віссю абсцис Ox .

3.1. Метод Ньютона

Нехай в рівнянні $f(x) = 0$ функція $f(x)$ має неперервну похідну $f'(x) \neq 0$; x_n — деяке наближення до кореня $\bar{x}$ розглядуваного рівняння. В околі точки x_n розкладемо функцію $f(x)$ у ряд Тейлора:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2!}(x - x_n)^2 + \dots + \frac{f^{(k)}(x_n)}{k!}(x - x_n)^k + \dots \quad (3.1)$$

Обмежимося членами, лінійними за x , включно:

$$0 = f(x) \approx f(x_n) + f'(x_n)(x - x_n) \quad (3.2)$$

Звідси:

$$x \approx x_n - \frac{f(x_n)}{f'(x_n)} \quad (3.3)$$

Згідно з ідеєю Ньютона, ліву частину цього виразу розглядатимемо як наступне, $(n + 1)$ -е, наближення деякого ітераційного процесу:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \dots \quad (3.4)$$

Цей ітераційний процес можна розглянути графічно (див. рис. 3.1). Нехай задано x_0 - початкове наближення кореня рівняння $f(x) = 0$. Побудуємо дотичну до функції $f(x)$ в цій точці і знайдемо x_1 точку перетину дотичної з віссю Ox . Записавши рівняння дотичної $y(x) = f'(x_0)(x - x_0) + f(x_0)$, і враховуючи $y(x_1) = 0$, отримуємо ітераційну формулу методу Ньютона.

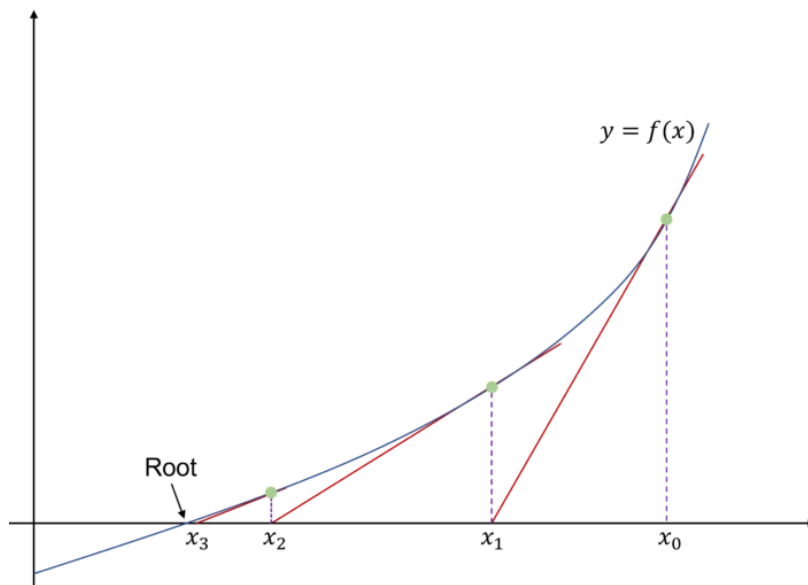


Рис. 3.1. Графічне зображення методу Ньютона.

Бувають випадки розбіжності кореня в ітераційному процесі. Приклад такої розбіжності зображено на рис.3.2. З рисунка видно, що причиною розбіжності є зміна знаку випуклості функції $f(x)$ на відрізку локалізації кореня. Отже, для надійної збіжності методу Ньютона необхідно задати таке початкове значення x_0 , при якому на відрізку локалізації кореня похідні $f'(x)$ та $f''(x)$ не змінюють знак.

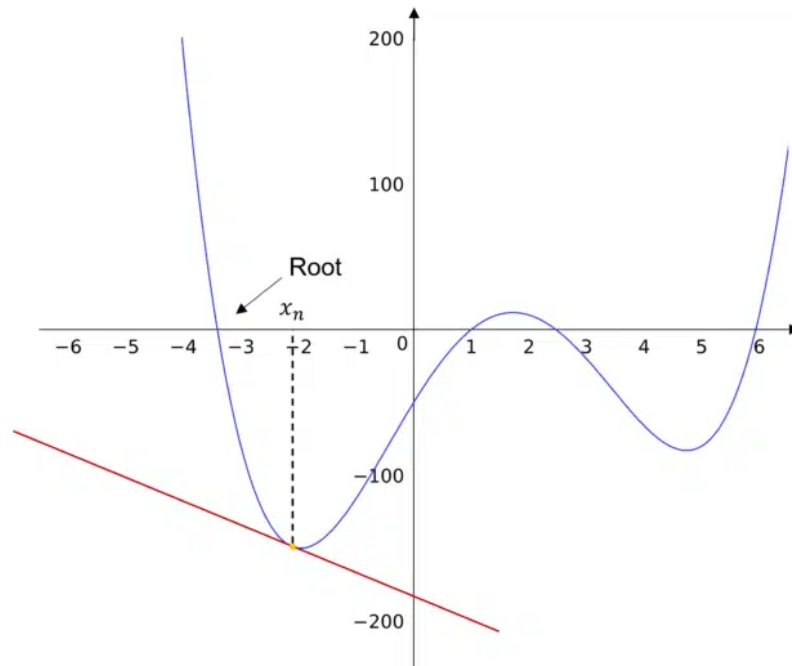


Рис. 3.2. Графічне зображення випадку розбіжності у методі Ньютона.

3.2. Метод половинного ділення (дихотомія)

Метод безпосередньо впливає з аналітичного способу ізоляції коренів. Нехай для рівняння $f(x) = 0$ знайдено **первинний відрізок** $[a, b]$ ізоляції кореня. Обчислимо середину відрізка $x_m = \frac{a+b}{2}$.

Якщо випадково виявиться, що $f(x_m) = 0$, то x_m є коренем рівняння $f(x) = 0$. Якщо ж $f(x_m) \neq 0$, то з двох половин $[a, x_m]$, $[x_m, b]$ первинного відрізка виберемо для подальшого ділення навпіл ту, на кінцях якої функція $f(x)$ набуває **значень протилежних знаків**. Для випадку, що зображено на рис. 3.3, таким відрізком буде $[x_m, b]$, оскільки $f(x_m) * f(b) < 0$. Обраний відрізок знову поділяємо навпіл і знаходимо половину з протилежними знаками $f(x)$ на кінцях і т.д.

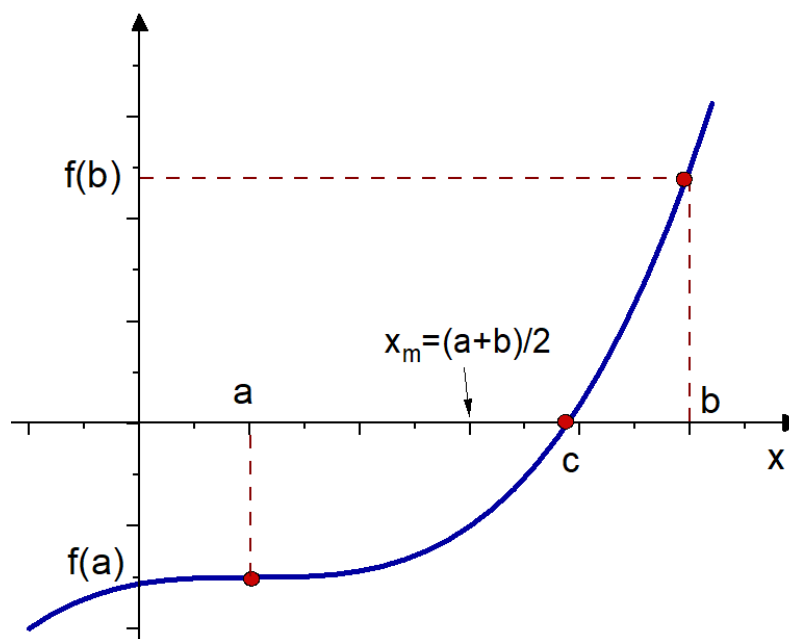


Рис. 3.3. Метод половинного ділення

Критерій досягнення потрібної точності (критерій зупинки обчислень): якщо корінь потрібно обчислити з точністю ε , то ділення навпіл продовжують доти, поки довжина відрізка не стане меншою за ε , тобто середина цього відрізка дає значення кореня з точністю ε .

Властивості дихотомії наступні:

- а) надзвичайно простий метод;
- б) не висуває вимог до властивостей функції $f(x)$ — вона повинна бути лише неперервною, диференційовність не передбачається;
- в) має істотний недолік — властиву йому повільну збіжність:
Нехай, наприклад, первинний відрізок ізоляції має одиничну довжину. Після першого поділу довжина ділянки дихотомії — до $(1/2)^1$, після другого — до $(1/2)^2$, і т.д. Оскільки $(1/2)^{10} = 1/1024 < 0,001$, то **після десяти кроків** дихотомії забезпечується точність менша ніж 10^{-3} ;
- г) метод **непридатний** для обчислення коренів парної кратності;
- г) **непридатний** для розв'язування **систем рівнянь**.

3.3. Метод січних

Враховуючи те, що математично похідна вводиться як границя відношення приросту функції до приросту її аргументу:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{\Delta f(x)}{\Delta x} = \lim_{z \rightarrow x} \frac{f(z) - f(x)}{z - x}, \quad (3.5)$$

то наближене значення похідної можна записати так:

$$f'(x) \approx \frac{f(z) - f(x)}{z - x}. \quad (3.6)$$

Відповідно до цього, у методі січних похідна наближено обчислюється за формулою:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}. \quad (3.7)$$

Підставляючи у формулу (3.4) цей вираз для похідної, отримуємо наступний ітераційний процес:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n), \quad n = 1, 2, 3, \dots, \quad (3.8)$$

для якого потрібно задати два початкові наближення x_0 та x_1 (на відміну від методу Ньютона, де потрібно було задати лише x_0).

З геометричної точки зору в методі січних (3.8) дотична замінюється січною, що проходить через точки кривої $y = f(x)$ з абсцисами x_n і x_{n-1} (рис.3.4).

Властивості методу січних полягають у наступному:

- а) оскільки метод січних є модифікацією методу Ньютона (методу дотичних), він успадковує всі властивості останнього;
- б) швидкість збіжності ітераційного процесу нижча, ніж у методі дотичних (внаслідок грубошого наближення похідної за формулою (3.7)), але залишається високою;
- в) похідна $f'(x_n)$ виключається із обчислювального процесу.

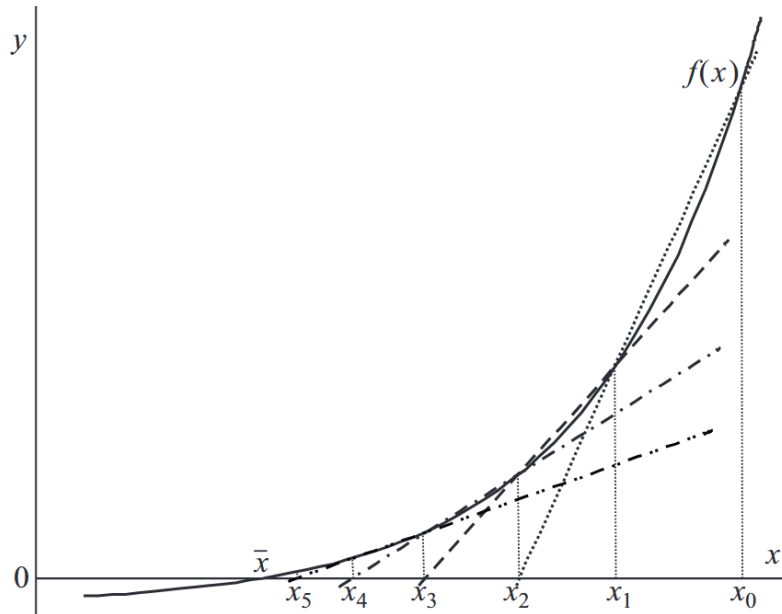


Рис. 3.4. Геометрична ілюстрація методу січних

3.4. Метод хорд

Застосуємо у формулі (3.4) ще грубіше наближення для похідної:

$$f'(x_n) \approx \frac{f(x_n) - f(x_0)}{x_n - x_0}$$

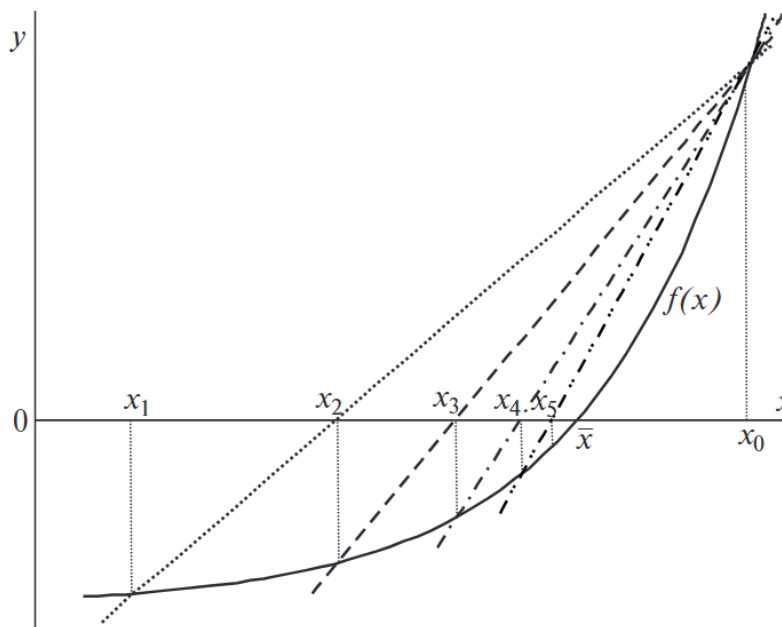


Рис. 3.5. Геометрична ілюстрація методу хорд

Підставляючи у формулу (3.4) цей вираз для похідної, отримаємо ітераційну формулу **методу хорд**:

$$x_{n+1} = x_n - \frac{x_n - x_0}{f(x_n) - f(x_0)} f(x_n), \quad n = 1, 2, 3, \dots \quad (3.9)$$

Для цього, як і в методі січних, необхідно задати **два** початкових наближення x_0 і x_1 . Геометрично у методі хорд **дотична** замінюється **хордою**, що проходить через точки кривої $y = f(x)$ з абсцисами x_n і x_0 (рис. 3.5).

Метод хорд має ті самі властивості, що й метод січних, але швидкість збіжності, за інших рівних умов, стає ще трохи нижчою, ніж у останнього (хоча і залишається високою).

Метод хорд називають ще *Regula Falsi*. Ця назва походить з латинської мови й означає «правило помилкового припущення» або «метод хибного положення». Цей термін використовували ще в античних та середньовічних математичних текстах для позначення методу розв'язання рівнянь за допомогою наближених гіпотез.

Суть класичного підходу полягала в тому, щоб:

1. зробити довільне припущення щодо розв'язку,
2. підставити його у рівняння,
3. обчислити помилку,
4. скоригувати припущення пропорційно до цієї помилки.

У чисельному аналізі ця ідея трансформувалась у сучасний **метод хорд**, або **метод січних**, який використовує два наближення a і b , між якими функція змінює знак, і будує січну (хорду) між точками $(a, f(a))$ і $(b, f(b))$. Перетин цієї прямої з віссю x використовується як нове наближення до кореня.

Таким чином, метод *Regula Falsi* є геометричним розвитком стародавнього принципу “хибної гіпотези”, що лежить в основі ітеративного наближення.

Завдання для самостійної роботи

Для заданого рівняння $f(x) = 0$ знайти один з його коренів методами дихотомії, ітерацій, Ньютона, хорд і січних; досягти точності 10^{-2} методом дихотомії і 10^{-3} рештою методів.

- | | |
|----------------------------|------------------------------|
| 1. $\ln x + x - 4 = 0$ | 16. $2 \ln x + x - 2 = 0$ |
| 2. $2 \ln x + x^2 - 8 = 0$ | 17. $4 \ln x + x^2 - 3 = 0$ |
| 3. $\ln x + 2x^2 - 6 = 0$ | 18. $2 \ln x + 2x^2 - 6 = 0$ |
| 4. $2 \ln x - x^2 + 5 = 0$ | 19. $\sin x - x^2 + 4 = 0$ |
| 5. $\ln x + 2x - 3 = 0$ | 20. $\sin x - x^2 - 1 = 0$ |
| 6. $\ln x + x^2 - 3 = 0$ | 21. $3 \sin x + x - 2 = 0$ |
| 7. $\ln x + 2x + 1 = 0$ | 22. $3 \sin x + x^2 - 1 = 0$ |
| 8. $\sin x - x^4 = 0$ | 23. $3 \sin x - x^2 = 0$ |
| 9. $\sin x - x^3 = 0$ | 24. $\cos x + x - 2 = 0$ |
| 10. $\sin x + x^3 = 0$ | 25. $\cos x + 3x - 6 = 0$ |
| 11. $3 \sin x + x^4 = 0$ | 26. $\cos x + 2x - 4 = 0$ |
| 12. $3 \sin x + x^2 = 0$ | 27. $\ln x + 0,25x - 3 = 0$ |
| 13. $3 \sin x - x^3 = 0$ | 28. $\ln x + 2x^2 - 5 = 0$ |
| 14. $\cos x - x = 0$ | 29. $\ln x + x^2 - 3 = 0$ |
| 15. $\ln x + 2x - 3 = 0$ | 30. $\ln x + x^2 - 2 = 0$ |

Розділ 4

Чисельне інтегрування

Якщо функція $f(x)$ задана аналітично (формулою), і її первісна $F(x)$ є елементарною функцією, тоді означений інтеграл

$$\int_a^b f(x) dx \quad (4.1)$$

обчислюється за формулою Ньютона—Лейбніца:

$$\int_a^b f(x) dx = F(x) \Big|_a^b = F(b) - F(a). \quad (4.2)$$

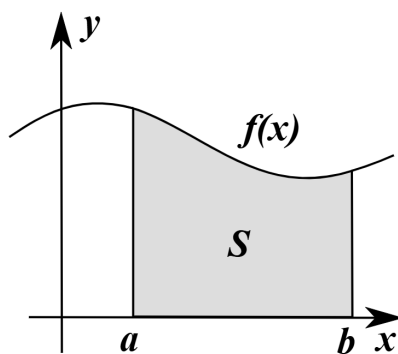


Рис. 4.1. Геометричний зміст визначеного інтегралу.

Існують ситуації, коли цією формулою неможливо або складно скористатися:

1. Підінтегральна функція $f(x)$ задана графічно або таблично; тоді первісна $F(x)$ не існує.
2. Підінтегральна функція $f(x)$ задана аналітично, але інтеграл $\int f(x) dx$ **не береться**, тобто не виражається в скінченному вигляді через елементарні функції (відомо, що багато важливих інтегралів, які часто трапляються в прикладних задачах, належать до таких). Наприклад:

$$\int e^{-x^2} dx, \quad \int \frac{\sin x}{x} dx;$$

3. Підінтегральна функція $f(x)$ задана аналітично, і інтеграл $\int f(x) dx$ береться, але первісна $F(x)$ занадто громіздка.

У всіх таких випадках доводиться вдаватися до чисельного (наближеного) обчислення означеного інтегралу. Для цього підінтегральну функцію $f(x)$ замінюють іншою, «близькою» до неї функцією, яка легко інтегрується.

4.1. Формула Ньютона — Котеса

Як функцію, «близьку» до $f(x)$, візьмемо інтерполяційний поліном Лагранжа $L_m(x)$, який збігається з $f(x)$ у вузлах інтерполяції $x_0, x_1, \dots, x_m$, що лежать на відрізку інтегрування $[a, b]$. Поліном Лагранжа має вигляд:

$$L_m(x) = \sum_{i=0}^m f(x_i) L_m^i(x), \quad m = 0, 1, 2, \dots \quad (4.3)$$

де $L_m^i(x)$ — коефіцієнти Лагранжа (поліноми степеня m): $L_0^0(x) = 1$,

$$L_m^i(x) = \frac{(x - x_0)(x - x_1) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_m)}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_m)}. \quad (4.4)$$

Якщо поліном Лагранжа «близький» до функції $f(x)$, то й відповідні інтеграли повинні бути близькими:

$$\int_a^b f(x) dx \approx \int_a^b L_m(x) dx = \sum_{i=0}^m f(x_i) \int_a^b L_m^i(x) dx. \quad (4.5)$$

Уводячи так звані коефіцієнти Котеса (1682–1716):

$$c_m^i = \int_a^b L_m^i(x) dx, \quad (4.6)$$

отримуємо формулу Ньютона — Котеса порядку m :

$$\int_a^b f(x) dx \approx \int_a^b L_m(x) dx = \sum_{i=0}^m c_m^i f(x_i). \quad (4.7)$$

Вона дає змогу наближено подати значення означеного інтеграла у вигляді лінійної комбінації значень підінтегральної функції у вузлах інтерполяції.

Задача. Обчислити коефіцієнти Котеса c_0^1 та c_1^1 , які відповідають апроксимації функції $f(x)$ поліномом першого степеня у двох вузлах.

Розв’язання. Нехай значення функції $f(x)$ відомі у двох точках $x_0 = a$ та $x_1 = b$. У цьому випадку функцію можна апроксимувати лінійним поліномом Лагранжа (першого порядку):

$$f(x) \approx L_1(x) = f(a) \cdot \frac{x - b}{a - b} + f(b) \cdot \frac{x - a}{b - a} \quad (4.8)$$

Обчислимо означений інтеграл від апроксимованої функції:

$$\int_a^b f(x) dx \approx \int_a^b \left[f(a) \cdot \frac{x - b}{a - b} + f(b) \cdot \frac{x - a}{b - a} \right] dx = \frac{b - a}{2} [f(a) + f(b)] \quad (4.9)$$

Таким чином, отримуємо вагові коефіцієнти Котеса першого порядку, які застосовуються до значень функції у вузлах a та b : $c_0^1 = c_1^1 = \frac{b-a}{2}$. Цей результат має наочне геометричне тлумачення: у найпростішому (лінійному) наближенні означений інтеграл можна розглядати як площу трапеції, основою якої є відрізок $[a, b]$, а висотами — значення функції в кінцях інтервалу (див. рис. 4.2).

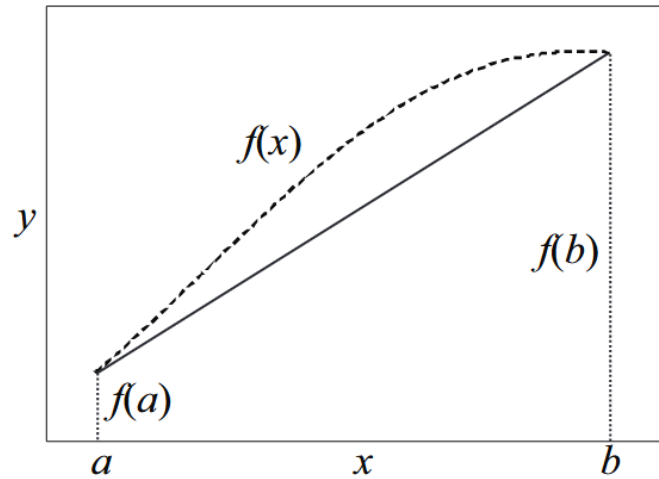


Рис. 4.2. Геометрична інтерпретація формули Ньютона—Котеса при $m = 1$: площа криволінійної трапеції апроксимується трапецією з висотами $f(a)$ та $f(b)$.

