

Оптимальне керування моделями соціально-економічної динаміки

у середовищі *Python*

В.М. Кирилич

О.В. Терещук

В.М. Флюд

Навчальний посібник

Львів

ЛНУ імені Івана Франка

2022

УДК 517;519.6;338

Рецензенти:

д-р. фіз.-мат. наук, проф. Бак С.М. (Вінницький державний педагогічний університет імені Михайла Коцюбинського);
д-р. фіз.-мат. наук, проф. Бандура А.І. (Івано-Франківський національний технічний університет нафти і газу);
dr. hab., prof. Sushch V. (Koszalin University of Technology).

Рекомендовано Вченою Радою

Львівського національного університету імені Івана Франка.
(Протокол № ?/? від ?? ????? 2022 року)

Кирилич В.М., Терещук О.В., Флюд В.М. Оптимальне керування моделями соціально-економічної динаміки у середовищі *Python*: Навч. посібник. – Львів: ЛНУ імені Івана Франка. 2022. – !!!*!!! с.**

Подано виклад комп'ютерної системи *Python*, основні функції та процедури, керуючі конструкції мови програмування та їхнє застосування для знаходження розв'язків задач оптимального керування, що виникають у моделях соціально-економічної динаміки. Розглянуто

Розглянуто систематичний виклад середовища *Python* і його застосування у задачах оптимального керування, математичні моделі яких описуються за допомогою задач для звичайних і частинних диференціальних рівнянь. У посібнику наведено багато прикладів і задач для самостійного опрацювання. Крім того, подано задачі, які потребують додаткового опрацювання і можуть слугувати основою студентських курсових робіт.

Для студентів математичних, економічних та інших прикладних спеціальностей.

©Кирилич В.М., Терещук О.В., Флюд В.М., 2022

ISBN !!!!!???

Зміст

Вступ	x
1 Короткий вступ до <i>Python</i>.	1
1.1. Синтаксис <i>Python</i>	2
1.1.1. Модулі і пакети	63
1.2. Короткий вступ до <i>NumPy</i>	67
1.2.1. Приклади	69
1.2.2. Друк масивів	74
1.2.3. Основні операції	76
1.2.4. Універсальні функції	83
1.2.5. Індекссування, підмасиви, ітерування	84
1.2.6. Маніпулювання формою масиву	91
1.2.7. Копіювання та вигляд	100
1.2.8. Функції та методи перегляду	106
1.2.9. Менш основне	107
1.2.10. Незвичайна індексація та трюки із індексами	107
1.2.11. Лінійна алгебра	120
1.2.12. Гістограми	125
1.2.13. Інші джерела для опрацювання	126
1.3. Короткий вступ до <i>SciPy</i>	127
1.3.1. Інтегрування і звичайні диференціальні рівняння.	127
1.3.2. Оптимізація, підгонка даних та чисельні методи розв’язування рівнянь.	152

1.4.	Короткий вступ до <i>Matplotlib</i>	168
1.4.1.	Прості графіки	168
1.4.2.	Мітки, підписи, налаштування параметрів графіків	175
1.4.3.	Графіки із спеціальними налаштуваннями параметрів.	177
1.4.4.	Графік у полярних координатах.	184
1.4.5.	Графік параметрично заданої функції.	185
1.4.6.	Побудова графіка на одному об'єкті осі.	186
1.4.7.	Декілька графіків на одному рисунку.	190
1.4.8.	Тривимірний графік у <i>Python</i>	197
1.5.	Застосування <i>Python</i> до розв'язування моделей	203
1.5.1.	Модель популяції молі spruce budworm	203
1.5.2.	Системи звичайних диференціальних рівнянь. Моделі виживання.	209
2	Математичний апарат оптимального керування	224
2.1.	Оптимізація функціоналів	224
2.2.	Абстрактна модель оптимального керування з використанням теорії систем диференціальних рівнянь	228
2.3.	Рівняння Euler'a	230
2.4.	Оптимальне керування гіперболічними системами рівнянь першого порядку з двома незалежними змінними	235
2.5.	Модель Solow як приклад математичного моделювання економічного процесу диференціальними рівняннями	239
2.6.	Завдання для самостійного опрацювання	243
3	Формулювання задачі оптимального керування	248
3.1.	Загальні поняття про керовані системи	248
3.2.	Задача керування із закріпленими кінцями	250
3.3.	Задача оптимального керування	251
3.4.	Завдання для самостійного опрацювання	253
4	Принцип максимуму	256
4.1.	Принцип максимуму для задачі із закріпленими кінцями	256

4.2.	Принцип максимуму із нескінченним обрієм керування	261
4.3.	Схема застосування принципу максимуму	263
4.4.	Задача із нефіксованим часом керування	268
4.5.	Задача оптимального керування з рухомими кінцями	272
4.6.	Економічна інтерпретація функцій спряженої системи	276
4.7.	Ілюстрація варіаційного принципу максимуму для модельного прикладу щодо гіперболічної системи двох рівнянь	279
4.8.	Рівняння Bellman'а	282
4.9.	Завдання для самостійного опрацювання	286
5	Оптимальне керування соціально-економічними моделями	291
5.1.	Формулювання математичної моделі оптимального використання енергії із врахуванням якості навколишнього середовища	291
5.2.	Однофакторна модель оптимального економічного росту	294
5.3.	Вплив на поведінку економічної системи податкових відрахувань	299
5.4.	Завдання для самостійного опрацювання	306
6	Математичні моделі біологічних спільнот	309
6.1.	Оптимальне керування чисельністю і продуктивністю популяції	310
6.2.	Задача про максимальний урожай	314
6.3.	Динаміка і керування віковою структурою популяції	319
6.4.	Задача оптимального керування з інтегральною умовою за віковим обмеженням популяції	324
6.5.	Завдання для самостійного опрацювання	330
	Бібліографія	334
	Предметний покажчик	339

Перелік ілюстрацій

1.1	Множина Mandelbrok'a	116
1.2	Гістограми.	126
1.3	Графік підінтегральної функції <code>myfunc2(x)</code>	130
1.4	Фігура, обмежена кардіоїдою $\rho = 2 + 2 \cos \theta$ та колом $\rho = 2$ (ззовні кола).	133
1.5	Експоненціальне спадання концентрації реагента у реакції першого порядку: точний розв'язок і розв'язок чисельним методом у точках, визначених методом розв'язування IVP ODE.	139
1.6	Експоненціальне спадання концентрації реагента у реакції першого порядку: точний розв'язок і розв'язок чисельним методом наперед визначених точках.	141
1.7	Дві взаємно пов'язані реакції першого порядку: чисельний і аналітичний (точний) розв'язки.	145
1.8	Генератор гармонічних коливань: чисельний і точний (аналітичний) розв'язки.	147
1.9	Система хімічних рівнянь (реакцій) Robertson'a, чисельно проінтегрована методом Radau ІА.	151
1.10	Графік функції Himmelblau.	154
1.11	Лінії рівня функції Himmelblau.	155
1.12	Графік полінома $P(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$	166
1.13	Простий лінійний (x, y) графік.	169
1.14	Проста точкова діаграма.	170

1.15	Графік функції $y = \cos^2 x$	172
1.16	Графіки функцій $y = \cos^2 x$ і $y = \cos^3 x$	174
1.17	Графіки функцій $f_n(x) = x^n \cos x$ ($n = 1, 2, 3, 4$).	177
1.18	Графіки функцій $y_1(x) = \frac{1}{\sqrt{x}}$, $y_2(x) = e^{-\sqrt{2}x}$	181
1.19	Графіки функції $y(x) = x + e^{x+\sin(20x)}$	182
1.20	Закон Мооре'а, який моделює експоненціальне зростання кількості транзисторів PCU.	184
1.21	Графік параметрично заданої функції – циклоїди.	186
1.22	Вікно графіка функції із додатковими параметрами методу <code>plt.figure</code>	188
1.23	Простий графік із двох ліній в одному об'єкті <code>Axes</code>	189
1.24	Графік циклоїди.	190
1.25	Графіки поширення тепла у двох металах.	194
1.26	Лінії рівня гіперболічного параболоїда.	195
1.27	Графік ліній рівня з підписами.	197
1.28	Графіки однієї функції, побудовані різними способами.	200
1.29	Графіки тора із різними кутами спостереження.	200
1.30	Тривимірний графік спіралі.	202
1.31	Графік $p(N)$ на проміжку $[0, 20]$ для $B = 1.5$, $A = 2$	205
1.32	Графіки $N(t)$ за різних вхідних даних: а) $r = 0.3$, $K = 5$, $A = 2$, $b = 1.5$, $T = 20$, $N(0) = 30$; б) $r = 1$, $K = 10$, $A = 3$, $b = 2$, $T = 10$, $N(0) = 50$	207
1.33	Графіки приплоду комах для $r = 0.3$, $K = 5$, $T = 20$, $N(0) = 30$, для значень $u = 0.5, 1, 2, 3$. Обчислені значення популяції комах в залежності від u	209
1.34	Графіки зміни динаміки популяцій моделі хижак-жертва.	212
1.35	Графіки зміни динаміки популяцій моделі хижак-жертва із даними з файлу <code>Model_Prey_Predator_2.txt</code>	215
1.36	Графіки станів збудження та відновлення	218
1.37	Графіки станів збудження та відновлення у випадку подразника та графік у фазовій площині.	220
1.38	Графіки станів збудження та відновлення у випадку подразника та графік у фазовій площині.	223

2.1	Існування розв'язку рівняння (2.5.4)	242
3.1	Траєкторія системи	250
4.1	Максимізація функції Hamilton'а з прикладу 4.3.1	267
4.2	Варіанти керування	268
4.3	Перехід стану системи із точки на множину.	273

Перелік таблиць

1.1	Прості арифметичні дії у <i>Python</i>	8
1.2	Пріоритети арифметичних операцій	9
1.3	Список вмонтованих функцій середовища <i>Python</i>	11
1.4	Деякі функції модуля math	14
1.5	Оператори порівняння <i>Python</i>	16
1.6	Найчастіше вживані ес-послідовності.	19
1.7	Деякі методи перетворення стрічок.	22
1.8	Деякі часто вживані методи списків у <i>Python</i>	29
1.9	Методи множини set	56
1.10	Модулі і пакети <i>Python</i>	65
1.11	Методи інтегрування ODE для <code>scipy.integrate.solve_ivp</code>	148
1.12	Опис інформації, яка міститься у словнику, який повертає метод <code>scipy.optimize.minimize</code>	157
1.13	Деякі методи оптимізації, які використовуються у методі <code>scipy.optimize.minimize</code>	159
1.14	Специфікатори місця розташування опису графіка.	176
1.15	Найбільш вживані стилі маркерів бібліотеки <i>Matplotlib</i>	178
1.16	Буквенні і стрічкові позначення кольорів <i>Matplotlib</i>	179
1.17	Стилі лінії <i>Matplotlib</i>	179
1.18	Аргументи метода <code>plt.figure</code>	187
1.19	Режими доступу до файлу.	213

Вступ

У цьому посібнику вивчається середовище *Python* для операційної системи Windows[®] та його застосування до обчислення задач і моделей математичної економіки, зокрема, оптимального керування соціально-економічними процесами, які можна описати диференціальними рівняннями. Оскільки будь-яка економічна система має визначену мету функціонування (критерій якості), а також свободу вибору, то математичним апаратом для таких систем є теорія оптимального керування (динамічна оптимізація).

У посібнику проаналізовано деякі задачі оптимального керування соціально-економічної динаміки моделями економіки, біології, теорії популяцій, природоохоронної діяльності тощо, тому зроблено детальний аналіз методів теорії оптимального керування, зокрема, огляд принципу максимуму для одержання необхідних умов оптимальності та принципу Bellman'a.

Посібник складається з вступу та шести???????????????? розділів, завдань для самостійної роботи, додатків і списку рекомендованої літератури. Для зручності читачів створено списки графіків і таблиць, які поміщені на початку посібника. Крім того, подано індексний перелік важливих термінів, які використовуються у навчальному посібнику.

Перший розділ присвячено системі *Python*, починаючи від основ програмування у програмному середовищі, спеціальних обчислень, операторів керування, побудови графіків, побудова програм і процедур.

У другому розділі розглянуто математичний апарат оптимального керування, зокрема, із використанням теорії систем диференціальних

рівнянь і рівнянь із частинними похідними та їхнім застосуванням у економічних процесах.

Третій розділ присвячений формулюванню задач оптимального керування для звичайних диференціальних рівнянь та систем.

У четвертому розділі посібника наведені основні засади принципу максимуму із застосуванням до задач оптимального керування у різноманітних формулюваннях. У цьому розділі також описані математичний алгоритм оптимального керування, принцип максимуму Понтрягіна та його застосування у багатьох конкретних задачах, наведено принцип Bellman'a, який гарантує існування достатніх умов оптимальності відповідних задач.

П'ятий розділ навчального посібника присвячений оптимальному керуванню соціально-економічними процесами. Наведено теоретичні обґрунтування існування оптимального керування, алгоритми знаходження розв'язків досліджуваних моделей.

У шостому розділі описано математичні моделі біологічних спільнот та застосування оптимального керування у задачах оптимальних популяцій

Посібник укладено так, що у кожному розділі розглянуто конкретні приклади, подано вказівки до розв'язування самостійних завдань, сформульовано завдання для самостійного опрацювання, які можуть слугувати основою для написання курсових чи магістерських робіт.

Наведено велику кількість літературних джерел, частина з яких є доступною в електронних варіантах і легко використати студентам для самостійної роботи. При розгляді конкретних тем є літературні посилання. Зазначимо, що наявність великої кількості літературних першоджерел із теорії оптимального керування соціально-економічними процесами свідчить про те, що проповані методи і підходи з використанням програмного середовища *Python* відкривають можливості для ефективного і простого розв'язування прикладних задач оптимального керування системами звичайних диференціальних рівнянь і рівнянь з частинними похідними.

У посібнику використано порозділову нумерацію формул, де перший номер (прикладів, завдань тощо) означає номер розділу, другим –

номер підрозділу, далі порядковий номер формули. Подібно пронумеровано рисунки та таблиці (першим є число, що означає номер розділу, другим – порядковий номер рисунка чи таблиці).

Навчальний посібник розрахований для студентів старших курсів бакалаврських та магістерських програм, побудований таким чином, що є лише певною канвою і потребує від учасників навчального процесу додаткового самостійного ознайомлення із багатьма математичними та природознавчими чи економічними поняттями та судженнями. Курс читали та проводили виробничі практики продовж декількох років для студентів механіко-математичного факультету, спеціалізації "Математична економіка та економетрика" Львівського національного університету імені Івана Франка. Доповнено завданнями для самостійного опрацювання, застосування *Python* для знаходження розв'язків сформульованих задач.

Автори

Розділ 1

Короткий вступ до *Python*.

Цей розділ присвячений основам програмування у середовищі *Python*. Описані основні конструкції мови та розглянуті приклади, які слугуватимуть елементами програм для розв'язування задач оптимального керування. Не претендуємо на детальне ознайомлення із *Python*, тому для детального опрацювання рекомендуємо спеціалізовані джерела із мови програмування *Python*, яку легко знайти у інтернеті (наприклад на офіційній інтернет-сторінці).

1.1. Синтаксис *Python*

Синтаксис мови програмування – набір правил, що описує комбінації символів алфавіту, які вважаються правильно структурованою програмою або її фрагментом.

Отже, синтаксис визначає як буде виглядати програма на цій мові, зокрема, як пишуться оператори, оголошення і інші мовні конструкції.

Основні правила Основні правила синтаксису мови програмування *Python* полягають у наступному

- кінець рядка є кінцем інструкції;
- в одному рядку можна поміщати процедуру – послідовність команд та інструкцій, розділених символом ”;”;
- якщо процедура чи інструкція не поміщається в одному рядку, то можна зробити у довільному місці ”перенесення” послідовності коду за допомогою символу ”\” (backslash);
- у машинному кодї *Python* застосовується ієрархія інструкцій: вкладені інструкції об’єднуються у блоки за величиною відступів. Як правило відступ становить чотири пробіли;
- вкладені інструкції в *Python* записуються відповідно до одного і того ж шаблону, коли основна інструкція завершується двокрапкою, слідом за якою розташовується вкладений блок коду, зазвичай з відступом під рядком основної інструкції.

Вище вказані синтаксичні конструкції будуть продемонстровано нижче за текстом, про що буде вказано.

Ідентифікатором називаємо послідовність символів, що використовується як ім’я елемента чи об’єкта у мові програмування. Зазвичай ідентифікатор створюється із послідовності латинських літер (розрізняються малі і великі), цифр, символа підкреслення ”_” і є унікальним.

Окрім того, ідентифікатор не може починатися із цифри. Творці кожної мови програмування рекомендують створювати такі ідентифікатори, які б відображали чи несли інформаційне навантаження про призначення елемента чи об'єкта. Тоді при аналізі машинного коду (програми) чи його відлагодження легко виконувати зміни.

Константа – це спосіб адресації даних, зміна яких програмою не передбачається а у багатьох випадках забороняється.

Константи поділяються на два види — *літерали* та *іменовані константи*. *Літерал* — стале значення певного типу даних, записане у вихідному коді комп'ютерної програми.

```
>>> 341          #числовий літерал цілого типу
>>> 9.01         #числовий літерал дійсного типу
>>> 2.718282     #числовий літерал для числа e
>>> "Python code" #рядковий літерал
```

Із наведеного вище означення одержуємо, що літерали діляться за типом і для кожного типу визначені дії та операції. Так на числових літералах можна виконувати математичні операції. Рядкові літерали беруть у апострофи, у подвійні лапки, потрібні апострофи чи потрібні подвійні лапки для того, щоб відрізнити їх від ідентифікаторів.

Іменована константа відрізняється від літералу тим, що крім сталого значення вона ще має ім'я. Таким чином, замість значення літералу можна використовувати це ім'я.

Змінна – іменована область пам'яті, ім'я якої (ідентифікатор) використовується для здійснення доступу до даних (значення змінної), що містяться у цій області пам'яті.

Значення змінної є літералом. Тип змінної визначається типом літералу, що є значенням цієї змінної.

```
>>> x=4          #x – змінна із значенням 4
>>> y=12-x       #y – змінна із значенням 8
```

Зауважимо, що змінна у *Python* є посиланням на область пам'яті, де зберігаються її дані. Розглянемо присвоєння

```
>>> a = 5
>>> b = a
```

Python інтерпретує ці два присвоєння не як дві змінні *a* і *b*, а одна змінна із значенням 5, яка має два різні ідентифікатори.

Змінні у *Python* поділяються на два види: *змінювані* (*mutable*) і *незмінювані* (*immutable*). До об'єктів змінюваних типів є безпосередній доступ для модифікації даних цих об'єктів (без створення нових). Модифікація даних об'єктів незмінюваних типів можлива лише через створення нових об'єктів (тобто через виділення пам'яті). Змінні всіх простих типів (*int*, *float*, *complex*) належать до незмінюваних.

Ключові слова мови програмування — це зарезервовані ідентифікатори, що наділені певним сенсом. Їх можна використовувати виключно відповідно до значення яке закріплене за ними у мові програмування.

Щоб одержати перелік зарезервованих слів, потрібно у командній стрічці редактора *Python* послідовно ввести

```
>>> import keyword
>>> keyword.kwlist
```

Після чого система поверне список (в залежності від редактора формат може бути відмінним від наведеного)

False	None	True	__peg_parser__
and	as	assert	async
await	break	class	continue
def	del	elif	else
except	finally	for	from
global	if	import	in
is	lambda	nonlocal	not
or	pass	raise	return
try	while	with	yield

Потрібно пам'ятати, що зарезервовані слова не можуть використо-

вуватися як ідентифікатори користувача.

Коментар – це анотація користувача стосовно об'єктів програми, які знаходяться у вихідному коді програми і створюються для кращого розуміння програми. При компіляції програми коментарі ігноруються. Також служать для створення документації програми. Коментарі є однорядкові та багаторядкові. Однорядковий коментар міститься у рядку програми, починається символом `#` і закінчується із кінцем рядка.

```
>>> l = [ 1 , 2 , 3] #визначили список
```

Багаторядковий коментар може міститися у кількох рядках і виділяється з обидвох боків потроєними подвійними лапками.

```
"""Така інформація розташовується на початку програми,  
щоб поінформувати користувачів із можливостями програми,  
ймовірно потрібними аргументами, які вимагаються для її  
правильного функціонування"""
```

Присвоєння — одна з центральних конструкцій в імперативних мовах програмування. *Присвоєння* — механізм у програмуванні, що дозволяє динамічно змінювати зв'язки об'єктів даних (наприклад, змінних) із їхніми значеннями. Синтаксис присвоєння у *Python* такий

```
>>> l=[1,2,3]
```

Тут `[1,2,3]` – визначений список, результат якого записується у змінну з ідентифікатором `x` (присвоюється змінній `x`). Після інструкції присвоєння попереднє значення змінної втрачається.

```
>>> l=[2,-3,5,2] #змінній l присвоїли обчислений список
```

```
>>> l.append(10) #до списку l додали число 10
```

```
>>> l#перевіримо список l
```

```
[2, -3, 5, 2, 10]
```

Як бачимо, список `l` змінив – доповнено на останній позиції числом 10, а первинне присвоєння втрачене.

Як кожна мова програмування, а *Python* не виняток, оперує об'єктами. Об'єкти середовища *Python* – це елементи конструкції програми, від простих до складених. Найпростішими, основними об'єктами є числа, змінні, рядки (стрічки, `string`), списки, кортежі, бібліотеки, функції (як внутрішні так і користувача), графіки, процедури, модулі тощо. Подамо коротке впровадження до кожного із основних об'єктів. Для більш детального ознайомлення можна звернутися до першоджерела за посиланням [39]

Розглянемо коротко про основні об'єкти програмного середовища *Python*.

Числа. Одним із самих простих об'єктів мови *Python* є числові значення, для яких визначені три типи: цілі числа (тип: `int`), числа із плаваючою комою (тип: `float`) і комплексні числа (тип: `complex`)

Цілі числа. Цілі числа (`integer`) – це математичні цілі числа, такі як 1, -8, 194, 3 654 451. У версії *Python 3* немає обмеження за їх величиною, є тільки обмеження за їх доступністю до оперативної пам'яті комп'ютера. Арифметика цілих чисел є точною. Для зручності дозволяється розділяти групи цифр (розряди) цілого числа символом підкреслення “_”.

```
>>> 3_456_252 + 1_987
```

```
3458239
```

Числа із плаваючою комою. Числа із плаваючою комою (`floating-point numbers`) є представленням дійсних чисел, таких як 3.5, -0.478, 167423×10^{-5} . На загал, не існує точних значень дійсних чисел у представлення *Python*, але ці числа зберігаються у бінарному (двійковому) виді із визначеною точністю (як правило, точність становить 15-16 розрядів). Через таку обмеженість точності арифметика чисел із плаваючою комою не є абсолютно точною, але при акуратному використанні вона “достатньо є точною” для більшості наукових обчислень.

Довільне число, яке містить символ крапки ".", у *Python* трактується як число із плаваючою комою. *Python* також підтримує науковий формат чисел із плаваючою комою із використанням із використанням букви *e* або *E* для розділення мантиси і показника степеня:

```
>>> 3.876397e-5 #представлення числа  $2.87628 \times 10^{-5}$ 
>>> 2_768_104e-9 #представлення числа  $2\,768\,104 \times 10^{-9}$ 
```

Останній код демонструє можливість розділення розрядів у науковому форматі числа.

Комплексні числа. Формат комплексного числа у *Python* має вигляд

```
3 + 4j
```

Через *j* позначається уявна одиниця. Зауважимо, що поодинокий символ *j* не є зарезервованим, тобто, *Python* трактує *j* як змінну із ідентифікатором *j*. Якщо хочемо використати комплексне число із дійсною частиною 0 і уявною 1, то у *Python* потрібно впровадити *1j*, без відступу між 1 і *j*. Зауважимо, що саме так зобразили наведений вище приклад комплексного числа $3+4j$ – без відступу між уявною частиною комплексного числа 4 та уявною одиницею *j*.

Дійсна та уявна частина комплексного числа у *Python* трактується як числа із плаваючою комою, хоч можуть бути записані цілими числами. Тому арифметика комплексних чисел не є точною, але забезпечує точність як і всі числа типу *float*.

Комплексне число у *Python* може бути представлене у арифметичному вигляді (приклад вище), або за допомогою пари чисел, дійсної і уявної частини, у форматі:

```
complex(3,4)
```

Прості арифметичні дії До трьох основних типів чисел, описаних у попередньому підрозділі, можна застосовувати оператори, наведені у

таблиці 1.1. Це бінарні оператори, так як вони визначені для двох чисел (операнди) і повертають третє число. У мові *Python* визначені два типи операції ділення: ділення чисел із плаваючою комою (*/*) завжди повертає число із плаваючою комою, навіть коли операндами виступають цілі числа. Цілочисельне ділення (*//*) завжди заокруглює результат із недостатчею (у меншу сторону). Типом результату є число *int*, якщо операнди також є типу *int*, у противному випадку повертає число типу *float*. Нижче наведені приклади дії цих операторів.

Табл. 1.1: Прості арифметичні дії у *Python*.

+	додавання
-	віднімання
*	множення
/	ділення із плаваючою комою
//	цілочисельне ділення
%	ділення за модулем (обчислення залишку від ділення)
**	піднесення до степеня

```
>>> 4.71 / 3
```

```
1.57
```

```
>>> 21/6
```

```
3.5
```

```
>>> 15/3
```

```
5.0
```

```
>>> 15//5
```

```
3
```

```
>>> 21//6
```

```
3
```

```
>>> 4.71/3
```

```
1.0
>>> 15%3
0
>>> 15%6
3
>>> 4.71%3
1.71
```

Пріоритет операторів Математичні операції можуть бути об'єднані у послідовності.

**	найвищий пріоритет
*, /, //, %	
+, -	найнижчий пріоритет

Табл. 1.2: Пріоритети арифметичних операцій

Тому природно постає питання пріоритету їх виконання. Наприклад, при обчислення виразу $21-3*5$ результатом є число 6 ($21-15$) чи 90 ($18*5$)? Із таблиці 1.2 випливає, що правильний результат виконання зазначених дій є число 6, оскільки операція множення має вищий пріоритет ніж віднімання, тому повинна виконуватися першою. Правила пріоритету можна змінити за допомогою дужок, наприклад, $(21-3)*5=90$. Оператори із однаковими пріоритетами виконуються зліва на право, за винятком операції піднесення до степеня (**), яка виконується справа наліво (або зверху до низу у рядковому записі виразу).

```
>>> 9/3/2 # рівнозначно 3/2
1.5
>>> 9/(3/2) # рівнозначно 9/1.5
6.0
>>> 2**3**2 # рівнозначно 2**(3**2)=2**9
```

```
512
>>> (2**3)**2 # рівнозначно 8**2
64
```

Методи і атрибути числових значень. Я було значено вище, числові значення у мові *Python* є об'єктами (насправді у *Python* все є об'єктами) і мають визначені *атрибути* (attributes), звернення до яких виконуємо у форматі

```
<object>.<attribut>
```

тобто, звертання до атрибуту об'єкта здійснюється за допомогою крапки ".". Деякі атрибути є простими значеннями, наприклад, об'єкти комплексних чисел мають атрибути `real` і `imag`, які відповідають дійсній і уявній частині комплексного числа:

```
>>> (4-3j).real
4.0
>>> (4-3j).imag
-3.0
```

Іншими атрибутами є **методи** (methods): це функції, які відповідним чином опрацьовують об'єкти. Наприклад, для комплексних чисел існує метод `conjugate`, який повертає комплексно-спряжене число:

```
>>> (3+4j).conjugate()
(3-4j)
```

Тут пара пустих круглих дужок означає властивість метода, тобто виконання обчислення спряженого комплексного числа для числа (у наведеному прикладі) `3+4j`. Якщо ж круглі дужки не вказано, тобто `(3+4j).conjugate`, то буде повернуто посилання на сам метод (без його виклику) – тому що сам також метод є об'єктом.

Табл. 1.3: Список вмонтованих функцій середовища *Python*.

A abs() aiter() all() any() anext() ascii()	E enumerate() eval() exec()	L len() list() locals()	R range() repr() reversed() round()
B bin() bool() breakpoint() bytearray() bytes()	F filter() float() format() frozenset()	M map() max() memoryview() min()	S set() setattr() slice() sorted() staticmethod() str() sum() super()
C callable() chr() classmethod() compile() complex()	G getattr() globals()	N next()	T tuple() type()
D delattr() dict() dir() divmod()	H hasattr() hash() help() hex()	O object() oct() open() ord()	V vars()
	I id() input() int() isinstance() issubclass() iter()	P pow() print() property()	Z zip() __import__()

Математичні функції. Список всіх вмонтованих (built-in) функцій інтерпретатора *Python* наведені у [таблиці 1.3](#) [40]. Серед всіх вмонтова-

них функцій *Python* розглянемо дві математичні функції – `abs` і `round`.

Функція `abs` повертає модуль числа:

```
>>> abs(-4.75)
4.75
>>> abs(-12)
12
>>> abs(3-4j)
5.0
```

Як слідує з наведених прикладів, якщо число є дійсне, то функція повертає невід’ємне число без врахування знаку. Якщо ж число є комплексне, то функція `abs` повертає модуль комплексного числа.

Функція `round` (із одним аргументом) округлює число з плаваючою комою до найближчого цілого числа за методом банкіра (Banker’s rounding):¹

```
>>> round(-8.57)
-9
>>> round(7.5)
8
>>> 8.5
8
```

Можна використовувати функцію `round` з двома аргументами: перший аргумент – число, що заокруглюється, другий – кількість точних розрядів після десяткової коми:

```
>>> round(2.718281828459045, 4)
2.7183
>>> round(3_474.987, -2)
3500.0
```

¹При заокругленні за методом банкіра число із 0.5 завжди округлюється до найближчого парного цілого числа

Python представляє собою найвищою мірою модульною мовою: функціональність мови є доступною у пакетах і модулях, які за потреби імпортуються, але за замовчуванням не завантажуються із запуском середовища. Це дозволяє економити пам'ять, яка потрібна для виконання *Python*-програми, зберігаючи її розмір мінімальним і тим самим збільшуючи продуктивність. Наприклад, багато корисних математичних функцій містяться у модулі `math`, який імпортується командою:

```
>>> import math
```

У модуль `math` входять операції для роботи із чисел із плаваючою комою і цілими числами (для роботи із комплексними числами служить модуль `cmath`). Для роботи функції із модуля `math` потрібно викликати функцію і у круглих дужках вписати число (або декілька чисел – аргументи функції). Наприклад:

```
>>> import math
>>> math.log(math.e) # натуральний логарифм числа e
1.0
>>> math.tanh(-1) # тангенс гіперболічний від -1
-0.7615941559557649
>>> math.sin(math.pi/2) #  $\sin \frac{\pi}{2}$ 
1.0
```

Повний список модуля `math` можна знайти у документації на інтернет-сторінці [41]. Найбільш вживані математичні функції наведені у таблиці 1.4.

Крім того, у модулі `math` поміщені два важливі нефункціональні атрибути: `math.pi` і `math.e` – дві математичні константи π і число e (основа натурального логарифма).

Табл. 1.4: Деякі функції модуля `math`.

<code>math.sqrt(x)</code>	\sqrt{x}
<code>math.exp(x)</code>	e^x
<code>math.log(x)</code>	$\ln x$
<code>math.log(x, b)</code>	$\log_b x$
<code>math.log10(x)</code>	$\lg x$
<code>math.sin(x)</code>	$\sin x$
<code>math.cos(x)</code>	$\cos x$
<code>math.tan(x)</code>	$\operatorname{tg} x$
<code>math.asin(x)</code>	$\arcsin x$
<code>math.acos(x)</code>	$\arccos x$
<code>math.atan(x)</code>	$\operatorname{arctg} x$
<code>math.sinh(x)</code>	$\operatorname{sh} x$
<code>math.cosh(x)</code>	$\operatorname{ch} x$
<code>math.tanh(x)</code>	$\operatorname{th} x$
<code>math.asinh(x)</code>	$\operatorname{arsh} x$
<code>math.acosh(x)</code>	$\operatorname{arch} x$
<code>math.atanh(x)</code>	$\operatorname{arth} x$
<code>math.hypot(x, y)</code>	Euclidean відстань $\sqrt{x^2 + y^2}$
<code>math.factorial(x)</code>	факторіал невід'ємного цілого числа $x!$
<code>math.erf(x)</code>	функція помилок за значенням x
<code>math.gamma(x)</code>	гамма-функція за x , $\Gamma(x)$
<code>math.degrees(x)</code>	перетворення x із радіан у градуси
<code>math.radians(x)</code>	перетворення x із градусів у радіани
<code>math.isclose(a, b)</code>	перевірка рівності a і b у межах допустимого відхилення

Модуль у *Python* можна завантажувати (імпортувати) на два способи:

- імпорт модуля із прямим доступом до всіх його функцій:

```
from <module> import *
```

- імпорт модуля, надаючи йому нове ім'я (ідентифікатор):

```
import <module> as <another_name>
```

Перший спосіб імпорту модуля є зручним при роботі у командній стрічці *Python*, але не рекомендується вживання такого завантаження у програмах. Виникає небезпека конфлікту імен (у тому випадку, коли імпортуються функції із декількох модулів), а тому важко розпізнати з якого модуля використовується функція. Імпорт командою `import math` зберігає зв'язок функцій із простором імен (namespace) відповідного модуля, а отже, щоб вписати інструкцію `math.cos` потрібно натиснути більше кнопок клавіатури, але у такий спосіб вихідний машинний код є зручнішим для розуміння.

За необхідності можна імпортувати не весь модуль, а тільки потрібні функції. При цьому, інші функції модуля будуть недоступними:

```
from <module> import <function_name>
```

Продемонструємо це на прикладах.

```
>>> from math import cos # завантажили із модуля тільки cos x
```

```
>>> cos(math.pi) # намагаємося обчислити cos  $\pi$ 
```

```
Traceback (most recent call last):
```

```
File «interactive input» line 1, in <module>
```

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

```
>>> cos(pi) # намагаємося обчислити cos  $\pi$ 
```

```
Traceback (most recent call last):
  File «interactive input» line 1, in <module>
NameError: name 'pi' is not defined
```

У наведеному прикладі намагалися обчислити $\cos \pi$, після того, як із модуля `math` імпортували тільки функцію. Але ж число π також належить до модуля, тому звертаючись до π як до сталої модуля `math` безпосередньо (`cos(pi)`) і вказуючи на модуль (`cos(math.pi)`), одержуємо повідомлення про помилку: такі імена не визначені (`name 'pi' is not defined`). Ще відзначимо, що імпортуючи тільки одну функцію, посилання на цю функцію здійснюємо безпосередньо.

Операції порівняння і логічні операції.

Оператори. Основні оператори, які використовуються у *Python* для порівняння об'єктів, наведені у таблиці 1.5. Після виконання оператора порівняння *Python* повертає логічний (boolean) об'єкт який може містити одне із двох можливих значень: `True` або `False`. Ці вмонтовані постійні ключові слова, яким не можуть бути присвоєні інші значення (зарезервовані слова).

<code>==</code>	перевірка рівності
<code>!=</code>	не дорівнює
<code>></code>	більше
<code><</code>	менше
<code>>=</code>	не менше
<code><=</code>	не більше

Табл. 1.5: Оператори порівняння *Python*.

Звернемо увагу на відмінність між оператором `==` перевірки рівності і оператором `=` присвоєння. Оператор присвоєння (assignment) повертає значення. Наприклад, операція `x=5` присвоює ім'я змінної `x` цілочисельному об'єкту `5`. Вираз `x==5` перевіряє умову рівності: якщо `x` на присвоєно значення `5`, то повертає `True`, в протилежному випадку `False`, тобто повертає результат порівняння. Крім того, як вказувалося

вище, обчислення із плаваючою комою проводяться у *Python* набли-

жено, тому треба обережно проводити порівняння, щоб не одержати помилковий результат:

```
>>> x = 0.01
>>> y = 0.1**2
>>> x == y
False
```

Але, наприклад

```
>>> u = 0.25
>>> v = 0.5**2
>>> u == v
True
```

Об'єкти *Python* (продовження). Стрічка.

Визначення об'єкта стрічка У *Python* об'єкт, який представляє стрічку (тип `str`), це впорядкована незмінна послідовність символів. Для визначення змінної, яка містить постійний текст (стрічковий літерал – string literal), потрібно помістити цей текст в одинарні чи подвійні лапки:

```
>>> inf="Функція приймає додатні значення"
>>> result='Одержано додатне значення визначника'
```

Над стрічками можна виконувати операцію об'єднання (конкатенація – concatenate) за допомогою оператора "+", або, що то само, розміщати послідовно їх у одній стрічці

```
>>> 'abc' + 'def'
'abcdef'
>>> 'one ' 'two' ' three'
'one two three'
```

У *Python* немає обмежень на довжину стрічки, тому стрічковий літерал можна визначити в одному блоці тексту, поміщеного у лапки. Але для зручності читання краще створювати у програмі стрічки, довжина якого знаходиться у межах тексту на екрані монітора (довжина тексту не перевищує 80 символів). Для розміщення текстової стрічки у двох більше рядках коду використовується символ продовження стрічки `"\"` (backslash), або помістити стрічковий літерал у круглі дужки:

```
>>> long_string = 'We hold these truths to be self -evident , '\
... ' that all men are created equal...'\
>>> long_string = ('We hold these truths to be self -evident , '\
... ' that all men are created equal...')
```

У цьому прикладі визначена змінна `long_string`, яка містить один рядок тексту (без символів переходу у новий рядок). Операція об'єднання не вставляє відступи, тому при необхідності потрібно їх включати в текст явно.

Якщо стрічка містить символи, які повторюються (один або кілька), то для визначення такої стрічки можна скористатися оператором `***` для об'єднання кратних груп символів:

```
>>> 'x Y z '*3
'x Y z x Y z x Y z '
```

Внутрішня функція `str` виконує перетворення типу об'єкта, який передається як аргумент, у стрічку разом із набором правил, притаманним вихідному об'єктові:

```
>>> str(54)
'54'
>>> str(3.45e4)
'34500.0'
>>> str(3.45e16)
'3.45e+16'
```

Escape-послідовності. Символ лапок, які використовуються у машинному коді для визначення стрічки (str), можна включати у сам текст стрічкового літералу і визначається цей символ так:

```
>>> some_txt='The name of this file is \"working\" '
'The name of this file is "working" '
```

\'	одинарна лапка
\"	подвійна лапка
\n	перехід на новий рядок
\r	повернення каретки
\t	горизонтальна табуляція
\b	повернення на одну позицію

Табл. 1.6: Найчастіше вживані esc-послідовності.

Інші можливості керування зображенням стрічкового літерала та включення символів у стрічковий літерал наведені у таблиці 1.6. Їх ще називають escape-послідовностями (або esc-послідовності) і визначаються за допомогою символа \ (backslash).

```
>>> numb='one \n \t two \n \t \t three'
>>> print(num)
one
  two
   three
```

Індексування і вирізання стрічок. Індексування (indexing, або subscripting) стрічки повертає один символ стрічки за заданої позиції. Як і всі послідовності у *Python*, стрічки є проіндексовані, тобто, кожному символу поставлена у відповідність індекс, який визначає місце цього символу у загальній послідовності. У *Python*, що важливо пам'ятати на відміну від більшості інших мов програмування, індексація починається від 0. А значить, останній символ стрічки, яка складається із n символів, має присвоєний індекс n-1. Наприклад:

```
>>> lng='Python'
>>> lng[0]
'P'
>>> lng[4]
'o'
```

Невід'ємний індекс веде відлік від початку стрічки до кінця. Від'ємний – у протилежному напрямку, від кінця до початку:

```
>>> lng[-1]
'n'
>>> lng[-6]
>>> 'P'
```

Вирізання (slicing) стрічки `s[i:j]` повертає підстрічку (фрагмент вихідної стрічки), яка розташована між символами із двома заданими індексами, включаючи першим і символ і закінчуючи `j-1` символом. Тобто, в результат вирізання `s[i:j]` одержуємо фрагмент вихідної стрічки `s` довжиною `j-i` символів (у випадку додатних `i, j`). Якщо не задається перший індекс `s[j]`, то за замовчуванням індексування починається від `0`, тобто це рівнозначно `s[0:j]`. Аналогічно, якщо відсутній другий індекс `s[i:]`, то підстрічка вирізається до кінця, починаючи із символа з індексом `i`. Наприклад:

```
>>> lng[1:4]
'yth'
>>> lng[:3]
'Pyt'
>>> lng[3:]
'hon'
>>> lng[:]
'Python'
```

Третій необов'язковий параметр операції вирізання підстрічки визначає крок вирізання (*stride*). Якщо крок вирізання не заданий, то за замовчуванням від дорівнює одиниці і тоді операція вирізання фрагменту стрічки повертає символи в указаному діапазоні. Якщо ж задається крок вирізання *k*, то операція повертає кожний *k* символ. Від'ємне значення *k* змінює порядок символів у підстрічці на протилежний. Наприклад:

```
>>> snt='Language Python'
```

```
>>> snt[2:15:3]
```

```
'na tn'
```

```
>>> snt[::]
```

```
'Language Python'
```

```
>>> snt[-1:-12:-2]
```

```
'nhy gu'
```

За допомогою операції вирізання легко одержати зручний спосіб зміни порядку символів у стрічці на протилежний, яке ще називають реверсуванням:

```
>>> snt[::-1]
```

```
'nohtyP egaugnaL'
```

Методи обробки стрічок. У *Python* стрічки є незмінюваними об'єктами, тому після присвоєння стрічку змінити не можна. Але можна формувати нові стрічки із існуючих, але тільки як нові об'єкти:

```
>>> snt='Language Python'
```

```
>>> a='Programming '
```

```
>>> a+= snt
```

```
>>> a
```

```
'Programming Language Python'
```

Щоб довідатися кількість символів у стрічці, використовується

вмонтований метод `len()`

```
>>> a='Programming '
>>> len(a)
11
```

Для стрічкових об'єктів у *Python* існує багато методів обробки і перетворення. Наведемо найчастіше вживаних із них. Кожний із наведених

Табл. 1.7: Деякі методи перетворення стрічок.

Метод	Опис
<code>center(<i>width</i>)</code>	повертає нову відцентровану стрічку із кількістю символів <i>width</i> ;
<code>endswith(<i>suffix</i>)</code>	повертає <code>True</code> , якщо стрічка закінчується підстрічкою <i>suffix</i> ;
<code>startswith(<i>prefix</i>)</code>	повертає <code>True</code> , якщо стрічка починається підстрічкою <i>prefix</i> ;
<code>index(<i>substring</i>)</code>	повертає найменший індекс у стрічці, який відповідає фрагменту <i>substring</i> ;
<code>lstrip(<i>[chars]</i>)</code>	повертає копію стрічки із видаленими початковими символами, заданими необов'язковим аргументом <i>[chars]</i> . Якщо аргумент <i>[chars]</i> не заданий, то видаляються всі початкові пробіли;
<code>rstrip(<i>[chars]</i>)</code>	повертає копію стрічки із видаленими кінцевими символами, заданими необов'язковим аргументом <i>[chars]</i> . Якщо аргумент <i>[chars]</i> не заданий, то видаляються всі кінцеві пробіли;
<code>strip(<i>[chars]</i>)</code>	повертає копію стрічки із видаленими початковими і кінцевими символами, заданими необов'язковим аргументом <i>[chars]</i> . Якщо аргумент <i>[chars]</i> не заданий, то видаляються всі початкові і кінцеві пробіли;

Метод	Опис
<code>upper()</code>	повертає копію стрічки, в якій всі символи переведені у верхній регістр;
<code>lower()</code>	повертає копію стрічки, в якій всі символи переведені у нижній регістр;
<code>title()</code>	повертає копію стрічки, в якій всі слова починаються із прописних букв (букв верхнього регістра), а все інші символи переведені у нижній регістр;
<code>replace(<i>old</i>, <i>new</i>)</code>	повертає копію стрічки, в якій кожна підстрічка <i>old</i> замінена на підстрічку <i>new</i> ;
<code>split([<i>sep</i>])</code>	повертає список підстрічок, які розділені заданою стрічкою <i>sep</i> . Якщо стрічка <i>sep</i> не задана, то розділювачем є люба кількість пробілів;
<code>join([<i>list</i>])</code>	використовує стрічку як розділювач при списку <i>list</i> стрічок;
<code>isalpha()</code>	повертає <code>True</code> , якщо всі символи у непустій стрічці є літерами і <code>False</code> , якщо інакше;
<code>isdigit()</code>	повертає <code>True</code> , якщо всі символи у непустій стрічці є цифрами і <code>False</code> , якщо інакше.

методів у таблиці 1.7 повертає нову стрічку, то методи можна об'єднати у ланцюг викликів:

```
>>> s = '-+-Python Wrangling for Beginners'
>>> s.lower().replace('wrangling', 'programming').lstrip('-+')
'python programming for beginners'
```

★ **Приклад 1.1.1.** Наведемо приклади перетворення стрічок за допомогою методів із таблиці 1.7.

```
>>> a = 'java python c++ fortran '
>>> a.isalpha()
False
```

```
>>> b = a.title()
>>> b
'Java Python C++ Fortran '
>>> c = b.replace(' ', '! \n')
>>> c
'Java! \nPython! \nC++! \nFortran! \n'
>>> print(c)
Java!
Python!
C++!
Fortran!
>>> c.index('Python')
6
>>> c[6:].startswith('Py')
True
>>> c[6:12].isalpha()
True
```

Функція `print` Розглянемо тепер важливу функцію, за допомогою якою можна контролювати хід виконання програми та візуально отримати результат. *Python* дозволяє у ручному форматі одержати результат виконання програми. Для цього служить внутрішня функція *Python* `print`. У *Python* функція `print` є внутрішньою функцією (так само як, наприклад, `len` і `round`). Обов'язковим параметром є список об'єктів для виводу і два необов'язкові параметри `end` і `set`. Ці необов'язкові параметри визначають які символи використовуються для завершення кінця рядка і які символи використовуються для розділення об'єктів.

★ *Приклад 1.1.2.*

```
>>> ans = 6
>>> print('Solve:', 2, 'x =', ans, 'for x')
```

```
Solve: 2 x = 6 for x
>>> print('Solve: ', 2, 'x = ', ans, ' for x', sep="", end='! \n')
Solve: 2x = 6 for x!
>>> print('Answer: x =', ans/2)
Answer: x = 3.0
```

★ **Приклад 1.1.3.** Функцію `print` можна використати для створення простих текстових таблиць:

```
>>> heading = '| Index of Dutch Tulip Prices |'
>>> line = '+' + '-'*16 + '-'*13 + '+'
>>> print(line , heading , line ,
...       '| Nov 23 1636 | 100 |',
...       '| Nov 25 1636 | 673 |',
...       '| Feb 1 1637 | 1366 |', line , sep=' \n')
...
+-----+
|Index of Dutch Tulip Prices|
+-----+
| Nov   23  1636 | 100 |
| Nov   25  1636 | 673 |
| Feb    1  1637 | 1366 |
+-----+
```

Форматування стрічок. Користуючись методом `format` у простій формі можна вставляти об'єкти у стрічку:

```
>>> '{ } додати { } дорівнює { }'.format(2, 3, 'п \ 'ять')
"2 додати 3 дорівнює п'ять"
```

Тут метод `format` викликається із стрічкового літерала із аргументами 2, 3, 'five', які вставляються у заданому порядку на місця заміни полів, які позначені парою фігурних дужок `{ }`.

Для зручності опрацювання довгих стрічок, а також при кратного вставляння значення, поля заміни можуть бути пронумеровані або поіменовані.

★ **Приклад 1.1.4.** Нумеровані поля індексуються, починаючи від 0, і можуть бути розміщені у стрічці у довільному порядку.

```
>>> '{} додати {} дорівнює {}'.format(2, 3, 'п \ 'ять')
"2 додати 3 дорівнює п'ять"
>>> '{num1} додати {num2} дорівнює {answer}' \
...     .format(num1=2, num2=3, answer='п \ 'ять')
"2 додати 3 дорівнює п'ять"
>>> '{0} додати {0} дорівнює {1}'.format(5.1, 5.1+5.1)
'5.1 додати 5.1 дорівнює 10.2'
```

Об'єкти *Python* (продовження). Список.

Ініціалізація і індексування списків. *Python* має у своєму розпорядженні структури даних для зберігання упорядкованого списку об'єктів. Такі базові структури можуть зберігати дані різних типів, на відміну деяких мов програмування. Наприклад, у програмному середовищі C чи Fortran, структура даних може містити дані тільки одного типу. Прийнято називати структуру даних про яку мова, масивом (array).

У *Python* список (list) – це упорядкований змінюваний (mutable) масив об'єктів. Список утворюємо із заданих об'єктів, які записуємо у квадратних дужках [] і розділяємо комою. Наприклад:

```
>>> list1=[1, 2.72, 'Three', 0]
>>> list1
[1, 2.72, 'Three', 0]
>>> x = 5
>>> list2=[3, -4, x, list1, 3.24]
```

```
>>> list2
[3, -4, 5, [1, 2.72, 'Three', 0], 3.24]
```

Із наведених прикладів слідує, що списки у *Python* можуть містити посилання на об'єкти довільного типу, наприклад, на числову змінну *x*, непустий список *list1*.

Викликати елементи списку можна за індексом, який, як вказувалося раніше, у *Python* починається із 0. Причому, якщо список має кілька рівнів, то при звертанні до внутрішнього рівня потрібно викликати спочатку зовнішній рівень. Пояснимо це на прикладі визначених вище списків *list1* і *list2*. Список *list1* є дворівневий: об'єкти цього списку утворюють перший рівень. Літерали стрічки 'Three' становлять другий, внутрішній рівень. А тому, щоб викликати літерал стрічки із другого рівня, потрібно спочатку викликати саму стрічку, як об'єкт першого рівня *list1[2]*, а після цього літерали другого рівня, наприклад, *list1[2][0]*. Після чого одержимо 'T':

```
>>> list1[2]
'Three'
>>> list1[2][0]
'T'
```

Для перевірки наявності об'єкта у даному списку, використовується оператор *in*:

```
>>> -4 in list2
True
>>> 1 in list2
False
>>> 1 in list2[3]
True
```

Властивість змінності списку На відміну від стрічки, списки у *Python* є змінними (mutable) об'єктами. Це означає, що елементам

заданого списку можна присвоювати нові елементи.

```
>>> list1
[1, 2.72, 'Three', 0]
>>> list1[1]=3.14
>>> list1
[1, 3.14, 'Three', 0]
>>> list2
[3, -4, 5, [1, 3.14, 'Three', 0], 3.24]
```

Звернемо увагу, що змінивши елемент списку `list1` із індексом 1 із 2.72 на 3.14, змінився елемент у списку `list2`, у якому список `list1` входить елементом. Про це слід пам'ятати, коли виконується переприсвоєння списку.

```
>>> lst1=[1, 2, 3, 4]
>>> lst2=lst1
>>> lst2[3]='Four'
>>> slt1
[1, 2, 3, 'Four']
```

Зміна елемента у списку `lst2` викликала зміну елемента у первинному списку `lst1`. Це відбулося тому, що змінні `lst1` і `lst2` посилаються на один і той же список, який зберігається у одній локації пам'яті.

Із списків можна вирізати (slice) групу елементів аналогічно як у випадку стрічок:

```
>>> l1=[0., 0.1, 0.2, 0.3, 0.4, 0.5]
>>> l1[2:5]
[0.2, 0.3, 0.4]
>>> l1[::-1] #повертає реверсну копію списку
[0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
```

```
>>> l1[1::2] #повертає елементи списку із заданим кроком 2
[0.1, 0.3, 0.5]
```

Операція вирізання копіює елементи у новий список, незалежний від вихідного:

```
>>> l2=l1[::1] #повна копія списку l1
>>> l2
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5] #копія списку l1
>>> l2[3]=333 #переприсвоєння елемента списку l2
>>> l2
[0.0, 0.1, 0.2, 222, 0.4, 0.5] #змінилося значення елемента списку l2
>>> l1
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5] #список l1залишився незмінним
```

Методи списку Як і для стрічок, для списків і *Python* існує багато корисних методів, деякі із них наведені у таблиці 1.8. Нагадаємо, що об'єкти типу `list` є змінними, вони можуть збільшувати чи зменшувати свій розмір без необхідності копіювати вмісту у новий об'єкт, так це було у випадку стрічок.

Табл. 1.8: Деякі часто вживані методи списків у *Python*.

Метод	Опис
<code>append(<i>element</i>)</code>	додає <i>element</i> на кінець списку
<code>extend(<i>list2</i>)</code>	доповнює список елементами списку <i>list2</i>
<code>index(<i>element</i>)</code>	повертає найменший індекс списку, що містить <i>element</i>
<code>insert(<i>index</i>, <i>element</i>)</code>	вставляє у список <i>element</i> за індексом <i>index</i>
<code>pop()</code>	видаляє і повертає останній елемент списку

Метод	Опис
<code>reverse()</code>	змінює порядок елементів списку на протилежний (реверсує список)
<code>remove(element)</code>	видаляє перший за порядком <i>element</i> списку
<code>sort()</code>	сортує список
<code>copy()</code>	створює копію списку
<code>count(element)</code>	повертає кількість елементів, рівних <i>element</i> , у списку

★ **Приклад 1.1.5.** Покажемо на прикладах використання методів на списках з метою перетворення списків.

```
>>> lst_1=[1, 2, 3]
>>> lst_1.append(4)
>>> lst_1
[1, 2, 3, 4]
>>> lst_1.extend([5, 6, 7])
>>> lst_1
[1, 2, 3, 4, 5, 6, 7]
>>> lst_1.insert(3,3.5) #вставляння 3.5 за індексом 3
>>> lst_1
[1, 2, 3, 3.5, 4, 5, 6, 7]
>>> lst_1.remove(3)
>>> lst_1
[1, 2, 3.5, 4, 5, 6, 7]
>>> lst_1.index(4)
3 #повертає індекс 3 елемента 4 списку
```

Об'єкти *Python* (продовження).Кортежі. Кортеж `tuple` можна трактувати як незмінюваний список. Утворюється кортеж шляхом розташування елементів у круглих дужках. Кортежі можна індексувати і вирізати із них елементи так само як і у списках, але оскільки кортежі є незмінюваними, то не можна додавати, виділяти елементи, розширювати кортежі. Але, якщо елементом кортежа є список, то елемента списку можна переприсвоювати нові значення. Продемонструємо сказане на прикладах.

```
>>> tp_1=(1, 'two', 3.)
>>> tp_1[2]
3
>>> tp_2=(1, ['x', 'y', 'z'], 2)
>>> tp_2[1][1]='U'
>>> tp_2
(1, ['x', 'U', 'z'], 2)
```

Пустий кортеж утворюється за допомогою пари пустих круглих дужок, `emt=()`. Але створення кортежа, який містить тільки один елемент недостатньо помістити цей елемент у круглі дужки, бо у цьому випадку програма потрактує таку конструкцію як, наприклад, зміну пріоритету виконання дій. Тому при створенні одноелементного кортежа потрібно після елемента у круглих дужках вписати кому, `sngt=('elem',)`.

Зупинимося на питанні використання кортежів. При простому створенні кортежа у *Python* допустимим є присвоєння послідовності елементів, розділених комами, без круглих дужок (синтаксис кортежа):

```
>>> tpl_1=1, 2, 3
>>> tpl_1
(1, 2, 3)
```

Таке використання утворення називають упакування кортежа (`tuple packing`). Протилежна дія – розпакування кортежа (`tuple unpacking`) –

це спосіб присвоєння значень декільком змінним у одному рядку:

```
>>> x, y, z=tpl_1
>>> z
3
```

Причому, кількість присвоєнь повинна бути така сама як кількість елементів кортежа. Наприклад, якщо треба виконати присвоєння другого і третього елементів кортежа `tpl_1` змінним `y` і `z`, то можна скористатися викликаним фрагментом кортежа, що містить ці елементи:

```
>>> u, v = tpl_1[1:3]
>>> u
2
>>> v
3
```

При виконанні операції присвоєння вираз справа від оператора "=" обчислюється першочергово. Це забезпечує зручний спосіб обміну значеннями між двома змінним, використовуючи кортежі:

```
>>> u, v = tpl_1[1:3]
>>> u
2
>>> v
3
>>> u, v = v, u
>>> u
3
>>> v
4
```

Цикли for Досить часто при написанні машинного коду програми виникає потреба у переборі елементів (по одному) ітерованого об'єкта та виконання визначених дій із кожним окремим елементом по черзі. У *Python* такий алгоритм здійснюється за допомогою зручного і приро-

дного способу у такий спосіб:

```
>>> for item in iterable_object:
```

При використанні цього способу кожний об'єкт по чергово передається на виконання до наступного блоку коду.

```
>>> auto_list=['Renault', 'Toyota', 'BMW', 'Mercedes', 'Ford']
```

```
>>> for auto in auto_list:
...     print(auto)
```

```
...
Renault
Toyota
BMW
Mercedes
Ford
```

Кожний елемент із списку `auto_list` по чергово викликається і присвоюється змінній циклу `auto` для дальшої передачі у блок інструкцій, який слідує після двокрапки ":" і згідно синтаксисом кожна інструкція у цьому блоці коду обов'язково повинна бути зміщена вправо на однакову кількість відступів (як вказувалося раніше за замовчуванням і за рекомендацією на 4 відступи). Цикли можуть бути вкладеними. Тоді блок коду внутрішнього циклу потрібно зробити відступ стосовно зовнішнього коду циклу на 4 пробіли, тобто, стосовно самого циклу на 8 відступів:

```
>>> for auto in auto_list:
...     for letter in auto:
...         print(letter, end=' ')
...     print()
... 
```

```
R_e_n_a_u_l_t_
T_o_y_o_t_a_
B_M_W_
M_e_r_c_e_d_e_s_
F_o_r_d_
```

Тип `range` *Python* має у своєму розпорядженні ефективний метод посилення на послідовність числових значень, які утворюють арифметичну прогресію: $a_n = a_0 + nd$, $n = 0, 1, 2$

ldots. У самому простому випадку по суті потрібний цілочисельний лічильник із кроком 1 і починається із 0 (нагадаємо, що індексація у *Python* починається від 0). Зрозуміло, що можна у *Python* сконструювати процедуру створення такого списку. Але для більшості випадків потрібний великий список, що проводить до великих затрат копірок пам'яті для збереження кожного цілочисельного значення. У *Python* для створення арифметичних прогресій з метою проходження ітеративних кроків служить тип `range`. Об'єкт типу `range` можна створити за допомогою від одного до трьох аргументів, які визначають початкове значення, кінцеве значення та крок (який може бути від'ємний), у форматі:

```
>>> range([a0=0],n,[strade=1])
```

Вищенаведений формат конструктора `range` означає, що у випадку відсутності аргумента `a0`, який задає початкове значення, за замовчуванням приймається 0. Крок `strade=1` також не є обов'язковим і при його відсутності крок за замовчуванням приймається рівний 1. Наведемо кілька прикладів:

```
>>> a = range(6) #0, 1, 2, 3, 4, 5
```

```
>>> b = range(1, 5) #1, 2, 3, 4
```

```
>>> c = range(0, 7, 3) #0, 3, 6
```

```
>>> d = range(8, 1, -2) #8, 6, 4, 2
```

У *Python* об'єкт, який створюється за допомогою інструкції `range`, не

є списком, але є ітерованим об'єктом, за допомогою якого можна створювати цілі числа за запитом: об'єкти типу `range` можна індексувати, перетворювати у списки і кортежі і за їхньою допомогою виконувати ітераційні процедури:

```
>>> list(b)
[1, 2, 3, 4]
>>> b[1:3]
range(2, 4)
>>> list(b[1:3])
[2, 3]
>>> for k in d:
...     print(d)
...
8
6
4
2
```

Метод `enumerate`. Оскільки об'єкт типу `range` використовується, зокрема, для генерування цілочисельної послідовності, то виникає спокуса використати цю послідовність для створення індексів списку списків і кортежів при ітерації у циклі `for`, тобто:

```
>>> auto_list=('Renault', 'Toyota', 'BMW', 'Mercedes', 'Ford')
>>> for i in range(len(auto_list)):
...     print("{}: {}".format(i, auto_list[i]))
...
0: Renault
1: Toyota
2: BMW
3: Mercedes
4: Ford
```

Ця процедура працює, але має певні недоліки: створюється послідовність за допомогою `range` та використовується внутрішня функція `len`. В той же час, що вся необхідна інформація міститься у первинному основному списку (у нашому прикладі це `auto_list`). З цією метою у *Python* є метод `enumerate`, вхідним аргументом для якого є ітерований об'єкт і створює для кожного елемента кортеж (`count, item`), який містить лічильник індекса і самого елемента. Тепер за допомогою методу `enumerate` попереднє завдання можна оформити більш вишуканіше:

```
>>> for i, aut in enumerate(auto_list, 1):
...     print("%d: %s"%(i, aut))
...
1: Renault
2: Toyota
3: BMW
4: Mercedes
5: Ford
```

Звернемо увагу, що у останній процедурі вжили іншого формату друку стрічки із вставленими змінними значеннями і застосовано природну індексацію списку (яка починається від 1). Для цього у методі `enumerate` використали другий параметр, який вказує на початок відліку (за його відсутності значення за замовчуванням рівний 0).

Функція `zip`. Якщо маємо на меті здійснити ітераційний прохід за двома списками, то для цього у *Python* передбачена вмонтована функція `zip`, яка створює об'єкт-ітератор, у якому кожний елемент представляє кортеж пар відповідних елементів двох списків:

```
>>> x = [1, 2, 3, 4]
>>> y = ['A', 'B', 'C', 'D']
>>> zip(x, y)
<zip object at 0x00000171AC578F80>
>>> list(zip(x, y)) #перетворення у список
```

```
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')]
```

```
>>> for par in zip(x, y):  
...     print(par)
```

```
...  
(1, 'A')  
(2, 'B')  
(3, 'C')  
(4, 'D')
```

За допомогою функції `zip` можна розділяти послідовності пар кортежів на окремі списки. Покажемо це на прикладі. Спочатку утворимо список кортежів, яка містять пари індекс та елементи списку `auto_list` за допомогою перетворення методу `enumerate` у список, а після розділимо на два списки: список індексів і список `auto_list`:

```
>>> auto_list=('Renault', 'Toyota', 'BMW', 'Mercedes', 'Ford')
```

```
>>> aut_lst=list(enumerate(auto_list))
```

```
>>> aut_lst
```

```
[(0, 'Renault'), (1, 'Toyota'), (2, 'BMW'), (3, 'Mercedes'), (4, 'Ford')]
```

```
>>> A, B = zip(*aut_lst) #роз'єднали на два кортежі
```

```
>>> A #кортеж індексів (перші елементи пар списку aut_lst)
```

```
(0, 1, 2, 3, 4)
```

```
>>> B #кортеж назв авто (другі елементи пар списку aut_lst)
```

```
('Renault', 'Toyota', 'BMW', 'Mercedes', 'Ford')
```

Керування потоком виконання. Тільки нечисленні комп'ютерні програми виконуються виключно лінійно, тобто інструкції виконуються по черговому у тому порядку, в якому вона=и записані у вихідному коді. Але дуже часто при виконанні програми потрібно вибрати об'єкти даних і блоки коду в залежності від умов, опираючись на оперативні перевірки. Тому всі мови програмування застосовують різні варіанти

конструкції `if-then-(else)`. У цьому параграфі коротко пишемо синтаксис *Python*-версію даної конструкції, а також розглянемо ще один тип циклу – `while`.

