Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

СУЧАСНІ МЕТОДИ АНАЛІТИЧНОГО І ЧИСЛОВОГО ПРОГРАМУВАННЯ В ЗАДАЧАХ ТЕОРЕТИЧНОЇ ФІЗИКИ

Методичні рекомендації

Чернівці
Чернівецький національний університет
2021

УДК 53.072; 53:004; 519.6:517

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

Сучасні методи аналітичного і числового програмування в задачах теоретичної фізики: метод. рекомендації / укл.: Ю.О. Сеті. – Чернівці : Чернівецький нац. ун-т, 2021. – 69 с.

Викладаються найбільш важливі у задачах теоретичній фізиці методи теорії чисельного аналізу та алгоритми їх програмної реалізації. Розглядаються можливості бібліотек науково-технічного спрямування мови програмування Python (SciPy, NumPy, Symbol, Matplotlib та інших) та системи комп'ютерної алгебри Wolfram Mathematica для чисельного та аналітичного (символьного) розв'язання задач інтерполяції, диференціювання, інтегрування, пошуку коренів лінійних і нелінійних рівнянь та їх систем, пошуку екстремуму функції, розрахунку сум і рядів.

Для студентів ОНП Теоретична фізика та комп'ютерне моделювання та інших ОП спеціальності Фізика та астрономія.

УДК 53.072; 53:004; 519.6:517

1. ВСТУП

Сучасні наукові дослідження потребують розв’язання різних математичних задач: знаходження коренів нелінійних рівнянь та алгебраїчних рівнянь, задач лінійної алгебри, обчислення інтегралів, розв’язання систем диференціальних рівнянь, інтерполяції тощо. Оскільки здійснити точний розрахунок часто не вдається, то це зумовлює необхідність використання алгоритмів чисельних методів, які є ядром комп’ютерного моделювання. Тому важливим етапом у підготовці фізиків-теоретиків є оволодіння чисельними методами не лише на рівні користувача, а й на рівні фахівця-розробника.

За останні 50 – 60 років, пов'язаних із використанням комп'ютерної техніки, в усьому світі накопичено велику кількість програмного забезпечення у вигляді бібліотек комп'ютерних підпрограм (написаних, у першу чергу, мовами Fortran та C), призначених для розв'язання типових математичних задач. Крім того, розроблено ряд універсальних математичних пакетів, за допомогою яких можна достатньо швидко як розв'язати багато відомих математичних задач, так і провести попередні розрахунки з подальшою реалізацією вже в проектованій комп'ютерній системі. Найбільш відомими комерційними пакетами для проведення наукових та інженерних розрахунків є пакет програм MATLAB, системи комп'ютерної алгебри Mathcad, Maple та Mathematica. Зазначені програмні пакети мають просунуті та зручні графічні інтерфейси, власні високорівневі мови програмування та дозволяють вирішувати велику кількість математичних та інженерних завдань, а також візуалізувати дані та отримувати графіки високої якості. Однак можливість використання цих пакетів індивідуальними дослідниками та невеликими організаціями ускладнена через високу вартість їхніх ліцензій.

Пакет Mathematica найпопулярніший у наукових колах, особливо у теоретиків. Він надає широкі можливості під час проведення аналітичних (символічних) розрахунків, дозволяє швидко й ефективно розв'язувати багато задач обчислювальної математики. В пакеті Mathematica більшість задач розв'язується в діалоговому режимі, без традиційного програмування з використовуванням стандартних операторів. У цьому пакеті реалізована мова програмування високого рівня, яка дозволяє створювати достатньо складні програми. Також цей пакет дозволяє конвертувати документи у формат LaTex, який є стандартним форматом переважної більшості наукових видавництв світу. Безкоштовне користування системою комп'ютерної алгебри Mathematica реалізоване у хмарному сервісі Wolfram Cloud (https://www.wolframcloud.com).

Програмна реалізація розв’язання задач обчислювальної математики також може бути ефективно реалізована на відносно новій алгоритмічній мові Python, що активно розвивається. Це високорівнева мова програмування загального призначення максимально орієнтована на продуктивність розробника та читання коду, підтримується більшістю використовуваних платформ і поширюється вільно. Python має багату стандартну бібліотеку. Крім того, для Python написано велику кількість прикладних бібліотек, у тому числі для наукових розрахунків, які дозволяють розв'язувати низку задач без необхідності самостійної розробки алгоритмів. Наразі реалізовано велику кількість бібліотек та програмних пакетів як для аналітичного, так і для чисельного розв'язання наукових задач. Основні пакети, які використовуються для наукових та інженерних розрахунків - NumPy, SciPy, matplotlib і SymPy. Пакет matplotlib має добре реалізовані можливості візуалізації двовимірних і тривимірних даних. Основою пакетів NumPy і SciPy є чисельні розрахунки, а символьні обчислення реалізовані у бібліотеці SymPy. Спільні можливості бібліотек

NumPy, Matplotlib та SciPy близькі до можливостей MATLAB. Дані бібліотеки активно розвиваються і є одними з основних безкоштовних інструментів для розв'язання задач математики та аналізу даних, які виникають в теоретичній фізиці.

Існує низка інших безкоштовних програмних пакетів та бібліотек для наукових розрахунків, аналізу та візуалізації даних. Серед них можна виділити пакети для чисельних розрахунків Scilab (https://www.scilab.org/) та Octave (https:// www.gnu.org/software/octave/), систему комп'ютерної алгебри Maxima (http://maxima.sourceforge.net/ru/index.html), систему для статистичних розрахунків та візуалізації даних на основі мови програмування R (https://www.r-project.org /).

За великим рахунком, не має значення, який математичний пакет чи мова застосовується, адже головне – це розуміння суті та особливостей чисельних методів і способів їх застосування для розв'язання практичних задач.

Видання присвячене розв'язанню основних задач чисельного аналізу, актуальних в теоретичній фізиці. Кожен розділ містить інформацію про суть чисельних методів, функції системи комп'ютерної алгебри Mathematica та библиотек NumPy, Matplotlib и SciPy мови Python, а також довідковий матеріал з алгоритмами програмної реалізації розглядуваних чисельних методів.

2. ІНТЕРПОЛЯЦІЯ ФУНКЦІЙ

Загальна постановка задачі інтерполяції така: є набір $y_1,..., y_n$ значень функції y(x) в точках $x_1, ..., x_n$. Потрібно отримати значення функції y(x) в деякій довільній точці x. Значення x може бути або дійсним, або комплексним, що цілком допускає чисельне «аналітичне продовження» функцій.

В теоретичній фізиці інтерполяція зустрічається не рідше, ніж в експериментальній. Наприклад, при табуляції спецфункції, яка обчислюється повільно, є сенс обчислити її на розрідженій сітці, а в усі інші точки проінтерполювати. Задача аналітичного продовження функції з дійсної осі теж зустрічається часто. Крім того, інтерполяція є складовою частиною багатьох інших алгоритмів чисельних методів.

2.1. Інтерполяція поліномами

Розглянемо формулу поліноміальної інтерполяції - формулу Лагранжа

$$P(x) = \sum_{i=1}^{n} y_i \prod_{k \neq i} \frac{x - x_k}{x_i - x_k}.$$

Користуватися безпосередньо цією формулою при реалізації поліноміальної інтерполяції не варто. Краще застосовувати рекурсивну процедуру. Для кожної точки *x*, в яку здійснюється інтерполяція, потрібно виконати такі операції:

Алгоритм (аглогитм Невилля)

Спочатку визначаються величини (одновимірний масив):

n величин

$$P_{1,1} \equiv y_1 \qquad P_{2,2} \equiv y_2 \qquad \dots \qquad P_{n,n} \equiv y_n.$$

Далі за формулою

$$P_{i,k} = \frac{P_{i,k-1}(x_k - x) + P_{i+1,k}(x - x_i)}{(x_k - x_i)} \tag{1}$$

обчислюють послідовно

n-1 величину:

$P_{1,2}$ $P_{2,3}$ $P_{n-1,n}$

n-2 величину:

$P_{1,3}$ $P_{2,4}$ $P_{n-2,n}$

…

дві величини:

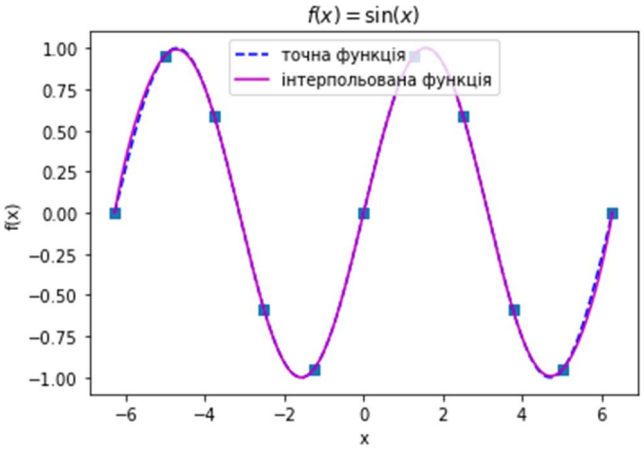
$P_{1,n-1}$ $P_{2,n}$

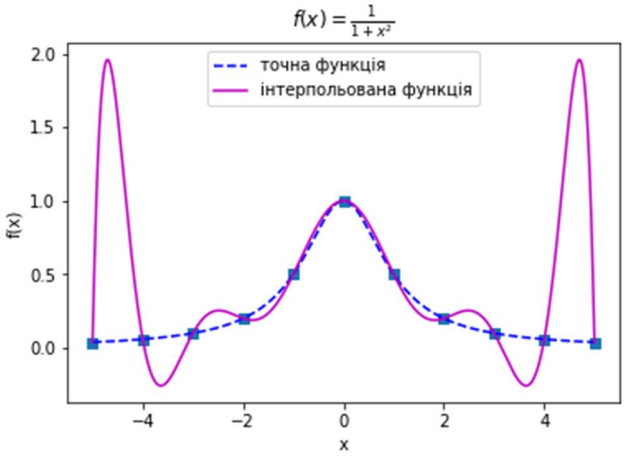
і, нарешті, одну величину $P_{1,n}$, яка і є шуканою відповіддю.

Формула (1) записана так, що $P_{i,k}$ є поліномом (k - i) -го порядку щодо x, який в точках $x = x_i, .., x_k$ має значення $y_i ..., y_k$. Для $P_{k,k}$ це вірно за визначенням, а при кожному наступному застосуванні (1) ми отримуємо, що: при $x=x_i$ $P_{i,k-1} = y_i$; при $x=x_k$ $P_{i+1,k} = y_k$; при інших x_j $P_{i,k-1} = P_{i+1,k} = y_j$.

Немає необхідності пам'ятати проміжні значення $P_{i,k}$, так що при реалізації алгоритму двовимірний масив не потрібен - достатньо одновимірного.

З одного боку, при цій процедурі для кожного чергового x всі викладки доводиться проводити заново. Але з іншого боку, ця рекурсивна процедура дозволяє контролювати похибки. Дійсно, похибку можна оцінити за останньою поправкою, тобто за різницею між остаточною відповіддю $P_{1,n}$ (інтерполяція по n точкам) і попередніми двома відповідями $P_{1,n-1}$ і $P_{2,n}$ (інтерполяції по n - 1 точкам).





Слід пам’ятати, що якщо число точок парне і при цьому функція y_k на цих точках парна, тобто $y_{n-k+1}=y_k$, то остання поправка тотожно дорівнює нулю. Те ж саме виходить у випадку непарної кількості точок для непарної функції y_k ($y_{n-k+1}= -y_k$). Обидва ці випадки нерідко зустрічаються в роботі фізика-теоретика. У цих випадках оцінити похибку можна по передостанній поправці, тобто порівнюючи $P_{1,n}$ з $P_{1,n-2}$, $P_{2,n-1}$ і $P_{3,n}$.

Оцінювати похибку абсолютно необхідно, інакше можна отримати безглуздий фізично некоректний результат!

Похибки при полиноміальній інтерполяції, як правило, зростають поблизу кінців інтервалу, на якому лежать x_1 ... x_n. Крім того, зазвичай, зі зростанням кількості точок n точність зростає. Однак це вірно тільки для тих функцій, які добре апроксимуються поліномом, тобто для функцій з великим радіусом збіжності степеневого ряду. Іншими словами, якщо полюса функції лежать досить далеко від точок x_1 ... x_n, то чим більша кількість точок n, тим точніше інтерполяція. Приклад

функції (~cos(z)), для якої поліноміальна інтерполяція досить добре працює і збільшення *n* призводить до зменшення помилок, приведена на рисунку. Тут інтерполяція йде по 10 точкам; фактична відносна похибка інтерполяції (різниця між функцією і полиномом) змінюється від 0.1% всередині інтервалу до 5% на його кінцях. Якщо інтерполяцію цієї ж функції проводити по 20 точкам, то похибка інтерполяції зменшиться до 10^{-6}.

Однак якщо умова віддаленості від полюсів не виконується, то збільшення числа точок *n* не покращує, а погіршує ситуацію - просте з'єднання сусідніх точок прямими (лінійна інтерполяція по двом сусіднім точкам) може дати більш точну відповідь, ніж поліноміальна інтерполяція по всіх точках. Типова функція $f=(x^2+1)^{-1}$, для якої поліноміальна інтерполяція не може бути застосована, зображена на рисунку. Ця функція має полюси $x = \pm i$ поблизу точок x_i тому радіус збіжності степеневого ряду дуже малий і результати поліноміальної інтерполяції незадовільні. Легко перевірити, що з ростом числа точок *n* відхилення від точної функції тільки зростає. Тому оцінка похибки, закладена в алгоритмі, дозволяє виявляти непридатність полиноміальної інтерполяції для цієї функції.

Розглянемо інше зображення формули Лагранжа, яке допускає послідовне уточнення результатів обчислень у разі додавання нових вузлів - це інтерполяційний багаточлен Ньютона

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

де $f(x_0,x,,...,x_k)$ - поділена різниця k-го порядку. Для неї відома рекурентна формула

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

Багаточлен

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

називають інтерполяційним багаточленом Ньютона інтерполювання вперед для нерівновіддалених вузлів. Його зазвичай використовують для наближення функції біля початкового вузла.

На відміну від полінома Лагранжа, в якого кожний член залежить від усіх вузлів інтерполювання, будь-який k -й член багаточлена Ньютона залежить лише від k перших вузлів інтерполювання і від значення функції в цих вузлах.

У випадку рівновіддалених вузлів $x_i = x_{i-1} + h$ поділені різниці замінюють скінченними різницями

$$f(x_k, x_{k-1}, ..., x_0) = \frac{\Delta^k y_0}{k! h^k}.$$

Одержимо, відповідно, інтерполяційну формулу Ньютона інтерполювання вперед для рівновіддалених вузлів

$$P_n(x) = y_0 + \frac{t}{1!}\Delta y_0 + \frac{t(t-1)}{2!}\Delta^2 y_0 + \frac{t(t-1)(t-2)}{3!}\Delta^3 y_0 + ... + \frac{t(t-1)...(t-k+1)}{k!}\Delta^k y_0;$$

та інтерполяційну формулу Ньютона інтерполювання назад для рівновіддалених вузлів

$$P_n(x) = y_n + \frac{t}{1!}\Delta y_{n-1} + \frac{t(t+1)}{2!}\Delta^2 y_{n-2} + \frac{t(t+1)(t+2)}{3!}\Delta^3 y_{n-3} + ... + \frac{t(t+1)...(t+k-1)}{k!}\Delta^k y_{n-k}$$

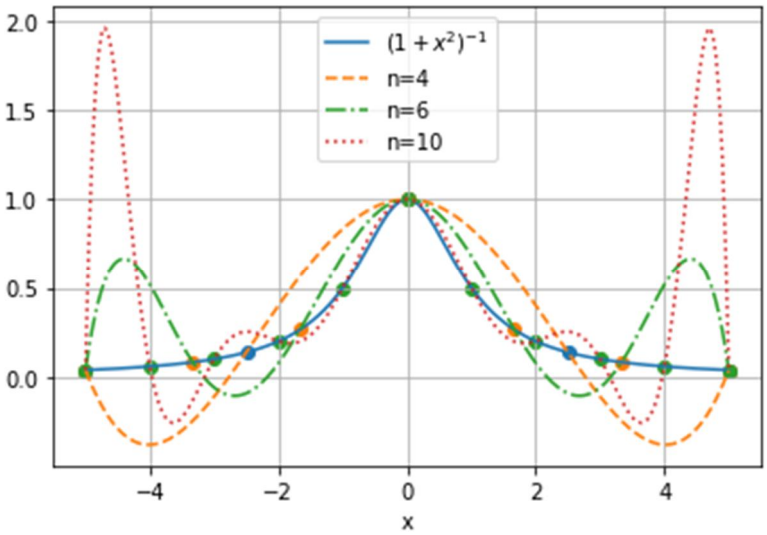
Формулу Ньютона інтерполювання вперед використовують у разі інтерполювання функції для значень x, близьких до початкового вузла x_0, а інтерполювання назад - до x_n.