Табличні коефіцієнти Котеса та їх застосування

Як ми вже переконалися, обчислення коефіцієнтів Котеса c_m^i не становить складності при малих значеннях m , але при зростанні порядку апроксимації воно швидко ускладнюється. На практиці, однак, немає потреби кожного разу обчислювати ці коефіцієнти вручну, оскільки їхні значення зазвичай попередньо табульовані.

Припустимо, що вузли інтерполяції є рівновіддаленими з кроком h , тобто:

$$x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \quad \dots, \quad x_m = a + mh = b. \quad (4.10)$$

У цьому випадку таблиця Котеса має вигляд:

Табл. 4.1. Коефіцієнти c_m^i для рівновіддалених вузлів

m	Коефіцієнти Котеса c_m^i
1	$c_0^1 = c_1^1 = \frac{b-a}{2}$
2	$c_0^2 = c_2^2 = \frac{b-a}{6}, \quad c_1^2 = \frac{4(b-a)}{6}$
3	$c_0^3 = c_3^3 = \frac{b-a}{8}, \quad c_1^3 = c_2^3 = \frac{3(b-a)}{8}$
4	$c_0^4 = c_4^4 = \frac{7(b-a)}{90}, \quad c_1^4 = c_3^4 = \frac{32(b-a)}{90}, \quad c_2^4 = \frac{12(b-a)}{90}$
5	$c_0^5 = c_5^5 = \frac{19(b-a)}{288}, \quad c_1^5 = c_4^5 = \frac{75(b-a)}{288}, \quad c_2^5 = c_3^5 = \frac{50(b-a)}{288}$
6	$c_0^6 = c_6^6 = \frac{41(b-a)}{840}, \quad c_1^6 = c_5^6 = \frac{216(b-a)}{840},$ $c_2^6 = c_4^6 = \frac{27(b-a)}{840}, \quad c_3^6 = \frac{272(b-a)}{840}$

Приклад застосування коефіцієнтів Котеса. Нехай функція $f(x)$ задана у трьох точках: на краях відрізка a і b , а також у середині $h = \frac{a+b}{2}$. У цьому випадку використовуємо формулу Котеса для $m = 2$ (три вузли), і підставляємо відповідні коефіцієнти:

$$\int_a^b f(x) dx \approx c_0^2 f(a) + c_1^2 f\left(\frac{a+b}{2}\right) + c_2^2 f(b) = \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]. \quad (4.11)$$

Цей вираз збігається з класичною квадратурною формулою Сімпсона, яка поєднує високу точність з обчислювальною простотою.

Оцінка похибки формули Котеса

Похибку апроксимації формули Котеса можна оцінити за наступною формулою, яка отримується з розкладу в ряд Тейлора:

$$R_m \leq \frac{M_{m+1}(b-a)^{m+2}}{(m+2)!}, \quad (4.12)$$

де M_{m+1} — оцінка максимальної абсолютної величини похідної порядку $m+1$ на відрізку $[a, b]$.

Таким чином, похибку можна зменшити двома шляхами:

- збільшенням порядку апроксимації m ;
- зменшенням довжини інтервалу $b-a$.

Проте перший підхід є малоефективним на практиці: при великих m формула Ньютона—Котеса стає громіздкою та незручною для обчислень. Значно ефективніше поділити відрізок інтегрування на менші підінтервали (з рівним кроком h), на кожному з яких застосовувати просту формулу Котеса невисокого порядку — наприклад, $m=0$ (метод прямокутників), $m=1$ (метод трапецій) або $m=2$ (формула Сімпсона). Цей підхід лежить в основі найбільш уживаних простих методів чисельного інтегрування.

4.2. Метод прямокутників

Розіб'ємо відрізок інтегрування $[a, b]$ на n рівних підвідрізків з кроком $h = \frac{b-a}{n}$, утворивши вузли:

$$a_0 = a, \quad a_1 = a + h, \quad a_2 = a + 2h, \quad \dots, \quad a_n = a + nh = b.$$

На кожному елементарному підвідрізку $[a_{k-1}, a_k]$, де $k = 1, 2, \dots, n$, апроксимуємо функцію $f(x)$ поліномом Лагранжа нульового степеня, що набуває значення $f(a_{k-1})$ на лівому кінці підвідрізка:

$$L_0(x) = f(a_{k-1}) = y_{k-1}.$$

Геометрично це означає, що криволінійна трапеція, обмежена зверху графіком $f(x)$, замінюється східчастою фігурою з прямокутників (див. рис. 4.3).

Оскільки отримана фігура складається з прямокутників з висотами $y_i = f(a_i)$, $i = 0, 1, \dots, n-1$, і основою h , то:

$$\int_a^b f(x) dx \approx h(y_0 + y_1 + \dots + y_{n-1}) = h \sum_{i=0}^{n-1} f(a_i). \quad (4.13)$$

Це є формулою прямокутників для наближеного обчислення визначеного інтеграла.

Для оцінки похибки скористаємося формулою (4.12) на одному елементарному відрізку $[a_{k-1}, a_k]$:

$$\int_{a_{k-1}}^{a_k} f(x) dx - \int_{a_{k-1}}^{a_k} L_0(x) dx = \frac{M_1 h^2}{2},$$

де $M_1 = \max_{x \in [a, b]} |f'(x)|$. Для всього відрізка $[a, b]$, який містить n підвідрізків, маємо сумарну похибку:

$$R_{\text{прям}} = \frac{M_1 h^2}{2} \cdot n = \frac{M_1}{2} (b-a) h = O(h). \quad (4.14)$$

Таким чином, похибка формули прямокутників зростає лінійно з кроком h .

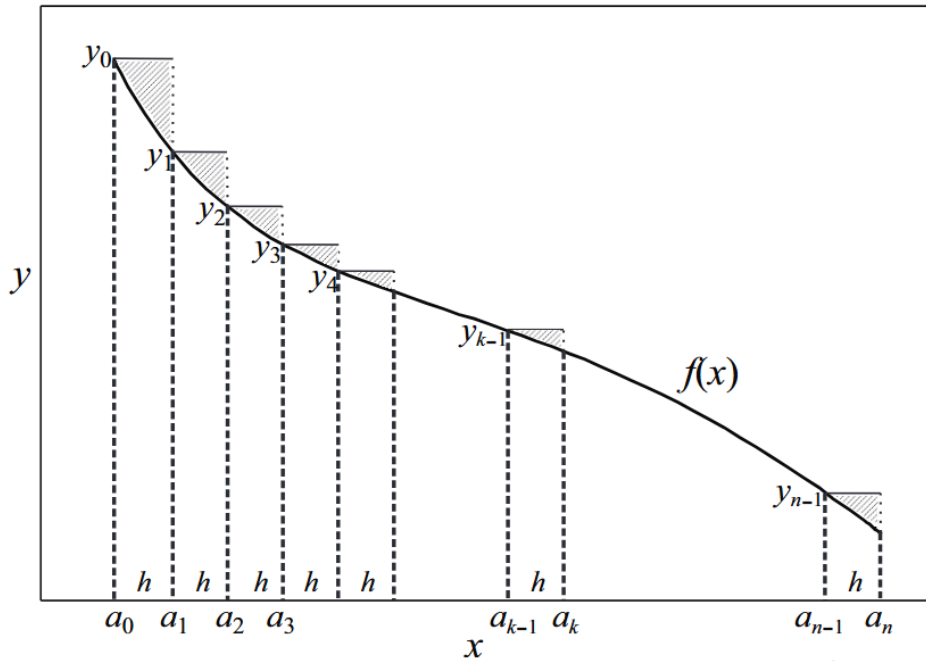


Рис. 4.3. Приблизне обчислення інтеграла за формулою прямокутників.

4.3. Метод трапецій

Поділимо відрізок $[a, b]$ на ті самі n рівних частин з кроком $h = \frac{b-a}{n}$, але тепер на кожному підвідрізку $[a_{k-1}, a_k]$ апроксимуємо функцію $f(x)$ лінійним поліномом Лагранжа з вузлами $x_0 = a_{k-1}$, $x_1 = a_k$. Це відповідає апроксимації дуги хордою.

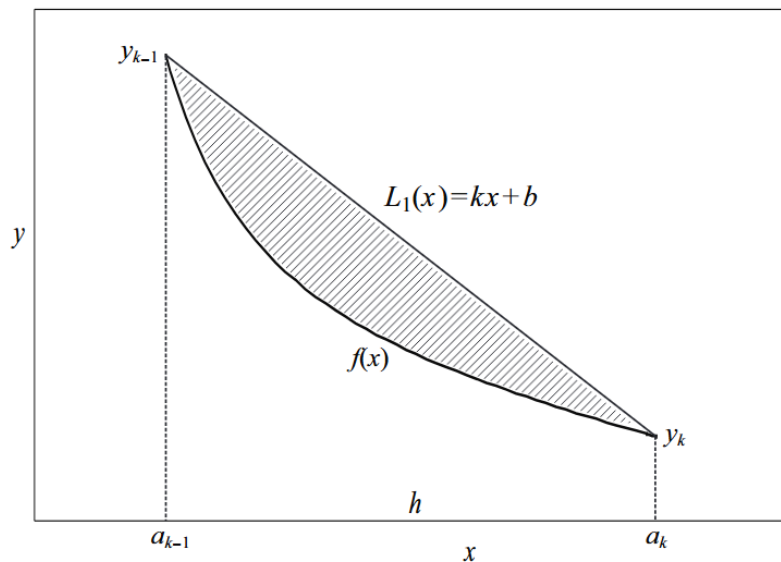


Рис. 4.4. Приблизне обчислення інтеграла за формулою трапецій.

Площа однієї трапеції дорівнює:

$$\frac{y_{k-1} + y_k}{2} \cdot h.$$

Отже, інтеграл наближено дорівнює сумі площ усіх трапецій:

$$\int_a^b f(x) dx \approx \frac{h}{2} (y_0 + 2y_1 + 2y_2 + \cdots + 2y_{n-1} + y_n). \quad (4.15)$$

Похибка для одного підвідрізка:

$$\int_{a_{k-1}}^{a_k} f(x) dx - \int_{a_{k-1}}^{a_k} L_1(x) dx = -\frac{M_2 h^3}{12},$$

де $M_2 = \max_{x \in [a,b]} |f''(x)|$, тому сумарна похибка:

$$R_{\text{трап}} = -\frac{M_2 h^3}{12} \cdot n = -\frac{M_2}{12} (b-a) h^2 = O(h^2). \quad (4.16)$$

4.4. Метод Сімпсона

Для методу Сімпсона розіб'ємо відрізок $[a, b]$ на парну кількість підвідрізків $n = 2s$. Для кожної пари сусідніх відрізків апроксимуємо $f(x)$ поліномом Лагранжа другого степеня за трьома вузлами a_{2k-2} , a_{2k-1} , a_{2k} .

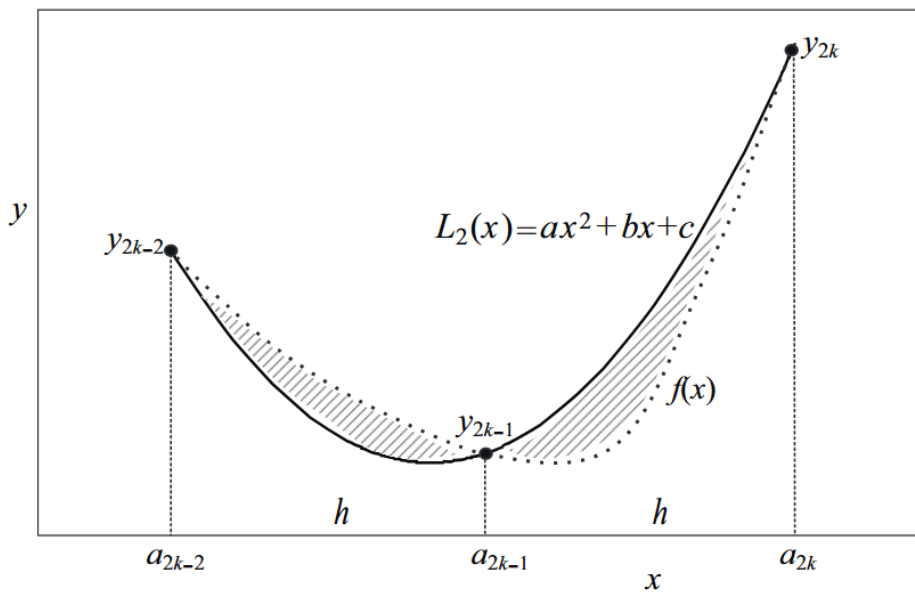


Рис. 4.5. Приблизне обчислення інтеграла за формулою Сімпсона.

Інтеграл на одному подвійному відрізку:

$$\int_{a_{2k-2}}^{a_{2k}} f(x) dx \approx \frac{h}{3} (y_{2k-2} + 4y_{2k-1} + y_{2k}).$$

Загальна формула Сімпсона.

Припустимо, що потрібно обчислити визначений інтеграл:

$$\int_a^b f(x) dx$$

і розбити інтервал $[a, b]$ на $2n$ рівних частин, де n — ціле число.

Метод Сімпсона використовує наступну формулу:

$$\int_a^b f(x) dx \approx \frac{h}{3} [y_0 + 4(y_1 + y_3 + \dots + y_{n-1}) + 2(y_2 + y_4 + \dots + y_{n-2}) + y_n]. \quad (4.17)$$

або через знак суми:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(x_0) + 4 \sum_{i=1,3,\dots}^{2n-1} f(x_i) + 2 \sum_{i=2,4,\dots}^{2n-2} f(x_i) + f(x_{2n}) \right], \quad (4.18)$$

де $h = \frac{b-a}{2n}$. У формулі (4.18) перша сума має на один доданок більше ніж друга, тому часто, для простішої реалізації алгоритму розрахунку, цю формулу модифікують, збільшуючи кількість доданків в другій сумі та змінюючи знак останнього доданку:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(x_0) + 4 \sum_{i=1,3,\dots}^{2n-1} f(x_i) + 2 \sum_{i=2,4,\dots}^{2n} f(x_i) - f(x_{2n}) \right] \quad (4.19)$$

Похибка для одного подвійного відрізка:

$$R_k = -\frac{M_4 h^5}{90}, \quad M_4 = \max_{x \in [a,b]} |f^{(4)}(x)|,$$

і сумарна похибка:

$$R_{\text{Сімп}} = -\frac{M_4 h^4}{180} (b-a) = O(h^4). \quad (4.20)$$

Висновок: Формула Сімпсона дає значно кращу точність порівняно з формулами трапецій і прямокутників, особливо при гладких функціях $f(x)$, оскільки похибка зменшується пропорційно до h^4 .

4.5. Порівняння чисельних методів інтегрування

Приклад. Обчислити інтеграл

$$I = \int_1^2 \frac{1}{x} dx$$

за формулою Ньютона—Котеса для $m = 4$, а також за формулами прямокутників, трапецій та Сімпсона при $n = 4$; порівняти результати з точним значенням інтеграла.

Розв’язання. Обчислення проводимо з точністю до п’яти десяткових знаків. Оскільки підінтегральна функція $f(x) = \frac{1}{x}$ задана аналітично, ми можемо обчислити точне значення інтеграла за формулою Ньютона—Лейбніца:

$$I = \int_1^2 \frac{1}{x} dx = \ln 2 = \mathbf{0.69315 \dots}$$

— нескінченний неперіодичний десятковий дріб.

Нехай крок між вузлами дорівнює $h = \frac{b-a}{m} = \frac{2-1}{4} = 0.25$. Тоді отримаємо таблицю значень:

Табл. 4.2. Значення функції $f(x) = \frac{1}{x}$ у вузлах для $m = 4$

k	$x_k = x_0 + kh$	$y_k = \frac{1}{x_k}$
0	$x_0 = 1.00$	$y_0 = 1.000$
1	$x_1 = 1.25$	$y_1 = 0.800$
2	$x_2 = 1.50$	$y_2 = 0.667$
3	$x_3 = 1.75$	$y_3 = 0.571$
4	$x_4 = 2.00$	$y_4 = 0.500$

Формула Ньютона—Котеса (порядку $m = 4$)

Використаємо табличні коефіцієнти Котеса:

$$I_{\text{Cotes}} = \sum_{i=0}^4 c_4^i y_i = \frac{7}{90} y_0 + \frac{32}{90} y_1 + \frac{12}{90} y_2 + \frac{32}{90} y_3 + \frac{7}{90} y_4 \approx \mathbf{0.69318}.$$

Порівнюючи з точним значенням $\ln 2 \approx 0.69315$, бачимо, що формула Ньютона—Котеса дає **4 правильні десяткові знаки**.

Формула лівих прямокутників

$$I_{\text{прям}} = h(y_0 + y_1 + y_2 + y_3) = 0.25(1.000 + 0.800 + 0.667 + 0.571) = \mathbf{0.75953}.$$

Жодного правильного знака після коми.

Формула трапецій

$$I_{\text{трап}} = \frac{h}{2}[y_0 + 2(y_1 + y_2 + y_3) + y_4] = \mathbf{0.69702}.$$

Два правильні знаки після коми.

Формула Сімпсона

$$I_{\text{Сімп}} = \frac{h}{3}[y_0 + 4(y_1 + y_3) + 2y_2 + y_4] = \mathbf{0.69325}.$$

Три правильні десяткові знаки.

Висновок

Найточнішим серед усіх методів виявився метод Ньютона—Котеса. Його похибка найменша, однак зауважимо, що цей метод потребує попередньо табульованих коефіцієнтів і є досить громіздким для практичного використання при великих m .

Формула Сімпсона, будучи значно простішою у реалізації, дає високу точність і практично не поступається методу Ньютона—Котеса. Формула трапецій забезпечує задовільну точність, а формула прямокутників — найменш точна при обраному кроці h , що робить її малопридатною для точних обчислень.

Таким чином, формули трапецій і Сімпсона є найкращими з точки зору практичного застосування: вони поєднують простоту реалізації з прийнятною точністю.

4.6. Обчислення багатовимірних інтегралів

У багатьох фізичних задачах виникає необхідність усереднення функції за багатьма змінними. Наприклад, припустімо, що відома залежність повної енергії системи з десяти взаємодіючих частинок від їх координат і швидкостей. Оскільки в тривимірному просторі кожна частинка має по три координати і три компоненти швидкості, повна енергія системи є функцією $10 \times 3 = 30$ змінних.

Отже, для обчислення середньої енергії такої системи необхідно обчислити 30-вимірний визначений інтеграл. Якщо розбити область зміни кожної змінної на n частин, то загальна кількість точок, у яких потрібно обчислювати функцію, становитиме n^{30} . Очевидно, що при такій розмірності класичні чисельні методи інтегрування стають непридатними через експоненціальне зростання обчислювальної складності. Тим не менш, ці методи залишаються застосовними у випадках, коли кількість змінних (кратність інтегралу) не перевищує п'яти.

Найпростішим підходом до обчислення багатовимірних інтегралів є зведення їх до добутку одномірних інтегралів, якщо функція допускає відповідне розділення змінних.

Розглянемо двохвимірні та трьохвимірні інтеграли, використовуючи метод зведення їх до одновимірних інтегралів, а також метод Монте-Карло, який застосовують для випадку інтегралів вищих розмірностей.

Подвійний інтеграл

Нехай у замкненій області D площини Oxy задано неперервну функцію $z = f(x, y)$. Розіб'ємо область D на n елементарних підобластей D_i , площі яких позначимо ΔS_i , а діаметри — d_i . У кожній області D_i виберемо довільну точку $M_i(x_i, y_i)$ і помножимо значення функції в цій точці на площу ΔS_i . Сума добутків має вигляд:

$$\sum_{i=1}^n f(x_i, y_i) \Delta S_i,$$

та називається інтегральною сумою функції $f(x, y)$ на області D .

Розглянемо границю цієї суми при $n \rightarrow \infty$, за умови що $\max d_i \rightarrow 0$. Якщо така границя існує і не залежить ні від способу розбиття області D , ні від вибору точок M_i , то ця границя називається подвійним інтегралом функції $f(x, y)$ на області D і позначається:

$$\iint_D f(x, y) dx dy = \lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i, y_i) \Delta S_i. \quad (4.21)$$

Тут $f(x, y)$ — функція, яку інтегруємо; D — область інтегрування; $dx dy$ — елемент площі.

Теорема. Якщо функція $z = f(x, y)$ є неперервною в замкненій області D , то вона інтегровна на цій області.

Потрійний інтеграл

Нехай у замкненій області V простору $Oxyz$ задано неперервну функцію $u = f(x, y, z)$. Розіб'ємо область V сіткою поверхонь на n елементарних підобластей V_i , кожна з яких має об'єм ΔV_i , і виберемо в кожній підобласті довільну точку $M_i(x_i, y_i, z_i)$. Тоді інтегральна сума має вигляд:

$$\sum_{i=1}^n f(x_i, y_i, z_i) \Delta V_i.$$

Якщо при $n \rightarrow \infty$ і $\max \Delta V_i \rightarrow 0$ ця сума збігається і не залежить від способу розбиття області та вибору точок, то її границя називається **потрійним інтегралом** функції $f(x, y, z)$ по області V і позначається:

$$\iiint_V f(x, y, z) dx dy dz = \lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i, y_i, z_i) \Delta V_i. \quad (4.22)$$

Теорема. Якщо функція $u = f(x, y, z)$ є неперервною в замкненій обмеженій області V , то потрійний інтеграл існує і не залежить від способу розбиття V на частини, а також від вибору точок M_i у них.

Зведення багатовимірного інтегралу до повторних одновимірних

У випадках, коли область інтегрування має прямокутну форму або може бути до неї приведена за допомогою заміни змінних, багатовимірний інтеграл зручно обчислювати шляхом поетапного (ітераційного) інтегрування за кожною змінною. Такий підхід називається **повторним інтегруванням**, а інтеграл — **повторним кратним інтегралом**.

Зокрема, подвійний інтеграл на прямокутній області $D = [a, b] \times [c, d]$ можна записати у вигляді:

$$\iint_D f(x, y) dx dy = \int_c^d \left(\int_a^b f(x, y) dx \right) dy. \quad (4.23)$$

Завдяки такому поданню подвійного інтеграла його чисельне обчислення можна реалізувати як два вкладені одновимірні інтеграли, для яких застосовуються класичні методи, наприклад, формула трапецій або Сімпсона.

Схема чисельного обчислення:

1. Вибираємо вузли $y_0, y_1, \dots, y_n$ на відрізку $[c, d]$.
2. Для кожного фіксованого y_j розглядаємо функцію $f(x, y_j)$ як функцію змінної x , і обчислюємо наближене значення одновимірного інтеграла

$$F(y_j) \approx \int_a^b f(x, y_j) dx,$$

застосовуючи, наприклад, формулу трапецій або Сімпсона.