Конструкція `if...elif...else`. Конструкція `if...elif...else` дозволяє виконувати інструкції у залежності від виконання заданих умов, тобто у залежності від результату однієї або декількох перевірок логічних виразів, які при обчисленні повертають логічні значення `True` або `False`:

```
if <логічний вираз 1>:  
    <блок інструкцій 1>  
elif <логічний вираз 2>:  
    <блок інструкцій 2>  
...  
else:  
    <блок інструкцій>
```

Ця конструкція працює так: якщо при обчисленні *<логічного виразу 1>* повертається результат `True`, то виконується *<блок інструкцій 1>*, інакше, якщо при обчисленні *<логічного виразу 2>* повертається результат `True`, то виконується *<блок інструкцій 2>* і т.д.. Якщо жодне із обчислених *<логічних виразів>* не повертає результату `True`, то виконується *<блок інструкцій>*, який слідує за ключовим словом `else`.

★ **Приклад 1.1.6.** Продемонструємо на прикладі застосування умовного оператора `if`. Утворити два списки, парних і непарних чисел із чисел від 0 до 11:

```
>>> odd_number=[] #пустий список для непарних чисел  
>>> even_number=[] #пустий список для парних чисел  
>>> for num in range(0, 12):  
...     if num % 2==0: #число парне
```

```
...         even_number.append(num) #до списку парних чисел
...         else: #якщо не є парним (є непарним)
...             odd_number.append(num) #до списку непарних чисел
...
>>> print("Odd number list: { }; \nEven number list: { }. \n"
          .format(odd_number, even_number))
Even list: [0, 2, 4, 6, 8, 10];
Odd list: [1, 3, 5, 7, 9, 11].
```

Цикл while. У циклі `for` встановлюється фіксована конкретна кількість ітерацій, тоді як в інструкції у блоці циклу `while` виконуються до тих, поки виконується деяка задана умова:

```
>>> str1 = "The world!"
>>> it = 0
>>> while it<len(l1):
...     print(str1[it], end='/')
...     it+=1
...
T/h/e/ /w/o/r/d/
```

У наведеному вище прикладі впроваджуємо стрічку `l1` (довільної наперед невідомої довжини). Визначаємо початкове значення змінної ітерування `it = 0`, за допомогою якої будемо контролювати кількість виконуваних дій (інструкція `print`). Завдання полягає у тому, щоб символи довільного тексту (довільно наперед невідомої довжини) розділити символом `"/"` (slash). Почергово викликаємо символи стрічки `str1[it]`, розділяємо символом `"/"`, друкуємо, і на кінець, у процедурі циклу збільшуємо на 1 змінну ітерування (`it+=1`). Після виконаного циклу робимо перевірку, чи значення змінної ітерування `it` не перевищує довжини кількості символів у заданій стрічці. Якщо умова не виконується – цикл закінчується. У результаті одержуємо надруковану перетворену згідно завдання стрічку.

Додаткові засоби керування потоком виконання: `break`, `continue`, `pass`, `else`. *Python* має у своєму розпорядженні три додаткові команди для керування потоком виконання програми.

Команда `break`. Команда `break`, яка міститься у тілі циклу, негайно завершує виконання цього циклу і передає керування інструкціям, які слідують за циклом. Наведемо приклад, записаний у вигляді процедури у редакторі GUI (Graphic User Interface).

★ *Приклад 1.1.7.*

```
x = 0
while True:
    x += 1
    if not (x % 10 or x % 16):
        break
print(x, 'is divisible by both 10 and 16.')
```

У цьому прикладі умова циклу `while` завжди дорівнює значенню `True`, тому вихід із циклу здійснюється за допомогою команди переривання циклу `break`. А виконання `break` відбудеться тоді, коли значення умови `if` буде `True`, тобто, значення умови `(x % 10 or x % 16)` буде рівне 0, що у булевих змінних означає `False`, а заперечення `not` поверне власне `True`. Тоді керування процедурою перейде до виконання тіла умовного оператора `if`, тобто виконання команди `break`, яка безумовно завершить цикл `while`. Згідно із машинним кодом, програма надрукує таке перше найменше значення `x`, яке без залишку ділиться на 10 і 16:

```
80 is divisible by both 10 and 16
```

Звернемо увагу, що у вище наведеній процедурі замість булевої змінної `True` можна записати `1`, і у цьому випадку *Python* інтерпретує це як `True`. У неявному вигляді така заміна використана в умові `if`, оскільки число `x` ділиться на 10 і 16, коли залишок рівний 0. Значення 0 в умові *Python* інтерпретує як значення булевої змінної `False`.

★ *Приклад 1.1.8.* Аналогічний наступний приклад: задано спи-

сок чисел, серед яких є від'ємні. Потрібно визначити індекс першого від'ємного числа у списку. Для цього згенеруємо список псевдовипадкових чисел, використавши при цьому наступний формат (код програми запишемо не у командній стрічці, а у редакторі у вигляді процедури):

```
from random import randint
lst=[randint(-3, 8) for i in range(9)]
print('L=' .format(lst))
L=[1, 7, 2, 1, 5, 6, 8, -3, -2] #згенерований список
```

Функція `randint(a, b)` із модуля `random` повертає псевдовипадкове число із проміжка $[a, b]$. Записана процедура `lst` повертає список псевдовипадкових чисел (зверніть увагу на формат процедури) і за кожною компіляцією та на різних комп'ютерах одержимо різні списки. Тут використано формування списку, щоб спростити впровадження такого списку. Можна замінити впровадженням безпосередньо.

```
for k, nmb in enumerate(lst):
    if nmb<0:
        break
print('First negative number { } in list L occurs index { }'.format(nmb, k))
First negative number -3 in list L occurs index 7.
```

Команда `continue`. Команда `continue` працює аналогічно команді `break`, але із точністю до навпаки. Якщо `break` негайно і безумовно виконує вихід із блоку циклу, то `continue` негайно і безумовно керує програмою на початок цього циклу без завершення блоку ітерацій для поточної ітерації.

★ **Приклад 1.1.9.** Із заданого списку чисел виділити парні. Згенеруємо список цілих чисел із проміжка $[-10, 10]$ і запишемо процедуру у такому вигляді:

```
from random import randint
nmblst = [randint(-10, 10) for k in range(7)]
print('L1={}'.format(nmblst))

evnlst=[]
for nmb in nmblst:
    if (nmb % 2):
        continue
    evnlst.append(nmb)
print('The even numbers list L={}'.format(evnlst))
```

```
L1=[-5, 7, -9, 4, 3, -6, -5].
The even numbers list L=[4, -6].
```

Команда pass. Команда `pass` не виконує жодної дії. Вона корисна як "корок" у машинному коді, який поки що ненаписаний, але у цьому місці синтаксис вимагає команду *Python*, узгоджену із відповідними відступами (тобто тіло підпрограми, яка займає нижчий рівень ієрархії).

```
>>> for k in range(13):
...     if k == 5:
...         pass #тут буде вписаний код, що відповідає i=5
...     if not (k % 4):
...         print(k, 'is divisible by 4;')
... 
```

```
0 is divisible by 4;
4 is divisible by 4;
8 is divisible by 4;
12 is divisible by 4;
```

Команда else. За тілом циклу `for` чи `while` може слідувати блок інструкцій `else`, який буде виконуватися тільки у тому випадку, якщо цикл завершиться бер втручання команди `break`. Для циклу `for` це означає,

що інструкції блоку `else` будуть виконуватися тільки тоді, коли виконуються до кінця всі ітерації `for`. Для циклу `while` – коли умова циклу `while` прийме значення `False`. Повернемося до [прикладу](#) знаходження індекса першого від'ємного числа у списку. Розглянемо ситуацію, коли список не містить від'ємних чисел. Отримаємо досить дивний результат:

```
from random import randint
lst=[randint(1, 10) for i in range(9)]
print('L=' .format(lst))
```

```
L=[6, 1, 4, 3, 6, 1, 7, 6, 3] #згенерований список
```

```
for k, nmb in enumerate(lst):
    if nmb<0:
        break
print('First negative number { } in list L occurs index { }'.format(nmb, k))
```

```
First negative number 3 in list L occurs index 8.
```

Переконуємося, що повертається індекс останнього числа, але яке не є від'ємним. Щоб виправити цю помилку (власне помилки немає, якщо вважати що перевіряється список у якому точно є від'ємне число. Але список може бути довільний, без від'ємних чисел, і це потрібно врахувати), потрібно дослідити як завершився цикл `for`. Якщо у результаті виконання циклу були перевірені всі елементи списку, то команда `break` не виконувалась, тоді модифікуємо програму так:

```
from random import randint
lst=[randint(3,10) for i in range(9)]
print('L={ }'.format(lst))
for k, nmb in enumerate(lst):
    if nmb < 0:
        print('First negative number in list L occurs index .'.format(nmb, k))
        break
else:
    print('No negative numbers in the list.')
```

```
L=[8, 8, 6, 4, 10, 10, 4, 4, 9]
No negative numbers in the list.
```

Пропонуємо читачеві самостійно переконатися, що остання програма повертає правильний результат у випадку, коли список містить щонайменше одне від’ємне число.

Функції користувача у *Python*. У *Python* функція (function) – це набір інструкцій, згрупованих в єдиний блок, якому присвоєний ідентифікатор (ім’я), завдяки чому можна багатократно звертатися до функції і виконувати приписані їй перетворення і операції. Використання функцій при програмуванні дає дві переваги. По-перше, функції дають змогу використовувати багатократно фрагмент коду не копіюючи його у різні частини програми, де він використовується. По-друге, функції дозволяють розділяти складні задачі на окремі процедури, кожна із яких реалізується за допомогою власної функції. Здебільшого набагато простіше окремо написати код для кожної процедури, аніж кодувати у цілому програму.

Визначення та виклик функції. Синтаксис визначення функції у *Python* такий:

```
def function_name(arg1,arg2,...):
    result = function_body
    return result
```

Тобто, функція визначається за допомогою ключового слова `def`, присвоюється ім’я функції і при необхідності задається список аргументів (може бути випадок відсутності аргументів), які функція приймає при зверненні. Інструкція у тілі функції записується у блоці із зміщенням вправо стосовно ключового слова `def`. Якщо при виконання цього блоку зустрічається команда `return`, то повертається обчислене значення. Наприклад, визначимо функцію двох аргументів, яка повертає їхнє середнє значення:

```
>>> def aver_arith(a,b):  
...     rs = (a + b)/2  
...     return rs  
...
```

Тепер використаємо цю функцію для обчислення середнього значення двох чисел:

```
>>> aver_arith(3,6.4)  
4.7
```

Якщо є потреба повернути два і більше значення функції, то відповідно ці обчислені значення у блоці `function_body` поміщаємо у рядку `return` у вигляді кортежу. Як приклад розглянемо визначення функції, яка повертає середнє арифметичне і середнє геометричне значення двох чисел. Причому, маючи вже визначену функцію обчислення середнього арифметичного, використаємо її при визначенні такої функції:

```
>>> import math  
  
>>> def aver2_ar_gm(x, y):  
...     ar = aver_arith(x, y)  
...     gm = math.sqrt(x*y)  
...     return ar, gm  
...  
  
>>> aver2_ar_gm(3, 4)  
(3.5, 5.0)
```

Визначення функцій можуть бути розташовані у довільному місці програми, але не можна викликати функцію до її визначення. Також функції можуть бути вкладеними, але слід пам'ятати, що функція, визначена всередині іншої, є недоступною за межами зовнішньої функції. Тобто, перевизначимо функцію, яка повертає середнє арифметичне та середнє геометричне значення двох чисел так, що внутрішніми блоками, які обчислюють ці два значення, є внутрішні функції:

```
>>> def avr2ar_2gm(a, b):
...     def avr2ar(a ,b):
...         return (a + b)/2
...     def avr2gm(a ,b):
...         return math.sqrt(a**2 + b**2)
...     return avr2ar(a ,b), def avr2gm(a ,b)
...
>>> avr2ar_2gm(4, 3)
(3.5, 5.0)
```

Але якщо спробуємо скористатися тепер функцією `avr2gm`, яку визначили всередині функції `avr2ar_2gm`, то програма поверне інформацію про помилку:

```
>>> avr2gm(4, 3)
Traceback (most recent call last):
  File «interactive input» line 1, in <module>
NameError: name 'avr2gm' is not defined
```

Стрічка документації `docstring`. Функція `docstring` – це стрічковий літерал, який надає першу інструкцію у визначенні функції. Якщо функція є простою, цей літерал записується як текст у потрібних лапках у одній стрічці. У випадку складної функції ця інформація міститься у початковій однорядковій анотації і подальшим більш детальним описом інформації щодо даної функції. Наприклад, визначимо функцію, яка за коефіцієнтами квадратного тричлена повертає його корені (випадок додатного дискримінанта), і другим рядком у визначенні помістимо інформацію про функцію:

```
import math
```

```
def roots(a, b, c):
    """Return the roots of  $ax^2 + bx + c$ ."""
    d = b**2 - 4*a*c
    r1 = (-b + math.sqrt(d)) / (2*a)
    r2 = (-b - math.sqrt(d)) / (2*a)
    return r1, r2
```

Обчислимо корені квадратного рівняння $2x^2 - 5x + 2 = 0$, вписавши у командний рядок побудовану функцію із аргументами, після чого одержимо:

```
>>> roots(2, -5, 2)
(2.0, 0.5)
```

Рядок документування тепер є спеціальним атрибутом `__doc__` функції:

```
>>> roots.__doc__
'Return the roots of  $ax^2 + bx + c$ .'
```

Тобто, рядок документації `docstring` повинен містити детальну інформацію про те, як користатися даною функцією, які аргументи їй передаються та які значення вона повертає.

Рекурсивна функція. Функція, яка може викликати сама себе, називається *рекурсивною функцією* (*recursive function*). Рекурсія не завжди потрібна, проте у деяких випадках дає змогу будувати витончені алгоритми. Наприклад, один із способів обчислення факторіала цілого числа $n \geq 1$ є визначення такої рекурсивної функції, тобто функції, у визначенні якої вона звертається сама до себе:

```
def fctrl(n):
    if n == 1:
        return 1
    return n*fctrl(n-1)
```

```
>>> fctrl(6)
720
```

Тут

при звертанні до функції `fctrl(n)` виконується виклик `n` разів після виклику функції `fctrl(n-1)`, при якому виконується звертання `n-1` разів після виклику `fctrl(n-2)`, і т.д. до того моменту, поки `fctrl(1)` не поверне `1` за замовчуванням. При реалізації подібних рекурсивних алгоритмів потрібно особливо увагу звернути на забезпеченні умови зупинки при виконанні деякої конкретної умови.

Об'єкти *Python*: словники і множини. У *Python* словник (dictionary) – це три "асоційованого масиву". Словник може містити довільні об'єкти як значення (value), але на відміну від стрічок і кортежів, у яких елементи індексуються цілими числами, починаючи від нуля, кожний елемент словника індексується унікальним ключем (key), яким може бути довільним незмінюваним об'єктом. Отож, словник представляє собою набір пар ключ-значення (key-value). Словники є незмінюваними об'єктами.

Визначення та індексування словника. Синтаксис словника такий:

```
{key_1: value_1, key_2: value_2, ..., key_n: value_n}
```

Наприклад:

```
>>> height = {'Burj Khalifa': 828., 'One World Trade Center': 541.3, \
              'Mercury City Tower': -1., 'Q1': 323., \
              'Carlton Centre': 223., 'Gran Torre Santiago': 300., \
              'Mercury City Tower': 339.}
```

```
>>> print(height)
```

```
{'Burj Khalifa': 828.0,  
'One World Trade Center': 541.3,  
'Mercury City Tower': 339.0,  
'Q1': 323.0,  
'Carlton Centre': 223.0,  
'Gran Torre Santiago': 300.0}
```

Команда `print(height)` повертає словник у тому ж форматі, тобто у фігурних дужках. Якщо один і той самий ключ приписаний до різних значень (ключ `'Mercury City Tower'` у наведеному вище прикладі), то зберігається тільки останнє значення, важливо пам'ятати, що ключі у словнику не повинні повторюватися.

Окремє значення можна викликати за його ключем, який грає роль індекса, або за літеральним значенням (`'Q1'`), або за допомогою змінної, значення якої дорівнює ключу:

```
>>> height['One World Trade Center']  
541.3  
>>> building = 'Carlton Centre'  
>>> height[building]  
223.0
```

Значення у словнику можна присвоювати за індексом:

```
>>> height['Empire State Building'] = 381.  
>>> height['The Shard'] = 306.
```

Інший спосіб визначення словника – за допомогою конструктора `dict`, або, іншими словами, зміна типу об'єкта *Python* списку чи кортежу пар кортежів (`key, value`) на словник. Якщо ключами служать прості стрічки (які можуть бути використані як імена змінних), то пари можна також визначати як іменовані аргументи для конструктора `dict`:

```
>>> ordinal = dict(((1, 'First'), (2, 'Second'), (3, 'Third')))  
>>> ordinal  
{1: 'First', 2: 'Second', 3: 'Third'}
```

```
>>> mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
>>> mass
{'Earth': 5.972e+24, 'Mercury': 3.301e+23, 'Venus': 4.867e+24}
>>> ordinal[2] #тут 2 є ключем, не індексом
'Second'
>>> mass['Earth']
5.972e+24
```

Ітерація у циклі `for` повертає ключі словника у порядку їх розташування:

```
>>> for ky in ordinal:
...     print(ky, ': ', ordinal[ky], ';')
...
1: First;
2: Second;
3: Third;
```

Методи словника.

Метод `get`. Зауважимо, що звертання до словника за неіснуючим індексом приводить до помилки. Але можна скористатися зручним методом `get()` для виклику значення, задаючи ключ, якщо він існує, або деяке значення за замовчуванням, якщо ключ не існує. Якщо значення за замовчуванням не задано, то повертається спеціальне значення `None`. Наприклад:

```
>>> print(mass.get('Pluto'))
None
>>> mass.get('Pluto', -1)
-1
```

Методи `keys`, `values`, `items`. Три методи `keys`, `values`, `items` повертають відповідно ключі, значення і пари ключі-значення (у вигляді кор-

тежів) словника. Наприклад:

```
>>> planets=mass.keys()
>>> planets
dict_keys(['Mercury', 'Venus', 'Earth'])

>>> for planet in planets:
...     print(planet, ':', mass[planet],';')
...
Mercury : 3.301e+23 ;
Venus : 4.867e+24 ;
Earth : 5.972e+24 ;
```

За допомогою об'єкту `dict_keys` можна виконувати ітераційні виклики довільну кількість разів, але треба пам'ятати, що це не є список а, отже, елементи цього об'єкту не підлягають індексуванню і операції заміни змінних. Якщо виникає потреба у списку ключів чи значень словника, то потрібно виконати зміну типу об'єкта за допомогою конструктора `list`:

```
>>> mass
{'Earth': 5.972e+24, 'Mercury': 3.301e+23, 'Venus': 4.867e+24}
>>> planets_list=list(mass.keys())
>>> planets_list
['Mercury', 'Venus', 'Earth']
>>> planets_list[1]
'Venus'
```

Існують аналогічні методи для виклику значень і елементів (пара ключ-значення) словника: повертається об'єкти `dict_values` і `dict_items`. Наприклад:

```
>>> mass.items()
dict_items([('Mercury', 3.301e+23), ('Venus', 4.867e+24), ('Earth', 5.972e+24)])
```

```
>>> for planets_data in mass.items():
...     print(planets_data)
...
('Mercury', 3.301e+23)
('Venus', 4.867e+24)
('Earth', 5.972e+24)

>>> mass_planets_list=list(mass.values())

>>> mass_planets_list
[3.301e+23, 4.867e+24, 5.972e+24]

>>> mass_planets_list[1]
4.867e+24
```

Іменовані аргументи. У підрозділі, у якому впровадили поняття **функції користувача у *Python***, розглянули основний синтаксис переказування аргументів у функції. При цьому припускали, що користувач завжди повинен знати, які аргументи потрібно передаватися функції і вказати їх при означенні функції.

Розглянемо випадок, коли наперед невідомо про кількість аргументів функції. *Python* має у своєму розпорядженні кілька зручних функціональних можливостей, які дають змогу реалізувати згадану ситуацію. Тобто, користувачеві необов'язково знати кількість аргументів функції. При включені `*args` (після всіх "формально визначених" аргументів) кожний додатковий позиційний аргумент поміщається у кортеж `args`. Продемонструємо це на прикладі. У підрозділі **Функції користувача у *Python*** визначено функцію `aver_arith`, яка повертає середнє арифметичне двох чисел. Розглянемо приклад.

★ **Приклад 1.1.10.** Визначити функцію, яка повертає середнє значення довільної, наперед невідомої кількості числових значень.

```
def avAr(x,y,*args):
    n=len(args)
    s=0
    for k in args:
        s=s+k
    return (x+y+s)/(n+2)
```

Тут x і y є обов'язковими аргументами, а кортеж `args` може бути довільної довжини і є необов'язковим. Тому у командному рядку виконаємо обчислення:

```
>>> avAr(4, 6)
5.0
>>> avAr(2, 4, 6, 5, 7, 9)
5.5
```

Множини. Множина `set` – це неупорядкований набір елементів, які не повторюються. Множину зручно використовувати для видалення кратних елементів із послідовності і для визначення об'єднання (`union`), перетину і різниці між двома наборами елементів. Оскільки, за означенням, елементи множини неупорядковані, об'єкти типу `set` не можна індексувати і не дозволяється виконувати операцію вирізання (`slice`), але можна проводити ітераційний перелік за множиною та перевіряти наявність елементів. Множини підтримують функцію `len`. Утворюється об'єкт `set` за допомогою переліку його елементів у фігурних дужках `{...}` або при передачі ітерованого об'єкту у конструктор `set`:

```
>>> st=set((4, 3, 2, 3, 2, 4, 7, 9, 'word')) #за допомогою кортежа
>>> st
{2, 3, 4, 'word', 7, 9}
>>> len(st) #потужність множини
6
>>> 2 in st, 6 not in st#перевірка на наявність елементів
(True, False)
```

```
>>> for item in st:
...     print(item, end='; ')
...
2; 3; 4; word; 7; 9;
```

Метод `add` множини `set` додає елементи до множини. Для видалення елементів існує декілька методів:

- `remove` – видаляє вказаний елемент, але генерує виняток `KeyError`, якщо елемент є відсутній у множині;
- `discard()` – виконує ту ж дію, але не генерує винятку;
- `pop` – видаляє (і повертає) і повертає елемент множини;
- `clear` – видаляє всі елементи.

```
>>> st={2.3, -3.2, 4}
>>> st
{2.3, 4, -3.2}
>>> st.add(2)
>>> st.add(-3)
>>> st.add(4.0)
>>> st
{2.3, 2, 4, -3.2, -3}
>>> st.remove(2)
>>> st
{2.3, 4, -3.2, -3}
>>> st.discard(3) #ніяких дій, відсутній елемент 3
>>> st
{2.3, 4, -3.2, -3}
>>> st.pop()
2.3
```

```
>>> st
{4, -3.2, -3}
>>> st.clear()
>>> st
set()
```

Об'єкти `set` володіють широким набором методів, які відповідають властивостям математичних множин. Найбільше уживані методи перераховані у [таблиці 1.9](#).

Існують дві форми для більшості виразів із використанням множин `set`: синтаксис операторів вимагає, щоб всі аргументи (операнди) були об'єктами типу `set`, тоді як явне звертання до методів виконують перетворення довільного ітерованого аргумента у множину `set`. Наведемо декілька прикладів.

```
>>> A = set([1, 2, 3])
>>> B = set((1, 2, 3, 4))
>>> A <= B
True
>>> A.issubset((1, 2, 3, 4)) #кортеж (1, 2, 3, 4) перетворює у множину
True
>>> C, D = set((3, 4, 5, 6)), set((7, 8, 9))
>>> B | C #об'єднання множин
{1, 2, 3, 4, 5, 6}
>>> A | C | D #об'єднання трьох множин
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> A & C #перетин множин
{3}
>>> C & D #порожня множина
set()
```

Табл. 1.9: Методи множини `set`.

Метод	Опис
<code>isdisjoint(<i>other</i>)</code>	множина <i>set</i> не перетинається із множиною <i>other</i> ;
<code>issubset(<i>other</i>), <i>set</i> <= <i>other</i></code>	множина <i>set</i> є підмножиною <i>other</i> ?
<code><i>set</i> < <i>other</i></code>	множина <i>set</i> є строгою підмножиною <i>other</i> ?
<code>issuperset(<i>other</i>), <i>set</i> >= <i>other</i></code>	множина <i>other</i> є підмножиною <i>set</i> ?
<code><i>set</i> > <i>other</i></code>	множина <i>other</i> є строгою підмножиною <i>set</i> ?
<code>union(<i>other</i>), <i>set</i> <i>other</i> ...</code>	об'єднання множин <i>set</i> і <i>other</i> (можливо декілька множин);
<code>intersection(<i>other</i>), <i>set</i> & <i>other</i> &...</code>	перетин множин <i>set</i> і <i>other</i> (можливо декілька множин);
<code>difference(<i>other</i>), <i>set</i> - <i>other</i> - ...</code>	різниця множин <i>set</i> і <i>other</i> (можливо декілька множин);
<code>symmetric_difference(<i>other</i>), <i>set</i> ^ <i>other</i> ^ ...</code>	симетрична різниця множин <i>set</i> і <i>other</i> (можливо декілька множин).

```
>>> C.isdisjoint(D)
```

```
True
```

```
>>> B - C #різниця множин
```

```
{1, 2}
```

```
>>> B ^ C #симетрична різниця множин
```

```
{1, 2, 5, 6}
```

Раціональні операції порівняння і присвоєння. Якщо один об'єкт потрібно присвоїти декільком змінним, то можна скористатися раціональною операцією присвоєння. Наприклад:

```
x = y = z = 3
```

Слід звернути увагу, що при такому способі присвоєння всі імена змінних вказують на той самий об'єкт, а не на його різні копії. Вище вказувалося, що декілька присвоєнь різних об'єктів можна виконувати в одному рядку за допомогою розгортання кортежа:

```
a, b, c = y + 1, 'world', -5.4
```

Кортеж справа цього виразу (у цьому випадку дужки є обов'язкові) розгортається для присвоєння його елементів іменам змінних у правій частині виразу. Цей рядок рівнозначний наступним трьом:

```
a = y+1  
b = 'world'  
c = -5.4
```

У таких виразах спочатку обчислюється права частина, потім виконуються присвоєння у лівій частині. Такий підхід дуже зручний для обміну значеннями двох змінних без використання допоміжної змінної:

```
a, b = b, a
```

Операції порівняння також можна об'єднати у ланцюжок цілком натуральним способом:

```
if a == b == 5:  
    print('a and b both equal 5')  
if 0 < x < 5:  
    print('x is between 0 and 5')
```

Python підтримує операцію умовного присвоєння: імені змінній може бути присвоєно це чи інше значення, яке залежить від результату обчислення виразу `if ... else` безпосередньо у рядку присвоєння:

```
y = (math.log(1+x))/x if x else 1
```

В останньому прикладі використано витончені можливості *Python* уникнення ділення на нуль. Проте не рекомендується використовувати цю у складніших випадках, а замінити її більш явною конструкцією, наприклад:

```
try:
    y = (math.log(1+x))/x
except ZeroDivisionError:
    y = 1
```

Генерування списку. Генератор списків у *Python* – це конструкція для створення списку на основі іншого ітерованого об’єкта в одному рядку коду. Наприклад, якщо заданий список чисел `xlist`, то список кубів цих чисел можна згенерувати так:

```
>>> xlist = list(range(1, 11))
>>> xlist
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x3list = [x**3 for x in xlist]
>>> x3list
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Зауважимо, що це швидший, ефективніший, синтаксично зручніший і зрозуміліший спосіб створення списку у порівнянні із створенням цього ж списку у блоці за допомогою циклу `for`:

```
>>> x3list = []
>>> for k in xlist:
...     x3list.append(k**2)
...
>>> x3list
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Інструкція генерації списку також може містити умовні оператори:

```
>>> x33list=[x**3 for x in xlist if x % 3]
>>> x33list
[1, 8, 64, 125, 343, 512, 1000]
```

Згенеровано список кубів чисел від 1 до 10, які не є подільними на 3. Для створення більш складнішого списку на основі заданого, то потрібно помістити конструкція `if ... else` перед циклом `for`:

```
>>> xOdd32list=[x**2 if x%2 else x**3 for x in xlist]
>>> xOdd32list
[1, 8, 9, 64, 25, 216, 49, 512, 81, 1000]
```

Згенеровано список на основі списку перших десяти натуральних чисел `xlist`, у якому непарні число піднесені до квадрату, а парні – до кубу.

Генератори списків можуть бути вкладеними. Наприклад, у прикладі нижче із двовимірного (вкладеного) списку згенеровано одновимірний список:

```
>>> tdlist = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> odlist = [el for rw in tdlist for el in rw]
>>> odlist
[1, 2, 3, 4, 5, 6, 7, 8]
```

І на закінчення, наведемо приклад генерування вкладених списків (матриць), використовую генератор випадкових чисел. Цей спосіб нам знадобиться у дальнішому, коли розглядатимемо перетворення матриць у *Python* і не суттєво буде які саме значення елементів матриці. Для цього спочатку викличемо функцію `randint(a, b)` із модуля `random`, яка повертає випадкове число на проміжку $[a, b]$. Ці випадкові числа будуть елементами матриці `mtr` розмірності $m \times n$:

```
>>> from random import randint
>>> m, n = 3, 4
```

```
>>> a, b = -5, 10
>>> mtr = [[randint(a,b) for j in range(n)] for i in range(m)]
>>> mtr
[[-4, 3, 10, -5], [4, 4, -5, -5], [-1, -4, -2, -4]]
```

Звернемо увагу, що при генеруванні списку `mtr` змінні ітерації `i` і `j` відіграють роль лічильників і не входять у функцію для обчислення елементів списку.

Лямбда-функції. Лямбда-функція (`lambda`) у *Python* – це тип простої анонімною функції. Лямбда-функція визначається за допомогою виразу (`expression`) і не може містити, наприклад, блоки циклів, умовні оператори, інструкцію `print` тощо. Лямбда-функції забезпечують обмежену підтримку парадигми програмування, відомої як ”функціональне програмування” (`functional programming`). Простий приклад визначення лямбда-функції відрізняється від звичайного способу визначення функції за допомогою `def` наведений нижче:

```
>>> func = lambda x: x**3 - 3*x + x**2 - 3
>>> func(1.)
4.0
```

Аргумент передається в `x`, а результат, обумовлений визначенням лямбда-функції після двокрапки, повертається у вигляді обчисленого значення. Для передачі декількох змінних у лямбда-функцію використовується кортеж без дужок:

```
>>> func3v = lambda u, v, w: u/v + v/w
>>> func3v(0, 4, 2)
2.0
```

У наведених вище прикладах на спостерігається особлива вигода від лямбда-функції, і зовсім вона не є анонімною, оскільки їй присвоєно ім'я `func` (`func3v`). Більш корисне застосування лямбда-виразів при створенні списків функцій. Оскільки функції у *Python* є об'єктами, тому їх можна

об'єднувати у списки. Якщо створювати список функцій за допомогою креатора `def`, то спочатку треба визначити функції, а потім помістити їх у список. Користуючись креатором лямбда, це можна виконати безпосередньо:

```
>>> funclist = [lambda x: 1, lambda x: x, lambda x: x**2, lambda x: x**3]
>>> funclist[2](4)#funclist[2] is x**2
16
>>> funclist[3](2)#funclist[3] is x**3
8
```

Генератори. Генератори – потужний інструмент мови *Python*, вони дозволяють оголосити функцію, поведінка якої є подібна до ітерованого об'єкту. Таку функцію можна використати у циклі `for`, бо вона буде генерувати за запитом по одному значенню. Такий метод є ефективніший, ніж обчислення і зберігання всіх значень, для яких повинні бути виконані ітерації (особливо коли таких значень є досить багато). Функція-генератор визначається майже так само як звичайна функція у *Python*, але замість повернення значення `return` вона містить ключове слово `yield`, яке повертає значення за кожним разом, коли вимагається при черговій ітерації. Наведемо приклад створення функції-генератора:

```
>>> def count(n):
...     i = 0
...     while i < n:
...         i += 1
...         yield i
...
>>> for j in count(5):
...     print(j, end='; ')
...
1; 2; 3; 4; 5;
```

Неможливо викликати цей генератор як звичайну функцію:

```
>>> count(5)
<generator object count at 0x000001E12EA49EB0>
```

Але, змінивши тип об'єкта, одержимо:

```
>>> list(count(5))
[1, 2, 3, 4, 5]
```

Генератор `count` працює так: він викликається при кожному звертанні до нього при кожній ітерації і на кожному кроці ітерації повертає результат (`yield`), зберігає поточне значення і до наступного звернення у циклі.

Функція `map`. Вмонтована функція `map` повертає ітератор, який задану функцію застосовує до кожного елемента послідовності, повертаючи результати так, як це виконав генератор. Наприклад, один спосіб сумування списків – визначення ітеративного застосування (`map`) функції `sum` до елементів списку:

```
>>> from random import randint
>>> mylist = [[randint(-10,10) for j in range(5)] for i in range(3)]
>>> mylist
[[1, 2, -9, 6, 2], [10, -10, -5, -4, -5], [1, 6, 1, -3, -8]]
>>> list(map(sum, mylist))
[2, -14, -3]
```

При виведенні результату застосували перетворенні у об'єкт типу `list`, оскільки функція `map` повертає об'єкт, який схожий на генератор. Ця інструкція рівнозначна генеруванню списку:

```
>>> [sum(rw) for rw in mylist]
[2, -14, -3]
```

1.1.1. Модулі і пакети

Python є модульною мовою програмування і його функціональність не обмежується основними принципами програмування. У довільному програмному середовищі доступні також розширені функціональні властивості, які викликаються за допомогою команди `import`. Ця команда створює посилання на модулі, які є звичайними *Python*-файлами, які містять означення та інструкції. Вчитавши рядок

```
import <module>
```

інтерпретатор *Python* виконує інструкції із файлу `<module>.py` і включає ім'я модуля `module` у поточний простір імен, після чого атрибути, які визначаються цим модулем, стають доступними із застосуванням синтаксису

```
<module>.<attribute>
```

Пакет (package) мови *Python* – це структурований набір модулів, розташованих у деякому каталозі файлової системи. Пакети представляють собою природний спосіб організації і поширення великих проєктів на мові *Python*. Для створення пакета файли модулів поміщають в один каталог разом із файлом `__init__.py`. Цей файл запускається при імпорті пакета і може виконувати деякі операції ініціювання і власні команди імпорту. Якщо у власному ініціюванні немає необхідності, то цей файл може бути порожнім (довжиною 0 байтів), але обов'язково повинен існувати, щоб *Python* розпізнавав такий каталог як пакет. Наприклад, пакет `NumPy` існує, як показаний нижче каталог (деякі файли і підкаталоги не показані з міркування економії місця):

```
numpy/  
  __init__.py  
  core/  
  fft/  
    __init__.py  
    fftpack.py  
    info.py
```

```
...
linalg/
  __init__.py
  linalg.py
  info.py
...
polynomial/
  __init__.py
  chebyshev.py
  hermite.py
  legendre.py
...
random/
  version.py
...
```

Тут, наприклад, `polynomial` – це вмонтований пакет у структурі паке-та `numpy`, який містить декілька модулів, зокрема `legendre` який можна імпортувати так:

```
import numpy.polynomial.legendre
```

Щоб уникнути використання розгорнутого із крапкою при звертанні до атрибутів цього модуля, зручніше користуватися такою командою:

```
from numpy.polynomial import legendre
```

У [таблиці 1.10](#) подані основні доступні модулі і пакети *Python* для програмних додатків загального призначення, а також для обчислення і наукових досліджень. Компоненти із міткою `">*` не є частиною *Python Standard Library*, тому їх потрібно встановлювати окремо, наприклад за допомогою утиліти `pip`. Деякі із них інсталиються разом із основною дистрибуцією *Python* (стандартна бібліотека *Standard Library*)², інші можна завантажити і встановити окремо. Перед тим як реалізувати який небудь власний алгоритм, потрібно переконатися, чи

²Повний список компонентів стандартної бібліотеки за посиланням [\[42\]](#)

не включена така реалізація до одного із пакетів *Python*.

Табл. 1.10: Модулі і пакети *Python*.

Модуль/пакет	Опис
os, sys	сервіси операційної системи;
math, cmath	математичні функції;
random	генератор випадкових чисел;
collections	типи даних для контейнерів, які розширюють функціональність кортежів, словників тощо;
itertools	інструменти для ефективних операторів, які розширюють функціональність простих циклів <i>Python</i> ;
fractions	арифметика раціональних чисел (правильних дробів);
re	регулярні вирази;
argparse	парсер для ключів і аргументів командного рядка;
urllib	відкриття, читання і парсинг URL;
* Django (django)	широко відоме середовище для веб-додатків;
* ryparsing	лексичний парсер для простих граматик;
pdb	налагоджувач мови <i>Python</i> ;
logging	вбудований у <i>Python</i> модуль ведення журналів;
xml, lxml	парсери мови розмітки XML;
* VPython (visual)	тривимірна візуалізація;
unittest	робоче середовище модульного тестування для систематичного проведення тестування і валідації окремих одиниць (модулів) коду;

Модуль/пакет	Опис
* NumPy (numpy)	наукові обчислення і чисельні методи;
* SciPy (scipy)	наукові обчислювальні алгоритми;
* Matplotlib (matplotlib)	графічне зображення даних;
* SymPy (sympy)	символьні обчислення;
* pandas	обробка і аналіз даних із використанням таблицьних структур даних;
* scikit-learn	machine learning (машинне навчання);

Парсер – це програма для збору і систематизації інформації, розташованої на різних сайтах. Джерелом даних може слугувати текстове наповнення, HTML-код сайту, заголовки, пункти меню, бази даних і інші елементи. Процес збору інформації називається парсингом (parsing).

Незважаючи на існування інших менеджерів пакетів³, застосунок `pip`⁴ став по суті стандартом. Застосунок `pip` зазвичай інсталується за замовчуванням при інсталюванні дистрибуції *Python* і чудово виконує роботу по керуванню версіями залежно від пакету. Повну інструкцію по інсталюванню додаткових пакетів можна знайти за посиланням [44]. На цьому завершуємо короткий вступ до синтаксису мови програмування *Python*. Автори усвідомлюють, що поза увагою залишилися багато цікавої і корисної інформації, але також свідомі того, що мета посібника – дати основи для роботи у середовищі *Python* з метою розв’язування задач оптимального керування. Зацікавлений читач за необхідності самостійно може знайти потрібну інформацію по роботі у програмному середовищі на сайтах *Python* [39, 40, 41, 42] чи у чисельній літературі, наприклад [21, 27, 28, 38, 34, 30, 33, 16, 29, 15, 5, 6].

Завдання для самостійного опрацювання.

³Наприклад, `conda` із дистрибуції *Anaconda*.

⁴Повну документацію можна знайти за посиланням [43].

1.2. Короткий вступ до *NumPy*

Основним об'єктом *NumPy* є однорідні багатовимірні масиви. Це є таблиці елементів (зазвичай чисел), одного типу, проіндексовані цілими додатними числами. У *NumPy* розмірності називаються *осями*. Число осей називається *рангом*.

Наприклад, координати точки у 3D просторі $[1, 2, 1]$ є масивом рангу 1, так як точка має тільки одну вісь. Але ця вісь має довжину 3. У наведеному нижче прикладі масив має ранг 2 (є двовимірним). Перша вимірність (вісь) має довжину 2, друга вимірність має довжину 3:

```
[[1., 0., 0.],  
 [0., 1., 2]]
```

Клас масивів *NumPy* називаються `ndarray`. Цей клас також називається `array`. Зауважимо, що `numpy.array` це не то саме, що клас `array.array` у Standard Python Library, який опрацьовує тільки одновимірні масиви і є менш функціональний. Найбільш важливими атрибутами об'єктів класу `ndarray` є наступні:

<code>ndarray.ndim</code>	число осей (розмірностей) масиву. Мовою <i>Python</i> , число розмірностей використовується як <i>ранг</i> .
<code>ndarray.shape</code>	розмірність масиву. Це є кортеж із цілих чисел, який вказує розмір масиву за кожною розмірністю. Для матриці із n рядків та m стовпців, значення <code>shape</code> дорівнює (n, m) . Тому довжина кортежу <code>shape</code> дає значення рангу (<i>rank</i>), або число розмірностей, <code>ndim</code> .
<code>ndarray.size</code>	загальне число елементів масиву, яке дорівнює добутку елементів <code>shape</code> .
<code>ndarray.dtype</code>	об'єкт, який повертає опис типу даних масиву. Його можна створити або описати тип елементів, використовуючи стандартні типи <i>Python</i> . Крім того, <i>NumPy</i> використовує власні типи, такі як, наприклад, <code>numpy.int32</code> , <code>numpy.int16</code> і <code>numpy.float64</code> .
<code>ndarray.itemsize</code>	розмір у байтах кожного елемент масиву. Наприклад, для масиву елементів типу <code>float64</code> значення <code>itemsize</code> дорівнює 8 ($= 64/8$), тоді як масив типу <code>complex32</code> його <code>itemsize</code> дорівнює 4 ($= 32/8$). Ця інструкція еквівалентна <code>ndarray.dtype.itemsize</code> .
<code>ndarray.data</code>	буфер, який містить елементи даного масиву. Зазвичай, немає необхідності використовувати цей атрибут, так як ми маємо вільний доступ до кожного елемента масиву за допомогою індексів.

1.2.1. Приклади

```
>>> from numpy import *
>>> a=arange(15).reshape(3,5)
>>> a
array([[0,1,2,3,4],
       [5,6,7,8,9],
       [10,11,12,13,14]])
>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b=array([6, 7, 8])
>>> b
array([6,7,8])
>>> type(b)
<type 'numpy.ndarray'>
```

Генерування масивів

Існує декілька способів генерування масивів. Наприклад, ми можемо створити масив із звичайного списку *Python*, використовуючи інструкцію `array`. Тип результуючого масиву продиктований типом елементів

послідовності.

```
>>> from numpy import *
```

```
>>> a=array([2,3,4])
```

```
>>> a
```

```
array([2, 3, 4])
```

```
>>> a.dtype
```

```
dtype('int32')
```

```
>>> b=array([1.2,3.5,5.1])
```

```
>>> b
```

```
array([ 1.2, 3.5, 5.1])
```

```
>>> b.dtype
```

```
dtype('float64')
```

Слід відзначити, що досить часто роблять помилку при створенні масиву за допомогою інструкції `array`, яка полягає у тому, що аргументом цієї інструкції вписують послідовність елементів, у той час, як повинен бути масив:

```
>>> a=array(1,2,3,4)           #невірно
```

```
>>> a=array([1,2,3,4])       #вірно
```

Інструкція `array` перетворює послідовність послідовностей у двовимірний масив, послідовність послідовностей послідовності у тривимірний масив, і т.д.:

```
>>> c=array([(2.7,3.5,4),(4.2,3.1,-8)])
```

```
>>> c
```

```
array([[ 2.7,  3.5,  4. ],
       [ 4.2,  3.1, -8. ]])
```

```
>>> c.dtype
```

```
dtype('float64')
```

Тип масиву можна визначити явно при створенні:

```
>>> d=array([[3,5],[7,0]],dtype=complex)
```

```
array([[ 3.+0.j,  5.+0.j],
       [ 7.+0.j,  0.+0.j]])
```

Досить часто, особливо при визначенні елементів, відома розмірність масиву, а не його елементи. Тому *NumPy* має у своєму розпорядженні декілька інструкцій, які створюють масиви відповідного розміру, елементи яких у процесі обчислення підлягають заміні. Це зводить до мінімуму необхідність створення зростаючих масивів складними операторами.

Функція `zeros` створює масив, елементами якого є нулі, функція `ones` створює масив із одиниць, і функція `empty` створює масив заповнений випадковими величинами, значення яких залежить від стану пам'яті. За замовчуванням дані масивів є типу `float64`.

```
>>> mz=zeros((4,3))
```

```
>>> mz
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

```
>>> mo=ones((2,3,4),dtype=np.int16) # тип даних визначається
                                     # при створенні масиву
```

```
>>> mo
```

```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],
       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

```
>>> me=empty((6,5))
```

```
>>> me
```

```
array([[2.69132261e-316, 2.69092577e-316, 2.69106806e-316,
        2.69092261e-316, 2.69138348e-316],
       [2.69132537e-316, 2.69133644e-316, 2.69107439e-316,
        2.69131154e-316, 2.69107123e-316],
       [2.69110917e-316, 2.69106490e-316, 2.69091629e-316,
        2.69132814e-316, 2.69131431e-316],
       [2.69138901e-316, 2.69131984e-316, 2.69131707e-316,
        2.69094158e-316, 2.69091945e-316],
       [2.69112340e-316, 2.69106174e-316, 2.69138901e-316,
        2.69106174e-316, 2.69092577e-316],
       [2.69131984e-316, 2.69091945e-316, 2.69112340e-316,
        2.70090037e-316, 2.69131707e-316]])
```

Для створення послідовності чисел у *NumPy* служить функція, аналогічна до функції `range` у *Python*, яка повертає масив у вигляді списку:

```
>>> arl=arange(2,25,3)
```

```
>>> arl
```

```
array([2,5,8,11,14,17,20,23])
```

```
>>> arlf=arange(1,5,0.4)
```

```
>>> arlf
```

```
array([1., 1.4, 1.8, 2.2, 2.6, 3., 3.4, 3.8, 4.2, 4.6])
```

Якщо `arange` використовується для створення списку чисел з рухомою комою, у загальному немає можливості наперед визначити кількість елементів списку через наближення із рухомою точкою. З цієї причини зазвичай краще скористатися функцією `linspace`, яка отримує в якості аргумента кількість очікуваних елементів, а не крок.

```
>>> linspace(1,4,13)
```

```
array([1., 1.25, 1.5, 1.75, 2., 2.25, 2.5, 2.75, 3.,
       3.25, 3.5, 3.75, 4.])
```

```
>>> mx=linspace(0,2*pi,99)
```

```
>>> mf=sin(mx)
```

Створення сітки вузлів.

Для обчислення функції багатьох змінних на решітці, яка складається із множини точок, зручно застосовувати сітки (`mesh`). До функції `np.meshgrid` передається послідовність `N` одновимірних масивів, які є координатами стосовного кожного напрямку, а функція повертає набір `N`-вимірних масивів, які формують сітку координат, в яких обчислюються значення заданої функції. Наприклад, для двовимірної сітки маємо таке:

```
>>> x = np.linspace(0, 5, 6)
```

```
>>> y = np.linspace(0, 3, 4)
```

```
>>> X, Y = np.meshgrid(x, y)
```

```
>>> X
```

```
array([[0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.]])
```

```
>>> Y
array([[0., 0., 0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1.],
       [2., 2., 2., 2., 2., 2.],
       [3., 3., 3., 3., 3., 3.]])
```

Кожний із масивів можна проіндексувати за допомогою індексів i і j : масив x повторюється як рядки, послідовно (зверху до низу) формуючи масив X , а масив y заповнює стовпці масиву Y . Отож, функцію двох змінних можна обчислювати на цій сітці як $f(X, Y)$.

Інші функції:

`array`, `zeros`, `zeros_like`, `ones`, `ones_like`, `empty`, `empty_like`,
`arange`, `linspace`, `numpy.random.rand`, `numpy.random.randn`,
`fromfunction`, `fromfile`

1.2.2. Друк масивів

При друку масиву *NumPy* відображає його у вигляді вкладеного списку, але у наступних форматах:

- останні осі друкуються зліва на право;
- другий після останнього друкуються зверху вниз;
- наступні друкуються зверху до низу із відступом на початку рядка.

Одновимірні масиви друкуються у вигляді списку, двовимірні – у вигляді матриці, тривимірні – у вигляді списку матриць.

```
>>> a=arange(6);print(a)           #1D array
```

```
[0 1 2 3 4 5]
```

```
>>> b=arange(121).reshape(4,3);print(b)       #2D array
```

```
[[ 0 1 2]
 [ 3 4 5]
 [ 6 7 8]
 [ 9 10 11]]
```

```
>>> c=floor(10*random.random((2,3,4)));print(c)      #3D
array
```

```
[[[ 8.  4.  9.  5.]
   [ 5.  7.  6.  1.]
   [ 5.  9.  5.  3.]]
 [[ 2.  9.  5.  1.]
   [ 2.  3.  2.  9.]
   [ 8.  5.  3.  2.]]]
```

Нижче ми детальніше розглянемо функцію `reshape`.

Якщо масив є занадто великий, щоб поміститися на моніторі, тоді при виведенні такого масиву друкується тільки початкові та кінцеві елементи, а внутрішні елементи опускаються і замінюються трьома крапками.

```
>>> print(arange(10000))
```

```
[ 0 1 2 ..., 9997 9998 9999]
```

```
>>> print(arange(10000).reshape(100,100))
```

```
[[ 0 1 2 ..., 97 98 99]
 [ 100 101 102 ..., 197 198 199]
 [ 200 201 202 ..., 297 298 299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

1.2.3. Основні операції

Арифметичні операції на масивах виконуються поелементно. Створюється новий масив, елементи якого є результатом дії арифметичного оператора на елементах масивів.

```
>>> from numpy import *
```

```
>>> from random import randint
```

```
>>> a=array([randint(-10,10) for k in range(4)])
```

```
>>> a
```

```
array([ 5, 7, -8, 7])
```

```
>>> b=array([randint(-10,10) for k in range(4)])
```

```
>>> b
```

```
array([ -3, 9, 4, -10])
```

```
>>> c=a-b
```

```
>>> c
```

```
array([ 8, -2, -12, 17])
```

```
>>> b**2
```

```
array([ 9, 81, 16, 100])
```

```
>>> d=10*tan(a)
```

```
>>> d
```

```
array([-33.80515006, 8.71447983, 67.99711455, 8.71447983])
```

```
>>> d>10
```

```
array([False, False, True, False], dtype=bool)
```

Як у більшості мовах програмування, у *NumPy* звичайний добуток `*` матриць виконується поелементно. Матричний добуток можна обчислити, використовуючи функцію або метод `dot`.

```
>>> mtrA1=array([[randint(-10,10) for j in range(3)]for i in
range(4)])
```

```
>>> mtrA1
```

```
array([[ -7,  1, -3],
       [-5,  7, -9],
       [-1, -2, -8],
       [-4, -4, -10]])
```

```
>>> mtrB1=array([[randint(-10,10) for j in range(3)]for i in
range(4)])
```

```
>>> mtrB1
```

```
array([[ 8, -4,  2],
       [-7,  8, -9],
       [-6, -4, -1],
       [-2,  5,  8]])
```

```
>>> print('dim A1=',shape(mtrA1),' \n dim B1=',shape(mtrB1))
```

```
dim A1=(4, 3)
```

```
dim B1=(4, 3)
```

```
>>> mtrA1*mtrB1
```

```
array([[ -56, -4, -6],
       [ 35, 56, 81],
       [  6,  8,  8],
       [  8, -20, -80]])
```

```
>>> mtrA=array([[randint(-10,20) for j in range(3)] for i
in range(4)])
```

```
>>> mtrA
```

```
array([[ -4,  8, 13],
       [ 2, 14, 15],
       [ 5,  2,  0],
       [ 4, 14, 10]])
```

```
>>> mtrB=array([[randint(-10,20) for j in range(5)] for i in
range(3)])
```

```
>>> mtrB
```

```
array([[17, 17,  4, 19, -6],
       [ 1,  4, -9, -3,  0],
       [11, 18,  0, 13,  0]])
```

```
>>> print('dim A=',shape(mtrA),' \n dim B=',shape(mtrB))
```

```
dim A=(4, 3)
dim B=(3, 5)
```

```
>>> dot(mtrA,mtrB)
```

```
array([[ 83, 198, -88, 69, 24],
       [213, 360, -118, 191, -12],
       [ 87, 93,  2, 89, -30],
       [192, 304, -110, 164, -24]])
```

```
>>> mtrA.dot(mtrB)
```

```
array([[ 83, 198, -88, 69, 24],
       [213, 360, -118, 191, -12],
       [ 87, 93,  2, 89, -30],
       [192, 304, -110, 164, -24]])
```

Деякі оператори, наприклад, такі як `''*=''` та `''+=''`, змінюють елементи існуючого масиву, не утворюючи нового.

```
>>> a=ones((3,4))
```

```
>>> a
```

```
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

```
>>> b=random.random((3,4))
```

```
>>> b
```

```
array([[ 0.20152787,  0.6274922 ,  0.16798812,  0.42958413],
       [ 0.52574655,  0.94839258,  0.7757936 ,  0.64959107],
       [ 0.70791796,  0.19341991,  0.31015702,  0.73856316]])
```

```
>>> a*=4
```

```
>>> a
```

```
array([[ 4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.]])
```

```
>>> b+=a
```

```
>>> b
```

```
array([[ 4.20152787,  4.6274922 ,  4.16798812,  4.42958413],
       [ 4.52574655,  4.94839258,  4.7757936 ,  4.64959107],
       [ 4.70791796,  4.19341991,  4.31015702,  4.73856316]])
```

Розглянемо такий приклад:

```
>>> a=ones((3,4),dtype=int)
```

```
>>> a
```

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

Слід звернути увагу на ту особливість, що при створенні матриці із одиниць у другому випадку задекларовано, що елементами матриці є цілі числа, на відміну від першого способу створення матриці **a** із одиниць. Зверніть увагу на наступну процедуру і повернуте повідомлення (матриця **b** та ж, що після останньої зміни).

```
>>> b+=a
```

```
>>> b
```

```
array([[ 5.20152787,  5.6274922 ,  5.16798812,  5.42958413],
       [ 5.52574655,  5.94839258,  5.7757936 ,  5.64959107],
       [ 5.70791796,  5.19341991,  5.31015702,  5.73856316]])
```

Але:

```
>>> a+=b
```

```
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: Cannot cast ufunc add output from dtype('float64')
to dtype('int32') with casting rule 'same_kind'
```

Це означає, що здійснювалася операція на масивах із різними типами даних, і у останньому випадку тип результату не відповідав первинному типу матриці **a**. Розглянемо наступні обчислювальні приклади.

```
>>> a=ones(3,dtype=int32)
```

```
>>> a
```

```
array([1, 1, 1])
```

```
>>> b=linspace(0,pi,3)
```

```
>>> b
```

```
array([ 0. , 1.57079633, 3.14159265])
```

```
>>> b.dtype.name
```

```
'float64'
```

```
>>> c=a+b
```

```
>>> c
```

```
array([ 1. , 2.57079633, 4.14159265])
```

```
>>> c.dtype.name
```

```
'float64'
```

```
>>> d=exp(c*1j)
```

```
>>> d
```

```
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,  
       -0.54030231-0.84147098j])
```

```
>>> d.dtype.name
```

```
'complex128'
```

Багато унарних операторів, наприклад, таких як обчислення суми елементів масиву, виконуються методами класу `ndarray`.

```
>>> a=random.random((2,3))
```

```
>>> a
```

```
array([[ 0.12923221, 0.39420352, 0.62761207],  
       [ 0.36205889, 0.97809008, 0.50282801]])
```

```
>>> a.sum()
```

```
2.9940247877505675
```

```
>>> a.min()
```

```
0.12923220935469626
```

```
>>> a.max()
```

```
0.97809008082157556
```

За замовчуванням ці оператори застосовуються до масиву як до списку чисел, незалежно від його вимірності. Однак, вказуючи `axis` як параметр, можна застосовувати ці унарні оператори до вказаних осей масиву.

```
>>> b=floor(10*random.random(12)).reshape(3,4)
```

```
>>> b
```

```
array([[ 1.,  3.,  9.,  3.],
       [ 8.,  6.,  7.,  7.],
       [ 3.,  4.,  5.,  5.]])
```

```
>>> b.sum(axis=0)      #сума елементів по стовпцях
```

```
array([ 12.,  13.,  21.,  15.] )
```

```
>>> b.sum(axis=1)     #сума елементів по рядках
```

```
array([ 16.,  28.,  17.] )
```

```
>>> b.min(axis=1)
```

```
array([ 1.,  6.,  3.] )
```

```
>>> b.cumsum(axis=1)  #кумулятивна сума елементів по
рядках
```

```
array([[ 1.,  4., 13., 16.],
       [ 8., 14., 21., 28.],
       [ 3.,  7., 12., 17.]])
```

1.2.4. Універсальні функції

Середовище *NumPy* забезпечене математичними функціями такими як, `sin`, `cos`, `exp` і т.д. Крім того, у модулі *NumPy* є так звана "універсальна функція" (`ufunc`). У самому середовищі *NumPy* ці функції здійснюють операції на масивах поелементно і повертають масиви у вигляді відповіді. Розглянемо відповідні приклади.

```
>>> from numpy import *
```

```
>>> from random import randint
```

```
>>> lstA=array([randint(0,10) for k in range(6)])
```

```
>>> lstA
```

```
array([ 6, 0, 1, 10, 9, 6])
```

```
>>> mtrA=lstA.reshape((2,3))
```

```
>>> mtrA
```

```
array([[ 6, 0, 1],
       [10, 9, 6]])
```

```
>>> power(2,lstA)
```

```
array([ 64, 1, 2, 1024, 512, 64], dtype=int32)
```

```
>>> power(lstA,2)
```

```
array([ 36, 0, 1, 100, 81, 36], dtype=int32)
```

```
>>> power(mtrA,1/2)
```

```
array([[ 2.44948974, 0. , 1. ],
       [ 3.16227766, 3. , 2.44948974]])
```

```
>>> sin(mtrA)
```

```
array([[ -0.2794155 ,  0.         ,  0.84147098],
       [ -0.54402111,  0.41211849, -0.2794155 ]])
```

```
>>> add(sin(mtrA)**2,cos(mtrA)**2)
```

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Інші функції

all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where.

1.2.5. Індексунання, підмасиви, ітерування

Одновимірні масиви можна проіндексувати, утворити підмасиви, ітерувати так само, як списки та інші послідовності *Python*. Розглянемо ці дії на прикладах.

```
>>> from numpy import *
```

```
>>> from random import randint
```

```
>>> lstA=array([randint(-20,20) for k in range(12)])
```

```
array([-20,  10,  -1,  16,  10, -18,  14, -14, -20,  -1,  7, 12])
```

```
>>> lstA[5]
```

```
-18
```

```
>>> lstA[3:7]
```

```
array([ 16,  10, -18,  14])
```

```
>>> lstB=arange(12)
```

```
>>> lstB
```

```
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

```
>>> lstB[2:9:3]      #вибірка із одновимірного масиву,  
починаючи із елемента  
                        #із індексом 2 до елемента із індексом  
9 із кроком 3
```

```
array([2, 5, 8])
```

```
>>> b=lstB**2
```

```
>>> b
```

```
array([ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121])
```

```
>>> b[:4:3]=pi
```

(заміна елементів масиву, починаючи від першого із індексом 0 до елемента із індексом 4 із кроком 3)

```
>>> b
```

```
array([ 3, 1, 4, 3, 16, 25, 36, 49, 64, 81, 100, 121])
```

(зауважимо, що відбулася заміна вказаних елементів на значення 3, а не як задекларовано, на значення π . Такий результат пояснюється тим, що тип елементів масиву є цілий, а тому результуючий масив також має цей самий тип елементів. Це пояснюється наступним прикладом)

```
>>> c=lstB**2.
```

```
>>> c
```

```
array([ 0., 1., 4., 9., 16., 25., 36., 49., 64.,  
       81., 100., 121.] )
```

```
>>> c[:8:3]=pi
```

```
>>> c
```

```
array([ 3.14159265, 1. , 4. , 3.14159265,
        16. , 25. , 3.14159265, 49. ,
        64. , 81. , 100. , 121. ])
```

*(у наступному прикладі елементи масиву **b** розташовані у зворотному порядку)*

```
>>> b[::-1]
```

```
array([121, 100, 81, 64, 49, 36, 25, 16, 3, 4, 1, 3])
```

(у наступному прикладі продемонструємо ітерацію на елементах масиву)

```
>>> for k in b:
...     print(k**(1/3.))
...
```

```
1.44224957031
1.0
1.58740105197
1.44224957031
2.51984209979
2.92401773821
3.30192724889
3.65930571002
4.0
4.32674871092
4.64158883361
4.94608744325
```

Багатовимірні масиви можуть бути проіндексовані одним індексом відносно відповідної осі. Це відображається комою у кортежі.

(розглянемо приклад створення двовимірного масиву за допомогою функції користувача. Визначимо у Python функцію)

```
>>> def myFunc(x,y):  
...     return (x+2)**y  
...
```

```
>>> a=fromfunction(myFunc, (5,4))
```

```
>>> a
```

```
array([[ 1.,  2.,  4.,  8.],  
       [ 1.,  3.,  9., 27.],  
       [ 1.,  4., 16., 64.],  
       [ 1.,  5., 25.,125.],  
       [ 1.,  6., 36.,216.]])
```

(у наступних прикладах проілюструємо звертання до визначених елементів масиву)

```
>>> a[3,2]
```

```
25.0
```

```
>>> a[0:4,2]
```

(елементи кожного рядка, розташовані у третьому стовпчику)

```
array([ 4.,  9., 16., 25.])
```

```
>>> a[:,2]
```

```
array([ 4.,  9., 16., 25.])
```

(еквівалентна команда до попередньої)

```
>>> a[1:3,:]
```

```
array([[ 1.,  3.,  9., 27.],  
       [ 1.,  4., 16., 64.]])
```

(другий та третій рядки матриці a)

Якщо кількість індексів менша від кількості осей масиву, тоді елементи, що відповідають відсутнім індексам, виписуються повністю.

```
>>> a[3]
```

```
array([ 1.,  5., 25., 125.])
```

(отримали четвертий рядок матриці a , що еквівалентно команді у наступному форматі)

```
>>> a[3,:]
```

```
array([ 1.,  5., 25., 125.])
```

Ще один приклад.