З оцінок похибки багаточлена Ньютона видно, що зі зменшенням кроку похибка методу буде спадати як h^{n+1}.

Приклад програмної реалізації інтерполяції за формулою Ньютона

```
def interpolation(c, x, x0):
    """Розрахунок полінома Ньютона в точці x0"""
    n = len(x) - 1  # Порядок полінома
    y0 = c[n]
    for k in range(1, n+1):
        y0 = c[n-k] + (x0 - x[n-k]) * y0
    return y0

def coef(x, y):
    """Розрахунок коефіцієнтів полінома Ньютона"""
    m = len(x)      # Кількіть точок
    c = y.copy()
    for k in range(1,m):
        c[k:m] = (c[k:m] - c[k-1])/(x[k:m] - x[k-1])
    return c

import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return 1 / (1 + x**2)

a = -5.
b = 5.
x1= np.linspace(a, b, 200)
y1 = f(x1)

plt.plot(x1, y1, label=r'$(1+x^2)^{-1}$')
nList = [4, 6, 10]
sglist = ['--', '-.', ':']

for k in range(len(nList)):
    n= nList[k]
    x= np.linspace(a, b, n+1)
    y= f(x)
    plt.scatter(x, y, marker='o')
    c= coef(x, y)
    y1 = interpolation(c, x, x1)
    s1= 'n=' + str(n)
    sg = sglist[k]
    plt.plot(x1, y1, sg, label=s1)
```

```
plt.xlabel('x')
plt.grid(True)
plt.legend(loc=0)
plt.show()
```



2.2. Раціональна інтерполяція

При раціональній інтерполяції будується дробно-раціональна функція

$$(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0)/(b_m x^m + b_{m-1}x^{m-1} + \ldots + b_0),$$

яка в точках x_i … x_N дорівнює y_i … y_N. Оскільки чисельник і знаменник визначені з точністю до загального множника, то кількість точок, які визначають цю функцію, буде $N = (m + 1) + (n + 1) - 1$.

За допомогою раціональної інтерполяції можна успішно проінтерполювати набагато ширший клас функцій, ніж за допомогою поліноміальної. Особливо це стосується задачі про аналітичне продовження функції. Однак, необхідно враховувати, що раціональна інтерполяція здатна давати сингулярності (в тому числі і на дійсній осі) в точках $x = x_0$, які є корнями знаменника. У деяких випадках (коли у функції дійсно є сингулярність) це є перевагою методу. На жаль, буває і так, що через похибки в початкових даних (y_k відомі з недостатньою точністю), раціональна інтерполяція дає неправдиві сингулярності на дійсній осі, в той час як сама функція в околі цих точок подібної поведінки не має.

Розглянемо алгоритми здійснення раціональної інтерполяції.

Метод прямого обчислення

З одного боку, цей алгоритм простий і наочний; він дозволяє брати довільну комбінацію m і n; отримати явний вигляд функції; для фіксованого набору даних x_1 ... x_N, y_1 … y_N коефіцієнти a_0 ... a_n і b_1 ... b_m знаходяться один раз, а далі можна інтерполювати в необмежену кількість точок x. З іншого боку, в цьому алгоритмі не закладений контроль точності; в ньому досить великий ризик отримати великі помилки округлення.

Алгоритм

Вважаємо $b_0 \equiv 1$ і запишемо систему лінійних рівнянь $(m+n+1)*(m+n+1)$ для коефіцієнтів $a_0, \ldots, a_n$ і $b_0, \ldots, b_m$:

$$(a_n x_k^n + a_{n-1}x_k^{n-1} + \ldots + a_0) - y_k(b_m x_k^m + b_{m-1}x_k^{m-1} + \ldots + b_1 x_k^1) = y_k;$$

k=1, ..., n+m+1.

Ця система рівнянь легко розв’язується.

Рекурсія переворотом дробу

Алгоритм

Будуємо рекурсію: раціональна функція $R^{(n)}(x)$ повинна дорівнювати y_k ° $y_k^{(n)}$ в точках x_k, k=1,....n. Визначаємо раціональну функцію $R^{(n-1)}(x)$ з

$$R^{(n)}(x) = (x - x_n)/R^{(n-1)}(x) + y_n^{(n)},$$

тоді $R^{(n-1)}(x)$ рівна $y_k^{(n-1)}$ в точках x_k, k = 1, ... n-1, де

$$y_k^{(n-1)} \equiv (x_k - x_n)/(y_k^{(n)} - y_n^{(n)}).$$

Далі визначаємо раціональну функцію $R^{(n-2)}(x)$ з

$$R^{(n-1)}(x) = (x - x_{n-1})/R^{(n-2)}(x) + y_{n-1}^{(n-1)},$$

тоді $R^{(1)}(x)$ повинна дорівнювати $y_k^{(n-2)}$ в точках x_k, k = 1, ... n-2, де

$$y_k^{(n-2)} \equiv (x_k - x_{n-1})/(y_k^{(n-1)} - y_{n-1}^{(n-1)}).$$

...

Нарешті,

$$R^{(2)}(x) = (x - x_2)/R^{(1)}(x) + y_2^{(2)},$$

тоді $R^{(1)}(x)$ дорівнює $y_1^{(1)}$ в точці x_1, де

$$y_1^{(1)} \equiv (x_1 - x_2)/(y_1^{(2)} - y_2^{(2)}).$$

Отже, $R^{(1)}(x) = y_1^{(1)}$

На прямому ході рекурсії обчислюються всі $y_k^{(n-1)}$, потім всі $y_k^{(n-2)}$ ... $y_1^{(1)}$. На зворотному ході рекурсії при інтерполяції в деяку точку x обчислюються $R^{(1)}(x) = y_1^{(1)}$, $R^{(2)}(x)$, $R^{(3)}(x)$, ... $R^{(n)}(x)$. Зауважимо, що на зворотному ході рекурсії потрібно пам'ятати тільки значення $y_1^{(k)}(k = 1,...,n)$, тому двовимірний масив для цього алгоритму створювати не потрібно.

На жаль, незважаючи на рекурсивний характер алгоритму, величини $R^{(k)}(x)$ різко змінюються від кроку до кроку і, відповідно, контролювати точність інтерполяції немає можливості.

Рекурсія, аналогічна поліноміальній

Алгоритм

Як і при поліноміальній інтерполяції, для кожної точки x визначаються два набори величин (два одновимірних масиви):

n величин

$$R_{1,0} \equiv 0 \qquad R_{2,1} \equiv 0 \qquad \ldots \qquad R_{n,n-1} \equiv 0$$

n величин

$$R_{1,1} \equiv y_1 \qquad R_{2,2} \equiv y_2 \qquad \ldots \qquad R_{n,n} \equiv y_n.$$

Після цього за допомогою формули

$$R_{i,k} = R_{i+1,k} + \frac{R_{i+1,k} - R_{i,k-1}}{\left(\frac{x - x_i}{x - x_k}\right)\left(1 - \frac{R_{i+1,k} - R_{i,k-1}}{R_{i+1,k} - R_{i+1,k-1}}\right) - 1} \qquad (2)$$

послідовно обчислюються:

n величин:

$$R_{1,2} \qquad R_{2,3} \qquad \dots \qquad R_{n-1,n}$$

n-1 величина:

$$R_{1,3} \qquad R_{2,4} \qquad \dots \qquad R_{n-2,n}$$

…

дві величини:

$$R_{1,n-1} \qquad R_{2,n}$$

одна величина

$$R_{1,n},$$

яка і є шуканою відповіддю.

Легко перевірити, що дробно-раціональна функція $R_{i,k}$ має в точності ті ж властивості, що і поліном $P_{i,k}$, тобто в точках x = x_i ,..., x_k приймає значення y_i, ..., y_k. Очевидно, що всі розрахунки за формулою (2) повинні супроводжуватися перевірками на нуль в знаменнику. Як і для поліноміальної інтерполяції, алгоритм дозволяє оцінити похибку за останньою поправкою.

Існує одна можливість збою цього алгоритму: він непридатний, якщо одна або декілька y_i рівні нулю (або виявляються машинним нулем в порівнянні з іншими y_k). Це пов'язано з тим, що на першому обороті алгоритму ми маємо функцію 1/(ax+b), яка не може в одній точці дорівнювати нулю, а в іншій точці бути відмінною від нуля. Цей недолік алгоритму легко виправити шляхом зсуву всіх y_i на довільну константу C до інтерполяції і зсуву отриманої відповіді на -C після інтерполяції.

2.3. Фур'є-інтерполяція

При Фур'є-інтерполяції набір значень x_i не довільний - він повинен бути рівномірним ($x_i = a + dx * i$). Очевидно, що будь-яку рівномірну сітку лінійним перетворенням можна перетворити у сітку типу $x_i = i$. Тоді для будь-якої проміжної точки x можна записати:

$$f(x) \simeq \sum_{0}^{N-1} c_k \exp(-2\pi i k x/N)/\sqrt{N}$$

де c_k та y_i визначаються прямим та зворотнім перетвореннями Фур'є:

$$c_k = \sum_{0}^{N-1} y_n \exp(2\pi i k n/N)/\sqrt{N}; \quad y_n = \sum_{0}^{N-1} c_k \exp(-2\pi i k n/N)/\sqrt{N}.$$

При обчисленні f(x) для кожного даного x потрібно обчислити весь набір значень $\exp(-2\pi i k x/N)$ (тобто $\sin(2\pi k x/N)$ та $\cos(2\pi k x/N)$) при k=0,…,N-1, де тригонометричні функції розраховуються тільки один раз - для k=1, а для інших k можна використати рекурсію.

Для фізика-теоретика основне джерело похибок - це помилки округлення і принципова неточність алгоритму. Тому якщо в значеннях y_i є похибки, то, з точки зору Фур'є-аналізу, вони, швидше за все є високочастотним шумом. У цьому випадку варто відхилитися від точної інтерполяції, при цьому в точках x_i інтерполяція потраплятиме не зовсім в y_i, але функція f(x) буде більш гладкою. Де саме знаходиться межа між високочастотними компонентами функції y(x) і шумом залежить від конкретної задачі. Як правило, шумом можна вважати діапазон значень k від N/4 до 3N/4 або N/6 до 5N/6.

2.4. Інтерполяція поліномами Чебишова

Поліноми Чебишова 1-го роду визначаються так:

$$T_n(x) = \cos(n \arccos(x)),$$

і обчислюються вони рекурсією:

$$T_{n\pm1}(x) = \cos(n \arccos(x)) \cos(\arccos(x)) \mp$$
$$\mp \sin(n \arccos(x)) \sin(\arccos(x)),$$

звідки

$$T_{n-1}(x) + T_{n+1}(x) = 2xT_n(x),$$
$$T_0(x) = 1 \qquad T_1(x) = x.$$

Коренями n-го полінома є $x_k = \cos(\pi(k+1/2)/n), k = 0,...,n-1$. Легко перевірити, що значення поліномів T_l при $l = 0,...,n-1$ в точках x_k утворюють базис. Тому якщо відомі значення деякої функції в точках x_k, то можна побудувати поліном

$$T(x) = \sum_{l=0}^{n-1} c_l T_l(x),$$

який в точках $x = x_k$ дорівнюватиме функції $f(x_k)$.

Тут

$$c_l = \sum_{k=0}^{n-1} f(x_k) T_l(x_k) / M_l$$
$$M_l = (1 + \delta_{l0}) \frac{n}{2}$$

Поліном, отриманий за допомогою набору точок x_k дуже близький до функції за нормою. Крім того, він дозволяє не просто обчислити поліном в будь-якій точці x, а й інтегрувати і диференціювати інтерпольовані функції.

2.5. Сплайни

Існують сплайни довільних степенів, але використовувати варто тільки кубічні сплайни. Основна ідея полягає в тому, що замість однієї функції, що проходить через усі точки, на кожному відрізку береться нова функція - кубічна парабола. Всі кубічні параболи на межах відрізків зшиваються, тобто забезпечується неперервність функції, її похідної та її другої похідної. Кубічна парабола визначається двома параметрами:

$$y(x) = [y_a(x - x_b) + y_b(x_a - x)]/(x_a - x_b) +$$
$$+ A(x - x_a)^2(x - x_b) + B(x - x_a)(x - x_b)^2,$$

при цьому

$$y'(x) = (y_a - y_b)/(x_a - x_b) + 2A(x - x_a)(x - x_b) + A(x - x_a)^2 + 2B(x - x_a)(x - x_b) + B(x - x_b)^2$$
$$y''(x) = (4A + 2B)(x - x_a) + (2A + 4B)(x - x_b).$$

тобто

$$A = (y''_a + 2y''_b)/[6(x_b - x_a)] \qquad B = (2y''_a + y''_b)/[6(x_a - x_b)].$$

Отже, на сітці $x_1, x_2, ..., x_n$ є N вільних параметрів $y_1, y_2, ..., y_n$. На них записуються N-2 умови неперервності перших похідних $y_1, y_2, ..., y_n$ на границях послідовних відрізків:

$$y'_{a:(n,n+1)} = (y_{n+1} - y_n)/(x_{n+1} - x_n) + (x_{n+1} - x_n)^2(2y''_n + y''_{n+1})/[6(x_n - x_{n+1})] =$$
$$= (y_n - y_{n-1})/(x_n - x_{n-1}) + (x_n - x_{n-1})^2(2y''_{n-1} + y''_n)/[6(x_{n-1} - x_n)] = y'_{b:(n-1,n)},$$

n=2,...,N-1.

Якщо відомі значення похідних y_1 і y_N, тоді отримуються два шуканих рівняння. Якщо похідні невідомі, то покладають $y_1 = 0 = y_N$.

Як правило, інтерполяція сплайном дає гладкий і найменш звивистий результат. Саме тому більшість графічних програм, які малюють графіки по точкам, використовують саме сплайн. Однак особливого математичного сенсу у сплайновій кривій немає, це просто спосіб провести щось «візуально-гладке» через даний набір точок.

2.6. Інтерполяція засобами бібліотек Python

2.6.1. Чисельна інтерполяція функціями бібліотеки scipy.interpolate

У модулі scipy.interpolate реалізовані алгоритми інтерполяції та згладжування одновимірних та двовимірних функцій. Зокрема реалізовані апроксимації та згладжуванню з використанням кубічних сплайнів. Використовується багато можливостей пакета програм на FORTRAN FITPACK23. Для побудови інтерполюючого полінома можна скористатися функцією lagrange(). Більш стійкі алгоритми обчислення значень інтерполюючого полінома реалізовані у функціях barycentric_interpolate() и krogh_interpolate().