3. Далі інтегруємо отриману таблично задану функцію $F(y)$ на відрізку $[c, d]$, знову використовуючи одновимірний чисельний метод:

$$\int_c^d F(y) dy \approx \sum_{j=0}^n w_j F(y_j),$$

де w_j — вагові коефіцієнти вибраної квадратурної формули.

Таким чином, подвійний інтеграл чисельно обчислюється як вкладена комбінація двох одновимірних чисельних інтегралів:

$$\iint_D f(x, y) dx dy \approx \sum_{j=0}^n w_j \left(\sum_{i=0}^m \tilde{w}_i f(x_i, y_j) \right),$$

де $x_i \in [a, b]$, $y_j \in [c, d]$, а $\tilde{w}_i$, w_j — відповідні коефіцієнти обраних квадратур.

Переваги методу:

- використання відпрацьованих методів одномірного інтегрування;
- можливість реалізації на сітці рівномірного кроку;
- хороша адаптивність до програмної реалізації та паралельних обчислень.

Обмеження:

- застосовується переважно до прямокутних або регулярних областей;
- у разі складної геометрії області або особливих меж інтегрування потрібна зміна змінних або розбиття області.
- для n - вимірних інтегралів при $n > 3$ різко зростає необхідна кількість точок, в яких розраховується підінтегральна функція (N^n , де N - кількість розбиттів одновимірного інтегралу)

Висновок. Повторне застосування одновимірних чисельних методів дозволяє ефективно обчислювати багатовимірні інтеграли, особливо подвійні та потрійні, на регулярних областях. Цей підхід лежить в основі багатьох прикладних методів чисельного інтегрування в обчислювальній математиці та фізиці, але має складності при розрахунку багатовимірних інтегралів.

Метод Монте-Карло для багатовимірного інтегрування

У випадках, коли кількість змінних у підінтегральній функції є великою, класичні чисельні методи, які ґрунтуються на сітковому розбитті області інтегрування, стають непридатними через експоненціальне зростання обсягу обчислень. Зокрема, при розбитті кожної змінної на n вузлів, загальна кількість точок для d -вимірного інтегралу становить n^d , що при великих d швидко робить задачу обчислювально нездійсненною.

Альтернативним підходом є **метод Монте-Карло**, який базується на використанні випадкових (псевдовипадкових) точок у області інтегрування. Метод дозволяє ефективно оцінити значення багатовимірного визначеного інтеграла навіть за великої кількості змінних.

Суть методу. Нехай потрібно обчислити інтеграл

$$I = \int_{\Omega} f(\mathbf{x}) d\mathbf{x},$$

де $\Omega \subset \mathbb{R}^n$ — область інтегрування, а $\mathbf{x} = (x_1, x_2, \dots, x_n)$. Метод Монте-Карло оцінює цей інтеграл за формулою:

$$I \approx V(\Omega) \cdot \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i),$$

де:

$V(\Omega)$ — об'єм (міра) області Ω ;

$\mathbf{x}_i$ — випадкові точки, рівномірно розподілені в Ω ;

N — кількість випадкових точок.

Це є оцінкою математичного сподівання $M[f(\mathbf{x})]$ — середнього значення функції, помноженого на об'єм області.

Похибка методу. Незалежно від розмірності області n , похибка оцінки методом Монте-Карло спадає як:

$$\varepsilon \sim \frac{1}{\sqrt{N}}.$$

Таким чином, точність методу повільно зростає з кількістю точок N , однак вона не залежить від розмірності області інтегрування. Це і робить метод Монте-Карло особливо привабливим у високорозмірних задачах, де сіткові методи є непридатними.

Переваги:

- ефективність при великій кількості змінних;
- простота реалізації;
- можливість паралельного виконання.

Недоліки:

- відносно повільна збіжність (похибка зменшується лише як $\frac{1}{\sqrt{N}}$);
- залежність від якості генератора випадкових чисел;
- необхідність знати або оцінити об'єм області $V(\Omega)$.

Висновок. Метод Монте-Карло є універсальним та ефективним інструментом для чисельного обчислення багатовимірних інтегралів. Незважаючи на порівняно повільну збіжність, він залишається практично єдиним методом, здатним забезпечити обчислювальну реалізованість інтегрування у високих розмірностях.

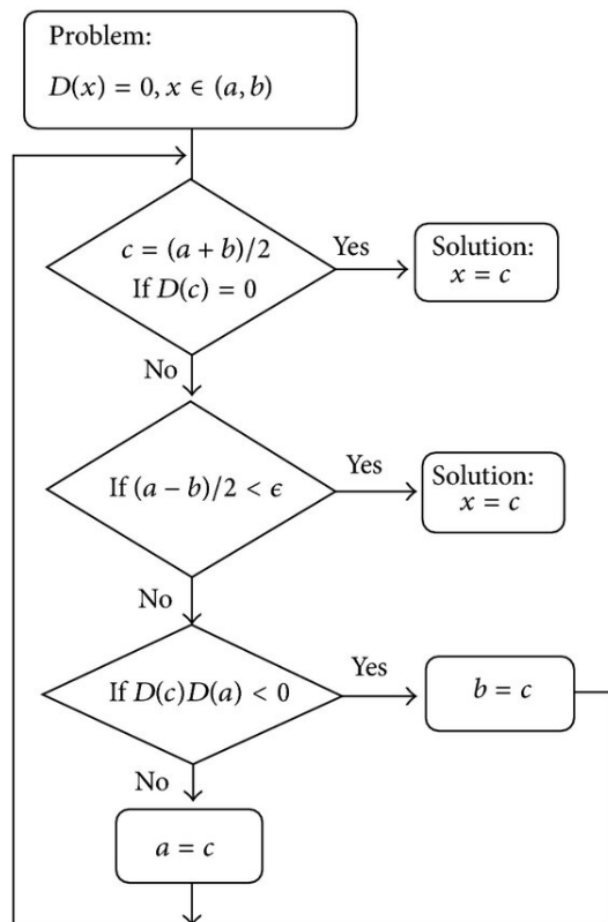


Рис. 4.6. Блок-схема методу половинного ділення.

Розділ 5

Розв'язок диференціальних рівнянь

5.1. Чисельне диференціювання

Цей розділ знайомить із методами чисельного диференціювання для обчислення першої та другої похідної. Розглядаються похибки наведених чисельних методів, а також вплив точності машинної арифметики на результати обчислень.

Обчислення першої похідної

За визначенням, перша похідна гладкої функції $f(x)$ у точці x обчислюється як

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (5.1)$$

При обчисленні похідної функції $f(x)$ на комп'ютері нескінченно мале $h \rightarrow 0$ замінюється на кінцеве значення h :

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + O(h) \quad (5.2)$$

де $O(h)$ — похибка обчислення похідної, яка природно залежить від h . Формула (1.1) називається різницевою схемою для обчислення першої похідної (точніше — правосторонньою різницевою схемою або просто правою різницею). Аналогічно можна записати ліву різницеву схему.

Як оцінити $O(h)$? Розкладемо функцію $f(x)$ у ряд Тейлора в точці $x+h$:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \dots \quad (5.3)$$

Звідси випливає, що в першому порядку розкладу

$$O(h) = \frac{h}{2}f''(x) + \dots \quad (5.4)$$

При виборі дуже малого h похибки округлення при обчисленнях на комп'ютері можуть бути порівнянними або й більшими. Тому нас цікавлять алгоритми, що забезпечують меншу похибку при тій же величині h .

Такий покращений алгоритм легко отримати, розклавши функцію $f(x)$ у ряд Тейлора в точках $x+h$ і $x-h$, а потім віднявши одне рівняння від іншого. Це дає:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \quad (5.5)$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \quad (5.6)$$

де похибка обчислення першої похідної має вигляд:

$$O(h^2) = -\frac{h^2}{6}f^{(3)}(x) + \dots \quad (5.7)$$

Це — **центральна різницева схема** (центральна різниця).

У принципі, можна покращувати точність методу обчислення першої похідної ще далі. Наприклад, розглянувши розклад функції $f(x)$ у ряд Тейлора в точках $x+h$, $x+2h$, $x-h$ і $x-2h$, можна отримати чотириточкову схему тощо.

Обчислення другої похідної

За аналогією з виведенням різницевих схем для першої похідної, можна отримати різницеві схеми для обчислення другої похідної функції $f(x)$. Ми не наводимо тут виведення, а запишемо лише одну з найчастіше використовуваних схем:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2) \quad (5.8)$$

з похибкою:

$$O(h^2) = -\frac{h^2}{12}f^{(4)}(x) + \dots \quad (5.9)$$

5.2. Суть методу скінченних різниць (сіток)

Метод сіток, або метод скінченних різниць, є одним із найпоширеніших методів чисельного розв'язання крайових задач для рівнянь математичної фізики. Ідея методу належить Л. Ейлеру. Суть полягає в наступному: область неперервної зміни аргументів шуканої функції в задачі з крайовими умовами заміняють на дискретну область, що складається з кінцевого числа вузлів. Ці точки називають *вузлами*, а множину вузлів — *різницевою сіткою*.

Пошук невідомої функції для неперервних аргументів зводиться до пошуку *сіткової функції* — значень шуканої функції у вузлах сітки. Для цього відповідні диференціальні рівняння (чи системи) та крайові умови заміняють на *різницеві аналоги*, тобто скінченнорізницеві рівняння. Такі рівняння пов'язують значення функції у вузлах і є дискретними аналогами диференціального рівняння.

Різницеві аналоги отримують шляхом:

- апроксимації похідних або диференціалів за формулами Тейлора;
- заміни інтегралів скінченними сумами;
- заміни функцій многочленами тощо.

Отриману таким чином систему алгебраїчних рівнянь називають *різничевою системою*, а саму систему — **різничевою схемою** або **різничевою апроксимацією**. Конфігурацію вузлів сітки, що використовується для побудови схеми, називають *шаблоном схеми*.

Процес побудови різничевої схеми на основі поставленої диференціальної задачі називають **дискретизацією задачі**. Заміна диференціальної задачі дискретним аналогом найчастіше відбувається одним з таких способів:

1. безпосередня заміна похідних їхніми різницевиими виразами;
2. метод невизначених коефіцієнтів апроксимаційних операторів (метод Рейнбаха), що дає змогу апроксимувати похідні за значеннями функції у групах сусідніх вузлів.

Особливої уваги потребує **апроксимація крайових умов**, особливо у випадку криволінійних меж області, де використання стандартного шаблону не дає змоги обійтися без модифікацій або спеціальних методів апроксимації.

Часто виникає потреба в апроксимації внутрішніх граничних умов (вузлів, найближчих до межі). Зазвичай у таких випадках використовують інтерполяційні або асиміляційні підходи другого порядку. Побудована таким чином різницева схема може бути розв'язана чисельними методами: ітераційними, прямими, проєкційними тощо.

Розділ 6

Алгоритми сортування

Алгоритми сортування є одними з найважливіших базових інструментів у програмуванні та алгоритміці. Вони дозволяють впорядковувати дані за певним критерієм — найчастіше у порядку зростання або спадання. Ефективність сортування безпосередньо впливає на загальну швидкість програм, особливо при роботі з великими обсягами інформації.

Сортування характеризується такими параметрами:

- **Складність за часом** (найгірший, середній, найкращий випадок)
- **Складність за пам'яттю**
- **Стабільність** (чи зберігається порядок однакових елементів)
- **Чи є алгоритм in-place** (без додаткової пам'яті)

Нижче розглянуто три базові алгоритми сортування: сортування вибором, вставками та бульбашкове сортування.

6.1. Сортування вибором (Selection Sort)

Цей алгоритм послідовно обирає найменший елемент з невідсортованої частини масиву та переставляє його на початок. Повторюється для наступної підпослідовності, поки весь масив не буде впорядкований.

Опис алгоритму:

1. Знайти найменший елемент у масиві.
2. Обміняти його з першим елементом.
3. Повторити ті самі дії для підмасиву, що починається з другого елемента.
4. Продовжувати до останнього елемента.

Приклад: Нехай маємо масив: [5, 3, 8, 4, 2]

- Знаходимо мінімум: 2. Обмін з 5 $\rightarrow$ [2, 3, 8, 4, 5]
- Мінімум у підмасиві [3, 8, 4, 5] — 3. Нічого не змінюється.
- Мінімум серед [8, 4, 5] — 4. Обмін з 8 $\rightarrow$ [2, 3, 4, 8, 5]
- Мінімум серед [8, 5] — 5. Обмін з 8 $\rightarrow$ [2, 3, 4, 5, 8]

		індекс елементів масиву						
		0	1	2	3	4		
вхідний масив		5	3	8	4	2	пошук найменшого елемента в комітках від 0 до 4	
		2	3	8	4	5	обмін місцями 2 і 5	
		2	3	8	4	5	пошук найменшого елемента в комітках від 1 до 4	
		2	3	8	4	5	обмін не здійснюється (елемент вже на своєму місці)	
		2	3	4	8	5	пошук найменшого елемента в комітках від 2 до 4 і перестановка	
		2	3	4	5	8	пошук найменшого елемента в комітках від 3 до 4 і перестановка	
вихідний масив		2	3	4	5	8	сортування закінчено (останній елемент вже на своєму місці)	

Рис. 6.1. Графічна ілюстрація сортування вибором

6.2. Сортування вставками (Insertion Sort)

Сортування вставками працює за принципом поступового «вставлення» кожного елемента у вже впорядковану частину масиву.

Опис алгоритму:

1. Починаємо з другого елемента.
2. Порівнюємо його з попередніми.
3. Переміщуємо більші елементи праворуч.
4. Вставляємо поточний елемент у відповідне місце.

Приклад: [5, 3, 8, 4, 2]

- $3 < 5 \rightarrow$ обмін: [3, 5, 8, 4, 2]
- 8 вже стоїть правильно.
- $4 < 8$, вставляємо між 5 і 8 $\rightarrow$ [3, 5, 4, 8, 2] $\rightarrow$ [3, 4, 5, 8, 2]
- $2 <$ усіх попередніх, вставляємо на початок $\rightarrow$ [2, 3, 4, 5, 8]

		індекс елементів масиву						
		0	1	2	3	4		
вхідний масив		5	3	8	4	2	починаємо з другого елемента (комірка 1) і порівнюємо з попереднім	
		3	5	8	4	2	обмін місцями 3 і 5	
		3	5	8	4	2	порівнюємо 3-й елемент з 2-м і нічого не переміщуємо	
		3	5	8	4	2	порівнюємо 4-й елемент (комірка 3) з попередніми	
		3	4	5	8	2	переміщуємо його на своє місце	
		3	4	5	8	2	порівнюємо 5-й елемент (комірка 4) з попередніми і переміщуємо його	
вихідний масив		2	3	4	5	8	сортування закінчено (всі елементи переміщено на свої місця)	

Рис. 6.2. Графічна ілюстрація сортування вставками

6.3. Бульбашкове сортування (Bubble Sort)

Цей алгоритм багаторазово проходить масив, порівнюючи попарно елементи і змінюючи їх місцями, якщо вони стоять у неправильному порядку. Найбільші елементи поступово «вспливають» наприкінці масиву.

Опис алгоритму:

1. Пройти масив зліва направо.
2. Якщо елемент більший за наступний — обміняти.
3. Повторити n-1 разів, зменшуючи довжину розгляду.

Приклад: [5, 3, 8, 4, 2]

- Прохід 1: [3, 5, 4, 2, 8]
- Прохід 2: [3, 4, 2, 5, 8]
- Прохід 3: [3, 2, 4, 5, 8]
- Прохід 4: [2, 3, 4, 5, 8]

	індекс елементів масиву				
	0	1	2	3	4
вхідний масив	5	3	8	4	2
	3	5	8	4	2
	3	5	4	8	2
	3	5	4	2	8
	3	4	5	2	8
	3	4	2	5	8
	3	2	4	5	8
вихідний масив	2	3	4	5	8

порівнюємо послідовно усі сусідні елементи і обмінюємо їх місцями, якщо потрібно
1-й прохід до кінця масиву
крок 1: обмін 5 і 3, **крок 2:** не змінюємо 5 і 8

крок 3: обмін 8 і 4

крок 4: обмін 8 і 2
2-й прохід до передостаннього елемента масиву
крок 1: не змінюємо 3 і 5, **крок 2:** обмін 5 і 4

крок 3: обмін 5 і 2
3-й прохід до n-2 елемента масиву
крок 1: не змінюємо 3 і 4, **крок 2:** обмін 4 і 2
4-й прохід до n-3 елемента масиву
крок 1: обмін 3 і 2. Сортування закінчено

Рис. 6.3. Графічна ілюстрація бульбашкового сортування

Висновок: Ці три методи є чудовою відправною точкою для розуміння принципів сортування. Хоча вони мають обмежену ефективність на великих обсягах даних, вони формують фундаментальні уявлення про алгоритмічне мислення та обробку масивів.

6.4. Швидке сортування (Quick Sort)

Швидке сортування (англ. *Quick Sort*) — це один з найефективніших алгоритмів сортування для загального випадку, що працює за принципом «розділяй і володарюй». Алгоритм рекурсивно ділить масив на підмасиви за опорним елементом (півотом) так, що ліворуч від опорного опиняються всі елементи, менші за нього, а праворуч — усі більші або рівні. Потім ці підмасиви сортуються незалежно тим самим методом.

Характеристики алгоритму:

- Середня складність: $O(n \log n)$
- Найгірший випадок: $O(n^2)$ (наприклад, якщо масив вже впорядкований)
- В середньому — один з найшвидших алгоритмів сортування
- Виконується in-place (не потребує додаткової пам'яті)

Опис алгоритму:

1. Вибрати опорний елемент (наприклад, останній або середній).
2. Розділити масив на дві частини: всі елементи, менші за опорний, — зліва, всі більші або рівні — справа.
3. Рекурсивно застосувати алгоритм до обох частин.
4. Після завершення рекурсії отримаємо впорядкований масив.

Приклад:

Нехай маємо масив: [5, 3, 8, 4, 2]

Рекурсивну функцію для реалізації алгоритму швидкого сортування назовемо QuickSort()

- Обираємо опорним останній елемент: **2**
елементи менші за 2: порожньо,
більші або рівні: [5, 3, 8, 4]
Після розбиття: [2] + QuickSort([5, 3, 8, 4])
- Опорним елементом у [5, 3, 8, 4] буде **4**
елементи менші за 4: [3],
більші: [5, 8]
Результат: QuickSort([3]) + [4] + QuickSort([5, 8])
- QuickSort([3]) → [3]
QuickSort([5, 8]) → опорний **8**, менші: [5], більших немає → [5, 8]
- Остаточна збірка: [2, 3, 4, 5, 8]

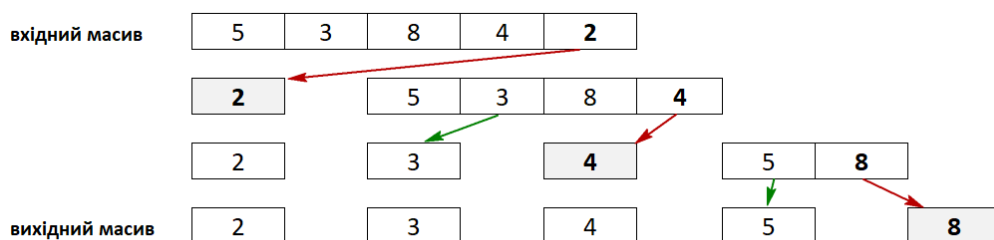


Рис. 6.4. Графічна ілюстрація швидкого сортування (червоні стрілки - переміщення опорних елементів, зелена - переміщення елементів менших за опорний).

Таким чином, швидке сортування ефективно розбиває задачу на менші підзадачі, які швидко розв'язуються рекурсивно. Воно добре працює з великими обсягами даних і є стандартом у багатьох мовах програмування (наприклад, у Python реалізовано як частину Timsort).

6.5. Сортування злиттям (Merge Sort)

Сортування злиттям (англ. *Merge Sort*) — це ефективний, стабільний алгоритм сортування, який реалізує стратегію «розділяй і володарюй». Він розбиває масив на дрібніші підмасиви, сортує кожен з них окремо, а потім об'єднує (зливає) у впорядкований масив.

Характеристики алгоритму:

- Складність у найгіршому, середньому та найкращому випадках: $O(n \log n)$
- Стабільний алгоритм (не змінює порядок однакових елементів)
- Потребує додаткової пам'яті $O(n)$
- Підходить для сортування великих наборів даних

Опис алгоритму:

1. Розділити масив навпіл.
2. Рекурсивно застосувати сортування злиттям до кожної половини.
3. Злити (merge) дві відсортовані половини в один впорядкований масив.

Приклад:

Нехай маємо масив: [5, 3, 8, 4, 2]

- Розділяємо: [5, 3] і [8, 4, 2]
- [5, 3] → розділити на [5] і [3] → злиття: [3, 5]
- [8, 4, 2] → розділити на [8] і [4, 2]
- [4, 2] → розділити на [4] і [2] → злиття: [2, 4]
- Тепер [8] і [2, 4] → злиття: [2, 4, 8]
- Злиття двох частин: [3, 5] і [2, 4, 8] → [2, 3, 4, 5, 8]

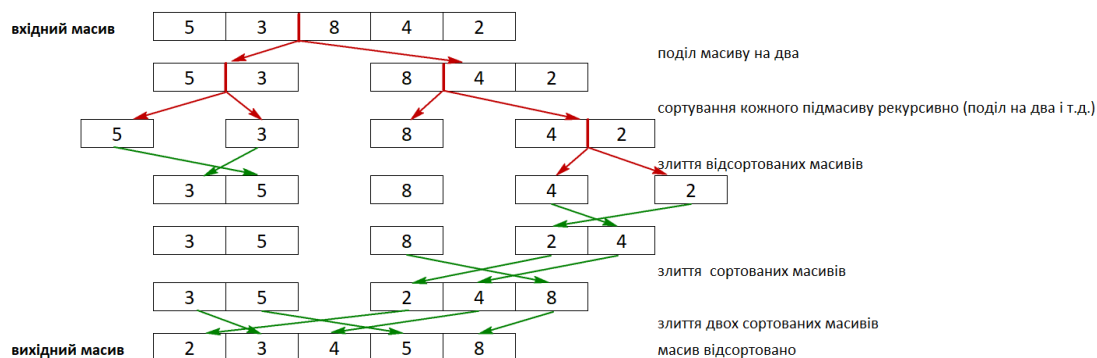


Рис. 6.5. Графічна ілюстрація сортування злиттям (червоні стрілки - поділ масиву на два, зелені стрілки - злиття двох масивів з сортуванням).

Таким чином, сортування злиттям забезпечує стабільне та ефективне сортування незалежно від початкового порядку елементів. Його особливо доцільно використовувати у випадках, коли важлива стабільність або обробляються великі обсяги даних у потоковому режимі.