```
>>> a[:3]
```

```
array([[ 1.,  2.,  4.,  8.],
       [ 1.,  3.,  9., 27.],
       [ 1.,  4., 16., 64.]])
```

(перші три рядки матриці a . Остання команда еквівалентна наступній $a[:3,:]$)

Вираз із квадратними дужками відображає значення індекса i , що приймає всі значення вимірності осі, яку він представляє. У *NumPy* дозволяється замість індексів, які приймають всі значення, три крапки.

Три крапки (...) використовуються для того, щоб зобразити повний кортеж індексів масиву. Наприклад, якщо x є масивом рангу 5 (тобто має 5 осей), тоді використовуються наступні формати, в залежності від розглядуваного випадку, тобто, наступні вирази еквівалентні:

- $x[1,2,\dots]$ і $x[1,2,::,::,::]$,
- $x[\dots,3]$ і $x[:,::,::,::,3]$,
- $x[4,\dots,5,:]$ і $x[4,::,::,5,:]$.

Розглянемо приклади. Створимо тривимірний масив.

```
>>> def myFunc1(x,y,z):  
...     return 2*x-3**y+4**z+x*y-x*z+y*z  
...
```

```
>>> b=fromfunction(myFunc1,(2,3,4),dtype=int)
```

```
>>> b
```

```
array([[[ 0,  3, 15, 63],  
        [-2,  2, 15, 64],  
        [-8, -3, 11, 61]],  
       [[ 2,  4, 15, 62],  
        [ 1,  4, 16, 64],  
        [-4,  0, 13, 62]])
```

```
>>> b.shape
```

```
(2, 3, 4)
```

(тривимірний масив, який складається із двох двовимірних масивів)

```
>>> b[1,...]
```

```
array([[ 2,  4, 15, 62],  
       [ 1,  4, 16, 64],  
       [-4,  0, 13, 62]])
```

(це те саме, що `b[1, :, :]` або `b[1]`)

```
>>> b[... ,3]
```

```
array([[63, 64, 61],  
       [62, 64, 62]])
```

(такий результат повертає вираз `b[:, :, 3]`)

Ітерування багатовимірного масиву відбувається спочатку за пер-

шою віссю.

(із тривимірного масиву *b* створимо двовимірний конкатенацією двох двовимірних масивів першого рівня)

```
>>> cncb=concatenate((b[0],b[1]))
```

```
>>> cncb
```

```
array([[ 0,  3, 15, 63],
       [-2,  2, 15, 64],
       [-8, -3, 11, 61],
       [ 2,  4, 15, 62],
       [ 1,  4, 16, 64],
       [-4,  0, 13, 62]])
```

```
>>> for rw in cncb:
...     print(rw)
...
```

```
[ 0 3 15 63]
[-2 2 15 64]
[-8 -3 11 61]
[ 2 4 15 62]
[ 1 4 16 64]
[-4 0 13 62]
```

Однак, якщо необхідно виконати операцію на кожному елементі масиву, потрібно масив розглядати з атрибутом `flat`, який повертає ітератор, утворений елементами масиву.

```
>>> for el in b[1].flat:
...     print(el)
...
```

```
2
4
15
62
1
4
16
64
-4
0
13
62
```

Інші функції

Indexing, `arrays.indexing (reference)`, `newaxis`, `ndenumerate`, `indices`.

1.2.6. Маніпулювання формою масиву

Зміна форми масиву.

Форма масиву визначається кількістю елементів вздовж кожної осі.

```
>>> from numpy import *
```

```
>>> from random import randint
```

```
>>> a=array([[randint(-10,30) for j in range(4)] for i in
range(3)])*1.
```

```
>>> a
```

```
array([[ 23., -7., 30., 6.],
       [ 19., 20., -9., 26.],
       [ 28., 23., 19., -5.]])
```

```
>>> a.shape
```

```
(3, 4)
```

Форму масиву можна змінити за допомогою різних команд.

```
>>> a.ravel()
```

```
array([ 23., -7., 30., 6., 19., 20., -9., 26., 28., 23., 19.,  
       -5.])
```

(цей атрибут рівнозначний знищенню всіх внутрішніх рівнів)

```
>>> a.shape=(6,2)
```

```
>>> a
```

```
array([[ 23., -7.],  
       [ 30., 6.],  
       [ 19., 20.],  
       [-9., 26.],  
       [ 28., 23.],  
       [ 19., -5.]])
```

(зміна форми масиву a за допомогою атрибуту shape, якому присвоєно значення (6,2))

```
>>> a.T
```

```
array([[ 23., 30., 19., -9., 28., 19.],  
       [-7., 6., 20., 26., 23., -5.]])
```

(транспонування масиву за допомогою атрибуту T)

Порядок елементів масиву у результаті дії `ravel()` за замовчуванням є *C-style*, що означає, що найшвидше змінюється індекс справа, тобто за елементом `a[0,0]` розташований елемент `a[0,1]`. Якщо змінюється форма масиву, то масив знову трактується як *C-style*. *NumPy* зазвичай створює масиви, що зберігаються у цьому порядку, тому `ravel()`, зазвичай, не копіює його елементи, але якщо масив створений із частин

іншого масиву або створений нестандартними методами, тоді, можливо, елементи будуть копіюватися. Функції `ravel()` і `reshape` можуть при виконанні, використовуючи опційні аргументи, використовувати масиви FORTRAN-стилю, у якому найшвидше змінюються крайні зліва індекси.

Функція `reshape` повертає елементи масиву у модифікованій формі, у той час, як `ndarray.resize` змінює сам масив:

```
>>> a
```

```
array([[ 23., -7.],
       [ 30.,  6.],
       [ 19., 20.],
       [-9., 26.],
       [ 28., 23.],
       [ 19., -5.]])
```

```
>>> a.resize((2,6))
```

```
>>> a
```

```
array([[ 23., -7., 30.,  6., 19., 20.],
       [-9., 26., 28., 23., 19., -5.]])
```

Якщо аргумент розмірності оператора перетворення масиву задається значенням `-1`, то ця розмірність обчислюється автоматично:

```
>>> a.reshape(4,-1)
```

```
array([[ 23., -7., 30.],
       [  6., 19., 20.],
       [-9., 26., 28.],
       [ 23., 19., -5.]])
```

```
>>> a
```

```
array([[ 23., -7., 30.,  6., 19., 20.],
       [-9., 26., 28., 23., 19., -5.]])
```

```
>>> a.reshape(-1,4)

array([[ 23., -7., 30., 6.],
       [ 19., 20., -9., 26.],
       [ 28., 23., 19., -5.]])
```

Інші функції: `ndarray.shape`, `reshape`, `resize`, `ravel`.

Об'єднання масивів

Кілька масивів можна об'єднати вздовж різних осей:

```
>>> from numpy import *

>>> from random import randint

>>> a=floor(10*random.random((3,2)))

>>> a

array([[ 5., 3.],
       [ 4., 1.],
       [ 4., 9.]])

>>> b=floor(10*random.random((3,2)))

>>> b

array([[ 6., 8.],
       [ 0., 1.],
       [ 5., 8.]])

>>> vstack((a,b))
```

```
array([[ 5.,  3.],
       [ 4.,  1.],
       [ 4.,  9.],
       [ 6.,  8.],
       [ 0.,  1.],
       [ 5.,  8.]])
```

```
>>> hstack((a,b))
```

```
array([[ 5.,  3.,  6.,  8.],
       [ 4.,  1.,  0.,  1.],
       [ 4.,  9.,  5.,  8.]])
```

Функція `column_stack` об'єднує 1D масиви у вигляді стовпців 2D масиву. Це еквівалентно до дії `vstack()` тільки для 1D масивів:

```
>>> a
```

```
array([[ 5.,  3.],
       [ 4.,  1.],
       [ 4.,  9.]])
```

```
>>> b
```

```
array([[ 6.,  8.],
       [ 0.,  1.],
       [ 5.,  8.]])
```

```
>>> a=a.ravel()[:4]
```

```
>>> a
```

```
array([ 5.,  3.,  4.,  1.])
```

```
>>> b=b.ravel()[:4]
```

```
>>> b
```

```
array([ 6.,  8.,  0.,  1.])
```

```
>>> a[:,newaxis]
```

```
array([[ 5.],
       [ 3.],
       [ 4.],
       [ 1.]])
```

```
>>>
```

```
array([[ 5., 6.],
       [ 3., 8.],
       [ 4., 0.],
       [ 1., 1.]])
```

```
>>> vstack((a[:,newaxis],b[:,newaxis]))
```

```
array([[ 5.],
       [ 3.],
       [ 4.],
       [ 1.],
       [ 6.],
       [ 8.],
       [ 0.],
       [ 1.]])
```

```
>>> hstack((a[:,newaxis],b[:,newaxis]))
```

```
array([[ 5., 6.],
       [ 3., 8.],
       [ 4., 0.],
       [ 1., 1.]])
```

Для масивів, розмірність яких є більшою від двох, функція `hstack` складає ці масиви вздовж своїх других осей, у той час як `vstack` – вздовж свої перших осей, а `concatenate` об'єднує згідно опційних аргу-

ментів, які означають число осей, вздовж яких має місце конкатенація масивів.

(утворимо два масиви a і b)

```
>>> a=floor([10*random.random((2,3))])
```

```
>>> a
```

```
array([[ 0.,  6.,  0.],  
       [ 4.,  3.,  9.]])
```

```
>>> b=floor([10*random.random((2,3))])
```

```
>>> b
```

```
array([[ 3.,  9.,  1.],  
       [ 5.,  1.,  5.]])
```

```
>>> concatenate((a,b),axis=0)
```

```
array([[ 0.,  6.,  0.],  
       [ 4.,  3.,  9.],  
       [ 3.,  9.,  1.],  
       [ 5.,  1.,  5.]])
```

(конкатенація двох масивів: два двовимірні масиви об'єднані у тривимірний)

```
>>> concatenate((a,b),axis=1)
```

```
array([[ 0.,  6.,  0.],  
       [ 4.,  3.,  9.],  
       [ 3.,  9.,  1.],  
       [ 5.,  1.,  5.]])
```

(конкатенація двох масивів по стовпчиках)

```
>>> concatenate((a,b),axis=2)
```

```
array([[[ 0., 6., 0., 3., 9., 1.],
        [ 4., 3., 9., 5., 1., 5.]])
```

(конкатенація двох масивів по рядках)

```
>>> c=floor([10*random.random((2,4))])
```

```
>>> c
```

```
array([[[ 2., 8., 1., 9.],
        [ 8., 8., 5., 8.]])
```

```
>>> concatenate((a,c),axis=2)
```

```
array([[[ 0., 6., 0., 2., 8., 1., 9.],
        [ 4., 3., 9., 8., 8., 5., 8.]])
```

У складних випадках `r_` та `c_` використовується для створення масивів чисел розташованих вздовж однієї осі. Причому, дозволяється використовувати літерали діапазону (':'):

```
>>> r_[2,3,-4:2,.3,-.5]
```

```
array([ 2. , 3. , -4. , -3. , -2. , -1. , 0. , 1. , 0.3,
       -0.5])
```

Якщо масиви використовуються у якості аргументів, то `r_` та `c_` діють аналогічно як `vstack`, `hstack` за замовчуванням, але дозволяється вказувати за допомогою опційного аргумента номер осі, вздовж якої маємо намір провести конкатенацію.

Інші функції: `hstack`, `vstack`, `column_stack`, `concatenate`, `c_`, `r_`.

Поділ масиву на декілька менших

За допомогою функції `hsplit` можна поділити масив вздовж горизонтальних осей, або визначаючи число рівних розмірів масивів, або визначивши стовпці, після яких виконується поділ:

```
>>> a=floor(10*random.random((2,12)))
```

```
>>> a
```

```
array([[ 4.,  3.,  8.,  8.,  6.,  8.,  2.,  1.,  9.,  6.,  6.,  8.],  
       [ 1.,  2.,  5.,  9.,  8.,  4.,  8.,  9.,  1.,  6.,  8.,  2.]])
```

```
>>> hsplit(a,3)
```

```
[array([[ 4.,  3.,  8.,  8.],  
        [ 1.,  2.,  5.,  9.]]) ,  
 array([[ 6.,  8.,  2.,  1.],  
        [ 8.,  4.,  8.,  9.]]) ,  
 array([[ 9.,  6.,  6.,  8.],  
        [ 1.,  6.,  8.,  2.]])]
```

(даний масив поділили на три підмасиви по $12/3=4$ стовпці у кожному)

```
>>> hsplit(a,2)
```

```
[array([[ 4.,  3.,  8.,  8.,  6.,  8.],  
        [ 1.,  2.,  5.,  9.,  8.,  4.]]) ,  
 array([[ 2.,  1.,  9.,  6.,  6.,  8.],  
        [ 8.,  9.,  1.,  6.,  8.,  2.]])]
```

(даний масив поділили на 2 підмасиви по $12/2=6$ стовпці у кожному)

```
>>> hsplit(a,(4,7,9))
```

```
[array([[ 4.,  3.,  8.,  8.],  
        [ 1.,  2.,  5.,  9.]]) ,  
 array([[ 6.,  8.,  2.],  
        [ 8.,  4.,  8.]]) ,  
 array([[ 1.,  9.],  
        [ 9.,  1.]]) ,  
 array([[ 6.,  6.,  8.],  
        [ 6.,  8.,  2.]])]
```

(даний масив поділили на підмасиви перших 4 стовпці, наступні від 5-го до 7-го стовпця, наступні 8-мий і 9-тій стовпці, і на кінець – решта, від 10-го по 12-ий стовпці)

Функція `vsplit` ділить масив згідно вертикальних осей, а функція `array_split` ділить масив згідно вказаних осей.

```
>>> array_split(a,4,axis=1)
```

```
[array([[ 4.,  3.,  8.],
        [ 1.,  2.,  5.]])],
 array([[ 8.,  6.,  8.],
        [ 9.,  8.,  4.]])],
 array([[ 2.,  1.,  9.],
        [ 8.,  9.,  1.]])],
 array([[ 6.,  6.,  8.],
        [ 6.,  8.,  2.]])]
```

(поділ масиву на 4 підмасиви відповідно до горизонтальної осі)

```
>>> array_split(a,4,axis=0)
```

```
[array([[ 4.,  3.,  8.,  8.,  6.,  8.,  2.,  1.,  9.,  6.,  6.,  8.]])],
 array([[ 1.,  2.,  5.,  9.,  8.,  4.,  8.,  9.,  1.,  6.,  8.,  2.]])],
 array([], shape=(0, 12), dtype=float64),
 array([], shape=(0, 12), dtype=float64)]
```

(поділ масиву на 4 підмасиви відповідно до вертикальної осі)

1.2.7. Копіювання та вигляд

При виконанні операцій на масивах та при маніпуляції масивами, їх елементи інколи можуть бути скопійовані до нового масиву, а часом і ні. Дуже часто такі явища є джерелом непорозуміння для початківців. Існують три різні випадки, які опишемо більш детально.

Не копіювати взагалі

Найпростіші перетворення створюють не копії масивів чи їх даних.

(завантажуємо модулі)

```
>>> from numpy import *
```

```
>>> from random import *
```

(створюємо одномірний масив a)

```
>>> a=r_[1:13]
```

```
>>> a
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

(створюємо новий масив b, але не новий об'єкт)

```
>>> b=a
```

(перевіримо це)

```
>>> b is a
```

```
True
```

(тепер змінимо структуру b)

```
>>> b.shape=4,3
```

```
>>> b
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
>>> a
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

(як бачимо, масив a також змінив структуру)

Python трактує змінні об'єкти як посилання, а тому функція не виконує копіювання.

```
>>> def f(x):
...     print(id(x))
... 
```

```
>>> id(a)
```

```
76975680
```

(id повертає унікальний ідентифікатор об'єкта)

```
>>> f(a)
```

```
76975680
```

Перегляд або поверхнева копія

Різні об'єкти масиву можуть ділити ті самі дані. Метод `view` утворює новий масив, який виглядає із тими ж даними.

(утворимо масив)

```
>>> a=r_[21:32,33.]
```

```
>>> size(a)
```

```
12
```

```
>>> a
```

```
array([ 21.,  22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,
        31.,  33.]
```

```
>>> c=a.view()
```

```
>>> c
```

```
array([ 21., 22., 23., 24., 25., 26., 27., 28., 29., 30.,
       31., 33.] )
```

```
>>> c is a
```

```
False
```

```
>>> c.base is a
```

```
True
```

(c є представлення даних, які належать a)

```
>>> c.flags.owndata
```

```
False
```

(змінимо розмірність масиву a)

```
>>> a.shape=4,3
```

```
>>> a
```

```
array([[ 21., 22., 23.],
       [ 24., 25., 26.],
       [ 27., 28., 29.],
       [ 30., 31., 33.]])
```

(змінимо розмірність масиву c)

```
>>> c.shape=2,6
```

```
>>> a.shape
```

```
(4,3)
```

(структура a не змінилася)

```
>>> c[0,2]=1234
```

```
>>> a  
  
array([[ 21.,  22., 1234.],  
       [ 24.,  25.,  26.],  
       [ 27.,  28.,  29.],  
       [ 30.,  31.,  33.]])
```

(заміна елемента [0,2] масиву с на 1234 викликала зміну елемента [0,2] масиву а)

Поділ масиву повертає його вигляд

```
>>> s=a[1:3,:]  
  
>>> s  
  
array([[ 24.,  25.,  26.],  
       [ 27.,  28.,  29.]])
```

(створили новий масив s)

```
>>> s[:]  
  
array([[ 24.,  25.,  26.],  
       [ 27.,  28.,  29.]])
```

(s[:] повертає вигляд масиву s)

```
>>> s[:]=100
```

(всім елементам масиву s присвоєно значення 100)

```
>>> a  
  
array([[ 21.,  22., 1234.],  
       [100., 100., 100.],  
       [100., 100., 100.],  
       [ 30.,  31.,  33.]])
```

(змінюються елементи масиву а, які відображені у масиві s)

Глибоке копіювання

Функція `copy` створює повну копію масиву і його даних

```
>>> d=a.copy()
```

```
>>> d
```

```
array([[ 21.,  22., 1234.],
       [ 100., 100., 100.],
       [ 100., 100., 100.],
       [ 30.,  31.,  33.]])
```

```
>>> d is a
```

```
False
```

```
>>> d.base is a
```

```
False
```

(d не ділить з a жодних елементів)

```
>>> d[1,0]=1111
```

```
>>> d
```

```
array([[ 21.,  22., 1234.],
       [ 1111., 100., 100.],
       [ 100., 100., 100.],
       [ 30.,  31.,  33.]])
```

```
>>> a
```

```
array([[ 21.,  22., 1234.],
       [ 100., 100., 100.],
       [ 100., 100., 100.],
       [ 30.,  31.,  33.]])
```

1.2.8. Функції та методи перегляду

Нижче приводимо список найбільш уживаних у *NumPy* функцій і методів складених за категоріями.

Array Creation

`arange`, `array`, `copy`, `empty`, `empty_like`, `eye`, `fromfile`, `fromfunction`, `identity`, `linspace`, `logspace`, `mgrid`, `ogrid`, `ones`, `ones_like`, `r`, `zeros`, `zeros_like`.

Conversions

`ndarray.astype`, `atleast_1d`, `atleast_2d`, `atleast_3d`, `mat`.

Manipulations

`array_split`, `column_stack`, `concatenate`, `diagonal`, `dsplit`, `dstack`, `hsplit`, `hstack`,
`ndarray.item`, `newaxis`, `ravel`, `repeat`, `reshape`, `resize`, `squeeze`,
`swapaxes`, `take`,
`transpose`, `vsplit`, `vstack`.

Questions

`all`, `any`, `nonzero`, `where`.

Ordering

`argmax`, `argmin`, `argsort`, `max`, `min`, `ptp`, `searchsorted`, `sort`.

Operations

`choose`, `compress`, `cumprod`, `cumsum`, `inner`, `ndarray.fill`, `imag`, `prod`,
`put`, `putmask`, `real`, `sum`.

Basic Statistics

cov, mean, std, var.

Basic Linear Algebra

cross, dot, outer, linalg.svd, vdot.

1.2.9. Менш основне

Правила мови програмування

Мова програмування допускає універсальні функції, щоб належним чином обробляти вхідні дані, які не мають точно такої ж форми.

Перше правило мови програмування полягає у тому, що у випадку масивів із різними розмірностями, до масивів найменшої розмірності постійно додається "1" до тих пір, поки розмірності не будуть однаковими.

Друге правило мови програмування гарантує, що масиви розмірності "1" вздовж певної розмірності поведуть себе так, якби вони мали розмірність масиву з найбільшою розмірністю вздовж цього виміру. Значення елемента масиву припускається однаковим за цим виміром для трансльованого масиву.

Після застосування цих правил мови розміри всіх масивів повинні бути рівними.

1.2.10. Незвичайна індексація та трюки із індексами

NumPy має у своєму розпорядженні більше можливостей індексування у порівнянні із стандартними послідовностями *Python*. На додаток, крім індексування цілих масивів або їх частини, як було показано вище, масиви можуть бути індексовані цілими масивами і булевими масивами.

Індексування масивами індексів

```
>>> from numpy import *
```

```
>>> from random import randint
```

(створимо масив квадратів перших 12 чисел)

```
>>> a=arange(12)**2
```

```
>>> a
```

```
array([ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121])
```

(створюємо масив індексів)

```
>>> i=array([1,2,3,2,7,9])
```

(за допомогою цього масиву утворимо масив із a)

```
>>> a[i]
```

```
array([ 1, 4, 9, 4, 49, 81])
```

(утворимо двовимірний масив індексів)

```
>>> j=array([[3,1],[10,7]])
```

```
>>> a[j]
```

```
array([[ 9, 1],
       [100, 49]])
```

(структура масиву індекса диктує структуру утвореного масиву)

Якщо масив індексів *a* є багатовимірний, тоді кожен простий масив відноситься до першої розмірності масиву *a*. Наступний приклад ілюструє такий випадок при перетворенні образу міток у кольоровий образ, при цьому використовуючи палетку кольорів.

```
>>> palette=array([[0,0,0],
...                [255,0,0],
...                [0,255,0],
...                [0,0,255],
...                [255,255,255]])
```

(утворили багатовимірний масив palette)

```
>>> image=array([[0,1,2,0],[0,3,4,0]])
```

(утворили багатовимірний масив індексів image)

```
>>> palette[image]
```

```
array([[[ 0, 0, 0],
         [255, 0, 0],
         [ 0, 255, 0],
         [ 0, 0, 0]],
       [[ 0, 0, 0],
         [ 0, 0, 255],
         [255, 255, 255],
         [ 0, 0, 0]])
```

(утворено масив palette трьох кольорів)

Можна також використовувати масив індексів розмірності більшої ніж один. Масив індексів за кожним виміром повинен мати однакову розмірність.

```
>>> a=array([randint(0,50) for k in range(12)]).reshape([3,4])
```

```
>>> a
```

```
array([[ 4, 10, 35, 2],
       [ 7, 18, 36, 10],
       [26, 37, 16, 19]])
```

(утворили двовимірний масив даних)

```
>>> i=array([[0,1],[1,2]])
```

```
>>> j=array([[2,1],[3,3]])
```

(утворили перший i та другий j двовимірні масиви індексів масиву даних a)

```
>>> a[i,j]
```

```
array([[35, 18],  
       [10, 19]])
```

(із масивів індексів формуються пари індексів $[i[s,r], j[s,r]]$ і утворюється масив із елементами масиву a : $a[0,2], a[1,1], a[1,3], a[2,3]$)

```
>>> a[i,1]
```

```
array([[10, 18],  
       [18, 37]])
```

```
>>> a[:,i]
```

```
array([[[ 4, 10],  
        [10, 35]],  
       [[ 7, 18],  
        [18, 36]],  
       [[26, 37],  
        [37, 16]]])
```

Очевидно, що масиви індексів i та j можна подати за допомогою списку l і ефект буде той самий.

```
>>> l=[i,j]
```

```
>>> a[l]
```

```
array([[10, 18],  
       [18, 37]])
```

Однак, ми не можемо підставити i та j до цього масиву, так як цей масив інтерпретується як показник індекса першого виміру масиву a .

Індексування масивами використовується також при пошуку максимуму часових рядів:

```
>>> time=linspace(20,145,5)
```

```
>>> data=sin(arange(20)).reshape(5,4)

>>> time

array([ 20. ,  51.25,  82.5 , 113.75, 145. ])

>>> data

array([[ 0. ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])

>>> ind=data.argmax(axis=0)

>>> ind

array([2, 0, 3, 1], dtype=int32)

>>> time_max=time[ind]

>>> data_max=data[ind,range(data.shape[1])]

>>> time_max

array([ 82.5 ,  20. , 113.75,  51.25])

>>> data_max

array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])

>>> all(data_max==data.max(axis=0))

True
```

Індексування масивами можемо скористатися у випадку, коли маємо на меті виконувати переписування значень елементів масиву:

```
>>> a=arange(5)
```

```
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]]=1000
>>> a
array([ 0, 1000, 2, 1000, 1000])
```

Однак, якщо список індексів містить однакові значення індексів а відповідним елементам присвоюються різні значення, тоді заміна триває дещо довше і вказаному елементу присвоюється останнє значення із списку присвоєнь:

```
>>> a=arange(5)
>>> a
>>> array([0, 1, 2, 3, 4])
>>> a[[1,1,3]]=[111,112,13]
>>> a
array([ 0, 112, 2, 13, 4])
```

Це є достатньо обґрунтовано, але потрібно бути обачним у випадку, якщо маємо намір скористатися із конструкції *Python +=*, так як можна отримати не той результат, на який очікуєте:

```
>>> a=arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Навіть якщо 0 двічі зустрічається на у списку індексів, нульовий елемент збільшується тільки один раз. Це відбувається так тому, що інструкція *Python* `a+=1` є еквівалентна дії `a=a+1`.

Індексування булевими масивами

Якщо ми індексуємо масив за допомогою масиву (цілих) індексів, то необхідно створити список індексів для вибору елементів масиву. Із логічними індексами цей підхід є просто чудовий; ми явно вказуємо на ті елементи масиву, які бажаємо та які не потребуємо.

Найбільш натуральним шляхом щодо індексування булевими змінними, є використання масиву булевих змінних, який має таку ж розмірність, що і оригінальний масив даних.

```
>>> from numpy import *
```

```
>>> from random import randint
```

```
>>> a=array([[randint(-20,20) for j in range(4)]for i in
range(5)])
```

```
>>> a
```

```
array([[ 1, 10, 12, -12],
       [ 10, 7, -5, 7],
       [-2, -20, 2, 20],
       [-19, -18, 20, -8],
       [ 17, -18, 9, -17]])
```

```
>>> b=a<0
```

```
>>> b
```

```
array([[False, False, False, True],
       [False, False, True, False],
       [ True, True, False, False],
       [ True, True, False, True],
       [False, True, False, True]], dtype=bool)
```

```
>>> a[b]
```

```
array([-12, -5, -2, -20, -19, -18, -8, -18, -17])
```

Ця властивість може бути прекрасно використана у підстановках.

```
>>> a[b]=1111
```

```
>>> a
```

```
array([[ 1, 10, 12, 1111],
       [ 10, 7, 1111, 7],
       [1111, 1111, 2, 20],
       [1111, 1111, 20, 1111],
       [ 17, 1111, 9, 1111]])
```

У наступному прикладі продемонструємо, як використовуючи індексування булевими масивами згенерувати зображення, відоме як *множина Mandelbrot'a*.

```
>>> from numpy import *
```

```
>>> from matplotlib.pyplot import *
```

```
>>> from random import randint
```

(визначимо функцію)

```
>>> def mandelbrot(h,w,max_it=20):
...     """Returns an image of the Mandelbrot fractal of
...     size (h,w)."""
...     y, x = ogrid[-1.4:1.4:h*1j,-2:0.8:w*1j]
...     c=x+y*1j
...     z=c
...     divtime=max_it+zeros(z.shape,dtype=int)
...     for i in range(max_it):
...         z=z**2+c
...         diverge=z*conj(z)>2**2
...         div_now=diverge & (divtime==max_it)
...         divtime[div_now]=i
...         z[diverge]=2
...     return divtime
```

```
>>> imshow(mandelbrot(400,400))
```

Обчислений образ зображено на [рисунку 1.1](#).

Другий спосіб індексування булевими масивами є аналогічний до цілочисельного індексування; до кожного виміру масиву ми додаємо одновимірний булевий масив та вибираємо ті частини масиву, які потрібно.

```
>>> from numpy import *
```

```
>>> from matplotlib.pyplot import *
```

```
>>> from random import randint
```

```
>>> a=floor(100*random.random([4,5])-50)
```

```
>>> a
```

```
array([[ -22.,  -2.,  -7.,  -5.,   2.],  
       [-25., -32., 29.,  -5.,  -9.],  
       [-27.,  1., 27., 13., 11.],  
       [-24., -37., -12., 33., -36.]])
```

```
>>> b1=array([False,True,True,False])
```

```
>>> b2=array([False,True,False,False,True])
```

```
>>> a[b1,:]
```

```
array([[ -25., -32., 29.,  -5.,  -9.],  
       [-27.,  1., 27., 13., 11.]])
```

```
>>> a[b1]
```

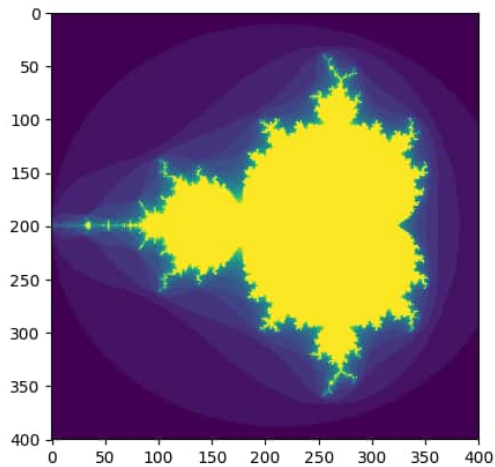


Рис. 1.1: Множина Mandelbrok'a

```
array([[ -25., -32., 29., -5., -9.],  
       [-27.,  1., 27., 13., 11.]])
```

```
>>> a[:,b2]
```

```
array([[ -2.,  2.],  
       [-32., -9.],  
       [  1., 11.],  
       [-37., -36.]])
```

```
>>> a[b1,b2]
```

```
array([-32., 11.])
```

Зазначимо, що довжина 1D булевого масиву повинна співпадати із довжиною виміру (або осі), яку ми маємо на меті виділити. У прикладі вище `b1` є одновимірним масивом довжини 3 (кількість рядків масиву `a`), а `b2` (довжини 4) відповідає другому рангу (кількість стовпців) масиву `a`.

`ix_()` функція

`ix_()` функція може бути використана для об'єднання різних векторів так, щоб отримати результат для кожного `n`-аплету. Наприклад, якщо ми хочемо обчислити всі значення `a+b*c` для всіх систем із трьох елементів, заданих векторами `a`, `b`, `c`, тоді:

```
>>> from numpy import *
```

```
>>> a=array([2,3,4,5])
```

```
>>> b=array([8,5,4])
```

```
>>> c=[5,4,6,8,3]
```

```
>>> ax,bx,cx=ix_(a,b,c)
```

```
>>> ax
```

```
array([[2],
       [3],
       [4],
       [5]])
```

```
>>> bx
```

```
array([[8],
       [5],
       [4]])
```

```
>>> cx
```

```
array([[5, 4, 6, 8, 3]])
```

```
>>> ax.shape, bx.shape, cx.shape
```

```
((4, 1, 1), (1, 3, 1), (1, 1, 5))
```

```
>>> result=ax+bx*cx
```

```
>>> result
```

```
array([[42, 34, 50, 66, 26],
       [27, 22, 32, 42, 17],
       [22, 18, 26, 34, 14]],

      [[43, 35, 51, 67, 27],
       [28, 23, 33, 43, 18],
       [23, 19, 27, 35, 15]],

      [[44, 36, 52, 68, 28],
       [29, 24, 34, 44, 19],
       [24, 20, 28, 36, 16]],

      [[45, 37, 53, 69, 29],
       [30, 25, 35, 45, 20],
       [25, 21, 29, 37, 17]])
```

```
>>> result[1,2,3]
```

```
35
```

```
>>> a[1]+b[2]*c[3]
```

```
35
```

Визначимо у скороченій формі наступне:

```
>>> def ufunc_reduce(ufct,*vctrs):
...     vs=ix_(*vctrs)
...     r=ufct.identity
...     for v in vs:
...         r=ufct(r,v)
...     return r
```

і тоді отримаємо таке

```
>>> ufunc_reduce(add,a,b,c)
```

```
array([[15, 14, 16, 18, 13],
       [12, 11, 13, 15, 10],
       [11, 10, 12, 14, 9]],

      [[16, 15, 17, 19, 14],
       [13, 12, 14, 16, 11],
       [12, 11, 13, 15, 10]],

      [[17, 16, 18, 20, 15],
       [14, 13, 15, 17, 12],
       [13, 12, 14, 16, 11]],

      [[18, 17, 19, 21, 16],
       [15, 14, 16, 18, 13],
       [14, 13, 15, 17, 12]]])
```

Перевага цього способу пониження у порівнянні із `ufunc.reduce` полягає у тому, що він робить можливим використання правило поширення для того, щоб уникнути обчислення елементів масиву розміру кратному числу векторів.

Індексування за допомогою стрічки

Див. `RecordArrays`.

1.2.11. Лінійна алгебра

Праці над створенням модуля лінійної алгебри у процесі. Основи лінійної алгебри включають наступні команди.

Найпростіші операції на масивах.

Більше про основні операції на масивах у середовищі *Python* можна знайти у `linalg.py` у папці *NumPy*.

```
>>> from numpy import *
```

```
>>> a=array(floor([[1,2],[3,4]]))
```

```
>>> print(a)
```

```
[[1. 2.]  
 [3. 4.]
```

```
>>> a.transpose()
```

```
array([[ 1.,  3.],  
       [ 2.,  4.]])
```

(обчислення транспонованої до a матриці)

```
>>> aInv=linalg.inv(a)
```

```
>>> aInv
```

```
array([[ -2. ,  1. ],  
       [ 1.5, -0.5]])
```

(обчислення оберненої до a матриці)

```
>>> aDaInv1=a.dot(aInv)
```

```
>>> aDaInv2=dot(a,aInv)
```

```
>>> print('a.a(-1): \n',aDaInv1,','; \n \n',aDaInv2)
```

```
a.a(-1):  
[[ 1.00000000e+00  1.11022302e-16]  
 [ 0.00000000e+00  1.00000000e+00]];  
[[ 1.00000000e+00  1.11022302e-16]  
 [ 0.00000000e+00  1.00000000e+00]]
```

(два формати добутку двох матриць)

```
>>> ind=eye(2)
```

```
>>> print('\nI=', ind)
```

```
I= [[ 1. 0.]  
     [ 0. 1.]
```

(обчислення одиничної матриці другого порядку)

```
>>> trace(a)
```

```
5
```

(слід матриці a)

```
>>> b=array([[8.],[18.]])
```

```
>>> y=linalg.solve(a,b)
```

```
>>> y
```

```
array([[ 2.],  
       [ 3.]])
```

(обчислення розв'язку лінійної системи)

```
>>> linalg.eig(a)
```

```
(array([-0.37228132, 5.37228132]),  
 array([[ -0.82456484, -0.41597356],  
        [ 0.56576746, -0.90937671]]))
```

(обчислення власних значень та відповідних власних векторів матриці a)

Прийоми та поради

Наведемо список коротких та корисних порад.

”Автоматична” зміна розмірності Щоб змінити розмірність масиву, можна знехтувати із однієї розмірності, яка пізніше визначається автоматично:

```
>>> a=arrange(30)
```

```
>>> a.shape=3,-1,5
```

(тут -1 означає, що розмірність добереться така, яка потрібно)

```
>>> a.shape
```

```
(3, 2, 5)
```

```
>>> a
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]],
      [[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]],
      [[20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

Складання векторів Як можна сконструювати двовимірний масив із одновимірних рядків однакової розмірності? У системі MatLab[®] це робиться дуже просто: якщо x та y два вектори однакової довжини, то достатньо впровадити наступний код: $m=[x;y]$. У програмному середовищі *NumPy* це можна виконати за допомогою функцій `column_stack`, `dstack`, `hstack` та `vstack`, в залежності від розмірності масиву, у який будемо ”складувати” ці вектори. Наприклад:

```
>>> x=arange(1,14,3)
```

```
>>> y=arange(13,31,4)
```

(утворення двох векторів $x=array([1,4,7,10,13])$ та $y=array([13, 17,21,25,29])$ довжиною 5)

```
>>> m=hstack([x,y])
```

```
>>> m
```

```
array([ 1, 4, 7, 10, 13, 13, 17, 21, 25, 29])
```

```
>>> xy=vstack([x,y])
```

```
array([[ 1, 4, 7, 10, 13],
       [13, 17, 21, 25, 29]])
```

(декілька прикладів застосування вищезгаданих функцій на багатовимірних масивах)

```
>>> a1=array([[1,2,3],[4,5,6]]);
```

```
>>> a2=array([[0,9,8],[9,8,7],[12,14,15]]);
```

```
>>> a1Va2=vstack([a1,a2])
```

```
>>> a1Va2
```

```
array([[ 1, 2, 3],
       [ 4, 5, 6],
       [ 0, 9, 8],
       [ 9, 8, 7],
       [12, 14, 15]])
```

```
>>> a1Ha3=hstack([a1,a2[1:3,:]])
```

```
>>> a1Ha3
```

```
array([[ 1, 2, 3, 9, 8, 7],
       [ 4, 5, 6, 12, 14, 15]])
```

```
>>> dstack([a1[1,:],a2[1,:]])
```

```
array([[4, 9],
       [5, 8],
       [6, 7]])
```

```
>>> dstack([a1,a2[0:2,:]])
```

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

Див. також *NumPy* для користувачів MatLab[®].

1.2.12. Гістограми

Функція *NumPy* `histogram`, застосована до масиву, повертає пару векторів: гістограму масиву та вектор інтервалів. Але треба бути обережним, оскільки у модулі `matplotlib` також є функція для побудови гістограми (її назва `hist`, так само як у MatLab[®]), яка відрізняється від відповідної у *NumPy*. Основна відмінність від `pylab.hist` буде графік гістограми автоматично, тоді як `numpy.histogram` генерує дані.

```
>>> import numpy as np
```

```
>>> import matplotlib.pyplot as plt
```

```
>>> mu, sigma = 2, 0.5
```

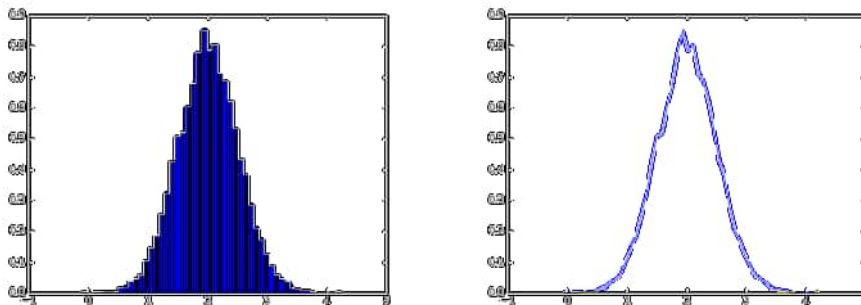
```
>>> v=np.random.normal(mu,sigma,10000)
```

(побудовано масив 10000 даних нормального розподілу із середньоквадратичним відхиленням $\sigma^2 = 0.5^2$ та середнім 2)

```
>>> plt.hist(v,bins=50,normed=1)
```

```
>>> plt.show()
```

(обчислення гістограми нормального розподілу на 50-ти інтервалах за допомогою функції модуля `matplotlib`)



(а) побудована функцією модуля `matplotlib`; (б) побудована функцією модуля `NumPy`;

Рис. 1.2: Гістограми.

(обчислення гістограми за допомогою модуля `NumPy`)

```
>>> (n,bins)=np.histogram(v,bins=50,normed=True)
```

```
>>> plt.plot(.5*(bins[1:]+bins[:-1]),n)
```

```
>>> plt.show()
```

1.2.13. Інші джерела для опрацювання

- [Посібник *Python*](#).
- [Посилання](#).
- [Посібник *SciPy*](#).
- [Конспект *SciPy*](#).
- [MatLab, R, IDL, NumPy \ SciPy словники](#).

1.3. Короткий вступ до *SciPy*

SciPy – це бібліотека модулів *Python* для наукових обчислень, яка має у своєму розпорядженні більш спеціалізовані функціональні можливості, ніж загальні структури даних і математичні алгоритми бібліотеки *NumPy*. Наприклад, бібліотека *SciPy* містить модулі для обчислення спеціальних функцій, які часто застосовують у наукових та інженерних дослідженнях, оптимізації, інтегрування, інтерполяції, обробки зображень. Як і у бібліотеці *NumPy*, численні внутрішні алгоритми бібліотеки *SciPy* виконуються як попередньо скомпільований код на мові C, забезпечуючи високу швидкість. Поза тим, як *NumPy* і сам *Python*, бібліотека *SciPy* є у вільній дистрибуції.

Для практичного використання підпрограм бібліотеки *SciPy* потрібно познайомитися із дещо новим синтаксисом і у цьому підрозділі основна увага зосереджена на прикладах практичного застосування цієї бібліотеки у коротких програмах, тісно пов'язаних із науковими і інженерними розрахунками.

1.3.1. Інтегрування і звичайні диференціальні рівняння.

Пакет `scipy.integrate` містить методи для обчислення визначених інтегралів. За допомогою цих методів можна обчислити властиві і невластиві інтеграли. Ці методи також дозволяють інтегрувати системи диференціальних рівнянь.

Визначені інтеграли функції однієї змінної.

Основною програмою чисельного інтегрування є `scipy.integrte.quad`. У цій програмі застосовується адаптивна квадратура для наближеного обчислення значення інтеграла шляхом поділу області інтегрування на менші інтервали, які вибираються ітеративно у відповідності до заданої інтервальної похибки. У найпростішій формі метод приймає три аргументи: об'єкт функції *Python*, який відповідає інтегровній функції, `func` і межі інтегрування `a` і `b`. В аргументі функції `func` обов'язково повинен

передаватися принаймні один об'єкт, якщо цей об'єкт містить більше одного об'єкта, то інтегрування здійснюється за координатою, яка відповідає першому значенню аргумента. При визначенні підінтегральної `func` функції зручно користуватися лямбда-конструктором. Наприклад,

для одержання значення інтеграла $\int_1^4 x^{-2} dx = \frac{3}{4}$:

```
>>> from scipy.integrate import quad
>>> func1 = lambda t: t**(-2)
>>> quad(func1, 1, 4)
(0.7500000000000002, 1.913234548258993e-09)
```

метод `quad` повертає кортеж двох значень – значення інтеграла і абсолютну похибку одержаного результату.

Для обчислення невластивих інтегралів використовуємо спеціальне значення `np.inf`. Крім того, будемо користуватися функціями із пакета *NumPy*, тому перш ніж запишемо код, потрібно завантажити *NumPy*, замінивши його ідентифікатор на коротший `np`. Код процедури обчислення невластивого інтеграла запишемо у редакторі, про що свідчитиме відсутність запрошення командного рядка `>>>`. Крім того, щоб побачити явно одержаний результат, до нього потрібно застосувати функцію `print`. Отож,

```
import numpy as np
from scipy.integrate import quad
nt = quad(lambda t: np.exp(-t**2), 0, np.inf)
print("Integral value: { }. \ nPrecision:{ }.".format(nt[0],nt[1]))
mn=np.sqrt(np.pi)/2
print("Manual value: { }".format(mn))
```

Після компіляції одержимо результат обчислень:

```
Integral value: 0.8862269254527579.  
Precision:7.101318378329813e-09.  
Manual value: 0.8862269254527579
```

Відомо, що значення невластивого інтеграла $\int_0^{\infty} e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$, тому у програмі вище безпосередньо аналітично обчислено і одержано значення його значення – можна порівняти результати. Крім того, підінтегральну функцію визначили за допомогою лямбда-конструктора безпосередньо як аргумент `quad`.

Для складніших функцій потрібно явно визначити функцію за допомогою ключового слова `def`. Розглянемо приклад інтегрування розривної функції:

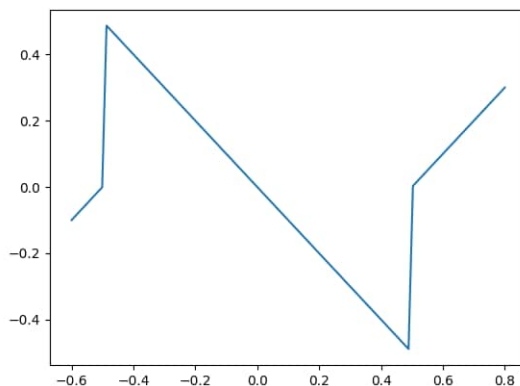
```
def myfunc2(x):  
    if abs(x) < 0.5:  
        return -x  
    return x - 0.5*np.sign(x)  
int2 = quad(myfunc2, -0.6, 0.8)  
print(int2)  
(0.039999999999999925, 8.326672684688674e-17)
```

Обчислимо графік функції `myfunc2`:

```
import numpy as np  
import matplotlib.pyplot as plt  
x = np.linspace(-0.6, 0.8, 100)  
y = np.array([myfunc2(t) for t in x])  
plt.plot(x,y)  
plt.show()
```

Програма поверне графік, зображений на [рис. 1.3](#). Вертикальні відрізки прямих на графіку функції – розриви першого роду функції.

Функції із сингулярними (особливими, критичними) точками або із

Рис. 1.3: Графік підінтегральної функції `myfunc2(x)`.

точками розриву можуть створювати проблеми для програми чисельного інтегрування, навіть якщо визначений інтеграл однозначно визначений. Наприклад, функція $f(x) = \frac{\sin x}{x}$ посідає усуну особливу точку $x = 0$, у якій за звичайного застосування методу `quad` програма повертає критичну помилку (пропонуємо читачеві переконатися у цьому самостійно). Усунути цю проблему можна шляхом застосування розширеного формату `quad`, вказавши четвертим аргументом `points=[x0, x1,...]` особливі точки x_0, x_1, \dots функції (ці точки не обов'язково повинні бути записані по порядку):

```
sinc = lambda x: np.sin(x)/x
res1 = quad(sinc, -2, 2, points=[0,])
print(res1)
(3.210825953605389, 3.5647329017567276e-14)
```

Кратні інтеграли.

Для обчислення подвійних, потрійних і n -кратних інтегралів у модулі `scipy.integrate` передбачені методи `dblquad`, `tplquad` і `nquad` відповідно. У загальному випадку межі інтегрування за однією змінною можуть залежати від іншої змінної, тому синтаксис вказаних вище методів є складніший у порівнянні із однократним інтегралом.

Метод `dblquad` обчислює подвійний інтеграл:

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y) dx dy.$$

Тут функція $f(x, y)$ передається у метод як функція щонайменше двох змінних `func(y, x, ...)`. Важливо, що першим аргументом цієї функції є y , а другим – x . Межі інтегрування передаються до `dblquad` як наступні чотири аргументи. Перші дві межі визначають проміжок інтегрування за змінною x , аналогічно як у методі `quad`. Наступні два аргументи `gfun` і `hfun` визначають нижню та верхню межу інтегрування за змінною y , і вони обов'язково повинні бути об'єктами, що залежать від одного аргумента – числа із плаваючою комою, значення x (тобто, ці аргументи повинні бути функціями x). Якщо щонайменше одна межа стосовно y не залежить від x , то `gfun` або `hfun` можуть повертати сталі значення.

Як приклад, обчислимо значення подвійного інтеграла

$$\int_1^2 \int_{-2}^{-1} \frac{2x}{y^2} dx dy$$

за допомогою процедури:

```
import numpy as np
from scipy.integrate import dblquad

func2v = lambda y, x: 2*x/y**2
a, b = 1, 2
gfunc, hfunc = (lambda x: -2), (lambda x: -1)
res = dblquad(func2v, a, b, gfunc, hfunc)
```

```
print(res)
(1.5, 2.2180352904498422e-14)
```

Тут важливо відзначити, що верхню та нижню межу інтегрування за змінною y передаємо як функції x , які при кожному значенні x приймають сталі значення. А також, що при визначенні підінтегральної функції `func2v` першим аргументом визначаємо y , а другим – x (хоч у звичайному математичному записі трактуємо порядковість аргументів навпаки). Пропонуємо читачеві переконатися, що не дотримання цих вимог приводить до невірному результату.

Код обчислення подвійного інтеграла можна записати в одному рядку:

```
dblquad(lambda y, x: 2*x/y**2, 1, 2, lambda x: -2, lambda x: -1)
```

★ *Приклад 1.3.1.* Розглянемо приклад обчислення площі плоскої фігури обмеженої кардіоїдою $\rho = 2 + 2 \cos \theta$ та колом $\rho = 2$ (ззовні кола) за допомогою подвійного інтеграла. Графіки кривих у полярній системі координат та фігуру обчислимо за допомогою програми:

```
import numpy as np
import matplotlib.pyplot as plt
pfunc1 = lambda t: 2+2*np.cos(t)
pfunc2 = lambda t: 2+t*0
theta = np.linspace(0, 2*np.pi, 200)
theta34 = np.linspace(-np.pi/2, np.pi/2, 200)
r1 = pfunc1(theta)
r2 = pfunc2(theta)
r3 = pfunc1(theta34)
r4 = pfunc2(theta34)
plt.polar(theta, r1, 'r-', lw = 2)
plt.polar(theta, r2, 'g-', lw = 2)
plt.fill_between(theta34, r4, r3, facecolor = 'brown', alpha=0.3)
plt.show()
```

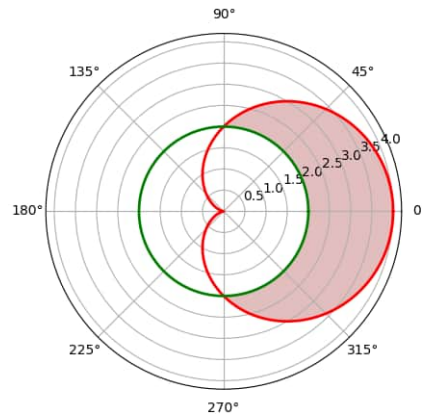


Рис. 1.4: Фігура, обмежена кардіоїдою $\rho = 2 + 2 \cos \theta$ та колом $\rho = 2$ (ззовні кола).

Програма поверне графік, зображений на [рис. 1.4](#)

Кардіоїда і коло перетинаються у точках $\theta_0 = -\pi/2$ і $\theta_1 = \pi/2$ і при кожному значенні θ із проміжку $-\pi/2 \leq \theta \leq \pi/2$ полярний радіус ρ знаходиться на проміжку $2 \leq \rho \leq 2 + 2 \cos \theta$. Отож, площа фігури дорівнює подвійному інтегралу:

$$S = \int_{-\pi/2}^{\pi/2} \int_2^{2+2\cos\theta} \rho \, d\rho d\theta$$

Без труднощів аналітично обчислимо подвійний інтеграл, значення якого $S = 8 + \pi$ і порівняємо результат застосування чисельного методу:

```
import numpy as np
from scipy.integrate import dblquad

func2v =
gfunc, hfunc =(lambda t: 2 + t*0), (lambda t: 2 + 2*np.cos(t))
```

```

a, b = -np.pi/2, np.pi/2
res = dblquad(func2v, a, b, gfunc, hfunc)
print("Python result: S= { }; \ nPrecision: epsilon= { }.".format(res[0], res[1]))
print("Manual result: S={ } .".format(8+np.pi))
Python result: S= 11.141592653589795;
Precision: epsilon= 1.2369652695009885e-13.
Manual result: S=11.141592653589793.

```

Тут підінтегральній функції присвоєно ідентифікатор (назву) `func2v`, яка визначена за допомогою `lambda`-функції `lambda ro, t: ro + t*0` і дорівнює `ro` радіусу для всіх значень кута `t`. Так як задекларовано `func2v` як функцію двох змінних радіуса `ro` і кута `t`, а фактично залежить тільки від `ro`, формально при визначенні аналітичного вигляду функції додали доданок `t*0`, який не впливає на значення функції. Цей формальний прийом застосовуватимемо для правильної компіляції програми, хоч у даному випадку система поверне правильний результат без цього формального кроку. Аналогічно у випадку визначення функції нижньої межі внутрішнього інтегрування `gfunc`. Зовнішнє інтегрування проводиться у межах від `a = -np.pi/2` до `b = np.pi/2`, а внутрішнє – від `gfunc = (lambda t: 2 + t*0)` до `hfunc = (lambda t: 2 + 2*np.cos(t))`.

Інтегрування із кратністю більшою від двох виконується методом `scipy.integrate.nquad`, документацію і застосування можна знайти за посиланням [45]. Розглянемо приклад.

★ **Приклад 1.3.2.** Обчислити масу і центр маси тетраедра, обмеженого координатними площинами і площиною $x + y + z = 1$, густина якого у кожній точці (x, y, z) задається функцією $\rho = \rho(x, y, z)$. Розглянемо випадки, коли $\rho(x, y, z) = 1$, $\rho(x, y, z) = x$, $\rho(x, y, z) = x^2 + y^2 + z^2$.

Маса тетраедра може бути обчислена за формулою:

$$m = \iiint_V \rho(x, y, z) dx dy dz = \int_0^1 dx \int_0^{1-x} dy \int_0^{1-x-y} \rho(x, y, z) dz,$$

а координати (x_m, y_m, z_m) центра маси за формулами:

$$x_m = \iiint_V x \cdot \rho(x, y, z) dx dy dz,$$

$$y_m = \iiint_V y \cdot \rho(x, y, z) dx dy dz,$$

$$z_m = \iiint_V z \cdot \rho(x, y, z) dx dy dz.$$

Код програми обчислення маси і центра маси для випадків різних густин тетраедра запишеться так:

```
import numpy as np
from scipy.integrate import tplquad

# Integration limits x, y, z.
a, b = 0, 1
gfun, hfun = (lambda x: 0), (lambda x: 1 - x)
qfun, rfun = (lambda x, y: 0), (lambda x, y: 1 - x - y)
limts = (a, b, gfun, hfun, qfun, rfun)

# Density functions.
rho123 = [lambda x, y, z: 1, \
          lambda x, y, z: x, \
          lambda x, y, z: x**2 + y**2 + z**2]

for rho in rho123:
    # Mass.
    m, _ = tplquad(rho, *limts)
    # Mass center (xm, ym, zm).
    xmrho, _ = tplquad(lambda x, y, z: x * rho(x, y, z), *limts)
    ymrho, _ = tplquad(lambda x, y, z: y * rho(x, y, z), *limts)
    zmrho, _ = tplquad(lambda x, y, z: z * rho(x, y, z), *limts)
    xm, ym, zm = xmrho / m, ymrho / m, zmrho / m
```

```
print('mass = :g, Coefficients Mass Center = (:g, :g, :g)'. \
      format(m, xm, ym, zm))
```

Після компіляції програма поверне результат у вигляді:

```
mass = 0.166667, Coefficients Mass Center = (0.25, 0.25, 0.25)
mass = 0.0416667, Coefficients Mass Center = (0.4, 0.2, 0.2)
mass = 0.05, Coefficients Mass Center = (0.277778, 0.277778, 0.277778)
```

Звичайні диференціальні рівняння (ODE – Ordinary Differential Equation)

Звичайні диференціальні рівняння у *Python* можна чисельно розв'язувати за допомогою двох методів: `scipy.integrate.odeint` або `scipy.integrate.solve_ivp` (solve an initial value problem – розв'язування задачі з початковими умовами (задача Cauchy)). Другий метод був розроблений у версії 1.0 бібліотеки *SciPy*, і власне цей метод рекомендують для практичного використання. З допомогою методу `solve_ivp` можна розв'язувати початкові задачі для диференціальних рівнянь та систем диференціальних рівнянь першого порядку. Однак, для знаходження розв'язку диференціального рівняння чи системи диференціальних рівнянь вищих порядків потрібно попередньо звести до еквівалентної задачі для системи диференціальних рівнянь першого порядку. Повну документацію та приклади застосування методів `odeint` і `solve_ivp` можна знайти за посиланнями [46, 47].

Знаходження чисельного розв'язку задачі Cauchy для звичайного диференціального рівняння (IVP ODE – Initial Value Problem for Ordinary Differential Equation). У найпростішому форматі вхідними даними методу `solve_ivp` для знаходження розв'язку початкової задачі для звичайного диференціального рівняння першого порядку

$$\frac{dy}{dx} = f(x, y)$$

є три аргументи: права частина рівняння $f(x, y)$ (Right Hand Side (RHS) – права частина), початкове і кінцеве значення аргументу (проміжок інтегрування), початкова умова y_0 . За цими даними метод розв’язування ODE вибирає і повертає послідовність точок із проміжка із заданого проміжка, у яких виконується інтегрування.

Для прикладу, розглянемо диференціальне рівняння першого порядку, яке описує швидкість реакції $A \rightarrow P$, що виражається у концентрації реагента A :

$$\frac{dA}{dt} = -kA.$$

Неважко проінтегрувати у явному вигляді останнє рівняння і записати розв’язок:

$$A = A_0 e^{-kt},$$

де A_0 – початкова концентрація реагента A .

Для знаходження чисельного розв’язку диференціального рівняння за допомогою методу `solve_ivp` потрібно записати його у вище наведеній формі (рівняння розв’язане стосовно похідної) із однією залежною змінною $y(t) = A$, яка є функцією незалежної змінної t (часу). Отож, у даному випадку диференціальне рівняння має вигляд:

$$\frac{dy}{dt} = -ky.$$

Визначимо праву частину рівняння $f(x, y)$ (у загальному випадку права частина є функцією двох змінних t і y), яка у даному випадку має простий вигляд:

```
def rhs(t,y):  
    return -k*y
```

Зауважимо, що важливим є порядок аргументів функції правої частини. Початковий та кінцевий моменти (точки) часу `t_span` повинні передаватися як кортеж `(t0,tf)`, а початкова умова – як об’єкт типу масив, навіть якщо, як у даному випадку, передається тільки одне значення. Одержимо:

```
soln = solve_ivp(rhs, (t0, tf), [y0])
```

Програма повертає об'єкт `soln` класу `OdeResult`, у якому міститься важлива інформація стосовно масиву точок `soln.t` часу, які використовуються при чисельному інтегруванні рівняння, масиву `soln.y` значення розв'язку початкової задачі у цих точках та `soln.success` – повідомлення про успішне чи ні завершення алгоритму знаходження розв'язку IVP ODE із заданою точністю. Продемонструємо на прикладі.

★ **Приклад 1.3.3.** Запишемо програму, за допомогою якої графічно порівняємо чисельний та аналітичний розв'язки розглянутої вище задачі для таких даних: $k = 0.2$, $y(0) = A_0 = 100$.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
# Constant first order reaction rate, 1/s.
k = 0.2
# Initial condition for y: 100% reagent at time t = 0.
y0 = 100
# Start and end points (moments) of time for integration.
t0, tf = 0, 20
def rhs(t, y):
    """ Return dy/dt = f(t, y) at time t. """
    return -k * y
# Integration of a differential equation.
soln = solve_ivp(rhs, (t0, tf), [y0])
t, y = soln.t, soln.y[0]
# Plotting and comparing numerical and exact (analytical) solutions.
plt.plot(t, y, 'o', color='r', label=r'solve_ivp')
plt.plot(t, y0 * np.exp(-k*t), color='green', label='Exact')
plt.xlabel(r'$t$ /s$')
plt.ylabel('Remaining reactant (%)')
plt.legend()
plt.show()
```

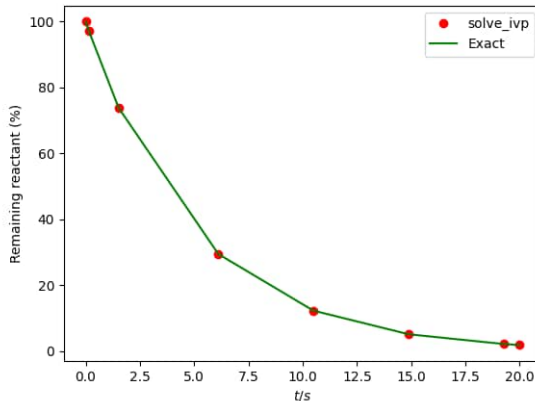


Рис. 1.5: Експоненціальне спадання концентрації реагента у реакції першого порядку: точний розв'язок і розв'язок чисельним методом у точках, визначених методом розв'язування IVP ODE.