Основною функцією одновимірної інтерполяції є interpld():
interp1d(x, y, kind='linear', axis=-1, copy=True, bounds_error=None, fill_value=nan, assume_sorted=False)

перший аргумент якої є масив вузлів, другий масив значень. Параметр kind задасть тип кусково-поліпоміальної інтерполяції. При kind = 'linear' (значення за замовчуванням) використовується кусково-лінійна інтерполяція (лінійні сплайни), kind = 'cubic' - кубічні сплайни, значення kind = 'nearest', 'zero' пов'язані з кусково-постійним апроксимаціями. Ціле значення kind відповідає степеню інтерполюючого полінома між вузлами. При заданні у вузлах інтерполяції похідних кусково -поліноміальна апроксимація будується за допомогою функції piecewise_polynomial_interpolate().

Функція interp1d() повертає клас, що реалізує функцію, яка інтерполює вихідні дані заданим методом.

Модуль interpolate містить багато інструментів для роботи зі сплайнами. Зокрема, можна не тільки обчислити значення сплайну в заданій точці, але і знайти коефіцієнти сплайну, обчислити похідну сплайну, знайти інтеграл на заданому інтервалі, корені сплайна.

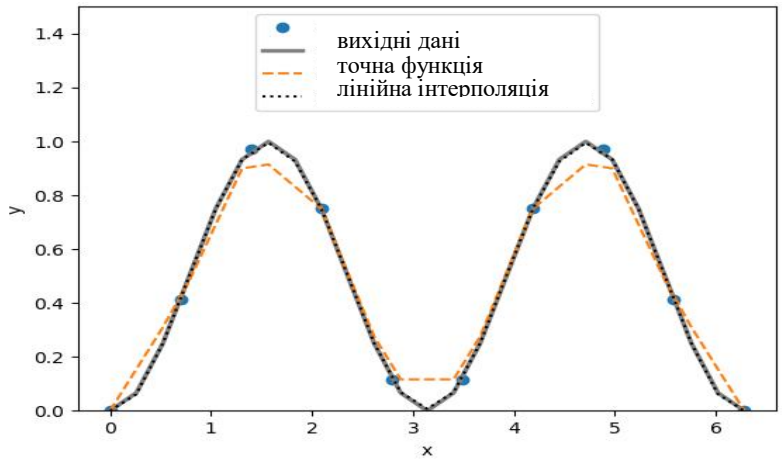
Для інтерполяції двовимірних даних на двовимірній сітці можна користуватися функціями bisplrep() (знаходження B-сплайну) та bisplev() (розрахунок значень двомірного B-сплайну).

З детальним описом функцій модуля scipy.interpolate можна ознайомитися за посиланням https://docs.scipy.org/doc/scipy/reference/interpolate.html

Приклад програм з використанням interp1d:

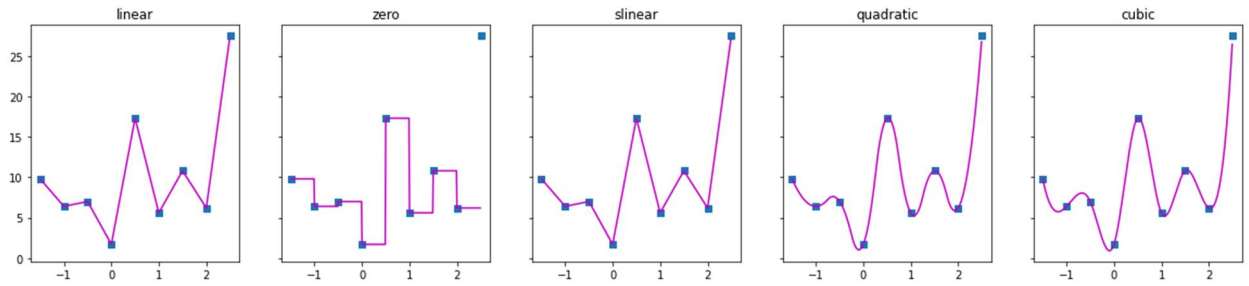
Інтерполяція функції, заданої аналітично

```
import matplotlib.pyplot as plt
from numpy import sin, linspace, pi
from scipy.interpolate import interp1d
f_exact = lambda x: sin(x)**2                    # функція, інтерполяція якої буде здійснюватися
xi = linspace (0, 2.0 * pi, 10) # масив координат, за якими робитиметься інтерполяція
yi = f_exact(x)                                  # масив значень функції, за якими робитиметься інтерполяція
fi_1 = interp1d(x, y)              # інтерполяція даних (за замовчуванням лінійна інтерполяція)
fi_2 = interp1d(x, y, 'cubic')               # кубічна інтерполяція
x = linspace(0, 2.0 * pi, 25) # масив координат для побудови графіка інтерп-ої функції
yi_1 = fi_1(xi) # масиви значень інтерпольованих функцій для побудови графіків
yi_2 = fi_2(xi)
# малювання графіків
plt.plot(x, y, 'o', label='вихідні дані')
plt.plot(xi, f_exact(xi), '-', color='grey', linewidth=2.5, label='точна функція' )
plt.plot(xi, yi_1, '--', label='лінійна інтерполяція')
plt.plot(xi, yi_2, 'k:', label='кубічна інтерполяція')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0.0, 1.5)
plt.legend(loc='upper center')
plt.show()
```



Приклад інтерполяції функції, заданої таблично

```
from scipy.interpolate import interp1d
from numpy import arange
import matplotlib.pyplot as plt

xi = [-1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5]
yi = [9.8, 6.4, 7.0, 1.7, 17.3, 5.6, 10.8, 6.2, 27.5]

step = 0.01
xp = arange(xi[0], xi[-1], step)

f_linear = interp1d(xi, yi)
y_linear = f_linear(xp)
f_zero = interp1d(xi, yi, kind='zero')
y_zero = f_zero(xp)

f_slinear = interp1d(xi, yi, kind='slinear')
y_slinear = f_slinear(xp)

f_quadratic = interp1d(xi, yi, kind='quadratic')
y_quadratic = f_quadratic(xp)

f_cubic = interp1d(xi, yi, kind='cubic')
y_cubic = f_cubic(xp)
fig, axs = plt.subplots(1, 5, figsize=(20, 9), sharey=True)
axs[0].plot(xi, yi, 's', xp, y_linear, 'm')
axs[0].set_title('linear')
axs[1].plot(xi, yi, 's', xp, y_zero, 'm')
axs[1].set_title('zero')
axs[2].plot(xi, yi, 's', xp, y_slinear, 'm')
axs[2].set_title('slinear')
axs[3].plot(xi, yi, 's', xp, y_quadratic, 'm')
axs[3].set_title('quadratic')
axs[4].plot(xi, yi, 's', xp, y_cubic, 'm')
axs[4].set_title('cubic')
```



2.6.2. Чисельна інтерполяція функціями бібліотеки numpy.interp

Функція interp повертає одновимірну кусково-лінійну інтерполяцію функції за заданими дискретними точками даних (xp, fp).

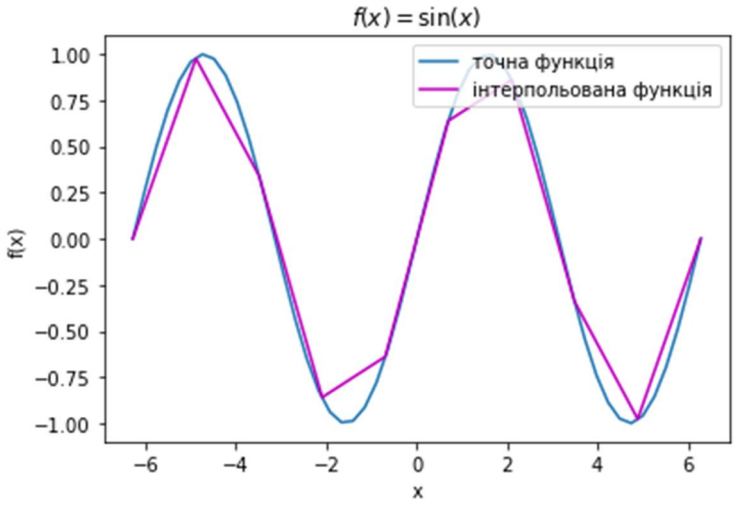
Синтаксис виклику функції такий:

fi = numpy.interp(x, xp, fp, left=None, right=None, period=None)

Детально про interp: https://numpy.org/doc/stable/reference/generated/numpy.interp.html

Приклад програми з використанням interp:

```
from numpy import interp
from numpy import sin, linspace, pi
import matplotlib.pyplot as plt

xi = linspace (-2 * pi, 2 * pi, 50)
yi = sin(xi)
xp = linspace (-2 * pi, 2 * pi, 10)
yp = interp(xp, xi, yi)
plt.title(r'$f(x)= \sin(x)$')
plt.ylabel('f(x)')
plt.xlabel('x')
plt.plot(xi, yi, label='точна функція')
plt.plot(xp, yp, 'm-', label='інтерпольована функція')
plt.legend(loc='upper right')
plt.show()
```



2.7. Інтерполяція засобами Mathematica

У системі комп’ютерної алгебри інтерполяція виконується функцією Interpolation. Ця функція повертає об'єкт InterpolatingFunction, який можна використовувати як будь-яку іншу функцію.

Основний синтаксис:

Interpolation[$\{f_1, f_2, \ldots\}$]

Інтерполяції значень функції, які відповідають значенням x=1,2,....

Interpolation [$\{\{x_1, f_1\}, \{x_2, f_2\}, \ldots\}$]

Інтерполяції значень функції f_i, відповідних значенням x_i.

Interpolation $[\{\{\{x_1, y_1, \ldots\}, f_1\}, \{\{x_2, y_2, \ldots\}, f_2\}, \ldots\}]$

Інтерполяції багатовимірних даних.

Interpolation $[\{\{\{x_1, \ldots\}, f_1, \mathrm{df}_1, \ldots\}, \ldots\}]$

Інтерполяцію, яка відтворює похідні, а також значення функції.

Interpolation[data, x]

Інтерполяцію даних у точці x

Інтерполяція працює шляхом підгонки поліноміальних кривих між послідовними точками даних. Ступінь поліноміальних кривих задається опцією InterpolationOrder. За замовчуванням - InterpolationOrder = 3.

Значення функції f_i, можуть бути дійсними чи комплексними числами або довільними символічними виразами. Функції f_i, можуть бути списками чи масивами довільної розмірності.

Аргументи функції x_i, y_i повинні бути дійсними числами.

Інтерполяція підтримує параметр Method. Можливі налаштування включають «Spline» для сплайн-інтерполяції та «Hermite» для інтерполяції Ерміта.

Детально про Interpolation: https://reference.wolfram.com/language/ref/Interpolation.html

Функція InterpolatingPolynomial створює поліном (аналітичний вираз), який проходить через вузлові точки.

InterpolatingPolynomial $[\{f_1, f_2, \ldots\}, x]$

будується інтерполюючий поліном за x, який відтворює значення функції f за послідовністю цілих значень 1,2,... x.

InterpolatingPolynomial $[\{\{x_1, f_1\}, \{x_2, f_2\}, \ldots\}, x]$

будується інтерполюючий поліном для значень функції f_i, що відповідають значенням x_i.

InterpolatingPolynomial $[\{\{\{x_1, f_1, \ldots\}, f_1\}, \{\{x_2, f_2, \ldots\}, f_2\}, \ldots\}, \{x, y, \ldots\}]$

будується багатовимірний інтерполюючий поліном за змінними x, y.

InterpolatingPolynomial$[\{\{\{x_1,\ldots\}, f_1, \mathrm{d}f_1,\ldots\},\ldots\},\{x,\ldots\}]$

будується інтерполюючий поліном, який відтворює похідні, а також значення функції.

В одновимірному випадку по n точках будується поліном степеня n-1. Інтерполяційний поліном повертається у формі Горнера.

Детально про InterpolatingPolynomial: https://reference.wolfram.com/language/ref/InterpolatingPolynomial.html?q=InterpolatingPolynomial

3. ДИФЕРЕНЦІЮВАННЯ

3.1 Методи чисельного диференціювання

Задача чисельного диференціювання полягає у знаходженні значень похідних функції $y = f(x)$ в заданих точках x_n у випадках, коли аналітичний вид функції $f(x)$ невідомий або функція $f(x)$ задана таблично: $y_i = f(x_i)$, де $x_i = x_0 + i\, h_i$, $i = 0, \pm 1, \pm 2, \ldots$. За точку x_0 доцільно вибрати ту точку, в якій обчислюється похідна.

Для отримання формул чисельного диференціювання функція $y = f(x)$ наближається інтерполяційним багаточленом $P_n(x)$, від якого розрахувується похідна $P'_n(x)$ в точці x_0, що визначає наближене значення шуканої похідної $f'(x_0) \approx P'_n(x_0)$. Найбільш зручним інтерполяційним багаточленом для чисельного диференціювання є поліном Ньютона. На його основі отримані формули різного порядку точності залежно від кількості заданих точок x_i:

Похідні першого порядку:

формула диференціювання вперед

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h} = \frac{y_1-y_0}{h}; \quad (1)$$

формула диференціювання назад

$$f'(x_0) \approx \frac{f(x_0)-f(x_0-h)}{h} = \frac{y_0-y_{-1}}{h}; \quad (2)$$

симетрична формула диференціювання

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h} = \frac{y_1-y_{-1}}{2h}. \quad (3)$$

$$f'(x_0) \approx \frac{-y_2+8y_1-8y_{-1}+y_{-2}}{12h}. \quad (4)$$

Погрішність формул (1) та (2) порядку $O(h)$, (3) - $O(h^2)$, а (4) - $O(h^4)$.

Похідні другого порядку:

$$f''(x_0) \approx \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} = \frac{y_1-2y_0+y_{-1}}{h^2}, \quad (5)$$

$$f''(x_0) \approx \frac{-y_2+16y_1-30y_0+16y_{-1}-y_{-2}}{12h^2}. \quad (6)$$

Погрішність формули (5) порядку $O(h)$, формули (6) - $O(h^3)$.

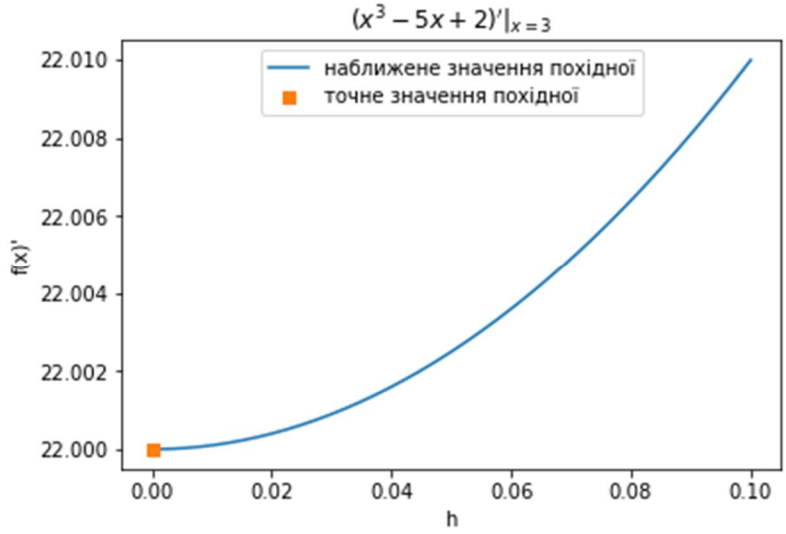
Величина h називається кроком чисельного диференціювання і вибирається залежно від поведінки функції $f(x)$ в околі точки x_0. Зауважимо, що вибір занадто малого кроку h може привести до значної похибки розрахунку, зумовленої похибками округлення на комп'ютері. Значення h краще вибирати залежно від величини x_0, наприклад $h = h_R |x_0|$, де h_R знаходиться у межах від 10^{-6} до 10^{-2} та підбирається залежно від поведінки функції $f(x)$ в околі точки x_0.

Запис формул (1)-(6) через y_i відповідає постановці задачі диференціювання з табличним заданням функції $y_i=f(x_i)$ в точках x_i. При цьому для крайніх точок x_i варто застосовувати формули диференціювання вперед або назад, а для всіх внутрішніх точок _ симетричні формули диференціювання.