Частина ІІІ

ДОДАТКИ. ПРИКЛАДИ PYTHON ПРОГРАМ

Розділ 1

Чисельне знаходження кореня рівняння

1.1. Метод Ньютона

Нижче наведено варіант реалізації методу Ньютона. Програма виконує ітеративний пошук кореня функції $f(x) = x^2 - 16$ з абсолютною точністю до $eps = 10^{-6}$. Після досягнення потрібної точності або вичерпання кількості ітерацій $n = 10$, результат виводиться на екран.

```
1 def f(x):
2     return x*x - 16
3
4 def df(f, x, h=1e-8):
5     return (f(x + h) - f(x - h)) / (2*h)
6
7 def find_root_newton(f, a, eps=1e-6, n=10):
8     for i in range(n):
9         x = a - f(a) / df(f, a)
10        if abs(x - a) < eps:
11            break
12        a = x
13    print(f"метод Ньютона: {i} ітерацій, точність {abs(x - a)}")
14    return x
15
16 print(find_root_newton(f, 5), '\n')
```

Результат виконання:

метод Ньютона: 3 ітерацій, точність 1.86e-07
4.0000000000000004

У цьому прикладі корінь рівняння $x^2 - 16 = 0$ успішно знайдено з точністю до 10^{-6} вже після кількох ітерацій, починаючи з початкового наближення $x_0 = 5$. Фактичний корінь — $x = 4$.

1.2. Метод січних (Secant Method)

Метод січних — це чисельний алгоритм для знаходження коренів рівняння $f(x) = 0$, що не потребує обчислення аналітичної похідної. Замість цього використовується апроксимація похідної за допомогою різниці значень функції в двох останніх точках. Алгоритм є ітераційним і збігається швидше, ніж метод простої ітерації, але повільніше, ніж метод Ньютона.

```
1 def f(x):
2     return x*x - 16
```

```

3
4 def find_root_secant(f, a, eps=1e-3, n=10):
5     xn = a
6     xn_1 = a - eps
7     for i in range(n):
8         xnn = xn - f(xn)*(xn - xn_1)/(f(xn) - f(xn_1))
9         if abs(xnn - xn) < eps:
10             print(f"метод січних: {i} ітерацій. точність {abs(xnn - xn):.3e}")
11             return xnn
12         else:
13             xn_1 = xn
14             xn = xnn
15     print(f"{n} ітерацій. точність {xnn - xn}")
16     return xnn
17
18 print(find_root_secant(f, 5))

```

Результат виконання:

метод січних: 3 ітерацій. точність 1.35e-04
4.000000185349389

Як бачимо, метод січних збігається до точного кореня $x = 4$ вже за кілька ітерацій, навіть без обчислення похідної. Це робить його зручним для задач, де похідна складна або відсутня.

1.3. Метод хорд (Regula Falsi)

Метод хорд є чисельним методом для знаходження кореня рівняння $f(x) = 0$ на відрізку $[a, b]$, де функція змінює знак. На відміну від методу січних, метод хорд зберігає гарантії збіжності, бо завжди зберігає інтервал, у якому функція змінює знак.

```

1 def f(x):
2     return x*x - 16
3
4 def find_root_hord(f, a, b, eps=1e-4, n=10):
5     fa = f(a)
6     fb = f(b)
7     if fa * fb > 0:
8         print("нема кореня в інтервалі, або їх парна кількість")
9     for i in range(n):
10         c = a - fa * (b - a) / (fb - fa)
11         fc = f(c)
12         if abs(a - c) < eps or abs(b - c) < eps:
13             break
14         elif fa * fc < 0:
15             b = c
16             fb = fc
17         elif fa * fc > 0:
18             a = c
19             fa = fc
20     print(f"метод хорд: {i} ітерацій. точність {min(c - a, b - c):.2e}")
21     return c
22
23 print(find_root_hord(f, 3, 5))

```

Результат виконання:

метод хорд: 5 ітерацій. точність 7.23e-05
3.9999909679633663

Як бачимо, метод хорд збігається до кореня $x = 4$ за кілька ітерацій, з досягнутою точністю 7.23×10^{-5} . Він зберігає інтервал, де функція змінює знак, що робить його надійним у порівнянні з методом січних.

1.4. Метод половинного ділення (Bisection Method)

Метод половинного ділення — це класичний чисельний метод для знаходження коренів рівняння $f(x) = 0$, який гарантує збіжність за умови, що функція неперервна на відрізку $[a, b]$ і $f(a) \cdot f(b) < 0$, тобто функція змінює знак. Метод ітеративно ділить відрізок навпіл і обирає ту підділянку, де зберігається зміна знаку, поступово звужуючи інтервал.

```
1 def f(x):
2     return x*x - 16
3
4 def find_root_bisec(f, a, b, eps=1e-3, n=10):
5     fa = f(a)
6     fb = f(b)
7     if fa * fb > 0:
8         print("нема кореня в інтервалі, або їх парна кількість")
9     for i in range(n):
10        c = (b + a) / 2
11        fc = f(c)
12        if abs(b - a) < eps:
13            break
14        elif fa * fc <= 0:
15            b = c
16            fb = fc
17        elif fa * fc > 0:
18            a = c
19            fa = fc
20    print(f"метод bisection: {i} ітерацій, точність {(b - a):.2e}")
21    return c
22
23 print(find_root_bisec(f, 1, 5, n=20))
```

Результат виконання:

метод bisection: 12 ітерацій, точність 9.77e-04
3.99951171875

Як видно з результату, метод половинного ділення досягає точності ($< 10^{-3}$) за 12 ітерацій, тобто гіршої точності за більшу кількість ітерацій. Хоча він повільніший порівняно з методами Ньютона або січних, його перевагою є гарантована збіжність при простій реалізації.

Розділ 2

Чисельне інтегрування

Розглянемо програми чисельного інтегрування різними методами на прикладі обчислення визначеного інтегралу:

$$\int_0^{\pi/2} x \sin(x) dx = 1$$

Цей інтеграл має важливе дидактичне значення, оскільки функція $x \sin(x)$ є гладкою, а результат інтегрування — добре відомий. Точне аналітичне значення цього інтегралу легко отримати інтегруванням частинами.

У наступних підрозділах ми реалізуємо різні чисельні методи обчислення цього інтегралу та порівняємо їх точність:

- метод прямокутників,
- метод трапецій,
- метод Сімпсона.

Кожен з методів дозволяє наближено обчислити площу під графіком функції на заданому інтервалі. Вигляд цієї функції наведено на рис. 2.1.

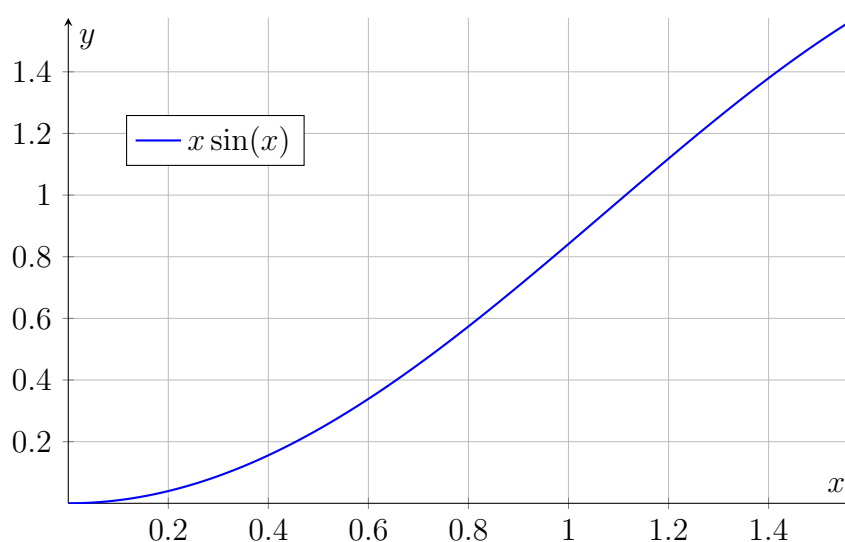


Рис. 2.1. Графік функції $x \sin(x)$ на інтервалі $[0, \pi]$

2.1. Метод прямокутників

Метод прямокутників є одним з найпростіших чисельних методів інтегрування. Його суть полягає у наближенні площі під кривою сумою площ прямокутників. Існує три основні варіанти

цього методу: з лівими, правими та середніми прямокутниками. Точність методу зростає при збільшенні кількості розбиттів інтервалу.

```
1  from math import sin, pi
2
3  def fun(x):
4      return x*sin(x)
5
6  n = 100
7  a = 0
8  b = pi/2
9  h = (b - a)/n
10
11  # Ліві прямокутники
12  x_list = [a + i*h for i in range(n)]
13  s = 0
14  for x in x_list:
15      s += fun(x)
16  integ1 = s*h
17  print(f"I = {integ1} - ліві прямокутники")
18
19  # Праві прямокутники
20  x_list = [a + i*h for i in range(1,n+1)]
21  s = 0
22  for x in x_list:
23      s += fun(x)
24  integ2 = s*h
25  print(f"I = {integ2} - праві прямокутники")
26
27  # Середні прямокутники
28  x_list = [a + h/2 + i*h for i in range(n)]
29  s = 0
30  for x in x_list:
31      s += fun(x)
32  integ3 = s*h
33  print(f"I = {integ3} - середні прямокутники")
```

Результат виконання при $n = 100$:

I = 0.98768356 - ліві прямокутники
I = 1.01235757 - праві прямокутники
I = 0.99998972 - середні прямокутники

Аналіз результату

- Ліві та праві прямокутники дають дещо зміщені значення, відповідно в бік недооцінки та переоцінки.
- Метод середніх прямокутників демонструє значно кращу точність: результат майже збігається з точним значенням інтегралу.
- При $n = 100$ метод середніх прямокутників дає результат з похибкою менше ніж 0.001%.

Отже, навіть простий метод прямокутників може дати хороші результати при правильному виборі варіанту (середні прямокутники) та достатньому числі розбиттів.

2.2. Метод трапецій

Метод трапецій є класичним чисельним методом наближеного обчислення визначених інтегралів. Його суть полягає в апроксимації підінтегральної функції на кожному малому відрізку відрізка інтегрування відрізком прямої, тобто побудові трапецій замість прямокутників.

```
1 from math import sin, pi
2
3 def f(x):
4     return x*sin(x)
5
6 def int_trapezoid(f, a, b, n=10):
7     h = (b - a) / n
8     s = 0.5 * (f(a) + f(b))
9     for i in range(1, n):
10        x = a + i * h
11        s += f(x)
12    return s * h
13
14 n = 10
15 print(f"I = {int_trapezoid(f, 0, pi/2, n):.8f}    n = {n}")
```

Результат виконання:

```
I = 1.00205871    n = 10
I = 1.00002056    n = 100
```

Аналіз результату

Отриманий результат при $n = 10$ дорівнює 1.00205871, що означає невелику переоцінку інтеграла. Відносна похибка становить приблизно 0.21% при $n = 10$ і 0.0021% при $n = 100$, що є прийнятним для багатьох практичних задач. Точність методу швидко покращується при збільшенні кількості розбиттів n , але він поступається методу Сімпсона за швидкістю збіжності.

Метод трапецій є ефективним та інтуїтивно зрозумілим способом обчислення інтегралів. Він особливо корисний, коли потрібно швидко отримати наближене значення з помірною точністю.

2.3. Метод Сімпсона

Метод Сімпсона є одним з найточніших класичних методів чисельного інтегрування. Він базується на апроксимації функції на кожному малому підінтервалі параболою, побудованою через три точки. Алгоритм дає високу точність навіть при невеликій кількості розбиттів.

```
1 from math import sin, pi
2
3 def f(x):
4     return x*sin(x)
5
6 def int_Simp(f, a, b, n=10):
7     h = (b - a) / (2 * n)
8     s = f(a) - f(b) # знак '-' тому, що в останній ітерації буде додано 2*f(b)
9     for i in range(n):
10        x = a + (2 * i + 1) * h
11        s = s + 4 * f(x)
```

```
12         x = x + h
13         s = s + 2 * f(x)
14     s = s * h / 3
15     return s
16
17 n = 10
18 print(f"I = {int_Simp(f, 0, pi/2, n):.8f}    n = {n}  ")
```

Результат виконання:

I = 0.99999937 n = 10

Аналіз результату

Отриманий результат при $n = 10$ дорівнює 0.99999937, що свідчить про надзвичайно малу похибку — менше ніж 0.0001%. Це демонструє високу ефективність методу Сімпсона для гладких функцій навіть при помірному числі розбиттів.

Розділ 3

Розв'язок диференціальних рівнянь

3.1. Розв'язання диференціального рівняння ітераційним методом

Розглянемо задачу Коші для диференціального рівняння першого порядку:

$$y' - y = 0, \quad y(0) = y_0$$

Це рівняння має точний аналітичний розв'язок $y(x) = y_0 e^x$. Ми використаємо ітераційний метод скінченних різниць для апроксимації розв'язку на відрізку $[0, 0.5]$ з кроком $h = 0.01$.

```
1 from math import exp
2
3 # Параметри задачі
4 xmin = 0
5 xmax = 0.5
6 y0 = 1
7 h = 0.01
8
9 n = int((xmax - xmin)/h) + 1
10 x = [h*i for i in range(n)]
11 y = [0 for i in range(n)]
12 y[0] = y0
13
14 # Ітераційний метод скінченних різниць
15 for i in range(n-1):
16     y[i+1] = y[i] * (1 + h) # формула на основі  $y' = (y[i+1] - y[i])/h$ 
17
18 # Порівняння з точним розв'язком
19 for i in range(n):
20     print(f'x = {x[i]:.2f} y = {y[i]:.4f} точне значення {exp(x[i]):.4f}')
```

Результат виконання (фрагмент):

```
x = 0.00 y = 1.0000 точне значення 1.0000
x = 0.01 y = 1.0100 точне значення 1.0101
x = 0.02 y = 1.0201 точне значення 1.0202
x = 0.03 y = 1.0303 точне значення 1.0305
x = 0.04 y = 1.0406 точне значення 1.0408
x = 0.05 y = 1.0510 точне значення 1.0513
x = 0.06 y = 1.0615 точне значення 1.0618
x = 0.07 y = 1.0721 точне значення 1.0725
x = 0.08 y = 1.0829 точне значення 1.0833
```

$x = 0.09$ $y = 1.0937$ точне значення 1.0942
 $x = 0.10$ $y = 1.1046$ точне значення 1.1052
 ...
 $x = 0.47$ $y = 1.5963$ точне значення 1.6000
 $x = 0.48$ $y = 1.6122$ точне значення 1.6161
 $x = 0.49$ $y = 1.6283$ точне значення 1.6323
 $x = 0.50$ $y = 1.6446$ точне значення 1.6487

Аналіз результату

- Ітераційний метод базується на заміні похідної скінченою різницею:

$$y'(x) \approx \frac{y_{i+1} - y_i}{h} \Rightarrow y_{i+1} = y_i(1 + h)$$

- При $h = 0.01$ розв'язок апроксимується з високою точністю: на всьому проміжку похибка не перевищує 0.004. Для збільшення точності необхідно зменшити крок.
- Метод легко реалізується й дозволяє наближено обчислювати розв'язки, коли аналітичний вираз відомий або складний.

Таким чином, ітераційний метод скінченних різниць демонструє високу ефективність для лінійних рівнянь з відомими розв'язками, і є основою для більш складних чисельних схем.

3.2. Скінченно-різницевий розв'язок рівняння з неоднорідними граничними умовами

Розглянемо задачу для диференціального рівняння другого порядку з неоднорідними граничними умовами:

$$y''(x) + y'(x) = 0, \quad x \in [0, 5], \quad y(0) = 1, \quad y(5) = 1.9933$$

Це рівняння має точний розв'язок:

$$y(x) = 2 - e^{-x}$$

Для чисельного розв'язання використаємо скінченно-різницеву апроксимацію похідних та метод Крамера для розв'язання системи лінійних рівнянь.

```

1  from math import exp
2
3  # Параметри задачі
4  xmin = 0
5  xmax = 5
6  y0 = 1
7  yN = 1.9933
8  h = 0.1
9
10 n = int((xmax - xmin)/h)
11 x = [h*i for i in range(n + 1)]
12
13 # Побудова матриці A та вектора B для системи
14 A = [[0 if i-j > 1 else h - 1 if i-j == 1 else 2 - h if i == j else -1 if j - i == 1 else 0
15       for j in range(n-1)] for i in range(n-1)]
16
17 B = [(1 - h)*y0 if i == 0 else yN if i == n - 2 else 0 for i in range(n - 1)]

```

```

18
19 # Функція Крамера (має бути визначена окремо)
20 y = [y0] + kramer(A, B) + [yN]
21
22 # Порівняння з точним розв'язком
23 for i in range(n + 1):
24     print(f'x = {x[i]:.2f} y = {y[i]:.4f} точне значення {2 - exp(-x[i]):.4f}')
```

Результат виконання (фрагмент):

```

x = 0.00 y = 1.0000 точне значення 1.0000
x = 0.10 y = 1.0998 точне значення 1.0952
x = 0.20 y = 1.1897 точне значення 1.1813
x = 0.30 y = 1.2706 точне значення 1.2592
x = 0.40 y = 1.3434 точне значення 1.3297
x = 0.50 y = 1.4089 точне значення 1.3935
x = 0.60 y = 1.4678 точне значення 1.4512
x = 0.70 y = 1.5209 точне значення 1.5034
x = 0.80 y = 1.5686 точне значення 1.5507
x = 0.90 y = 1.6116 точне значення 1.5934
x = 1.00 y = 1.6503 точне значення 1.6321
...
x = 4.10 y = 1.9852 точне значення 1.9834
x = 4.20 y = 1.9865 точне значення 1.9850
x = 4.30 y = 1.9877 точне значення 1.9864
x = 4.40 y = 1.9888 точне значення 1.9877
x = 4.50 y = 1.9897 точне значення 1.9889
x = 4.60 y = 1.9906 точне значення 1.9899
x = 4.70 y = 1.9914 точне значення 1.9909
x = 4.80 y = 1.9921 точне значення 1.9918
x = 4.90 y = 1.9927 точне значення 1.9926
x = 5.00 y = 1.9933 точне значення 1.9933
```

Аналіз результату

- Використано рівномірне розбиття інтервалу довжиною $h = 0,1$.
- Для апроксимації похідних використано центральні-різницькі формули другого порядку точності.
- Сформована система лінійних рівнянь розв'язується методом Крамера.
- Отриманий результат з високою точністю збігається з аналітичним розв'язком.
- При зменшенні кроку h , точність методу зростає, але різко зростає час розрахунку розв'язків системи лінійних рівнянь методом Крамера через ріст розмірності матриць (рекомендується використовувати метод Гауса).

Цей приклад демонструє можливість точного чисельного розв'язання рівнянь другого порядку навіть при складних граничних умовах — коли значення розв'язку задане на протилежних кінцях інтервалу.

Розділ 4

Сортування масивів

Сортування є фундаментальною задачею алгоритмічної обробки даних. У цьому розділі представлено реалізації п'яти класичних методів сортування, а також вбудовану функцію сортування Python. Для кожного методу проведено оцінку швидкодії.

4.1. Програми сортування різними алгоритмами та порівняння швидкості їх роботи

1. **Метод бульбашки** — послідовне порівняння сусідніх елементів з обміном, якщо вони стоять у неправильному порядку.
2. **Метод вставки** — кожен елемент вставляється у вже відсортовану частину масиву.
3. **Метод вибору** — для кожної позиції вибирається найменший елемент з решти масиву.
4. **Метод злиття (рекурсивний)** — масив ділиться на дві частини, кожна з яких сортується, після чого відбувається злиття.
5. **Швидке сортування (quick sort)** — вибирається опорний елемент, масив розбивається на три частини і рекурсивно сортується.
6. **Функція sorted() Python** — реалізована на базі Timsort, поєднує ідеї insertion sort та merge sort.

Виконаємо порівняння різних методів сортування масиву з 100 чисел, який згенерований генератором випадкових чисел. Для підрахунку часу сортування використаємо функцію *perf_counter* з модуля **time**

```
1 from random import randint
2 from time import perf_counter
3
4 # Генерація масиву випадкових чисел
5 def rand_list(n, a_min=0, a_max=9):
6     return [randint(a_min, a_max) for _ in range(n)]
7
8 # Класичне бульбашкове сортування
9 def sort_bubble_classic(ar):
10     n = len(ar)
11     for i in range(n):
12         swapped = False
13         for j in range(n - i - 1):
14             if ar[j] > ar[j + 1]:
15                 ar[j], ar[j + 1] = ar[j + 1], ar[j]
```

```
16         swapped = True
17     if not swapped:
18         break
19     return ar
20
21 # Сорткування вставками
22 def sort_insert(ar):
23     for i in range(1, len(ar)):
24         im = i
25         while im > 0 and ar[im] < ar[im - 1]:
26             ar[im], ar[im - 1] = ar[im - 1], ar[im]
27             im -= 1
28     return ar
29
30 # Сорткування вибором
31 def sort_select(ar):
32     n = len(ar)
33     for i in range(n - 1):
34         for j in range(i + 1, n):
35             if ar[i] > ar[j]:
36                 ar[i], ar[j] = ar[j], ar[i]
37     return ar
38
39 # Сорткування злиттям (рекурсивне)
40 def sort_bisect(ar):
41     if len(ar) < 2:
42         return ar
43     n1 = len(ar) // 2
44     ar1 = sort_bisect(ar[:n1])
45     ar2 = sort_bisect(ar[n1:])
46     sar = []
47     im1 = im2 = 0
48     while im1 < len(ar1) and im2 < len(ar2):
49         if ar1[im1] < ar2[im2]:
50             sar.append(ar1[im1])
51             im1 += 1
52         else:
53             sar.append(ar2[im2])
54             im2 += 1
55     sar += ar1[im1:] + ar2[im2:]
56     return sar
57
58 # Швидке сорткування (quick sort)
59 def quick_sort(arr):
60     if len(arr) <= 1:
61         return arr
62     pivot = arr[len(arr) // 2]
63     left = [x for x in arr if x < pivot]
64     middle = [x for x in arr if x == pivot]
65     right = [x for x in arr if x > pivot]
66     return quick_sort(left) + middle + quick_sort(right)
67
68 # Тестування швидкодії методів
69 ar0 = rand_list(100, 1, 999)
70 print(ar0[:5])
71 for name, func in [
72     ("класичною бульбашкою", sort_bubble_classic),
```

```
73     ("вставкою", sort_insert),
74     ("вибором", sort_select),
75     ("злиттям", sort_bisect),
76     ("quick_sort", quick_sort),
77     ("sorted", sorted),
78 ]:
79     ar = ar0[:]
80     t1 = perf_counter()
81     sort_ar = func(ar)
82     t2 = perf_counter()
83     print(sort_ar[:5], f'сортування {name} {(t2 - t1):.6f}')
```

Результати виконання (фрагмент):

```
[568, 925, 365, 932, 91]
[9, 12, 19, 21, 30] сортування класичною бульбашкою 0.000391
[9, 12, 19, 21, 30] сортування вставкою 0.000294
[9, 12, 19, 21, 30] сортування вибором 0.000211
[9, 12, 19, 21, 30] сортування злиттям 0.000136
[9, 12, 19, 21, 30] сортування quick_sort 0.000121
[9, 12, 19, 21, 30] сортування sorted 0.000009
```

Аналіз результатів

- Всі методи дали однаковий результат — правильне зростання елементів (на екран виведено лише перші 5 елементів зі 100).
- Найшвидшим виявився вбудований метод `sorted()`, що реалізує оптимізований алгоритм Timsort.
- Найповільнішим був класичний метод бульбашки, що очікувано через складність $O(n^2)$.
- Алгоритми злиття та швидкого сортування є ефективнішими ($O(n \log n)$) і рекомендовані для великих масивів.