Програма повертає графіки точного (Exact – лінія зеленого кольору) та наближеного (solve_ivp – значення чисельного розв'язку у вузлах сітки, точки червоного кольору) розв'язків задачі, зображені на [рис. 1.5](#). Крім того, у процесі виконання програми одержимо об'єкт soln типу бібліотеки, у якій міститься, зокрема, повідомлення про перебіг виконання програми (ключ 'message'), вузли сітки (ключ 't'), значення наближеного розв'язку у вузлах сітки (ключ 'y'):

```
>>> soln['message']
'The solver successfully reached the end of the integration interval.'

>>> soln['t']
array([ 0. ,  0.13797324,  1.51770566,  6.11233264, 10.4942851 , \
        14.88942184, 19.28378064, 20. ])

>>> soln['y']
array([[100. , 97.27826059, 73.81996665, 29.46989971, \
```

```
12.27361105, 5.09829981, 2.11809624, 1.83542088]])
```

Розглянутий приклад демонструє той випадок, коли важливо одержати кінцеве значення концентрації. Коли ж стоїть завдання дослідження концентрації із більшою точністю у проміжних точках, то побудованої сітки вузлів за замовчуванням (пропонуємо читачеві переконаватися, що кількість вузлів сітки у прикладі вище дорівнює 8), на якій побудований чисельний розв'язок, є замало. Тоді спеціально побудовану послідовність точок можна передати `solve_ivp` за допомогою опційного (додаткового) аргумента `t_eval`. Крім того, присвоїмо аргументу `dense_output` значення `True`, щоб визначити об'єкт `OdeSolution` із іменем `soln` як один із об'єктів, які повертає метод. Такий підхід можна використати для генерування інтерполяції розв'язку. Отож, після вказаних вище змін, процедура запишеться у вигляді:

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

k = 0.2
y0 = 100
t0, tf = 0, 20
t_eval = np.linspace(t0, tf, 21)
def rhs(t, y):
    """ Return dy/dt = f(t, y) at time t. """
    return -k * y
# Integration of a differential equation.
soln = solve_ivp(rhs, (t0, tf), [y0], t_eval=t_eval, dense_output=True)
t = soln.t
y = soln.sol(t)[0]
plt.plot(t, y, 'o', color='r', label=r'solve_ivp')
plt.plot(t, y0 * np.exp(-k*t), color='green', label='Exact')
plt.xlabel(r'$t / s$')
plt.ylabel('Remaining reactant (%)')
```

```
plt.legend()  
plt.show()
```

Зауважимо, що об'єкт `soln.sol` представляє собою масив розв'язку на побудованій сітці `t_eval` і відповідає одній залежній змінній `y`, тому масив має індекс `[0]`. Графік із точками розв'язку `y`, аналогічний одержаному у попередньому прикладі, зображено на [рис. 1.6](#).

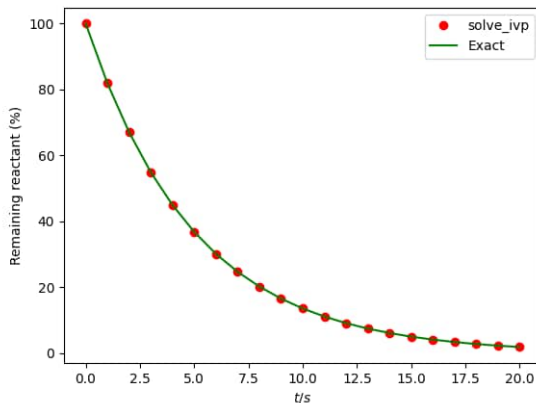


Рис. 1.6: Експоненціальне спадання концентрації реагента у реакції першого порядку: точний розв'язок і розв'язок чисельним методом наперед визначених точках.

Якщо права частина диференціального рівняння залежить від параметрів, то їх також можна передати у метод `solve_ivp` за допомогою опційного параметра `args`. У наведеному вище прикладі `k` належить до глобальної області видимості (не є аргументом функції, належить до глобальної змінної), але цей параметр `k` можна передати у явному вигляді:

```
def rhs(t, y, k):  
    return - k * y
```


★ **Приклад 1.3.4.** Для прикладу розглянемо реакцію, яка протікає у формі двох етапів реакцій першого порядку: $A \rightarrow B \rightarrow P$ із постійними швидкостями k_1 і k_2 . Рівняння, які визначають швидкість зміни реагентів A і B , мають вигляд:

$$\begin{cases} \frac{d[A]}{dt} = -k_1[A], \\ \frac{d[B]}{dt} = k_1[A] - k_2[B]. \end{cases}$$

Не складає зусиль знайти аналітичний розв'язок цієї системи двох рівнянь першого порядку із постійними коефіцієнтами, а для чисельного знаходження розв'язку нехай $y_1 \equiv [A]$, $y_2 \equiv [B]$. За допомогою програми нижче знайдемо чисельний розв'язок системи диференціальних рівнянь для значень параметрів $k_1 = 0.2$, $k_2 = 0.8$, за початкових умов $y_1(0) = 100$, $y_2(0) = 0$ і порівняємо його із аналітичним розв'язком (див. [рис. 1.7](#)):

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
# Rate constants of first order reactions, 1/s.
k1, k2 = 0.2, 0.8
# Initial conditions for y1, y2: [A](t=0) = 100, [B](t=0) = 0.
A0, B0 = 100, 0
# Correctly chosen grid of time points for this reaction.
t0, tf = 0, 20
def rhssyst(t, y, k1, k2):
    """ Return dy_i/dt = f(y_i, t) at time t. """
    y1, y2 = y
    rhs1eq = -k1 * y1
    rhs2eq = k1 * y1 - k2 * y2
    return rhs1eq, rhs2eq
# Integration of a differential equation.
y0 = A0, B0
soln = solve_ivp(rhssyst, (t0, tf), y0, dense_output=True, args=(k1, k2))
```

```

t = np.linspace(t0, tf, 100)
A, B = soln.sol(t)
# [P] is determined by the law of conservation of matter (mass).
P = A0 - A - B
# Analytical result.
Aexact = A0 * np.exp(-k1*t)
Bexact = A0 * k1/(k2-k1) * (np.exp(-k1*t) - np.exp(-k2*t))
Pexact = A0 - Aexact - Bexact
plt.plot(t, A, 'g.', label='[A]')
plt.plot(t, B, 'b.', label='[B]')
plt.plot(t, P, 'r.', label='[P]')
plt.plot(t, Aexact, 'm', label='exactA')
plt.plot(t, Bexact, 'y', label='exactB')
plt.plot(t, Pexact, 'c', label='exactC')
plt.xlabel(r'$t/s$')
plt.ylabel('Concentration (arb. units)')
plt.legend()
plt.show()

```

Звичайні диференціальні рівняння вищих порядків.

Для знаходження чисельного розв'язку початкової задачі для диференціального рівняння порядку вищого ніж один методами *Python* потрібно перш за все звести цю задачу до еквівалентної задачі для системи звичайних диференціальних рівнянь першого порядку. Із теорії диференціальних рівнянь відомо, що довільне звичайне диференціальне порядку n зводиться до відповідної системи n диференціальних рівнянь першого порядку стосовно n невідомих [3].

★ **Приклад 1.3.5.** Розглянемо для приклад диференціальне рівняння, яке описує дію генератора гармонічних коливань, яке записується так:

$$\frac{d^2x}{dx^2} = -\omega^2 x,$$

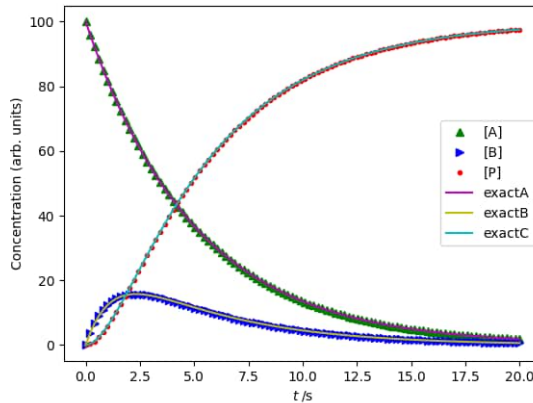


Рис. 1.7: Дві взаємно пов'язані реакції першого порядку: чисельний і аналітичний (точний) розв'язки.

де x – зміщення стосовно положення рівноваги, ω – кутова частота. Це рівняння відомим методом зводиться до системи двох диференціальних рівнянь першого порядку:

$$\begin{cases} \frac{dx_1}{dt} = x_2, \\ \frac{dx_2}{dt} = -\omega^2 x_1, \end{cases}$$

де x_1 – перепозначення x . Одержану систему диференціальних рівнянь можна чисельно розв'язати методом, описаним у попередньому підрозділі. Допишемо до рівняння початкові умови початкові умови:

$$x(0) = 3, \quad \frac{dx(0)}{dt} = 0.$$

Програма, яка повертає чисельний розв'язок початкової задачі для рівняння коливання запишеться так:

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
# Harmonic Oscillator Frequency (1/s).
omega = 0.9
# Initial conditions for x1 = x and x2 = dx/dt at the time t = 0.
A, v0 = 3, 0 # cm, cm/s.
x0 = A, v0
# Correctly chosen grid of time points.
t0, tf = 0, 20
def rhssyst(t, x, omega):
    """ Return dx/dt = f(t, x) at time t. """
    x1, x2 = x
    rhs1eq = x2
    rhs2eq = -omega**2 * x1
    return rhs1eq, rhs2eq

# Integration of a differential equation.
soln = solve_ivp(rhssyst, (t0, tf), x0, dense_output=True, args=(omega, ))
t = np.linspace(t0, tf, 100)
x1, x2 = soln.sol(t)
# Plotting and comparing numerical and exact (analytical) solutions.
plt.plot(t, x1, 'o', color='k', label=r'solve_ivp()')
plt.plot(t, A * np.cos(omega * t), color='gray', label='Exact')
plt.xlabel(r'$t/s$')
plt.ylabel(r'$x/cm$')
plt.legend()
plt.show()
```

Графіки розв'язків зображені на [рис. 1.8](#).

Метод `solve_ivp` можна сконфігурувати для використання різних алгоритмів розв'язування системи ODE за допомогою значення опційного атрибуту `method`. У [таблиці 1.11](#) наведені можливі значення атрибуту

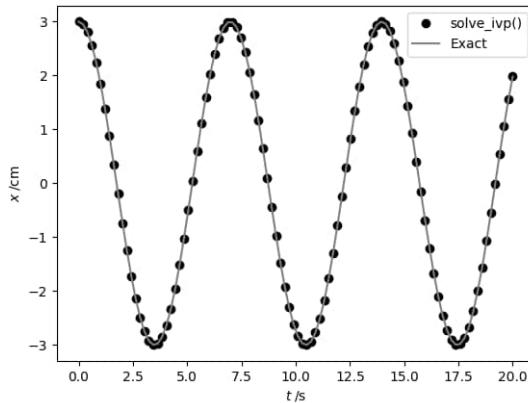


Рис. 1.8: Генератор гармонічних коливань: чисельний і точний (аналітичний) розв'язки.

`method`. З замовчуванням це значення `'RK45'` – явний метод Runge-Kutta порядку 5 (4) – це надійний узагальнений метод для нежорстких систем. Задача називається *жорсткою*, якщо у процесі виконання алгоритму потрібно застосувати нескінченно малі кроки у інтервалах інтегрування, щоб одержати розв'язок із заданою точністю. Жорсткі задачі часто виникають тоді, коли члени в ODE змінюються по величині у сильно відмінних часових масштабах. Власне методи `'Radau'`, `'BDF'` і `'LSODA'` заслуговують на увагу, якщо припускається, що досліджувана задача для рівняння чи системи є жорсткою.

★ **Приклад 1.3.6.** Класичний приклад жорсткої системи ODE – кінетичний аналіз автокаталітичної реакції Robertson'a [32] трьох компонент (реагентів) $x = [X]$, $y = [Y]$, $z = [Z]$ за початкових умов

Табл. 1.11: Методи інтегрування ODE для `scipy.integrate.solve_ivp`.

Модуль	Опис
'RK45'	явний метод Runge-Kutta порядку 5 (4);
'RK23'	явний метод Runge-Kutta порядку 3 (2);
'Radau'	неявний метод Runge-Kutta сімейства Radau ІІА порядку 5; застосовується також для розв'язування жорстких задач;
'BDF'	наближена формула оберненого диференціювання (BDF), явний метод, можна застосувати для жорстких задач;
'LSODA'	гнучкий метод, який може автоматично визначити жорсткість задачі і переключитися між методом (алгоритмом) Adams'a (для нежорстких задач) і BDF (для жорстких задач).

$x = 1, y = z = 0$:

$$\begin{cases} \dot{x} \equiv \frac{dx}{dt} = -0.04x + 10^4 yz, \\ \dot{y} \equiv \frac{dy}{dt} = 0.04x - 10^4 yz - 3 \times 10^7 y^2, \\ \dot{z} \equiv \frac{dz}{dt} = 3 \times 10^7 y^2. \end{cases}$$

Зауважимо, що масштаби часу цих реакцій суттєво відрізняються між собою, особливо для $[Y]$.

За допомогою методу `solve_ivp`, при цьому за замовчуванням використовується алгоритм Runge-Kutta, на часовому інтервалі $[0,500]$ програма знаходження чисельного розв'язку задачі автокаталітичної реакції Robertson'a має вигляд:

```
import numpy as np
from scipy.integrate import solve_ivp
def rhssyst(t, y):
```

```
u, v, w = y
rhs1eq = - 0.04 * u + (1.e+4) * v * w
rhs2eq = 0.04 * u - (1.e+4) * v * w - (3.e+7) * v**2
rhs3eq = (3.e+7) * v**2
return rhs1eq, rhs2eq, rhs3eq
t0, tf = 0, 500
y0 = 1, 0, 0
soln = solve_ivp(rhssyst, (t0, tf), y0)
print(soln)
```

Після компіляції повертає результат:

```
message: 'Required step size is less than spacing between numbers.'
nfev: 2546
njev: 0
nlu: 0
sol: None
status: -1
success: False
      t: array([0. , 0.0005 , 0.001 , 0.0015 , 0.002 , ...
0.34000034, 0.34050034, 0.34100034, 0.34150034, 0.34200034])
t_events: None
      y: array([[ 1.00000000e+00,  9.99980000e-01,  9.99960002e-01, ...,
 9.87222926e-01,  9.87213324e-01,  9.87190116e-01],
 [ 0.00000000e+00,  1.82136213e-05,  2.91886215e-05, ...,
 1.20683091e-04,  1.74765050e-04, -3.06270592e-04],
 [ 0.00000000e+00,  1.78616405e-06,  1.08098537e-05, ...,
 1.26563913e-02,  1.26119109e-02,  1.31161546e-02]])
y_events: None
```

Тобто, метод інформує, що після 2546 звертань до функції алгоритм зупинений, так як 'Required step size is less than spacing between numbers.' – Необхідний розмір кроку менший за відстань між числами.. За замовчуванням вибрана сітка вузлів, на якій алгоритм не збігається, тобто розв'язок не побудований. Це є наслідок того, що задача, як відзна-

чено вище, для системи диференціальних рівнянь є жорсткою. Крім того, крайній правий вузол рівний 0.34200034, що означає розбіжність алгоритму у початкових точках сітки, побудованої за замовчуванням методом `solve_ivp`.

Скористаємося тепер методом `Radau` для знаходження чисельного розв'язку досліджуваної жорсткої задачі автокаталітичної реакції.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
def rhsSODE(t, y):
    """ODEs for Robertson ' s chemical reaction system."""
    x, y, z = y
    rhs1eq = -0.04 * x + 1.e4 * y * z
    rhs2eq = 0.04 * x - 1.e4 * y * z - 3.e7 * y**2
    rhs3eq = 3.e7 * y**2
    return rhs1eq , rhs2eq , rhs3eq
# Start and end times.
t0, tf = 0, 500
# Initial conditions: [X] = 1; [Y] = [Z] = 0.
y0 = 1, 0, 0
# Solution using a method that allows reducing the problem \
# to a stiff system of ODEs.
soln = solve_ivp(rhsSODE, (t0, tf), y0, method='Radau')
print(soln.nfev, 'evaluations required.')
# Plotting concentrations as a function of time. Scale [Y] by 10**YFAC
# so that variations are visible on the axis used for [X] and [Z].
YFAC = 4
plt.plot(soln.t, soln.y[0], label='[X]')
plt.plot(soln.t, 10**YFAC*soln.y[1], label= \
    r'$10^{\ } \times [Y]'.format(YFAC))
```

```
plt.plot(soln.t, soln.y[2], label='[Z]')
plt.xlabel('time /s')
plt.ylabel('concentration /arb. units')
plt.legend()
plt.show()
```

Програма поверне інформацію, що для одержання результату використано:

248 evaluations required.

та графіки зміни концентрації кожного із трьох реагентів $[X]$, $[Y]$, $[Z]$, зображені на [рис. 1.9](#).

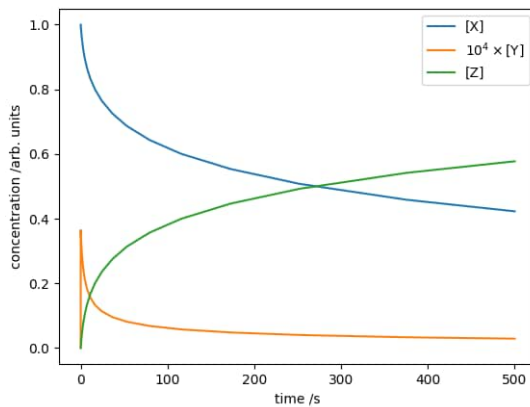


Рис. 1.9: Система хімічних рівнянь (реакцій) Robertson'а, чисельно проінтегрована методом Radau ІА.

1.3.2. Оптимізація, підгонка даних та чисельні методи розв’язування рівнянь.

Пакет `scipy.optimize` містить різноманітні алгоритми мінімізації (безумовної та умовної) функції багатьох змінних, підгонки даних методом найменших квадратів, визначення розв’язків рівнянь та систем із багатьма змінними. У цьому підрозділі проведемо загальний огляд найбільш важливих доступних варіантів реалізації, але завжди слід пам’ятати, що вибір алгоритму залежить від конкретної задачі. Немає впевненості, що для довільної функції конкретний метод буде збіжний до мінімум (чи кореня рівняння тощо) або буде забезпечена швидка збіжність. Деякі алгоритми в більшій мірі застосовуються для конкретних функцій у порівнянні із іншими і свідчить про те, що чим більшою інформацією володіємо про дану функцію, тим краще. Бібліотеку *SciPy* можна сконфігурувати так, щоб виводилося попередження при критичних збогах конкретного алгоритму, і це повідомлення, як правило, може допомогти при аналізі проблеми.

Мінімізація.

Програми оптимізації бібліотеки функції *SciPy* мінімізують функцію однієї чи багатьох змінних $f(x_1, x_2, \dots, x_n)$. Для пошуку максимуму визначається мінімум функції $-f(x_1, x_2, \dots, x_n)$.

Деякі алгоритми мінімізації вимагають тільки саму функцію, для інших потрібна похідна функції за кожною із змінних (часткові похідні) у вигляді масиву, званого *матрицею Якобі* (*Jacobian-matrix*):

$$J(f) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right).$$

У деяких алгоритмах проводиться оцінка матриці Якобі чисельними методами, якщо її не можна представити як функцію у явному вигляді.

Крім того, для деяких алгоритмів оптимізації вимагається інформація про другі (частинні) похідні досліджуваної функції у вигляді симе-

тричної матриці, яка називається матрицею Hess'а (Hessian'ом)

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Матриця Якобі скалярної функції представляє локальний градієнт функції багатьох змінних, а матриця Hess'а – локальний радіус кривизни.

Безумовна оптимізація. У найпростішому форматі алгоритм мінімізації скалярної функції багатьох змінних `scipy.optimize.minimize` приймає два обов'язкові параметри і має вигляд:

```
minimize(func, x0, ...)
```

Перший аргумент – об'єкт функції `func` для обчислення (за алгоритмом) мінімізуючої функції: ця функція повинна приймати масив значень x , який визначає точки, у який повинні виконуватися обчислення (x_1, x_2, \dots, x_n) із наступними необхідними довільними аргументами. Другий обов'язковий аргумент `x0` – початкова точка, починаючи із якої алгоритм буде послідовність наближень до точки мінімуму.

Розглянемо застосування методу `minimize` для дослідження мінімумів функції *Himmelblau*, простої функції двох змінних із деякими складними властивостями, завдяки яким функція *Himmelblau* є цілком придатною для тестування алгоритмів оптимізації. Функція *Himmelblau* має вигляд:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

Графік функції Himmelblau та лінії рівня зображені на рисунках 1.10 та 1.11 відповідно.

В області $-5 \leq x \leq 5$, $-5 \leq y \leq 5$ функція Himmelblau містяться один локальний максимум

$$f(-0.270845, -0.923039) = 181.617,$$

чотири локальні мінімуми:

$$\begin{aligned} f(3, 2) &= 0, \\ f(-2.805118, 3.131312) &= 0, \\ f(-3.779310, -3.283186) &= 0, \\ f(3.584428, -1.848126) &= 0, \end{aligned}$$

та чотири сідлові точки, зображені на рис. 1.11.

Визначимо у *Python* функцію Himmelblau стандартним способом

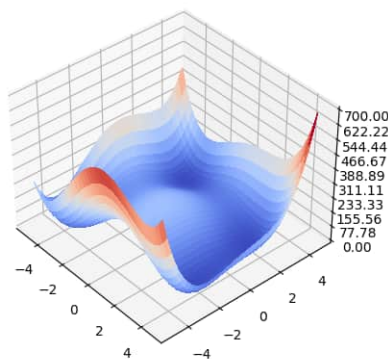


Рис. 1.10: Графік функції Himmelblau.

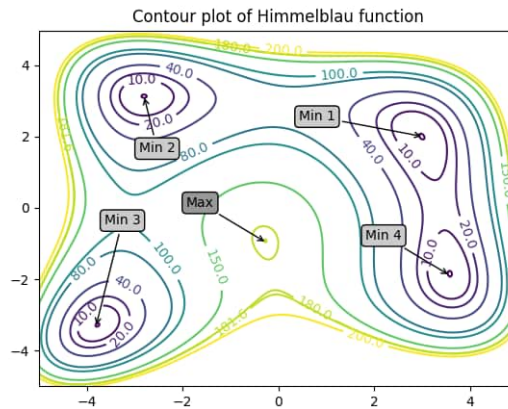


Рис. 1.11: Лінії рівня функції Himmelblau.

```
>>> def himmfunc(X):  
...     x, y = X  
...     return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

де для кращого розуміння масив X , що містить (x_1, x_2) , розгорнутий в іменовані змінні $x_1 \equiv x$, $x_2 \equiv y$.

Для пошуку мінімуму використовуємо метод `minimize` із початковою точкою $(x, y) = (0, 0)$:

```
>>> from scipy.optimize import minimize  
>>> minimize(himmfunc, (0, 0))
```

```

>>>
      fun: 1.3782267979368085e-13
      hess_inv: array([[ 0.01578229, -0.0094806 ], [-0.0094806 , 0.03494937]])
      jac: array([-3.95019932e-06, -1.19075692e-06])
      message: 'Optimization terminated successfully.'
      nfev: 48
      nit: 10
      njev: 16
      status: 0
      success: True
      x: array([2.99999994, 1.99999999])

```

Метод повертає `minimize` об'єкт типу словника, який містить інформацію про мінімізацію. У [таблиці 1.12](#) міститься опис полів словника. Якщо мінімізація завершилася успішно, то точка мінімуму передається із ключем `x` у цьому об'єкті. У даному прикладі одержана збіжність, яка близька до мінімуму функції: $f(3, 2) = 0$ (на [рис. 1.11](#) ця точка позначена Min 1).

Алгоритм, який використовується методом `minimize`, визначається за допомогою аргумента `method` у вигляді стрічки, допустимі варіанти яких наведені у [таблиці 1.13](#). Алгоритм BFGS, який використовується за замовчуванням, є ефективним квазіньютонівським методом, який апроксимує Якобіан, якщо вона не передається і не використовується Hessian. Однак, при застосуванні цього алгоритму для пошуку максимуму функції Himmelblau виникають труднощі (`himmfunc` – функція Himmelblau визначена вище):

```

>>> mxhimmfunc = lambda X: -himmfunc(X) # To find the maximum,
                                         # the -f(x, y) function is minimized.
>>> minimize(mxhimmfunc, (0.1, -0.2))

```

Табл. 1.12: Опис інформації, яка міститься у словнику, який повертає метод `scipy.optimize.minimize`.

Ключ	Опис
<code>success</code>	логічне значення, яке інформує успішно чи ні завершився алгоритм пошуку мінімуму;
<code>x</code>	якщо мінімізація завершилася успішно, то містить розв'язок: значення (x_1, x_2, \dots, x_n) , при якому функція досягає мінімуму. Якщо алгоритм завершується невдало, то ключ містить точку аварійної зупинки;
<code>fun</code>	при успішній мінімізації містить значення функції у точці мінімуму, яка визначена ключем <code>x</code> ;
<code>message</code>	стрічка, яка описує процес мінімізації;
<code>jac</code>	значення Jacobian'а: якщо мінімізація успішна, то значення Jacobian'а близька до нуля;
<code>hess, hess_inv</code>	Hessian і обернена до неї матриця (якщо була використана у процесі пошуку мінімуму);
<code>nfev, njev, nhev</code>	число операцій обчислення функції, Jacobian'а і Hessian'а.

```

>>>
  fun: -8122685219740.447
  hess_inv: array([[0.03043408, 0.00669823], [0.00669823, 0.00147411]])
  jac: array([-3.85738342e+09, 1.75265546e+10])
  message: 'Desired error not necessarily achieved due to precision loss.'

  nfev: 351
  nit: 2
  njev: 113
  status: 2
  success: False
  x: array([ 988.07169246, -1636.22149234])

```

Як бачимо, програма повертає помилку і цілі не досягає – при підході до точки максимуму програма терпить збій на крутому схилі (див. точку *Max* на [рис. 1.11](#)). Крім того, повідомлення `success` повертає значення `False`, повернена значення `message`: `'Desired error not necessarily achieved due to precision loss.'` – Величина похибки, що вимагається, не гарантована внаслідок втрати точності. Розумно було б спробувати вибрати за початкову точку, ближчою до точки максимуму. Але такий підхід також не завжди вдалий (пропонуємо читачеві самостійно переконатися у цьому).

Отже, якщо наперед не відомо, де знаходиться точка екстремуму, застосуємо інший алгоритм мінімізації і виберемо за початкову для алгоритму точку $(0, 0)$ (`mxfimfunc` визначена вище):

```
>>> minimize(mxfimfunc, (0, 0), method='Nelder-Mead')
final simplex: (array([[ -0.27086815, -0.92300745], [ -0.27082188,
                    -0.92309634], [ -0.27084492, -0.92296765]]),
              array([ -181.61652151, -181.61652149, -181.61652148]))
fun: -181.61652150549165
message: 'Optimization terminated successfully.'
nfev: 115
nit: 59
status: 0
success: True
x: array([ -0.27086815, -0.92300745])
```

Алгоритм *Nelder-Mead*'а – це симплекс-метод, який не вимагає обчислення та оцінки похідних функції a , отже, не 'пробує' підходити до точки екстремуму у напрямку крутих схилів поверхні графіка. Але все ж виконує обчислення за 115 функціями для забезпечення збіжності до локального максимуму.

І на кінець, як останній приклад, розглянемо алгоритм `dogleg`, який у методі `minimize` вимагає обчислення *Jacobian*'а і *Hessian*'а. Для функції

Табл. 1.13: Деякі методи оптимізації, які використовуються у методі `scipy.optimize.minimize`.

Ключ	Опис
'BFGS'	алгоритм BFGS (Broyden-Fletcher-Goldfarb-Shanno), який використовується за замовчуванням для пошуку безумовного чи умовного мінімуму;
'Nelder-Mead'	алгоритм Nelder-Mead'a, відомий також як симплекс-метод (спуску) або метод деформованого багатогранника. Не вимагається похідних;
'CG'	метод спряжених градієнтів;
'Powell'	метод Powell'a (для цього методу не потрібно похідних);
'dogleg'	ламано-лінійний алгоритм у довірчій області (необмежена мінімізація). При застосуванні потрібна Якобiан і Hessian (останній обов'язково повинен бути додатно визначений);
'TNC'	алгоритм Newton'a мінімізації із граничними умовами;
'l-bfgs-b'	умовна мінімізація із використанням алгоритму L-BFGS-B;
'slsqp'	sequential least-squares programming – послідовне програмування методом найменших квадратів із граничними умовами та обмеженнями типу рівності і нерівності;
'cobyla'	метод умовної оптимізації із використанням лінійної апроксимації (constrained optimization by linear approximation);

Himmelblau ці похідні мають простий аналітичний вигляд:

$$\frac{\partial f}{\partial x} = 4x(x^2 + y - 11) + 2(x + y^2 - 7),$$

$$\frac{\partial f}{\partial y} = 2(x^2 + y - 11) + 4y(x + y^2 - 7),$$

$$\frac{\partial^2 f}{\partial x^2} = 12x^2 + 4y - 42,$$

$$\frac{\partial^2 f}{\partial y^2} = 12y^2 + 4x - 26,$$

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x} = 4x + 4y.$$

Визначимо Jacobian і Hessian у *Python* так:

```
>>> def gradHimmFunc(X):
    x, y = X
    expr1 = x**2 + y - 11
    expr2 = x + y**2 - 7
    dfdx = 4*x*expr1 + 2*expr2
    dfdy = 2*expr1 + 4*y*expr2
    return np.array([dfdx, dfdy])

>>> def hessHimmFunc(X):
    x, y = X
    d2fdx2 = 12*x**2 + 4*y - 42
    d2fdy2 = 12*y**2 + 4*x - 26
    d2fdxdy = 4*(x + y)
    return np.array([[d2fdx2, d2fdxdy], [d2fdxdy, d2fdy2]])
```

Пригадаємо, що завдання полягає у визначенні максимуму функції Himmelblau за допомогою методу `minimize`, а тому Jacobian і Hessian матимуть протилежний знак:

```
>>> mxDHimmFunc = lambda X: -gradHimmFunc(X)
>>> mxDDHimmFunc = lambda X: -hessHimmFunc(X)
```

Отож, після всіх приготувань, розв'яжемо задачу пошуку максимуму функції Himmelblau, використавши алгоритм `dogleg`:

```
>>> minimize(mxhimmfunc, (0, 0), jac=mxDHimmFunc, \
            hess=mxDDHimmFunc, method='dogleg')
```

```
>>>
  fun: -181.6165215225827
  hess: array([[44.81187272,4.77553259],[4.77553259,16.85937624]])
  jac: array([-1.26922473e-10, 1.23685240e-09])
  message: 'Optimization terminated successfully.'
  nfev: 5
  nhev: 4
  nit: 4
  njev: 5
  status: 0
  success: True
  x: array([-0.27084459, -0.92303856])
```

Одержаний результат свідчить сам за себе – алгоритм завершився успішно ('Optimization terminated successfully.'). при цьому 5 разів обчислювалися функції, точка максимуму (x : `array([-0.27084459, -0.92303856])`) одержана після чотирьох ітерацій (`nit: 4`), значення локального максимуму `fun: -181.6165215225827`.

Умовна оптимізація. У прикладних задачах виникають задачі пошуку екстремуму функції із багатьма змінними за певних обмежень. Щоб скористатися описаним у попередньому підрозділі методом, розглянемо приклад: знайти точку мінімуму функції $f(x, y)$, яка задовольняє умову $x < 0$, $y < 0$, або значення мінімуму функції на прямій $x = y$.

Алгоритми `l-bfgs-b`, `tnc` і `slsqp` підтримують аргумент `bounds` для передачі у метод `minimize`. Значення аргумента `bounds` – послідовність кортежів, кожний із яких містить пари (`min`, `max`) для кожної змінної функції, які визначають межі мінімізації для кожної змінної відповідно. Якщо у якомусь напрямку відсутнє обмеження, то використовують значення `None`.

Наприклад, якщо спробувати знайти мінімум функції $f(x, y)$, починаючи із точки $(1/2, 1/2)$ без будь-яких обмежень, то метод `slsqp` забезпечить збіжність (майже точну) до мінімуму у точці

(-2.805118, 3.131312) (himmfunc – функція Himmelblau, визначена вище):

```
>>> minimize(himmfunc, (-0.5, -0.5), method='slsqp')
>>>
      fun: 4.0198726971069946e-07
      jac: array([-0.00721077, 0.00037714])
message: 'Optimization terminated successfully'
      nfev: 36
       nit: 10
      njev: 10
      status: 0
    success: True
       x: array([-2.80522924, 3.131319 ])
```

Щоб залишатися у квадранті $x < 0$, $y < 0$, потрібно встановити значення `bounds` без нижніх обмежень за x і y , а верхні межі визначити $x = 0$, $y = 0$:

```
>>> xbounds = (None, 0)
>>> ybounds = (None, 0)
>>> xybounds = (xbounds, ybounds)
>>> minimize(himmfunc, (-0.5, -0.5), bounds=xybounds, \
              method='slsqp')
>>>
      fun: 4.115667606325133e-08
      jac: array([-0.00283595, -0.00034243])
message: 'Optimization terminated successfully'
      nfev: 39
       nit: 11
      njev: 11
      status: 0
    success: True
       x: array([-3.77933774, -3.28319868])
```

Припустимо, що потрібно знайти точки екстремуму функції Himmelblau, координати яких задовольняють умову $x = y$ (лежать на діагоналі графіка, зображеного на [рис. 1.11](#)). Із методів мінімізації, наведених у [таблиці 1.13](#), два – `cobyqa` і `slsqp` – дозволяють застосувати обмеження, тому скористаємося одним із них.

Обмеження визначаються як аргумент `constraints` у методі `minimize` у вигляді словників, які визначають стрічкові ключі: `'type'` – тип обмеження і `'fun'` – об'єкт, що викликається і реалізує це обмеження. Ключ `'type'` може приймати два значення – `'eq'` або `'ineq'`, які відповідають обмеженню типу рівності (наприклад $x = y$) або нерівності (наприклад $y > 2x - 1$). Слід відзначити, що алгоритм `cobyqa` не підтримує обмеження типу рівності.

Важливо, функція обмеження типу рівності повинна повертати значення нуль, якщо умова виконується, а функція обмеження типу нерівності у випадку виконання умови повинна повертати від'ємне значення.

Для пошуку мінімуму функції Himmelblau із обмеженням $x = y$ використаємо алгоритм `slsqp` із функцією обмеження типу рівності $x - y = 0$:

```
>>> constr = 'type': 'eq', 'fun': lambda X: X[0] - X[1]
>>> minimize(himmfunc, (0, 0), constraints=constr, method='slsqp')
>>>
      fun:  8.000000000716087
      jac:  array([-16.33084416,  16.33130538])
message:  'Optimization terminated successfully'
  nfev:   25
   nit:    7
  njev:    7
status:   0
success:  True
      x:  array([2.54138438,  2.54138438])
```

Метод збігається до одного локального умовного мінімуму (Існує другий мінімум, який можна знайти, стартуючи із точки $(-2, -2)$). Про-

понуємо читачеві самостійно знайти другий локальний умовний мінімум функції Himmelblau за умови $x = y$).

А що можна сказати про локальний умовний максимум? Пропонуємо читачеві самостійно переконатися, що пошук максимуму:

```
>>> minimize(mxhimmfunc, (0, 0), constraints=constr, method='slsqp')
```

закінчиться невдачею (зауважимо, що для пошуку максимуму методом `minimize` потрібно змінити знак функції: `mxhimmfunc` є функцією Himmelblau з протилежним знаком, визначена вище).

Для пошуку локального умовно максимуму функції із обмеженням типу рівності застосуємо алгоритм `cobyla`. Для того щоб обійти обмеження на умови в алгоритмі, визначимо обмеження типу рівності $x = y$ за допомогою двох нерівностей: $x > y$, $x < y$:

```
>>> constr1 = 'type': 'ineq', 'fun': lambda X: X[0] - X[1]
>>> constr2 = 'type': 'ineq', 'fun': lambda X: X[1] - X[0]
>>> minimize(mxhimmfunc, (0, 0), constraints=(constr1, constr2), \
           method='cobyla')
>>>
      fun: -179.12499987327624
      maxcv: 0.0
      message: 'Optimization terminated successfully.'
      nfev: 34
      status: 1
      success: True
      x: array([-0.49994148, -0.49994148])
```

Отже, алгоритм пошуку локального умовного максимуму функції Himmelblau завершився успішно.

Мінімізація функції однієї змінної.

Для мінімізації функції однієї змінної *Python* має швидкий алгоритм `scipy.optimize.minimize_scalar`. Щоб одержати мінімум, потрібно

викликати цей алгоритм, вказавши аргументом досліджувану функцію. Розглянемо приклад. Визначимо алгебричну (поліном) функцію $P(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$:

```
import numpy as np
Polynomial = np.polynomial.Polynomial
polnmFunc=Polynomial((48., 28., -24., -3., 1.))
```

Побудуємо графік полінома:

```
import matplotlib.pyplot as plt
X=np.linspace(-6, 6, 100)
Y=funcPolnm(X)

fig, ax = plt.subplots()
ax.plot(X, Y)
ax.set(xlabel='X', ylabel='Y', \
       title='Polynomial  $x^4 - 3x^3 - 24x^2 + 28x + 48$ ')
ax.grid()
plt.show()
```

Застосуємо метод `minimize_scalar` для мінімізації функції $P(x)$:

```
from scipy.optimize import minimize_scalar
mnmPlnm=minimize_scalar(funcPolnm)
print(mnmPlnm)
```

Після чого одержимо результат, що алгоритм `minimize_scalar` забезпечує збіжність до мінімуму у точці x : -2.841044326595826 функції, графік якої зображено на [рис. 1.12](#):

```
>>>
fun: -91.32163915433344
message: 'Optimization terminated successfully; The returned value
satisfies the termination criteria (using xtol = 1.48e-08 )'
nfev: 15
nit: 10
```

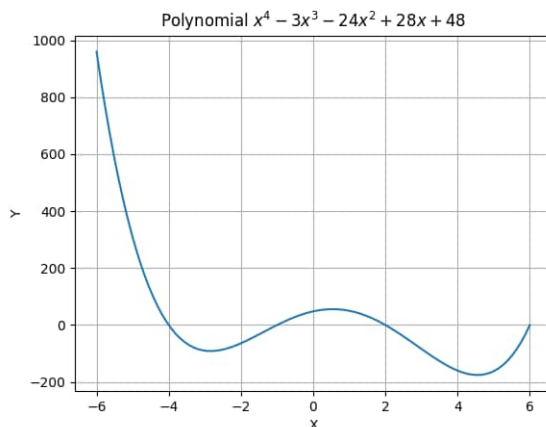


Рис. 1.12: Графік полінома $P(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$.

```
success: True
x: -2.841044326595826
```

Щоб визначити другий мінімум функції, визначимо обмежену область пошуку цього мінімуму, передаючи значення (4, 5) аргументу `bracket` (див. [рис. 1.12](#)):

```
>>> minimize_scalar(funcPolnm, bracket=(4, 5))
>>> fun: -175.45563549487977
message: 'Optimization terminated successfully; The returned value
satisfies the termination criteria (using xtol = 1.48e-08 )'
nfev: 15
nit: 11
success: True
x: 4.549468384622812
```

Накінець, визначимо максимум полінома $P(x)$, мінімізуючи функцію $-P(x)$ із вказаним значенням аргумента `bracket=(0, 1)` (див. [рис.](#)

1.12):

```
>>> minimize_scalar(-funcPolnm, bracket=(0, 1))
>>>
      fun: -55.734305899213226
  message: 'Optimization terminated successfully; The returned value
           satisfies the termination criteria (using xtol = 1.48e-08 )'
      nfev: 13
       nit: 19
  success: True
         x: 0.5415759589626433
```

1.4. Короткий вступ до *Matplotlib*

Із часу заснування росте популярність мови програмування *Python*. Разом з тим росте кількість доступних бібліотек пакетів і модулів, які розширюють функціональні можливості цієї мови. Однією із таких бібліотек є *Matplotlib*. Бібліотека *Matplotlib* містить засоби побудови графічних схем, які можуть включатися у застосунки, відображатися на екрані або повертатися у вигляді файлів зображення високої якості для публікацій.

Бібліотека *Matplotlib* має у своєму розпорядженні повнофункціональний об'єктно-орієнтований інтерфейс, однак для створення простих графічних об'єктів у режимі інтерактивного сеансу командної оболонки спрощений процедурний інтерфейс `pyplot` дає зручний спосіб візуалізації даних. У цьому підрозділі коротко зупинимося на використанні `pyplot` разом з деякими функціями *NumPy* (бібліотека *NumPy* описана у [підрозділі 1.2](#)).

Перед початком роботи із бібліотеками *Matplotlib* і *NumPy* завантажуюмо їх за допомогою інструкцій імпорту методів цих бібліотек із префіксами `plt` і `np` (загальноприйняті префікси, можна вписати власні):

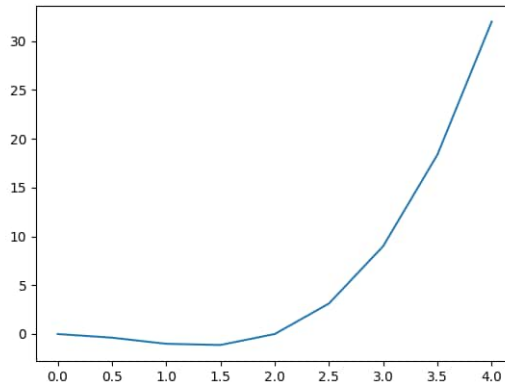
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

1.4.1. Прості графіки

Лінійні графіки та точкові діаграми.

Найпростіший лінійний (x, y) графік обчислюється за допомогою метода `plt.plot` передачею до нього двох ітерованих об'єктів однакової довжини (зазвичай це списки рівної довжини або масиви *NumPy*). Наприклад:

```
>>> nx = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
>>> ny = [0.0, -0.375, -1.0, -1.125, 0.0, 3.125, 9.0, 18.375, 32.0]
```

Рис. 1.13: Простий лінійний (x, y) графік.

```
>>> plt.plot(nx, ny)
```

```
>>> plt.show()
```

Метод `plt.plot` створює об'єкт `Matplot` (у наведеному прикладі об'єкт `Line2D`), а метод `plt.show()` виводить цей об'єкт на екран. Результат зображений на [рис. 1.13](#). За замовчуванням лінія має синій колір.

Для відображення точок (x, y) у вигляді точкової діаграми (діаграми розсіювання), а не лінійного графіка викликається метод `plt.scatter()`:

```
>>> nx = [random.random() for i in range(100) ]
```

```
>>> ny = [random.random() for i in range(100) ]
```

```
>>> plt.scatter(nx, ny)
```

```
>>> plt.show()
```

Одержана у результаті обчислення діаграма зображена на [рис. 1.14](#)

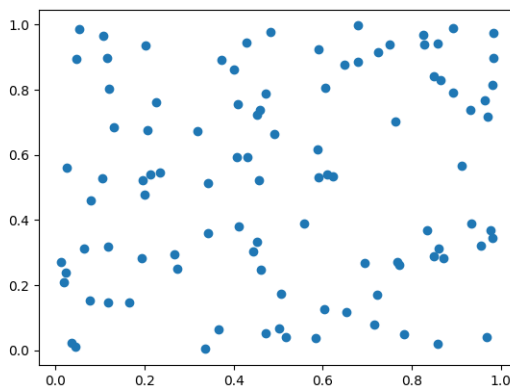


Рис. 1.14: Проста точкова діаграма.

Діаграму можна зберегти як зображення, викликавши метод `savefig(ім'я_файла)`. Потрібний формат зображення задається у розширенні файла. Наприклад:

```
savefig(file_name.png)#зберегти зображення у форматі *.png,  
savefig(file_name.pdf)#зберегти зображення у форматі *.pdf,  
savefig(file_name.eps)#зберегти зображення у форматі *.eps,  
savefig(file_name.jpeg)#зберегти зображення у форматі *.jpeg.
```

★ **Приклад 1.4.1.** Розглянемо побудову графіка функції $y = \cos^2 x$ на проміжку $[-2\pi, 2\pi]$. Для побудови графіка використаємо засоби *Python*, які розглядалися у [підрозділі 1.2.](#)

Обчислимо і виконаємо відображення y 1000 вузлів рівномірної сітки на осі OX і OY , які збережемо у списках `px` і `py`. Для обчислення списку `px` вузлів на осі OX не можна безпосередньо використати `range`, тому що цей метод генерує цілочисельні послідовності. Тому спочатку задамо крок між двома сусідніми вузлами рівномірної сітки за форму-

ЛОЮ

$$\Delta x = \frac{x_{\max} - x_{\min}}{n - 1}$$

(якщо до крайніх вузлів належать x_{\min} , x_{\max} , тоді існує $n - 1$ інтервалів довжиною Δx), тоді точки абсцис обчислюються за формулою

$$x_i = x_{\min} + i \cdot \Delta x \quad \text{для} \quad i = 0, 1, 2, \dots, n - 1.$$

Відповідні координати y обчислюються за формулою

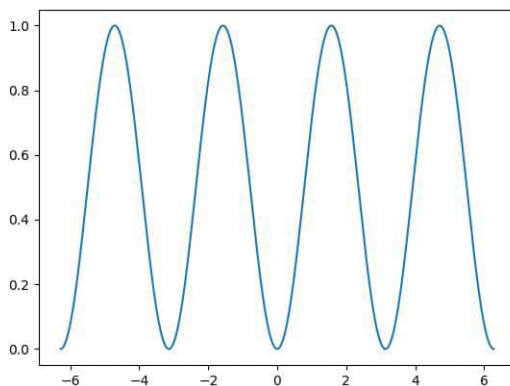
$$y_i = \cos^2(x_i).$$

Програма, записана нижче, реалізує описаний вище підхід і повертає лінійний графік функції $y = \cos^2 x$, зображений на [рис. 1.15](#):

```
import math
import matplotlib.pyplot as plt
xmin , xmax = -2. * math.pi, 2. * math.pi
n = 1000
x = [0.] * n
y = [0.] * n
dx = (xmax - xmin)/(n-1)
for i in range(n):
    xpt = xmin + i * dx
    x[i] = xpt
    y[i] = math.cos(xpt)**2
plt.plot(x,y)
plt.show()
plt.savefig('Figure_1.jpg')
```

Метод `linspace` і векторизація

. Для побудови і візуалізації графіка функції $y = \cos^2 x$ у прикладі із попереднього рідрозділу потрібно було виконати досить багато допоміжних операцій і найбільша частина припала на формуванні списків

Рис. 1.15: Графік функції $y = \cos^2 x$.

x і y . За допомогою бібліотеки *NumPy* формування, зокрема, вказаних списків, можна виконати простіше.

По-перше, послідовність координат x із постійним кроком (список x) можна створити за допомогою метода `linspace`. Цей метод є подібний до функції `range` для чисел із плаваючою комою: приймає початкове та кінцеве значення, а також кількість значень у послідовності і генерує масив значень, які утворюють арифметичну прогресію між двома заданими значеннями (включаючи самі ці значення). Наприклад:

```
>>> import numpy as np
>>> x = np.linspace(-2, 2, 11)
>>> x
array([-2. , -1.6, -1.2, -0.8, -0.4, 0. , 0.4, 0.8, 1.2, 1.6, 2. ])
```

По-друге, аналоги методів із модуля `math` у бібліотеці *NumPy* можуть виконувати обчислення на ітерованих об'єктах (таких як списки або масиви *NumPy*). Тому вираз `y=np.cos(x)` створює послідовність значень (насправді це є масив *NumPy ndarray*), рівних $\cos(x_i)$ для кожного

значення x_i із масиву x :

```
import numpy as np
import matplotlib.pyplot as plt
n = 1000
xmin , xmax = -2*np.pi, 2*np.pi
x = np.linspace(xmin , xmax , n)
y = np.cos(x)**2
plt.plot(x,y)
plt.show()
```

Ця операція називається векторизацією. Списки і кортежі можна перетворювати у об'єкти-масиви, які підтримують операцію векторизації, за допомогою метода-конструктора `array`:

```
>>> lst=[1.0, 2.0, 3.0, 4.0]
>>> lst
[1.0, 2.0, 3.0, 4.0] #створений список
>>> ar2lst=np.array(lst) #конвертація списку у об'єкт array
>>> ar2lst
array([1., 2., 3., 4.]) #виконано векторизацію
```

Тому маємо такі результати:

```
>>> lst*2
[1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0, 4.0] #двократна конкатенація списку
>>> ar2lst*2
array([2., 4., 6., 8.]) #кожний елемент помножений на 2
```

Щоб додати другу лінію до графіка потрібно викликати ще раз метод `np.plot`:

```
...  
x = np.linspace(xmin , xmax , n)  
y1 = np.cos(x)**2  
y2 = np.cos(x)**3  
plt.plot(x,y1)  
plt.plot(x,y2)  
plt.show()
```

Програма поверне графік, зображений на [рис. 1.16](#):

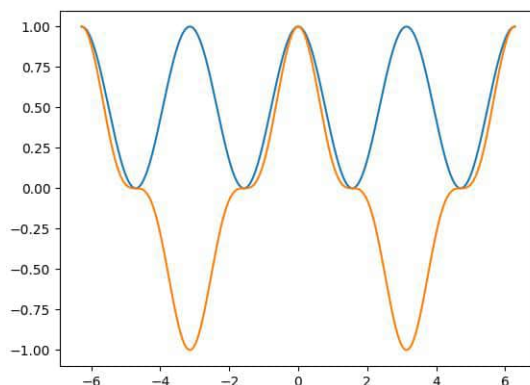


Рис. 1.16: Графіки функцій $y = \cos^2 x$ і $y = \cos^3 x$.

Зауважимо, що після виводу графіка за допомогою метода `show` чи після його зберігання за допомогою метода `savefig` графік стає недоступний для повторного висвітлення – для цього потрібно ще раз викликати метод `plt.plot`. Причина полягає у процедурній сутності інтерфейса `pyplot`: кожне звертання до метода `pyplot` змінює внутрішній стан об'єкта графіка. Об'єкт графіка створюється послідовними звертаннями до таких методів (додавання ліній, підписи і мітки, встановлення граничних значень осей тощо), і лише після того об'єкт графіка виводиться на екран чи зберігається як образ у вигляді файлу.

1.4.2. Мітки, підписи, налаштування параметрів графіків

Мітки і підписи

Для кожної лінії на графіку можна створити мітку за допомогою аргумента `label`, якому передається стрічковий літерал. Ця мітка буде зображена на побудованому графіку тільки після застосування методу `plt.legend`.

```
plt.plot(x, y1, label = 'cos^2(x)')
plt.legend()
plt.show()
```

Місце розташування опису за замовчуванням – верхній правий кут графіка, яке можна змінити, якщо у методі `legend` застосувати аргумент `loc` із одним із стрічкових або цілочисельних значень, наведених у [таблиці 1.14](#)

Назва графіка і мітка осей.

За допомогою методу `plt.title` можна присвоїти назву, яка розташована вище осей координат. Це здійснюється шляхом передачі стрічки назви графіка. Методи `plt.xlabel` і `plt.ylabel` керують мітками осей x і y : потрібно передати мітку у вигляді стрічки у ці методи. Необов'язковий додатковий атрибут `fontsize` встановлює розмір шрифту у пунктах.

Використання \LaTeX у `pyplot`.

У графіках `pyplot` можна використовувати мову розмітки даних \LaTeX . Для цього потрібно дозволити застосування цієї можливості у `rc`-налаштуваннях *Matplotlib* так:

```
plt.rc('text', usetex=True)
```

Після цього можна передати команду мови розмітки \LaTeX як стрічку до довільної мітки, яка виводиться таким способом. Рекомендує-

Табл. 1.14: Специфікатори місця розташування опису графіка.

Стрічка	Ціле число
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

ться використовувати неформатовані стрічки, щоб *Python* не виводив на екран символи із "\ " (backslash – обернений слеш), які характерні для L^AT_EX-а.

★ **Приклад 1.4.2.** Побудуємо і виведемо на екран графіки функцій $f_n(x) = x^n \cos x$ для $n = 1, 2, 3, 4$ за допомогою коду:

```
import matplotlib.pyplot as plt
import numpy as np
plt.rc('text', usetex=True)
x = np.linspace(-3*np.pi, 3*np.pi, 1001)
for n in range(1,5):
    y = x**n*np.cos(x)
    y /= max(y)
    plt.plot(x,y, label=r'$x^n \cos\{x\}$'.format(n))
```

```
plt.legend(loc='lower center')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$f_n(x)$', fontsize=14)
plt.title(r'$f_n(x)=x^n \cos\{x\} \quad (n=1,2,3,4)$', fontsize=14)
plt.savefig('Figure_2')
plt.show()
```

Для зручного порівняння графіків функцій значення промасштабовані до максимального значення 1 у розглядуваній області. Одержані графіки зображені на [рис. 1.17](#).

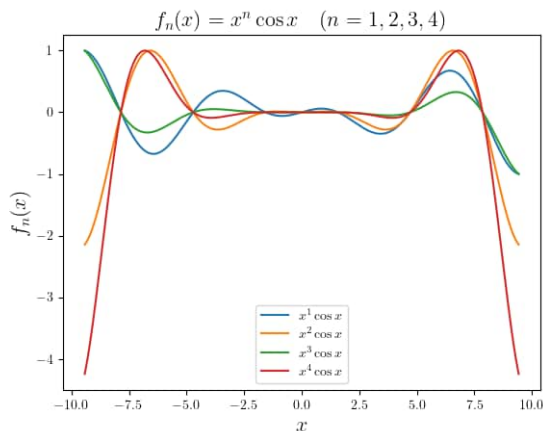


Рис. 1.17: Графіки функцій $f_n(x) = x^n \cos x$ ($n = 1, 2, 3, 4$).

1.4.3. Графіки із спеціальними налаштуваннями параметрів.

Маркери.

За замовчуванням метод `plot` генерує лінійний графік без маркерів у вузлах сітки на площині. Для зображення маркера у кожній точці даних

Табл. 1.15: Найбільш вживані стилі маркерів бібліотеки *Matplotlib*.

Код	Маркер	Опис
.	·	точка
o	○	круг
+	+	знак плюс
x	×	хрест
D	◇	ромб
v	▽	трикутник вершиною до низу
^	△	трикутник вершиною до верху
s	□	квадрат
*	★	зірка

на графіку використовується аргумент `marker`. Можна використовувати маркери різних стилів, повний перелік яких можна знайти за посиланням [48]. Деякі із маркерів, які часто використовуються, наведені у таблиці 1.15.

Кольори.

Колір лінії і/або маркера можна визначити за допомогою аргумента `color`. Бібліотека *Matplotlib* підтримує декілька форматів визначення кольору. По-перше, однобуквенні коди для деяких основних кольорів наведені у таблиці 1.16.

Крім того, можна визначити відтінки сірого кольору у вигляді стрічки, що представляє число із плаваючою комою `float` у діапазоні 0-1 (0 відповідає чорному кольору, 1 – білому). Шістнадцяткові стрічки HTML, які визначають червону, зелену і синю (RGB) компоненти кольору у діапазоні 00-ff, також можуть передаватися аргументом `color` (наприклад, `color='#ff00ff'` визначає фіолетовий (magenta) колір). Крім того, компоненти RGB можна передавати як кортеж (tuple) із трьох значень (відповідають насиченню кожного із основних кольорів) у діа-

Табл. 1.16: Буквенні і стрічкові позначення кольорів *Matplotlib*.

Коди основних кольорів	Живі (Tableau) кольори
b = синій	tab:blue
g = зелений	tab:orange
r = червоний	tab:green
c = бірюзовий	tab:red
m = фіолетовий	tab:purple
y = жовтий	tab:brown
k = чорний	tab:pink
w = білий	tab:gray
	tab:olive
	tab:cyan

пазоні 0-1 (наприклад, `color=(0.5, 0, 0)` – темно червоний колір).

Стили і товщина лінії.

Код	Стиль лінії
-	суцільна
–	штрихована
:	пунктирна
-.	штрихпунктирна

Табл. 1.17: Стили лінії *Matplotlib*.

За замовчуванням прийнято такий стиль графіка: суцільна лінія товщиною 1.5 пункта. Налаштування цього параметра здійснюється за допомогою значення аргумента `linestyle` (значення передається у вигляді стрічки). Деякі можливі значення, які визначають стиль лінії, наведені у [таблиці 1.17](#).

Щоб лінія не відображалася при побудові, встановлюється значення `linestyle=""` (пуста стрічка). Товщину лінії можна задавати у пунктах, встановлюючи значення атрибута `linewidth` типу `float`. Наприклад:

```

import matplotlib.pyplot as plt
import numpy as np
plt.rc('text', usetex=True)
x = np.linspace(0.5, 2., 100)
y1 = 1. / np.sqrt(x)
y2 = 10. * np.exp(-np.sqrt(2) * x)
plt.plot(x, y1, color='r', linestyle=':', linewidth=2., \
         label=r'$\frac{1}{\sqrt{x}}$')
plt.plot(x, y2, color='m', linestyle='-', linewidth=1.5, \
         label=r'$e^{-\sqrt{2}x}$')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16)
plt.title(r'Function Plots $y_1(x)=\frac{1}{\sqrt{x}}$, \
         $y_2(x)=e^{-\sqrt{2}x}$.', fontsize=16)
plt.legend()
plt.savefig('Figure_3')
plt.show()

```

Результат виконання цього коду зображено на [рис. 1.18](#).

Крім того, для визначення властивостей лінії можна користуватися такими скороченнями:

`c` – `color` – колір;

`ls` – `linestyle` – стиль лінії;

`lw` – `linewidth` – товщина лінії.

Наприклад:

```
plt.plot(x, y, c='g', ls='--', lw=2) #потовщена зелена штрихована лінія
```

Дозволяється визначити колір, стиль лінії і маркера в одній стрічці:

```
plt.plot(x, y, 'r:') #червона пунктирна лінія з трикутними маркерами
```

Останнє, можна зобразити декілька ліній, використовуючи послідовність аргументів `x`, `y`, `format`:

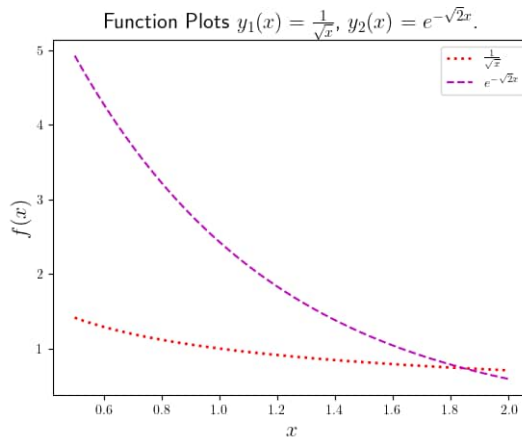


Рис. 1.18: Графіки функцій $y_1(x) = \frac{1}{\sqrt{x}}$, $y_2(x) = e^{-\sqrt{2}x}$.

```
plt.plot(x, y1, 'r--', x, y2, 'k-.')
```

Обчислюється червона штрихована лінія для (x, y_1) і чорна штрихпунктирна лінія для (x, y_2) .

Межі графіка.

Методи `plt.xlim` і `plt.ylim` визначають границі графіка за осями x і y відповідно. Ці методи розташовані після всіх інструкцій `plt.plot`, але перед виведенням і/або збереженням зображення. Наприклад, код нижче обчислює графік за послідовностями даних між вказаними границями (див. [рис. 1.19](#)):

```
import matplotlib.pyplot as plt
import numpy as np
plt.rc('text', usetex=True)
t = np.linspace(0, 2, 1000)
f = t * np.exp(t + np.sin(20*t))
plt.plot(t, f)
plt.xlabel('$X$', fontsize=16)
```

```
plt.ylabel('$Y$', fontsize=16)
plt.title(r'Function Plot $y(x)=xe^{\{x+\sin(20x)\}}$', fontsize=16)
plt.xlim(1.5,1.8)
plt.ylim(0,30)
plt.savefig('Figure_3')
plt.show()
```

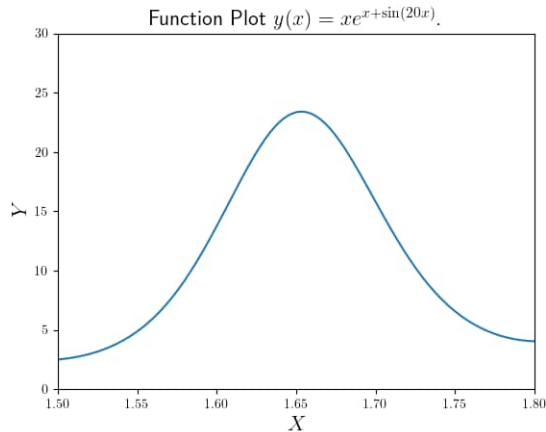


Рис. 1.19: Графіки функції $y(x) = x + e^{x+\sin(20x)}$.

★ **Приклад 1.4.3.** Закон Мооре'а базується на наступному спостереженні: кількість транзисторів у мікросхемах центральних процесорних пристроях (CPU) подвоюється приблизно через кожні 2 роки. Програма `moore_law.py` демонструє дію цього закону за допомогою порівняння між реальною кількістю транзисторів у CPU провідних виробників від 1972 до 2012 р., а після виконується прогнозування за законом Мооре'а, який записується за допомогою формули:

$$n_i = n_0 2^{\frac{y_i - y_0}{T}},$$

де n_0 – кількість транзисторів у деякому році y_0 , прийнятому за ета-

лонний, $T = 2$ – кількість років, необхідних для подвоєння цієї кількості. Так як для дослідження використовуються дані протягом 40 років, то значення n_i охоплює декілька порядків величини, тому вигідно застосувати закон Мооре'а до логарифмів цих величин, що дозволяє показати лінійну залежність від y :

$$\log_{10} n_i = \log_{10} n_0 + \frac{y_i - y_0}{T} \log_{10} 2$$

```
# moores_law.py
import numpy as np
import matplotlib.pyplot as plt
# The data - lists of years:
year = [1972, 1974, 1978, 1982, 1985, 1989, 1993, 1997, 1999, 2000, 2003, \
        2004, 2007, 2008, 2012]
# And number of transistors (ntrans) on CPUs in millions:
ntrans = [0.0025, 0.005, 0.029, 0.12, 0.275, 1.18, 3.1, 7.5, 24.0, 42.0, \
          220.0, 592.0, 1720.0, 2046.0, 3100.0]
# Turn the ntrans list into a NumPy array and multiply by 1 million.
ntrans = np.array(ntrans) * 1.e6
y0, n0 = year[0], ntrans[0]
# A linear array of years spanning the data's years.
y = np.linspace(y0, year[-1], year[-1] - y0 + 1)
# Time taken in years for the number of transistors to double.
T2 = 2.
moore = np.log10(n0) + (y - y0) / T2 * np.log10(2)
plt.plot(year, np.log10(ntrans), '*', markersize=12, color='r',
         markeredgewidth=1, label='observed')
plt.plot(y, moore, linewidth=2, color='b', linestyle='--', label='predicted')
plt.legend(fontsize=16, loc='upper left')
plt.xlabel('Year')
plt.ylabel('log(ntrans)')
plt.title('Moore's law')
plt.savefig('Figure_5')
plt.show()
```

У цьому прикладі дані містяться у двох списках рівної довжини – рік і репрезентативна кількість транзисторів в CPU у цьому році. Наведена формула закону Моогое'а реалізується у логарифмічній шкалі із використанням масиву років, які охоплюють представлені дані. Для побудови графіка, зображеного на [рис. 1.20](#), точки даних зображені у вигляді зірок, а прогноз за законом Моогое'а – штрихованою синьою лінією.

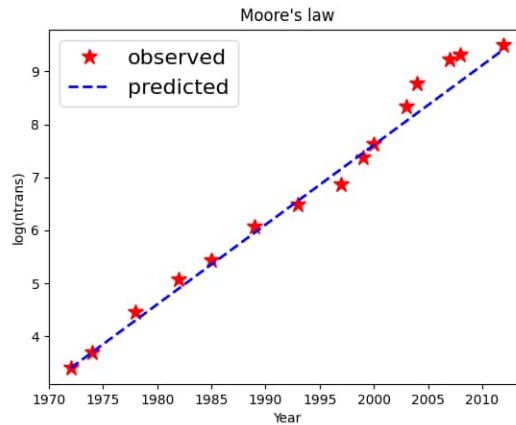


Рис. 1.20: Закон Моогое'а, який моделює експоненціальне зростання кількості транзисторів PCU.

1.4.4. Графік у полярних координатах.

Метод `plt.plot` обчислює графіки у декартовій системі координат (x, y) . Для побудови графіка у полярній системі координат (ρ, θ) використовується метод `plt.polar`, якому передаються аргументи θ (зазвичай є незалежною змінною) і ρ .

На сторінці [132](#) наведений приклад обчислення площі плоскої фігури у полярній системі координат і наведений машинний код побудови

графіка 1.4.

1.4.5. Графік параметрично заданої функції.

Нехай функція $y = f(x)$ задана своїм параметричним поданням на проміжку $[\alpha, \beta]$:

$$\begin{cases} x = \varphi(t), \\ y = \psi(t), \end{cases}$$

де t – параметр із заданого проміжка. Графік параметрично заданої функції обчислюється за допомогою методу `plt.plot` у декартовій системі координат. Для цього на сітці вузлів t_i параметра $t \in [\alpha, \beta]$ обчислюються два масиви: значення функцій $x_i = \varphi(t_i)$ і $y_i = \psi(t_i)$. За допомогою цих масивів $X = \{x_i\}$, $Y = \{y_i\}$, як аргументів метода `plt.plot`, обчислюється графік параметрично заданої функції.

Розглянемо приклад побудови чотирьох арок циклоїди.

★ **Приклад 1.4.4.** На проміжку $[-4\pi, 4\pi]$ побудувати циклоїду, параметричне подання якої

$$\begin{cases} x = t - \sin t, \\ y = 1 - \cos t, \end{cases}$$

Код програми обчислення графіка циклоїди такий:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rc('text', usetex=True)
t = np.linspace(-4*np.pi, 4*np.pi, 1000)
x = t - np.sin(t)
y = 1 - np.cos(t)
plt.plot(x,y, 'r-', lw='3')
plt.title('Cycloid', fontsize='16')
plt.xlabel(r'$x(t)=t-\sin(t)$', fontsize='16')
plt.ylabel(r'$y(t)=1-\cos(t)$', fontsize='16')
plt.show()
```

Результат виконання програми зображений на [рис. 1.21](#).