Приклад програмної реалізації чисельного розрахунку пешої похідної за формулою (3)

```
def derivative_sym(f, x0, h):
   return (f(x0 + h) - f(x0 - h))/(2 * h)

f = lambda x: x**3 - 5 * x + 2
import matplotlib.pyplot as plt
```

```
from numpy import arange
exact = 22
hi = arange(1e-6, 0.1, 1e-6)
df = [derivative_sym(f, x0, h) for h in hi]

plt.title(r"$(x^{3} - 5x + 2)'|_{x = 3}$")
plt.plot(hi, df, label='наближене значення похідної')
plt.plot(0, exact, 's', label='точне значення похідної')
plt.legend(loc='upper center')
plt.ylabel("f(x)'")
plt.xlabel("h")
plt.show()
```



3.2. Диференціювання засобами бібліотек Python

3.2.1 Чисельне диференціювання scipy.misc.derivative

Функція derivative в модулі scipy.misc чисельно розраховує похідну від функції в заданій точці.

Синатксис запису функції:

scipy.misc.derivative(*func, x0, dx=1.0, n=1, args=(), order=3*)

Тут func – аналітично задана функція, x0 - точка, в якій похідна порядку n, dx – крок диференціювання (за замовчуванням 1), n - порядок похідної (за замовчуванням 1), arg – аргументи, orde – кількість кроків (непарне).

Детально про derivative:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html

3.2.2 Аналітичне та числове диференціювання sympy. diff

Для аналітичного диференціювання в бібліотеці sympy використовується функція diff(expr, variable). Першим аргументом expr вона приймає символьний вираз, а другим variable - змінну, по якій обчислюється похідна. Похідні другого і вищих порядків n обчислюються за правилом: diff(expr, variable, n). Також функція diff() дозволяє обчислювати змішані частинні похідні: diff(expr(var1, var2, …), var1, n1, var2, n2…).

Детально про derivative:https://www.tutorialspoint.com/sympy/sympy_derivative.htm

```
from sympy import symbols, diff
x = symbols('x')
f = x**3 - 13 * x **2 + 5
diff(f, x)
```

$$3x^2 - 26x$$

У функції diff () є еквівалент - оператор Derivative (), який повертає вираз типу 'sympy.core.function.Derivative'. Щоб обчислити вираз, створений цим оператором, потрібно використати метод doit ().

```
from sympy import *
x, y, z = symbols ( 'x y z')
f = x ** 2 + y ** 2 + z ** 2
f1 = Derivative (f, x)
f1.doit ()
```

$$2x$$

3.3. Диференціювання засобами Mathematica

Для обчислення похідних в символьній формі в Mathematica використовуються функції Derivative та D. Перша з них призначена для диференціювання функцій, друга – для диференціювання виразів (якщо після імені функції в квадратних дужках стоїть аргумент, то це вираз).

Функція Derivative[n][f] обчислює n–ту похідну функції f одного аргументу. Також можна писати звичайне позначення зі штрихом, наприклад f′, f″…, що аналогічне Derivative[1][f] та Derivative[2][f].

Функція D використовується для диференціювання виразів. У форматі D[expr,x] вона обчислює частинну похідну виразу expr за змінною x. Якщо x єдина змінна, то обчислюється звичайна похідна функції/виразу однієї змінної. У форматі D[expr,{x1,n1},{x2,n2}…] обчислюється частинна похідна виразу expr n1-го порядку за змінною x1, n2-го порядку за змінною x2 і т.д.. Запис D[f,x] також можна замінити на ¶xf.

Детально про Derivative та D: https://reference.wolfram.com/language/ref/Derivative.html, https://reference.wolfram.com/language/ref/D.html

4. ПОШУК КОРЕНІВ НЕЛІНІЙНИХ РІВНЯНЬ

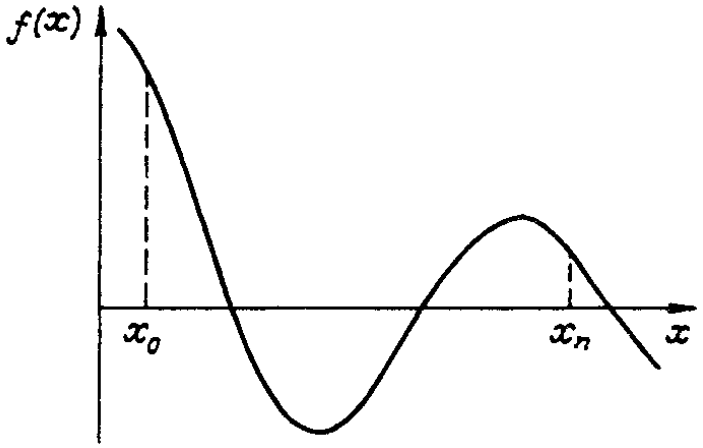
Ітераційні методи обчислення ізольованого (відокремленого від інших) кореня рівняння f(x) = 0, як правило, вимагають задання області, що містить цей корінь, та алгоритму знаходження чергового наближення x_{n+1} за вже наявними x_n,.., x_{n-k}.

Широко використовувані способи відділення коренів - графічний та табличний - базуються на властивостях гладкості функції; у випадку ж, коли f(x) є алгебраїчним поліномом ступеня n, існують аналітичні підходи.

Якщо функція f(x) - неперервна, то корінь належить будь-якому відрізку, на кінцях якого ця функція має значення різних знаків. Розділяючи відрізок навпіл, отримуємо універсальний метод обчислення кореня (метод дихотомії). Цей підхід не вимагає знання початкового наближення. Якщо воно відоме, то для гладких функцій використовують ефективніші ітераційні методи.

Особлива увага в теорії розв'язання нелінійних рівнянь приділяється методам із надлінійною швидкістю збіжності. При практичних розрахунках зазвичай застосовують способи з квадратичною збіжністю, оскільки ітераційні процеси вищого порядку часто вимагають значного збільшення обчислювальних витрат.

Отже, пошук одновимірних коренів (коренів функцій одного аргументу) є досить простим завданням оскільки одновимірний корінь можна легко локалізувати. Отже, якщо знак функції на кінцях деякого відрізка різний, то на цьому відрізку знаходиться один корінь (або непарне число коренів). Далі будемо вважаит, що корінь вже локалізований на деякому відрізку, і потрібно визначити з заданою точністю його положення.



В реальних ситуаціях, як правило, задача полягає у знаходженні всіх коренів на деякому відрізку, яка розв’язується табуляцією функції на цьому відрізку з досить дрібним кроком, як методом попередньої локалізації коренів, і подальшим уточненням чисельних значень всіх коренів. Для надійної локалізалізації коренів слід зменшити крок (наприклад, на порядок) і перевірити чи не збільшилась кількість знайдених коренів.

4.1 Метод дихотомії

Нехай відрізок [a, b] локалізує корінь рівняння f(x)=0.

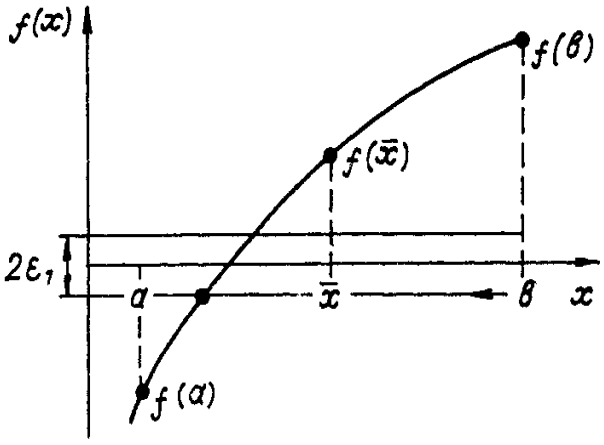
Алгоритм

Обчислюємо $y_a = f(a), y_b = f(b)$ і перевіряємо знаки y_a і y_b.

Якщо знаки y_a і y_b різні, то повторюємо такі дії:

Обчислюємо $c = (a+b)/2, y_c = f(c)$.

Якщо знаки y_a і y_c однакові, то корінь лежить на відрізку [c, b]. Тоді оголошуємо точку с новою точкою a: $a = c, y_a = y_c$.



В іншому випадку корінь лежить на відрізку [a, c]. Тоді оголошуємо точку с новою точкою b: $b = c, y_b = y_c$. Якщо довжина нового відрізка $|a - b|$ менше замовленої точності визначення кореня, то завершуємо роботу і повертаємо в якості відповіді (a+b)/2.

Метод дихотомії збігається лінійно, тобто число знайдених вірних знаків зростає лінійно з кількістю операцій n. Якщо швидкість знаходження кореня не критична,

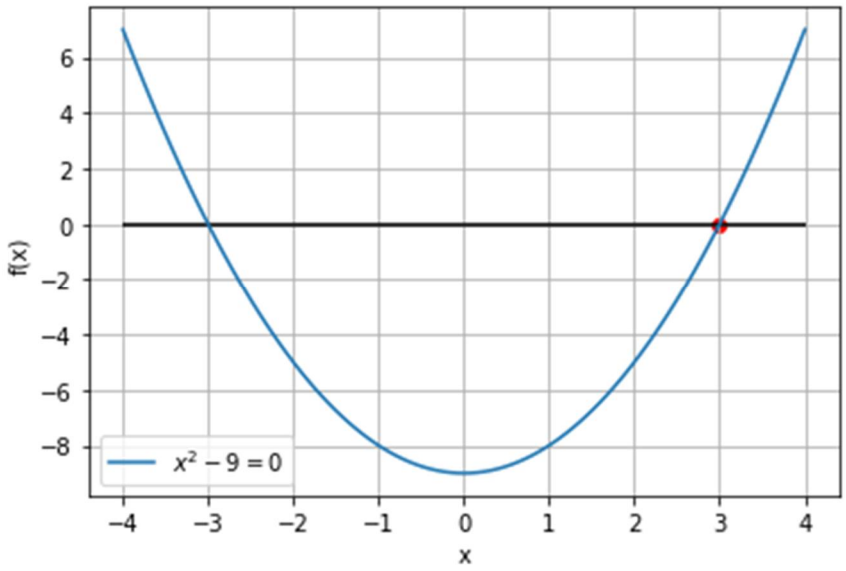
то цей метод цілком можна застосовувати.

Приклад програмної реалізації методу дихотомії

```
def dichotomy(f, a, b, eps):
    ya, yb = f(a), f(b)

    assert ya * yb <= 0, 'Інтервал [a, b] не містить кореня рівняння'

    if ya==0:
        return a, 0
    elif yb==0:
        return b, 0

    while abs(a-b) > eps:
        c = (a + b)/2
        yc = f(c)
        if yc == 0:
            return c
        elif ya * yc < 0:
            b, yb = c; yc
        else:
            a, ya = c, yc
    return (a + b)/2

# Знаходження кореня рівняння x^2 – 9 = 0 в інтервалі [2, 4]
from numpy import linspace
from scipy.optimize import root_scalar
import matplotlib.pyplot as plt

def f(x): return x**2 - 9

a, b = 2, 4
eps = 1e-3

x = linspace(-4, 4, 101)
y = f(x)
x0 = dichotomy(f, a, b, eps)
print(f'x0 = {x0}')

plt.plot(x, y, label=r'$x^2 - 9=0$')
plt.scatter(x0, 0, c='r')
plt.hlines(0, -4, 4)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend(loc=0)
plt.grid(True)
```

```
plt.show()
```

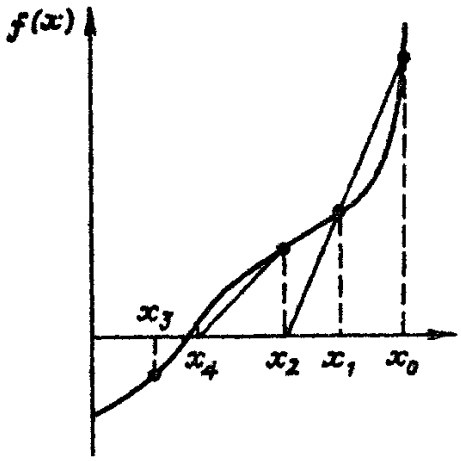
```
>>x0 = 2.9998779296875
```



4.2 Метод січних без перевірки знаків

По двом точкам (x_{n-1}, y_{n-1}) і (x_n, y_n), які є попередніми наближеннями до кореня, будуються пряма і знаходиться її корінь:

$$x_{n+1} = \frac{x_{n-1}y_n - x_n y_{n-1}}{y_n - y_{n-1}}, \qquad y_{n+1} = f(x_{n+1}).$$



Точка (x_{n+1}, y_{n+1}) є наступним наближенням до кореня. При цьому умова локалізації кореня не перевіряється, тобто не перевіряються знаки y_{n-1}, y_n, y_{n+1}. Наступна пряма будується по точках (x_n, y_n) та (x_{n+1}, y_{n+1}). Якщо корінь знайдений з похибкою ε, то лінійна інтерполяція усуває члени O(ε), а нова похибка буде порядка $O(\varepsilon^2)$, тобто збіжність квадратична (число вірних знаків на кожній ітерації подвоюється). На жаль, цей метод вкрай ненадійний. У деяких випадках він дійсно дає квадратичну збіжність, але він також може збігатися повільно (повільніше ділення навпіл), а може взагалі не зійтися.

Приклад програмної реалізації методу січних без перевірки знаків

```
def linearIterpolationRoot(f, a, b, eps):
    ya = f(a)
    yb = f(b)
    while abs(a - b) > eps:
        c = (a * yb - b * ya) / (yb - ya)
        a, b = b, c
        ya, yb = yb, f(c)
    return b
```

```
# Знаходження кореня рівняння x^2 – 9 = 0 в інтервалі [2, 4]
def f(x): return x**2 - 9
a, b = 2, 4
eps = 1e-3
x0 = linearIterpolation(f, a, b, eps)
print(f'x0 = {x0}')
```

```
x0 = 2.9999999868869875
```

4.3 Метод січних з перевіркою знаків

На відміну від методу січних без перевірки знаків в цьому методі необхідно слідкувати за локалізацією корня, тобто y_{n-1} і y_n повинні мати різні знаки, тому точка x_{n+1} повинна знаходитися між x_{n-1} і x_n, а в якості пари точок наступної ітерації, у залежності від того, на якому відрізку локалізований корінь, вибираються або x_{n-1} і x_{n+1}, або x_{n+1} і x_n. На кожному кроці ітерації перевіряється умова $|x_n - x_{n-1}| < e$, при виконанні якої процес уточнення кореня зупиняється. Використання точки x_{n-1} замість x_n не прискорює збіжність, проте воно гарантує локалізацію корня і збіжність. Однак, як і в попередньому методі, збіжність може виявитися як квадратичною, так і вкрай повільною (повільніше ділення навпіл).

4.4 Зворотна квадратична інтерполяція

Цей метод, на відміну від вже описаних двоточкових, є триточковим і призначений для з таких типів коренів, для яких непридатна лінійна інтерполяція. По трьох точках (a, y_a) (b, y_b) (c, y_c) будується зворотна парабола (поліном другого порядку $x(y)$). Після цього шукається її корінь, тобто x_0, відповідний $y=0$:

$$x = a\frac{(y-y_b)(y-y_c)}{(y_a-y_b)(y_a-y_c)} + b\frac{(y-y_a)(y-y_c)}{(y_b-y_a)(y_b-y_c)} + c\frac{(y-y_a)(y-y_b)}{(y_c-y_a)(y_c-y_a)}.$$

Отримавши наближення до корня x_0 далі мона по-різному вибрати три нові точки для побудови наступної параболи. Найпростіший варіант - додати x_0 і відкинути найбільш старе наближення до кореня. Інший варіант - вибрати ті три точки, які утворюють найкоротший відрізок. На жаль, цей метод не гарантує збіжність не гірше лінійної.