Таким чином, вибір методу сортування залежить від задачі, розміру масиву та обмежень на ресурси. Для практичного використання доцільно надавати перевагу швидким алгоритмам або вбудованим реалізаціям.

Розділ 5

Робота з матрицями

Масиви двох вимірів — матриці — є базовими об'єктами у чисельних обчисленнях, лінійній алгебрі та візуалізації даних. У цьому розділі розглянемо основні операції над матрицями, включаючи створення, транспонування, добуток, обчислення визначника та розв'язання систем лінійних рівнянь.

5.1. Створення матриць

Найпростіший спосіб створення матриць це використання оператора присвоєння та вкладених списків. Наприклад, матриця 3×3 :

```
1
2 # створення матриці присвоєнням
3 M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
4 print(M)
```

Результат виконання:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Для зручності сприйняття матриці віддруковують у прямокутній формі:

```
1
2 # виведення матриці у зручному вигляді
3 def print_matrix(M):
4     for row in M:
5         print(*row)
6
7 print_matrix(M)
```

Результат виконання:

```
1 2 3
4 5 6
7 8 9
```

Генерація одиничної матриці

```
1 # створення одиничної матриці
2 def ident_matrix(n):
```

```
3     A = [[1 if i==j else 0 for i in range(n)] for j in range(n)]
4     return A
5
6 print_matrix(ident_matrix(4))
```

Результат виконання:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Генерація матриці з випадкових чисел

```
1 from random import randint
2
3 # створення матриці з випадкових чисел
4 def rand_matrix(n, m, min=0, max=9):
5     A = [[randint(min, max) for i in range(m)] for j in range(n)]
6     return A
7 print_matrix(rand_matrix(3, 4))
```

Результат виконання:

```
2 5 2 1
0 6 1 8
0 8 9 2
```

5.2. Транспонування матриці

```
1 def transpose(M):
2     return [[M[i][j] for i in range(len(M))] for j in range(len(M[0]))]
```

5.3. Додавання та множення матриць

Додавання матриць однакових розмірів:

```
1 def add_matrices(A, B):
2     n, m = len(A), len(A[0])
3     return [[A[i][j] + B[i][j] for j in range(m)] for i in range(n)]
```

Множення матриць $A(n,m) * B(m,p)$:

```
1
2 def multiply_matrices(A, B):
3     # Перевірка розмірів: кількість стовпців A має дорівнювати кількості рядків B
4     if len(A[0]) != len(B):
5         raise ValueError("Кількість стовпців у матриці A має дорівнювати кількості рядків B")
6     # Створення матриці з нулями
```

```

7     M = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]
8     # Множення
9     for i in range(len(A)): # рядки A
10        for j in range(len(B[0])): # стовпці B
11            for k in range(len(B)): # стовпці A / рядки B
12                M[i][j] += A[i][k] * B[k][j]
13    return M
14
15    # простіший варіант програми множення матриць
16    def multiply_matrices2(A, B):
17        n, m, p = len(A), len(A[0]), len(B[0])
18        if n == len(B[0]) and m == len(B):
19            return [[sum(A[i][k] * B[k][j] for k in range(m)) for j in range(p)] for i in range(n)]
20        else:
21            return 0
22
23    A = rand_matrix(3,2,0,5)
24    print("\nmatrix A")
25    print_matrix(A)
26
27    B = rand_matrix(2,3,0,5)
28    print("\nmatrix B")
29    print_matrix(B)
30    print("\nmatrix A*B")
31    print_matrix(multiply_matrices(A, B))

```

Результат виконання:

```

matrix A
5 3
1 4
5 0

matrix B
5 5 5
2 1 0

matrix A*B
31 28 25
13 9 5
25 25 25

```

5.4. Обчислення визначника матриці рекурсивним методом

Для квадратної матриці $n \times n$ можна обчислити визначник рекурсивно, розкладаючи його за елементами першого рядка. Це називається "розкладом за Лапласом". Для цього використовуються мінори — визначники менших матриць, утворених шляхом викреслення рядка і стовпця.

Програма обчислення визначника `determ(matr)`

```

1    def determ(matr):
2        n = len(matr)

```

```
3     m = len(matr[0])
4     if n != m:
5         print('Визначник розрахувати неможливо, бо матриця не квадратна')
6         return 0
7     elif n == 1:
8         return matr[0][0]
9
10    det = 0
11    for j in range(n):
12        det += (-1)**(j+1+1) * matr[0][j] * determ(minor(matr, 0, j))
13    return det
14
15    def minor(matr, di, dj):
16        n = len(matr)
17        minor = []
18        for i in range(n):
19            if i == di:
20                continue
21            row = []
22            for j in range(n):
23                if j == dj:
24                    continue
25                row.append(matr[i][j])
26            minor.append(row)
27    return minor
```

Інший спосіб побудови мінору

```
1    def minor2(matr, di, dj):
2        minor = [[a for col_idx, a in enumerate(row) if col_idx != dj]
3                  for row_idx, row in enumerate(matr) if row_idx != di]
4    return minor
```

Зчитування матриці з файлу

```
1    f_in = "matrix.txt"
2
3    with open(f_in, "r") as file:
4        matr = [[int(i) for i in line.split()] for line in file]
5
6    print(f'Файл {f_in} зчитано!\n')
7
8    for row in matr:
9        print(*row)
10
11    print(f'det = {determ(matr)}')
12
13    # інший спосіб розрахунку визначника (використання модуля NumPy)
14    import numpy as np
15    A_np = np.array(matr)    # конвертація у NumPy-масив
16
17    det = np.linalg.det(A_np)
```

```
18 print(f"det A = {det:.2f}")
```

Приклад вмісту файлу matrix.txt:

```
3 8 6 2
1 2 4 1
5 4 8 1
4 2 6 2
```

Результат виконання програми:

Файл matrix.txt зчитано!

```
3 8 6 2
1 2 4 1
5 4 8 1
4 2 6 2
det = 92
det A = 92.00
```

Аналіз результату

- Реалізація методу Лапласа є наочним, але не найефективнішим способом для великих розмірів $n > 5$, оскільки його складність — факторіальна: $O(n!)$.
- Цей метод добре підходить для навчання, оскільки ілюструє суть поняття мінору та рекурсії.
- Для великих матриць доцільно використовувати бібліотеки, що реалізують методи Гаусса або функції з пакету NumPy.

5.5. Обчислення визначника методом Гауса

Метод Гауса полягає у приведенні матриці до верхньої трикутної форми шляхом елементарних перетворень над рядками. Визначник квадратної матриці дорівнює добутку елементів головної діагоналі після приведення, з урахуванням знаку (який змінюється при перестановці рядків).

Програма gaus_det(matrix)

```
1 def gaus_det(matrix, debug=False):
2     """
3     Функція для обчислення визначника методом Гауса.
4     Параметри
5     matrix: Вхідна матриця (список списків)
6     debug: Якщо True, виводить проміжні етапи розрахунку
7     Повертає: Визначник матриці
8     """
9     n = len(matrix)
10    det = 1
11    mat = [row[:] for row in matrix]
12
13    if debug:
14        print("Початкова матриця:")
```

```

15     for row in mat:
16         print(row)
17
18     for i in range(n):
19         if mat[i][i] == 0:
20             for j in range(i + 1, n):
21                 if mat[j][i] != 0:
22                     mat[i], mat[j] = mat[j], mat[i]
23                     det = -det
24                     if debug:
25                         print(f"Після обміну рядків {i+1} і {j+1}:")
26                         for row in mat:
27                             print(row)
28                     break
29             if mat[i][i] == 0:
30                 return 0
31
32     for j in range(i + 1, n):
33         coef = mat[j][i] / mat[i][i]
34         for k in range(n):
35             mat[j][k] -= coef * mat[i][k]
36         if debug:
37             print(f"Після обнулення елемента [{j+1},{i+1}]:")
38             for row in mat:
39                 print(row)
40
41     for i in range(n):
42         det *= mat[i][i]
43
44     return det

```

Тестовий приклад

```

1  A = [
2      [2, 1, 3],
3      [1, 2, 1],
4      [3, 1, 2]
5  ]
6
7  determinant = gaus_det(A, debug=False)
8  print("Задано матрицю A:")
9  for row in A:
10     print(row)
11  print(f"Визначник матриці: {determinant}")

```

Результат виконання

Задано матрицю A:

[2, 1, 3]

[1, 2, 1]

[3, 1, 2]

Визначник матриці: -8.0

Аналіз результату

- Метод Гауса дозволяє ефективно обчислювати визначник великих матриць зі складністю $O(n^3)$.
- У випадку перестановок рядків знак визначника змінюється.
- Якщо під час приведення на головній діагоналі з'являється 0, матриця вироджена і $\det = 0$.
- У наведеному прикладі визначник $\det(A) = -8$, що збігається з результатами інших методів.

5.6. Обернена матриця

Для квадратної матриці A існує обернена матриця A^{-1} , якщо виконується умова:

$$A \cdot A^{-1} = A^{-1} \cdot A = I,$$

де I — одинична матриця. Обернена матриця існує лише тоді, коли $\det(A) \neq 0$.

Алгоритм побудови оберненої матриці

Обернену матрицю можна знайти за формулою:

$$A^{-1} = \frac{1}{\det(A)} \cdot \text{adj}(A),$$

де $\text{adj}(A)$ — союзна матриця (транспонована матриця алгебраїчних доповнень).

Програма побудови оберненої матриці

```

1  from mylib import determ, minor, transp_m, multiply_matrices
2
3  def mInvert(matr):
4      k = 1 / determ(matr)
5      sm = [[(-1)**(di+dj) * k * determ(minor(matr, dj, di))
6              for dj in range(len(matr[0]))]
7              for di in range(len(matr))]
8      return sm
9
10 matr = [[2, 0, 1],
11          [0, 1, 2],
12          [0, 2, 3]]
13
14 mInver = mInvert(matr)
15
16 print('Матриця matr:')
17 print(matr)
18
19 print('Обернена матриця mInver:')
20 print(mInver)
21
22 print('Добуток matr * mInver:')
23 print(multiply_matrices(matr, mInver))

```

Результат виконання

Матриця `matr`:

```
[[2, 0, 1],  
 [0, 1, 2],  
 [0, 2, 3]]
```

Обернена матриця `mInver`:

```
[[0.5, 0.0, -0.5],  
 [0.0, -3.0, 2.0],  
 [0.0, 1.0, -1.0]]
```

Добуток `matr * mInver`:

```
[[1.0, 0.0, 0.0],  
 [0.0, 1.0, 0.0],  
 [0.0, 0.0, 1.0]]
```

Аналіз результату

- Програма успішно обчислює обернену матрицю за допомогою методу алгебраїчних доповнень.
- Добуток вихідної матриці на обернену дає одиничну матрицю, що підтверджує правильність розрахунку.
- Цей метод є наочним, однак має високу обчислювальну складність ($O(n^4)$), тому для великих розмірів доцільно використовувати бібліотеки, наприклад `numpy.linalg.inv()`.

5.7. Обернена матриця (модуль NumPy)

Обернена матриця A^{-1} задовольняє умову $AA^{-1} = I$. Її можна обчислити через метод Гауса-Жордана або використати бібліотеку NumPy.

```
1 import numpy as np  
2  
3 A = np.array([[1, 2], [3, 4]])  
4 A_inv = np.linalg.inv(A)  
5 print(A_inv)
```

5.8. Розв'язання систем лінійних рівнянь

Система $AX = B$ розв'язується як $X = A^{-1}B$:

```
1 A = np.array([[2, 1], [5, 7]])  
2 B = np.array([11, 13])  
3 X = np.linalg.solve(A, B)  
4 print("Розв'язок:", X)
```

5.9. Метод Крамера

Інший спосіб розв'язку систем лінійних рівнянь - метод Крамера:

```
1 from numpy.linalg import det
2
3 def kramer(A, B):
4     n = len(B)
5     D = det(A)
6     X = []
7     for i in range(n):
8         Ai = [row[:] for row in A]
9         for j in range(n):
10             Ai[j][i] = B[j]
11         X.append(det(Ai) / D)
12     return X
```

Частина IV

ОСНОВИ Wolfram Mathematica

Розділ 1

Синтаксис, базові конструкції, елементи програмування

1.1. Загальні відомості. Ядро та інтерфейс пакета *Mathematica*

Пакет прикладних програм *Mathematica* розроблений компанією *Wolfram Research*. *Wolfram Mathematica* — це сучасне інтегроване середовище для символічних, чисельних і графічних обчислень. Програма поєднує мову програмування *Wolfram Language*, систему комп'ютерної алгебри, чисельні алгоритми, побудову графіків, обробку даних, аналітику, доступ до баз знань і багато іншого.

Пакет *Mathematica* дозволяє:

- виконувати чисельні та аналітичні обчислення;
- об'єднувати послідовності обчислень у програми;
- створювати інтерактивні документи, що містять активні формули, текст, графіки та звук;
- підключати зовнішні програми;
- використовувати систему як вбудовану, викликану з інших програм.

Mathematica складається з двох частин: ядра (*Kernel*) та інтерфейсу (*Front End*). Ядро є незмінною частиною, тоді як інтерфейс налаштовується під конкретний комп'ютер і графічну оболонку. Ядро містить основний набір математичних функцій. Запуск ядра відбувається під час виконання першої математичної операції. Після цього ядро починає взаємодіяти з інтерфейсом: приймає вхідні дані, обробляє їх, виконує обчислення і надсилає результати назад до інтерфейсу для відображення. Такий розподіл дозволяє розділити обчислювальний процес і візуалізацію, а також забезпечити гнучкість — наприклад, можна запускати ядро на окремому сервері, а працювати з ним через локальний інтерфейс. Крім роботи з *Wolfram Mathematica*, встановлений на локальному комп'ютері можлива робота через браузер в хмарному середовищі як за допомогою ПК так і смартфона. Для цього потрібно зареєструватись на сайті <https://www.wolframcloud.com/>. Безкоштовно можна писати програми, запускати їх на виконання. Але безкоштовний доступ до ваших програм буде не більше 2 місяців. Також буде встановлено обмеження на час виконання програм.

Для запуску ядра введемо, наприклад, команду:

$2 * 2$

та натиснемо **Shift+Enter**. Після короткого завантаження отримаємо:

In[1]:= $2 * 2$

Out[1]= 4

Після цього система готова до роботи.

1.2. Точні та наближені обчислення. Чисельні та аналітичні операції

Розглянемо приклади обчислень. Обчислимо число $100!$:

```
In[2] := 100!
```

Результат займає кілька рядків, але він абсолютно точний

```
Out[2]= 93326215443944152681699238856266700490715968264381621468592963  
89521759999322991560894146397615651828625369792082722375825118  
52109168640000000000000000000000000000
```

Щоб отримати наближене значення, додамо десяткову крапку:

```
In[3] := 100.!
```

```
Out[3]= 9.33262*10^157
```

Якщо явно не вказано, що потрібне наближене значення, *Mathematica* намагається знайти точний результат.

Наприклад:

```
In[4] := Sqrt[2]
```

$$\text{Out}[4] = \sqrt{2}$$

Щоб отримати чисельне значення, використаємо: `Sqrt[2]/N` або `N[Sqrt[2]]`

За замовчуванням точність приблизно 10^{-15} . Для більшої точності вкажемо:

```
In[5] := N[Sqrt[2], 20]
```

```
Out[5]= 1.4142135623730950488
```

У наведених прикладах кожна команда вводиться в окремій вхідній комірці, яка автоматично нумерується, як і відповідна вихідна комірка. Зліва відображаються позначення $\text{In}[\mathbf{n}] :=$ та $\text{Out}[\mathbf{n}] =$, де $\mathbf{n}$ — порядковий номер комірки.

Якщо у вхідній комірці ввести декілька команд, то всі вони будуть виконані послідовно, а для кожної з них (якщо вона повертає результат) буде створено окрему вихідну комірку.

Якщо деякі проміжні результати не потрібно виводити на екран, наприкінці відповідного рядка ставиться крапка з комою ;. Такий рядок буде виконано, але результат не виведеться. Це дозволяє будувати складні програми з виведенням лише потрібних результатів. Для запуску вмісту комірки достатньо розмістити курсор у будь-якому місці цієї комірки та натиснути комбінацію клавіш **Shift+Enter**, або **Enter** на цифровій панелі.

Приклад. Декілька команд у одній вхідній комірці з приховуванням проміжних результатів:

```
In[1]:= a = 5;  
        b = a^2;  
        b + 1
```

```
Out[1]= 26
```

У цьому прикладі результати обчислень змінних **a** та **b** не виводяться, оскільки команди завершуються символом **;**.

Mathematica виконує також аналітичні перетворення:

```
In[1] := D[Sin[x], x]
```

Out[1]= Cos[x]

```
In[2]:= Integrate[x^2, x]
```

```
Out[2]= x^3/3
```

```
In[3]:= Integrate[x^2, {x, 0, 1}]
```

```
Out[3]= 1/3
```

```
In[4]:= Simplify[(x + y)^4 - (x - y)^4]
```

```
Out[4]= 8 x y (x^2 + y^2)
```

Обчислимо визначений інтеграл:

```
In[5]:= Integrate[Sin[x]*Log[Sin[x]], {x, 0, Pi/2}]
```

```
Out[5]= -1 + Log[2]
```

Для визначених інтегралів, які не мають аналітичного розв'язку можна знайти чисельне значення:

```
In[6]:= NIntegrate[Log[x]/Sin[x], {x, 1, 3}]
```

```
Out[6]= 2.52998
```

Базовий синтаксис

- Виклики функцій мають вигляд **Function[a1, a2, ...]** (у квадратних дужках перелік аргументів)
- Імена стандартних функцій пишуться з великої літери:
`Sin[x], Plot[Sin[x], {x, 0, Pi}]`
- Імена функцій, створених користувачем, бажано з малої літери:
`y[x_] = x^2`
- Круглі дужки використовують лише для групування виразів:
`a * (x + y)`
- Фігурні дужки використовують для списків елементів та масивів:
`s = {1, 3, 5, 7}` (одновимірний список);
`matr = {{1, 2}, {3, 4}}` (матриця 2x2)
- Подвійні квадратні дужки використовують для виклику елементів списків:
`s[[3]]` (3-й елемент списку s - 5);
`matr[[2,1]]` (перший елемент другого рядка - 3)

Арифметичні операції

- Додавання: `3 + 5`
- Множення: `2 x` або `2*x` (пробіл може використовуватись як знак множення)
- Степінь: `x^2`
- Ділення: `a/b`
- Краще залишати вирази в символьному вигляді: `Sin[Pi/4]`

Коментарі

(* це коментар і він може займати довільну кількість рядків *)

Присвоєння значень змінним

$x = 3$ — присвоєння значення.

$y := d$ — відкладене присвоєння (оцінка відбувається при використанні, а значення змінної d на той момент може бути іншим ніж під час присвоєння).

1.3. Робота з текстовими рядками

У середовищі *Mathematica* текстові рядки (стрічки символів) задаються за допомогою лапок, наприклад:

"Це текстовий рядок"

Для роботи з текстом використовуються різноманітні функції обробки рядків:

- `StringLength["Mathematica"]` — повертає довжину рядка (11).
- `StringJoin["Hello , "World"]` — об'єднує рядки в один: "Hello World".
- `StringTake["Example 3"]` — повертає перші 3 символи: "Exa".
- `StringDrop["Example -2"]` — видаляє останні два символи: "Examp".
- `StringSplit["a,b,c "]` — розбиває рядок за роздільником: {"a" "b" "c"}.
- `StringReplace["абаб "a> "o"]` — замінює всі входження: "обоб".

Текстові рядки можна легко форматовувати за допомогою функцій:

- `ToUpperCase["text"]` — перетворює у верхній регістр: "ТЕХТ".
- `ToLowerCase["ТЕХТ"]` — перетворює у нижній регістр: "text".
- `StringReverse["abc"]` — повертає рядок у зворотному порядку: "cba".

Важливим є також перетворення між текстом і числовими значеннями:

```
ToString[123.45]      (* "123.45" *)
ToExpression["2_+_2"] (* 4 *)
```

Такі інструменти дозволяють ефективно обробляти, перетворювати та аналізувати текстову інформацію в рамках програмної логіки *Mathematica*.

1.4. Масиви та операції з ними

У Wolfram Language масиви (вектори, матриці, багатовимірні масиви) представлені у вигляді списків. Списки можуть бути вкладеними — це дозволяє зручно працювати з матрицями та тензорами довільної розмірності.

Створення та візуалізація масивів

Для створення одномірного масиву (вектора) використовується звичайний список:

$v = \{1, 2, 3, 4\}$

Двовимірний масив (матриця) задається як список списків:

$m = \{\{1, 2\}, \{3, 4\}\}$

Багатовимірний масив можна створити, використовуючи функцію `Array`:

```
Array[f, {2, 3}] (* Створює масив з f[i, j] *)
```

Візуалізація

Матрицю можна відобразити як таблицю:

```
MatrixForm[m] або m//MatrixForm
```

Ця форма є лише візуальною — вона не змінює саму матрицю.

Основні операції з масивами

- Доступ до елементів:

```
v[[2]] (* другий елемент вектора *)
m[[2, 1]] (* елемент другої строки, першого стовпця *)
```

- Розмірність масиву:

```
Dimensions[m] (*для прямокутної матриці повертає {n, m}. Якщо рядки різної довжини
Length[m] (*кількість рядків*)
Length[m[[1]]] (*кількість елементів в першому рядку матриці m*)
```

- Транспонування матриці:

```
Transpose[m]
```

- Масив з нулів та одинична матриця:

```
ConstantArray[0, {3, 3}]
IdentityMatrix[3]
```

- Арифметичні операції виконуються поелементно:

```
In[1]:= v = {1, 2, 3, 4};
        v + 10 (*додавання до масива числа*)
        v + {10, 10, 10, 10} (*додавання двох масивів*)

Out[1]= {11, 12, 13, 14}
Out[2]= {11, 12, 13, 14}
```

При додаванні двох масивів необхідно, щоб їх кількість елементів була однаковою. Якщо застосувати знак віднімання, множення чи ділення, отримаємо масив з елементів, які результатом віднімання віднімання, множення чи ділення відповідних елементів двох масивів.

- **Матричне множення:** Правильне матричне множення реалізується функцією `Dot` або оператором `"."`, що їй відповідає: `Dot[M1, M2]` або `M1.M2`.