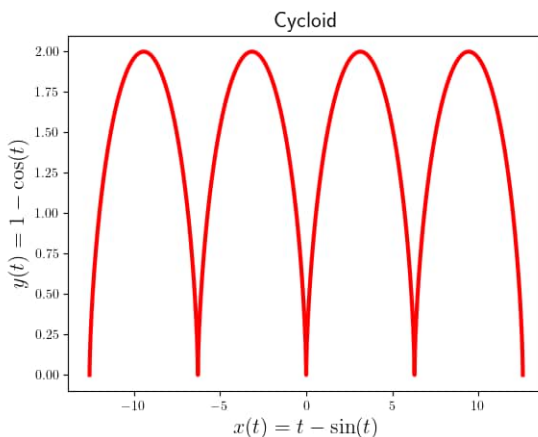


Рис. 1.21: Графік параметрично заданої функції – циклоїди.

1.4.6. Побудова графіка на одному об'єкті осі.

Об'єкт найвищого рівня, який містить всі елементи графіка (зазначимо, що до цього часу розглянуто побудову найпростіший ліній на площині!), називається `Figure`. Для створення об'єкта `Figure` викликається метод `plt.figure`. Його можна викликати без аргумента, але можна задавати додаткові параметри параметри спеціального налаштування, які подаються у [таблиці 1.18](#).

Наприклад, побудуємо найпростіший (без описів самого графіка) графік функції $f(x) = e^{-x \sin^2 x}$, використавши вказані у [таблиці 1.18](#) параметри методу `plt.figure`: назва вікна – 'Population density', розміри вікна – 6×4 дюйми, колір вікна – 'khaki'.

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

Табл. 1.18: Аргументи метода `plt.figure`.

<code>num</code>	ідентифікатор малюнка – якщо значення не задано, то використовується ціле число, починаючи із 1 із кроком 1 для наступних створюваних рисунків. Крім того, використання стрічки встановить заголовок вікна цього рисунку при виводі його за допомогою метода <code>plt.show()</code> ;
<code>figsize</code>	розміри (у дюймах: 1 inch=2.54cm) малюнка у вигляді кортежа (<code>width, height</code>);
<code>dpi</code>	роздільність малюнка у точках на дюйм;
<code>facecolor</code>	колір фону малюнка;
<code>edgecolor</code>	колір межі малюнка.

```
t = np.linspace(0, 2*np.pi, 1000)
x = np.exp(-t*np.sin(t)**2)
plt.figure('Population density', figsize=(6, 4), facecolor='khaki')
plt.plot(t, x, 'g-', lw='3')
plt.show()
```

Результат зображений на [рис. 1.22](#).

Щоб відобразити дані графічно, потрібно створити об'єкт осі `Axes` – область рисунка, яка містить осі, графічні мітки, підписи, лінії, маркери тощо. Найпростіший рисунок, який складається із одного об'єкта `Axes`, створюється і повертається так:

```
ax = fig.add_subplot()
```

В об'єкті `Axes` за допомогою методу `ax.plot` можна дійсно зображати графічні дані. У цьому випадку метод `plot` повертає список (`list`) об'єктів, які графічно представляють лінії. У найпростішому випадку зображається одна лінія, а значить список містить тільки один об'єкт типу `Line2D`, який при необхідності можна присвоїти змінній. Розглянемо приклад, у якому порівнюємо ланцюгову лінію $y = \sinh x$ та її параболічну апроксимацію $y = 1 + x^2/2$.

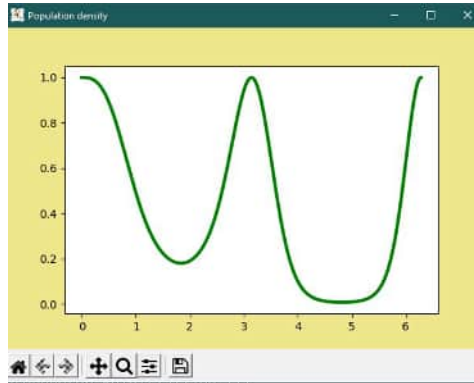


Рис. 1.22: Вікно графіка функції із додатковими параметрами методу `plt.figure`.

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.add_subplot()
x = np.linspace(-2, 2, 1000)
line_cosh, = ax.plot(x, np.cosh(x))
line_quad, = ax.plot(x, 1 + x**2 / 2)
plt.show()
```

Слід звернути увагу на синтаксис `line_cosh, = ...` для присвоєння змінній `line_cosh` повернутого об'єкта лінії, а не списку, що містить цей об'єкт.

Дві лінії, побудовані цим кодом, зображені на [рис. 1.23](#).

Якщо у методі `fig.add_subplot()` не передаємо жодних аргументів, тоді можна створити об'єкти `fig` і `ax` у одній стрічці коду за допомогою зручної функції `plt.subplots()`:

```
fig, ax = plt.subplots()
```

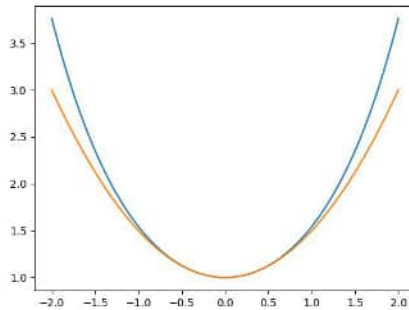


Рис. 1.23: Простий графік із двох ліній в одному об'єкті `Axes`.

Повернемося тепер до графіка [циклоїди 1.21](#). Вигляд графіка не є "натуральним", так як не дотримані пропорції осей – побудовані лінії є розтягнуті вздовж вертикальної осі. Це відбулося так тому, що *Python* за замовчуванням повертає стандартні розміри вікна графіка, тобто співвідношення між осями, які встановлені у властивостях програми. Тобто, щоб графік циклоїди мав властивий вигляд, потрібно встановити співвідношення між масштаби вертикальної і горизонтальної осей як 1:1. Виконаємо це за допомогою методу `ax.set_box_aspect`, тобто змодифікуємо код створення циклоїди так (обчислюються дві арки циклоїди):

```
import numpy as np
import matplotlib.pyplot as plt
plt.rc('text', usetex=True)
t = np.linspace(-2*np.pi, 2*np.pi, 1000)
x = t - np.sin(t)
y = 1 - np.cos(t)
fig, ax = plt.subplots()
ax.plot(x,y, 'r-', lw='3')
plt.title('Cycloid', fontsize='16')
```

```
plt.xlabel(r'$x(t)=t-\sin(t)$', fontsize='16')
plt.ylabel(r'$y(t)=1-\cos(t)$', fontsize='16')
ax.set_box_aspect(2./(4*np.pi))
ax.grid(True, linestyle='-.')
ax.tick_params(labelcolor='r', labelsz='medium', width=3)
plt.savefig('Figure_6')
plt.show()
```

У результаті виконання коду програма поверне графік, зображений на [рис. 1.24](#) (порівняйте із графіком на [рис. 1.21](#))

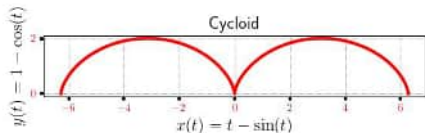


Рис. 1.24: Графік циклоїди.

1.4.7. Декілька графіків на одному рисунку.

Для створення рисунку із декількома графіками (тобто із декількома об'єктами *Axes*) із об'єкта *Figure* викликається метод `add_subplot` (розглядали на стр. 188), аргумент якого визначає, де буде розташований даний графік. Кожне звертання до цього методу повертає об'єкт *Axes*. Рисунки із більш ніж 10 графіків зустрічають рідко, тому аргументом виступає тризначне число, де кожна цифра означає кількість

рядків, кількість стовпців і порядковий номер графіка, починаючи з лівого верхнього кута зліва на право. Наприклад, рисунок, який містить шість графіків, розміщених у трьох рядках та двох стовпцях, можна сформувати так:

```
fig = plt.figure()
ax1 = fig.add_subplot(321) # верхній лівий,
ax2 = fig.add_subplot(322) # верхній правий,
ax3 = fig.add_subplot(323) # середній лівий,
....
ax6 = fig.add_subplot(326) # нижній правий.
```

Інший спосіб створення рисунку із декількома графіками: звертаємося до методу `plt.subplots`, визначивши аргументи `nrows` (визначаємо кількість рядків) і `ncols` (визначаємо кількість стовпців). Метод повертає об'єкт `Figure` і впорядковану таблицю об'єктів `Axes`, яку можна проіндексувати по рядках і стовпцях:

```
>>> fig, axes = plt.subplots(nrows=3, ncols=2)
>>> axes.shape
(3, 2)
>>> ax1 = axes[0, 0] #верхній лівий,
>>> ax2 = axes[2, 1] #нижній правий.
```

★ **Приклад 1.4.5.** Розглянемо металевий брус, площа поперечного перерізу якого дорівнює A , при рівномірно розподіленій температурі Θ_0 . Розглянемо миттєве нагрівання точно у центрі (поперечного перерізу) при передачі деякої кількості енергії H . Температура бруса (стосовно Θ_0), як функція часу t і координати x , визначається як розв'язок одномірного рівняння дифузії:

$$\Theta(x, t) = \frac{H}{c_p A} \frac{1}{\sqrt{Dt}} \frac{1}{\sqrt{4\pi}} e^{-\frac{x^2}{4Dt}},$$

де c_p і D – питома теплоємність металу на одиницю об'єму і коефіцієнт теплопровідності (припускається сталим стосовно температури).

Код нижче створює графіки у трьох заданих часових інтервалах і порівнює графіки для двох металів із різними коефіцієнтами теплопровідності, але рівними питомими теплоємностями – мідь і залізо.

```
import numpy as np
import matplotlib.pyplot as plt
#
A, H = 1.e-4, 1.e3
#
theta0 = 300
#
# ( /m3.K).
# (2/) Cu Fe.
metals = np.array([('Cu', 3.45e7, 1.11e-4), ('Fe', 3.50e7, 2.3e-5)], \
                  dtype=[('symbol', '|S2'), ('cp', 'f8'), ('D', 'f8')])
# -xlim xlim ().
xlim, nx = 0.05, 1000
x = np.linspace(-xlim, xlim, nx)
# .
times = (1e-2, 0.1, 1)
# : ,
# .
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(7, 8))
for j, t in enumerate(times):
    for i, metal in enumerate(metals):
        symbol, cp, D = metal
        ax = axes[j, i]
        # .
        theta=theta0+H/(cp*A*np.sqrt(D*t*4*np.pi))*np.exp(-x**2/4/D/t)
        # .
        ax.plot(x*100, theta , 'k')
        ax.set_title('{ } , $t={ }$ s'.format(symbol.decode('utf8'), t))
```

```
ax.set_xlim(-4, 4)
ax.set_xlabel('$x/\mathrm{cm}$')
ax.set_ylabel('$\Theta/\mathrm{K}$')
# y, t.
for j in (0, 1, 2):
    ymax = max(axes[j, 0].get_ylim()[1], axes[j, 1].get_ylim()[1])
    print(axes[j, 0].get_ylim(), axes[j, 1].get_ylim())
    for i in (0, 1):
        ax = axes[j, i]
        ax.set_ylim(theta0, ymax)
        # y.
        ax.set_yticks([theta0, (ymax + theta0)/2, ymax])
# : tight_layout().
fig.tight_layout()
plt.show()
```

Мідь є кращим провідником тепла у порівнянні із залізом, тому із графіків (зображених на [рис. 1.25](#)) одержуємо, що підвищення температури у цьому металі відбувається швидше.

Контурні графіки.

У модулі `pyplot` метод `contour` обчислює контурні графіки на основі даних двовимірного масиву вузлів. У найпростішому випадку `contour(Z)` не вимагає жодних інших аргументів: значення (x, y) проіндексовані у двовимірному масиві Z , а інтервали контурів вибираються автоматично. Для явного визначення координат (x, y) потрібно передати їх у вигляді `contour(X, Y, Z)`. Масиви X і Y повинні мати таку ж розмірність що і масив Z (наприклад, так як це зроблено при використанні методу `np.meshgrid`, описаного у [підрозділі 1.2.1.](#)), або повинні бути одновимірними масивами, такими, що довжина масиву X дорівнює кількості стовпців масиву Z , а довжина масиву Y повинна дорівнювати кількості рядків масиву Z .

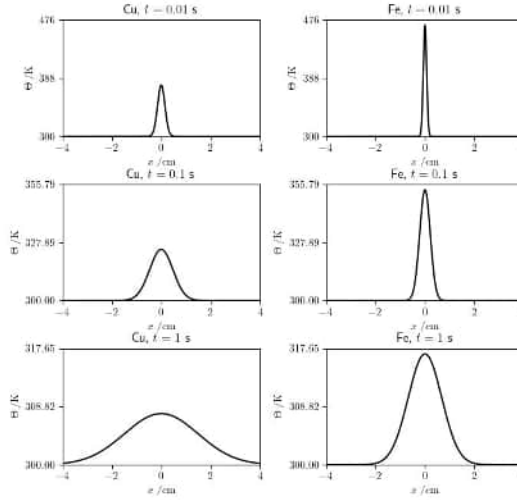


Рис. 1.25: Графіки поширення тепла у двох металах.

Рівнями контурів можна керувати за допомогою додаткового аргумента: або скалярне значення N , яке визначає загальну кількість рівнів контурів, або послідовності V , яка явно задає значення Z , для яких потрібно відобразити контурні лінії.

Кольори контурних ліній визначаються у відповідності із картою кольорів (colormap) *Matplotlib*, яка приймається за замовчуванням. У цьому процесі дані лінійно нормовані на проміжку $[0, 1]$, який у подальшому відображається у список кольорів, які використовуються для визначення стилів контурів стосовно відповідних значень. Модуль `matplotlib.cm` містить декілька схем карт кольорів⁵. Найчастіше на практиці використовують такі схеми: `cm.viridis`, `cm.hot`, `cm.bone`, `cm.winter`, `cm.jet`, `cm.Greys`, `cm.hsv`. Якщо виникає необхідність використання схем кольорів із зміною на протилежні кольори, то в кінці імені схеми потрібно додати суфікс `_r` (наприклад, `cm.hot_r`).

Метод `contour` підтримує інший спосіб визначення кольорів – в аргу-

⁵З повним списком можна ознайомитися за посиланням на веб-сторінку [49]

менті `colors` передається або один специфікатор кольору *Matplotlib*, або послідовність таких специфікаторів. На одноколірних контурних графіках лінії рівня, які відповідають від'ємним значенням функції, зображаються штрихованими лініями. Для товщини ліній контурів можна визначати стилі для окремо взятої або для всіх ліній разом за допомогою аргумента `linewidths`.

★ **Приклад 1.4.6.** Покажемо на прикладі побудову ліній рівня гіперболічного параболоїда $f(x, y) = x^2 - y^2$, щоб продемонструвати від'ємні і додатні рівні. Результатом виконання програми побудови контурного графіка, записаної нижче, є [рис. 1.26](#).

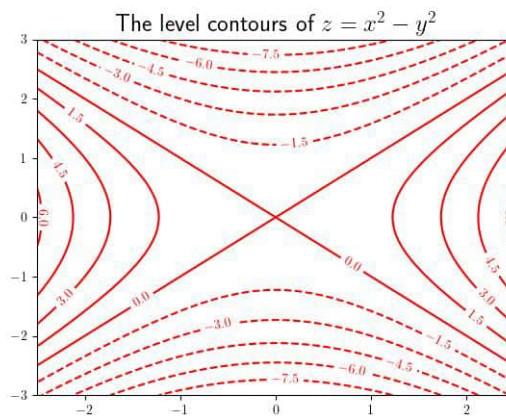


Рис. 1.26: Лінії рівня гіперболічного параболоїда.

```
import numpy as np
import matplotlib.pyplot as plt
plt.rc('text', usetex=True)
fig = plt.figure()
ax = fig.add_subplot(111)
[X,Y] = np.mgrid[-2.5:2.5:51j,-3:3:61j]
```

```
Z=X**2-Y**2
curves=ax.contour(X,Y,Z,12,colors='r',linewidths=1.5)
ax.clabel(curves)
plt.title(r'The level contours of  $z=x^2-y^2$ ',fontsize=18)
plt.show()
```

Щоб додати підписи до контурного графіка, потрібно зберегти об'єкт `Contour-Set`, який повертається після виклику методу `ax.contour`, і передати цей об'єкту у метод `ax.label` (можливо з деякими додатковими параметрами, які визначають властивості шрифту). Ще один метод `ax.contourf` приймає ті ж аргументи, що `ax.contour`, але зображає замальовані контури (тобто замальовує області контурів). Як показано у прикладі нижче, методи `ax.contour` і `ax.contourf` можна використовувати одночасно.

★ *Приклад 1.4.7.* Приведена нижче програма створює графік функції із замальованими контурами, додає підписи для контурів і визначає деякі стилі для кольорів контурів (див. [рис. 1.27](#)).

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
X = np.linspace(0, 1, 100)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
alpha = np.radians(25)
cX, cY = 0.5, 0.5
sigX, sigY = 0.2, 0.3
rX = np.cos(alpha) * (X-cX) - np.sin(alpha) * (Y-cY) + cX
rY = np.sin(alpha) * (X-cX) + np.cos(alpha) * (Y-cY) + cY
Z = (rX-cX)*np.exp(-((rX-cX)/sigX)**2) * np.exp(-((rY-cY)/sigY)**2)
fig = plt.figure()
ax = fig.add_subplot()
#
cpf = ax.contourf(X, Y, Z, 20, cmap=cm.YlOrRd_r)
```

```
#  
#  
colors = ['y' if level < 0 else 'm' for level in cpf.levels]  
cp = ax.contour(X, Y, Z, 20, colors=colors)  
ax.clabel(cp, fontsize=12, colors=colors)  
plt.show()
```

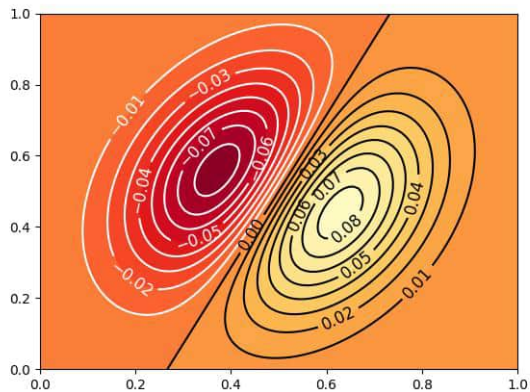


Рис. 1.27: Графік ліній рівня з підписами.

1.4.8. Тривимірна графіка у *Python*.

Бібліотека *Matplotlib* головним чином призначена для створення двовірних графіків, але вона підтримує і функціональні можливості для побудови тривимірних графіків, які цілком придатні для багатьох для багатьох цілей і завдань. Найпростіший спосіб створення тривимірного графіка – імпорт об'єкта *Axes3D* із модуля *mpl_toolkits.mplot3d* і присвоєння для аргумента *projection* внутрішнього графіка значення *'3d'*:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
```

Після цього відповідний об'єкт `Axes` може відображати дані у трьох вимірах як лінійний графік, сукупність точок у просторі, каркасну поверхню, об'ємну поверхню.

Каркасні і об'ємні поверхні.

Каркасна поверхня є найпростішим типом об'ємної поверхні, яка зображає лінії у тривимірному просторі, об'єднуючи двовимірний масив точок Z із сіткою значень даних, які передаються у двовимірних масивах X і Y . Іншими словами, каркасний графік утворюється як сукупність ліній перетину даної поверхні із площинами, перпендикулярними до осей OX і OY і проходять через вузлові точки на відповідних осях. За замовчуванням лінії у тривимірному просторі зображаються для кожної точки масиву, але у випадку, якщо кількість точок достатньо велика, то за допомогою задання значень аргументам `rstride` і `cstride` визначаємо кількість точок у рядку (`rstride (row)`) і стовпці (`cstride (column)`) відповідно.

Метод `ax.plot_surface` створює просторову поверхню із замальованих клаптиків (патчів). Для патчів можна встановити єдиний колір у аргументі `color` або встановити стиль за допомогою спеціальної кольорової схеми в аргументі `cmap`. Для методу `ax.plot_surface` аргументам `rstride` і `cstride` за замовчуванням присвоєне значення 10. Розглянемо приклад застосування обидвох згаданих вище методів.

★ **Приклад 1.4.8.** Код нижче демонструє використання параметрів для створення різних графіків поверхонь. Результат виконання зображений на [рис. 1.28](#).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import matplotlib.cm as cm
L, n = 2, 400
x = np.linspace(-L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(-(X**2 + Y**2))
fig, ax = plt.subplots(nrows=2, ncols=2, subplot_kw={'projection': '3d'})
ax[0, 0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0, 1].plot_surface(X, Y, Z, rstride=40, cstride=40, color='m')
ax[1, 0].plot_surface(X, Y, Z, rstride=12, cstride=12, color='m')
ax[1, 1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([-2, -1, 0, 1, 2])
    axes.set_yticks([-2, -1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
plt.show()
```

★ **Приклад 1.4.9.** Параметричне задання тора із головним радіусом c і радіусом твірного кола a записується так:

$$\begin{cases} x = (c + a \cos \theta) \cos \varphi, \\ y = (c + a \cos \theta) \sin \varphi, \\ z = a \sin \theta. \end{cases}$$

Значення аргументів θ і φ належать до $[0, 2\pi]$. Код, наведений нижче, повертає два графіка тора, які відрізняються різними кутами спостереження, зображені на [рис. 1.29](#).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
98797
n = 100
```

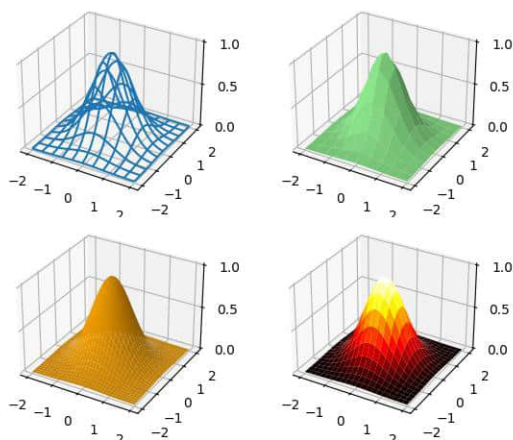


Рис. 1.28: Графіки однієї функції, побудовані різними способами.

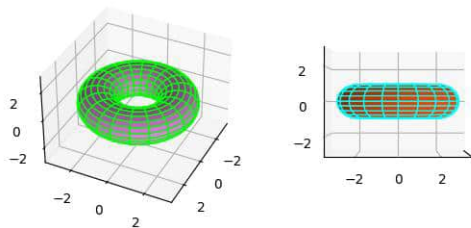


Рис. 1.29: Графіки тора із різними кутами спостереження.

```
theta = np.linspace(0, 2.*np.pi, n)
phi = np.linspace(0, 2.*np.pi, n)
theta, phi = np.meshgrid(theta, phi)
c, a = 2, 1

x = (c + a*np.cos(theta)) * np.cos(phi)
y = (c + a*np.cos(theta)) * np.sin(phi)
z = a * np.sin(theta)
fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.set_zlim(-3, 3)
ax1.plot_surface(x, y, z, rstride=5, cstride=5, color='violet',
edgecolors='lime')
ax1.view_init(36, 26)
ax2 = fig.add_subplot(122, projection='3d')
ax2.set_zlim(-3, 3)
ax2.plot_surface(x, y, z, rstride=5, cstride=5, color='orangered',
edgecolors='cyan')
ax2.view_init(0, 0)
ax2.set_xticks([])
plt.show()
```

Лінійні і точкові просторові графіки.

Лінійні і точкові (графіки масиву точок) графіки у тривимірному просторі обчислюються таким самим способом, як у двовимірному випадку, за допомогою основних методів `ax.plot(x, y, z)` і `ax.scatter(x, y, z)` відповідно, де `x`, `y`, `z` – одновимірні масиви однакової довжини. Для таких графіків можлива обмежена кількість опцій, якщо не користуватися методами для розширення функціональних можливостей.

★ **Приклад 1.4.10.** Нижче наведений код програми побудови гвинтової лінії спіралі, яка повертає тривимірний графік лінії, зображений на [рис. 1.30](#).

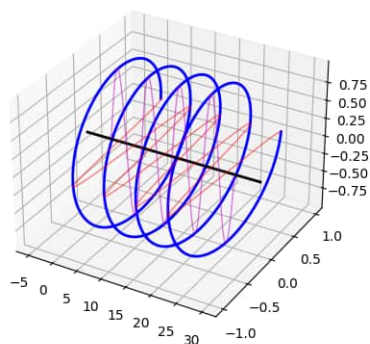


Рис. 1.30: Тривимірний графік спіралі.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
n = 1000
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

theta_max = 8 * np.pi
theta = np.linspace(0, theta_max, n)
x = theta
z = np.sin(theta)
y = np.cos(theta)
ax.plot(x, y, z, 'b', lw=2)
ax.plot((-theta_max*0.2, theta_max * 1.2), (0, 0), (0, 0), color='k', lw=2)
ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color='m', lw=1, alpha=0.5)
plt.show()
```

1.5. Застосування *Python* до розв'язування моделей

У цьому підрозділі розглянемо моделі, що описуються початковими задачами для звичайних диференціальних рівнянь. Усі моделі мають конкретне практичне застосування і зазвичай їх використовують для розв'язування задач оптимального керування. Детальніше про моделі цього підрозділу читач знайде у посібнику авторів [7]. Тут ці задачі адаптовані до програмного середовища *Python* і використовуються пакети і методи *Python*, описані у попередніх підрозділах.

1.5.1. Модель популяції молі spruce budworm

Розглянемо модель поширення популяції молі, яка описується початковою задачею для звичайного диференціального рівняння [25, Розділ 2.3], [17]. Ця моль є комахою, яка нищить ліси північної Америки; пожива для неї – голки хвойних дерев. Нехай $N(t)$ кількість особин у момент часу $t \in [0, T]$. Поширення популяції можна описати за допомогою моделі

$$N'(t) = rN(t) \left(1 - \frac{N(t)}{K} \right),$$

яка означає, що поширення популяції не тільки залежить від природної народжуваності та смертності, а також від ступеня смертності, зумовленого швидким розмноженням. У наведеному вище рівнянні r і K масштабовані параметри, зокрема r є природним коефіцієнтом росту і K виражає властивості середовища. Додатково до коефіцієнта вимірювання, який відповідає перенаселенню, $(r/K)N(t)$, додамо доданок, який відповідає хижакам. Наприклад, хижаків репрезентують птахи. Після модифікації рівняння запишеться у вигляді

$$N'(t) = rN(t) \left(1 - \frac{N(t)}{K} \right) - p(N(t)), \quad t \in [0, T],$$

де

$$p(N) = \frac{BN^2}{A^2 + N^2}.$$

Графік функції $p(N)$ для $B = 1.5$, $A = 2$ на проміжку $[0, L]$ для $L = 20$, побудований за допомогою програми:

```
import matplotlib.pyplot as plt
import numpy as np
plt.rc('text', usetex=True)
b=1.5
a=2
N=np.linspace(0,20,2000)
p=(b*N**2)/(a**2+N**2)
plt.plot(N,p,'r-', linewidth=3)
plt.xlabel('$N$', fontsize=16)
plt.ylabel('$p(N)$', fontsize=16)
plt.title(r'Function plot $p(N)= \frac{BN^2A^2+N^2$ \
          for $A=2$, $B=1.5$', fontsize=16)
plt.grid(True, linestyle='-.')
plt.show()
```

виглядає як на [рис. 1.31](#).

Розглянемо тепер початкову задачу для рівняння поширення популяції

$$\begin{cases} N'(t) = rN(t) \left(1 - \frac{N(t)}{K}\right) - \frac{BN^2(t)}{A^2 + N^2(t)}, & t \in [0, T], \\ N(0) = N_0. \end{cases}$$

Програма обчислення наближеного розв'язку початкової задачі така:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
plt.rc('text', usetex=True)

r1, r2 = 0.3, 1.
K1, K2 = 5., 10
A1, A2 = 2., 3
B1, B2 = 1.5, 2
N01, N02 = 30, 50
T0=0
T=20

def rhsEq(t, N, r, K, A, B):
    res = r*N*(1-N/K)-B*N**2/(A**2+N**2)
    return res

t=np.linspace(T0, T, 200)
```

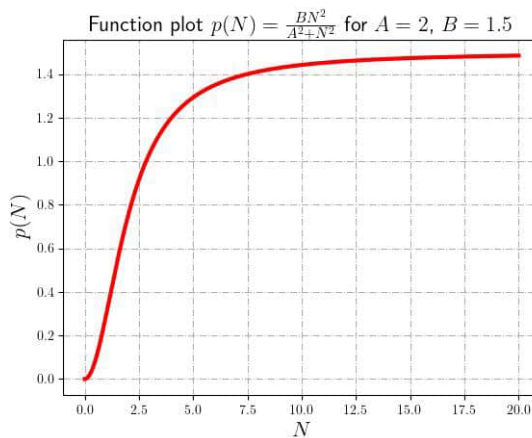


Рис. 1.31: Графік $p(N)$ на проміжку $[0, 20]$ для $B = 1.5$, $A = 2$.

```

soln1 = solve_ivp(rhsEq, (T0, T), [N01], dense_output=True, \
                  args=(r1, K1, A1, B1))
soln2 = solve_ivp(rhsEq, (T0, T), [N02], dense_output=True, \
                  args=(r2, K2, A2, B2))

N1 = soln1.sol(t)[0]
N2 = soln2.sol(t)[0]

plt.plot(t, N1, 'r-', linewidth=2)
plt.grid(True)
plt.title(r' $r=0.3, \; K=5, \; A=2, \; B=1.3, \; N(0)=30.$', fontsize=20)
plt.xlabel(r'$t$', fontsize=20)
plt.ylabel(r'$N(t)$', fontsize=20)
plt.savefig('Figure_28')
plt.show()

plt.plot(t, N2, 'g-', linewidth=2)
plt.grid(True)
plt.title(r' $r=1, \; K=10, \; A=3, \; B=2, \; N(0)=50.$', fontsize=20)
plt.xlabel(r'$t$', fontsize=20)
plt.ylabel(r'$N(t)$', fontsize=20)
plt.savefig('Figure_29')
plt.show()

```

Динаміку поширення популяції моли spruce budworm для різних значень параметрів зображено на [рис. 1.32](#).

★ **Приклад 1.5.1.** Дослідити початкову задачу, яка є моделлю поширення популяції комах (подібно як для моделі поширення популяції моли spruce budworm)

$$\begin{cases} N'(t) = rN(t) \left(1 - \frac{N(t)}{K}\right) - u(t)N(t), & t \in [0, T], \\ N(0) = N_0. \end{cases}$$

Значення параметрів моделі взяти такі самі, як у моделі розмноження

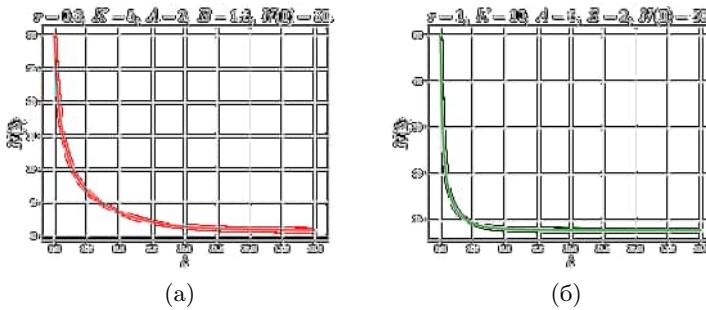


Рис. 1.32: Графіки $N(t)$ за різних вхідних даних: а) $r = 0.3$, $K = 5$, $A = 2$, $b = 1.5$, $T = 20$, $N(0) = 30$; б) $r = 1$, $K = 10$, $A = 3$, $b = 2$, $T = 10$, $N(0) = 50$.

молі spruce budworm, а саме: $r = 0.3$, $K = 5$, $T = 0$, $N_0 = 30$. Обчислити графіки поширення популяції $N(t)$ за таких значень плодючості $u(t)$:

- (i) $u(t) = 0.5$ на проміжку $[0, T]$;
- (ii) $u(t) = 1$ на проміжку $[0, T]$;
- (iii) $u(t) = 2$ на проміжку $[0, T]$;
- (iv) $u(t) = 3$ на проміжку $[0, T]$.

У всіх випадках обчислити популяцію за формулою $\int_0^T u(t)N(t)dt$.

Програма обчислення популяції комах запишеться так:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.integrate import quad
plt.rc('text', usetex=True)

r, K = 0.3, 5.
N0 = 30
T0, T = 0, 20
```

```
def rhsEq(t, N, r, K, u):
    res = r*N*(1-N/K)-u*N
    return res

t=np.linspace(T0, T, 200)
contr = (0.5, 1, 2, 3)
clr = ('brown', 'blueviolet', 'mediumvioletred', 'teal')
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6, 7))
k=0
for i in range(2):
    for j in range(2):
        ax = axes[i,j]
        u = contr[k]
        soln = solve_ivp(rhsEq, (T0, T), [N0], dense_output=True, \
                        args=(r, K, u))
        ax.plot(t, soln.sol(t)[0], clr[k], lw=2)
        ax.grid(True)
        nt = quad(lambda t: u*soln.sol(t)[0], 0, T)
        ax.set_title('u={}, \n $ \int \limits_0^T uN(t)dt=${}'. \
                    format(u, round(nt[0], 2)), fontsize=16)
        ax.set_xlabel(r'$t$', fontsize=16)
        ax.set_ylabel(r'$N(t)$', fontsize=16)
        k+=1

fig.tight_layout()
plt.savefig('Figure_30')
plt.show()
```

Графіки динаміки поширення популяції комах у залежності від значення параметра плодючості u , як розв'язки початкової задачі, та обчислення чисельності популяції, зображені на [рис. 1.33](#).

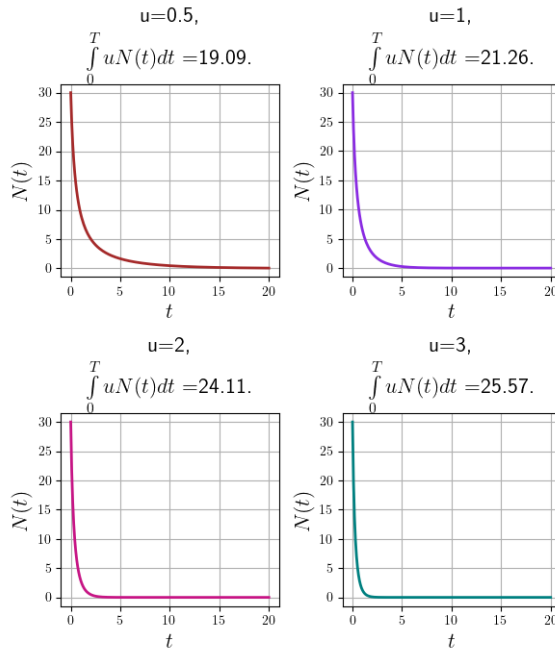


Рис. 1.33: Графіки приплоду комах для $r = 0.3$, $K = 5$, $T = 20$, $N(0) = 30$, для значень $u = 0.5, 1, 2, 3$. Обчислені значення популяції комах в залежності від u .

1.5.2. Системи звичайних диференціальних рівнянь. Моделі виживання.

У цьому підрозділі розглянемо біологічну модель, яка описується системою n звичайних диференціальних рівнянь із n невідомими функціями.

Модель хижак-жертва

Розглянемо модель, в якій спостерігається перенаселення популяції жертви [18, Розділ 5.4], [36, Розділ 14]. Тобто, апетит хижаків задоволений і популяція жертви зростає. Позначимо популяцію хижаків у момент часу t через $y_1(t)$, а кількість особин жертви – $y_2(t)$. Тоді зміна у системі *хижак-жертва* описується такою системою звичайних диференціальних рівнянь:

$$\begin{cases} y_1' = -ay_1 + \frac{by_2}{c + ky_2}y_1, \\ y_2' = (d - ey_2)y_2 - \frac{fy_2}{c + ky_2}y_1, \end{cases} \quad (1.5.1)$$

для $t \in [0, L]$. Тут a, b, c, d, e, f суть невід'ємні сталі величини. Додатний параметр k означає граничне перенасичення хижаків. Малі значення k означають, що хижаки жеребкують жертви перед кожним своїм насиченням. Велике значення k означає, що перенасичення швидко з'являється, як тільки збільшується кількість жертв. Для розв'язку системи (1.5.1) сформулюємо такі початкові умови:

$$y_1(0) = 0.5, \quad y_2(0) = 1. \quad (1.5.2)$$

Для нашого чисельного експерименту розглянемо такі значення параметрів: $a = 0.5$, $b = d = e = f = 1$, $c = 0.3$, $k = 0.7$, кінцевий момент часу $L = 200$. Сформулюємо завдання:

а) знайти чисельний розв'язок задачі (1.5.1)-(1.5.2) при зазначених вище значеннях параметрів системи;

б) побудувати у одному вікні динаміку зміни чисельності хижаків, жертв, залежність популяції хижаків від жертв (динаміка у фазовій площині), на одному графіку зміну популяцій хижак-жертва.

Отож, використовуючи внутрішні сольвери (методи розв'язування) середовища *Python*, код програми має вигляд:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.integrate import quad
plt.rc('text', usetex=True)
dataPPM = open('Model_Prey_Predator.txt')
dataR=dataPPM.readlines()
data1F=map(float, dataR)
data1F=tuple(data1F)
a, b, c, d, e, f, k = data1F
dataPPM.close()

def rhssyst(t, y, a, b, c, d, e, f, k):
    """ Return dy_i/dt = f(y_i, t) at time t. """
    y1, y2 = y
    rhs1eq = -a * y1 + (b*y1*y2)/(c+k*y2)
    rhs2eq = (d - e*y2)*y2 - (f*y1*y2)/(c+k*y2)
    return rhs1eq, rhs2eq

y0=(0.5, 1)
L=200
clr = ('mediumvioletred', 'teal')
ttl=('Predator', 'Prey')
soln = solve_ivp(rhssyst, [0,L], y0, dense_output=True, \
                 args=(a, b, c, d, e, f, k))

tspan=np.linspace(0,L,1000)
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 4) )
for j in range(2):
    ax=axes[0,j]
    ax.grid(True)
    ax.plot(tspan, soln.sol(tspan)[j], clr[j], lw = 2 )
    ax.set_title(ttl[j], fontsize=16)
    ax.set_xlabel(r'$t$', fontsize=16)
    ax.set_ylabel(r'$y_{\{j\}}(t)$'.format(j+1), fontsize=16)
```

```

axes[1,0].plot(soln.sol(tspan)[1], soln.sol(tspan)[0], 'r-', lw = 1 )
axes[1,0].grid(True)
axes[1,0].set_title('Trajectory in the Phase Plane ', fontsize=16)
axes[1,0].set_xlabel(r'$y_2$', fontsize=16)
axes[1,0].set_ylabel(r'$y_1$', fontsize=16)
axes[1,1].plot(tspan, soln.sol(tspan)[1], tspan, soln.sol(tspan)[0], \
                'mediumvioletred', 'teal', lw = 1 )
axes[1,1].grid(True)
axes[1,1].set_title('Predator - Prey', fontsize=16)
axes[1,1].set_xlabel(r'$t$', fontsize=16)
axes[1,1].set_ylabel(r'$y_1, y_2$', fontsize=16)
fig.tight_layout()
plt.savefig('Figure_31')
plt.show()

```

Результат виконання програми зображений [рис. 1.34](#).

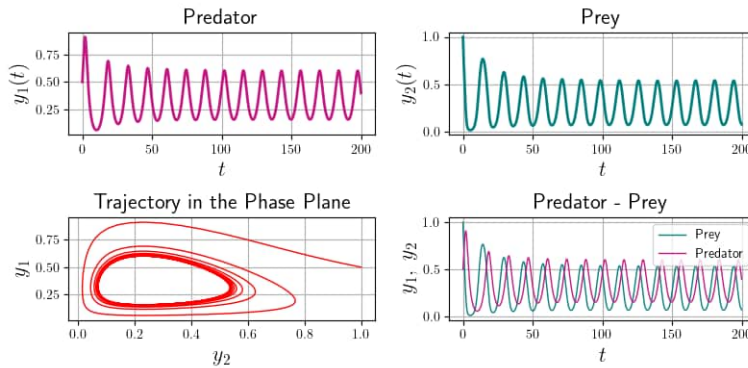


Рис. 1.34: Графіки зміни динаміки популяцій моделі хижак-жертва.

Прокоментуємо програму чисельного розв'язування задачі (1.5.1)-(1.5.2) хижак-жертва. Значення параметрів задачі a, b, c, d, e, f, k за-

Табл. 1.19: Режими доступу до файлу.

Аргумент mode	Режим доступу при відкриванні файлу
r	текстовий, тільки для читання;
w	текстовий, записування (існуючий файл із тим же іменем буде перезаписаний);
a	текстовий, доповнення існуючого файлу;
r+	текстовий, читання і записування;
rb	бінарний, тільки для читання;
wb	бінарний, записування (існуючий файл із тим же іменем буде перезаписаний);
ab	бінарний, доповнення існуючого файлу;
rb+	бінарний, читання і записування.

вантажуються із текстового файлу `Model_Prey_Predator.txt`, який міститься у робочій директорії середовища. У цьому файлі кожна числова величина записана в окремому рядку. Цей текстовий файл може бути створений редактором *Python* або кожним іншим текстовим процесором OS (Operating System). Команда `open` відкриває файл для його подальшого опрацювання.

Об'єкт типу `file` створюється при відкриванні файлу із заданим іменем (`filename`) і режимом доступу (`mode`). Ім'я файлу може бути задане як абсолютне ім'я шляху, який визначається стосовно каталога, у якому виконується програма. Режим доступу `mode` – це стрічка, яка містить одне із значень, приведених у [таблиці 1.19](#).

Отож, стрічкою коду

```
dataPPM = open('Model_Prey_Predator.txt')
```

відкрили текстовий файл (за замовчуванням для читання – не вказали другого аргумента для `open`), що міститься у робочому каталозі, у якому записані порядково числові значення параметрів задачі, і присвоїли цей об'єкт змінній `dataPPM`. Метод `readlines` зчитує всі рядки

`dataPPM` (тобто вміст текстового файлу) і створює список цих рядків як стрічок разом із командами форматування (перехід на нову стрічку тощо). Тому, щоб одержати числа, які записані у вихідному текстовому файлі, потрібно змінити об'єкти `string` списку на об'єкти `float`. Це виконано за допомогою команди `map` інструкції `float` на елементи списку `dataR`. Наступний крок – створення кортежа, елементами якого є числові значення, які на наступному кроці присвоюються параметрам `a`, `b`, `c`, `d`, `e`, `f`, `k`. Цей спосіб визначення числових значень параметрів довільної досліджуваної задачі є вигідним тим, що змінюючи числові значення параметрів чи файл-носії нових, інших значень, дає можливість легко впроваджувати нові дані і тим самим полегшує проведення чисельних експериментів задачі. Наприклад, створивши файл `Model_Prey_Predator_2.txt` із числовими даними

```
0.2  
0.8  
0.5  
0.9  
1.1  
0.9  
0.5
```

і замінивши у програмі ім'я текстового файлу `Model_Prey_Predator.txt` на `Model_Prey_Predator_2.txt`, після компіляції одержимо результат, зображений на [рис. 1.35](#):

Тобто, із одержаних результатів маємо висновок, що при значеннях параметрів моделі $a=0.2$, $b=0.8$, $c=0.5$, $d=0.9$, $e=1.1$, $f=0.9$, $k=0.5$ існує залежність між величиною популяції хижаків і жертв, але у другій половині досліджуваного часового інтервалу чисельність популяцій стабілізується, причому популяція хижаків є більшою від популяції жертв.

Наступний блок у програмі – визначення правої частини системи диференціальних рівнянь (1.5.1). Слід звернути увагу, що параметри `a`, `b`, `c`, `d`, `e`, `f`, `k` вписали як аргументи функції правої системи рівняння. Проте, застосовуючи далі у програмі метод `solve_ivp`, додатковим па-

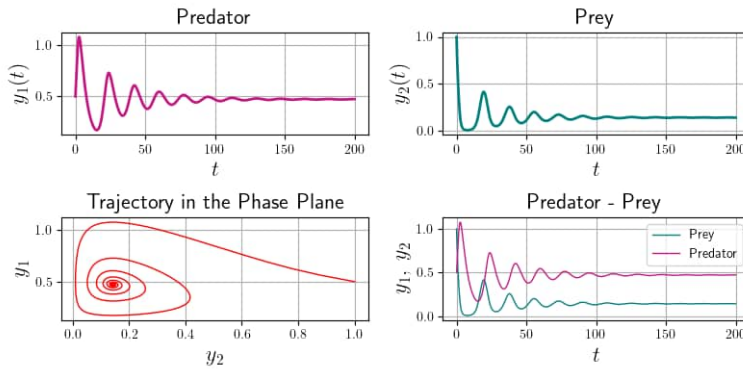


Рис. 1.35: Графіки зміни динаміки популяцій моделі хижак-жертва із даними з файла `Model_Prey_Predator_2.txt`.

раметром вказано, що ці аргументи насправді є сталими параметрами системи, які замінюються на відповідні числові значення.

І накінець, метод `fig.tight_layout()` об'єкта `Figure` використаний для того, щоб графіки, побудовані в одному вікні і підписи до графіків, на накладалися.

Модель нейронної активності. Рівняння Fitzhugh–Nagumo

Рівняння *Fitzhugh–Nagumo* описує електричну активність нейрона (див. [31, Розділ 6.5], [36, Розділ 14]). Нейрон є збуджуючою системою, яка стимулюється зовнішніми чинниками, наприклад, електричним розрядом. Стан збудження описується функцією y_1 , яка означає напругу у нейроні як функція часу. Коли нейрон збуджений, фізіологічні процеси змушують відновлюватися після збудження. Відновлення описується

функцією y_2 . Запишемо систему

$$\begin{cases} y_1' = c \left(y_1 + y_2 - \frac{y_1^3}{3} \right), \\ y_2' = \frac{1}{c} (a - y_1 - by_2), \end{cases} \quad (1.5.3)$$

при $t \in [0, L]$. Можна зауважити два явища у нейронах:

- реакція y_1 нейрона приводить до стійкого стану після сильного збудження; нейрон збуджується; це одинарний потенціал дії;
- реакція (відгук) y_1 – періодична функція; нейрон отримує періодичні імпульси.

Параметри a , b , c задовольняють такі обмеження, які виражають поведінку

$$1 - \frac{2}{3}b < a < 1, \quad 0 < b < 1, \quad b < c^2.$$

У чисельних експериментах візьмемо такі значення параметрів $a = 0.75$, $b = 0.5$, $c = 1$. Запишемо їх у текстовому файлі `fileFN.txt`. Початкові дані виберемо такі:

$$y_1(0) = 3, \quad y_2(0) = 0. \quad (1.5.4)$$

Аналогічно, як при розв'язуванні задачі у [підрозділі 1.5.2.](#), утворимо у робочому каталозі текстовий файл `Model_NeuroActivity.txt` із значеннями параметрів $a = 0.75$, $b = 0.5$, $c = 1$. Тоді програма знаходження чисельного розв'язку задачі нейронної активності (1.5.3)-(1.5.4) запишеться так:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
plt.rc('text', usetex=True)

dataNAM = open('Model_NeuronActivity.txt')
rdData=dataNAM.readlines()
mpData=map(float, rdData)
mpData=tuple(mpData)
```

```
a, b, c = mpData
dataNAM.close()

def rhsSyst(t, y, a, b, c):
    """ Return dy_i/dt = f(y_i, t) at time t. """
    y1, y2 = y
    rhs1eq = c*(y1 + y2 - y1**3/3)
    rhs2eq = (1/c)*(a - y1 - b*y2)
    return rhs1eq, rhs2eq

y0=(3, 0)
L=20
clr = ('orangered', 'darkmagenta')
ttl=('Excitation', 'Recovery')
soln = solve_ivp(rhsSyst, [0,L], y0, dense_output=True, args=(a, b, c))
tspan=np.linspace(0,L,200)

fig, ax = plt.subplots()
ax.plot(tspan, soln.sol(tspan)[0], clr[0], label=ttl[0], lw = 2 )
ax.plot(tspan, soln.sol(tspan)[1], clr[1], label=ttl[1], lw = 2 )
ax.grid(True)
ax.set_title(r'Excitation-Recovery Plot: $a=${}$, $b=${}$, $c=${}$.' \
            format(a, b, c), fontsize=16)
ax.set_xlabel(r'$t$', fontsize=16)
ax.set_ylabel(r'$y_1, y_2$', fontsize=16)

plt.legend()
plt.savefig('Figure_32')
plt.show()
```

Після виконання програма поверне графічне зображення розв'язку (див. [рис. 1.37](#)).

Уведемо у модель подразник, визначений залежною від часу функ-

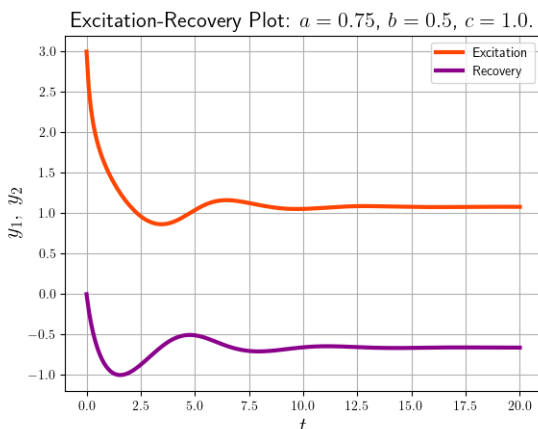


Рис. 1.36: Графіки станів збудження та відновлення

цією z

$$\begin{cases} y_1' = c \left(y_1 + y_2 - \frac{y_1^3}{3} \right) - z, \\ y_2' = \frac{a - y_1 - by_2}{c}, \end{cases} \quad (1.5.5)$$

для $t \in [0, L]$. Для числових розрахунків моделі приймемо

$$z(t) = \begin{cases} 0, & \text{для } t \in [0, t^*], \\ v & \text{для } t \in [t^*, L], \end{cases} \quad (1.5.6)$$

де $0 < t^* < L$ є точкою перемикання і $0 < v < 1$ – величина стимулювання. Змодифікуємо програму обчислення розв'язку задачі (1.5.3)-(1.5.4), увівши у праву частину системи функцію z , яка моделює подразник нейрона, та визначивши саму функцію. Тоді програма обчислення задачі (1.5.5)-(1.5.4) запишеться так:

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

```
from scipy.integrate import solve_ivp
plt.rc('text', usetex=True)
dataNAM = open('Model_NeuronActivity_2.txt')
rdData=dataNAM.readlines()
mpData=map(float, rdData)
mpData=tuple(mpData)
a, b, c, v, astrt= mpData
dataNAM.close()

def stmF(x, v, astrt):
    c1=0
    c2=v
    r=np.where(x < astrt, c1, c2)
    return r

def rhsSyst(t, y, a, b, c, v, astrt):
    """ Return dy_i/dt = f(y_i, t) at time t."""
    y1, y2 = y
    rhs1eq = c*(y1 + y2 - y1**3/3) - stmF(t, v, astrt)
    rhs2eq = (a - y1 - b*y2)/c
    return rhs1eq, rhs2eq

y0=(3, 0)
L=100
clr = ('firebrick', 'mediumblue')
ttl=('Excitation', 'Recovery')
soln = solve_ivp(rhsSyst, [0,L], y0, dense_output=True, \
                 args=(a, b, c, v, astrt))

tspan=np.linspace(0,L,1000)
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
ax[0].plot(tspan, soln.sol(tspan)[0], clr[0], label=ttl[0], lw = 3)
ax[0].plot(tspan, soln.sol(tspan)[1], clr[1], label=ttl[1], lw = 3)
ax[0].grid(True)
ax[0].set_title(r'The Excitation and Recovery \\ (with stimulus): \\
```

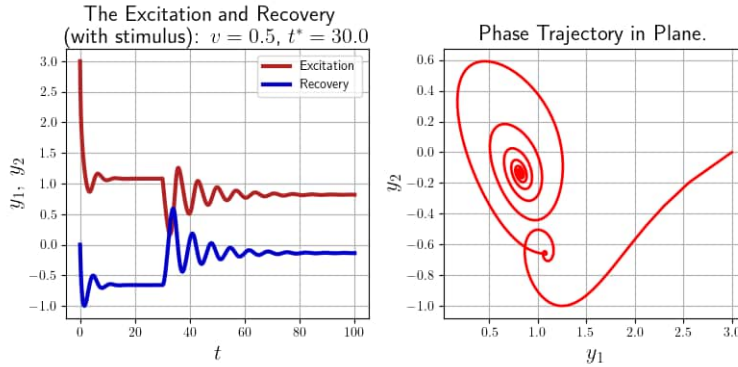


Рис. 1.37: Графіки станів збудження та відновлення у випадку подра-
зника та графік у фазовій площині.

```

    $v={}$, $t^*={}$'.format(v, astrt), fontsize=16)
ax[0].set_xlabel(r'$t$', fontsize=16)
ax[0].set_ylabel(r'$y_1, y_2$', fontsize=16)
ax[1].plot(soln.sol(tspan)[0], soln.sol(tspan)[1], 'r', lw = 2 )
ax[1].grid(True)
ax[1].set_title(r'Phase Trajectory in Plane.', fontsize=16)
ax[1].set_xlabel(r'$y_1$', fontsize=16)
ax[1].set_ylabel(r'$y_2$', fontsize=16)

ax[0].legend()
fig.tight_layout()
plt.savefig('Figure_33')
plt.show()

```

Зупинимось на поясненні цієї програми. Тут важливо відзначити визначення у *Python* функції z стимулювання, яка має вигляд (1.5.6). Функція z є розривною, тобто задається різними формулами (значеннями) на різних відрізках. Якщо визначати у *Python* цю функцію тра-

диційним способом за допомогою функції `def`, то таким способом визначена функція повертає значення тільки на скалярних аргументах, а у програмі у процесі обчислення потрібно значення функції у вузлах сітки (у програмі на множині `tspan`). Звичайно, можна додатково написати цикл, у якому обчислюється функція в окремо взятій точці з множини вузлів і потім цим значенням доповнюємо список значень функції, до тих пір, поки не переберемо у циклі всі точки сітки вузлів. Проте *Python* дає можливість спростити у таких випадках обчислення за допомогою методів, які застосовано у програмі (фрагмент програми визначення функції подразнення):

```
def stmF(x, v, astrt):
    c1=0
    c2=v
    r=np.where(x < astrt, c1, c2)
    return r
```

де v – величина стрибка функції, `astrt` – точка переключення. Так визначена функція повертає значення як на скалярних аргументах, та і на списках.

Цей формат працює тільки у випадку одного стрибка. Розглянемо приклад для складнішого випадку. Наприклад, визначимо у *Python* функцію Courant'a, система яких використовується у методі скінчених елементів (Finite Element Method, FEM):

$$c_a(x) = \begin{cases} 0, & x \leq a - 1, \\ x - a + 1, & a - 1 < x \leq a, \\ a + 1 - x, & a < x \leq a + 1, \\ 0, & x > a + 1. \end{cases} \quad (1.5.7)$$

Тут прийняли що вузли сітки співпадають із цілими числами (для зручності). Визначимо та побудуємо графік функції c_a на проміжку $[0, 5]$ для $a = 2$:

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

def ca(x):
    condition1 = x < 1
    condition2 = np.logical_and(1 <= x, x < 2)
    condition3 = np.logical_and(2 <= x, x < 3)
    condition4 = x >= 3
    r = np.where(condition1, 0.0, 0.0)
    r = np.where(condition2, x-1, r)
    r = np.where(condition3, 3-x, r)
    r = np.where(condition4, 0.0, r)
    return r

tspan=np.arange(0,5,0.01)
vca=ca(tspan)
plt.plot(tspan, vca, 'r', lw=3)
plt.grid(True)
plt.savefig('Figure_34')
plt.show()
```

Програма поверне графік функції Courant'a, зображений на [рис. 1.38](#)

Не складає труднощів тепер перевірити, що функція повертає значення як на скалярному аргументі, так і на списку типу `array` (це важливо!):

```
>>> ca(1.3)
array(0.3)

>>> ca(6.7)
array(0.)