4.5 Метод Ньютона

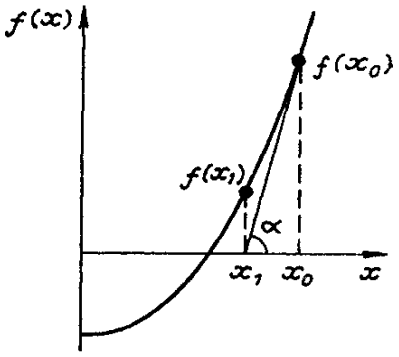
Метод Ньютона взагалі одноточковий. Він полягає у тому, що з точки (x_{n-1}, y_{n-1}) попереднього наближення будується дотична до функції і знаходиться її корінь:

$$x_n = x_{n-1} - \frac{y_{n-1}}{y'_{n-1}},$$

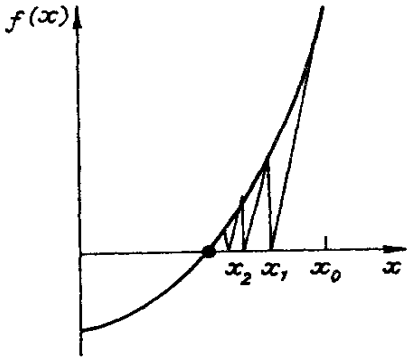
який є наступним наближенням (x_n, y_n) до кореня функції. При досягненні заданої точності $|x_n - x_{n-1}| < e$ ітераційна процедура зупиняється.

Метод Ньютона можна використовувати для уточнення не лише дійсних коренів, але й коренів в області комплексних значень x, що необхідно при розв’язанні багатьох

теоретичних задач. В цьому випадку початкове наближення до кореня x_0 необхідно обирати також комплексним.



Метод Ньютона.



Модифікований метод Ньютона.

Теоретично збіжність методу повинна бути квадратичною. Однак він може сходитися вкрай повільно (збіжність повільніше лінійної), а може і зовсім не зійтися. Крім того, недоліком методу є необхідність обчислення на кожній ітерації як лівої частини рівняння, так й її похідної. Оскільки нерідко доводиться шукати корні функцій, для яких неможливо виписати явно аналітичний вираз, то це ускладнює застосування методу. Можна, трохи зменшивши швидкість збіжності, обмежитися обчисленням похідної тільки першої ітерації, а на наступних ітераціях обчислювати лише значення f(x), не змінюючи похідної f '(x). Це алгоритм так званого модифікованого методу Ньютона.

У більшості систем аналітичних обчислень, у тому числі і в Mathematica, вбудована функція чисельного пошуку одновимірних коренів за замовчуванням використовує саме метод Ньютона. Досвід використання цих функцій показує, що цей метод не достатньо надійний.

4.6 Адаптований метод Брендта

Адаптований метод Брендта є спрощеним варіантом методу Брендта і є двокроковим. З вихідної пари точок, що локалізують корінь, за два кроки будуються ще дві точки, які також локалізують корінь. Ідея полягає в тому, що потрібно побудувати не одну, а дві близькі до кореня точки, як знаходяться з різних боків від нього. Такий підхід значно прискорює збіжність. Крім того, в методі використовуються перевірки, які гарантують збіжність не гірше лінійної.

Алгоритм

На першому кроці здійснюється лінійна інтерполяція за двома початковими точками a і b, які локалізують корінь. Це визначає першу з двох нових точок

$$c = \frac{ay_b - by_a}{y_b - y_a}, \qquad y_c = f(c).$$

Друга з точок будується в залежності від знаку y_c.

Якщо знаки в y_c і y_a однакові,

то проводиться лінійна інтерполяція по точках a і c, що визначає другу точку

$$d = \frac{ay_c - cy_a}{y_c - y_a}.$$

Далі проходять наступні перевірки:

По-перше, d не повинно бути більше $(c+b) / 2$. Якщо $d > (c+b) / 2$, то покладається $d = (c+b) / 2$, що гарантує збіжність не гірше лінійної - інтервал, в якому локалізований корінь зменшується не менше ніж в 2 рази.

По-друге, d не повинно бути менше, ніж $c+\varepsilon$, де ε - замовлена точність. Якщо $d < c+\varepsilon$, тоді покладається $d = c+\varepsilon$. Це гарантує, що точки c і d лежать по різні боки від кореня і локалізують його оскільки корінь лежить правіше від с.

Далі обчислюється $y_d = f(d)$.

Якщо знаки y_d і y_c протилежні, тобто корінь локалізований на відрізку $[c, d]$, тоді $a = c, b = d, y_a = y_c, y_b = y_d$. Оскільки точки c і d лежать набагато ближче до кореня, ніж вихідні точки a і b, то збіжність в цьому випадку квадратична. Як правило, зі зменшенням відрізка цей випадок зустрічається все частіше, тобто в середньому збіжність методу, як правило, квадратична. В іншому випадку, коли знаки y_d і y_c однакові, корінь локалізований на відрізку $[d, b]$, тоді $a = b, b = d, y_a = y_b, y_b = y_d$. У цьому випадку збіжність не гірше лінійної.

Якщо знаки y_c і y_a різні,

то виконується все те ж саме, з точністю до заміни $a \leftrightarrow b$:

$$d = \frac{by_c - cy_b}{y_c - y_a}.$$

Якщо $d < (c+a)/2$, то $d=(c+a)/2$.

Якщо $d > c - \varepsilon$, то $d = c - \varepsilon$.

Обчислюється $y_d = f(d)$.

Якщо знаки y_c і y_d різні, то корінь локалізований на відрізку [c, d], тобто $a = c, b = d, y_a = y_c, y_b = y_d$, інакше, коли знаки y_c і y_d однакові, він локалізований на відрізку [a, d], $b = d$, $y_b = y_d$.

Після цього все повторюється, поки не буде досягнута замовлена точність.

Адаптований метод Брендта, як правило, забезпечує квадратичну збіжність, але в будь-якому випадку збіжність не гірше лінійної. Цей метод у середньому швидший за більшість інших квадратичних методів. Тому, якщо швидкість знаходження кореня критична, його можна застосовувати.

4.7. Розв'язання нелінійних рівнянь засобами бібліотек Python

4.7.1 Чисельне розв'язання нелінійних рівнянь засобами бібліотеки scipy

У модулі optimize пакету scipy представлені засоби для розв'язання нелінійних рівнянь та їх систем. Для наближеного розв'язання задачі знаходження розв'язку нелінійного рівняння f(x)=0 на інтервалі [a, b], на якому функція f(x) змінює знак (f(a)f(b)<0), можна використовувати різні методи, реалізовані в функції

root_scalar(f[, args, method, bracket, …]),

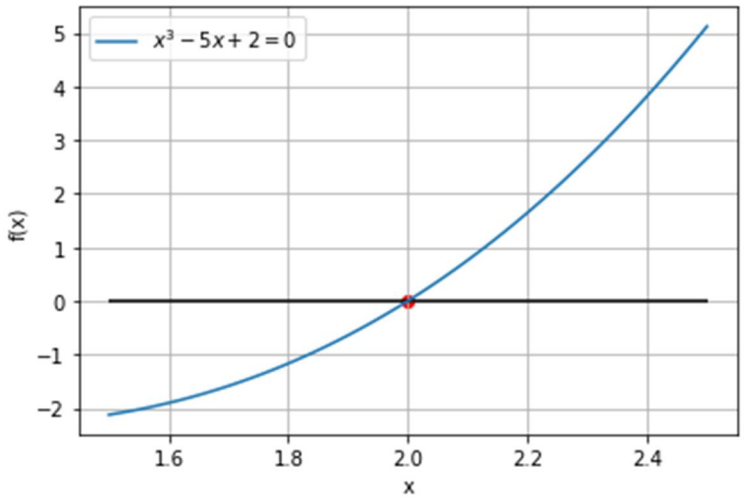
де аргумент method визначає застосований чисельний метод знаходження кореня: 'bisect', 'brenth', 'ridder', 'toms748' , 'newton', 'secant' , 'halley'.

Детальніше про scipy.optimize.root_scalar: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root_scalar.html#scipy.optimize.root_scalar

Приклад знаходження розв'язку рівняння $x^3 - 5x + 2 = 0$ *на інтервалі [1.5, 2.5] методом Брендта*

```
from numpy import linspace
from scipy.optimize import root_scalar
import matplotlib.pyplot as plt

def f(x): return x**3 - 5 * x + 2

x = linspace(1.5, 2.5, 101)
y = f(x)

x0 = root_scalar(f, bracket=[1.5, 2.5], method='brentq').root
print(f'x0 = {x0}')
plt.plot(x, y, label=r'$x^3 - 5 x + 2=0$')
plt.scatter(x0, 0, c='r')
plt.hlines(0, 1.5, 2.5)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend(loc=0)
plt.grid(True)
plt.show()
```

```
>>x0 = 1.9999999999999831
```



Для знаходження всіх розв'язків нелінійного рівняння можна застосовувати функцію

root(fun, x0[, args, method, jac, tol, …])

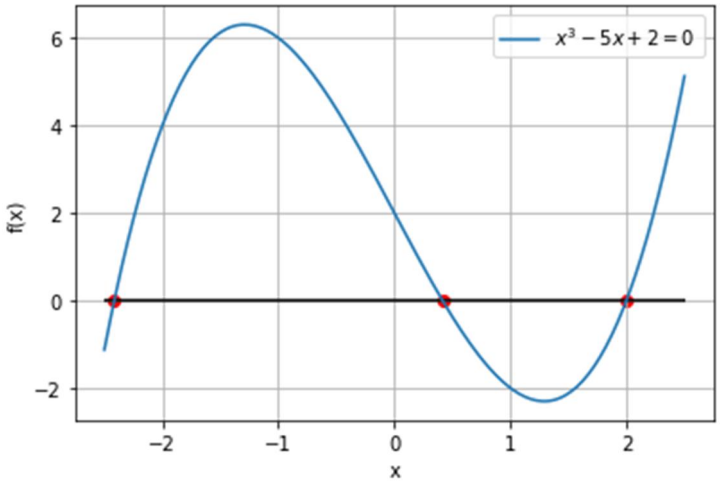
Детальніше про scipy.optimize.root: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html#scipy.optimize.root

Приклад знаходження всіх розв'язків рівняння $x^3 - 5x + 2 = 0$

```
from numpy import linspace
from scipy.optimize import root
import matplotlib.pyplot as plt

def f(x): return x**3 - 5 * x + 2
```

```
x = linspace(-2.5, 2.5, 201)
y = f(x)

x0 = root(f, [-2.5, 0, 2.2], method='hybr').x
print(f'x0 = {x0}')

plt.plot(x, y, label=r'$x^3 - 5 x + 2=0$')
plt.scatter(x0, [0, 0, 0], c='r')
plt.hlines(0, -2.5, 2.5)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend(loc=0)
plt.grid(True)
plt.show()
```

```
>>x0 = [-2.41421356, 0.41421356, 2.]
```



Для розв’язання систем нелінійних рівнянь використовується функція

fsolve(func, x0[, args, fprime, …]),

в якій потрібно задати систему рівнянь та початкове наближення до кореня.

Детальніше про scipy.optimize.fsolve: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html#scipy.optimize.fsolve

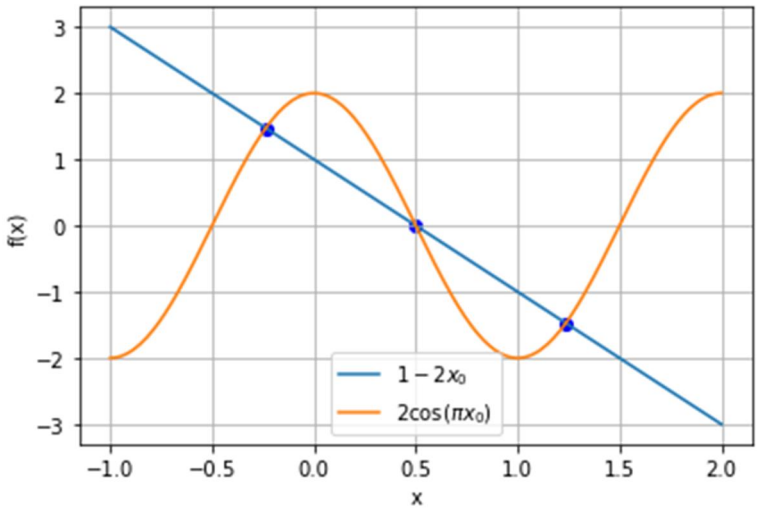
Приклад розв’язання системи нелінійних рівнянь

$$\begin{cases} 1 - 2x_0 - x_1 = 0 \\ x_1 - 2\cos(\pi x_0) = 0 \end{cases}$$

```
from numpy import linspace, cos, pi
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
def F(x):
    return [1 - 2*x[0] - x[1], x[1] - 2 * cos(pi*x[0])]

x= linspace(-1, 2, 201)
```

```
y1= 1 - 2*x
y2= 2*cos(pi*x)

x0 = fsolve(F, [-0.5, 0.])
x1 = fsolve(F, [0.5, 0.])
x3 = fsolve(F, [1.5, 0.])
print(f'x: {x0}, {x1}, {x3}')

plt.plot(x, y1, label=r'$1 - 2x_{0}$')
plt.plot(x, y2, label=r'$2\cos(\pi x_{0})$')
plt.scatter(x0[0], x0[1], c='b')
plt.scatter(x1[0], x1[1], c='b')
plt.scatter(x3[0], x3[1], c='b')
plt.legend(loc=0)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.show()
```

```
>>x: [-0.23648445 1.4729689 ], [0.5 0. ], [ 1.23648445 -1.4729689 ]
```



4.7.2 Розв'язання нелінійних рівнянь засобами бібліотеки sympy

Для символьного розв'язання алгебраїчних рівнянь використовується функція

solveset(equation,variable=None[,domain=S.Complexes]),

де equation може бути записано у формі рівняння або бути виразом, який вважається рівним нулю. Другий аргумент variable задає ім'я змінної, третій domain – визначає дійсну або комплексну область, в якій шукається корінь. У результаті функція повертає множину розв'язків.

Для знаходження тільки дійсних розв'язків застосовується функція

real_roots().

Функція

roots()

повертає символьний вираз коренів полінома однієї змінної. Результат повертається у формі словаря з парами {корінь: кратність}.

У модулі Solveset нелінійна система рівнянь розв’язується за допомогою nonlinsolve().

Для отримання аналітичних розв’язків алгебраїчних рівнянь і систем призначена функція

solve(equation, variable).

Першим аргументом вона також приймає рівняння (або список рівнянь) у формі Eq() або вираз (список виразів), який рівний нулю. Другий аргумент - ім'я шуканої змінної. Аргумент variable необов'язковий, але його краще завжди вказувати. Форма результату функції solve() може бути списком, словником, множиною і її можна задати опціями. Наприклад, опція dict=False скасовує вивід результату у вигляді словаря і включає у вигляді списку.

Детальніше про символьне розв’язання рівнянь в sympy: https://docs.sympy.org/latest/modules/solvers/solvers.html, https://docs.sympy.org/latest/tutorial/solvers.html

4.8. Розв’язання нелінійних рівнянь засобами Mathematica

Для символьного розв'язання поліноміальних рівнянь (або системи рівнянь) в Mathematica використовується функція

Solve[expr, var]

Перший аргумент expr – це рівняння, яке потрібно розв’язати. Другий аргумент var – змінна, відносно якої розв’язується рівняння. Результат повертається у формі списку правил підстановки.

Детальніше про Solve: https://reference.wolfram.com/language/ref/Solve.html?q=Solve

Функція

Roots[lhs==rhs,var]

також дозволяє знаходити корені поліноміальних рівнянь.

Детальніше про Roots: https://reference.wolfram.com/language/ref/Roots.html?q=Roots

Функція

NSolve[expr, var]

знаходить наближено всі корені поліноміального рівняння.