При множенні матриць $M1(n \times m1)$ і $M2(m2 \times p)$ необхідне виконання такої умови: кількість стовпчиків першої матриці ($m1$) повинно дорівнювати кількості рядків другої матриці ($m2$). У результаті множення отримується матриця розмірністю $(n \times p)$. Наприклад:

```
In[1]:= {{1, 2, 3}}.{{1}, {2}, {3}}
```

```
Out[1]= {{14}}
```

- Застосування функції f до всіх елементів масиву v ($\text{Map}[f, v]$):

```
In[1]:= Map[Sin, {0, Pi/6, Pi/4, Pi/2}]
```

```
Out[1]= {0, 1/2, 1/√2, 1}
```

Замість $\text{Map}[f, v]$ можна використовувати операторний запис $f /@ v$

1.5. Правила підстановки

У середовищі *Mathematica* важливим механізмом є система правил підстановки, яка дозволяє замінювати вирази згідно із заданими шаблонами. Для підстановки використовується оператор $\rightarrow$, а застосування правила виконується за допомогою оператора $/.$ (скорочення від *ReplaceAll*).

Приклади простих підстановок

```
In[1]:= x^2 + y /. x -> 3
```

```
Out[1]= 9 + y
```

```
In[2]:= expr = a + b + c;
        expr /. {a -> 1, c -> 5}
```

```
Out[2]= 6 + b
```

Таким чином, можна обчислювати значення виразів без необхідності присвоювати змінним числові значення, залишаючи їх невизначеними для подальших аналітичних перетворень.

Правила з шаблонами

Можна задавати правила, які працюють за шаблоном:

```
In[3]:= f[x^2] /. f[z_] -> z + 1
```

```
Out[3]= 1 + x^2
```

Комбінування з іншими операціями

Підстановку можна застосовувати разом із функціями *Replace*, *ReplaceAll*, *ReplaceRepeated*:

```
In[4]:= expr = 2*(Sin[a]^2 + Cos[a]^2);
        expr /. {Sin[x_]^2 + Cos[x_]^2 -> 1}
```

```
Out[4]= 2
```

Це дозволяє проводити аналітичні спрощення, трансформації виразів та реалізовувати алгебраїчні алгоритми на основі шаблонного зіставлення.

1.6. Логічні та умовні оператори

У Wolfram Mathematica логічні оператори застосовуються для побудови умовних виразів, що повертають логічні константи `True` або `False`. Основні оператори наведено в таблиці:

Оператор	Опис	Приклад
And або <code>&&</code>	Логічне «І»	<code>(x > 0) && (y > 0)</code>
Or або <code> </code>	Логічне «АБО»	<code>(x > 0) (y > 0)</code>
Not або <code>!</code>	Логічне заперечення	<code>! (x > 0)</code>
Xor	Виключне «АБО»	<code>Xor[x > 0, y > 0]</code>
Equivalent	Логічна еквівалентність	<code>Equivalent[a, b]</code>

Пріоритет: `Not` > `And` > `Or`.

Приклади:

```
In[1]:= x = 2; y = 4;
        (x > 0) && (y < 5)
```

```
Out[2]= True
```

```
In[3]:= Not[x == 0]
```

```
Out[3]= True
```

Умовні конструкції

В Mathematica умовні вирази реалізуються за допомогою таких конструкцій:

If[умова, вираз1, вираз2] – якщо умова істинна, виконується **вираз1**, інакше **вираз2**.

Which[умова1, дія1, умова2, дія2, ...] – послідовно перевіряються умови і виконується та дія, яка відповідає першій умові, що виконується (решта умов не перевіряється).

```
In[1]:= x = 1;
        If[x > 0, "positive", "non-positive"]
Out[1]= "positive"
```

```
In[2]:= x = 0;
        Which[x < 0, "negative", x > 0, "positive", x == 0, "zero"]
Out[2]= "zero"
```

1.7. Цикли і рекурсія

У Wolfram Mathematica цикли дозволяють повторювати певні обчислення. Найчастіше вони використовуються для простих ітерацій, але загалом у Mathematica перевага віддається вбудованим функціям, що працюють зі списками.

Цикл For

Використовується для організації класичного циклу з лічильником:

For[ініціалізація, умова, оновлення, тіло]

ініціалізація — задає початкове значення лічильника (наприклад, `i = 1`);

умова — вираз, який перевіряється перед кожною ітерацією (наприклад, `i <= 5`);

оновлення — операція, що виконується після кожної ітерації (наприклад, `n++`).

тіло — команда, або декілька команд, розділених крапкою з комою (;), які виконуються під час кожної ітерації.

Приклад: вивести числа від 1 до 5:

```
In[1]:= For[i = 1, i <= 4, i++, Print[i]]
```

```
1
2
3
4
```

Цикл While

Виконує тіло циклу, поки умова істинна:

```
In[2]:= n = 1;
While[n! <= 150, Print[Row[{n, "!_=_", n!}]]]; n++]
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Функція `Row[ ]` створює рядок з елементів списку, навіть якщо елементи різного типу. Цикл виконує тіло доти, поки умова істинна. Тобто кількість ітерацій заздалегідь може бути невідома. Важливо не забути команду `n++` у тілі циклу. Якщо її пропустити, умова завжди залишатиметься істинною, і цикл стане нескінченним.

Цикл Do

Часто використовується для повторення дії задану кількість разів:

```
In[3]:= Do[Print[i^2], {i, 0, 3}]
```

```
0
1
4
9
```

Table

Функція `Table` створює список значень (генератор списків за заданим правилом):

```
In[4]:= Table[i^2, {i, 1, 5}]
```

```
Out[4]= {1, 4, 9, 16, 25}
```

Рекурсивні функції

Рекурсія дозволяє визначити функцію через саму себе. Наприклад, числа Фібоначчі:

```
In[5]:= fibonacci[0] = 0;
        fibonacci[1] = 1;
        fibonacci[n_] := fibonacci[n - 1] + fibonacci[n - 2]
```

```
fibonacci[10]
```

```
Out[5]= 55
```

Часті помилки новачків при роботі з Wolfram Mathematica - Не використовують квадратні дужки для функцій

- Забувають про те, що імена стандартних функцій пишуться з великої літери
- Плутають знаки = та ==
- Забувають про **Shift+Enter** для виконання коду, та про те, що курсор повинен знаходитись в тій комірці, код якої необхідно виконати.

1.8. Операції математичного аналізу у Wolfram Mathematica

У цьому розділі розглянемо основні засоби Wolfram Mathematica для виконання операцій математичного аналізу: обчислення похідних, інтегралів, границь, сум нескінченних рядів, а також чисельне обчислення цих величин.

Границі

Для обчислення границі від функції $f(x)$, коли змінна x прямує до значення x_0 , використовується команда **Limit**. Формат команди має наступний вигляд:

$$\text{Limit}[f(x), x \rightarrow x_0]$$

Команда **Limit** має одну опцію **Direction**, яка може набувати двох значень: 1 та -1 .

- Опція **Direction** $\rightarrow -1$ дозволяє обчислити **лівосторонню границю** (тобто $x \rightarrow x_0^-$).
- Опція **Direction** $\rightarrow 1$ — **правосторонню границю** (тобто $x \rightarrow x_0^+$).

Нижче наведено синтаксис використання функції **Limit** та приклади розрахунків:

<code>Limit[f[x], x -> a]</code>	(* Границя при $x \rightarrow a$ *)
<code>Limit[f[x], x -> Infinity]</code>	(* Границя при $x \rightarrow \text{Infinity}$ *)
<code>Limit[f[x], x -> 0, Direction -> 1]</code>	(* Правостороння границя *)
<code>Limit[f[x], x -> 0, Direction -> -1]</code>	(* Лівостороння границя *)

```
In[1]:= Limit[1/x, x -> 0]
```

```
Out[1]= ∞
```

```
In[2]:= Limit[1/(x - a), x -> a, Direction -> 1]
```

```
Out[2]= -∞
```

```
In[3]:= Limit[Sin[n*x] / x, x -> 0]
```

```
Out[3]= n
```

```
In[4]:= Limit[Sin[x] / x, x -> Infinity]
```

```
Out[4]= 0
```

Рис. 1.1. Приклади розрахунку границь

Похідні

Для обчислення похідних у Mathematica застосовується функція `D`. Синтаксис:

<code>D[f[x], x]</code>	(* Похідна функції <code>f[x]</code> за <code>x</code> *)
<code>D[f[x, y], x]</code>	(* Часткова похідна за <code>x</code> *)
<code>D[f[x], {x, 2}]</code>	(* Друга похідна *)
<code>D[f[x, y], x, y]</code>	(* Змішана похідна *)

In[1]:= `D[x^n, x]`

Out[1]= $n x^{-1+n}$

In[2]:= `D[Log[x], x]`

Out[2]= $\frac{1}{x}$

In[3]:= `D[Log[b, x], x]`

Out[3]= $\frac{1}{x \operatorname{Log}[b]}$

In[4]:= `D[Log[x, a], x]`

Out[4]= $-\frac{\operatorname{Log}[a]}{x \operatorname{Log}[x]^2}$

In[5]:= `D[Sin[x y] / (x^2 + y^2), x, y]`

Out[5]=
$$-\frac{2 x^2 \operatorname{Cos}[x y]}{(x^2 + y^2)^2} - \frac{2 y^2 \operatorname{Cos}[x y]}{(x^2 + y^2)^2} + \frac{\operatorname{Cos}[x y]}{x^2 + y^2} + \frac{8 x y \operatorname{Sin}[x y]}{(x^2 + y^2)^3} - \frac{x y \operatorname{Sin}[x y]}{x^2 + y^2}$$

Рис. 1.2. Приклади похідних різних функцій

Функція `Derivative` дозволяє створювати анонімні функції з фіксованим порядком похідних `Derivative[1][f][x]`, або їй еквівалентний операторний вигляд `f'[x]` - перша похідна, `f''[x]` - друга похідна. У такому вигляді зручно записувати диференціальні рівняння відносно функції `f[x]`.

Інтегрування

Обчислення визначених і невизначених інтегралів є однією з базових задач математичного аналізу. У системі *Wolfram Mathematica* для цього використовуються команди `Integrate` та `NIntegrate`.

Невизначені інтеграли (первісні)

Невизначений інтеграл (первісна) функції $f(x)$ обчислюється за допомогою команди:

`Integrate[f[x], x]`

Це дає аналітичний вираз первісної від функції $f(x)$ за змінною x .

Приклад. Обчислити первісну функції $\frac{1}{1+x^2}$:

```
Integrate[1/(1 + x^2), x]
(* ArcTan[x] *)
```

Визначені інтеграли

Щоб обчислити визначений інтеграл на відрізку $[a, b]$, необхідно вказати межі інтегрування:

```
Integrate[f[x], {x, a, b}]
```

Приклад. Обчислити визначений інтеграл від e^{-x^2} на всій числовій прямій:

```
Integrate[Exp[-x^2], {x, -Infinity, Infinity}]
(* Sqrt[Pi] *)
```

Чисельне інтегрування

Якщо аналітичне інтегрування складне або неможливе, використовують команду:

```
NIntegrate[f[x], {x, a, b}]
```

Приклад. Обчислити наближене значення інтеграла:

```
NIntegrate[Sin[x]/x, {x, 1, 100}]
(* 1.56367 *)
```

Команда `NIntegrate` підтримує вибір чисельного методу через опцію `Method` $\rightarrow$ Серед доступних методів:

- "GaussKronrod"
- "Trapezoidal"
- "MonteCarlo"
- "QuasiMonteCarlo"
- "Oscillatory"
- "DoubleExponential"

Порівняння аналітичного та чисельного інтегрування

Визначимо інтеграл $\int_0^\infty \cos^2 x \cdot e^{-x^2} dx$ аналітично і чисельно, та порівняємо час виконання. Для оцінки продуктивності обчислень в Mathematica використовується функція `Timing`, яка повертає список із часу виконання та результату:

```
In[1]:= Timing[Integrate[Cos[x]^2 Exp[-x^2], {x, 0, Infinity}]]

Out[1]= {0.1875, (1 + e) Sqrt[Pi] / (4 e)}

In[2]:= (1 + e) Sqrt[Pi] / (4 e) // N

Out[2]= 0.606126

In[3]:= methods = {"GaussKronrod", "Trapezoidal", "MonteCarlo", "QuasiMonteCarlo",
  "Oscillatory", "DoubleExponential"};

In[4]:= Timing[
  Table[NIntegrate[Cos[x]^2 Exp[-x^2], {x, 0, Infinity},
    Method -> methods[[i]]], {i, 1, 6}]]

Out[4]= {0.046875, {0.606126, 0.606126, 0.615121, 0.606197, 0.606126, 0.606126}}
```

Як видно, результати числового інтегрування різними методами (за винятком ймовірнісних методів "MonteCarlo" та "QuasiMonteCarlo") збігаються з аналітичним розв'язком. При цьому числові методи забезпечують значно швидше отримання результату (шість числових значень інтегралу отримано в 4 рази швидше ніж аналітичний розрахунок). Водночас аналітичний підхід має вищу цінність, оскільки дає абсолютно точний вираз, а не лише числове наближення.

Розклад функції в ряд Тейлора

У математичному аналізі розклад функції $f(x)$ у ряд Тейлора поблизу деякої точки x_0 дозволяє наблизити її значення за допомогою многочлена, що містить похідні функції в цій точці. Такий розклад з точністю до n -го степеня має вигляд:

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \mathcal{O}((x - x_0)^{n+1}) \quad (1.1)$$

Тут $\mathcal{O}((x - x_0)^{n+1})$ — залишковий член, який характеризує точність наближення.

Розклад у ряд Тейлора є потужним інструментом для локального наближення складних функцій. Він широко використовується у чисельному аналізі, символьних обчисленнях, а також для побудови апроксимацій та аналізу похибок.

У середовищі *Wolfram Mathematica* для побудови розкладу в ряд Тейлора використовується команда `Series` з таким загальним синтаксисом:

`Series[f[x], {x, x0, n}]`

де:

- `f[x]` — функція, яку потрібно розкласти;
- `x` — змінна;
- `x0` — точка розкладу;
- `n` — максимальний степінь наближення.

Розкладемо декілька відомих функцій у ряд Тейлора поблизу точки $x_0 = 0$ (ряд Маклорена) до п'ятого степеня:

In[1]:= `Series[Sin[x], {x, 0, 5}]`

Out[1]= $x - \frac{x^3}{6} + \frac{x^5}{120} + \mathcal{O}[x]^6$

In[2]:= `Series[Cos[x], {x, 0, 5}]`

Out[2]= $1 - \frac{x^2}{2} + \frac{x^4}{24} + \mathcal{O}[x]^6$

In[3]:= `Series[Exp[x], {x, 0, 5}]`

Out[3]= $1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \mathcal{O}[x]^6$

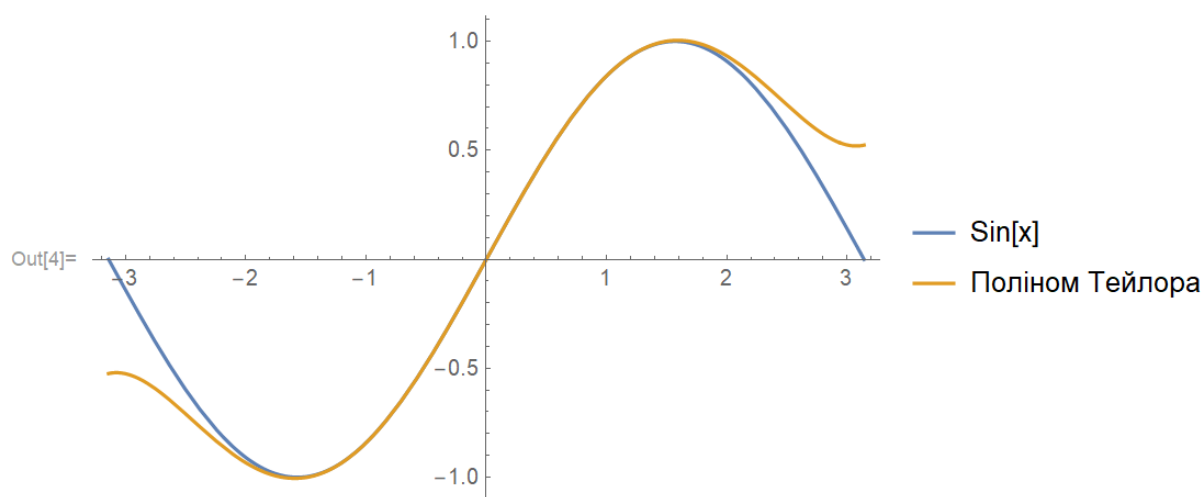
In[4]:= `Series[Log[1 + x], {x, 0, 5}]`

Out[4]= $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + \mathcal{O}[x]^6$

Отримані степеневі ряди добре наближують відповідні функції поблизу точки $x = 0$. Ряд *Тейлора* біля точки розкладу $x = 0$ називають також *рядом Маклорена*. Наприклад для функції $\text{Sin}[x]$ три доданки розкладу біля точки $x = 0$ задовільно описують функцію в межах $[-\pi/2, \pi/2]$:

```
In[1]:= f[x_] := Sin[x]
g = Normal[Series[f[x], {x, 0, 5}]]
ft = Function[x, g];
Plot[{f[x], ft[x]}, {x, -Pi, Pi}, PlotLegends -> {"Sin[x]", "Поліном Тейлора"}]
```

Out[2]= $x - \frac{x^3}{6} + \frac{x^5}{120}$



Розділ 2

Побудова графіків функцій у Wolfram Mathematica

У Wolfram Mathematica графіки будуються за допомогою потужних візуалізаційних функцій. Найчастіше використовуються: `Plot`, `ListPlot`, `Plot3D`, `ListPlot3D` та інші.

2.1. Побудова графіка за допомогою функції `Plot[]`

Для найпростішого способу побудови графіка функції $f(x)$ на відрізку $[a, b]$ використовується команда: `Plot[f[x], x, a, b]`

Приклад:

```
In[1] := Plot[Log[x]*Sin[x], {x, 0, 15}]
```

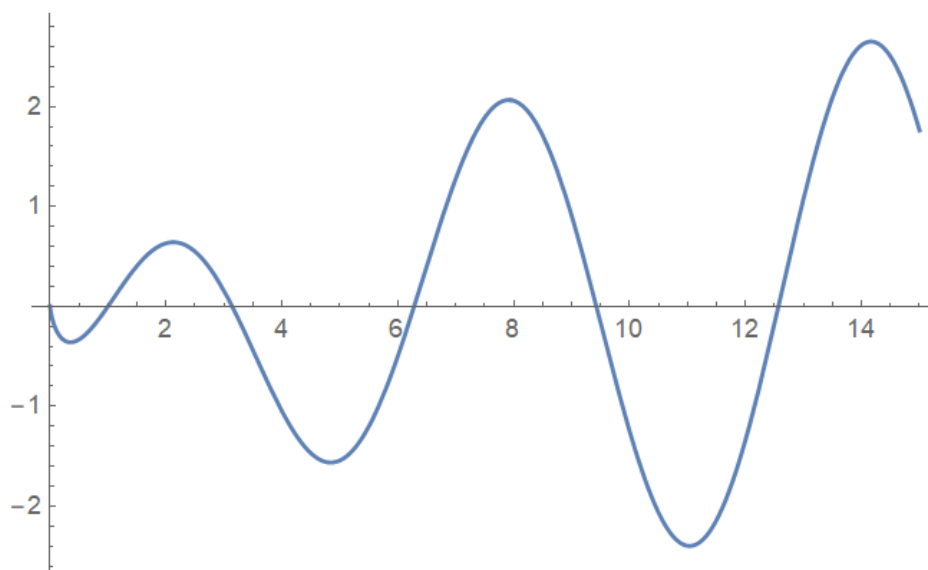


Рис. 2.1. Приклад побудови графіка функцією `Plot`

Цей приклад демонструє побудову функції в заданому інтервалі з стандартним налаштуваннями. Для побудови графіка необхідно в першому аргументі вказати функцію, яка повертає числові значення при різних значеннях параметра та в другому аргументі задати список з 3 елементів ім'я параметра, початкове значення, кінцеве значення. Після цих обов'язкових аргументів слідує опції (в довільному порядку), які дозволяють налаштовувати графік. Список та можливі значення опцій наведено нижче в таблиці.

Табл. 2.1. Найпоширеніші опції функції Plot у Wolfram Mathematica

Опція	Призначення і приклад
PlotStyle	Стиль кривої: колір, товщина, тип лінії. Приклад: PlotStyle -> {Red, Thick}
PlotRange	Межі відображення по осі y . Приклад: PlotRange -> {0, 2}
AxesLabel	Підписи осей. Приклад: AxesLabel -> {"x", "y"}
Frame	Рамка навколо графіка (True/False). Приклад: Frame -> True
GridLines	Додавання сітки. Приклад: GridLines -> Automatic
PlotLegends	Легенда до графіка. Приклад: PlotLegends -> {"sin", "cos"}
PlotLabel	Заголовок графіка. Приклад: PlotLabel -> "Графік sin(x)"
ImageSize	Розмір зображення. Приклад: ImageSize -> 400
AspectRatio	Співвідношення сторін графіка. Приклад: AspectRatio -> 1/GoldenRatio
Filling	Заповнення області під кривою. Приклад: Filling -> Axis
Mesh	Лінії або точки на кривій. Приклад: Mesh -> 10
Ticks	Стиль і положення поділок на осях. Приклад: Ticks -> Automatic
FrameTicks	Поділки на рамці (якщо Frame -> True). Приклад: FrameTicks -> Automatic
PlotPoints	Кількість початкових точок дискретизації. Приклад: PlotPoints -> 100
PlotTheme	Стиль оформлення (тема). Приклад: PlotTheme -> "Scientific"

2.2. Графік кількох функцій за допомогою Plot з використанням опцій

Можна одночасно побудувати кілька функцій:

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 Pi},
  PlotStyle -> {{Red, Thick}, {Blue, Dashed, Thick}},
  AxesLabel -> {"x", "y"}, PlotLabel -> "Графіки sin(x) та cos(x)",
  PlotLegends -> {"sin(x)", "cos(x)"}, GridLines -> Automatic,
  Frame -> True, FrameLabel -> {"x", "Значення функцій"},
  FrameStyle -> Black, Filling -> {1 -> {2}},
  AspectRatio -> 1/GoldenRatio, ImageSize -> 400, PlotRange -> All,
  Background -> LightYellow, Mesh -> 10,
  MeshStyle -> {PointSize[Medium]},
  Epilog -> {Text[Style["Максимум", Bold, 12, Red], {Pi/2, 1.05}]}
```

Ця команда створює один графік із двома кривими: $\sin(x)$ і $\cos(x)$. Для побудови використано декілька опцій, значення яких легко встановити співставивши код команди та отриманий результат.