>>> lst=np.array([0.2, 1, 1.5, 2.1, 3, 7.23])

>>> ca(lst)
array([0. , 0. , 0.5, 0.9, 0. , 0. ])
```

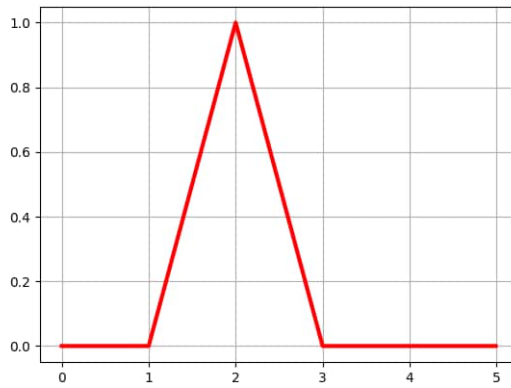


Рис. 1.38: Графіки станів збудження та відновлення у випадку подра-
зника та графік у фазовій площині.

Розділ 2

Математичний апарат ОПТИМАЛЬНОГО КЕРУВАННЯ

Матеріал цієї глави необхідний для розуміння викладених далі положень та фактів. Читачеві, який ознайомлений з теорією диференціальних рівнянь, варіаційного числення та функціонального аналізу можна перейти до наступних розділів.

Основна мета нашого викладу у цьому посібнику – використання середовища *Python* в оптимальну керуванні динамічними процесами які можна описати системами диференціальних рівнянь.

2.1. Оптимізація функціоналів

Розглянемо всеможливі функції аргументу x , визначені на деякому проміжку. Сумою двох функцій f і g називають таку функцію h , задану на тому ж проміжку, значення якої визначається за правилом $h(x) = (f + g)(x) = f(x) + g(x)$ для всіх x із даного проміжка. Для того щоб функцію f помножити на число λ , потрібно всі її значення помножити на це число, тобто $(\lambda f)(x) = \lambda f(x) \quad \forall x$. Множина функцій із операціями додавання і множення на число утворює лінійний простір функцій (функціональний простір).

Нехай маємо неперервні на відрізку $[a, b]$ функції. Сума двох неперервних функцій є неперервною функцією і при множенні неперервної функції на число знову одержимо неперервну функцію. Отже, множина неперервних функцій також утворює лінійний простір, який називається *простором неперервних функцій* і позначається $C[a, b]$.

Аналогічно можна говорити про простір диференційовних функцій, неперервно-диференційовних, нескінченно диференційовних тощо функцій.

На функціональних просторах, як і на множинах, можна задавати відображення, які відіграють дуже важливу роль в функціональному аналізі та оптимізаційних задачах.

Нехай маємо деякий простір \mathcal{R} і в ньому зафіксовано деяку підмножину $X \subset \mathcal{R}$.

• *Означення 2.1.1* Відображення $I : X \rightarrow \mathbb{R}$, яке кожній функції $f \in X$ ставить у відповідність певне число $I[f]$, називається *функціоналом*.

Найпростішим прикладом функціоналу може слугувати визначений інтеграл

$$I[f] = \int_a^b f(x) dx,$$

заданий на просторі неперервних функцій $C[a, b]$.

Складнішим прикладом інтегрального функціоналу є функціонал, визначений співвідношенням

$$I[f] = \int_a^b F(x, f(x)) dx,$$

де F – деяка задана функція двох змінних. Часто зустрічаються інтегральні функціонали, у яких підінтегральна функція залежить не тільки від змінної x і функції f , а й від її похідних f' , тобто $F(x, f(x), f'(x))$.

Іншим прикладом функціоналу є термінальний функціонал

$$I[f] = \Phi(f(x))|_{x=b}$$

де Φ – задана функція однієї змінної.

• *Означення 2.1.2* Функціонал $I : X \rightarrow \mathbb{R}$ називають лінійним функціоналом, якщо для будь-яких функцій $f, g \in X$ і довільних чисел λ_1, λ_2 виконується рівність

$$I[\lambda_1 \cdot f + \lambda_2 \cdot g] = \lambda_1 \cdot I[f] + \lambda_2 \cdot I[g].$$

Оскільки інтеграл від суми функцій дорівнює сумі інтегралів, а сталий множник можна винести за знак інтегралу, то розглянутий вище перший інтегральний функціонал є лінійним.

Формулювання задачі оптимізації (екстремальної задачі) у функціональному просторі \mathcal{R} вимагає наявності двох об'єктів – не пустої (допустимої) множини X , $X \subset \mathcal{R}$ і функціоналу цілі I , який задано на множині X .

Задача оптимізації (мінімізації чи максимізації) полягає у відшуванні такого елемента (функції) $x^* \in X$, для котрого виконується одна із нерівностей:

$$\begin{aligned} I[x^*] &\leq I[x] \quad \text{для всіх } x \in X \quad (\text{задача мінімізації}); \\ I[x^*] &\geq I[x] \quad \text{для всіх } x \in X \quad (\text{задача максимізації}). \end{aligned}$$

Для формулювання задач мінімізації/максимізації використовують відповідні позначення:

$$I[x] \rightarrow \min_{x \in X} \quad \text{або} \quad I[x] \rightarrow \max_{x \in X}.$$

Назагал задача оптимізації заданого функціоналу I на множині X може:

- не мати жодного розв'язку;
- мати єдиний розв'язок;

- мати більше одного розв'язку.

Реалізація наведених трьох можливостей залежить від конкретного виду функціоналу і структури множини, на якій він оптимізується.

★ **Приклад 2.1.1.** Потрібно знайти $x^* \in S = (a, b)$, на якій лінійний функціонал $I[x]$ приймає би якнайменше значення.

Очевидно, що коли $I[x]$ строго зростаюча функція $\forall x \in (a, b)$, то найменше значення функціонал досягає при $x = a$, але точка a – лівий кінець інтервалу (a, b) – не належить йому. Аналогічна ситуація буде, коли потрібно знайти максимум лінійного функціоналу на (a, b) , який досягається у точці $x = b$, яка не належить до (a, b) . Очевидно, що на множині $S = (a, b)$ не існує точки x^* , в якій лінійний функціонал $I[x]$ приймає найменше (найбільше) значення.

★ **Приклад 2.1.2.** Нехай маємо інтегральний функціонал

$$I[x] = \int_a^b \sqrt{1 + (\dot{x}(t))^2} dt,$$

а множина S складається із неперервно-диференційовних функцій, що задовольняють умови $x(a) = A$, $x(b) = B$, $A, B = \text{const}$. Легко зрозуміти, що графіками функцій множини S будуть плоскі криві, які з'єднують точки (a, A) і (b, B) , інтегральний функціонал є довжиною цих кривих. Оскільки, найкоротшою лінією, що з'єднує дві точки є пряма, то мінімум даний функціонал досягає на єдиній функції

$$x(t) = \frac{B - A}{b - a}(t - a) + A,$$

графіком якої є пряма лінія, що з'єднує точки (a, A) і (b, B) . Водночас, задача про максимізацію такого функціоналу на множині S розв'язку не має.

2.2. Абстрактна модель оптимального керування з використанням теорії систем диференціальних рівнянь

При формальному описанні динамічних процесів залежних від часу завжди присутні дві компоненти: вектор стану і вектор керування.

Нехай X – простір стану процесу з елементами $x = (x_1, \dots, x_n)$ (тобто X – n -вимірний евклідовий простір) і нехай U – простір керувань з елементами $u = (u_1, \dots, u_r)$ (тут U – r -вимірний евклідовий простір). Нехай t – час, на відрізку $[0, T]$ ($t \in [0, T]$). Стан процесу та керуючий вектор, зазвичай, залежить від часу, тому відобразимо це через запис $x(t)$, $u(t)$, назвемо $x(t)$ – траєкторією процесу, а $u(t)$ – керуючою функцією.

Тепер будемо розглядати динамічні процеси, котрі описуються системами диференціальних рівнянь

$$\frac{dx}{dt} = f(t, x(t), u(t)), \quad (2.2.1)$$

де права частина $f(t, x, u) \in n$ -вимірною векторною функцією змінних (t, x, u) . Через покомпонентний запис ця система представлена так

$$\frac{dx_i(t)}{dt} = f_i(t, x_1(t), \dots, x_n(t), u_1(t), \dots, u_r(t)) \quad (i = \overline{1, n}). \quad (2.2.1^*)$$

Формальна різниця між станом та керуванням в тому, що саме стан описаний системою диференціальних рівнянь (їхня кількість дорівнює n -числу компонент вектора x), а керування $u(t)$ входить у праву частину системи диференціальних рівнянь, причому похідні від керувань у цьому описі відсутні.

Очевидно, що як математичний об'єкт (2.2.1) або (2.2.1*) має зміст далеко не при всіх функціях $x(t)$, $u(t)$. Тобто потрібно визначити простір траєкторій $x(t)$ і керуючих функцій $u(t)$, а також умови, яким повинні задовольняти праві частини систем (2.2.1) або (2.2.1*).

Практичним вимогам відповідають такі припущення:

- $u(t)$ – кусково-неперервна функція (неперервна всюди на $[0, T]$, за винятком скінченного числа точок, у яких допускаються розриви першого роду);
- $x(t)$ – неперервна та кусково-диференційовна на $[0, T]$;
- $f(t, x, u)$ – кусково-неперервна за t , неперервна і достатньо гладка (має потрібно число похідних) за x і u .

Крім диференціального зв'язку (2.2.1), від динамічного процесу вимагають, зазвичай, ще низку обмежень. Наприклад, початкові умови для диференціальної системи, а також різноманітні обмеження на значення керуючих функцій і траєкторії процесу. Нагадаємо, що

$$(x(t), u(t)) \in V(t), \quad (2.2.2)$$

де $V(t) = X \times U$, тобто є множиною розмірності $(n + r)$ і залежить від часу t .

Множину пар функцій $\{x(t), u(t)\}$ позначимо через D і назвемо *множиною допустимих пар функцій* $\{(x(t), u(t))\}$.

Для завершення формулювання задачі оптимального керування залишилося ввести на множині *цільовий функціонал (критерій якості)*. Найбільш практичною є структура такого функціоналу у вигляді:

$$I[x(t), u(t)] = \int_0^T f_0(t, x(t), u(t)) dt + F(x(T)) \quad (2.2.3)$$

де $f_0(t, x, u)$, $F(x)$ – задані скалярні достатньо гладкі функції своїх аргументів.

Отже, перед нами стоїть задача про мінімізацію функціоналу (2.2.3) на множині D , яка складається із пар функцій $x(t)$, $u(t)$, при виконанні $\forall t \in [0, T]$ умов:

- 1) $u(t)$ – кусково-неперервна;
- 2) $x(t)$ – неперервна і кусково-диференційовна;

- 3) виконуються обмеження $(x(t), u(t)) \in V(t)$;
 4) виконується диференціальний зв'язок

$$\frac{dx(t)}{dt} = f(t, x(t), u(t)).$$

Надалі вважатимемо, що умови **умови 1)-2)** завжди виконуються і не будемо без потреби їх виписувати.

2.3. Рівняння Euler'а

Розглянемо деякий частинний варіант задачі оптимального керування, сформульованого у попередньому параграфі.

Потрібно мінімізувати функціонал

$$I[x(t), u(t)] = \int_0^T f_0(t, x(t), u(t)) dt$$

на множині D , яка складається із функцій $\{x(t), u(t)\}$, що задовольняють умови:

$$\frac{dx(t)}{dt} = u(t), \quad x(0) = x^0, \quad x(T) = x^T,$$

причому розмірності векторів x і u дорівнюють одиниці; x^0 і x^T – задані сталі. Наведену тут задачу називають *найпростішою задачею варіаційного числення*, математичної дисципліни, яка стала основою *теорії оптимального керування*.

Зазвичай, *найпростішу задачу варіаційного числення* записують у вигляді:

$$I[x(t)] = \int_0^T f_0(t, x(t), \dot{x}(t)) dt \rightarrow \min, \quad (2.3.1)$$

$$x(0) = x^0, \quad x(T) = x^T. \quad (2.3.2)$$

Тут в f_0 замість $u(t)$ записано $\dot{x}(t) = \frac{dx(t)}{dt}$.

У задачі (2.3.1), (2.3.2) крива $x(t)$ – допустима, якщо вона є неперервною і кусково-диференційовною та задовольняє (2.3.2).

• **О з н а ч е н н я** 2.3.1 Функціонал $I[x(t)]$ на кривій $x^*(t)$ досягає сильного мінімуму, якщо існує число $\varepsilon > 0$ таке, що для всіх допустимих функцій $x(t)$, що задовольняють умову

$$|x^*(t) - x(t)| \leq \varepsilon \quad \forall t \in [0, T] \quad (2.3.3)$$

і виконується нерівність

$$I[x^*(t)] \leq I[x(t)]. \quad (2.3.4)$$

Якщо ж нерівність (2.3.4) виконується для всіх допустимих функцій $x(t)$ з умовою (2.3.3) та умовою

$$\left| \frac{dx^*(t)}{dt} - \frac{dx(t)}{dt} \right| \leq \varepsilon, \quad \forall t \in [0, T],$$

то на кривій $x^*(t)$ функціонал I має слабкий мінімум.

У першому випадку криву $x^*(t)$ деколи називають *сильною мінімаллю*, а у другому – *слабкою*, а у більшості випадків просто *мінімаллю*.

Далі розглянемо *необхідні умови слабого мінімуму* інтегрального функціоналу, використовуючи метод варіації.

Нехай у задачі (2.3.1), (2.3.2) маємо деяку *допустиму траєкторію* $x(t)$. Нам потрібно знайти умови, яким би задовольняла ця крива, якщо б вона була *слабкою мінімаллю*. Будь-яку іншу допустиму криву можна записати у вигляді $\tilde{x}(t) = x(t) + \delta x(t)$, де $\delta x(t)$ – приріст або, за термінологією варіаційного числення, *варіація кривої* $x(t)$. Варіацію зручно представляти у вигляді $\delta x(t) = \varepsilon \xi(t)$, де $\xi(t)$ – неперервна і кусково-гладка крива, що задовольняє умови $\xi(0) = \xi(T) = 0$, а $\varepsilon > 0$ достатньо мале число. Тоді $\tilde{x}(t) = x(t) + \varepsilon \xi(t)$ – *допустима крива*.

Тепер розглянемо функціонал I на кривих $\tilde{x}(t)$, які близькі до $x(t)$, тобто

$$\Delta I[x(t)] = I[x(t) + \varepsilon \xi(t)] - I[x(t)].$$

Враховуючи інтегральний вигляд функціоналу I , можемо записати

$$\Delta I[x(t)] = \int_0^T \left[f_0(t, x(t) + \varepsilon \xi(t), \dot{x}(t) + \varepsilon \dot{\xi}(t)) - f_0(t, x(t), \dot{x}(t)) \right] dt. \quad (2.3.5)$$

Якщо функції $x(t)$, $\xi(t)$ вважати заданими, то *приріст функціоналу* ΔI є функцією числового параметру ε . Тоді підінтегральний вираз у (2.3.5) можна $\forall t \in [0, T]$ представити, використовуючи формулу Тейлора, у вигляді:

$$f_0(t, x + \varepsilon \xi, \dot{x} + \varepsilon \dot{\xi}) - f_0(t, x, \dot{x}) = \frac{\partial f_0(t, x, \dot{x})}{\partial x} \varepsilon \xi(t) + \frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \varepsilon \dot{\xi}(t) + o(\varepsilon),$$

де $o(\varepsilon)$ – члени більшого порядку малості, ніж ε $\left(\lim_{\varepsilon \rightarrow 0} \frac{o(\varepsilon)}{\varepsilon} = 0 \right)$. Після цього приріст (2.3.5) можна записати так:

$$\Delta I[x(t)] = \varepsilon \delta I[x(t)] + o(\varepsilon), \quad (2.3.6)$$

де

$$\delta I[x(t)] = \int_0^T \left[\frac{\partial f_0(t, x, \dot{x})}{\partial x} \varepsilon \xi(t) + \frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \varepsilon \dot{\xi}(t) \right] dt.$$

Вираз (2.3.6) називають *першою варіацією функціоналу*.

Теорема 2.3.1 (необхідна умова слабкого мінімуму [10]). *Нехай $x^*(t)$ слабка мінімаль функціоналу I . Тоді для будь-яких $\xi(t)$ таких що $\xi(0) = \xi(T) = 0$ виконується умова*

$$\delta I[x^*(t)] = 0 \quad (2.3.7)$$

Доведення. нехай $\delta I[x^*(t)] = \alpha \neq 0$. Тоді із (2.3.6) одержимо $\Delta I[x^*(t)] = \varepsilon \left(\alpha + \frac{o(\varepsilon)}{\varepsilon} \right)$ для будь-якого достатньо малого ε . Виберемо ε так, щоб $\varepsilon\alpha < 0$. Оскільки $\frac{o(\varepsilon)}{\varepsilon} \rightarrow 0$, то знайдеться $\exists \varepsilon_0 > 0$ таке, що при $|\varepsilon| < \varepsilon_0$ виконується нерівність $\Delta I[x^*(t)] < 0$. В той же час маємо

$$\begin{aligned} |\tilde{x}(t) - x^*(t)| &= |\varepsilon| \cdot |\xi(t)|, \\ |\dot{\tilde{x}}(t) - \dot{x}^*(t)| &= |\varepsilon| \cdot |\dot{\xi}(t)|, \quad \forall t \in [0, T]. \end{aligned}$$

Отже ми знаходимося в умовах означення слабкого мінімуму функціоналу I , тобто нерівність $\Delta I[x^*(t)] < 0$ є неможливою. Значить, величина $\delta I[x^*(t)]$ не може бути відмінною від нуля

□

Насправді умова слабкого мінімуму (2.3.7) є неконструктивною. Тому зробимо деякі перетворення, проінтегрувавши частинами другий доданок в (2.3.6), врахувавши, що $\xi(0) = \xi(T) = 0$. Тоді

$$\int_0^T \frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \dot{\xi}(t) dt = - \int_0^T \frac{d}{dt} \left(\frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \right) \xi(t) dt$$

і вираз (2.3.6) для *варіації функціоналу* прийме вигляд

$$\delta I[x(t)] = \int_0^T \left[\frac{\partial f_0(t, x, \dot{x})}{\partial x} - \frac{d}{dt} \left(\frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \right) \right] \xi(t) dt.$$

Тепер вираз (2.3.6) дещо спростився, але він може набути простішого вигляду, якщо скористатися *основною лемою варіаційного числення*.

Лема 2.3.1 ([10, 9]). *Нехай рівність*

$$\int_0^T A(t) \xi(t) dt = 0$$

виконується для всіх неперервних $\xi(t)$, з умовами $\xi(0) = \xi(T) = 0$, і деякою неперервною функцією $A(t)$. Тоді $\forall t \in [0, T]$ $A(t) = 0$.

Доведення. Припустимо, що $A(\tau) \neq 0$ при $\tau \in (0, T)$. Для визначеності нехай $A(\tau) > 0$. Оскільки $A(t)$ – неперервна, то існує $\varepsilon > 0$ таке, що $A(t) > 0 \quad \forall t \in [\tau - \varepsilon, \tau + \varepsilon]$. Візьмемо функцію $\xi(t)$ додатною на $(\tau - \varepsilon, \tau + \varepsilon)$ і рівною нулеві поза цим інтервалом. Тоді

$$\int_0^T A(t)\xi(t)dt = \int_{\tau-\varepsilon}^{\tau+\varepsilon} A(t)\xi(t)dt > 0,$$

що суперечить умові леми. Отже, $A(t) = 0$ для всіх $t \in (0, T)$, зокрема, рівність нулеві на кінцях інтервалу $(0, T)$ впливає із неперервності $A(t)$.

□

Використовуючи [лема 2.3.1](#), а також останній вираз для $\delta I[x(t)]$, одержимо

$$\left[\frac{\partial f_0(t, x, \dot{x})}{\partial x} - \frac{d}{dt} \left(\frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \right) \right] \Big|_{x=x^*} = 0. \quad (2.3.8)$$

Співвідношення [\(2.3.8\)](#) називається *рівнянням Euler'a*, яке є диференціальним рівнянням другого порядку стосовно $x^*(t)$, оскільки

$$\begin{aligned} \frac{d}{dt} \left(\frac{\partial f_0(t, x, \dot{x})}{\partial \dot{x}} \right) &= \frac{\partial^2 f_0(t, x, \dot{x})}{\partial t \partial \dot{x}} + \\ &\quad \frac{\partial^2 f_0(t, x, \dot{x})}{\partial x \partial \dot{x}} \dot{x} + \frac{\partial^2 f_0(t, x, \dot{x})}{\partial \dot{x}^2} \ddot{x}. \end{aligned}$$

★ *Приклад 2.3.1.* ([10]) Мінімізувати функціонал

$$I[x(t)] = \int_0^1 [x(t) - (\dot{x}(t))^2] dt,$$

за умов $x(0) = 1$, $x(1) = 0$.

Розв'язання. ∇ Тут $f_0(t, x, \dot{x}) = (\dot{x}(t))^2 - x(t)$, звідки

$$\frac{\partial f_0}{\partial x} = -1, \quad \frac{\partial f_0}{\partial \dot{x}} = 2\dot{x}.$$

Рівняння Euler'а має вигляд

$$-1 - 2\ddot{x}^*(t) = 0, \quad \text{або} \quad \ddot{x}^*(t) = -\frac{1}{2}.$$

Нехай $\dot{x}^*(0) = x_0^1$, тоді $\dot{x}^*(t) = x_0^1 - \frac{t}{2}$. Враховуючи $x^*(0) = 1$, одержимо $x^*(t) = 1 + x_0^1 t - \frac{t^2}{4}$. Стали x_0^1 знайдемо із умови $x^*(1) = 0$, тоді $x_0^1 = -\frac{3}{4}$. У підсумку

$$x^*(t) = 1 - \frac{3t}{4} - \frac{t^2}{4},$$

рівняння для слабкої мінімалі, якщо вона існує.

\triangle

Питання існування є непростим. Наприклад, якщо у функціонала **прикладу 2.3.1** змінити знак перед інтегралом, рівняння Euler'а не зміниться, але мінімаль відсутня. Для одержання підтвердженої відповіді потрібно використовувати сильніші необхідні умови або достатні умови локального мінімуму.

2.4. Оптимальне керування гіперболічними системами рівнянь першого порядку з двома незалежними змінними

Розглянемо задачу оптимізації для системи напівлінійних гіперболічних рівнянь першого порядку з двома незалежними змінними

$$\frac{\partial x(s, t)}{\partial t} + A(s, t) \frac{\partial x(s, t)}{\partial s} = f(x, s, t) \quad (2.4.1)$$

де $(s, t) \in \Pi = (s_0, s_1) \times (t_0, t_1)$.

Припустимо, що $x(s, t)$ – n -вимірний вектор-функція. Вважаємо також, що система (2.4.1) записана у інваріантному вигляді, тобто матриця $A(s, t)$ представлена у діагональному вигляді. Відзначимо, що за допомогою невиродженого лінійного перетворення змінних матрицю $A(s, t)$ завжди можна привести до діагонального вигляду [1, 3].

Додатково припускаємо, що діагональні елементи $a_i(s, t)$ ($i = \overline{1, n}$) матриці $A(s, t)$ є знакосталі в області Π :

$$\begin{aligned} a_i(s, t) &> 0, & i = \overline{1, m_1}; \\ a_i(s, t) &= 0, & i = \overline{m_1 + 1, m_2}; \\ a_i(s, t) &< 0, & i = \overline{m_2 + 1, n}. \end{aligned}$$

Складемо дві діагональні підматриці: $A^+(s, t)$ розміру $m_1 \times m_1$ і $A^-(s, t)$ розміру $(n - m_2) \times (n - m_2)$ із додатних та від'ємних елементів A , відповідно. Із вектора стану $x = x(s, t)$ також виділимо два підвектори: $x^+ = (x_1, x_2, \dots, x_{m_1})$ і $x^- = (x_{m_2+1}, x_{m_2+2}, \dots, x_n)$, що відповідають додатним та від'ємним діагональним елементам матриці A .

Задачу мінімізації будемо розглядати, наприклад, для крайових і стартових керувань, що входять у початково-крайові умови системи (2.4.1) при різних зв'язках між компонентами вектора стану і керування на межі області Π .

Для простоти викладу розглянемо для системи (2.4.1) початково-крайові умови:

$$x(s, t_0) = x^0(s), \quad s \in S; \quad (2.4.2)$$

$$x^-(s_1, t) = q(t), \quad t \in T; \quad (2.4.3)$$

$$\begin{cases} \frac{\partial x^+(s_0, t)}{\partial t} = g(x^+(s_0, t), u(t), t), & t \in T; \\ x^+(s_0, t_0) = (x^0(s_0))^+. \end{cases} \quad (2.4.4)$$

Тут $S = [s_0, s_1]$, $T = [t_0, t_1]$.

За множину допустимих керувань виберемо сукупність обмежених

і вимірних на відрізку T r -вимірних вектор-функцій $u = u(t)$, що майже всюди задовольняють на цьому відрізку обмеженням

$$u(t) \in U, \quad t \in T, \quad (2.4.5)$$

де U – компакт із r -вимірною евклідовою простору.

Метою задачі оптимального керування є мінімізація функціоналу

$$I[u(t)] = \int_S \varphi(x(s, t), s) ds + \iint_{\Pi} F(x, s, t) ds dt, \quad (2.4.6)$$

визначеного на розв'язках задачі (2.4.1)-(2.4.4) при допустимих керуваннях, що задовольняють умову (2.4.5).

Повний аналіз задачі оптимального керування для гіперболічної системи рівнянь буде проведено у наступних главах даного посібника для задачі біопопуляцій, тому обмежимося тут наведенням схеми дослідження таких задач [1].

Отже для знаходження вектор-функції стану, що відповідає деякому фіксованому керуванню $u = u(t)$, потрібно:

- а) підставити керування u в праву частину системи диференціальних рівнянь (2.4.4) та розв'язати відповідну задачу Cauchy для визначення $x^+(s_0, t)$;
- б) розв'язати першу крайову задачу для системи (2.4.1) з відомими значеннями $x^+(s_0, t)$, $t \in T$, $x^-(s_1, t)$, $t \in T$, $s \in S$ методом характеристик [3].

Задачу оптимального керування (2.4.1)-(2.4.6) розглянемо при таких припущеннях:

- 1) діагональні елементи матриці A є неперервно диференційовні в Π ;
- 2) вектор-функції $x^0(s)$ і $q(t)$ неперервні відповідно на S і T з умовами погодження $q(t_0) = (x^0(s_1))^-$;

- 3) вектор-функції $f(x, s, t)$, $g(x^+, u, t)$ і скалярні функції $\varphi(x, s)$, $F(x, s, t)$ неперервні за всіма своїми аргументами і мають неперервні і обмежені частинні похідні за x , відповідно, на $\mathbb{R}^n \times S \times T$, $\mathbb{R}^{m_1} \times U \times T$, $\mathbb{R}^n \times S$, $\mathbb{R}^n \times S \times T$.

При наведених умовах для будь-якого допустимого керування існує єдиний узагальнений розв'язок початково-крайової задачі (2.4.1)-(2.4.4) із класу неперервних в Π функцій, які також мають неперервні похідні вздовж відповідного i -го сімейства характеристик [3].

• *Означення 2.4.1* Пару функцій $\{u, x\}$ називають допустимим процесом, якщо керування $u = u(t)$ допустиме, а $x = x(t)$ – відповідний даному керуванню узагальнений розв'язок задачі (2.4.1)-(2.4.4).

Зазначимо, що під узагальненим розв'язком задачі (2.4.1)-(2.4.4) при заданому керуванні розуміємо неперервний розв'язок відповідної системи інтегральних рівнянь Volterra другого роду, одержаної інтегруванням системи (2.4.1) вздовж характеристик $\xi = s^i(\tau; s, t)$ на відрізках $[\tau^i, t]$, тобто неперервні розв'язки системи інтегральних рівнянь [3].

$$x_i(s, t) = x_i(\xi^i(s, t), \tau^i(s, t)) + \int_{\tau^i(s, t)}^t f_i(x(\xi, \tau), \xi, \tau) \Big|_{\xi=s^i(\tau; s, t)} d\tau, \quad (s, t) \in \Pi, \quad i = \overline{1, n}.$$

Можливі інші формулювання задач оптимального керування для системи (2.4.1), зокрема, керування може входити в праві частини системи (2.4.1), у крайові чи початкові умови для таких систем, можуть бути складнішими, наприклад, інтегральними [1, 20].

2.5. Модель Solow як приклад математичного моделювання економічного процесу диференціальними рівняннями

Розглянемо *модель економічного зростання* запропоновану лавреатом Нобелівської премії з економіки в 1987 році Robert'ом Solow. Ця модель, основу якої складає диференціальне рівняння першого порядку, відіграє важливу роль у неокласичній теорії економічного росту [22].

Модель Solow (в деяких літературних джерелах її ще називають *моделью Solow-Swan'a*) є *однофакторною моделлю економічного росту*, в якій економічна система виробляє один продукт. Модель достатньо адекватно відображає важливі макроекономічні елементи процесу виробництва.

Стан економіки в моделі Solow задається п'ятьма *ендогенними змінними* (викликані причинами внутрішнього походження):

1. X – валовий внутрішній продукт (ВВП);
2. C – фонд не виробничого споживання;
3. I – інвестиції;
4. L – праця (число працюючих);
5. K – капітал (основні виробничі фонди).

У моделі також використано *екзогенні показники* (задані поза системою): ν – річний темп приросту числа працюючих; μ – доля амортизації за рік основних виробничих фондів; ρ – норма накопичення (доля валових інвестицій у ВВП). Екзогенні параметри підпорядковані певним обмеженням: $-1 < \nu < 1$, $0 < \mu < 1$, $0 < \rho < 1$.

Вважаємо, що ендогенні змінні залежать від часу а екзогенні змінні є сталими, зокрема, норма накопичення є керуючим параметром.

Час t змінюється неперервно від початкового моменту $t_0 = 0$ впродовж одного року.

Припустимо, що річний випуск в кожний момент часу визначено *лінійно-однорідною неокласичною виробничою функцією* (для зручності аргумент t будемо опускати, пам'ятаючи про його присутність за замовчуванням)

$$X = F(K, L), \quad (2.5.1)$$

яка є невід'ємною при невід'ємних K , L і $F(0, 0) = 0$. Вважаємо, що функція $F(K, L)$ має додатні частинні похідні першого порядку, а $F''_{KK} < 0$. Крім того, ця *функція є однорідною*, тобто, $F(\lambda K, \lambda L) = \lambda F(K, L)$, λ – додатний параметр.

Розглянемо приріст основних ресурсів за малий проміжок часу Δt . Кількість трудового ресурсу L може змінюватися з часом, а темп його зміни $\frac{dL}{dt}$ буде залежати від кількості L – працюючих, тобто

$$\begin{cases} \frac{dL}{dt} = \nu L, \\ L(0) = L_0. \end{cases}$$

Звідки $L(t) = L_0 e^{\nu t}$.

Амортизація основних виробничих фондів та інвестиції підпорядковані диференціальному рівнянню першого порядку

$$\begin{cases} \frac{dK}{dt} = -\mu K + I, \\ K(0) = K_0, \end{cases}$$

причому $I = \rho X$, а фонд споживання $C = (1 - \rho)X$.

У підсумку одержимо *модель Solow* в абсолютних показниках

$$\begin{aligned} L &= L_0 e^{\nu t}, & \frac{dK}{dt} &= -\mu K + \rho X, & K(0) &= K_0, \\ X &= F(K, L), & C &= (1 - \rho)X, & I &= \rho X. \end{aligned} \quad (2.5.2)$$

Для зручності введемо відносні показники:

$$k = \frac{K}{L} \text{ – фондозабезпеченість праці;}$$

$x = \frac{X}{L}$ – продуктивність праці;

$i = \frac{I}{L}$ – питома інвестиція на одного працюючого;

$c = \frac{C}{L}$ – споживання на одного працюючого.

Оскільки

$$x = \frac{F(K, L)}{L} = F\left(\frac{K}{L}, 1\right) = f(k), \quad i = \rho x, \quad c = (1 - \rho)x,$$

$$\frac{dK}{dt} = \frac{d}{dt}(kL) = \nu kL + L \frac{dk}{dt},$$

то кінцева модель Solow у відносних показниках є такою:

$$\begin{aligned} \frac{dk}{dt} &= -\lambda k + \rho f(k), & \lambda &= \mu + \nu, & k(0) &= k_0 = \frac{K_0}{L_0}, \\ x &= f(k), & i &= \rho f(k), & c &= (1 - \rho)f(k). \end{aligned} \quad (2.5.3)$$

Враховуючи, що кожний абсолютний чи відносний показник змінюється в часі, то також можна говорити про траєкторії системи (2.5.3) в абсолютних чи відносних показниках.

• *О з н а ч е н н я 2.5.1* Траєкторію називають рівноважною (стаціонарною, стійкою), коли основні показники не змінюються з часом, тобто

$$\begin{aligned} k(t) &= k^* = \text{const}, & x(t) &= x^* = \text{const}, \\ i(t) &= i^* = \text{const}, & c(t) &= c^* = \text{const}. \end{aligned}$$

Із (2.5.3) зразу одержуємо, що стаціонарність фондозабезпечення k тягне за собою стаціонарність всіх інших показників. Тому розгляд стаціонарності системи (2.5.3) можна обмежити лише одним показником k .

Для стаціонарної траєкторії $k(t) = k^*$ повинна виконуватися рівність $\frac{dk^*}{dt} = 0 \quad \forall t$, тому із першого рівняння (2.5.3) одержимо

$$-\lambda k^* + \rho f(k^*) = 0, \quad \text{або} \quad \rho f(k^*) = \lambda k^*.$$

Отже, якщо стаціонарна траєкторія $k(t) = k^*$ існує, то для неї виконується останнє рівняння, тобто питання існування такої траєкторії еквівалентне існуванню додатної сталої k^* , яка є коренем рівняння

$$\rho f(k) = \lambda k. \quad (2.5.4)$$

Повернемося до виробничої функції $F(K, L)$ та побудованої на її основі функції f . З неокласичності виробничої функції випливає $f(0) = 0$, $f' > 0$, $f'' < 0$, а сама функція f є зростаючою і строго вгнутою на проміжку $[0, +\infty)$. Зокрема, якщо вимагати, щоб

$$0 < \frac{\lambda}{\rho} < f'(0), \quad (2.5.5)$$

то рівняння (2.5.4) має єдиний додатний розв'язок $k = k^*$ (див. рис. 2.1). Одержаний результат сформулюємо у вигляді теореми.

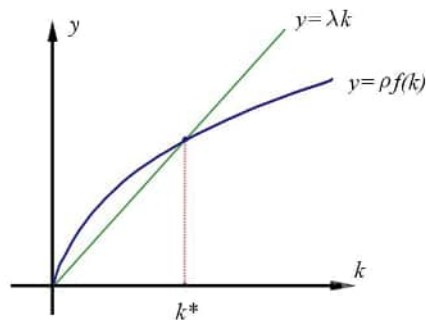


Рис. 2.1: Існування розв'язку рівняння (2.5.4)

Теорема 2.5.1 (існування рівноваги [22, 11]). При виконанні умови (2.5.5) існує єдиний додатний корінь рівняння (2.5.4) (стаціонарна точка існує і єдина).

2.6. Завдання для самостійного опрацювання

- ◆ *Завдання 2.6.1.* Знайти розв'язок задачі Cauchy

$$\begin{cases} \frac{dx}{dt} = 2te^{-t}, \\ x(0) = 1. \end{cases}$$

- ◆ *Завдання 2.6.2.* Знайти загальний розв'язок рівняння

$$\frac{dx}{dt} = \frac{1}{\sqrt{1-t}}.$$

- ◆ *Завдання 2.6.3.* Знайти розв'язок задачі Cauchy для системи диференціальних рівнянь

$$\begin{cases} \frac{dx}{dt} = 1 - \frac{1}{y}, \\ \frac{dy}{dt} = \frac{1}{x-t}, \\ x(0) = -1 \\ y(0) = 1. \end{cases}$$

- ◆ *Завдання 2.6.4.* Знайти загальний розв'язок системи диференціальних рівнянь

$$\begin{cases} \frac{dx}{dt} = y + 2e^t, \\ \frac{dy}{dt} = x + t^2. \end{cases}$$

- ◆ *Завдання 2.6.5.* Знайти екстремум функції

$$f(x_1, x_2) = x_1^2 - x_2^2, \quad \text{в області } D = \{(x_1, x_2) \in \mathbb{R}^2 : x_1^2 + x_2^2 \leq 1\}.$$

◆ *Завдання 2.6.6.* Дослідити екстремум функції

$$f(x_1, x_2) = (x - 1)^2 - 2x_2^2, \quad (x_1, x_2) \in \mathbb{R}^2.$$

◆ *Завдання 2.6.7.* Показати, що функція $f(x_1, x_2) = (1 + e^{x_2}) \cos x_1 - x_2 e^{x_2}$ має нескінченну множину максимумів і жодного мінімуму.

◆ *Завдання 2.6.8.* Знайти віддаль між кривими

1. $f_1(x) = xe^{-x}, \quad f_2(x) = 0, \quad x \in [0, 2];$
2. $f_1(x) = x, \quad f_2(x) = x^2, \quad x \in [0, 1].$

◆ *Завдання 2.6.9.* Показати, що функціонал $L[y(x)] = y(x_0) -$ лінійний.

◆ *Завдання 2.6.10.* Знайти варіацію функціоналу

$$I[y] = \int_{-1}^1 (y'e^y + xy^2) dx.$$

◆ *Завдання 2.6.11.* Визначити допустимі криві, на яких функціонал

$$I[y(x)] = \int_1^2 (y' - 2xy) dx, \quad \text{за умов } y(1) = 0, y(2) = -1,$$

досягає екстремуму.

◆ *Завдання 2.6.12.* Знайти екстремалі функціоналу

$$I[y(x)] = \int_0^{2\pi} (y'^2 - y^2) dx, \quad y(0) = 1, y(2\pi) = 1.$$

◆ *Завдання 2.6.13.* У прямокутнику $S \times T = \{(s, t) : 0 \leq s \leq s_1, 0 \leq t \leq t_1\}$ керуючий процес описується початково-крайовою задачею

$$\begin{cases} \frac{\partial x_1}{\partial t} + \lambda \frac{\partial x_1}{\partial s} = \alpha(s)(x_1 - x_2), \\ \frac{\partial x_2}{\partial t} - \lambda \frac{\partial x_2}{\partial s} = \beta(s)(x_1 - x_2), \end{cases}$$

$$x_1(s, 0) = u(s) + q(s), \quad x_2(s, 0) = u(s) - q(s), \quad s \in S,$$

$$|u(s)| \leq 1, \quad s \in S.$$

Мінімізувати функціонал

$$I[u] = \int_S (x_1(s, t_1) + x_2(s, t_1) - \eta(s))^2 ds. \quad (2.6.1)$$

Функції α , β , q , η – задані, $u = u(t)$ – керування.

◆ *Завдання 2.6.14.* Мінімізувати функціонал (2.6.1) із завдання 2.6.13, якщо

$$\begin{cases} \frac{\partial x_1}{\partial t} + \frac{\partial x_1}{\partial s} = \alpha(s, t)x_1 + \beta(s, t)x_2, \\ \frac{\partial x_2}{\partial t} + \frac{\partial x_2}{\partial s} = \mu(s, t)x_1 + \gamma(s, t)x_2, \end{cases}$$

$$x_1(0, t) = 1 - v(t), \quad x_2(0, t) = v(t), \quad t \in T,$$

$$x_1(s, 0) = x_1^0(s), \quad x_2(s, 0) = x_2^0(s), \quad s \in S,$$

$$v(t) \in [0, 1], \quad t \in T.$$

◆ *Завдання 2.6.15.* Продемонструвати модель Solow для виробничої функції Cobb-Douglas'a

$$F(K, L) = AK^\alpha L^{1-\alpha}, \quad A = \text{const}, \quad \alpha \in (0, 1).$$

◆ *Завдання 2.6.16.* Нехай виробнича функція $F(K, L) = 5K^{\frac{1}{3}}l^{\frac{2}{3}}e^{0.03t}$. Норма амортизації становить 0.08. Чисельність працюючих зростає на 2% щорічно, а норма накопичення рівна 25%. Який рівноважний рівень фондозабезпечення за сталих ефективності, інвестицій і споживання?

◆ *Завдання 2.6.17.* Нехай $k_1(t)$, $k_2(t)$ – розв’язки початкових задач

$$\begin{cases} \frac{dk_i}{dt} = sf(k_i) - (\delta + n)k_i, \\ k_i(0) = k_i^0, \end{cases} \quad i = 1, 2,$$

де s , δ – const, $n = n(t)$ – задана функція. Довести, що при умові $k_1^0 < k_2^0$ справедливо $k_1(t) < k_2(t) \quad \forall t > 0$.

◆ *Завдання 2.6.18.* Довести наступне: якщо $k_i(t)$ ($i = 1, 2$) є розв’язками початкових задач відповідно

$$\begin{cases} \frac{dk}{dt} = sf(k) - (\delta + n_i(t))k, \\ k(0) = k_0, \end{cases}$$

то при $n_1(t) \leq n_2(t)$, $\forall t \geq 0$ справедливо, $k_1(t) \geq k_2(t)$.

◆ *Завдання 2.6.19.* Нехай виробнича функція Cobb-Douglas має вигляд $K^\alpha(t)L^{1-\alpha}(t)$, $\alpha \in (0, 1)$, то має місце

Твердження 2.6.1 *Якщо $k(t)$ є розв’язком задачі (модифікована модель Solow-Swan’a)*

$$\begin{cases} \frac{dk}{dt} = sf(k) - (\delta + n(t))k, \\ k(0) = k_0, \end{cases}$$

то $k(t)$ має вигляд

$$k(t) = \frac{1}{e^{\delta t} L(t)} \left[k_0^{1-\alpha} + (1-\alpha)s \int_0^t \left(e^{\delta \sigma} L(\sigma) \right)^{1-\alpha} d\sigma \right]^{\frac{1}{1-\alpha}}.$$

◆ *Завдання 2.6.20.* Показати, що рівняння, яке описує динаміку зміни робочої сили за виробничою функцією Cobb-Douglas'a

$$\frac{dL}{dt} = \alpha L(t) - bL^{2-\beta}(t), \quad 0 < b < \alpha, \quad 0 \leq \beta < 1,$$

має розв'язок

$$L(t) = \left(1 - \frac{b}{a}\right)^{-\frac{1}{1-\beta}} e^{at} \left(1 - \frac{b}{b-a} e^{(1-\beta)at}\right)^{-\frac{1}{1-\beta}}.$$

Розділ 3

Формулювання задачі ОПТИМАЛЬНОГО керування

3.1. Загальні поняття про керовані системи

Поняття системи (стан об'єкта) та керування є найважливішими в теорії оптимального керування. Говорячи про стан системи, розуміємо, що мова йтиме про динамічну систему, тобто систему, яка розвивається, еволюціонує з часом і в кожен момент часу може перебувати в одному із можливих станів. Саме зміна стану системи з часом характеризує розвиток або функціонування даної системи.

Векторний простір \mathbb{R}^n , котрому належать всеможливі стани системи, називають *простором стану* або *фазовим простором*. Оскільки система розвивається з часом, то параметри системи є функціями часу, тобто $x(t) = (x_1(t), \dots, x_n(t))$. При зміні часу від значення $t = t_0$ до деякого скінченного значення $t = T$ точка $x(t)$ у фазовому просторі рухається вздовж відповідної кривої, яку називають *траєкторією системи*.

Серед існуючих систем є такі, на котрі можна впливати певним чином, залежно від сформульованої мети. Керування є та дія, що може змінювати існуючий стан системи, впливаючи на її розвиток позитивно

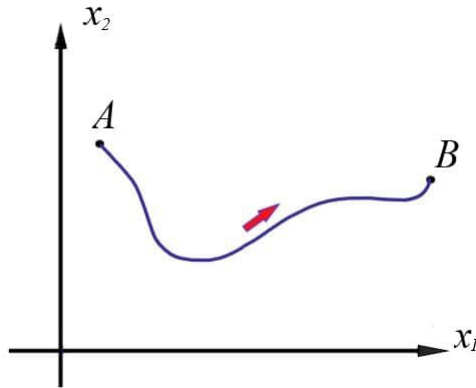


Рис. 3.1: Траєкторія системи

переводить систему із початкового стану $x^{(0)} \in \mathbb{R}^n$ у кінцевий стан $x(T) \in \mathbb{R}^n$ за фазовою траєкторією $x = x(t)$, $t_0 \leq t \leq T$.

3.2. Задача керування із закріпленими кінцями

Сформулюємо допоміжні задачі, які на відміну від задач оптимального керування назвемо просто задачами керування.

Насамперед розглянемо задачу *керування із фіксованою дією* керування, яка полягає у заданні у фазовому просторі двох точок (*стани системи*) $A, B \in \mathbb{R}^n$ і серед всіх допустимих керувань, які переводять систему з початкового стану $x(t_0) = A$, у стан $x(T) = B$ за наперед заданий час $t = T$ (див. [рис. 3.1](#)).

Задача *керування із нефіксованою дією* керування також передбачає задання початкового A і кінцевого B станів і полягає у пошуку серед всіх допустимих такого керування, котре систему із початкового стану переводить у кінцевий, але кінцевий момент часу T наперед не заданий і його теж потрібно знайти.

Відмінність сформульованих задач лише в тому, що у другій із них

час переведення системи із початкового стану у кінцевий невідомий, тобто може виявитися як завгодно довгим.

Для таких задач важливу роль відіграє питання існування допустимого керування, яке переводить дану систему із заданого початкового стану у заданий кінцевий стан. Це питання щодо *керуваності системи*. Якщо вказане керування не існує (*система некерована*), то обидві сформульовані задачі розв'язку не мають, а отже процес розв'язності цих задач не має сенсу.

Назагал, питання керуваності є доволі складним. Однак, якщо обмежитися деякими спрощеними випадками систем, то можна одержати вичерпні результати.

Прикладом спрощеного випадку можуть слугувати лінійні системи керування виду

$$\frac{dx}{dt} = Kx + Lu,$$

де K і L – числові матриці відповідних розмірів. Таку систему називають *цілком керованою*, якщо для довільної пари точок $A, B \in \mathbb{R}^n$ існує допустиме керування, котре переводить за деякий час дану систему із одного стану в інший.

Зазвичай, говорячи про керовану систему диференціальних рівнянь (3.1.1)-(3.1.2), вважають, що допустиме керування, яке переводить її із початкового стану у кінцевий, існує.

3.3. Задача оптимального керування

Для більшості реальних прикладних задач керування існує, зазвичай, набір керуючих функцій $u = u(t) = (u_1(t), \dots, u_r(t))$, які дозволяють розв'язати задачу керування із фіксованим чи нефіксованим часом керування. Тобто є можливість вибору із всіх розв'язків задачі керування такого набору керуючих функцій, які у певному сенсі були б найкращими (*оптимальними*).

Щоб сформулювати задачу оптимального керування необхідно знайти умову, яка б дозволила вибирати найвигідніший розв'язок даної зада-

чі. Для цієї мети використовують *критерій оптимальності* (цільовий функціонал)

$$I[u] = \int_{t_0}^T f_0(t, x_1(t), \dots, x_n(t), u_1(t), \dots, u_r(t)) dt. \quad (3.3.1)$$

Тут $f_0(t, x_1, \dots, x_n, u_1, \dots, u_r)$ – задана (фіксована) функція $n + r + 1$ змінних і володіє властивостями для забезпечення існування інтегралу. Ознакою більшої чи меншої вигоди вибору керування u буде слугувати значення інтегрального функціоналу (3.3.1), тобто число $I[u]$. Для визначеності завжди можна вважати, що чим більше значення критерію оптимальності $I[u]$, тим більш вигідним є керування u , тобто те, яке надає найбільше можливе значення інтегральному функціоналу (3.3.1).

Зазначимо, що у деяких випадках підінтегральна функція f_0 із (3.3.1) може явно не залежати від часу, тобто $f_0(x_1, \dots, x_n, u_1, \dots, u_r)$.

Верхня межа інтегрування T у (3.3.1) залежно від типу задачі може бути фіксованою або ні. У другому випадку цільовий функціонал залежить не тільки від керування u , але й від кінцевого моменту часу T . Тому деколи у таких випадках пишуть $I[u, T]$.

Отже *задача оптимального керування* для системи диференціальних рівнянь (3.1.1) полягає у максимізації функціоналу (3.3.1) на множині всіх допустимих керувань, які переводять систему (3.1.1) із заданого початкового стану у заданий кінцевий стан. З математичних міркувань це є спеціальною задачею оптимізації інтегрального функціоналу на визначеній множині функціонального простору кусково-неперервних керуючих функцій. Розв'язком такої задачі є керування $u^*(t) = (u_1^*(t), \dots, u_r^*(t))$, яке називають *оптимальним керуванням*. Цьому керуванню однозначно відповідає визначена траєкторія $x^*(t) = (x_1^*(t), \dots, x_n^*(t))$, яку називають *оптимальною траєкторією*. Пару векторних функцій $\{x^*(t), u^*(t)\}$ називають *оптимальним процесом*.

Розглянемо випадок, коли в (3.3.1) функція $f_0 \equiv -1$. Тоді

$$I[u] = \int_{t_0}^T (-1) dt = -(T - t_0)$$

і максимізація функціоналу I при фіксованому t_0 і нефіксованому T рівносильна мінімізації періоду керування $T - t_0$. Задачу керування у такому випадку називають *задачею оптимальної швидкодії*, в якій потрібно знайти керування, що переводить систему із одного стану в інший за найменш можливий час.

Функцію керування вибирають із спеціальної множини допустимих керувань, яка складається із кусково-неперервних вектор-функцій $u = u(t)$ на відрізку $[t_0, T]$ із значеннями у заданій допустимій області $U \in \mathbb{R}^r$.

Найпростішим способом задання допустимої області є

$$\alpha_i \leq u_i(t) \leq \beta_i, \quad i = \overline{1, r}, \quad \forall t \in [t_0, T]$$

де α_i, β_i ($i = \overline{1, r}$) – задані сталі.

3.4. Завдання для самостійного опрацювання

◆ *Завдання 3.4.1.* Система лінійних диференціальних рівнянь $dx/dt = Kx + Lu$ є цілком регулярною тоді і тільки тоді, коли $\text{rank}[L, KL, K^2L, \dots, K^{n-1}L]_{(n \times nr)} = n$.

Чи є система лінійних диференціальних рівнянь цілком регулярною, якщо

- а) $K = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, L = \begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix};$
- б) $K = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{pmatrix}, L = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$

◆ *Завдання 3.4.2.* Для системи із завдання 3.4.1 при $u_1 = u_2 = 0$ знайти траєкторію у фазовій площині, що проходить через точку $M_0(1, 0)$.

◆ *Завдання 3.4.3.* Сформулювати задачу мінімізації

$$T \rightarrow \min,$$

за умов

$$\begin{aligned} |\ddot{x}(t)| &= 2, & x(-1) &= 1, & x(T) &= -1 \\ \dot{x}(-1) &= \dot{x}(T) &= 0, \end{aligned}$$

у формі задачі оптимальної швидкодії. Описати допустиму область керувань.

◆ *Завдання 3.4.4.* Навести приклад економічної керованої системи.

◆ *Завдання 3.4.5.* Знайти розв'язок задачі

$$I[u] = \int_0^{t_1} x_1^2(t) dt \rightarrow \min,$$

за умов

$$\begin{cases} \dot{x}_1 = x_2 + u, \\ \dot{x}_2 = -u, \\ x_1(0) = x_2(0) = 0, \\ |u| \leq 1. \end{cases}$$

◆ *Завдання 3.4.6.* Задавши необхідні крайові умови, знайти розв'язок задачі

$$I[u] = \int_0^T \sqrt{1 + u^2(t)} dt \rightarrow \min,$$

за умов

$$\begin{cases} \dot{x}_1 = x_2, \\ \dot{x}_2 = u, \\ |u| \leq 1. \end{cases}$$

◆ *Завдання 3.4.7.* Виписати функціонал, який визначає віддаль між параболою $x(t) = t^2$ і прямою $x(t) = t - 5$.

◆ *Завдання 3.4.8.* Звести задачу оптимального керування диференціальним рівнянням другого порядку

$$\begin{cases} \ddot{x} = -\dot{x} - x + u, \\ x(0) = x_0, \\ \dot{x}(0) = 2, \\ |u| \leq 1, \end{cases}$$

до задачі оптимального керування системою диференціальних рівнянь.

із заданим початковим моментом часу $t = t_0$ і фіксованим скінченим моментом часу керування $t = T$, тобто

$$x(t_0) = x^{(0)} \in \mathbb{R}^n, \quad x(T) = x^{(1)} \in \mathbb{R}^n \quad (4.1.2)$$

Критерій якості керування запишемо у вигляді

$$I[u] = \int_{t_0}^T f_0(t, x_1(t), \dots, x_n(t), u_1(t), \dots, u_r(t)) dt, \quad (4.1.3)$$

із областю керування $U \subset \mathbb{R}^r$. Всі задані функції в (4.1.1)-(4.1.3) вважаємо достатньо гладкими, щоб сформулювати принцип максимуму.

Задача оптимального керування для системи (4.1.1) полягає у максимізації інтегрального функціоналу (4.1.3) на множині допустимих керувань, що переводять систему (4.1.1) із заданого початкового стану $x^{(0)}$ в початковий момент часу t_0 у заданий кінцевий стан $x^{(1)}$ у фіксований момент часу T . Розв'язком цієї задачі є *оптимальне керування* $u^*(t) = (u_1^*(t), \dots, u_r^*(t))$ і *оптимальна траєкторія* $x^*(t) = (x_1^*(t), \dots, x_n^*(t))$, які разом утворюють *оптимальний процес* $\{x^*(t), u^*(t)\}$.

Ідея принципу максимуму має багато спільного з відомим із курсу математичного аналізу критерієм екстремуму функцій багатьох змінних, хоча за формою значно відрізняється.

При формулюванні необхідних умов екстремуму для функцій багатьох змінних з обмеженнями-рівностями важливу роль відіграє функція Lagrange'a. У теорії оптимального керування подібну роль відіграє *функція Hamilton'a*, або *hamiltonian (гамільтоніан)*.

$$H(t, x, \hat{\psi}, u) = \sum_{i=0}^n \psi_i f_i, \quad (4.1.4)$$

де $x = (x_1, \dots, x_n)$, $u = (u_1, \dots, u_r)$, f_0, f_1, \dots, f_n функції із (4.1.1) і (4.1.3), а $\hat{\psi} = (\psi_0, \psi_1, \dots, \psi_n)$ називають *спряженими змінними*. Очевидно, що функція Hamilton'a є функцією $2n + r + 2$ змінних. Для

Спряжена система (4.1.5) є системою із n лінійних звичайних диференціальних рівнянь відносно n невідомих *спряжених функцій* $\psi_1, \psi_2, \dots, \psi_n$.

Принцип максимуму назагал є необхідною умовою оптимальності, тому якщо деяке допустиме керування задовольняє цей принцип, то воно не обов'язково буде оптимальним. Однак, при розв'язуванні конкретних задач принцип максимуму часто дозволяє однозначно визначити оптимальне керування, зокрема, якщо:

- а) наперед відомо існування оптимального керування;
- б) оптимальна траєкторія стану системи знайдена і єдина,

то такий процес $\{x^*(t), u^*(t)\}$ буде оптимальним.

★ **Приклад 4.1.1.** Розглянемо найпростішу задачу оптимального керування, для якої

$$\frac{dx}{dt} = u, \quad 0 \leq t \leq T,$$

де $x(t)$, $u(t)$ – скалярні функції. Вважатимемо, що $U = \mathbb{R}$ та задані крайові умови $x(0) = x(T) = 0$, причому зафіксовано початковий та кінцевий моменти часу. На множині допустимих керувань потрібно мінімізувати інтегральний функціонал

$$I[u] = \int_0^T (x^2(t) + u^2(t)) dt.$$

Очевидно, що $x^*(t) = 0$, $u^*(t) = 0$ задовольняє всі умови сформульованої задачі і надає мінімального значення цільовому функціоналу, тобто даний процес $\{x^*(t), u^*(t)\}$ є оптимальним.

Однак продемонструємо застосування принципу максимуму на цій простій задачі. Насамперед, оскільки говоримо про принцип максимуму, то потрібно розглянути функціонал

$$I[u] = - \int_0^T (x^2(t) + u^2(t)) dt.$$

Випишемо функцію *Hamilton'a*

$$H(t, x(t), \widehat{\psi}, u) = -\psi_0(x^2(t) + u^2(t)) + \psi u,$$

і спряжену систему

$$\frac{d\psi}{dt} = -\frac{dH}{dx} = 2\psi_0 x.$$

Розглянемо можливість вибору ψ_0 . У цьому випадку функціонал *Hamilton'a* $H = \psi u$ може досягати максимального за u значення на всій числовій осі \mathbb{R} лише при $\psi_0 = 0$. Але $\psi_0 = \psi = 0$ заперечує умову 1) теореми 4.1.1. Тому $\psi_0 \neq 0$. Легко бачити, що за такого випадку, розділивши функцію *Hamilton'a* і обидві частини спряженого рівняння на коефіцієнт $\psi_0 \neq 0$, доб'ємося, що $\psi_0^* = 1$. Тоді функція *Hamilton'a* прийме вигляд

$$H(t, x(t), \widehat{\psi}, u) = -x^2 - u^2 + \psi u.$$

На підставі теореми 4.1.1 ця функція на оптимальному розв'язку повинна досягати максимуму, тобто, застосувавши відому із математичного аналізу теорему *Fermat'a*, можна, прирівнявши до нуля похідну за u від H , а саме

$$\frac{\partial H}{\partial u} = -2u + \psi = 0, \quad \text{або} \quad u = \frac{\psi}{2}.$$

У підсумку одержимо

$$\begin{cases} \frac{dx}{dt} = \frac{1}{2}\psi, \\ \frac{d\psi}{dt} = 2x, \end{cases}$$

$$x(0) = x(T) = 0.$$

Звівши систему до рівняння другого порядку $\psi''(t) = \psi(t)$ знаходимо його загальний розв'язок

$$\psi(t) = C_1 e^t - C_2 e^{-t}.$$

в якій $\psi_0 = \psi_0^*$;

- 3) при кожному значенні $t \geq t_0$ всі компоненти керуючої вектор-функції $u^*(t)$ є неперервними, а функція Hamilton'a $H(t, x^*(t), \hat{\psi}^*(t), u)$ за векторною змінною $u = (u_1, \dots, u_r)$ досягає максимуму на множині U при $u = u^*(t)$, тобто

$$H(t, x^*(t), \hat{\psi}^*(t), u^*(t)) = \max_{u \in U} H(t, x^*(t), \hat{\psi}^*(t), u), \quad t \in [t_0, +\infty).$$

4.3. Схема застосування принципу максимуму

Розглянемо схему застосування принципу максимуму на прикладі задачі оптимального керування із закріпленими кінцями при $n = r = 2$.

Насамперед, розглянемо можливості щодо зміни числа ψ_0^* . На підставі принципу максимуму це число повинно бути невід'ємним.

Виділимо два випадки $\psi_0^* > 0$ і $\psi_0^* = 0$. Для першого варіанту, поділивши рівності (4.1.5)-(4.1.6) на ψ_0^* і ввівши нові спряжені змінні, які відрізняються від старих $\psi(t) = (\psi_1(t), \dots, \psi_n(t))$ множителем $(\psi_0^*)^{-1}$, одержимо рівності подібні до (4.1.5)-(4.1.6), у котрих $\psi_0^* = 1$.

Тому завжди можна обмежитися розглядом лише двох випадків: $\psi_0^* = 1$ або $\psi_0^* = 0$. У другому випадку функція Hamilton'a (4.1.4) не матиме доданку із функцією f_0 із критерія оптимальності (4.1.3). Це означає, що при $\psi_0^* = 0$ твердження принципу максимуму не містить жодної інформації про критерій, тобто, замінивши вихідний критерій оптимальності будь-яким іншим, одержимо такі ж умови оптимальності у формі принципу максимуму. Подібні задачі називають особливими і вони потребують додаткових досліджень [9, 26].

Застосування принципу максимуму зазвичай починають із побудови функції Hamilton'a

$$H = H(t, x_1, x_2, \psi_0, \psi_1, \psi_2, u_1, u_2)$$

і використання умови максимуму

$$H \rightarrow \max_{(u_1, u_2) \in U}.$$

Із цієї умови для фіксованого набору параметрів $t, x_1, x_2, \psi_1, \psi_2$ знаходимо керування $u = (u_1, u_2)$, яке також залежить від усіх зазначених параметрів і повертає можливе значення функції Hamilton'а H . Нехай

$$u = (u_1(t, x, \widehat{\psi}), u_2(t, x, \widehat{\psi})) \quad (4.3.1)$$

Зазвичай знайти керування у вигляді (4.3.1) не просто, однак для деякого класу задач керування функцію (4.3.1) вдається знайти у явному вигляді. Наприклад, у лінійному випадку

$$\begin{aligned} f_k(t, x, u) &= f_k(t, x) + f_{k1}(t, x)u_1 + f_{k2}(t, x)u_2, \quad k = 0, 1, 2, \\ U &= \{u = (u_1, u_2) : \alpha_i \leq u_i \leq \beta_i, \alpha_i, \beta_i - \text{const}, i = 1, 2\} \end{aligned}$$

функцію Hamilton'а можна записати так

$$\begin{aligned} H &= \psi_0 f_0(t, x) + \psi_1 f_1(t, x) + \psi_2 f_2(t, x) \\ &\quad + (\psi_0 f_{01}(t, x) + \psi_1 f_{11}(t, x) + \psi_2 f_{21}(t, x)) u_1 \\ &\quad + (\psi_0 f_{02}(t, x) + \psi_1 f_{12}(t, x) + \psi_2 f_{22}(t, x)) u_2. \end{aligned}$$

Ця функція завдяки лінійності за u і специфічному виборі області керування U досягає свого максимального значення тільки у граничних точках множини U , а саме

$$u_k = \begin{cases} \beta_k, & \psi_0 f_{0k}(t, x) + \psi_1 f_{1k}(t, x) + \psi_2 f_{2k}(t, x) > 0, \\ \alpha_k, & \psi_0 f_{0k}(t, x) + \psi_1 f_{1k}(t, x) + \psi_2 f_{2k}(t, x) < 0, \quad k = 1, 2. \end{cases}$$

Знайшовши функцію (4.3.1), підставляємо її у вихідну і спряжену системи:

$$\begin{cases} \frac{dx_1}{dt} = f_1(t, x_1, x_2, u_1(t, x, \widehat{\psi}), u_2(t, x, \widehat{\psi})), \\ \frac{dx_2}{dt} = f_2(t, x_1, x_2, u_1(t, x, \widehat{\psi}), u_2(t, x, \widehat{\psi})), \end{cases}$$

$$\left\{ \begin{array}{l} \frac{d\psi_1}{dt} = -\psi_0 \frac{\partial f_0(t, x, u(t, x, \hat{\psi}))}{\partial x_1} \\ \quad - \psi_1 \frac{\partial f_1(t, x, u(t, x, \hat{\psi}))}{\partial x_1} - \psi_2 \frac{\partial f_2(t, x, u(t, x, \hat{\psi}))}{\partial x_1}, \\ \frac{d\psi_2}{dt} = -\psi_0 \frac{\partial f_0(t, x, u(t, x, \hat{\psi}))}{\partial x_2} \\ \quad - \psi_1 \frac{\partial f_1(t, x, u(t, x, \hat{\psi}))}{\partial x_2} - \psi_2 \frac{\partial f_2(t, x, u(t, x, \hat{\psi}))}{\partial x_2}. \end{array} \right.$$

Одержана система із звичайних диференціальних рівнянь з невідомими функціями x_1 , x_2 , ψ_1 , ψ_2 (число ψ_0 не враховуємо). Загальний розв'язок такої системи буде виражатися через чотири довільні сталі, які конкретно виражаємо із крайових умов:

$$x_1(t_0) = x_1^{(0)}, x_2(t_0) = x_2^{(0)}, x_1(T) = x_1^{(1)}, x_2(T) = x_2^{(1)}.$$