Детальніше про NSolve: https://reference.wolfram.com/language/ref/NSolve.html?q=NSolve

Для знаходження кореня трансцедентного рівняння f(x)=0 в околі точки x0 використовується функція

FindRoot[f, {x, x0}]

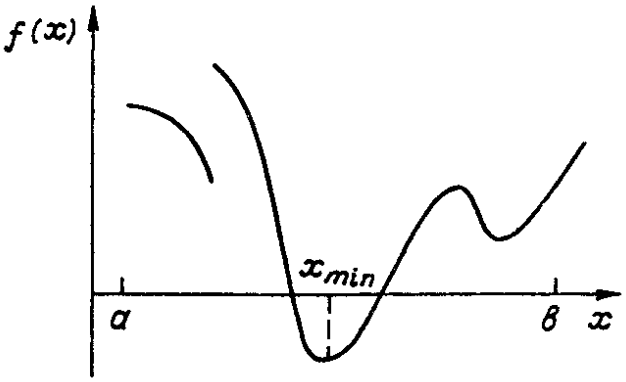
З замовчуванням використовується метод Ньютона. Якщо початкове наближення x0 до кореня задати комплексним, то функція FindRoot поверне комплексний корінь (найближчий до початкового наближення).

Функція FindRoot має декілька опційних аргументів. Опція AccuracyGoal->m визначає точність (кількість цифр m) результату. Опція WorkingPrecision->n визначає точність (кількість цифр n) внутрішніх обчислень. Опція MaxIterations->p задає кількість p ітерацій.

Детальніше про FindRoot: https://reference.wolfram.com/language/ref/FindRoot.html?q=FindRoot

5. ПОШУК МІНІМУМІВ

Розглянемо задачу пошуку мінімуму функції однієї змінної f(x) на інтервалі [a, b], де функція може мати декілька локальних мінімумів, серед яких є глобальний. У загальному випадку функція f(x) може бути як неперервною, так і розривною в області пошуку мінімуму. Щоб перейти до пошуку максимумів функції f(x), достатньо змінити знак функції на протилежний та діяти за алгоритмами відшукання мінімумів.Одновимірний мінімум локалізується трьома точками: якщо точка x лежіт між точками *a* і *b*, то умови f(a) > f(c) та f(b) > f(c) означають наявність мінімуму (або декількох мінімумів) на відрізку [*a, b*].



Типові методи пошуку одновимірних мінімумів припускають, що мінімум вже локалізований на деякому відрізку. Задача початкової локалізації мінімуму вирішується просто: встановлюється напрямок спадання функції та область, де спадання змінюється зростанням. Для цього спочатку в початковій точці x_0, обчислюється значення функції $f(x_0)=y_0$, далі в точці $x_1=x_0+d$ зміщеній на величину d відносно x_0 обчислюється $f(x_1)=y_1$ і визначається напрямок спадання функції. Тоді для знаходження точки початку зростання достатньо обчислювати функцію в усе більш віддалених точках. Доцільно вибирати ці точкі так, щоб $(x_{n+1}-x_n) = (1 - r)/r$, де $r = (3 - \sqrt{5}) / 2 \approx 1.618$ - золотий переріз. Якщо чергове y_{n+1} виявиться більше y_n, то мінімум локалізований точками x_{n-1}, x_n, x_{n+1}. Цей спосіб дозволяє досить швидко локалізувати мінімум. З іншого боку, цей метод не гарантує втрати деякого локального мінімуму. Тому, якщо необхідно знайти всі мінімуми на деякому відрізку, то доречніше просто протабулювати функцію на цьому відрізку з досить дрібним кроком.

Відмітимо також, що точність, з якою можна знайти мінімум функції, користуючись тільки значеннями функції, набагато менша, ніж точність, з якою можна знайти корінь. Кількість вірних знаків, які можна отримати при пошуках кореня порядка кількості знаків, який забезпечує тип «double». Кількість вірних знаків при пошуку мінімуму приблизно вдвічі менше. Це зумовлено тим, що функція $f(x)$ в околиці мінімуму x_0 дорівнює $f(x) \cong f(x_0) + 0 + \frac{f''(x_0)(x-x_0)^2}{2}$..., тобто різницю в значеннях функції можна помітити лише при зміщенях $(x - x_0)$. Якщо потрібна більш висока точність при знаходженні мінімуму, то необіхдно шукати корені рівняння $f'(x_0) = 0$.

5. 1 Метод золотого перерізу

Метод золотого перерізу є своєрідним аналогом методу дихотомії

Алгоритм

Нехай задано початкові дві точки a, $y_a = f(a)$ та c, $y_c = f(c)$, які визначають інтервал, в якому локалізований мінімум функції. При цьому ми не припускаємо, що $a<c$ - може бути і навпаки. Третя точка розташовується так $b=a+r(c-a)$, $y_b = f(b)$, $r = (3 - \sqrt{5}) / 2$, що вона лежить ближче до a, ніж до c.

Будується нова точка $d=c+r(a-c)$, $y_d = f(d)$.

Якщо $y_d < y_b$,

то мінімум локалізований на відрізку [*b, c*] точками *b,d,c*. Тому далі покладаємо $a = b$, $y_a = y_b$, $b = d$, $y_b = y_d$. Умова про те, що *b* лежить ближче до *a*, ніж до *c*, при цьому виконується.

Якщо $y_d > y_b$,

то мінімум локалізований на відрізку [*a,d*] точками *a,b,d*. Тому далі покладаємо $c = a$, $y_c = y_a$, $a = d$, $y_a = y_d$.

Після цього все повторюється, поки не буде досягнута замовлена точність.

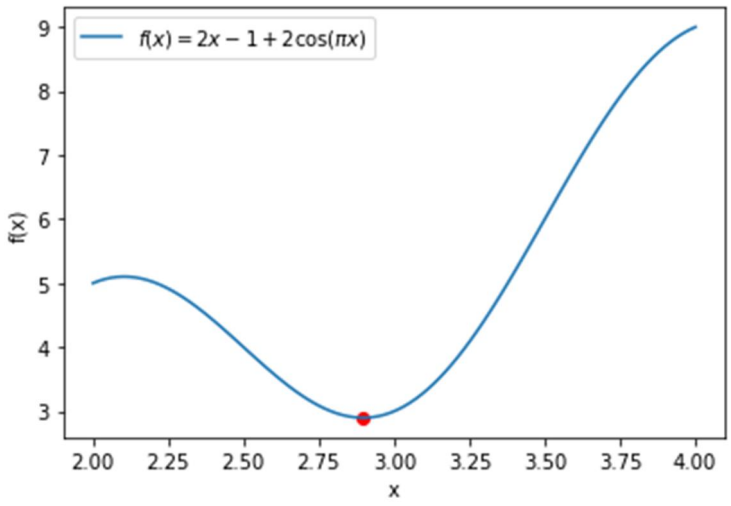
Збіжність методу лінійна, він не складніший за метод поділу навпіл, тому якщо швидкість пошуку мінімуму не критична, то його цілком можна застосовувати.

Приклад програмної реалізації методу золотого перерізу

```
def secant(f, x0, x1, eps):
    r = (3 - 5**0.5) / 2
    a, c = x0, x1
    ya, yc = f(a), f(c)
    b = a + r * (c - a)
    yb = f(b)
    while abs(c - a) > eps:
        d = c + r * (a - c)
        yd = f(d)
        if yd == yb:
            d += eps
            yd = f(d)
        if yd < yb:
            a, b = b, d
            ya, yb = yb, yd
        elif yd > yb:
            c, a = a, d
            yc, ya = ya, yd
    if yb < yd:
        return b
    else:
        return d

# Знаходженнямінімуму функції f(x)= 2 x - 1 + 2 cos(pi x) в інтервалі [2, 4]
from numpy import linspace
from math import cos, pi
import matplotlib.pyplot as plt
def f(x): return (2 * x - 1 + 2 * cos(pi * x))

a, c = 2, 4
eps = 1e-3

xmin = secant(f, a, c, eps)
print(f'xmin = {xmin}')
```

```
xi = linspace(a, c, 100)
fi = [f(x) for x in xi]

plt.plot(xi, fi, label=r'$f(x) = 2x - 1 + 2 \cos(\pi x)$')
plt.scatter(xmin, f(xmin), c='r')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend(loc=0)
plt.show()
```

```
>>xmin = 2.8969544723361738
```



5.2 Адаптований метод Брендта

Адаптований метод Брендта - це чотирьохточковий двокроковий метод. На кожній ітерації знаходяться дві нові точки, які розташовані все ближче до мінімуму і разом з попередніми точками локалізують його. Збіжність методу не гірше, ніж лінійна і в ідеальному випадку може бути квадратичною.

Алгоритм

На початковому етапі за відомими двома точками *a* та *d*, які локалізують мінімум функції, будуються ще дві точки золотого перетину $b=a+r(d-a)$ та $c=d+r(a-d)$, $r = (3 - \sqrt{5}) / 2$. Отже, отримується впорядкований список точок *a, b, c, d* і значення функції в цих точках y_a, y_b, y_c, y_d, причому $y_a > y_b$ і $y_c < y_d$.

Далі через три точки *a, b, d,* будується парабола і знаходиться її мінімум *e*. Аналогічно, будується парабола по точках *a, c, d* і шукається її мінімум *f*. Для цього використовується формула параболи, що проходить через точки x_1, x_2 *і* x_3

$$y = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$

звідки отримується координата її мінімуму

$$x_0 = \frac{1}{2} \frac{x_1^2(y_2 - y_3) + x_2^2(y_3 - y_1) + x_3^2(y_1 - y_2)}{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}$$

Після знаходження точок *e* та *f*, здійснюється їх перевірка на прийнятність.

На положення точки *e* накладаються умови: якщо $e < (a + b)/2$, то $e = (a + b)/2$, і якщо $e > (3c + d)/4$, то $e = (3c + d)/4$.

На положення точки *f* накладаються умови: якщо $f < (a+3b)/4$, то $f = (a+3b)/4$, і якщо $f > (c+d)/2$, то $f = (c+d)/2$.

Далі перевіряється умова $|f-e|>\epsilon$ чи $|f-e|<\epsilon$, де ϵ - задана точність знаходження мінімуму. Якщо $|f-e|<\epsilon$, то вважаємо $f = e + \epsilon$ або $e = f+ \epsilon$.

Після цього точки *a, b, c, d, e, f* впорядковуються у порядку зростання. В отриманому списку залишаються такі чотири точки *a, b, c, d,* в яких для функції виконуються нерівноситі $y_a > y_b$ та $y_d > y_c$.

Після цього все повторюється, поки не буде досягнута замовлена точність ϵ.

В ідеальному випадку збіжність виявляється квадратичною, і в будь-якому випадку не гірше лінійної. Якщо швидкість знаходження одновимірного мінімуму критична, то має сенс застосовувати цей метод.

5.3. Знаходження мінімумів засобами бібліотек Python

5.3.1 Чисельне знаходження мінімумів засобами бібліотеки scipy.optimize

В бібліотеці scipy.optimize для знаходження локалізованого мімімума функції однієї змінної передбачена функція

minimize_scalar(fun, bracket=None, bounds=None, args=(), method='brent', tol=None, opti
ons=None)

Опція method задає метод (Brent, Bounded, Golden) чисельного розрахунку мінімуму. За замовчуванням встановлений метод Брендта.

Детальніше про minimize_scalar: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html#scipy.optimize.minimize_scalar

Функція

minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=Non
e, constraints=(), tol=None, callback=None, options=None)

дозволяє знайти мінімум функції однієї або більше змінних. Тут опційно передбачено велика різноманітність методів розрахунку.

Детальніше про minimize: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

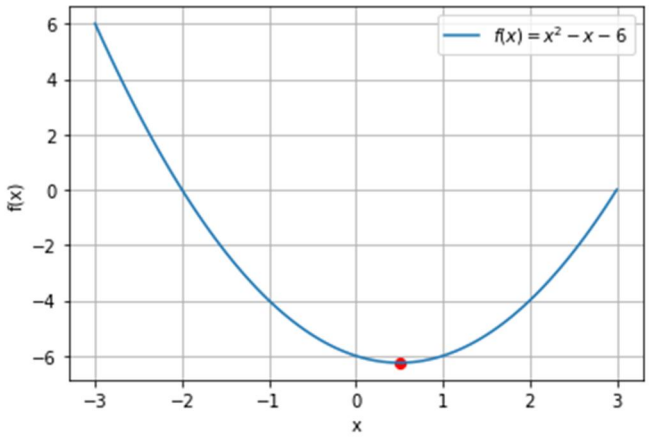
В scipy.optimize також представлені різноманітні багатофакторні методи загального призначення, важливі для задач опимізації, з якими можна детально ознайомитися за посиланням https://docs.scipy.org/doc/scipy/reference/optimize.html

Приклад знаходження мінімуму функції $f(x) = x^2 - x - 6$ *в інтервалі [-3, 3]*

```
from numpy import linspace
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt

f =lambda x: x**2 - x - 6
x = linspace(-3., 3., 101)
xmin = minimize_scalar(f, bracket=[-3, 3]).x
print('xmin = ', xmin)
```

```
plt.plot(x, y, label=r'$f(x) = x^2 - x - 6$')
plt.scatter(xmin, f(xmin), c='r')
plt.legend(loc=0)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.show()
```

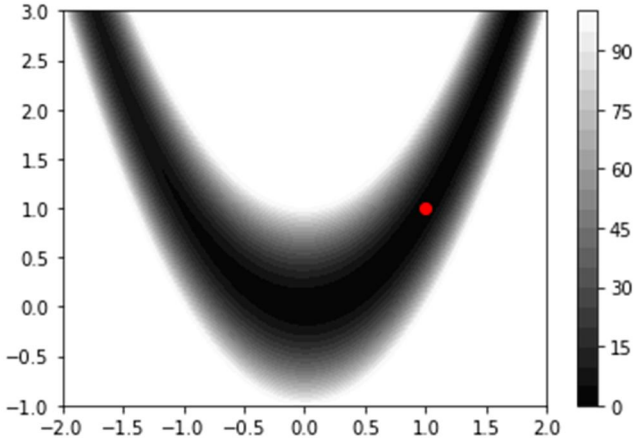
```
>> xmin = 0.4999999851800009
```



Приклад знаходження мінімуму функції $f(x) = 100\,(x_1 - x_0^2)^2 + (1 - x_0)^2$

```
from numpy import linspace, meshgrid
from scipy.optimize import fmin
import matplotlib.pyplot as plt

def f(x):
    return 100*(x[1] - x[0]**2)**2 + (1 - x[0])**2

x = linspace(-2., 2., 101)
y = linspace(-1., 3., 101)
X, Y = meshgrid(x, y)
z = f([X, Y])
v = linspace(0, 100, 21)

xmin = fmin(f, [0, 0])
print('xmin: ', xmin)

plt.contourf(x, y, z, v, cmap=plt.cm.gray)
plt.colorbar()
plt.scatter(xmin[0], xmin[1], c='r')
plt. show()
```

```
Optimization terminated successfully.
Current function value: 0.000000
Iterations: 79
Function evaluations: 146
xmin: [1.00000439 1.00001064]
```



5.4. Знаходження мінімумів засобами Mathematica

У системі Mathematica для пошуку мінімуму (або максимуму) функції, заданої аналітично, використовуються функції Minimize (Maximize). Ці функції мають однакові формати виклику і вміють працювати з символьними виразами:

Minimize[expr,x]

визначається мінімум виразу expr за змінною x;

Minimize[{expr, limitation},{x1,x2,...}]

визначається мінімум виразу expr за змінними x1, x2,... з урахуванням обмежуючих додаткових умов limitation. Якщо обмеження задає відкриту область, а екстремум знаходиться на межі, то функція Minimize можуть повернути точку, яка лежить на межі. Обмежуючі умови можуть містити рівняння, нерівності, їхню логічну комбінацію. Якщо вираз f та обмеження є лінійними або поліноміальними функціями Minimize завжди шукатиме глобальний екстремум. Якщо цій функції передані вирази, що містять наближені числа, екстремум вони будуть шукати не символьно, а чисельно, автоматично викликаючи функції NMinimize. Якщо область визначення змінних не задана, то вважається, що змінні дійсні.