2.3. Графік кількох функцій з використанням Plot та додаткових опцій оформлення

Команда `Plot` дозволяє одночасно будувати графіки кількох функцій з гнучкими параметрами візуалізації. У прикладі нижче побудовано графіки функцій $\sin(x)$ та $\cos(x)$ на інтервалі від 0 до 2π з додатковими елементами оформлення:

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 Pi},
  PlotStyle -> {{Red, Thick}, {Blue, Dashed, Thick}},
  AxesLabel -> {"x", "y"},
  PlotLabel -> "Графіки sin(x) та cos(x)",
  PlotLegends -> {"sin(x)", "cos(x)"},
  GridLines -> Automatic,
  Frame -> True,
  FrameLabel -> {"x", "Значення функцій"},
  FrameStyle -> Black,
  Filling -> {1 -> {2}},
  AspectRatio -> 1/GoldenRatio,
  ImageSize -> 400,
  PlotRange -> All,
  Background -> LightYellow,
  Mesh -> 10,
  MeshStyle -> {PointSize[Medium]},
  Epilog -> {
    Text[Style["cos(x)=sin(x)", Bold, 12, Darker[Green]], {Pi/4, 1.05}],
    Darker[Green],
    PointSize -> 0.02,
    Point[{Pi/4, Cos[Pi/4]}],
    Dashed,
    Line[{Pi/4, Cos[Pi/4]}, {Pi/4, -1.1}],
    Text[Style["\[Pi]/4", 12], {1, -1}]
  }
]
```

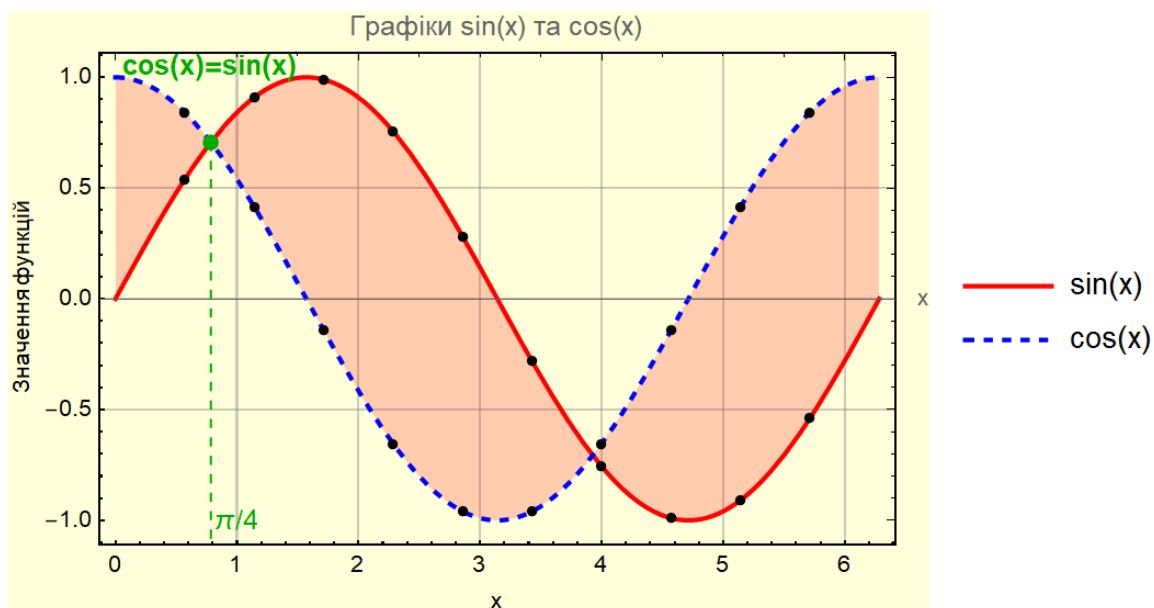


Рис. 2.2. Побудова графіка функцією Plot з використанням опцій

Цей графік містить дві криві: $\sin(x)$ (червона суцільна лінія) та $\cos(x)$ (синя пунктирна). Окрім стандартного підпису осей, заголовка та легенди, додано:

- точку перетину $\cos(x) = \sin(x)$ при $x = \pi/4$, виділену зеленим кольором;
- підпис “ $\cos(x)=\sin(x)$ ” над цією точкою;
- вертикальну пунктирну лінію до осі x ;
- позначку $\pi/4$ на осі x .

Таке оформлення наочно демонструє можливості функції `Plot` у середовищі *Mathematica* щодо налаштування зовнішнього вигляду графіків, зручного підписування, стилізації, а також роботи з кількома функціями одночасно.

2.4. Побудова точкових графіків

Для візуалізації набору дискретних значень використовується функція `ListPlot`. Якщо задано лише список значень, то координати по осі x призначаються автоматично — за порядковими номерами елементів списку:

```
ListPlot[{1, 3, -2, 4, 1}]
```

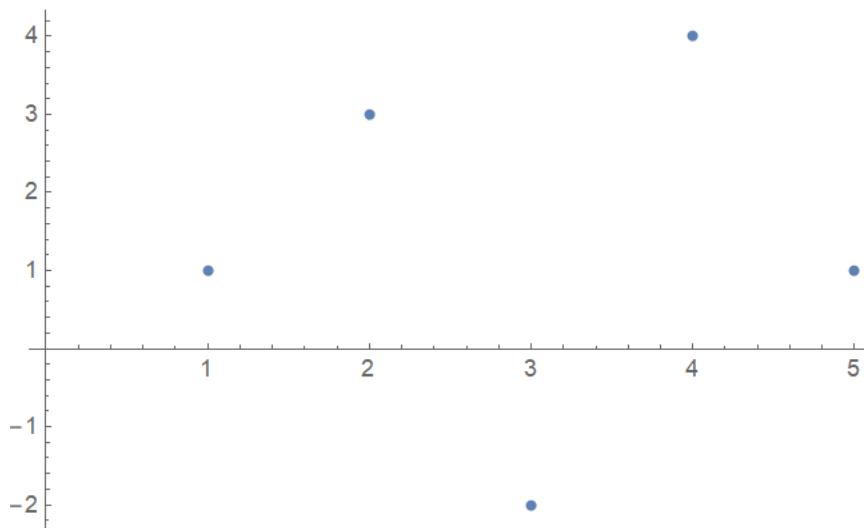


Рис. 2.3. Побудова точкового графіка функцією `ListPlot`

Цей код побудує графік для точок $(1,1)$, $(2,3)$, $(3,-2)$, $(4,4)$, $(5,1)$. Тобто значення координат точок по вісі абсцис вибрано автоматично. Щоб явно задати координати кожної точки, необхідно використовувати список пар координат $\{0, 1, 1, 3, 2, -2, 3, 4, 4, 1\}$.

2.5. Об'єднання графіків точок і функцій

Функція `Show` дозволяє поєднувати кілька графічних об'єктів в одному зображенні. Наприклад, поєднаймо точковий графік і графік функції:

```
point := ListPlot[{{0, 0.5}, {1, 2}, {2, -1.8}, {3, 0}, {4, 2}},
  PlotStyle -> {Red, PointSize[Large]}, AxesLabel -> {"x", "y"}]
fun := Plot[2*Sin[2 x], {x, 0, 4}]
Show[point, fun, PlotLabel -> "Об'єднання графіків"]
```

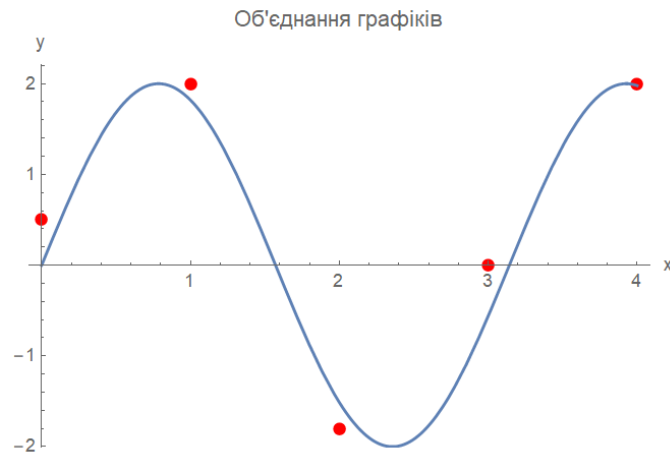


Рис. 2.4. Об'єднання графіків Plot та ListPlot

У цьому прикладі:

- ListPlot відображає набір точок, які показані великими червоними маркерами.
- Plot будує графік функції $y = 2 \sin(2x)$ на інтервалі від 0 до 4.
- Show об'єднує обидва графіки на одній системі координат і додає заголовок.

Цей підхід зручний, коли потрібно одночасно зобразити аналітичну залежність і експериментальні або табличні дані для порівняння.

2.6. Графік у тривимірному просторі

Для функцій двох змінних використовується Plot3D:

```
Plot3D[Sin[x y], {x, -3, 3}, {y, -3, 3}]
```

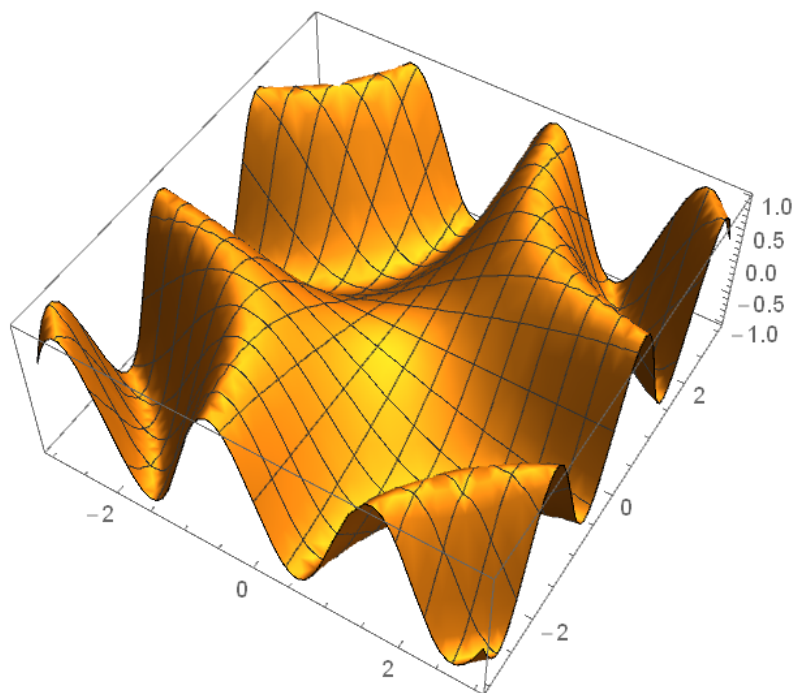


Рис. 2.5. Побудова 3D графіка за допомогою функції Plot3D

Цей приклад створює поверхню для функції $z = \sin(x * y)$. Функція Plot3D має велику кількість опцій, застосування яких дозволяє отримати графік функції з необхідними елементами оформлення. Список цих опцій та їх значення за замовчуванням можна отримати командою `Text[Plot3D]`.

2.7. Графік 3D поверхні з додатковими просторовими об'єктами

Нижче подано приклад побудови тривимірної поверхні функції $z = \sin(xy)$ з використанням кількох опцій, а також додавання до графіка допоміжних об'єктів: сфери та ліній-перетинів.

```
function3D :=
  Plot3D[Sin[x y], {x, -3, 3}, {y, -3, 3},
    PlotStyle -> Opacity[0.4], Mesh -> 20, PlotPoints -> 50,
    AxesLabel -> {"x", "y", "z"},
    PlotLabel -> Style["Графік z(x,y) = sin(x y)", Bold, Black]]

redSphere := Graphics3D[{Red, Sphere[{0, 0, 0.4}, 0.4]}]

line1 := Graphics3D[{Red, Thick, Dashed,
  Line[Table[{t, t, Sin[t^2]}, {t, -3, 3, 0.1}]]}]

line2 := Graphics3D[{Blue, Thick, Dashed,
  Line[Table[{t, -t, -Sin[t^2]}, {t, -3, 3, 0.1}]]}]

Show[function3D, redSphere, line1, line2]
```

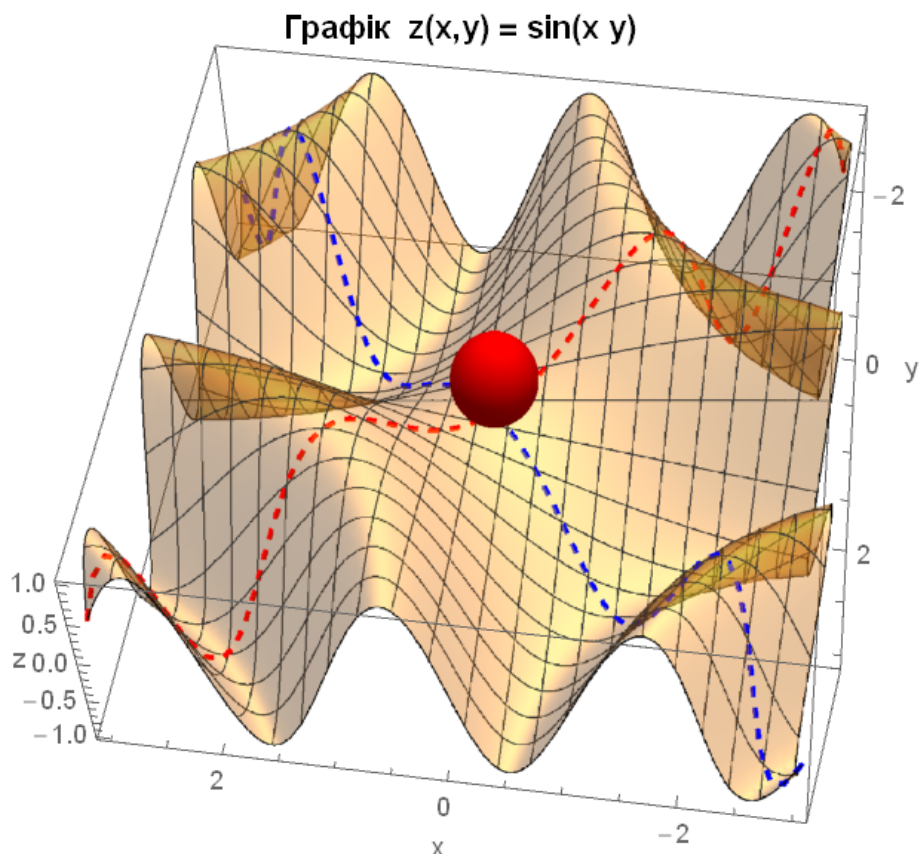


Рис. 2.6. Об'єднання графіків Plot3D та Graphics3D за допомогою Show

Цей код буде напівпрозору поверхню функції $z = \sin(xy)$ з густою сіткою (20 ліній), збільшеною точністю побудови (`PlotPoints` $\rightarrow$ 50) та підписами осей.

До неї додаються:

- **Червона сфера** в центрі координат з радіусом 0.4.
- **Червона пунктирна лінія** — перетин поверхні з площиною $x = y$.
- **Синя пунктирна лінія** — перетин поверхні з площиною $x = -y$.

Таке поєднання графіків дозволяє краще зрозуміти геометрію функції у просторі та її симетрії.

Розділ 3

Інтерактивні демонстрації у Wolfram Mathematica

У Wolfram Mathematica команда **Manipulate** дозволяє створювати інтерактивні демонстрації, де користувач може керувати параметрами за допомогою повзунків, прапорців, меню тощо.

3.1. Основний синтаксис

`Manipulate[вираз, {змінна, min, max}]`

де:

- **вираз** — об'єкт, який буде динамічно оновлюватися;
- **змінна** — параметр, що змінюється;
- **min, max** — межі зміни параметра.

Приклад роботи Manipulate для зміни частоти та фази

```
Manipulate[  
  Plot[Sin[a x + b], {x, 0, 2 Pi}],  
  {a, 1, 5},  
  {b, 0, Pi} ]
```

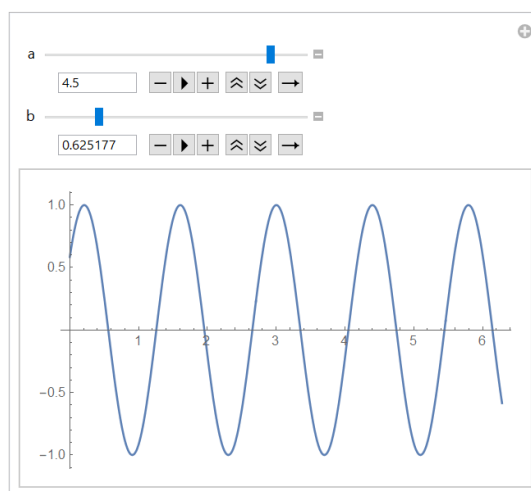


Рис. 3.1. Інтерактивний графік $\text{Sin}[a x + b]$

Цей приклад дозволяє змінювати частоту синусоїди та зсув фаз за допомогою повзунків.

3.2. Налаштування елементів керування ControlType

SetterBar

```
Manipulate[
  Plot[x^n, {x, 0, 3}], {{n, 2, "Степінь"}, {0.25, 0.5, 1, 2, 4},
    ControlType -> SetterBar}]
```

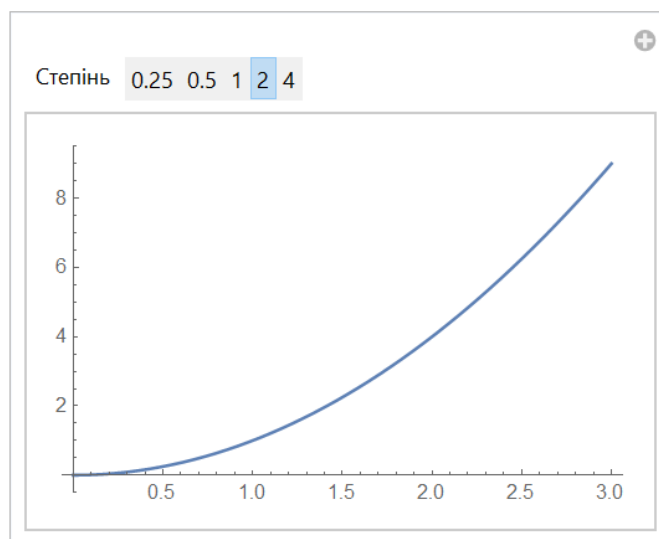


Рис. 3.2. Керування за допомогою SetterBar

```
Manipulate[Plot[f[x], {x, 0, 10}], {f, {Sin, Cos, Exp, Log}}]
```

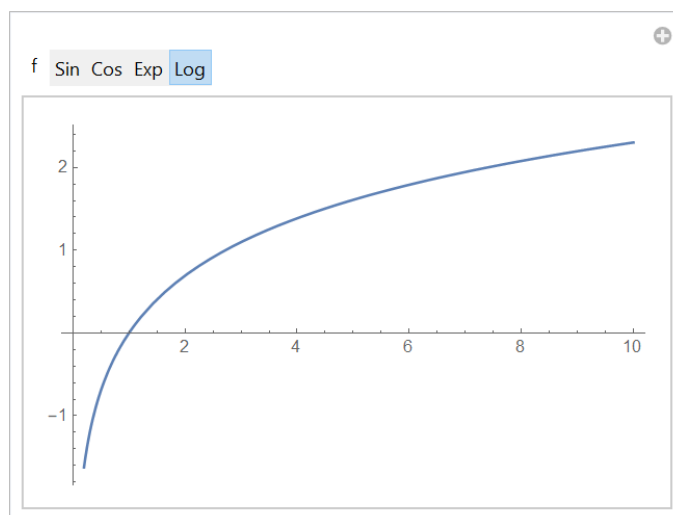


Рис. 3.3. Вибір функції для відображення на графіку

RadioButtonBar

```
Manipulate[
  Plot[x^n, {x, 0, 3}], {{n, 2, "Степінь"}, {0.25, 0.5, 1, 2, 4},
    ControlType -> RadioButtonBar}]
```

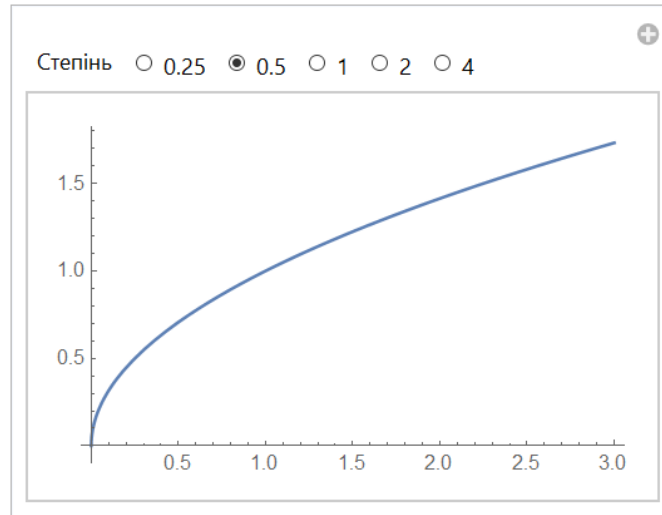


Рис. 3.4. Керування за допомогою RadioButtonBar

Використання прапорців (Checkbox)

```
Manipulate[
  Plot[{If[showSin, Sin[x], Nothing],
        If[showCos, Cos[x], Nothing]}, {x, 0, 2 Pi},
        PlotRange -> {-1.5, 1.5},
        Epilog -> {
          Red, If[info && showSin, Text[Style["sin(x)", 16], {1.8, 1.1}]],
          Blue, If[info && showCos, Text[Style["cos(x)", 16], {0.4, 1.1}]]
        },
        PlotStyle -> {Red, Blue}, ImageSize -> 450],
  Row[{
    "sin(x) ", Checkbox[Dynamic[showSin]],
    "cos(x) ", Checkbox[Dynamic[showCos]],
    "labels ", Checkbox[Dynamic[info]]
  }]]
```

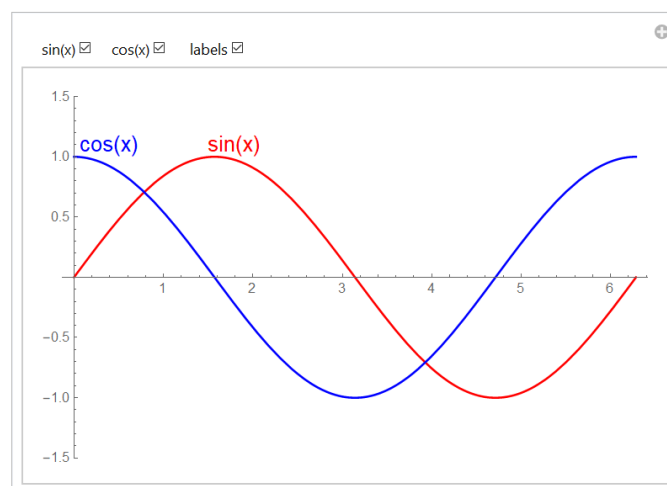


Рис. 3.5. Використання Checkbox для вибору графіка

Анімація гармонічного осцилятора