Тоді знайдемо деякі функції $x_1^*(t)$, $x_2^*(t)$, $\psi_1^*(t)$, $\psi_2^*(t)$. Для визначення числа ψ_0^* потрібно встановити, який із двох випадків $\psi_0^* = 0$ або $\psi_0^* = 1$ підходить у цьому випадку. Звичайно, що потрібно врахувати умову принципу максимуму, при якій вектор-функція $\hat{\psi}^*(t) = (\psi_0^*, \psi_1^*(t), \psi_2^*(t))$ не повинна бути тотожно рівною нулеві на проміжку $t \in [t_0, T]$.

Знайшовши тепер всі функції $x^*(t)$, $\psi^*(t)$ та число ψ_0^* , підставивши у рівняння (4.3.1), одержимо

$$u^* = (u_1(t, x^*(t), \hat{\psi}^*(t)), u_2(t, x^*(t), \hat{\psi}^*(t))).$$

Якщо значення цієї двохкомпонентної функції не виходить за межі допустимої області U , то вона є екстремаллю функціонала якості та може претендувати на роль оптимального керування.

★ **Приклад 4.3.1.** Використовуючи викладену вище схему розглянемо задачу оптимального керування щодо максимізації функціона-

лу

$$I[u] = \int_0^t (x(t) + u^2(t)) dt \rightarrow \max,$$

якщо стан системи описується задачею

$$\begin{cases} \dot{x} = u, \\ x(0) = 0, x(T) = 0, |u(t)| \leq 1, t \in [0, T], \end{cases}$$

T – фіксоване. Функція Hamilton'а для цієї задачі і спряжене рівняння мають відповідно вигляди

$$\begin{aligned} H(t, x(t), \psi_0(t), \psi(t), u(t)) &= \psi_0(u^2 + x(t)) + \psi u, \\ \psi'(t) &= \psi_0. \end{aligned}$$

Якщо у спряженому рівнянні покласти $\psi_0 = 0$, то $\psi(t) = C = \text{const}$. Рівність $C = 0$ є неможливою завдяки умові **1** теорему 4.1.1. Якщо ж $C > 0$, то максимальне значення функції Hamilton'а $H = Cu$ досягається при $u = 1 \quad \forall t \in [0, T]$. Але тоді для задачі

$$\begin{cases} \dot{x} = u, \\ x(0) = 0, \end{cases}$$

при керуванні $u(t) \equiv 1$ одержимо траєкторію $x(t) = t$, для якої не виконується крайова умова $x(T) = 0$.

Аналогічно можна перевірити невиконання умови $C < 0$. Отож, $\psi_0 \neq 0$, тобто можна обмежитися випадком $\psi_0^* = 1$.

Із спряженого рівняння, при $\psi_0^* = 1$, знайдемо

$$\psi(t) = C_1 - t, \quad C_1 = \text{const}.$$

Таким чином, спряжена змінна ψ є лінійною функцією часу, а значить існує єдина точка, в якій вона міняє свій знак. Перепишемо функцію Hamilton'а із врахуванням $\psi_0^* = 1$, тобто

$$H = u^2 + x + \psi u.$$

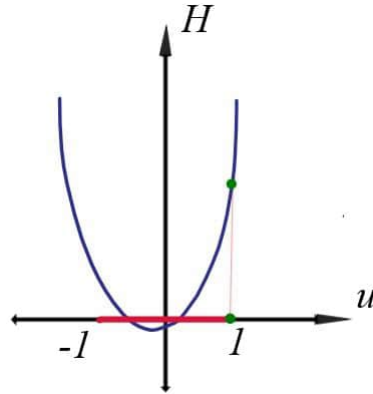


Рис. 4.1: Максимізація функції Неймана з прикладу 4.3.1

Це квадратична функція змінної u , графіком якої є парабола з вітками доверху. Отже, свого максимального значення на відрізку $[-1, 1]$ функція H може досягати лише у точках $u = \pm 1$. Якщо вершина параболи лежить у лівій півплощині, то максимум досягається на правому кінці, тобто при $u = 1$, а якщо у правій півплощині, то при $u = -1$ (див. рис. 4.1).

Абсциса вершини параболи знаходиться у точці $u = -\frac{\psi}{2}$. Лінійна функція $\psi = \psi(t)$ може змінювати один раз свій знак. Якщо $-\frac{\psi}{2} = -\frac{t - C_1}{2} > 0$, тобто $t > C_1$, то максимум функції H досягається при $u = 1$, а у випадку $t < C_1$ при $u = -1$.

Виходячи із диференціального рівняння $x' = u$, маємо, що керуванню $u(t) \equiv 1$ відповідає траєкторія $x(t) = t + C_2$, а керуванню $u(t) \equiv -1$, відповідно $x(t) = -t + C_2$.

Отож, враховуючи крайові умови $x(0) = 0$, $x(T) = 0$, приходимо до двох можливих варіантів:

- а) спочатку застосовується керування $u(t) \equiv 1$, а потім $u(t) \equiv -1$ (див. траєкторію а) на рис. 4.2);

- б) спочатку використовується керування $u(t) \equiv -1$, а потім $u(t) \equiv 1$ (див. траєкторію б) на рис. 4.2).

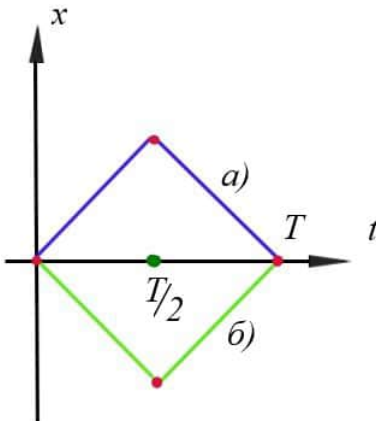


Рис. 4.2: Варіанти керування

варіант б) не задовольняє принцип максимуму, а єдиною екстремаллю в цій задачі є

$$u^*(t) = \begin{cases} 1, & t \leq \frac{T}{2}, \\ -1, & t > \frac{T}{2}. \end{cases}$$

Для виконання крайової умови $x(T) = 0$ потрібно, щоб у обох випадках точка \hat{t} , як перемикання значень керування була розміщена симетрично відносно кінців відрізка $[0, T]$, тобто $\hat{t} = T/2$, при цьому $C_1 = T/2$, а $\psi^*(t) = T/2 - t$.

Повернувшись до умови максимуму функції *Hamilton'a* $H(x, \psi_0, \psi, u)$ на відрізку $[0, T]$ бачимо, що для варіанту а) це значення $t + T/2$, а для випадку б) $t - T/2$. Очевидно, що варіант б) не задовольняє принцип максимуму, а єдиною екстремаллю в цій задачі є

4.4. Задача із нефіксованим часом керування

Повернемося до задачі керування системою (4.1.1) з умовами (4.1.2) та критерієм якості (4.1.3). Однак на відміну від задачі оптимального керування (4.1.1)-(4.1.3), вважатимемо кінцевий момент часу T не фіксованим, який потребує визначення.

Випишемо введenu раніше *функцію Hamilton'a*

$$H(t, x, \widehat{\psi}, u) = \sum_{i=0}^n \psi_i f_i(t, x, u). \quad (4.4.1)$$

Задача оптимального керування із нефіксованим скінченим моментом часу полягає у максимізації інтегрального функціоналу (4.1.2) на множині всіх допустимих керувань, що переводять систему (4.1.1) із заданого початкового стану $x^{(0)}$ у заданий кінцевий стан $x^{(1)}$ за деякий наперед невідомий проміжок часу T .

Розв'язком цієї задачі є оптимальне керування $u^*(t) = (u_1^*(t), \dots, u_n^*(t))$, відповідна оптимальна траєкторія $x^*(t) = (x_1^*(t), \dots, x_n^*(t))$ і оптимальний час T .

Звичайно, що принцип максимуму для цієї задачі, на відміну від відповідного принципу для задачі із фіксованим часом, потрібно підкоректувати для визначення оптимального часу T^* .

Теорема 4.4.1 ([9, 11]). *Нехай $\{x^*(t), u^*(t)\}$ – оптимальний процес у задачі оптимального керування з фіксованим часом керування системою (4.1.1)-(4.1.3).*

Тоді існують невід'ємне число ψ_0^ і неперервна функція $\psi^*(t) = (\psi_1^*(t), \dots, \psi_n^*(t))$ такі, що виконуються умови 1)-3) теореми 4.1.1 при $T = T^*$ і крім того, справедлива рівність*

$$\max_{u \in U} H(T, x^*(T), \widehat{\psi}^*(T), u) \Big|_{T=T^*} = 0. \quad (4.4.2)$$

Застосуємо **теорему 4.4.1** для знаходження розв'язку задачі на швидкодію.

◆ *Завдання 4.4.1.* Нехай матеріальна точка з координатою $x(t)$ рухається вздовж осі OX за законом

$$x''(t) = u(t), \quad t \geq 0.$$

Потрібно знайти кусково-неперервне керування з умовою

$$|u(t)| \leq 1, \quad 0 \leq t \leq T,$$

таке, щоб матеріальна точка, вийшовши з нульовою швидкістю із початкового стану

$$x(0) = 1, \quad x'(0) = 0,$$

попала в початок координат з нульовою швидкістю за найкоротший час T .

Розв'язування цього завдання розпочнемо із зведення рівняння до системи рівнянь першого порядку, увівши нові фазові змінні

$$z_1(t) = x(t), \quad z_2(t) = x'(t).$$

Тоді

$$\begin{cases} z_1' = z_2, \\ z_2' = u, \quad t \geq 0, \\ z_1(0) = 1, \quad z_2(0) = 0, \\ z_1(T) = z_2(T) = 0, \quad U = [-1, 1], \end{cases}$$

з максимізацією функціоналу

$$I[u] = - \int_0^T dt = -T.$$

Для функції *Hamilton'a*

$$H(t, x, \hat{\psi}, u) = -\psi_0 + \psi_1 z_2 + \psi_2 u$$

випишемо спряжену систему для всіх $t \geq 0$

$$\begin{cases} \psi_1' = -\frac{\partial H}{\partial z_1} = 0, \\ \psi_2' = -\frac{\partial H}{\partial z_2} = -\psi_1. \end{cases}$$

Звідки знаходимо $\psi_1(t) = C$, $\psi_2(t) = -Ct + K$, де $C, K = \text{const}$. Якщо припустити, що $\psi_2(t) \equiv 0$, то $C = K = 0$ і $\psi_1(t) \equiv 0$. Але тоді з умови

$$\max_{u \in [-1, 1]} (-\psi_0 + C - (Ct - K)u) = 0,$$

маємо $\psi_0 = 0$, що разом з одержаними $\psi_1(t) = \psi_2(t) \equiv 0$ протиріччя з умовою 1) принципу максимуму. Отож, $\psi_2(t) \neq 0$.

Оскільки записана під знаком максимуму функція є лінійною, то максимальне значення може приймати лише на кінцях відрізка $[-1, 1]$:

$$u(t) = \text{sign}(K - Ct).$$

Тепер бачимо, що оптимальне керування при його існуванні є кусково-неперервною функцією і приймає значення ± 1 , причому із за лінійності $\psi_2(t) = K - Ct$, керування має не більше однієї точки \check{t} зміни знаків (\check{t} – точка перемикання).

Легко перекоонатися, що траєкторії, які виходять із початкової точки та відповідні режими керування при $t \geq 0$:

а) $u(t) = -1$;

б) $u(t) = 1$;

в)
$$\begin{cases} 1, & 0 \leq t \leq \check{t}, \\ -1, & t > \check{t}, \end{cases}$$

ніколи не проходять через кінцеву точку $(0, 0)$.

Залишилося дослідити єдиний можливий режим

$$u(t) = \begin{cases} -1, & 0 \leq t \leq \check{t}, \\ 1, & t > \check{t}, \end{cases}$$

якому відповідатиме траєкторія $(z_1(t), z_2(t))$, де

$$z_1(t) = \begin{cases} 1 - \frac{t^2}{2}, & 0 \leq t \leq \check{t}, \\ \frac{t^2}{2} - 2\check{t}t + \check{t}^2 + 1, & t > \check{t}, \end{cases}$$

$$z_2(t) = \begin{cases} -t, & 0 \leq t \leq \check{t}, \\ t - 2\check{t}, & t > \check{t}. \end{cases}$$

З умов $z_1(T) = z_2(T) = 0$ одержимо $\check{t} = 1$, $T = T^* = 2$. Тому траєкторії можна переписати у вигляді

$$z_1(t) = \begin{cases} 1 - \frac{t^2}{2}, & 0 \leq t \leq 1, \\ \frac{(t-2)^2}{2}, & 1 < t \leq 2, \end{cases} \quad z_2(t) = \begin{cases} -t, & 0 \leq t \leq 1, \\ t-2, & 1 < t \leq 2, \end{cases}$$

при цьому $\psi_0^* = 0$, $\psi_1^*(t) = -1$, $\psi_2^*(t) = t - 1 \quad \forall t \in [0, 2]$. Такий процес задовольняє умови принципу максимуму і є єдиним.

Отже, відповідно до принципу максимуму, для того, щоб за найкоротший час із точки, яка розміщена на віддалі одиниці справа від початку координат, потрапити у початок координат потрібно у початковий момент часу розпочати рух вліво з максимальним прискоренням ($u(t) = -1$). Після одиниці часу потрібно перемкнути керування у режим $u(t) = +1$ (відповідає режиму максимального гальмування). Тоді через одиницю часу після перемикання матеріальна точка попадає у початок координат з нульовою кінцевою швидкістю.

4.5. Задача оптимального керування з рухомими кінцями

Часто виникають задачі керування, для яких один або обидва кінці траєкторії, що відповідає допустимому керуванню, не є жорстко закріпленими, тобто можуть вибиратися із деяких заданих множин. У таких випадках задачі керування називають *задачами із рухомими кінцями*.

Насамперед розглянемо задачу оптимального керування з рухомим правим кінцем, у якій функціонал якості потрібно максимізувати на множині всіх допустимих керувань, що переводять керуючу систему із заданої точки $x(t_0)$ фазового простору у довільну точку заданої множини $G \subset \mathbb{R}^n$ (див. рис. 4.3).

Якщо $G = \mathbb{R}^n$, то задачу називають *задачею із вільним правим кінцем*.

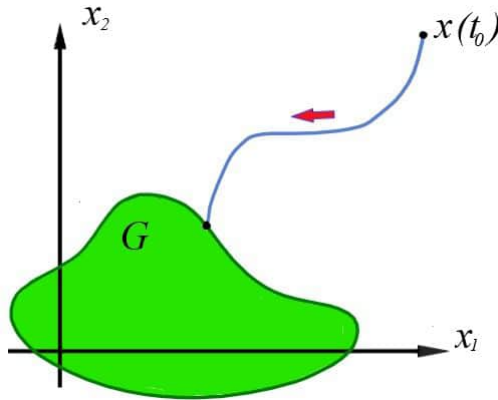


Рис. 4.3: Перехід стану системи із точки на множину.

Аналогічно можна говорити про *задачу керування із лівим рухомим кінцем* або про *обидва рухомі кінці траєкторії*.

Нехай обмеження на початковий $x(t_0)$ або кінцевий $x(T)$ стани системи задано у вигляді

$$\begin{cases} g_i(x(t_0), x(T)) \geq 0, & i = \overline{1, m}, \\ g_j(x(t_0), x(T)) = 0, & j = \overline{m+1, M}. \end{cases} \quad (4.5.1)$$

Якщо $m = 0$, то відсутні обмеження-нерівності, а коли $M = m$, то немає обмежень-рівностей. Будемо вважати, що функції g_i і g_j задовольняють всім вимогам, необхідним для виконання принципу максимуму (див. [9]).

Задача оптимального керування з рухомими кінцями полягає у тому, що задано систему (4.1.1*), функціонал (4.1.2*) і серед усіх допустимих керувань, котрі переводять систему (4.1.1*) із деякої початкової точки $x(t_0)$, яка задовольняє обмеження (4.5.1), на множину таких кінцевих станів $x(T)$, для яких також має місце (4.5.1), потрібно знайти таке керування, яке надає найбільше можливе значення інтегральному функціоналові (4.1.2*). Визначенню підлягають також і кінці оптимальної траєкторії, тобто $x^*(t_0)$ і $x^*(T)$. При цьому час може бути фіксо-

ваним або нефіксованим.

Останню задачу оптимального керування з фіксованими кінцями можна одержати із задачі оптимального керування з нефіксованими кінцями, то основна частина принципу максимуму для задачі із рухомими кінцями буде такою ж, як і для задачі із фіксованими кінцями.

Зміна формулювання необхідної умови оптимальності у формі принципу максимуму для задачі із рухомими кінцями полягає (див. [9]) у заміні крайових умов $x(t_0) = x^{(0)}$, $x(T) = x^{(1)}$ для кінців оптимальної траєкторії на умови

$$\psi_i^*(t_0) = \sum_{j=1}^M \alpha_j \frac{\partial g_j(x(t_0), x(T))}{\partial x_i(t_0)} \Bigg|_{(x^*(t_0), x^*(T))}, \quad i = \overline{1, n},$$

$$\psi_i^*(T) = - \sum_{j=1}^M \alpha_j \frac{\partial g_j(x(t_0), x(T))}{\partial x_i(T)} \Bigg|_{(x^*(t_0), x^*(T))}, \quad i = \overline{1, n},$$

які називаються *умовами трансверсальності*, та *умови додаткової (доповнюючої) нежорсткості*

$$\alpha_j g_j(x^*(t_0), x^*(T)) = 0, \quad j = \overline{1, m},$$

для деяких, що підлягають визначенню, невід'ємних чисел $\alpha_1, \dots, \alpha_m$ та чисел довільного знаку $\alpha_{m+1}, \dots, \alpha_M$. Ці числа назагал володіють властивістю

$$(\psi_0^*)^2 + \sum_{s=1}^M \alpha_s^2 \neq 0.$$

У випадку, коли один із кінців траєкторії є закріпленим, то *умова трансверсальності* на закріпленому кінці є зайвою.

Зазначимо, що у підсумку одержимо $2n + M$ співвідношень для визначення невідомих компонент $x_1^*(t_0), \dots, x_n^*(t_0), x_1^*(T), \dots, x_n^*(T)$ і сталих $\alpha_1, \dots, \alpha_M$.

Розглянемо тепер деякі найпростіші випадки *умов трансверсальності* та *умов додаткової нежорсткості*. Випадок задачі керування

із вільним правим кінцем, наприклад, може бути при $x(t_0) = x^{(0)}$, із обмеженнями-рівностями

$$g_j(x(t_0), x(T)) = x_j(t_0) - x_j(T) = 0, \quad j = \overline{1, n}.$$

Тут немає обмежень-нерівностей, то умови доповнюючої нежорсткості є відсутніми. Оскільки лівий кінець є закріплений, то умови трансверсальності потрібно записати тільки для кінцевого моменту часу $t = T$, причому жодна із функцій g_j не залежить від $x_i(T)$. А це означає, що всі похідні в умовах трансверсальності від функцій g_j за $x_i(T)$ дорівнюють нулеві, тобто *для задачі оптимального керування із вільним правим кінцем умова трансверсальності має вигляд*

$$\psi^*(T) = 0.$$

У випадку закріпленого лівого кінця траєкторії, а для правого виконуються нерівності

$$x_i(T) \geq a_i, \quad i = \overline{1, n},$$

де a_i – задані сталі, то обмеження-нерівності запишемо, наприклад, у вигляді

$$g_i(x(t_0), x(T)) = x_i(T) - a_i \geq 0, \quad i = \overline{1, n}.$$

Тепер для такого випадку умови трансверсальності і додаткової нежорсткості на правому кінці траєкторії мають вигляд:

$$\begin{aligned} \psi_i^*(T) &= \alpha_i, \quad i = \overline{1, n}, \\ \alpha_i(x_i^*(T) - a_i) &= 0, \quad i = \overline{1, n}. \end{aligned}$$

Завдяки тому, що $\forall i, i = \overline{1, n}: \alpha_i \geq 0$, то у підсумку одержимо

$$\psi_i^*(T) \geq 0, \quad \psi_i^*(T)(x_i^*(T) - a_i) = 0, \quad i = \overline{1, n}.$$

Отож, можемо стверджувати, що загальна схема застосування принципу максимуму для задачі із рухомими кінцями є такою ж, як і для задачі із закріпленими кінцями. Однак, через умови трансверсальності і додаткової нежорсткості значно збільшується кількість шуканих величин.

4.6. Економічна інтерпретація функцій спряженої системи

Припустимо, що деякий суб'єкт господарювання ставить собі за мету одержати максимальний прибуток впродовж періоду $[0, T]$. Нехай єдиною змінною стану є капітал $K = K(t)$, а керуючою змінною є $u = u(t)$ (наприклад, кошти втрачені на рекламу, застосування нових технологій тощо). У кожний момент часу t прибуток залежить від величини капіталу K і керування u . Тому через функцію прибутку $\pi = \pi(t, K, u)$ буде виражатися сумарний прибуток упродовж $[0, T]$, а саме

$$\Pi[u] = \int_0^T \pi(t, K(t), u(t)) dt \rightarrow \max. \quad (4.6.1)$$

Швидкість зміни величини капіталу буде виражатися через деяку функцію, що залежить від K і u

$$\frac{dK}{dt} = f(t, K, u), \quad t \in [0, T], \quad (4.6.2)$$

$$K(0) = K_0, \quad (4.6.3)$$

а скінчений момент часу T забезпечує вільність траєкторії $K(t)$.

Перейдемо до опису спряженої змінної $\psi = \psi(t)$ і з'ясуємо економічний зміст $\psi^* = \psi^*(t)$.

Обмежимося випадком $\psi_0^* = 1$ і запишемо для оптимального процесу функцію Hamilton'а, тобто

$$H(t, x^*, u^*, \psi^*) = \pi(t, K^*, u^*) + \psi^* f(t, K^*, u^*)$$

Врахувавши (4.6.2)-(4.6.3) цільовий функціонал можна записати

$$\Pi^*[u] = \int_0^T \left[\pi(t, K^*(t), u^*(t)) + \psi^*(t) \left(f(t, K^*(t), u^*(t)) - \frac{dK^*(t)}{dt} \right) \right] dt,$$

або врахувавши функції Hamilton'а, одержимо

$$\begin{aligned}
 \Pi^*[u] &= \int_0^T \left(H(t, K^*(t), u^*(t), \psi^*(t)) - \psi^*(t) \frac{dK^*(t)}{dt} \right) dt \\
 &= \int_0^T H(t, K^*(t), u^*(t), \psi^*(t)) dt - \int_0^T \psi^*(t) \frac{dK^*(t)}{dt} dt \\
 &= \int_0^T H(t, K^*(t), u^*(t), \psi^*(t)) dt - \psi^*(t) K^*(t) \Big|_0^T \\
 &\quad + \int_0^T K^*(t) \frac{d\psi^*(t)}{dt} dt \\
 &= \int_0^T \left(H(t, K^*(t), u^*(t), \psi^*(t)) + K^*(t) \frac{d\psi^*(t)}{dt} \right) dt \\
 &\quad - \psi^*(T) K^*(T) + \psi^*(0) K_0.
 \end{aligned}$$

Обчислимо частинні похідні

$$\frac{\partial \Pi^*}{\partial K_0} = \psi^*(0), \quad \frac{\partial \Pi^*}{\partial K^*(T)} = -\psi^*(T). \quad (4.6.4)$$

З першої рівності (4.6.4) можна стверджувати, що $\psi^*(0)$ є мірою зміни оптимального сумарного прибутку стосовно заданого початкового капіталу K_0 . Це означає, що коли збільшити початковий капітал на умовну одиницю, то величина сумарного прибутку зросте на величину $\psi^*(0)$.

Тобто, $\psi^*(0)$ можна розглядати як тіньову ціну одиниці початкового капіталу.

Аналогічно із другого співвідношення (4.6.4) бачимо, що $\psi^*(T)$ є від'ємною швидкістю зміни сумарного прибутку Π^* .

Отож, щоб зберегти (зменшити на одиницю) умовну одиницю капіталу на кінці T , то потрібно зменшити сумарний прибуток на величину $\psi^*(T)$. А це означає, що $\psi^*(T)$ є *тіньовою ціною одиниці кінцевого капіталу*.

Назагал, в рамках цієї задачі, величину $\psi^*(t)$ потрібно розглядати як *тіньову ціну капіталу у біжучий момент часу t* .

Оскільки правий кінець траєкторії є вільним, то умова трансверсальності на правому кінці має вигляд $\psi^*(T) = 0$, а це означає, що для оптимального процесу тіньова ціна капіталу у кінцевий момент часу дорівнює нулеві (втратила свою цінність).

Для суб'єкта господарювання, який хоче продовжити роботу після часу $t = T$, розумно було б зарезервувати деяку мінімально можливу кількість капіталу, тобто повинна виконуватися умова $K(T) \geq K_{\min}$. У такому випадку умови трансверсальності та додаткової нежорсткості мають вигляд

$$\psi^*(T) \geq 0, \quad \psi^*(T)(K^*(T) - K_{\min}) = 0.$$

Аналіз цих умов показує, що коли можливе перевищення $K^*(T)$ над граничним значенням K_{\min} , то *тіньова ціна* $\psi^*(T) = 0$. Але, поки $\psi^*(T) > 0$ (капітал повністю не використаний), то обов'язково повинно бути, що $K^*(T) = K_{\min}$, тобто діючий капітал у кінцевий момент часу потрібно використати до мінімально можливого.

4.7. Ілюстрація варіаційного принципу максимуму для модельного прикладу щодо гіперболічної системи двох рівнянь

Нехай у прямокутнику $S \times T$, $S = [s_n, s_k]$, $T = [t_0, t_1]$, керуючий процес представлений мішаною задачею

$$\begin{cases} \frac{\partial x_1}{\partial t} + \lambda \frac{\partial x_1}{\partial s} = \alpha(s)(x_1 - x_2), \\ \frac{\partial x_2}{\partial t} - \lambda \frac{\partial x_2}{\partial s} = \beta(s)(x_1 - x_2), \end{cases}$$

$$x_1(s, t_0) = u(s) + q(s), \quad x_2(s, t_0) = u(s) - q(s), \quad s \in S.$$

Допустиме керування $u(s)$ задовольняє обмеження

$$u \in U \subset \mathbb{R}, \quad s \in S.$$

Метою задачі є мінімізація функціоналу

$$I[u] = \int_S (x_1(s, t) + x_2(s, t) - \eta(s))^2 ds.$$

Вважаємо, що функції α , β , q , η і додатна стала λ є заданими. Додатково вимагаємо виконання умови

$$s_k - s_n > 2\lambda(t_1 - t_0), \quad (4.7.1)$$

до якої повернемося нижче.

Подібні задачі оптимального керування виникають при дослідженні процесів збурення і поширення гравітаційних хвиль [1].

У цій моделі два сімейства характеристик визначаються розв'язками відповідних задач Cauchy

$$\begin{cases} \frac{ds}{dt} = \pm \lambda, \\ s(t_0) = \xi, \end{cases}$$

тобто $s = \lambda(t - t_0) + \xi$ та $s = \lambda(t_0 - t) + \xi$.

Нехай $\{u, x\}$ – оптимальний процес, а $\psi(s, t)$ – розв'язок спряженої задачі

$$\begin{cases} \frac{\partial \psi_1}{\partial t} + \lambda \frac{\partial \psi_1}{\partial s} = -\alpha \psi_1 - \beta \psi_2, \\ \frac{\partial \psi_2}{\partial t} - \lambda \frac{\partial \psi_2}{\partial s} = \alpha \psi_1 + \beta \psi_2, \end{cases}$$

$$\psi_1(s, t_1) = \psi_2(s, t_1) = 2(\eta(s) - x_1(s, t_1) - x_2(s, t_1)), \quad s \in S,$$

$$\psi_1(s_k, t) = \psi_2(s_n, t) = 0, \quad t \in T.$$

Виходячи із принципу максимуму

$$I[u(\xi), \xi] = \max_{v \in U} I[v, \xi], \quad s \in S,$$

маємо, що умови цього принципу для розглядуваного функціоналу виконуються на відрізку S . Однак вигляд функціоналу $I[v, \xi]$ залежить від розміщення точки ξ на відрізку S . Нехай

$$\text{а) } s_n + (t_1 - t_0)\lambda \leq \xi \leq s_k - (t_1 - t_0)\lambda.$$

Кінцевими точками цих характеристик є точки відрізка $\{(s, t) : s \in S, t = t_1\}$. Для цього випадку одержимо

$$\begin{aligned} I[v, \xi] &= -[y_1(t_1) + x_2(\lambda(t_0 - t_1) + \xi, t_1) - \eta(\lambda(t_0 - t_1) + \xi)]^2 \\ &+ \int_{t_0}^{t_1} \psi_2(\lambda(t_0 - t) + \xi, t) \cdot \beta(\lambda(t_0 - t) + \xi) [y_1(t) - x_2(\lambda(t_0 - t) + \xi, t)] dt \\ &- [y_2(t_1) + x_1(\lambda(t_1 - t_0) + \xi, t_1) - \eta(\lambda(t_1 - t_0) + \xi)]^2 \\ &+ \int_{t_0}^{t_1} \psi_1(\lambda(t - t_0) + \xi, t) \cdot \alpha(\lambda(t - t_0) + \xi) [x_1(\lambda(t - t_0) + \xi, t) - y_2(t)] dt, \end{aligned}$$

$$\begin{cases} \frac{dy_1}{dt} = \alpha(\lambda(t_0 - t) + \xi) [y_1(t) - x_2(\lambda(t_0 - t) + \xi, t)], & t \in T, \\ \frac{dy_2}{dt} = \beta(\lambda(t - t_0) + \xi) [x_1(\lambda(t - t_0) + \xi, t) - y_2(t)], & t \in T, \\ y_1(t_0) = \delta(\xi) + q(\xi), & y_2(t_0) = \delta(\xi) - q(\xi). \end{cases} \quad (4.7.2)$$

Якщо

$$\bar{6}) \quad s_n \leq \xi \leq s_n + (t_1 - t_0)\lambda,$$

то у цьому випадку характеристика першого сімейства, що проходить через точку (ξ, t_0) , має кінцеву точку $\left(s_n, t_0 + \frac{\xi - s_n}{\lambda}\right)$, а характеристика другого сімейства закінчується у точці $(\lambda(t_1 - t_0) + \xi, t_1)$. Тоді

$$\begin{aligned} I[v, \xi] = & \int_{t_0}^{t_0 + \frac{\xi - s_n}{\lambda}} \psi_2(\lambda(t - t_0) + \xi, t) \cdot \beta(\lambda(t - t_0) + \xi) \\ & \times [y_1(t) - x_2(\lambda(t - t_0) + \xi, t)] dt \\ & - [y_2(t_1) + x_2(\lambda(t_1 - t_0) + \xi, t_1) - \eta(\lambda(t_1 - t_0) + \xi)]^2 \\ & + \int_{t_0}^{t_1} \psi_1(\lambda(t - t_0) + \xi, t) \cdot \alpha(\lambda(t - t_0) + \xi) \\ & \times [x_1(\lambda(t - t_0) + \xi, t) - y_2(t)] dt. \end{aligned}$$

Накінєць при в) $s_k - (t_1 - t_0)\lambda < \xi \leq s_k$

маємо, що цей випадок симетричний до випадку **б**). Характеристика другого сімейства має кінцеву точку $(s_k, t_0 + (s_k - \xi)/\lambda)$, а характери-

стика першого сімейства закінчується у точці $(\lambda(t_0 - t_1) + \xi, t_1)$. Тоді

$$\begin{aligned} I[v, \xi] = & - [y_1(t_1) + x_2(\lambda(t_0 - t_1) + \xi, t_1) - \eta(\lambda(t_0 - t_1) + \xi)]^2 \\ & + \int_{t_0}^{t_1} \psi_2(\lambda(t_0 - t) + \xi, t) \cdot \beta(\lambda(t_0 - t) + \xi) \\ & \times [y_1(t) - x_2(\lambda(t_0 - t) + \xi, t)] dt \\ & + \int_{t_0}^{t_0 + \frac{s_k - \xi}{\lambda}} \psi_1(\lambda(t - t_0) + \xi, t) \cdot \alpha(\lambda(t - t_0) + \xi) \\ & \times [x_1(\lambda(t - t_0) + \xi, t) - y_2(t)] dt. \end{aligned}$$

У всіх трьох випадках функції $y_1(t)$, $y_2(t)$ є розв'язками *задачі Cauchy* (4.7.2) на відповідних відрізках визначення $t \in [t_0, t_1]$. Цей модельний приклад підкреслює складність задач оптимального керування для гіперболічних систем, оскільки кожна із задач для функціоналу $I[v, \xi]$ не розпадається на задачі оптимізації для окремих звичайних диференціальних рівнянь, побудованих вздовж характеристик гіперболічної системи і, крім того, кожне із диференціальних рівнянь розглядається на окремих відрізках незалежних змінних.

Зазначимо також, що умова (4.7.1) гарантує, що характеристики гіперболічної системи, які виходять із точок $(0, \tau)$, $\tau \in [t_0, t_1]$ матимуть кінцеві точки на відрізку $\{(s, t) : s \in S, t = t_1\}$.

4.8. Рівняння Bellman'а

Принцип максимуму при розв'язуванні задач оптимального керування дає необхідні умови оптимальності. Цей недостаток принципу стимулював багатьох дослідників для одержання сильніших умов оптимізації, зокрема, достатніх. Над цією проблемою у 60-их роках минулого

століття працювала група американських вчених під керівництвом R. Bellman'a, які запропонували новий розділ оптимального керування під назвою *динамічне програмування* [9].

В основу ідеї динамічного програмування було закладено простий принцип оптимальності, який у найпростішому варіанті звучить так: будь-який правий кусок оптимальної траєкторії також є оптимальною траєкторією для відповідної задачі оптимального керування.

Методика одержання оптимальності у динамічному програмуванні не використовує варіацію керування та траєкторії і дає достатні умови оптимальності.

Отож, розглянемо задачу про мінімізацію функціоналу

$$I[x(t), u(t)] = \int_0^T f_0(t, x(t), u(t)) dt + F(x(t)), \quad (4.8.1)$$

за умов

$$\frac{dx(t)}{dt} = f(t, x(t), u(t)), \quad x(0) = x_0, \quad (4.8.2)$$

$$u(t) \in U. \quad (4.8.3)$$

Нехай $\{x^*(t), u^*(t)\}$ – єдина мінімаль.

Нехай $t \in (0, T)$ і розглянемо задачу про мінімізацію функціоналу

$$I_1[x(\tau), u(\tau)] = \int_t^T f_0(\tau, x(\tau), u(\tau)) d\tau + F(x(\tau)) \quad (4.8.4)$$

на множині із пар функцій $\{x(\tau), u(\tau)\}$ з умовами

$$\frac{dx(t)}{d\tau} = f(\tau, x(\tau), u(\tau)), \quad x(t) = x, \quad (4.8.5)$$

$$u(\tau) \in U. \quad (4.8.6)$$

Нехай $\{\bar{x}(\tau), \bar{u}(\tau)\}$, $\tau \in (t, T]$ – мінімаль задачі (4.8.4)-(4.8.6). Тоді функцію

$$\mathcal{B}(t, x) = \min_{???} I_1[x(\tau), u(\tau)] \quad (4.8.7)$$

називають *функцією Bellman'a*.

Принцип оптимальності Bellman'a полягає у наступному. Якщо $x = x^*(t)$ (точка x лежить на оптимальній траєкторії задачі (4.8.1)-(4.8.3)), то мінімаль задачі (4.8.4)-(4.8.6) співпадає із мінімаллю задачі (4.8.1)-(4.8.3) на відрізку $[t, T]$:

$$\bar{x}(\tau) = x^*(\tau), \quad \bar{u}(\tau) = u^*(\tau) \quad \text{при } \tau \in [t, T].$$

доведення цього факту можна знайти в [10, 9].

Припустимо тепер, що функція Bellman'a $\mathcal{B}(t, x)$, визначена (4.8.7), неперервно диференційовна за обома змінними. Тоді на підставі формули Taylor'a можна записати

$$\begin{aligned} \mathcal{B}(t + \Delta t, x + \Delta x) &= \mathcal{B}(t, x) + \frac{\partial \mathcal{B}(t, x)}{\partial t} \Delta t \\ &+ \frac{\partial \mathcal{B}(t, x)}{\partial x} \Delta x + o(\Delta t, \Delta x). \end{aligned}$$

Відповідно до принципу оптимальності одержуємо:

$$\begin{aligned} \mathcal{B}(t, x) &= \min_{x, u} \left\{ \int_t^{t+\Delta t} f_0(\tau, x(\tau), u(\tau)) d\tau \right. \\ &+ \left. \int_{t+\Delta t}^T f_0(\tau, x(\tau), u(\tau)) d\tau + F(x(T)) \right\} \\ &= \min_{x, t??} \left\{ \int_t^{t+\Delta t} f_0(\tau, x(\tau), u(\tau)) d\tau + \mathcal{B}(t + \Delta t, x + \Delta x) \right\}. \end{aligned} \quad (4.8.8)$$

Приріст Δx визначаємо із (4.8.5), тобто

$$\Delta x = f(t, x, u) \Delta t + o(\Delta t). \quad (4.8.9)$$

Враховуючи малість відрізка Δt , можемо записати

$$\int_t^{t+\Delta t} f_0(\tau, x(\tau), u(\tau)) d\tau = f_0(t, x, u)\Delta t + o(\Delta t)/$$

Тепер, на підставі (4.8.9) і формули Taylor'а, перепишемо (4.8.8) у вигляді

$$\mathcal{B}(t, x) = \min_{u \in U} \left\{ f_0(t, x, u)\Delta t + \mathcal{B}(t, x) + \frac{\partial \mathcal{B}(t, x)}{\partial t} \Delta t + \frac{\partial \mathcal{B}(t, x)}{\partial x} f(t, x, u)\Delta t + o(\Delta t) \right\},$$

або

$$\min_{u \in U} \left\{ f_0(t, x, u)\Delta t + \frac{\partial \mathcal{B}(t, x)}{\partial t} \Delta t + \frac{\partial \mathcal{B}(t, x)}{\partial x} f(t, x, u)\Delta t + o(\Delta t) \right\} = 0.$$

Поділимо обидві сторони останньої рівності на Δt та здійснимо граничний перехід $\Delta t \rightarrow 0$. Після чого одержимо

$$-\frac{\partial \mathcal{B}(t, x)}{\partial t} = \min_{u \in U} \left\{ f_0(t, x, u) + \frac{\partial \mathcal{B}(t, x)}{\partial x} f(t, x, u) \right\}. \quad (4.8.10)$$

Це рівняння з частинними похідними першого порядку відносно функції $\mathcal{B}(t, x)$ називають *рівнянням Bellman'а*.

Із визначення (4.8.7) *функції Bellman'а* маємо початкову умову

$$\mathcal{B}(T, x) = F(x). \quad (4.8.11)$$

Повернемося до рівняння (4.8.10), яке є незвичним, оскільки у його правій частині присутня операція мінімізації. Зазвичай, спочатку мінімізують вираз

$$f_0(t, x, u) + pf(t, x, u) \rightarrow \min_{u \in U} \quad (4.8.12)$$

при довільних (t, x, p) , $p = \frac{\partial \mathcal{B}}{\partial x}$ і нехай мінімум досягається для $u = \tilde{U}(t, x, p)$. Тоді (4.8.10) запишеться у вигляді

$$-\frac{\partial \mathcal{B}}{\partial t} = f_0 \left(t, x, \tilde{U} \left(t, x, \frac{\partial \mathcal{B}}{\partial x} \right) \right) + \frac{\partial \mathcal{B}}{\partial x} f \left(t, x, \tilde{U} \left(t, x, \frac{\partial \mathcal{B}}{\partial x} \right) \right). \quad (4.8.13)$$

Якщо знайти розв'язок нелінійного рівняння з частинними похідними (4.8.13) з умовою (4.8.11), то оптимальне керування виражається через (4.8.12) таким чином

$$u^*(t, x) = \tilde{U} \left(t, x, \frac{\partial \mathcal{B}(t, x)}{\partial x} \right). \quad (4.8.14)$$

Однак, у цього керування є залежність не тільки від часу t , але й від стану системи x . Такий вигляд оптимального керування називають *оптимальним синтезом* або *оптимальним керуванням у формі зворотного зв'язку*.

Зазначимо, що перехід до $u^*(t)$ є доволі простий. Для цього у задачі Cauchy (4.8.2) потрібно підставити замість $u(t)$ функцію (4.8.14), після чого розв'язок задачі Cauchy (4.8.2) визначає оптимальну траєкторію $x^*(t)$, тобто $u^*(t) = u^*(t, x(t))$.

Процедура знаходження оптимального керування через розв'язування рівняння Bellman'а говорить про те, що маємо справу із *достатньою умовою оптимальності*.

Визначення функції Bellman'а у багатьох випадках є непростою математичною процедурою. Більш-менш просто знаходити функцію Bellman'а коли, наприклад, розглядати лінійно-квадратичний клас задач оптимального керування, тобто, коли функція Bellman'а має структуру $\mathcal{B}(t, x) = b_0(t) + b_1(t)x + b_2(t)x^2$.

4.9. Завдання для самостійного опрацювання

◆ *Завдання 4.9.1.* Сформулювати принцип максимуму для задачі оптимального керування із закріпленими кінцями і фіксованим часом у випадку двох фазових змінних і однієї керуючої функції.

◆ *Завдання 4.9.2.* Сформулювати принцип максимуму для задачі оптимальної швидкодії для лінійної та нелінійної систем стану та керування.

◆ *Завдання 4.9.3.* Сформулювати принцип максимуму для задачі

$$I[u] = \int_0^T (x^2(t) - u^2(t)) dt \rightarrow \max,$$

$$\frac{dx}{dt} = u(t), \quad 0 \leq t \leq T,$$

$$x(0) = x(T) = 0, \quad T - \text{фіксоване.}$$

◆ *Завдання 4.9.4.* Застосувати принцип максимуму для лінійної задачі:

$$\begin{cases} \frac{dx}{dt} = -x(t) + u(t), & t \geq 0, \\ x(0) = x_0 > 0, \end{cases}$$

T та $x(T)$ не фіксовані,

$$I[u] = \int_0^T (x^2(t) + u^2(t)) dt \rightarrow \min.$$

◆ *Завдання 4.9.5.* Сформулювати у формі оптимального керування задачу

$$I[x] = \int_{-\pi}^{\pi} x(t) \sin t dt \rightarrow \min,$$

$$\begin{cases} \left| \frac{dx}{dt} \right| \leq 1, \\ x(-\pi) = x(\pi) = 0, \end{cases}$$

та застосувати до неї принцип максимуму.

◆ *Завдання 4.9.6.* Розв'язати задачу мінімізації кінцевого моменту часу T (швидкодії) за умов:

$$\begin{cases} \left| \frac{d^2x}{dt^2} \right| \leq 2, \\ x(-1) = 1, \quad x(T) = -1, \\ x'(-1) = x'(T) = 0. \end{cases}$$

◆ *Завдання 4.9.7.* Максимізувати інтегральний функціонал

$$I[x] = \int_0^2 x(t) dt,$$

за умов

$$\begin{cases} \left| \frac{d^2x}{dt^2} \right| \leq 2, \\ x(0) = x'(0) = x(2) = 0. \end{cases}$$

◆ *Завдання 4.9.8.* Використовуючи принцип максимуму, визначити $u(t)$, при якому чистий прибуток

$$\Pi[u] = \int_0^T e^{\alpha t} [1 - u(t) - \beta] x(t) dt \rightarrow \max,$$

при цьому

$$\begin{cases} \frac{dx}{dt} = \alpha u(t)x(t), \\ 0 \leq u(t) \leq (1 - \beta), \\ \alpha > 0, \quad \beta > 0. \end{cases}$$

◆ *Завдання 4.9.9.* Звести задачу до задачі оптимального керування та застосувати для розв'язування принцип максимуму

$$\int_0^T \left| \frac{dx(t)}{dt} \right| dt \rightarrow \min,$$

$$\begin{cases} \frac{dx(t)}{dt} \geq A, & A < 0, \\ x(0) = 0, & x(T) = b. \end{cases}$$

◆ *Завдання 4.9.10.* Звести задачу оптимального керування

$$T = \int_0^T 1 dt \rightarrow \min,$$

$$\begin{cases} \frac{dx_1(t)}{dt} = x_2(t), \\ \frac{dx_2(t)}{dt} = u(t), \\ x_1(0) = x_0, & x_2(0) = \delta_0, \\ x_1(t) = x_2(T) = 0, \\ u \in [-1, 1]. \end{cases}$$

до задачі дослідження функціоналу на оптимальну швидкість.

◆ *Завдання 4.9.11.* За допомогою лінійно-квадратичної функції Bellman'a мінімізувати функціонал

$$I[u] = \int_0^1 (x^2(t) + u^2(t)) dt$$

з умовами

$$\begin{cases} \frac{dx}{dt} = x + u, \\ x(0) = 1. \end{cases}$$

◆ *Завдання 4.9.12.* Знайти лінійно-квадратичну функцію Беллмана для оптимального процесу

$$I[u] = \int_{t_0}^{t_1} (x(t) - u(t)) dt \rightarrow \max,$$
$$\begin{cases} \dot{x}(t) = \sqrt{u(t)}, \\ x(t_0) = x_0, \\ x(t_1) = x_1, \end{cases}$$
$$0 \leq u \leq x.$$

Розділ 5

Оптимальне керування соціально-економічними моделями

У цьому розділі розглянуто математичні моделі соціально-економічних процесів, розв'язування яких зводиться до дослідження задач оптимального керування із використання принципу максимуму.

5.1. Формулювання математичної моделі оптимального використання енергії із врахуванням якості навколишнього середовища

Очевидно, що процес виробництва енергії впливає на якість навколишнього середовища. Тому виникає задача визначення такого режиму вироблення та використання енергії, який веде до найвигідніших соціально-економічних наслідків.

Нехай $E = E(t)$ – кількість енергії того чи іншого виду, а $V = V(t)$ – темп використання цієї енергії у момент часу $t \in [0, T]$. Тоді між цими

величинами повинен бути очевидний взаємозв'язок

$$\frac{dE(t)}{dt} = -V(t). \quad (5.1.1)$$

Використання енергії, з одного боку, веде до позитивного фактору $C = C(V)$, що виражається у збільшенні матеріальних благ, а з іншого – до негативного фактору $P = P(V)$, як, наприклад, забруднення навколишнього середовища. Зрозуміло, що збільшення використання енергії приводить до росту обох факторів, хоча темп швидкості їх росту має протилежні значення. Математично це відображається нерівностями:

$$C'(V) > 0, \quad C''(V) < 0, \quad P'(V) > 0, \quad P''(V) > 0. \quad (5.1.2)$$

Уведемо функцію корисності $U = U(C, P)$, яка визначає корисність використання енергії, залежно від величини C і P . На цьому етапі нас не цікавить аналітичний вигляд функції корисності, але вважатимемо, що для всіх $t \in [0, T]$ виконуються умови:

$$\frac{\partial U}{\partial C} > 0, \quad \frac{\partial U}{\partial P} < 0, \quad \frac{\partial^2 U}{\partial C^2} < 0, \quad \frac{\partial^2 U}{\partial P^2} < 0, \quad \frac{\partial^2 U}{\partial C \partial P} = 0. \quad (5.1.3)$$

Тут перші дві нерівності природньо означають, що функція корисності є зростаючою за C і спадною за P , а наступні співвідношення (5.1.3) характеризують, що темпи зміни корисності з ростом відповідних змінних уповільнюються.

У цій моделі на роль змінних керування та стану претендують як величина енергії E , так і темп її використання V . Виходячи із економічних міркувань будемо вважати V змінною керування, а E – змінною стану (фазова змінна).

Критерієм оптимальності буде інтеграл від функції корисності, а саме

$$I[V] = \int_0^T U(C(V(t)), P(V(t))) dt \rightarrow \max, \quad (5.1.4)$$

за умов

$$\begin{cases} \frac{dE(t)}{dt} = -V(t), \\ E(0) = E_0, \quad E(T) \geq 0, \\ V(t) > 0, \quad t \in [0, T], \end{cases} \quad (5.1.5)$$

при заданих сталих E_0 і $E(T)$.

Застосуємо принцип максимуму до розв'язування задачі (5.1.4)-(5.1.5) та аналізу одержаних результатів.

Функція Hamilton'a у цьому випадку має вигляд (обмежимося випадком $\psi_0^* = 1$)

$$H(V, \psi) = U(C(V), P(V)) - \psi V.$$

Тоді, вважаючи функцію H диференційовною за змінною V , запишемо необхідну умову існування екстремуму

$$\frac{\partial H}{\partial V} = \frac{\partial U}{\partial C} C'(V) + \frac{\partial U}{\partial P} P'(V) - \psi = 0. \quad (5.1.6)$$

Або, врахувавши (5.1.2)-(5.1.3), одержимо співвідношення

$$\frac{\partial^2 H}{\partial V^2} = \frac{\partial^2 U}{\partial C^2} (C')^2 + \frac{\partial U}{\partial C} C'' + \frac{\partial^2 U}{\partial P^2} (P')^2 + \frac{\partial U}{\partial P} P'' < 0,$$

яке свідчить про те, що розв'язок рівняння (5.1.6) буде екстремалією функції Hamilton'a за змінною V .

Спряжене рівняння у цьому випадку виглядає так

$$\frac{d\psi}{dt} = -\frac{\partial H}{\partial E} = 0,$$

звідки і знаходимо $\psi^*(t) = c = \text{const} \quad \forall t \in [0, T]$. Для визначення сталої c використаємо умову трансверсальності на правому кінці проміжку, тобто $\psi^*(T) \geq 0$ і $\psi^*(T)E(T) = 0$. На підставі того, що $\psi^*(t) \equiv c$, випливає $\psi^*(t) = c \geq 0 \quad \forall t \in [0, T]$.

Тому рівняння (5.1.6) можна переписати так

$$\frac{\partial U}{\partial C} C'(V) + \frac{\partial U}{\partial P} P'(V) = c.$$

Оскільки в це рівняння змінна t явно не входить, а його розв'язок $V(t) = V^*$, де V^* – деяка додатна стала для всіх $t \in [0, T]$.

Звідси можна зробити висновок, що у цій моделі оптимальний режим характеризується сталим темпом використання енергії.

При $V(t) \equiv V^*$ задача Cauchy для диференціального рівняння (5.1.5) має оптимальний розв'язок $E^*(t) = E_0 - V^*t$, що визначає кількість енергії у момент часу t .

Оскільки у цій задачі функції $C(V)$, $P(V)$ та $U(C, P)$ конкретно не задані, то неможливо визначити оптимального значення V^* . Але при $t = T$, у випадку, коли $E^*(T) = 0$ (повне використання енергії), величина оптимального значення темпу використання енергії має вигляд $V^* = \frac{E_0}{T}$.

5.2. Однофакторна модель оптимального економічного росту

Повернемося до [підрозділу 2.5.](#), в якому описана *модель Solow*, і розглянемо поведінку *однофакторної економічної системи* на основі цієї моделі. Тобто

$$\begin{aligned} \frac{dk}{dt} &= -\lambda k + \rho f(k), \quad \lambda = \mu + \nu, \\ k(0) &= k_0 = \frac{K_0}{L_0}, \\ x &= f(k), \quad i = \rho f(k), \quad c = (1 - \rho)f(k), \end{aligned} \tag{5.2.1}$$

де:

$k = \frac{K}{L}$ – фондозабезпеченість;

K – основні фонди;

L – праця (кількість зайнятих);

$x = \frac{X}{L}$ – продуктивність праці;

X – валовий внутрішній продукт;

$i = \frac{I}{L}$ – питомі інвестиції на одного працюючого;

$c = \frac{C}{L}$ – питома споживання;

C – фонд невиробничого споживання;

ν – річний темп приросту працюючих;

ρ – норма накопичення;

μ – доля зношеності за рік основних виробничих фондів.

Ця модель була розглянута у [підрозділі 2.5](#). за умови сталості норми накопичення ($\rho = \text{const}$). Тепер будемо вважати, що $\rho \neq \text{const}$ і сформулюємо задачу оптимального ведення бізнесу, використовуючи принцип максимуму.

Отож, із останнього рівняння ([5.2.1](#)) виразимо $\rho f(k) = f(k) - c$, а перше перепишемо так

$$\begin{cases} \frac{dk}{dt} = f(k) - (\mu - \nu)k - c, \\ k(0) = k_0. \end{cases}$$

Тоді за керуючу функцію виберемо питома споживання $c = c(t)$,

$$0 < c_0 \leq c(t) \leq f(k(t)), \quad t \in [0, T], \quad (5.2.2)$$

де c_0 – нижня гранично допустима норма питомого споживання.

Потрібно так керувати податково-кредитною політикою, змінюючи $c(t)$, щоб упродовж довгого періоду часу дисконтна корисність від споживання була максимальною, тобто

$$I[c] = \int_0^{+\infty} e^{-\delta t} u(c(t)) dt \rightarrow \max. \quad (5.2.3)$$

Тут δ – додатний параметр дисконтування, за допомогою якого майбутні можливі корисності зводилися до теперішнього часу, виходячи з того, що ближче в часі споживання важливіше від віддаленого майбутнього; $u = u(c)$ – функція корисності споживання. Від функції u , зазвичай, вимагають, щоб вона була додатною, строго зростаючою за умови $u(0) = 0$. Для спрощення викладу, вважатимемо що корисність прямо пропорційна питомому споживанню, тобто

$$u(c) = \alpha c, \quad \alpha > 0, \quad \alpha - \text{const}.$$

Отже, задача про максимальний ріст однопараметричної замкнутої економічної системи з нескінченним обрієм керування і додатною нормою дисконтування полягає у максимізації функціоналу

$$I[c] = \alpha \int_0^{+\infty} e^{-\delta t} c(t) dt \rightarrow \max, \quad (5.2.4)$$

за умов

$$\begin{aligned} \frac{dk}{dt} &= f(k) - \lambda k - c(t), \quad \lambda = \mu + \nu, \\ k(0) &= k_0, \\ 0 < c_0 &\leq c(t) \leq f(k(t)), \quad k(t) \geq 0, \quad t \in [0, T]. \end{aligned} \quad (5.2.5)$$

Інтеграл (5.2.4) називають *інтегралом благополуччя*, а від функції $k = k(t)$ потрібно також вимагати умови $\lim_{t \rightarrow +\infty} k(t) \geq 0$.

Застосовуючи принцип максимуму, замість спряженої змінної $\psi_1(t)$ введемо нову $\psi(t) = e^{\delta t} \psi_1(t)$, вважаючи, що $\psi_0^* = 1$, *hamiltonian (гамільтоніан)*

$$H = e^{-\delta t} [\alpha c + \psi(f(t) - \lambda k - c)]. \quad (5.2.6)$$

Виходячи із (5.2.6), спряжену змінну ψ можна трактувати як *тинову ціну приросту фондів*. Вираз у квадратних дужках (5.2.6) характеризує *корисність питомого випуску*, оскільки αc – корисність, яка

йде на невиробниче споживання, а $\psi(f(t) - \lambda k - c)$ – корисність тієї її частини, яка йде на розширення фондів. Дисконтний множник регулює приведення питомої корисності до моменту реальної дійсності її використання.

Випишемо спряжену систему

$$\frac{d}{dt}(e^{-\delta t}\psi(t)) = -\frac{\partial H}{\partial k},$$

яка разом із (5.2.5) зводиться до системи

$$\begin{cases} \frac{d\psi}{dt} = [(\lambda + \delta) - f'(k)]\psi, \\ \frac{dk}{dt} = f(k) - \lambda(k - c), \\ k(0) = k_0. \end{cases} \quad (5.2.7)$$

Оскільки похідні $f'(k) > 0$, $f''(k) < 0$, то f строго зростає, її похідна строго спадає для всіх невід'ємних k , а графік починається у початку координат, $f'(0) = 0$. Крім того вважаємо, що $\lim_{k \rightarrow +0} f'(k) = +\infty$ і

$$\lim_{k \rightarrow +\infty} f'(k) = 0.$$

Із системи (5.2.7) маємо існування стаціонарного розв'язку $k(t) = k^* = \text{const}$, $c(t) = c^* = \text{const}$, який знаходимо, прирівнюючи до нуля праві частини цих рівнянь

$$\begin{cases} f'(k^*) = \lambda + \delta, \\ c^* = f(k^*) - \lambda k^*. \end{cases} \quad (5.2.8)$$

Перейдемо до існування та єдиності сталих c^* і k^* з умовами (5.2.8). Враховуючи умови на функцію f , бачимо, що при зміні k від 0 до $+\infty$, f' зменшується від $+\infty$ до 0, тобто, знайдеться єдине додатне число, $k = k^*$, при якому ця похідна точно прийме значення $\lambda + \delta$, причому

$$\begin{aligned} (\lambda + \delta) - f'(k) &> 0 \quad \text{при } k < k^*, \\ (\lambda + \delta) - f'(k) &< 0 \quad \text{при } k > k^*. \end{aligned} \quad (5.2.9)$$

Отже, перша рівність у (5.2.8), для заданого фіксованого $\delta > 0$, виконується. Тому після цього число c^* , для визначення k^* , можна однозначно знайти з другої рівності (5.2.8). Причому, на підставі нерівності $f'(k^*) = \lambda + \delta > \lambda = (\lambda k)'|_{k=k^*}$, між кутовими коефіцієнтами дотичної до графіка функції $y = f(k)$ у точці $k = k^*$ і прямою $y = \lambda k$ буде виконуватися $f(k^*) > \lambda k^*$, звідки $0 < c^* < f(k^*)$.

Значення сталих c^* і k^* відповідає *стаціонарному режиму траєкторії* (траєкторія збалансованого росту), що з економічної точки зору відповідає ситуації, коли здійснюється виробництво, яке дозволяє при сталому питомому споживанні підтримувати фондозабезпеченість на стаціонарному рівні $k = k^*$.

Перепишемо hamiltonian (5.2.4) у формі

$$H[c] = e^{-\delta t} \left\{ (\alpha - \psi)c + \psi [f(k) - \lambda k] \right\}.$$

Оскільки він лінійно залежить від c , то його максимум за c визначається знаком виразу $(\alpha - \psi)$ і, на підставі нерівності (5.2.2), досягається при зміні закону питомого споживання

$$c^*(t) = \begin{cases} c^*, & \psi^*(t) > \alpha, \\ f(k(t), t), & \psi^*(t) < \alpha, \end{cases}$$

а умова трансверсальності на правому кінці є такою

$$\lim_{t \rightarrow +\infty} \psi(t)k^*(t) = 0.$$

Як було зазначено вище, найбільший економічний інтерес є до траєкторії збалансованого росту, що відповідає $k(t) \equiv k^*$, $c(t) \equiv c^*$. Якщо у початковий момент часу $k_0 = k(0) = k^*$, то із керування $c^*(t) \equiv c^*$, одержимо для всіх $t \geq 0$ *стаціонарний режим*, що відповідає *траєкторії збалансованого росту*. Якщо ж $k_0 \neq k^*$, то стаціонарний режим еволюційного процесу керування можна досягнути лише через деякий час, або ніколи не вийти нього. Детальний аналіз інших режимів траєкторій потребує додаткових математичних досліджень.

5.3. Вплив на поведінку економічної системи податкових відрахувань

Нехай маємо фірму, яка випускає однорідну продукцію. Виходимо із виробничої функції

$$v = F(\Phi, L). \quad (5.3.1)$$

Тут v – випуск за одиницю часу; Φ – кількість основних фондів; L – кількість трудових ресурсів.

Будемо розглядати відносно невеликий проміжок часу, коли можна вважати, що об'єм основних фондів незмінний. Затрати на виробництво складаються із придбання оборотних фондів і виплати заробітної плати

$$Z(t) = Z(t)L(t) + Ц(t)f(t), \quad (5.3.2)$$

де Z – середня зарплата одного робітника; f – темп використання оборотних фондів для виробництва; $Ц$ – ціна одиниці оборотних фондів. Вважатимемо, що чисельність персоналу L пропорційна темпові використання оборотних фондів

$$L(t) = af(t), \quad a = \text{const}.$$

Тоді із (5.3.2) маємо

$$L(t) = b(t)Z(t),$$

де $b(t) = (Z(t) + Ц(t)a^{-1})^{-1}$. Підставивши це співвідношення у (5.3.1) одержимо

$$v(t) = \mathcal{F}(t, Z(t)), \quad (5.3.3)$$

де

$$\mathcal{F} = F(\Phi, b(t)Z). \quad (5.3.4)$$

Зазначимо, що об'єм основних фондів Φ є сталим. Співвідношення (5.3.3)-(5.3.4) дає залежність випуску від затрат.

Розглянемо тепер модель оподаткування на прибуток. Прибуток, з якого знімають податок, має вигляд

$$\mathcal{D}(t, Z) = \mathcal{F}(t, Z) - Z.$$

Якщо врахувати податкову ставку $\nu(t)$, то

$$\mathcal{N}(t) = \nu(t)D(t, Z) = \nu(t) \left(\mathcal{F}(t, Z(t)) - Z(t) \right) \quad (5.3.5)$$

– величина податку. Тоді

$$\Pi(t) = \left(1 - \nu(t) \right) D(t, Z) = (1 - \nu(t)) \left(\mathcal{F}(t, Z(t)) - Z(t) \right)$$

є прибуток, що залишається підприємству. Частина цього прибутку йде на виплату інших податків, які ми тут не враховуємо. Нехай $\mu(t)\Pi(t)$, $\mu \in (0, 1)$ – частина прибутку, яка піде на затрати виробництва у наступному році ($t + 1$), тоді

$$\begin{aligned} Z(t+1) &= \mu(t)(1 - \nu(t))(\mathcal{F}(t, Z(t)) - Z(t)), \\ Z &= Z_0, \quad t = 0, 1, \dots, \end{aligned} \quad (5.3.6)$$

де Z_0 – затрати у початковий момент. У (5.3.6) функція $Z(t)$ є фазовою координатою; $\nu(t)$ – керування, яке підпорядковане обмеженню

$$\nu_m \leq \nu(t) \leq \nu_s, \quad (5.3.7)$$

де $0 \leq \nu_m < \nu_s \leq 1$ – задані числа.

За критерій оптимальності візьмемо функцію

$$I = \sum_{t=0}^T e^{-\delta t} \mathcal{N}(t), \quad (5.3.8)$$

як дисконтно зважену суму податкових відрахувань з дисконтним коефіцієнтом $\delta > 0$.

Задача оптимального керування полягає у знаходженні такої функції $\nu(t)$ (*стратегія оподаткування*), з обмеженнями (5.3.7), щоб функціонал (5.3.8), в якому $Z(t)$ є розв'язком дискретного ланцюга (5.3.6), досягав свого максимального значення (забезпечував максимум дисконтно зваженої суми податкових надходжень у бюджет).

Розглянемо тепер неперервний випадок цієї моделі оподаткування і відповідної задачі оптимального керування. Перепишемо для цього рівняння (5.3.6) у вигляді

$$Z(t+1) - Z(t) = \mu(t)(1 - \nu(t)) \left(\mathcal{F}(t, Z(t)) - Z(t) \right) - Z(t).$$

Ліва частина цього співвідношення є приростом затрат (додатним чи від'ємним) протягом року, тобто приріст затрат за відрізок Δt буде

$$Z(t+1) - Z(t) = k \left[\mu(t)(1 - \nu(t)) \left(\mathcal{F}(t, Z(t)) - Z(t) \right) - Z(t) \right] \Delta t,$$

де k – коефіцієнт розмірності, тобто $[k] = \frac{1}{[t]}$. Розділивши дане співвідношення на Δt і перейшовши до границі при $\Delta t \rightarrow 0$, одержимо

$$\begin{cases} \frac{dZ}{dt} = k\mu(t)(1 - \nu(t)) \left(\mathcal{F}(t, Z(t)) - Z(t) \right) - kZ(t), \\ Z(0) = Z_0. \end{cases} \quad (5.3.9)$$

Неперервний аналог критерію (5.3.8) має вигляд

$$I[\nu] = \int_0^T e^{-\delta t} \left(\mathcal{F}(t, Z(t)) - Z(t) \right) \nu(t) dt, \quad (5.3.10)$$

і задача оптимізації податкової ставки полягає у максимізації цього функціоналу на розв'язках задачі Cauchy (5.3.9) із урахуванням обмеження (5.3.7).

Цікавим є факт падіння затрат. Якщо у задачі Cauchy (5.3.9) взяти $\nu(t) = 1$, то для деякого моменту часу $t \geq t_1$, маємо

$$\frac{dZ}{dt} = -kZ, \quad \forall t \geq t_1, \quad Z(t_1) > 0.$$

Тоді при $t \geq t_1$ відбувається падіння затрат за експоненціальним законом, що відповідно до (5.3.5) приводить і до падіння темпу податкових

надходжень, хоча $Z(t) = 0$ або $\mathcal{N}(t) = 0$ не досягається для довільних скінчених t . У мікроекономіці це явище називають *ефектом Laffer'a*, який показує, що зниження ставки оподаткування має стимулюючий вплив на виробництво [19].

Розглянемо тепер лінійний варіант вищенаведеної задачі і спробуємо застосувати до неї принцип максимуму. Отже, потрібно мінімізувати функціонал (5.3.10) на множині D , яка складається із функцій $\{Z(t), \nu(t)\}$, що задовольняють умови (5.3.9) та обмеження (5.3.7), причому вибираємо

$$\begin{aligned} \mathcal{F}(t, Z) &= \varkappa(t)Z, & \varkappa &= v_0 \frac{b(t)}{L_0} > 1, \\ b(t) &= \left(3(t) + \Pi(t)a^{-1} \right)^{-1}, & [t] &= 1, k = 1. \end{aligned}$$

Враховуючи лінійність $\mathcal{F}(t, Z)$, перепишемо цю задачу такими чином

$$I[\nu] = - \int_0^T e^{-\delta t} (\varkappa(t) - 1) Z(t) \nu(t) dt \rightarrow \min \quad (5.3.11)$$

за умов

$$\begin{cases} \frac{dZ}{dt} = [\mu(t)(\varkappa(t) - 1)(1 - \nu(t)) - 1]Z, \\ Z(0) = Z_0. \end{cases} \quad (5.3.12)$$

та обмеженнях (5.3.7).

За класифікацією задач оптимального керування задача (5.3.11)-(5.3.12), (5.3.7) належить до класу білінійних оптимальних задач.

Позначивши

$$\alpha(t) = \mu(t)(\varkappa - 1) - 1,$$

перепишемо (5.3.12) так

$$\begin{cases} \frac{dZ}{dt} = \alpha(t)Z - (\alpha(t) + 1)Z\nu(t), \\ Z(0) = Z_0. \end{cases} \quad (5.3.13)$$

З метою спрощення задачі підберемо функцію $A(t)$ у заміні $Z = A(t)y$, тобто підставивши цю заміну в (5.3.13), одержимо