Функції MinValue, ArgMin, MaxValue та ArgMax розв'язують завдання, аналогічні до Maximize та Minimize. Функції MinValue та MaxValue повертають значення мінімуму та максимуму, а функції ArgMin та ArgMax повертають координати точки, в якій досягається екстремум.

Функція NMinimize (NMaximize) чисельно знаходить мінімум (максимум) функції чи виразу. Вона повертає результат у такій формі як і функція Minimize.

NMinimize[expr,x]

чисельно визначається мінімум виразу expr за змінною x.

NMinimize[expr,{x1,x2,...}]

чисельно визначається мінімум виразу expr за змінними x1, x2,....

NMinimize[{expr, limitation },{x1,x2,...}]

чисельно визначається мінімум виразу expr за змінними x1,x2,... з урахуванням обмежень.

Для функції NMaximize (NMinimize) існують опції, які доречно використовувати у чисельних алгоритмах:

WorkingPrecision визначає кількість значущих цифр, що використовуються у внутрішніх обчисленнях; WorkingPrecision-> MachinePrecision (за замовчуванням) призводить до обчислень з процесорною точністю.

AccuracyGoal->n визначає кількість значущих цифр абсолютної похибки, яка вважається рівною 10^{-n} .

PrecisionGoal->m визначає відносну похибку $|x|\ 10^{-m}$ обчислень і визначається кількістю значущих цифр. За замовчуванням для цих опцій є WorkingPrecision/2.

EvaluationMonitor задає набір команд, які обчислюються щоразу, коли обчислюється значення виразу у функції NMinimize.

У функці] NMaximize (NMinimize) можна використовувати декілька різних алгоритмів чисельного пошуку екстремуму, які можна задати за допомогою опції Method.

Для пошуку локального мінімуму (максимуму) функції f(x) в околі точки x=x0, використовується функція FindMinimum (FindMaximum). Результат повертається у такому ж вигляді, як і в NMaximize (NMinimize).

У форматі

FindMinimum[f[x],{x,x0}]

чисельно шукається локальний мінімум виразу f[x] в околі початкової точки x0;

FindMinimum[f[x],x]

чисельно шукається локальний мінімум виразу f[x], починаючи пошук із випадкової точки;

FindMinimum[f[x,y,...],{{x,x0},{y,y0},...}]

чисельно шукається локальний мінімум функції кількох змінних в околі точки {x0,y0,...};

FindMinimum[{f[x,y,...], limitation },{{x,x0},{y,y0},...}]

чисельно шукається локальний мінімум із урахуванням обмежень. При цьому точку {x0,y0,...} можна не задавати - вона автоматично вибиратиметься в області, що визначається обмеженнями.

Для функції FindMinimum (FindMaximum) передбачено використання опцій. Опції WorkingPrecision, AccuracyGoal, PrecisionGoal використовуються так само, як і в функції NMinimize. Опція Method має інші значення, оскільки алгоритми пошуку локального та глобального екстремумів відрізняються.

Детально про Minimize: https://reference.wolfram.com/language/ref/Minimize.html?q=Minimize

Детально про NMinimize: https://reference.wolfram.com/language/ref/NMinimize.html?q=NMinimize

Детально про FindMinimum: https://reference.wolfram.com/language/ref/FindMinimum.html?q=FindMinimum

6. ІНТЕГРУВАННЯ

Нехай потрібно обчислити інтеграл

$$I = \int_a^b F(x)\,dx$$

де [a, b] - скінченний або нескінченний проміжок числової осі; F(x) – неперервна функція на відрізку [a, b].

На практиці у випадках, коли підінтегральна функція F(x) задана таблично або функція має особливості чи складний аналітичний вигляд, постає питання наближенного обчислення інтеграла від неї. Наближені формули обчислення інтеграла ґрунтуються на апроксимації підінтегральної функції іншою функцією або алгебричним багаточленом, або раціональною функцією тощо, для якої легко записується первісна

$$\int_a^b f(x)dx = \int_a^b \varphi(x)dx + R = S + R$$

де S - наближене значення інтеграла, R – похибка розрахунку інтеграла.

Методи чисельного інтегрування, що використовуються на практиці, можна згрупувати в залежності від способу апроксимації підінтегральної функції. Розглянемо коротку характеристику груп найпоширеніших методів.

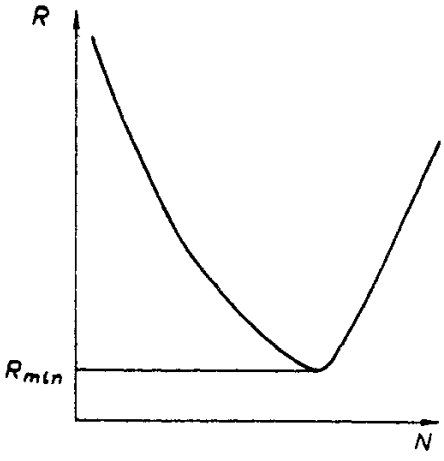
Методи Ньютона-Котеса базуються на поліноміальній апроксимації підінтегральної функції. Методи цього класу відрізняються один від одного ступенем полінома, від якого залежить кількість вузлів, де необхідно обчислити функцію f(x). Алгоритми методів прості та легко піддаються програмній реалізації.

Сплайнові методи базуються на апроксимації підінтегральної функції сплайнами. Методи розрізняються за типом вибраних сплайнів. Такі методи є сенс використовувати в тих задачах, де алгоритми сплайнової апроксимації застосовуються для обробки даних.

У методах найвищої алгебраїчної точності (методи Гаусса-Крістофеля та інші) використовують нерівновіддалені вузли, розташовані за алгоритмом, що забезпечує мінімальну похибку інтегрування для найбільш складних функцій при заданій кількості вузлів. Методи розрізняються способами вибору вузлів і широко використовуються для інтегрування, і в тому числі застосовні для невласних інтегралів. Хоча через необхідність зберігання числових констант та стандартизації меж інтегрування програми зазначених методів вимагають дещо більшого обсягу пам'яті порівняно з методами Ньютона-Котеса.

У методах Монте-Карло вузли вибираються випадково, відповідь має імовірнісний характер. Методи виявляються ефективними при обчисленні великої кратності.

У клас спеціальних групуються методи, алгоритми яких розробляються на основі врахування особливостей конкретних підінтегральних функцій, що дозволяє суттєво скоротити час та зменшити похибку обчислення інтегралів.



Незалежно від обраного методу у процесі чисельного інтегрування необхідно обчислити наближене значення S інтеграла та оцінити похибку R. Похибка зменшуватиметься при збільшенні кількості розбиття N інтервалу інтегрування [a, b] за рахунок більш точної апроксимації підінтегральної функції, проте при цьому зростатиме похибка за рахунок підсумовування часткових інтегралів, і остання похибка починаючи з деякого значення N_0 переважає. Ця обставина повинна застерегти від вибору надмірно великого числа N і привести до розробки способу оцінки похибки R обраного методу інтегрування.

6.1. Різноманітні n-точкові формули

Розглянемо інтеграл

$$I(f) = \int_a^b p(x) f(x)\, dx,$$

де p(x) - деяка функція, яку називають ваговою. Вважатимемо функцію p(x) фіксованою і такою, що добуток p(x)f(x) є абсолютно інтегрована функція на проміжку [a, b].

Для наближеного обчислення інтеграла I(f) будують лінійні квадратурні формули (квадратури) наступного виду:

$$S_n(f) = \sum_{i=1}^{n} c_i f(x_i).$$

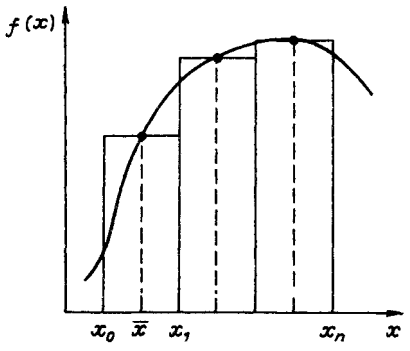
Постійні c_i називають коефіцієнтами (вагами) квадратури, x_i - її вузлами.

Для кожної функції f(x) похибка квадратурної формули $S_n(f)$ визначається як $R_n(f) = I(f) - S_n(f)$.

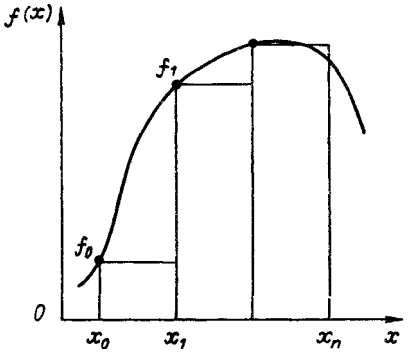
Отже, створити n-точкову інтегральну формулу (квадратурну схему) можна, якщо взяти n рівномірно розташованних точок $x_1 \dots x_n$ і підібрати в них ваги $c_1 \dots c_n$ так, щоб для еталонних функцій - поліномів $f_k(x) = x^k \quad (k = 0 \dots n-1)$ – вирази $\sum c_i f_k(x_i)$ збігалися з точними відповідями для відповідних інтегралів $\int_{a=x_1}^{b=x_n} f_k(x)dx$. Для спрощення викладок точки x_i доцільно розташовувати симетрично відносно нуля, тоді рівняння для ваг також виявляться симетричними (як і самі ваги), зокрема, рівняння для непарних еталонних функцій перетворяться в тотожності.

6.1.1. Метод прямокутників

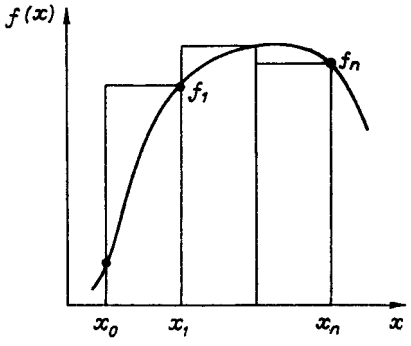
Розглянемо найпростіші n-точкові методи з класу методів Ньютона-Котеса, коли підінтегральна функція f(x) на інтервалі інтегрування замінюється поліномом нульового ступеня, тобто константою. Подібна заміна є неоднозначною, оскільки константу можна вибрати рівною значенню підінтегральної функції в будь-якій точці на інтервалі інтегрування Наближене значення інтеграла визначиться як площа прямокутника, одна зі сторін якого є довжина відрізка інтегрування, а інша - апроксимуюча константа, звідси походить назва методів. Найменшу похибку має метод середніх прямокутників, коли константа вибирається рівною значенню f(x) у середній точці інтервалу інтегрування. Методи лівих та правих прямокутників мають порівняно високу похибку.



Метод середніх прямокутників



Метод лівих прямокутників



Метод правих прямокутників

Узагальнені вирази для інтеграла на інтервалі [a, b], отримані методом прямокутників такі:

Формула середніх прямокутників

$$\int_a^b f(x)dx = h\left(f\left(x_0 + \frac{h}{2}\right) + f\left(x_1 + \frac{h}{2}\right) + \dots + f\left(x_{n-1} + \frac{h}{2}\right)\right) + h^2\frac{(b-a)}{24}f''(x); \qquad (1)$$

Формула лівих прямокутників

$$\int_a^b f(x)dx = h(f_0 + f_1 + \dots + f_{n-1}) + h\frac{(b-a)}{2}f'(x); \qquad (2)$$

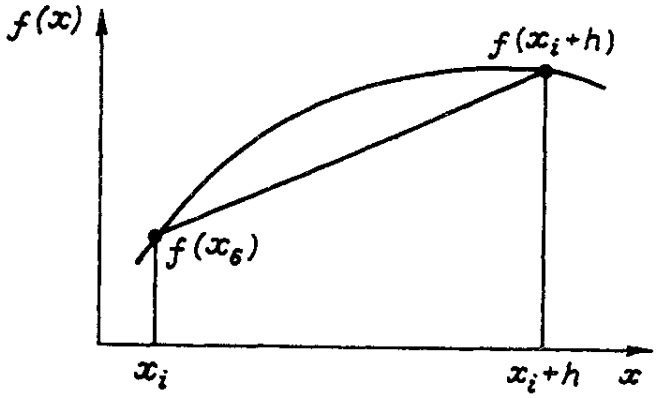
Формула правих прямокутників

$$\int_a^b f(x)dx = h(f_1 + f_2 + \dots + f_n) - h\frac{(b-a)}{2}f'(x), \qquad (3)$$

де $h = \frac{b-a}{n}$, n – кількість частин, на які поділено інтервал [a, b], $f_i = f(x_i)$, $x_i = x_0 + ih$.

6.1.2. Формула трапецій

Підінтегральну функцію замінимо на інтервалі $[x_i,\ x_i + h)]$ поліномом першого ступеня P(x). Як і в методі прямокутників, така апроксимація неоднозначна. Одним із можливих способів є проведення прямої через значення функції на межах інтервалу інтегрування. В цьому випадку наближене значення інтеграла визначається площею трапеції.

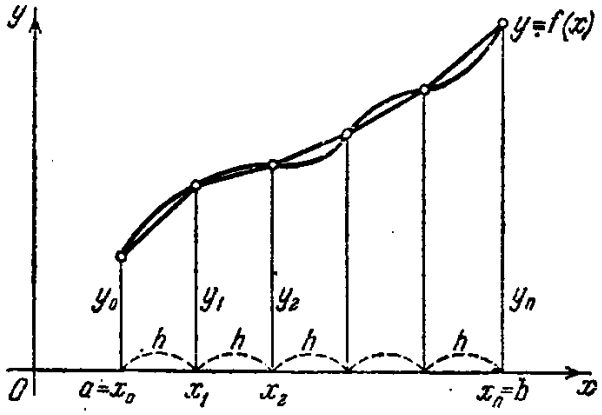


Якщо інтегрування виконується шляхом розбиття відрізка [a, b] на кілька інтервалів з кроком h, то узагальнена формула трапецій має вигляд:

$$\int_a^b f(x)dx = \frac{h}{2}(f_0 + f_n + 2(f_1 + \cdots + f_{n-1})) - h^2\frac{(b-a)}{12}f''(x). \qquad (4)$$

Отже, метод трапецій, як і метод середніх прямокутників, має другий порядок, однак його похибка вдвічі більша за абсолютною величиною порівняно з методом середніх прямокутників, хоча апроксимація підінтегральної функції проводилася поліномом першого, а не нульового ступеня. Очевидно, обраний варіант апроксимації підінтегральної функції прямою, що проходить через її значення на межах, не є оптимальним. Якщо підінтегральна функція задана аналітично, то краще з методів другого порядку застосовувати метод середніх прямокутників внаслідок його меншої похибки. Задача вибору способу апроксимації поліномом заданого ступеня з найменшою можливою похибкою була розв'язана Гаусом, що привело до розвитку цілого класу методів.

6.1.3. Формула Сімпсона



Підінтегральна функція f(x) замінюється інтерполяційним поліномом другого ступеня $P_2(x)$ - параболою, що проходить через вузли x_{i-1}, x_i, x_{i+1}, тому метод Сімпсона ще називається методом парабол.

Якщо зістикувати багато таких триточкових відрізків на інтервалі [a, b], то отримається узагальнена формула Сімпсона:

$$\int_a^b f(x)dx = \frac{h}{3}(f_0 + f_{2m} + 4\,(f_1 + \cdots + f_{2m-1}) + 2(f_2 + \cdots + f_{2m-2})) - h^4\frac{(b-a)}{180}f^{(4)}(x), \quad (5)$$

$x_0 = a, x_{2m} = b, x_i = x_0 + ih, h = \frac{b-a}{n}, n = 2m$ – парне.