```
Manipulate[
  Graphics[{Red, Dashed, Circle[{0, 0}, 1],
```

```

Blue, Disk[{Cos[w t], Sin[w t]}, 0.1]},
PlotRange -> {{-1.1, 1.1}, {-1.1, 1.1}},
Axes -> True],
{t, 0, 6, AnimationRate -> 0.2},
{w, 1, 3}
]

```

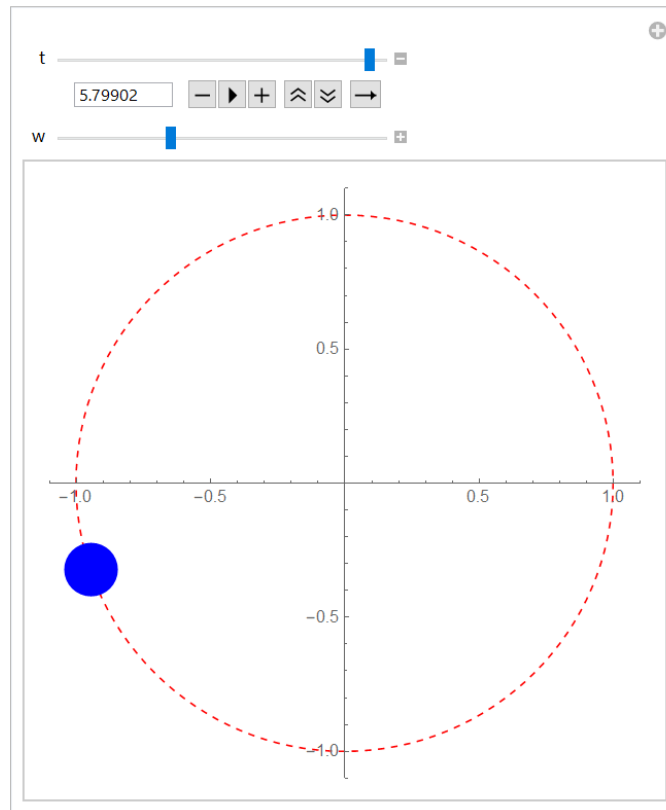


Рис. 3.6. Анімація гармонічного осцилятора

Використання ControlType для різних елементів

```

Manipulate[ Plot[a Sin[b x], {x, 0, 10}],
{{a, 1, "Амплітуда"}, {0.1, 1, 3, 5}, ControlType -> PopupMenu},
{{b, 1, "Частота"}, 0.5, 3, ControlType -> Slider}]

```

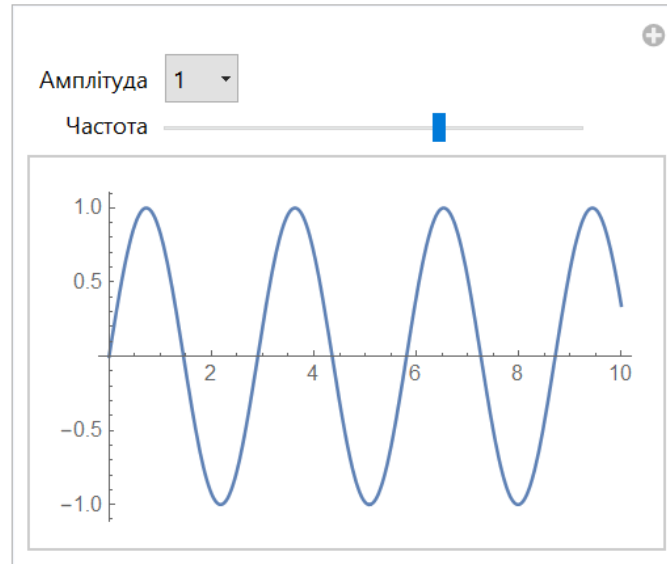


Рис. 3.7. Демонстрація інтерактивного графіка незатухаючих коливань

3.3. Демонстрація затухаючих коливань з використанням Manipulate

Інтерактивні можливості Wolfram Mathematica дозволяють досліджувати затухаючі коливання в реальному часі, змінюючи параметри за допомогою повзунків. Нижче наведено приклад використання `Manipulate` для побудови графіка функції:

$$y(t) = a \cdot e^{-dt} \cdot \cos(bt),$$

де a — амплітуда, b — частота, d — коефіцієнт затухання.

Код програми

```
Manipulate[
  Column[{Style[
    "a = " <> ToString[NumberForm[a, {3, 1}]] <>
    ", b = " <> ToString[NumberForm[b, {3, 1}]] <>
    ", d = " <> ToString[NumberForm[d, {3, 2}]], 12, Bold, Blue],
    Plot[{a*Exp[-d t]*Cos[b t], a*Exp[-d t], -a*Exp[-d t]}, {t, 0, 10},
      PlotRange -> {{0, 10}, {-5, 5}},
      PlotStyle -> {{Thick, Red}, {Dashed, Blue}, {Dashed, Blue}},
      Background -> Lighter[Yellow, 0.9],
      AxesLabel -> {"t", "y"}, GridLines -> Automatic, ImageSize -> 450,
      PlotLabel ->
        Style["Затухаючі коливання a*Exp[-d t]*Cos[b t]", 14, Black]]},
  {{a, 5, "Амплітуда"}, 0.1, 5, Appearance -> "Labeled"},
  {{b, 4, "Частота"}, 0.5, 5, Appearance -> "Labeled"},
  {{d, 0.4, "Затухання"}, 0, 1, Appearance -> "Labeled"}]
```

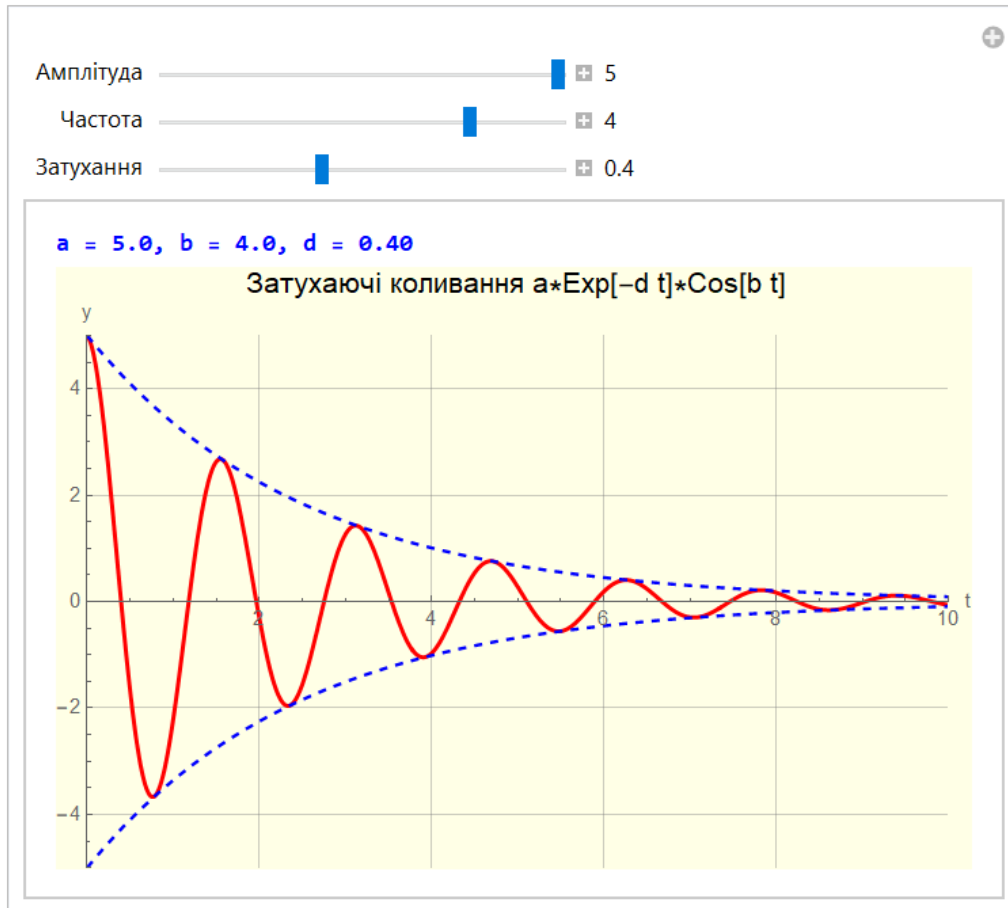


Рис. 3.8. Інтерактивна демонстрація коливань затухаючого осцилятора

Пояснення роботи

- Основна функція:

$$y(t) = a \cdot e^{-dt} \cdot \cos(bt)$$

відображається червоною товстою кривою.

- Додатково показано дві огинаючі криві:

$$y_{\text{верхня}} = a \cdot e^{-dt}, \quad y_{\text{нижня}} = -a \cdot e^{-dt}$$

синіми пунктирними лініями.

- Manipulate дозволяє змінювати параметри:

Амплітуда a : від 0.1 до 5;

Частота b : від 0.5 до 5;

Затухання d : від 0 до 1.

- Поточні значення параметрів відображаються над графіком.
- Область побудови: $t \in [0, 10]$, $y \in [-5, 5]$.

Особливості реалізації

1. Використано Column для розміщення тексту і графіка.
2. Форматування чисел через NumberForm.

3. Додано сітку (`GridLines -> Automatic`) і підписи осей.

4. Графік масштабовано до `ImageSize -> 450`.

Розклад в ряд Фур'є

Розглянемо функцію, яка на інтервалі $0 < x < 2\pi$ задається формулою:

$$f(x) = \frac{\pi - x}{2} \quad (3.1)$$

і періодично повторюється при розширенні цього інтервалу. Для такого періодичного продовження потрібно у формулу (3.1) замість x підставити вираз $\text{Mod}[x, 2\pi]$, де функція Mod - повертає остачу від ділення першого аргументу на другий і таким чином змінюється в межах $(0, 2\pi)$. Графік цієї функції на інтервалі в межах $(0, 2\pi)$ являє собою відрізок прямої, що з'єднує точки $(0, \pi/2)$ і $(2\pi, -\pi/2)$, і при періодичному продовженні утворює **пилкоподібну криву (sawtooth wave)**.

```
Manipulate[
  Plot[(Pi - Mod[x, 2 Pi])/2, {x, 0, xmax},
    Exclusions -> None], {{xmax, 4 Pi}, 2 Pi, 20 Pi}]
```

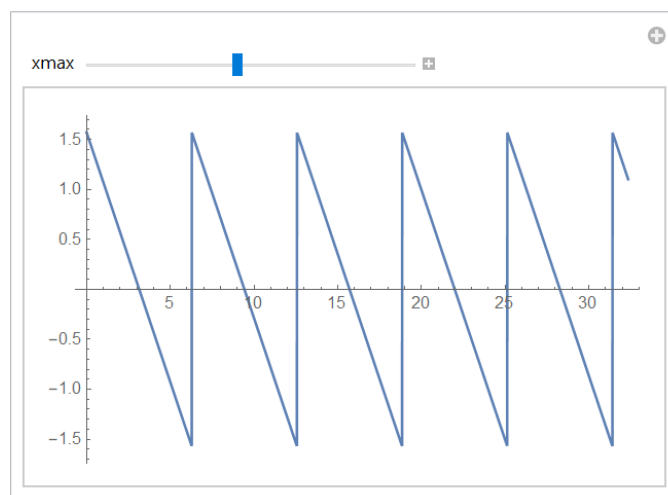


Рис. 3.9. Графік пилкоподібної функції

Ряд Фур'є функції

Функція $f(x)$ є непарною (відносно точки $x = \pi$ після зсуву), тому її розклад у тригонометричний ряд Фур'є містить лише синусоїдальні доданки:

$$f(x) \sim \sum_{n=1}^{\infty} \frac{\sin(nx)}{n}. \quad (3.2)$$

Часткові суми ряду

Часткова сума N перших членів ряду має вигляд:

$$S_N(x) = \sum_{n=1}^N \frac{\sin(nx)}{n}. \quad (3.3)$$

Приклад побудови в середовищі Wolfram Mathematica:

```

Manipulate[
  Plot[{If[showsum, Sum[Sin[n x]/n, {n, 1, nmax}]],
    If[showfp, (Pi - Mod[x, 2 Pi])/2]}, {x, -0.1, 20},
  PlotRange -> {-1.7, 2.1}, AxesLabel -> {"x", "f"},
  PlotStyle -> Thick, ImageSize -> 400, Exclusions -> None,
  Epilog -> {If[info,
    Text[Style[TraditionalForm[Sum[Sin[n x]/n, {n, 1, nmax}]], 11,
      Blue], {10, 1.8}]]], {nmax, 1, 50, 1,
  Appearance -> "Labeled"},
  Row[{"Пилкоподібна функція ", Checkbox[Dynamic[showfp]],
    "      Розклад в ряд Фур'є ", Checkbox[Dynamic[showsum]],
    "      Формула ", Checkbox[Dynamic[info]]}]]

```

На рис. 3.10 показано наближення функції $f(x)$ частковими сумами ряду Фур'є при різних кількості доданків:

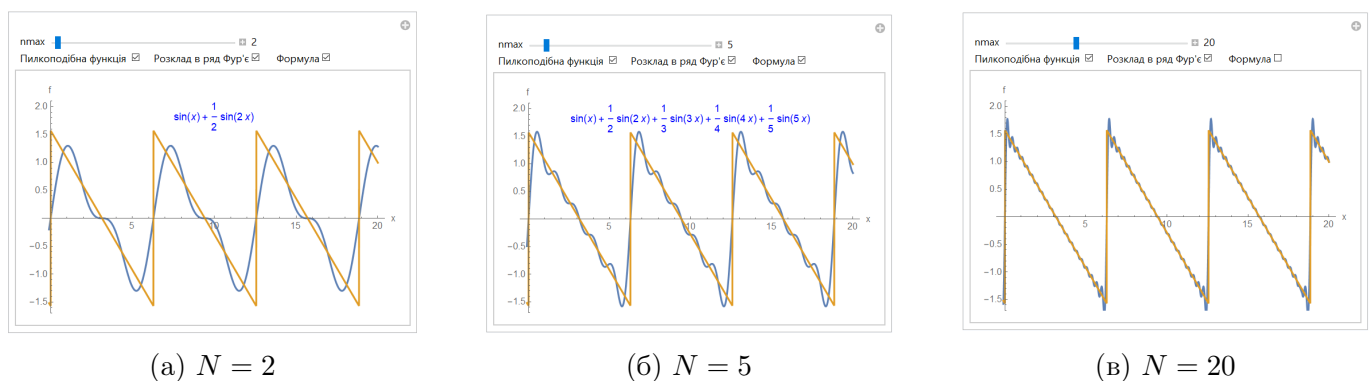


Рис. 3.10. Графіки часткових сум ряду Фур'є для різних N .

Опис програми. Наведена програма створює інтерактивний інтерфейс для дослідження наближення періодичної пилкоподібної функції рядом Фур'є. Вона реалізована за допомогою конструкції `Manipulate`, що дозволяє змінювати параметри в реальному часі.

Основні можливості програми:

- Зміна кількості доданків у частковій сумі ряду Фур'є ($n_{\max}$) у діапазоні від 1 до 50.
- Відображення за вибором:
 - вихідної пилкоподібної функції

$$f(x) = \frac{\pi - \text{Mod}(x, 2\pi)}{2},$$

- часткової суми ряду Фур'є

$$S_N(x) = \sum_{n=1}^N \frac{\sin(nx)}{n},$$

- аналітичної формули на графіку.

- Керування через елементи інтерфейсу (`Checkbox`) з можливістю вибору потрібних компонентів.

Результати. На рис. 3.10 показано приклади наближення функції частковими сумами ряду Фур'є при різних кількості доданків N :

- При $N = 2$ наближення є грубим, видно лише основні коливання.

- При $N = 5$ точність суттєво зростає, але спостерігається ефект коливань біля точок розриву (*феномен Гіббса*).
- При $N = 20$ апроксимація в середині інтервалів практично збігається з вихідною функцією, однак коливання в околі стрибків залишаються.

Таким чином, приклад ілюструє характерні властивості ряду Фур'є: із зростанням кількості доданків наближення покращується, але точне відтворення розривів залишається неможливим навіть при великих N .

3.4. Фігури Ліссажу

Фігури Ліссажу описують траєкторію точки, яка одночасно здійснює два гармонічні коливання взаємно перпендикулярно одне до одного:

$$\begin{cases} x(t) = A_x \sin(\omega_x t + \varphi), \\ y(t) = A_y \sin(\omega_y t), \end{cases}$$

де ω_x, ω_y — кутові частоти коливань, φ — початкова фаза.

Форма фігури залежить від відношення частот $\frac{\omega_x}{\omega_y} = \frac{n}{m}$ та різниці початкових фаз φ .

Програма в Mathematica

Наведений нижче код реалізує інтерактивну побудову фігур Ліссажу:

```
Manipulate[
  ParametricPlot[
    {Sin[n t + f], Sin[m t]}, {t, 0, 2 Pi},
    Frame -> True,
    GridLines -> Automatic,
    FrameLabel -> {
      "X", "Y",
      Style["Фігура Ліссажу " <> ToString[n] <> "/" <> ToString[m], 14, Black]
    }
  ],
  {{n, 6}, 1, 10, 1},
  {{m, 5}, 1, 10, 1},
  {{f, 0, "Початкова фаза"}, 0, Pi/2}
]
```

Інтерфейс програми

- Повзунки для зміни частот n, m (від 1 до 10).
- Повзунок для початкової фази φ (від 0 до $\pi/2$).
- Графік автоматично оновлюється при зміні параметрів.

Результати

Приклад побудови фігур Ліссажу для різних співвідношень частот показано на рис. 3.11.

Фігури Ліссажу широко застосовуються для візуалізації відношення частот двох гармонічних сигналів, зокрема на екранах осцилографів.

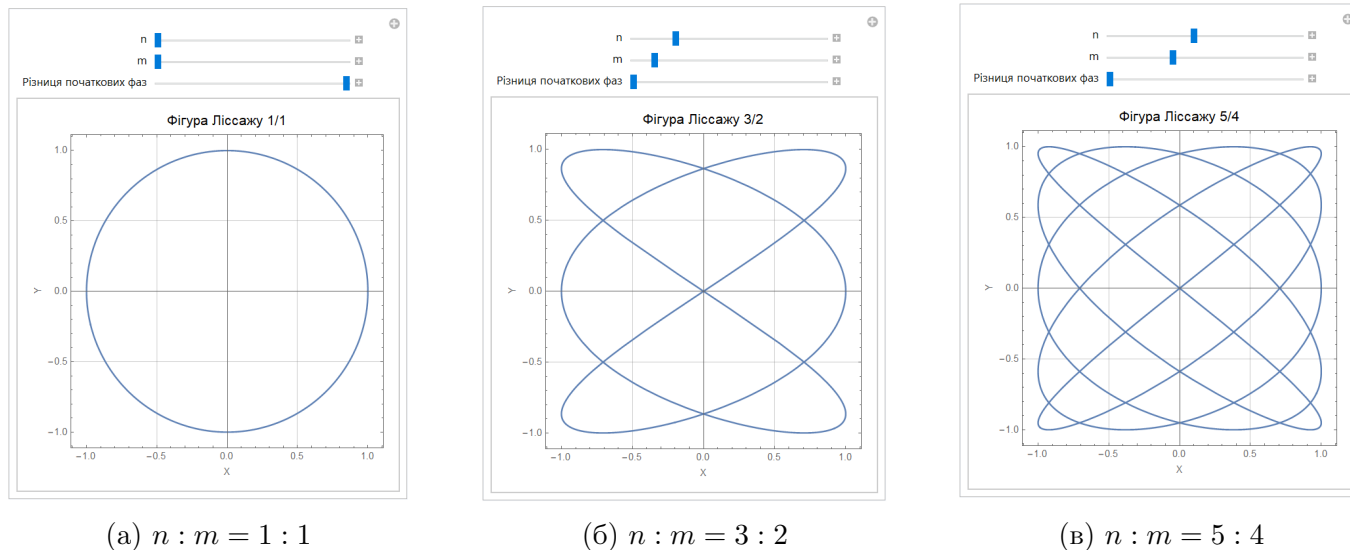


Рис. 3.11. Фігури Ліссажу при різних відношеннях частот.

Висновок

Wolfram Mathematica — потужний інструмент для аналітичних, чисельних, графічних і символічних обчислень. Володіння основами мови Wolfram дозволяє вирішувати задачі з математики, фізики, інженерії, фінансів, біології та ін. з великою гнучкістю та швидкістю.

Бібліографія

- [1] OnlineGDB – Online Compiler and Debugger [Електронний ресурс]. – Режим доступу: <https://www.onlinegdb.com/> – Назва з екрана.
- [2] Google Colaboratory [Електронний ресурс]. – Режим доступу: <https://colab.research.google.com/> – Назва з екрана.
- [3] Чорний О. П., Титюк В. К., Істоміна Н. М., Власенко В. А. Математичні методи моделювання [Навчальний посібник] – Кременчук, 2016. – 234 с.
- [4] Основи програмування мовою Python : навчальний посібник / Селіверстов Р. Г., Мельничин А. В. – Львів : ЛНУ імені Івана Франка, 2020. – 190 с. https://ami.lnu.edu.ua/wp-content/uploads/2014/02/python_book_popravlenyyuA5_maket_druk.pdf#page=2.09
- [5] Зелениця А. М. Інтерактивний україномовний підручник з Wolfram Mathematica [Електронний ресурс] / А. М. Зелениця. – Київ, 2014. – Режим доступу: <ftp://ftp.bakotech.com/>, логін: Mathematica, пароль: Wolfram. – Назва з екрана.
- [6] Анісімов А. В. Програмування Python в освітніх цілях [Навчальний посібник] / А. В. Анісімов. – Київ: Видавничо-поліграфічний центр «Київський університет», 2014. – 200 с.
- [7] Головацький В. А. Система комп'ютерної алгебри Mathematica 5: Навчальний посібник. – Чернівці: Рута, 2008. – 352 с.
- [8] Stephen Wolfram. An Elementary Introduction to the Wolfram Language. – Wolfram Media, 2017. – 325 p.
- [9] Математичне моделювання в науково-технічних дослідженнях. Моделювання у середовищі Wolfram Mathematica: навчально-методичний посібник / Укладачі: Петрик М. Р., Бойко І. В. – Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2017. – 108 с. [Електронний ресурс]. <https://core.ac.uk/download/pdf/161834216.pdf>
- [10] Шигорін П.П. Математичні обчислення в програмному пакеті Mathematica 5: Методичні рекомендації. – Луцьк: РВВ «Вежа» Волинського національного університету імені Лесі Українки, 2009. – 48 с. – Режим доступу: <https://evnuir.vnu.edu.ua/bitstream/123456789/18858/1/WolframMath.pdf>
- [11] Holovatsky V., Holovatska Y. Oscillations of an elastic pendulum. 2019. – Режим доступу: <http://demonstrations.wolfram.com/OscillationsOfAnElasticPendulum/>
- [12] Holovatsky V., Holovatskyi I., Holovatska Ya., Struk Ya. (2023). Rozrakhunok syly vzaïemodii tsylindrychnykh ta kiltsevykh mahnitiv [Oscillations of the resonant elastic pendulum]. Physics and Educational Technology, 1, 10–17, doi: <https://doi.org/10.32782/pet-2023-1-2>
- [13] Volodymyr Holovatsky, Yana Holovatska (2020), "Magnetic Field of a Hollow Cylindrical Magnet"Wolfram Demonstrations Project. demonstrations.wolfram.com/MagneticFieldOfAHollowCylindricalMagnet/

- [14] Volodymyr Holovatsky, Yana Holovatska (2020), "Interacting Cylindrical Magnets" Wolfram Demonstrations Project. demonstrations.wolfram.com/InteractingCylindricalMagnets/
- [15] Головацький, В., Головацький, І. (2022). Розрахунок сили взаємодії циліндричних магнітів. Фізика та освітні технології, (1), 3–14. <https://doi.org/10.32782/pet-2022-1-1>