$$\frac{dA}{dt}y + A\frac{dy}{dt} = \alpha Ay - (\alpha - 1)\nu AY.$$

Якщо тепер вибрати

$$\begin{cases} \frac{dA}{dt} = \alpha(t)A, \\ A(0) = Z_0, \end{cases}$$

або

$$A(t) = Z_0 e^{\int_0^t \alpha(\tau) d\tau}, \quad (5.3.14)$$

то (5.3.13) прийме вигляд

$$\begin{cases} \frac{dy}{dt} = -(\alpha(t) + 1)y\nu(t), \\ y(0) = 1. \end{cases}$$

Тоді задача (5.3.11)-(5.3.12), (5.3.7) зведеться до задачі

$$I[\nu] = -\int_0^T a(t)y(t)\nu(t)dt \rightarrow \min,$$

на множині D_2 із функцій $\{y(t), \nu(t)\}$, з умовами

$$\begin{cases} \frac{dy}{dt} = -\chi(t)y\nu(t), \\ y(0) = 1, \\ \nu(t) \in [\nu_m, \nu_s], \end{cases} \quad (5.3.15)$$

де

$$a(t) = e^{-\delta t} \left(\chi(t) - 1 \right) A(t), \quad \chi(t) = \alpha(t) + 1.$$

Зазначимо, що нас цікавить тільки оптимальне керування $\nu^*(t)$, тому повертатися до фазової змінної Z не має сенсу.

Для розв'язування задачі (5.3.15) застосуємо принцип максимуму Понтрягіна.

Функцію *Hamilton'a* у такому випадку виражаємо співвідношенням

$$H(t, y, \nu) = a(t)\nu y - \psi(t)\chi(t)\nu y = \{a(t) - \psi(t)\chi(t)\}\nu y.$$

Спряжена система виглядає так:

$$\begin{cases} \frac{d\psi}{dt} = -\{a(t) - \psi\chi(t)\}\nu(t), \\ \psi(T) = 0. \end{cases} \quad (5.3.16)$$

Оптимальне керування $\nu^*(t)$ знаходимо із умов максимуму функції H за ν . Оскільки $y = y(t) > 0$, то одержимо

$$\nu^*(t) = \begin{cases} \nu_m, & \{a(t) - \psi(t)\chi(t)\} < 0, \\ \forall \nu \in [\nu_m, \nu_s], & \{a(t) - \psi(t)\chi(t)\} = 0, \\ \nu_s, & \{a(t) - \psi(t)\chi(t)\} > 0. \end{cases} \quad (5.3.17)$$

Після цього функцію $\psi(t)$ знаходимо як розв'язок задачі Cauchy (5.3.16) при $\nu = \nu^*(t)$ із врахуванням (5.3.17). Тобто тепер задача Cauchy (5.3.16) прийме вигляд

$$\frac{d\psi}{dt} = \begin{cases} -\{a(t) - \psi(t)\chi(t)\}\nu_m, & \{a(t) - \psi(t)\chi(t)\} < 0, \\ 0, & \{a(t) - \psi(t)\chi(t)\} = 0, \\ -\{a(t) - \psi(t)\chi(t)\}\nu_s, & \{a(t) - \psi(t)\chi(t)\} > 0, \end{cases} \quad (5.3.18)$$

$$\psi(T) = 0.$$

Інтегрування задачі (5.3.18) починаємо із моменту $t = T$. Оскільки $\psi(T) = 0$ і функція $a(t)$ невід'ємна (див. її вибір), то

$$\{a(T) - \psi(T)\chi(T)\} = a(T) > 0. \quad (5.3.19)$$

Із (5.3.17) маємо, що $\nu^*(T) = \nu_s$. Оскільки розв'язок задачі Cauchy (5.3.18) є неперервною функцією, то вираз $\{a(t) - \psi(t)\chi(t)\}$ також є неперервним. Це означає, що на підставі (5.3.19), $\{a(t) - \psi(t)\chi(t)\} > 0$ при $t = T$, звідки і випливає його неперервність і на деякому напівінтервалі $t \in (t^*, T]$. Отож, існує напівінтервал $(t^*, T]$, на якому податкова ставка $\nu^*(t) = \nu_s$ приймає максимальне значення.

Із (5.3.18) видно, що при $t \in (t^*, T)$ похідна $\frac{d\psi}{dt} < 0$. На підставі (5.3.19) маємо, що на інтервалі (t^*, T) функція $\psi(t)$ – додатна і спадає при зростанні t , або тож саме, що зростає при зменшенні t від $t = T$. Це є основою для припущення, що вираз $\{a(t) - \psi(t)\chi(t)\}|_{t=t^*} = 0$ і змінює знак при $t < t^*$. Тоді відповідно до (5.3.17) точка t^* є *точкою перемикання керування*. Зазвичай, таких точок буває декілька.

Аналогічне розв'язування задачі Cauchy (5.3.16) у випадку змінних коефіцієнтів є складним, тому до їх розв'язування потрібно застосувати чисельні методи.

Обговоримо тепер економічну інтерпретацію оптимального розв'язку $\nu^*(t)$. Виходячи з того, що критерієм оптимальності нашої задачі є максимум дисконтно зваженої суми податкових відрахувань, може скластися враження, що при такому критерії відображаються інтереси податкової організації і ніяк не захищаються інтереси виробника, коли оптимальний розв'язок $\nu^*(t) = \nu_s$, тобто максимально можлива податкова ставка. Однак, це не так, оскільки максимальну податкову ставку потрібно призначати лише при $t > t^*$. Відрізок $[0, t^*]$, на якому $\nu^*(t) = \nu_m$ мінімально можливе податкової ставки природньо інтерпретувати як відрізок *"податкових канікул"*. Дійсно, на цьому відрізку фірма має можливість направляти ресурси на розвиток виробництва, що забезпечить високий прибуток на відрізку $[t^*, T]$, а значить і більші суми податкових відрахувань.

Отже, виявлення ефекту *"податкових канікул"* є основним результатом математичного аналізу розглядуваної задачі.

5.4. Завдання для самостійного опрацювання

◆ *Завдання 5.4.1.* Розглянути новий функціонал з дисконтним множником у задачі найкращого використання енергії

$$I[v] = \int_0^T e^{-\delta t} F(C(v(t)), P(v(t))) dt$$

і для цього випадку:

- а) виписати функціонал Hamilton'а і відповідний принцип максимуму;
- б) знайти оптимальне значення спряженої змінної;
- в) обґрунтувати питання щодо сталості оптимального значення енергії E^* ;
- г) знайти $\frac{dv}{dt}$ і визначити характер поведінки $v^*(t)$.

◆ *Завдання 5.4.2.* Як зміниться оптимальний розв'язок у випадку двовимірної задачі про найкраще використання енергії, якщо керуюча змінна A буде підпорядкована умові $A(t) \geq 0$?

◆ *Завдання 5.4.3.* Записати функцію Hamilton'а та сформулювати відповідний принцип максимуму для цільового функціоналу з дисконтним множником у двовимірній задачі найкращого використання енергії.

◆ *Завдання 5.4.4.* Для задачі про оптимальний економічний ріст однопараметричної економіки записати її варіант відносного споживання та побудувати для цього випадку графіки оптимальних траєкторій.

◆ *Завдання 5.4.5.* При яких умовах в задачі оптимального економічного росту однофакторної економіки значення норми накопичення приводить до довгострокового зниження споживання в розрахунку на одного працюючого?

◆ *Завдання 5.4.6.* Розглянути варіант задачі про оптимізацію податкової ставки у випадку, коли коефіцієнти $\kappa(t)$ і $\mu(t)$ є сталими.

◆ *Завдання 5.4.7.* Мінімізувати функціонал

$$I[u] = - \int_0^T e^{-\delta t} \left[\nu(t)(F(t, Z(t)) - Z(t)) + u(t) \right] dt \rightarrow \min$$

на множині D функцій $\{Z(t), B(t), \nu(t), u(t)\}$, що задовольняють умови

$$\begin{cases} \frac{dZ}{dt} = \mu(t)(1 - \nu(t))\{\mathcal{F}(t, Z(t)) - Z(t)\} - Z(t) - u(t), \\ \frac{dB}{dt} = -u(t), \\ Z(0) = Z_0, B(0) = B_0, B(T) = 0, \end{cases}$$

$$\nu(t) \in [\nu_m, \nu_s], u(t) \in [u_m, u_s].$$

Тут T – кінцевий час виплати боргу $B(t)$ перед податковим органом підприємства, а $u(t)$ – термін виконання боргу. Всі інші параметри такі ж як і у задачі про оподаткування підприємства.

◆ *Завдання 5.4.8.* Розглянути лінійний варіант задачі **завдання 5.4.7**, тобто $\mathcal{F}(t, Z) = \varkappa(t)Z$, а саме:

$$I[u] = - \int_0^T e^{-\delta t} \{ \nu(t)(\varkappa(t) - 1)Z(t) + u(t) \} dt \rightarrow \min,$$

$$\begin{cases} \frac{dZ}{dt} = \{ \mu(t)(\varkappa(t) - 1)(1 - \nu(t)) - 1 \} Z(t) - u(t), \\ \frac{dB}{dt} = -u(t), \\ Z(0) = Z_0, B(0) = B_0, B(T) = 0, \end{cases}$$

$$\nu(t) \in [\nu_m, \nu_s], u(t) \in [u_m, u_s].$$

◆ *Завдання 5.4.9.* Мінімізувати функціонал

$$I[u] = - \int_0^T e^{-\delta t} \frac{P(t)}{L(t)} dt \rightarrow \min,$$

на множині D функцій $\{\Phi(t), L(t), v(t), u(t)\}$ за умов

$$\begin{cases} \frac{d\Phi}{dt} = b^{-1}(t)(1 - \alpha(t))(1 - u(t))v(t) - \chi(t)\Phi(t), \\ \frac{dL}{dt} = -n(t)L(t), \\ \Phi(0) = \Phi_0, \Phi(T) = \Phi_T, L(0) = L_0, \end{cases}$$

$$0 \leq v(t) \leq F(t, \Phi(t), L(t)),$$

$$u_{\min} \leq u(t) \leq 1,$$

$$P(t) = (1 - \alpha(t))u(t)v(t).$$

◆ *Завдання 5.4.10.* Розглянути варіант задачі однопараметричної фірми із [завдання 5.4.9](#), якщо фірма застосовує технологію Cobb-Douglas'a

$$F(t, \Phi, L) = v_0 e^{\rho t} \Phi^\alpha(t) L^{1-\alpha}(t),$$

$$\alpha \in (0, 1), \rho > 0, v_0 = \text{const}.$$

Розділ 6

Математичні моделі біологічних спільнот і задачі керування

При виборі раціональних методів експлуатації природних ресурсів потрібно враховувати специфіку біологічного процесу, що відповідає за відновлення самого ресурсу. Тобто у деяких випадках потрібно розвивати популяцію біологічних особин, зокрема тих, які безпосередньо володіють споживчою цінністю або культивуються для одержання споживчих продуктів.

Очевидно, що при розгляді задач керування у таких випадках, завжди додатково намагаємося одержати максимальний результат з мінімальними затратами.

Саме у цьому розділі будемо розглядати математичні моделі теорії оптимального керування та експлуатації біологічних популяцій (скупність біологічних особин певного виду), використавши результати досліджень із [1, 3, 7, 2]

6.1. Оптимальне керування чисельністю і продуктивністю популяції

У цьому підрозділі на основі *принципу максимуму Понтрягіна* розглянемо задачу оптимального керування чисельністю популяцій.

Зміна чисельності популяції, яка відображає її народжуваність та смертність, зазвичай можна описати рівнянням

$$\frac{dN}{dt} = g(N, t) \cdot N = f(N, t),$$

де $N(t)$ – чисельність популяції у момент часу t . Швидкість вільного розмноження популяції $f(N, t)$ є пропорційною до чисельності $N(t)$, а питома швидкість росту $g(N, t)$ враховує залежність між народжуваністю і смертністю особин залежно від різних факторів (вплив середовища, конкуренції, обмеженість території тощо).

Уведемо керування $u(t)$ – швидкість штучного відбору особин із популяції, тоді одержимо

$$\frac{dN}{dt} = g(N, t) \cdot N - u(t). \quad (6.1.1)$$

Нехай область допустимих керувань визначається обмеженням

$$0 \leq u(t) \leq u_{\max}. \quad (6.1.2)$$

Під *продуктивністю популяції* будемо розуміти загальний дохід, одержаний від експлуатації популяції у момент часу t :

$$P[t] = \int_0^t c(t)u(t)dt, \quad (6.1.3)$$

де $c(t)$ – ціна одиниці продукції на виході при експлуатації популяції.

Задача продуктивності популяції зводиться до відшукування керування $u(t)$ із області допустимих значень (6.1.2), яка на підставі (6.1.1)

переводить популяцію із заданого початкового стану $N(0) = N_0$ у заданий кінцевий стан $N(T) = N_T$ і забезпечує екстремум функціоналу (6.1.3) у заданий кінцевий момент часу T . Якщо ввести позначення $I[t] = -P[t]$, то одержимо вимогу $I[T] \rightarrow \min$.

Надалі вважатимемо, що всі задані величини у задачі (6.1.1)-(6.1.3) є достатньо гладкими. Диференціюючи (6.1.3), одержимо еквівалентну систему

$$\begin{cases} \frac{dI}{dt} = -c(t)u, \\ \frac{dN}{dt} = g(N, t)N - u, \end{cases} \quad (6.1.4)$$

з крайовими умовами

$$\begin{cases} I(0) = 0, & I(T) \rightarrow \min, \\ N(0) = N_0, & N(T) = N_T, \end{cases} \quad (6.1.5)$$

і обмеженнями (6.1.2). Задача (6.1.4)-(6.1.5), (6.1.2) може бути розв'язана за допомогою *принципу максимуму Понтрягіна*.

Зупинимось на деяких конкретних випадках, зокрема, $c(t) = \text{const}$, $G(N, t) = g(N)$, тобто, вважаємо, що остання система є автономною. Насамперед розглянемо систему

$$\begin{cases} \frac{dI}{dt} = -cu, \\ \frac{dN}{dt} = g(N)N - u, \end{cases}$$

з умовами (6.1.2) та (6.1.5).

Запишемо *функцію Hamilton'a*

$$\begin{aligned} H &= -\psi_0 c u + \psi_1 \{g(N)N - u\} = \psi_1 g(N)N - u(\psi_0 c - \psi_1) = \\ &= \psi_1 f(N) - u(\psi_0 c - \psi_1), \end{aligned}$$

де функції $\psi_0(t)$, $\psi_1(t)$ задовольняють *спряжену систему*

$$\begin{cases} \frac{d\psi_0}{dt} = -\frac{\partial H}{\partial t} = 0, & \psi_0 = \text{const} \leq 0, \\ \frac{d\psi_1}{dt} = -\frac{\partial H}{\partial N} = -\psi_1 \frac{df}{dN}, & \psi_1(t) = A_0 e^{-\int_0^t \frac{df}{dN} d\tau}. \end{cases} \quad (6.1.6)$$

Оскільки H – нелінійна за u функція, то із умови $\max_u H$, тобто

$$c\psi_0 + \psi_1 = \varphi(t),$$

або

$$u(t) = \begin{cases} u_{\max}, & \varphi(t) < 0, \\ 0, & \varphi(t) > 0. \end{cases} \quad (6.1.7)$$

Умова $u(t) = 0$ відповідає вільному росту популяції, а при $u(t) = u_{\max}$ маємо її максимальну продуктивність.

Перевіримо у системі наявність *особливих керувань*, тобто керувань, при яких $\varphi(t) = 0$ на деякому інтервалі (t_1, t_2) . Розглянемо

$$\frac{d\varphi}{dt} = \frac{d\psi_1}{dt} = -\psi_1 \frac{df}{dN} = c\psi_0 \cdot \frac{df}{dN} = 0.$$

Оскільки при оптимальному керуванні ψ_0 і ψ_1 одночасно не можуть бути нульовими, то $\psi_0 \neq 0$. Справді, у протилежному випадку із того, що $\varphi(t) = 0$ одержимо $\psi_1 = 0$, що заперечує умовам теореми максимуму. Тому або $\frac{df}{dN} = 0$, або особливе керування відсутнє. У першому випадку у точці $N^* = \text{const}$, маємо $u^* = f(N^*)$, де N^* знаходимо із рівняння $\frac{df}{dN} \Big|_{N=N^*} = 0$. У другому випадку, коли $\frac{df}{dN} \neq 0 \quad \forall N$, оптимальним буде керування (6.1.7).

Зупинимось, зокрема, на деяких коментарях, коли розглядається *модель динаміки вікового складу і чисельності популяції*. При цьому потрібно врахувати стать популяції.

Уведемо функцію $f(x, t)$ – густини чисельності особин віку x у момент часу t , визначену віком особин від x_1 до x_2 у момент часу t ,

тобто $\int_{x_1}^{x_2} f(x, t) dx$, $\forall x_2 \geq x_1 \geq 0$. Процес старіння і смертності можна описати рівнянням нерозривності

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} = -d_1(x, t)f(x, t) - u(x, t), \quad (6.1.8)$$

де $d_1(x, t)$ – коефіцієнт природної смертності, $u(x, t)$ – інтенсивність вилучення особин із популяції ($u \geq 0$).

Для народжуваності особин маємо

$$f(0, t) = \int_A^B \alpha(x, t)f(x, t)dx, \quad (6.1.9)$$

де $f(0, t)$ – густина новонароджених у момент часу t ; $\alpha(x, t)$ – коефіцієнт народжуваності; A, B – початок та кінець репродуктивного періоду. Система (6.1.8)-(6.1.9) описує динаміку вікового складу популяції без врахування її статі, якщо ще додати початковий розподіл $f(0, t)$.

Для двостатевої популяції співвідношення (6.1.8)-(6.1.9) можна віднести до особин жіночої статі, якщо коефіцієнт народжуваності α не залежить від особин самців.

Рівняння, що описують динаміку вікової густини самців $g(x, t)$, має подібний вигляд

$$\frac{\partial g}{\partial t} + \frac{\partial g}{\partial x} = -d_2(x, t)g(x, t) - v(x, t), \quad (6.1.10)$$

$$g(0, t) = \int_A^B \beta(x, t)f(x, t)dt, \quad (6.1.11)$$

де $d_2(x, t)$ – коефіцієнт природної смертності самців; $g(0, t)$ – густина новонароджених; $\beta(x, t)$ – коефіцієнт народжуваності; $v(x, t)$ – коефіцієнт вилучення самців ($v \geq 0$).

Задачу оптимізації у цьому випадку сформулюємо так. Потрібно

знайти вікову структуру, яка забезпечує заданий характер її відновлення і задовольняє деякі додаткові умови, що зумовлюють оптимальність за вибраним критерієм.

Говорячи про додаткові умови, маємо на увазі обмеження, що не відносяться безпосередньо до відновлення популяції, а, наприклад, обмеження на загальну чисельність популяції, на затрати на утримання популяції, обмеження у кормах тощо.

Ці умови для загального ресурсу можна представити як

$$W = \int_0^T dt \int_0^D \left(\rho_1(t)f + \rho_2(t)g \right) dx \leq W^*,$$

де ρ_1 , ρ_2 – інтенсивність використання ресурсу особами обох статей у момент часу t ; W^* – загальний запас ресурсу у період часу T ; D – граничний вік особин.

Розв'язування подібних аналогів задач продемонстровано нижче.

6.2. Задача про максимальний урожай

Використання різноманітних законів збереження (маси, енергії тощо) є поширеним прийомом при побудові математичних моделей біологічних спільнот. Зазвичай, закони зміни параметрів біологічних об'єктів (розміри особин, їх біомаса, вік, рухливість тощо) описуються динамічними системами.

Припустимо, досліджувана спільнота складається із n видів і описана системою диференціальних рівнянь

$$\frac{dx_i}{dt} = f_i(x, \alpha), \quad i = \overline{1, n}. \quad (6.2.1)$$

Тут $x = (x_1, \dots, x_n)$, $\alpha = (\alpha_1, \dots, \alpha_m)$, α – деякі параметри моделі.

Для цієї біоспільноти розглянемо збір урожаю шляхом відбору частини біомаси різних видів та виведення її із репродуктивного циклу.

Процес відбору біомаси вважаємо дискретним за часом із рівними часовими інтервалами. Нехай маємо ціну одиниці біомаси кожного виду $c = (c_1, \dots, c_n)$. Тоді можна сформулювати задачу про оптимальне керування цією системою, тобто про визначення величини вилученої біомаси кожного виду з метою, щоб сумарна вартість всієї зібраної за скінчений час T біомаси була максимальною. У кінцевий момент часу процес завершується через повний відбір біомаси (ця вимога не є принциповою, можна ставити інші обмеження). Припустимо, що інтервали між зборами біомаси є малими, тобто зміну чисельності на цьому інтервалі можна вважати лінійною за часом:

$$\begin{aligned} x_i^+(t_{k+1}) &= x_i^-(t_k) + f_i(x_j^i(t_k), \alpha_s) \cdot h, \\ t_k &= k \cdot h, \quad i, j = \overline{1, n}, \quad s = \overline{1, m}, \quad k = \overline{0, p}, \quad T = p \cdot h. \end{aligned} \quad (6.2.2)$$

Індексом "+" позначено стан системи у момент збору урожаю зліва від t_k , індексом "-" – справа від t_k . Вартість збору урожаю на кроці процесу дорівнює

$$g(t_k) = \sum_{i=1}^n c_i q_i(t_k). \quad (6.2.3)$$

Тут $q_i(t_k)$ – допустиме керування (кількість збору у момент t_k біомаси i -го виду). Природньо, що q_i повинні задовольняти обмеженням

$$0 \leq q_i(t_k) \leq x_i^+(t_k), \quad i = \overline{1, n}, \quad k = \overline{0, p}. \quad (6.2.4)$$

Крім того, очевидно, що

$$x_i(t_k) \geq 0. \quad (6.2.5)$$

Отже, задача зводиться до відшукування функцій $q_i(t_k)$, які задовольняють обмеження (6.2.4)-(6.2.5) і надають максимуму функціоналу

$$G[g] = \sum_{k=0}^p g(t_k) = \sum_{k=0}^p \sum_{i=1}^n c_i q_i(t_k). \quad (6.2.6)$$

Для розв'язування цієї задачі скористаємося методом динамічного програмування Bellman'a [10]. Уведемо функцію доходу $F_i(x_i)$, що дорівнює доходу, одержаного за l кроків збору урожаю, при умові, що на

Остаточно

$$\begin{cases} q_i^* = x_i^+(t_k) - \tilde{x}_i^+(t_k), & \text{при } x_i^+ \geq \tilde{x}_i^+, \\ q_i^* = 0, & \text{при } x_i^+ < \tilde{x}_i^+, \quad q_i^* = \{q_i(t_k)\}_{\text{opt}}. \end{cases} \quad (6.2.9)$$

Тут \tilde{x}_i^+ – розв’язок системи

$$\sum_{j=1}^n c_i \frac{\partial f_i(x_j, \alpha_s)}{\partial x_j} = 0, \quad i = \overline{1, n}.$$

Далі продовжуємо за індукцією. Зазначимо, що коли на якомусь кроці процесу керування ненульове, то воно буде ненульовим і на всіх наступних кроках.

За припущення, що керування на всій траєкторії ненульове (цього завжди можна добитися вибором початкових умов), то (6.2.8) є системою рівнянь для знаходження оптимальних значень відповідних керувань.

Сумарну вартість всієї спільноти можна записати

$$M = \sum_{i=1}^n c_i x_i. \quad (6.2.10)$$

Диференціюючи це співвідношення за t , одержимо рівняння, яке описує зміну вартості неексплуатованої спільноти у часі

$$\frac{dM}{dt} = \sum_{i=1}^n c_i \frac{dx_i}{dt} = \sum_{i=1}^n c_i f_i(x, \alpha). \quad (6.2.11)$$

Точку максимуму для (6.2.11) позначаємо через $\{\tilde{x}_i, i = \overline{1, n}\}$ і називаємо *оптимальним станом біологічної спільноти*. У цій точці

$$\sum_{i=1}^n c_i \frac{\partial f_i(x_j, \alpha_s)}{\partial x_j} = 0, \quad j = \overline{1, n}, \quad s = \overline{1, m}. \quad (6.2.12)$$

Система (6.2.12) при заміні x_j на ξ_j співпадає із системою (6.2.8). Отже,

$$\begin{aligned}\tilde{x}_i &= \xi_i = x_i - q_i^*, \quad i = \overline{1, n}, \\ q_i^* &= (q_i)_{\text{opt.}} = x_i(t) - \tilde{x}_i.\end{aligned}$$

Це означає, що оптимальним буде стан, при якому ріст вартості спільноти (приріст) визначеним чином узагальненої біомаси є максимальний. За урожай тут розуміємо всю біомасу, як приріст над рівнем, що визначений тим станом. Якщо біомаса деякого виду нижче цього рівня, то її збір заборонений до моменту досягнення відповідного рівня.

Отже, для одержання максимального доходу від збору урожаю біоспільнота повинна бути приведена до стану, коли швидкість приросту вартості спільноти $\frac{dM}{dt}$ є максимальною і зберігається у цьому стані за рахунок збору урожаю до кінця процесу.

Як випадок застосування цієї теорії розглянемо задачу про оптимальний збір урожаю у популяції, динаміка якої описана однією змінною – загальною чисельністю.

Нехай для заданої функції $f(x)$,

$$\frac{dx}{dt} = f(x), \quad t \in [0, T]. \quad (6.2.13)$$

Тоді оптимальним станом популяції означає стан, коли $\frac{df}{dx} = 0$ і $f(x)$ досягає максимуму. Оптимальним керуванням буде збір усієї біомаси, що перевищує цей рівень. Нехай

$$f(x) = \alpha x - \beta x^2, \quad (6.2.14)$$

що вважають *логістичним законом росту популяції*. Звідси зразу одержимо, що *оптимальним станом популяції* буде точка $\tilde{x} = \alpha/2\beta$, а *оптимальним керуванням*

$$\begin{cases} q^*(t) = x(t) - \frac{\alpha}{2\beta}, & x \geq \frac{\alpha}{2\beta}, \\ q^*(t) = 0, & x < \frac{\alpha}{2\beta}. \end{cases} \quad (6.2.15)$$

Сумарний урожай впродовж неперервного процесу збору урожаю за час T має вигляд

$$G(T) = x(t_0) + \frac{\alpha^2 T}{4\beta}.$$

Повторивши проведені вище міркування кінцево одержимо *оптимальне керування*

$$\begin{cases} q^*(t_k) = x^+(t_k) - \frac{\alpha}{\beta} \left(1 + e^{\frac{\alpha h}{2}}\right), & \text{при } x^*(t_k) \geq \frac{\alpha}{\beta} \left(1 + e^{\frac{\alpha h}{2}}\right), \\ q^*(t_k) = 0, & \text{при } x^*(t_k) < \frac{\alpha}{\beta} \left(1 + e^{\frac{\alpha h}{2}}\right). \end{cases}$$

Функція доходу

$$F_p(x^+) = \frac{\alpha x^+(t_0) e^{\alpha \Delta T}}{\alpha + \beta x^+(t_0) (e^{\alpha \Delta T} - 1)} + \frac{\alpha(P - I - 1)}{\beta} \cdot \frac{e^{\frac{\alpha h}{2}} - 1}{e^{\frac{\alpha h}{2}} + 1},$$

де $I = E(\Delta T/h) + 1$, $E(z)$ – ціла частина числа z , ΔT – корені рівняння

$$\frac{\alpha}{\beta \left(e^{\frac{\alpha h}{2}} + 1\right)} = \frac{\alpha x^+(t_0) e^{\alpha \Delta T}}{\alpha + \beta x^+(t_0) (e^{\alpha \Delta T} - 1)}.$$

Зазначимо, що $q^* \equiv 0$ тільки на перших I кроках процесу, там де не виконується умова $x^* \geq \frac{\alpha}{\beta} \left(e^{\frac{\alpha h}{2}} + 1\right)$. Отже, при певних умовах оптимальною політикою є повна заборона збору урожаю. Із вигляду $F_p(x^+)$ сумарна величина зібраного урожаю залежить від величини кроку процесу і максимум досягається при $h \rightarrow 0$ (неперервний збір).

6.3. Динаміка і керування віковою структурою популяції

Вивчення динаміки в часі чисельності природних популяцій мають практичне значення у багатьох прикладних задачах керування розвитку природних і штучних біопопуляцій. Найбільше поширеною математичною концепцією є принцип "хижак-жертва" Lotka-Volterra [7], яка

піддавалася удосконаленню багатьма вченими. Не вдаючись у деталі, зазначимо, що основні зауваження до цієї моделі полягають у тому, що потрібно враховувати великі проміжки часу, внутрішньопопуляційні фактори, нерівномірний розподіл особин у популяції за віком тощо.

Перейдемо до побудови математичної моделі *росту популяції із врахуванням її вікового складу*. Функцію двох змінних $u(x, t)$ будемо називати *віковою густиною чисельності особин* у момент часу t , якщо для будь-якого віку a і b чисельність особин з віком на проміжку $[a, b]$ виражається через $\int_a^b u(x, t) dx$. Очевидно, що функцію

$$\tilde{u}(x, t) = \frac{u(x, t)}{\int_0^{\infty} u(x, t) dx}$$

можна трактувати як густину ймовірності, що навмання вибрана особина має вік, не більший ніж x у момент часу t .

Розглянемо випадок двостатевої популяції з врахуванням розділеної динаміки статей, а саме

$$\left\{ \begin{array}{l} \frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = -d_u(x, t)u, \\ \frac{\partial v}{\partial t} - \frac{\partial v}{\partial x} = -d_v(x, t)v, \\ u(0, t) = \int_0^{\infty} b_u(x, t)u dx, \\ v(0, t) = \int_0^{\infty} b_v(x, t)u dx, \\ u(x, 0) = g_u(x), \\ v(x, 0) = g_v(x). \end{array} \right. \quad (6.3.1)$$

Тут $u(x, t)$ – густина чисельності самок; $v(x, t)$ – густина чисельності самців; d_u, d_v, b_u, b_v – відповідно коефіцієнти смертності і народжуваності самок і самців.

Уведемо позначення:

$$\begin{aligned}
 D_u(x, t) &\stackrel{\text{def}}{=} \int_0^x d_u(\xi, t - x + \xi) d\xi, \\
 D_v(x, t) &\stackrel{\text{def}}{=} \int_0^x d_v(\xi, t - x + \xi) d\xi, \\
 K_u(x, t) &\stackrel{\text{def}}{=} b_u(x, t) e^{-D_u(x, t)}, \\
 K_v(x, t) &\stackrel{\text{def}}{=} b_v(x, t) e^{-D_v(x, t)}, \\
 \varphi_u(x, t) &\stackrel{\text{def}}{=} g_u(x) e^{D_u(x, 0)}, \\
 \varphi_v(x, t) &\stackrel{\text{def}}{=} g_v(x) e^{D_v(x, 0)},
 \end{aligned} \tag{6.3.2}$$

Відповідно до введених позначень, використовуюючи метод характеристик [3], одержимо

$$\begin{cases} u(x, t) = \Omega_u(t - x) e^{-D_u(x, t)}, \\ v(x, t) = \Omega_v(t - x) e^{-D_v(x, t)}, \end{cases} \tag{6.3.3}$$

де Ω_u і Ω_v визначаємо із рівнянь

$$\Omega_u(t) = \int_0^\infty K_u(x, t) \Omega_u(t - x) dx, \tag{6.3.4}$$

$$\Omega_v(t) = \int_0^\infty K_v(x, t) \Omega_v(t - x) dx, \tag{6.3.5}$$

$$\Omega_u(-x) = \varphi_u(x), \tag{6.3.6}$$

$$\Omega_v(-x) = \varphi_v(x). \tag{6.3.7}$$

Інтегральні рівняння Volterra другого роду (6.3.4)-(6.3.5) із умовами (6.3.6)-(6.3.7) завжди можна розв'язати методом послідовних наближень [3], записавши їхні значення через відповідні резольвенти. У кінцевому випадку динаміка чисельності статей у двостатевій популяції визначається формулою (6.3.3) з врахуванням знайдених Ω_u і Ω_v .

Тепер розглянемо, як, знаючи народжуваність $b(x, t)$, смертність $d(x, t)$ і початковий розподіл чисельності популяції за віком $g(x)$, можна прогнозувати чисельність популяції у часі з врахуванням вікового складу. Тобто, розглянути питання про керування чисельністю із певною метою. Такою метою можуть бути, наприклад, боротьба з популяцією шкідників, підтримання чисельності популяції на деякому сталому рівні, оптимізація деякого економічного критерію тощо.

Система рівнянь у таких випадках має вигляд

$$\begin{cases} \frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = -d(x, t)u + w(x, t), \\ u(0, t) = \int_0^{\infty} b(x, t)u dx + w(0, t). \end{cases}$$

Тут функція w описує інтенсивність штучного притоку або відтоку особин віку x із популяції у момент часу t .

Розглянемо для прикладу випадок, коли керуванням є додатковий коефіцієнт смертності $\mu(x, t)$ (відлов частинами популяції, забій певних особин), тобто рівняння смертності з керуванням має вигляд

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = -\{d(x, t) + \mu(x, t)\}u.$$

Для випадку стаціонарних характеристик $b(x)$ і $d(x)$ маємо для вище згаданої функції співвідношення

$$\Omega(t) = \int_0^t K(x)\Omega(t-x)dx + \int_t^{\infty} K(x)\varphi(x-t)dx.$$

Якщо можна виміряти число новонароджених у популяції при $0 < t < \infty$, то одержимо інтегральне рівняння

$$\int_t^{\infty} K(x)\varphi(x-t)dx = \varphi(t),$$

з якого, знаючи $K(x)$, можна визначити початковий віковий склад популяції $\varphi(x)$, або, знаючи $\varphi(x)$, можна знайти $K(x)$, тобто, деякий зв'язок між народжуваністю та смертністю. Далі, знаючи народжуваність $b(x)$, яка зазвичай відома на відміну від смертності $d(x)$ за формулою

$$d(x) = \frac{d}{dx} \ln \frac{b(x)}{K(x)}.$$

Насправді початковий розподіл $\varphi(x)$ можна одержати, оперуючи не тільки функцією $\Omega(t)$, але й динамікою чисельності популяції, оскільки $N(t) = \int_0^{\infty} u(x,t)dx$ є параметром, який найдоступніший для спостереження.

Іншою проблемою є питання, як за керуванням $\mu(x,t)$ досягти деякої вікової структури

$$\begin{cases} G(x) = G(0)e^{-\int_0^x (d(\xi)+\mu(\xi))d\xi}, \\ G(0) = \int_0^{\infty} b(x)G(x)dx. \end{cases} \quad (6.3.8)$$

З (6.3.8) одержимо умову на керування

$$\int_0^{\infty} K(x)e^{-\int_0^x \mu(\xi)d\xi} dx = 1. \quad (6.3.9)$$

Це рівняння є обмеженням на керування $\mu(x)$ у стаціонарному випадку. Отже, розподіл оптимальної стаціонарної вікової структури виражається співвідношенням (6.3.8) з умовою (6.3.9) і є досяжним.

Подібні моделі використовують у демографічних [37] та інших біопопуляціях [35, 24], для яких методами оптимального керування (принцип максимуму, принцип Bellman'а) можна керувати динамікою природних та штучних популяцій.

6.4. Задача оптимального керування з інтегральною умовою за віковим обмеженням популяції

Цей підрозділ є демонстрацією розв'язування задач оптимального керування для біологічних спільнот і дозволяє одержати необхідні умови оптимальності задачі оптимального керування динамікою популяції, для якої керуюча функція задовольняє додаткову інтегральну умову [1].

На деякому відрізку часу $T = [0, t_k]$ розглянемо функцію двох змінних $x(s, t)$, яка характеризує густину розподілу популяції деякого виду залежно від віку $s \in S = [0, s_k]$, де s_k – максимальний вік.

За припущення, що зміна чисельності популяції відбувається тільки за рахунок народжуваності та смертності особин популяції, одержимо рівняння

$$\frac{\partial x(s, t)}{\partial t} + \frac{\partial x(s, t)}{\partial s} = -\mu(s)x(s, t), \quad s \in S, t \in T, \quad (6.4.1)$$

з умовами

$$x(s, 0) = x_0(s), \quad s \in S, \quad (6.4.2)$$

$$x(0, t) = \beta(t) \int_{s_1}^{s_2} K(s)u(s)x(s, t)ds, \quad t \in T. \quad (6.4.3)$$

Тут $\mu(s)$ – коефіцієнт смертності; $x_0(s)$ – початковий розподіл популяції за віком; $\beta(t)$ – середній рівень народжуваності у момент часу t ; $K(s)$ – доля самок.

Керуванням є функція $u(s)$, яка відображає віковий розподіл самок і підлягає обчисленню

$$\int_{s_1}^{s_2} u(s) ds = 1, \quad (6.4.4)$$

де $0 < s_1 < s_2 \leq s_k$ – межі дітородного віку.

Метою задачі будемо вважати мінімізацію функціоналу

$$I[u] = \int_S \varphi(x(s, t_k), s) ds. \quad (6.4.5)$$

Задачу оптимального керування (6.4.1)-(6.4.5) будемо досліджувати при таких припущеннях на вихідні дані:

1) $u, K \in C^1([s_1, s_2])$,

$$u(s) \equiv 0, K(s) \equiv 0 \quad \forall s \notin [s_1, s_2]; \quad (6.4.6)$$

2) $x_0 \in C^1(S), \beta \in C^1(T)$;

3) $\mu \in C([0, s_k])$,

$$\int_0^{s_k} \mu(s) ds = +\infty, \quad (6.4.7)$$

(ця умова забезпечує нульову густину, якщо вік популяції перевищує s_k);

4) $\varphi(x, s)$ є неперервною за обома змінними і має неперервну і обмежену похідну за x всюди у області свого визначення.

Використовуючи метод характеристик [3], задачу (6.4.1)-(6.4.3) можна звести до інтегрального рівняння, побудованого на сімействі характе-

ристик рівняння (6.4.1)

$$x(s, t) = \begin{cases} x_0(s-t) \cdot e^{-\int_{s-t}^s \mu(\rho) d\rho}, & s \geq t, \\ \beta(t-s) \cdot e^{-\int_0^s \mu(\rho) d\rho} \int_{s_1}^{s_2} K(r)u(r)x(r, t-s)dr, & s < t. \end{cases} \quad (6.4.8)$$

За зроблених припущень для довільного допустимого керування розв'язок задачі (6.4.1)-(6.4.3) в розумінні (6.4.8), буде невід'ємною функцією, гладкою в областях $s < t$ і $s > t$ [3]. Можливою лінією розриву є пряма $s = t$. Тому неперервність функції $x(s, t)$ всюди у розглядуваній області гарантує умова

$$x_0(0) = \beta(0) \int_{s_1}^{s_2} K(s)u(s)x_0(s)ds, \quad (6.4.9)$$

яку називають умовою погодження нульового порядку і вона є додатковим обмеженням на керування.

Задачу (6.4.1)-(6.4.5) дослідимо за допомогою *принципу максимуму Понтрягіна* [1, 9, 20, 12, 13, 14]. Розглянемо приріст цільового функціоналу (6.4.5) на двох допустимих процесах $\{u, x\}$ і $\{\tilde{u} = u + \Delta u, \tilde{x} = x + \Delta x\}$. Функція $\Delta x = \Delta x(s, t)$ є розв'язком початково-крайової задачі

$$\begin{cases} \frac{\partial \Delta x}{\partial t} + \frac{\partial \Delta x}{\partial s} = -\mu(s)\Delta x, & s \in S, t \in T, \\ \Delta x(s, 0) = 0, & s \in S, \\ \Delta x(0, t) = \beta(t) \int_{s_1}^{s_2} K(s)[\tilde{u}(s)\tilde{x}(s, t) - u(s)x(s, t)]ds, & t \in T. \end{cases} \quad (6.4.10)$$

З врахуванням (6.4.10) формулу приросту цільового функціоналу запишемо у вигляді

$$\begin{aligned} \Delta I[u] &= I[\tilde{u}] - I[u] \\ &= \int_S \Delta\varphi(x(s, t_k), s) ds + \int_S \int_T \psi(s, t) \left[\frac{\partial \Delta x}{\partial t} + \frac{\partial \Delta x}{\partial s} + \mu(s)\Delta x \right] dt ds, \end{aligned}$$

де $\Delta\varphi = \varphi(\tilde{x}(s, t_k), s) - \varphi(x(s, t_k), s)$, а $\psi = \psi(s, t)$ – наразі невідома довільна кусково-гладка функція (спряжена функція).

Проведемо такі стандартні перетворення:

- розвинемо приріст $\Delta\varphi$ за формулою Taylor'a першого порядку, виділивши лінійну частину відносно стану процесу;
- використовуємо формулу інтегрування частинами;
- доданки із формули інтегрування частинами

$$\int_T \psi(0, t)\Delta x(0, t) dt$$

перетворюємо із врахуванням (6.4.10);

- використавши (6.4.6), переходимо від інтегрування на відріжку $[s_1, s_2]$ до інтегрування по S , міняючи порядок інтегрування;
- функцію $\psi = \psi(s, t)$ виберемо із спряженої задачі

$$\frac{\partial \psi(s, t)}{\partial t} + \frac{\partial \psi(s, t)}{\partial s} = \mu(s)\psi(s, t) - \psi(0, t)\beta(t)K(s)u(s), \quad s \in S, t \in T, \quad (6.4.11)$$

$$\begin{cases} \psi(s, t_k) = -\frac{\partial \varphi(x(s, t_k), s)}{\partial x}, & s \in S, \\ \psi(s_k, t) = 0, & t \in T. \end{cases}$$

Тоді кінцевий варіант формули приросту можна записати у вигляді

$$\Delta I[u] = -\int_{s_1}^{s_2} K(s)\Delta u(s) \int_T \psi(0, t)\beta(t)x(s, t) dt ds + \eta, \quad (6.4.12)$$

де

$$\eta = - \int_{s_1}^{s_2} K(s) \Delta u(s) \int_T \psi(0, t) \beta(t) \Delta x(s, t) dt ds$$

$$+ \int_S o_\varphi(|\Delta x(s, t_k)|) ds. \quad (6.4.13)$$

Тут останній доданок у (6.4.13) – величина вищого порядку малості щодо $|\Delta x(s, t_k)|$, яку одержуємо із застосуванням формули Таулог’а для $\Delta \varphi$.

Формула приросту функціоналу (6.4.12) одержана для двох довільних допустимих процесів.

Далі дослідження задачі побудована на застосуванні *некласичної варіації допустимих керувань* [1, 20] із врахуванням обмежень (6.4.6).

Нехай $u = u(s)$ – допустиме керування, варіація якого задається формулою

$$u_{\varepsilon, \delta}(s) = \left(1 + \varepsilon \cdot \frac{d\delta(s)}{ds} \right) u(s + \varepsilon\delta(s)).$$

Тут $\varepsilon \in [0, 1]$ – параметр, що характеризує приріст варіювання, $\delta = \delta(s)$ – двічі неперервно-диференційовна функція з умовами

$$s_1 \leq s + \delta(s) \leq s_2, \quad \frac{d\delta(s)}{ds} \geq -1, \quad s \in [s_1, s_2], \quad \delta(s_1) = \delta(s_2) = 0.$$

Застосування цієї варіації для допустимого керування зберігає для керування умову (6.4.4). Справді,

$$\int_{s_1}^{s_2} u_{\varepsilon, \delta}(s) ds = \int_{s_1}^{s_2} \left(1 + \varepsilon \frac{d\delta(s)}{ds} \right) u(s + \varepsilon\delta(s)) ds$$

при заміні $\rho = s + \varepsilon\delta(s)$ співпадає із (6.4.4).

Оцінимо залишковий член (6.4.13) у формулі приросту функціоналу

для двох допустимих процесів $\{u, x\}$ та $\{u_{\varepsilon, \delta} = u + \Delta u, x_{\varepsilon} = x + \Delta x\}$. Скориставшись неперервною диференційовністю допустимих керувань, представимо

$$\begin{aligned} \Delta u(s) &= \left(1 + \varepsilon \frac{d\delta(s)}{ds}\right) u(s + \varepsilon\delta(s)) - u(s) = \\ &= u(s + \varepsilon\delta(s)) - u(s) + \varepsilon \frac{d\delta(s)}{ds} u(s + \varepsilon\delta(s)) = \\ &= \varepsilon \left(\delta(s) \frac{du}{ds} + \frac{d\delta}{ds} u(s) \right) + o(\varepsilon). \end{aligned}$$

Отож, приріст керування є величиною порядку ε .

Інтегрування представлення розв'язку (6.4.8) дає можливість оцінити приріст стану, тобто

$$\Delta x(s, t) = \begin{cases} 0, & s \geq t, \\ \beta(t-s) \cdot e^{-\int_0^s \mu(\rho) d\rho} \times \\ \int_{s_1}^{s_2} K(r) \left[u_{\varepsilon}(r) x_{\varepsilon}(r, t-s) - u(r) x(r, t-s) \right] dr, & s < t. \end{cases}$$

Враховуючи обмеженість вихідних функцій легко одержати нерівність

$$|\Delta x(s, t)| \leq M \int_{s_1}^{s_2} |\Delta x(r, t-s)| dr + L \int_{s_1}^{s_2} |\Delta u(r)| dr,$$

$M, L = \text{const} \geq 0$. З цієї нерівності, враховуючи, що керування є величиною порядку ε , шляхом послідовних наближень одержимо

$$|\Delta x(s, t)| \sim \varepsilon, \quad s \in S, t \in T.$$

Тоді формулу приросту функціоналу можна представити

$$I[u_{\varepsilon, \delta}] - I[u] = -\varepsilon \int_{s_1}^{s_2} K(s) \frac{d}{ds} [\delta(s)u(s)] \int_T \psi(0, t) \beta(t) x(s, t) dt ds + o(\varepsilon). \quad (6.4.14)$$

Підставивши представлення (6.4.14) у формулу приросту функціоналу (6.4.12) і знову інтегруючи частинами, одержимо необхідну умову оптимальності

Теорема 6.4.1. *Нехай $u^* = u^*(s)$ – оптимальне керування задачі (6.4.1)-(6.4.5) і $x^* = x^*(s, t)$ – відповідний йому стан, а $\psi^* = \psi^*(s, t)$ – розв’язок спряженої задачі (6.4.11) при $u^* = u^*(s)$, $x^* = x^*(s, t)$. Тоді $\forall s \in [s_1, s_2]$ виконується умова*

$$\int_T u^*(s) \psi^*(0, t) \beta(t) \frac{\partial}{\partial s} \left(K(s) x^*(s, t) \right) dt = 0. \quad (6.4.15)$$

Умова (6.4.15) є необхідною умовою оптимальності розглянутої задачі.

6.5. Завдання для самостійного опрацювання

◆ *Завдання 6.5.1.* Розв’язати стаціонарну задачу оптимальної

структури популяції

$$\begin{cases} \frac{du}{dt} = -(d_u(x) + \mu_u(x))u, \\ \frac{dv}{dt} = -(d_v(x) + \mu_v(x))v, \\ u(0) = \int_0^{\infty} b_u(x)u dx, \\ v(0) = \int_0^{\infty} b_v(x)u dx. \end{cases}$$

◆ *Завдання 6.5.2.* Методом характеристик та ітерацій розв'язати задачу вікової структури популяції

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial u}{\partial t} = \lambda(x, t, U(t)), \\ u(x, 0) = f_0(x), \quad x \geq 0, \\ u(0, t) = B(t) = \int_a^b \beta(x, t, U(t))u(x, t) dx, \quad t \geq 0, \end{cases}$$

де

$$U(t) = \int_a^b u(x, t) dx, \quad 0 < a < b < \infty.$$

◆ *Завдання 6.5.3.* Розв'язати задачу "лімітованої" популяції під дією керуючого впливу u :

$$\begin{cases} I[u] = \int_0^T u N(t) dt + N(T) \rightarrow \min, \\ \frac{dN}{dt} = (\varepsilon - \gamma N)N - u N, \\ N(0) = N_0. \end{cases}$$

◆ *Завдання 6.5.4.* Задача, що враховує міграцію популяції, зводиться до співвідношень

$$\begin{cases} \frac{\partial x}{\partial t} + \frac{\partial x}{\partial s} = -d(s, t)x + m(s, t), \\ x(0, t) = \int_0^{\infty} b(s, t)x ds, \\ x(s, 0) = \varphi(s). \end{cases}$$

Знак m відповідає притоку ($m > 0$) або відтоку ($m < 0$) особин популяції віком s в момент часу t . Насправді $m(s, t)$ є керуванням динамікою вікової структури популяції. Показати, що розв'язок має вигляд

$$x(s, t) = \tilde{m}(s, t) e^{-\int_0^s d(\xi, t-s+\xi) d\xi},$$

де

$$\tilde{m}(s, t) = \int_0^s m(\xi, t-s+\xi) d\xi.$$

◆ *Завдання 6.5.5.* Якщо у [завданні 6.5.4](#) $m(s, t) = 0$, то

$$x(s, t) = \Omega(t-s) e^{-\int_0^s d(\xi, t-s+\xi) d\xi},$$

де для функції Ω одержуємо інтегральне рівняння

$$\Omega(t) = \int_0^{\infty} K(s, t) \Omega(t-s) ds,$$

$$\Omega(-s) = \tilde{\varphi}(s),$$

$$K(s, t) = b(s, t) e^{-\int_0^s d(\xi, t-\zeta+\xi) d\xi},$$

$$\tilde{\varphi}(s) = \varphi(s) e^{\int_0^s d(\xi, \xi-\zeta) d\xi}.$$

Побудувати розв'язок цього інтегрального рівняння методом послідовних наближень.

◆ *Завдання 6.5.6.* Довести існування та єдиність розв'язку задачі із завдання 6.5.5 методом стискуючих відображень.

◆ *Завдання 6.5.7.* Знайти розв'язок задачі, що описує залежність популяції із параметром запізнення

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = -\lambda(x, z(t))u(x, t),$$

$$u(x, 0) = \nu(x),$$

$$u(0, t) = \int_0^{\infty} \rho(x, z(t))u(x, t)dx$$

$$z(t) = \int_0^{\infty} u(x, t)dx.$$

Бібліографія

- [1] Аргучинцев А.В. *Оптимальное управление гиперболическими системами*. – М: ФИЗМАТЛИТ, 2007. – 168 с.
- [2] Гинзбург Л.Р. *О динамике и управлении возрастнй структурой популяции*// Проблемы кибернетики, 1970. – вып. 23. – С. 261-274.
- [3] Головатий Ю.Д., Кирилич В.М., Лавренюк С.П. *Дифференціальні рівняння: навчальний посібник*. – Львів, ЛНУ імені Івана Франка, 2011. – 470 с.
- [4] Istas, J. *Mathematical Modeling for the Life Sciences*. Springer-Verlag, New York, (2005)
- [5] Лутц Марк. *Изучаем Python, том 1, 5-е изд.: Пер. с англ.* – СПб.: ООО "Диалектика". – 2019. – 832 с.
- [6] Лутц Марк *Изучаем Python, том 2, 5-е изд. : Пер. с англ.* – СПб. : ООО "Диалектика" – 2020. – 720 с.
- [7] Кирилич В.М., Терещук О.В., Флюд В.М. *Оптимальне керування соціально-економічними системами у середовищі MatLab. Навчальний посібник*. – Львів: ЛНУ імені Івана Франка. 2021. – 412 с.

-
- [8] Матвеев А.С., Якубович В.А. Оптимальные системы управления: Обыкновенные дифференциальные уравнения. Специальные задачи: учебн. пособие – Санкт-Петербург: Из-во С.-Петербургского ун-та. 2003. 540.
- [9] Моклячук М.П. *Варіаційне числення. Екстремальні задачі. Підручник.* – Київ, 2004. – 384 с.
- [10] Москаленко А.И. *Оптимальное управление моделями экономической динамики.* – Новосибирск: Наука. – 1999. – 185 с.
- [11] Ногин В.Д. *Введение в оптимальное управление. Учебно-методическое пособие.* – СПб: Изд-во "ЮТАС". 2008. – 92 с.
- [12] Розоноэр Л.И. Принцип максимума Л. С. Понтрягина в теории оптимальных систем. I, Автомат. и телемех., 1959, том 20, выпуск 10, 1320–1334.
- [13] Розоноэр Л.И. Принцип максимума Л. С. Понтрягина в теории оптимальных систем. II, Автомат. и телемех., 1959, том 20, выпуск 11, 1441–1458.
- [14] Розоноэр Л.И. Принцип максимума Л. С. Понтрягина в теории оптимальных систем. III, Автомат. и телемех., 1959, том 20, выпуск 12, 1561–1578.
- [15] Хиллард Дейн. *Секреты Python Pro.* – СПб.: Питер. – 2021. – 320 с.
- [16] Зед А. Шоу. *Легкий способ выучить Python.* – Москва: Издательство «Э»ю – 2017. – 352 с.
- [17] Allen, L.J.S. *An Introduction to Mathematical Biology.* Pearson Prentice-Hall, Upper Saddle River, NJ (2007)
- [18] Borrelli, R.L., Coleman, C.S. *Differential Equations. A Modeling Perspective.* John Wiley & Sons, New York (1998)

-
- [19] Victor A. Canto, Douglas H. Joines, Arthur B. Laffer. *Foundations of Supply-Side Economics – Theory and Evidence*. – New York: Academic Press. – 1982.
- [20] Derevianko T.O., Kyrylych V.M. *Problem of optimal control for a semilinear hyperbolic system of equations of the first order of infinite horizon planning*// Ukrainian Math. Journal. – 2015. – Vol. 67. N2 – P. 211-229.
- [21] Claus Führer, Jan-Erik Solem, Olivier Verdier. *Scientific Computing with Python 3. An example-rich, comprehensive guide for all of your Python computational needs* – Packt Publishing. – 2016. – P. 322.
- [22] Guerrini L. *The Solow-Swan model with a bounded population growth rate*. – J. of Mathematical Economics. – 2006. – Vol. 42. – P. 14-21.
- [23] Christian Hill *Learning Scientific Programming with Python. Second Edition* – Cambridge University Press. – 2020 – P. 571.
- [24] Hoppensteadt F. *Mathematical Theories of Populations: Demographics, Genetics and Epidemics*. – Soc. Ind. Appl. Math. – 1975, Philadelphia.
- [25] Istas, J. *Mathematical Modeling for the Life Sciences*. Springer-Verlag, New York, (2005)
- [26] M.Lachowicz *Teoria sterowania*// Uniwersytet Warszawski. – 2012. – 88s.(<http://mst.mimuw.edu.pl/lecture.php?lecture=tst>)
- [27] Hans Petter Langtangen. *A Primer on Scientific Programming with Python. 5th Edition*. – Springer-Verlag Berlin Heidelberg. – 2016. – P. 942.
- [28] Hans Petter Langtangen. *Python Scripting for Computational Science. Third Edition*. – Springer-Verlag Berlin Heidelberg. – 2008. – P.769.

-
- [29] Steven F. Lott. *Python Essentials. Modernize existing Python code and plan code migrations to Python using this definitive guide.* – Packt Publishing. – 2015. – P. 298.
- [30] Rob Mastrodomenico. *The Python Book.* – John Wiley and Sons Ltd. – 2022. – P. 259.
- [31] Murray, J.D. *Mathematical Biology.* Springer-Verlag, Berlin (1989)
- [32] H. H. Robertson. *The solution of a set of reaction rate equations*, in J. Walsh (Ed.), *Numerical Analysis: An Introduction*, pp. 178–182, Academic Press, London (1966).
- [33] Sandeep Nagar. *Introduction to Python for Engineers and Scientists: Open Source Solutions for Numerical Computation.* – Apress. – 2018. – P. 259.
- [34] *Python. Notes for Professionals.* – Free Programming Books. – 2021. – P. 855.
- [35] Sanchez D.A. *Iteration and Nonlinear Equations of Age-Dependent Population Growth with a Birth Window.* – Mathematical Biosciences. – 1985. – Vol. 73. – P. 61-69.
- [36] Smoller, J. *Shock Waves and Reaction-Diffusion Equations.* Springer-Verlag, New York (1983)
- [37] Song J. *Some Developments in Mathematical Demography and Their Application to the People's Republic of China.* – Theoretical Population Biology. – 1982. – Vol. 22. – P. 382-391.
- [38] John M. Stewart. *Python for Scientists. Second Edition.* – University Printing House, Cambridge. – 2017. – P. 271.
- [39] <https://www.python.org/doc/>
- [40] <https://docs.python.org/3/library/functions.html>

-
- [41] <https://docs.python.org/3/library/math.html>
 - [42] <https://docs.python.org/3/library/index.html>
 - [43] <https://pip.pypa.io/en/stable/>
 - [44] <https://packaging.python.org/en/latest/tutorials/installing-packages/>
 - [45] <https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.nquad.html>
 - [46] <https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html#scipy.integrate.odeint>
 - [47] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html#scipy.integrate.solve_ivp
 - [48] https://matplotlib.org/stable/api/markers_api.html
 - [49] <https://matplotlib.org/stable/tutorials/colors/colormaps.html>

Предметний покажчик

- "податкові канікули", 305
атрибути об'єктів програмування, 10
автономна система, 249
цілком керована система, 251
цілком регулярна система диференціальних рівнянь, 253
цільовий функціонал, 229, 252
динамічне програмування, 283
доля амортизації за рік основних виробничих фондів, 239
допустима функція, 231
допустима крива, 231
допустима область, 265
допустима траєкторія, 231
допустиме керування, 249
допустимий процес, 238, 249
достатня умова оптимальності, 286
ефект Laffer'a, 302
екстремаль функціонала якості, 265
екзогенні змінні, 239
ендогенні змінні, 239
фазовий простір, 248
функціонал, 225
функціонал Hamilton'a, 260, 306
функція
Bellman'a, 285
функція Bellman'a, 284
функція Hamilton'a, 257, 258, 260, 263, 264, 266, 268–270, 276, 277, 293, 304, 311
функція Himmelblau, 153
функція доходу, 319
інтеграл благополуччя, 296
керованість системи, 251
керування із фіксованою дією, 250
керування із нефіксованою дією, 250
корисність питомого випуску, 296
критерій оптимальності, 252
критерій якості, 229
кусково-неперервна функція, 229
лінійний функціонал, 226
лінійно-однорідна неокласична виробнича функція, 240
логістичний закон росту популяції, 318

- матриця Якобі, 152
- метод послідовних наближень, 333
- метод стискуючих відображень, 333
- методи об'єктів програмування, 10
- мінімаль, 231
- множина Mandelbrot'a, 114
- множина допустимих пар функцій, 229
- модель Solow, 239, 240, 245, 294
- модель Solow у відносних показниках, 241
- модель Solow-Swan'a, 239
- модель динаміки вікового складу і чисельності популяції, 312
- модель економічного зростання, 239
- модель оподаткування на прибуток, 299
- модель росту популяції з врахуванням її вікового складу, 320
- модифікована модель Solow-Swan'a, 246
- найпростіша задача варіаційного числення, 230
- некерована система, 251
- некласична варіація допустимих керувань, 328
- необхідна умова оптимальності, 330
- необхідна умова слабкого мінімуму, 232
- необхідні умови слабкого мінімуму, 231
- норма накопичення, 239
- однофакторна економічна система, 294
- однофакторна модель економічного росту, 239
- однорідна функція, 240
- оптимальна траєкторія, 252, 257, 262
- оптимальне керування, 251, 252, 257, 262, 319
- оптимальне керування популяції, 318
- оптимальне керуванням у формі зворотного зв'язку, 286
- оптимальний процес, 252, 257
- оптимальний синтез, 286
- оптимальний стан біологічної спільноти, 317
- оптимальний стан популяції, 318
- основна лема варіаційного числення, 233
- особливі керування, 312
- перша варіація функціоналу, 232
- початкова задача, 249
- принцип максимуму, 258
- принцип максимуму Понтрягіна, 256, 310, 311, 326
- принцип максимуму для задачі із нескінченним об'ємом керування, 262
- принцип оптимальності Bellman'a, 284
- приріст функціоналу, 232
- продуктивність популяції, 310

- простір неперервних функцій, 225
- простір стану, 248
- рекурсивна функція (recursive function), 47
- річний темп приросту числа працюючих, 239
- рівняння
- нерозривності, 313
 - Bellman'a, 285
 - Euler'a, 234, 235
 - Fitzhugh–Nagumo, 215
- розв'язок спряженої системи рівнянь, 262
- розв'язок спряженої задачі, 280
- сильна мінімаль, 231
- сильний мінімум функціоналу, 231
- система
- хижак-жертва, 210
 - слабка мінімаль, 231
 - слабкий мінімум функціоналу, 231
 - спряжена система, 259, 260, 312
 - спряжена система рівнянь, 262
 - спряжене рівняння, 266
 - спряжені функції, 259
 - спряжені змінні, 257, 263
 - стаціонарний режим, 298
 - стаціонарний режим траєкторії, 298
 - стани системи, 250
 - стратегія оподаткування, 300
 - теорія оптимального керування, 230
 - термінальний функціонал, 226
 - тіньова ціна, 278
 - тіньова ціна капіталу, 278
 - тіньова ціна одиниці кінцевого капіталу, 278
 - тіньова ціна приросту фондів, 296
 - точка перемикання керування, 305
 - траєкторія рівноважна (стаціонарна, стійка), 241
 - траєкторія системи, 248
 - траєкторія збалансованого росту, 298
 - умова погодження нульового порядку, 326
 - умова слабого мінімуму, 233
 - умова трансверсальності, 274
 - умова трансверсальності для задачі оптимального керування із вільним правим кінцем, 275
 - умови додаткової (доповнюючої) нежорсткості, 274
 - умови додаткової нежорсткості, 274
 - умови слабкої мінімалі, 231
 - умови трансверсальності, 274
 - варіація функціоналу, 233
 - варіація кривої, 231
 - виробнича функція Cobb-Douglas'a, 245, 247
 - вікова густина чисельності особин, 320
 - задача Cauchy, 249, 282, 301
 - задача із рухомими кінцями, 272
 - задача із вільним правим кінцем траєкторії, 272

-
- задача керування із лівим рухомим кінцем траєкторії, 273
- задача оптимального керування, 252
- задача оптимального керування із нескінченним обрієм, 262
- задача оптимального керування з рухомими кінцями, 273
- задача оптимальної швидкодії, 253
- задача про обидва рухомі кінці траєкторії, 273
- жорстка задача, 147
- axes (осі), 67
- hamiltonian (гамільтоніан), 257, 296
- Jacobian-matrix, 152
- rank (ранг), 67
- ufunc (універсальна функція), 83

Навчальне видання

Кирилич Володимир Михайлович
Терещук Оксана Володимирівна
Флод Володимир Михайлович

**Оптимальне керування моделями соціально-економічної
динаміки у середовищі *Python***

Навчальний посібник

Редактор *Н. Й. Плиса*
Комп'ютерний набір і верстання *В. М. Флод*

Підп. до друку __.__.2022. Формат __ × __/__. Папір друк. Друк офсет.
Гарнітура TeX. Умовн. друк. арк. __.__. Обл.-вид. арк. __.__. Тираж ___
прим.

Видавець та виготовлювач:
Львівський національний університет імені Івана Франка,
вул. Університетська, 1, м. Львів, 79000

СВІДОЦТВО

*про внесення суб'єкта видавничої справи до Державного реєстру видавців,
виготівників і розповсюджувачів видавничої продукції: Серія ДК №3059 від
13.12.2007 р.*