Отже, формула Сімпсона має четвертий порядок точності з дуже малим чисельним коефіцієнтом залишкового члена. Формула Сімпсона дозволяє отримати високу точність, якщо четверта похідна підінтегральної функції не надто велика. В іншому випадку методи другого порядку можуть дати більшу точність, ніж метод Сімпсона. Наприклад, для функції $f(x) = -25x^4 + 45x^2 - 7$ формула трапецій при n = 2 для інтеграла в межах [-1, 1] дає

точний результат, що дорівнює 4, тоді як за формулою Сімпсона отримується результат, який не збігається навіть за знаком (-8/3).

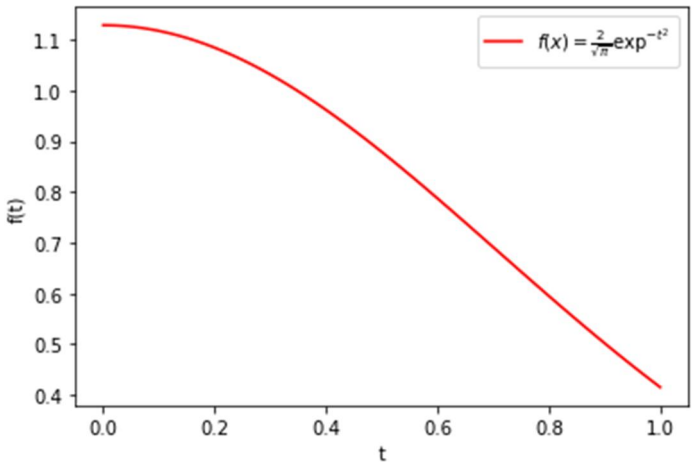
Зрозуміло, що процес побудови n-точкових формул можна продовжувати до нескінченності. Однак, потрібно пам’ятати, що якщо при вибраному кроці формула Сімпсона, як правило, набагато краще формули трапецій, то це не означає, що семиточкова схема настільки ж краща за пятиточкову оскільки порядок схеми зменшення кроку сильніше впливає на точність, ніж застосування схеми більш високого порядку. Навіть за наявності дуже швидкого комп’ютера застосування неефективного алгоритму при дуже дрібному кроці зовсім не гарантує точної відповіді - невеликий крок означає велику кількість операцій, тобто більшу накопичену помилку округлення.

Приклад програмної реалізації методу прямокутників для розрахунку інтеграла похибок

$$I(x) = \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\,dt \quad \text{при} \quad x = 1$$

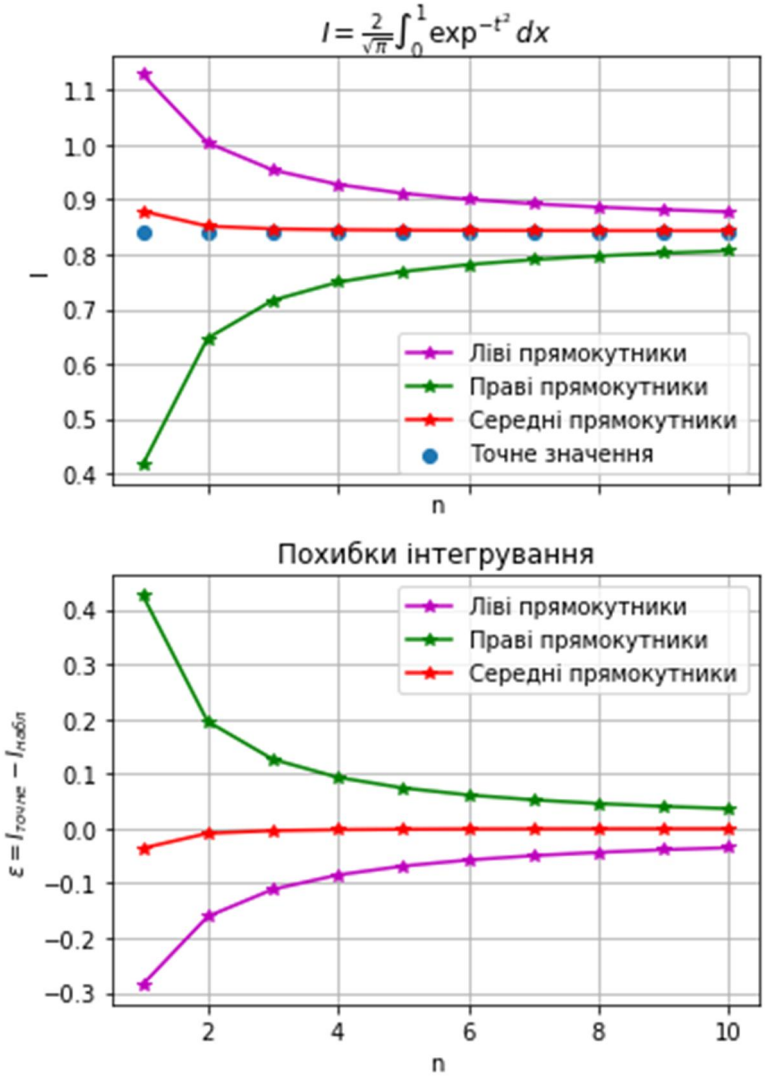
```
def rectAvr(f, a, b, n):
    """Метод середніх прямокутників"""
    h = (b - a)/n
    xi = [(a + (i + 1/2) * h) for i in range(n)]
    fi = [f(x) for x in xi]
    return h * sum(fi)

def rectLeft(f, a, b, n):
    """Метод лівих прямокутників"""
    h = (b - a)/n
    xi = [a + i * h for i in range(n)]
    si = [f(x) for x in xi]
    return h * sum(si)

def rectRight(f, a, b, n):
    """Метод правих прямокутників"""
    h = (b - a)/n
    xi = [a + i * h for i in range(1, n + 1)]
    si = [f(x) for x in xi]
    return h * sum(si)

import matplotlib.pyplot as plt
from numpy import import pi, exp, linspace

f = lambda t: 2/pi**0.5 * exp(- t**2)

a, b = 0, 1
xi = linspace(a, b, 101)
yi = f(xi)

plt.plot(xi, yi, 'r', label=r'$f(x) = \frac{2}{\sqrt{\pi}} \exp^{- t^2}$')
```

```
plt.ylabel('f(t)')
plt.xlabel('t')
plt.legend()
plt.show()
```




```
from scipy.special import erf

a, b = 0, 1
exact_val = erf(1)
n_max = 10
ni = range(1, n_max+1)
sLeft = [rectLeft(f, a, b, n) for n in ni]
sRight = [rectRight(f, a, b, n) for n in ni]
sAvr = [rectAvr(f, a, b, n) for n in ni]

epsLeft = [exact_val - rectLeft(f, a, b, n) for n in ni]
epsRight = [exact_val - rectRight(f, a, b, n) for n in ni]
epsAvr = [exact_val - rectAvr(f, a, b, n) for n in ni]

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(5, 7), sharex=True, tight_layout=True)
fig.subplots_adjust(hspace=0.5)
ax1.plot(ni, sLeft, 'm*-', label='Ліві прямокутники')
ax1.plot(ni, sRight, 'g*-', label='Праві прямокутники')
ax1.plot(ni, sAvr, 'r*-', label='Середні прямокутники')
ax1.scatter(ni, [exact_val]*n_max, marker='o', label='Точне значення')
ax1.set_title(r'$I =\frac{2}{\sqrt{\pi}} \int_{0}^{1} \exp^{- t^2}\, dx$')
ax1.set_ylabel('I')
ax1.set_xlabel('n')
ax1.grid(True)
ax1.legend()
ax2.plot(ni, epsLeft, 'm*-', label='Ліві прямокутники')
ax2.plot(ni, epsRight, 'g*-', label='Праві прямокутники')
ax2.plot(ni, epsAvr, 'r*-', label='Середні прямокутники')
ax2.set_title('Похибки інтегрування')
```

```
ax2.set_ylabel(r'$\varepsilon = I_{точне} - I_{набл}$')
ax2.set_xlabel('n')
ax2.grid(True)
ax2.legend()
plt.show()
```



6.2. Алгоритм Ромберга

Алгоритм Ромберга є інтерполяційним алгоритмом, який використовує найпростішу з квадратурних схем – двоточкову (формулу трапецій). Фактично реалізується той метод, який вже був викладений для формули трапецій, але з невеликим доповненням, яке і робить метод надзвичайно ефективним.

Програмна реалізація алгоритму Ромберга із використання формул Рунге або Ейткена дозволяє обчислити інтеграли із заданою точністю, коли вибір необхідного числа розбиттів інтервалу інтегрування здійснюється автоматично. При цьому, звісно, можна використовувати багаторазове звернення до підпрограм відповідних методів інтегрування, не змінюючи алгоритмів самих методів. Однак для методів, що використовують рівновіддалені вузли, вдається модифікувати алгоритми та зменшити вдвічі кількість обчислень підінтегральної функції за рахунок використання інтегральних сум, накопичених при попередніх кратних розбиття інтервалу інтегрування.

Отже, суть алгоритму полягає у тому, що для інтеграла

$$\int_{x_0}^{x_n} f(x)dx$$

розраховуються два наближені значення S_k та S_{k-1} за формулою трапецій з кроками h_k та h_{k-1} відповідно, які пов'язані співвідношенням

$$S_k = \frac{S_{k-1}}{2} + h_k \sum_{i=1}^{2^{k-1}} f(x_0 + (2i-1)h_k), \quad (1)$$

де

$$h_k = \frac{h_{k-1}}{2} = \frac{h_0}{2^k}. \quad (2)$$

Якщо на інтервалі $[x_0, x_n]$ вибрати початковий крок інтегрування $h_0 = x_n - x_0$, то наближене значення інтеграла за методом трапецій запишеться у вигляді

$$S_0 = \frac{h_0}{2}(f_0 + f_n), \qquad f_i = f(x_i). \quad (3)$$

При зменшенні кроку h_0 вдвічі отримаємо наближене значення того ж інтеграла

$$S_1 = \frac{h_1}{2}\left(f_0 + 2f_{n/2} - f_n\right), \quad h_1 = \frac{h_0}{2}. \quad (4)$$

З формул (3) і (4) отримується співвідношення між наближеними значеннями S_0 та S_1

$$S_1 = \frac{S_0}{2} + f_{n/2}h_1,$$

яке дозволяє одержувати наближене значення інтеграла S_1 з кроком h_1 обчисливши підінтегральну функцію лише в одному додатковому вузлі $x_n/2$. Аналогічним способом на i-тій ітерації отримаються співвідношення між двома наближеними значеннями S_k та S_{k-1}

$$S_{k,i} = \frac{2^{2i}S_{k,i-1} + S_{k-1,i-1}}{2^{2i} - 1}, \quad i = 0, \dots, k.$$

Продовжуючи процес зменшення кроку інтегрування вдвічі, приходимо до формули (1), за якою кожне нове наближене значення інтеграла отримуємо, обчисливши додатково підінтегральну функцію лише в 2^{k-1} вузлі. Звернення до підпрограми методу трапецій вимагало б обчислення функції в (2^k+1) вузлах. Похибкою вважається різниця між поточним результатом і результатом, отриманим на попередньому кроці: $|S_k - S_{k-1}|$. Як тільки похибка виявляється менше замовленої точності, процедура зупиняється.

Подальший розвиток алгоритму дозволяє перейти до так званих адаптивних квадратур, в яких скорочується кількість обчислень підінтегральної функції за рахунок вибору різного кроку інтегрування в різних частинах інтервалу в залежності від поведінки функції.

Збіжність методу вражає уяву. Вона досягається при настільки малій кількості точок (і, відповідно, великому кроці h), що не тільки формула трапецій, а й формула Сімпсона ще дуже далека від збіжності. Якщо замовлена точність близька до машинного нуля, то метод Сімпсона може не зійтися взагалі (при дуже маленькому кроці буде занадто багато обчислень функції і накопичиться велика помилка округлення), в той час як метод Ромберга цілком може зійтися.

6.3. Змінний крок

Зрозуміло, метод Ромберга треба застосовувати свідомо. Саме, якщо функція майже постійна в одній області і різко змінюється в іншій, то краще використовувати метод Ромберга два рази - окремо для кожної з областей. Плавну область він пройде з великим кроком, а область різкої зміни - з маленьким.

Цей процес можна автоматизувати адже обчислення певного інтеграла $\int_a^b f(x)dx$ можна розглядати як розв'язання диференціального рівняння $y' = f(x)$ з початковою умовою y(a)=0. Тоді шуканий інтеграл є в точності $y(b) = \int_a^b f(x)dx$. Розроблені методи чисельного розв'язання диференціальних рівнянь, що допускають автоматичну зміну кроку в залежності від властивостей функції, можуть бути використані при наближеному розоахунку інтегралів.

Справа в тому, що при розв'язання диференціального рівняння контролюється тільки локальна точність інтегрування. Глобальна точність, тобто точність шуканого інтеграла, є локальна точність, помножена на число кроків, яке заздалегідь невідоме. Таму для оцінки похибки шуканого інтеграла потрібно розв'язувати диференціальне рівняння кілька разів, задаючи різну локальну точність. Якщо при помітному (як мінімум в 1.5 рази) зменшенні кроку відповідь майже не змінюється, то можна вважати, що інтеграл обчислений правильно. Досвід показує, що доцільніше все ж таки використовувати алгоритм Ромберга.

6.4. Метод Гаусса (метод найвищої алгебраїчної точності)

Підінтегральну функцію f(x) так само, як і в методах Ньютона-Котеса, будемо апроксимувати поліномами різних ступенів. Однак, на відміну від методів Ньютона-Котеса, вузли для побудови інтерполяційного полінома виберемо з умови забезпечення мінімальної похибки інтегрування. Вперше задача побудови квадратурних формул подібного типу була розв'язана Гаусом для інтегралів виду $\int_a^b f(x)dx$, а для інтегралів $\int_a^b \rho(x)f(x)dx$ з довільною ваговою функцією $\rho(x)$ – Крістоффелем.

Інтегрування по Гауса полягає в наступному:

Алгоритм

Будуємо N+1 полином (від P_0 до P_N), що належать деякому сімейству класичних ортогональних поліномів (КОП), використовуючи відомі рекурентні співвідношення

$$P_{n+1} = xP_n + a_nP_n + b_nP_{n-1}$$

$$a_n = -\langle P_n|x|P_n\rangle/\langle P_n|P_n\rangle, \quad b_n = -\langle P_n|P_n\rangle/\langle P_{n-1}|P_{n-1}\rangle.$$

Шукаємо усі N коренів ($x_{1,..}x_N$) полінома P_N. Далі, розв'язуємо систему лінійних рівнянь N×N відносно ваг a_i:

$$\sum_{i=1}^{N} \rho(x_i)P_0(x_i)a_i = \int_a^b \rho(x)\,dx$$

$$\sum_{i=1}^{N} \rho(x_i)P_k(x_i)a_i = 0, \qquad k = 1,\ldots N-1.$$

За відомими точками x_i та коефіцієнтами a_i апроксимуємо довільний інтеграл на відрізку [a, b] сумою:

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} a_i f(x_i).$$

Очевидно, що цей вираз дасть точну відповідь для f(x), що збігається з будь-якою лінійною комбінацією перших N+1 поліномів (від P_0 до P_N), помноженої на $\rho(x)$. Так що будь-яка функція f(x), яка володіє тією властивістю, що відношення $f(x)/\rho(x)$ добре разкладається з даного сімейства КОП (тобто коефіцієнти розкладу досить швидко зменшуються з ростом номера полінома), методом Гаусса буде проінтегрована правильно.

Зрозуміло, не завжди відрізок, на якому обчислюється інтеграл, збігається з відрізком, на якому визначено КОП. Ця проблема вирішується тривіальним лінійнім перетворенням змінної інтегрування *x*.

Зауважимо, що метод Гауса природним чином дозволяє знаходити коефіцієнти розкладу довільної функції з даного сімейства КОП. Дійсно, коефіцієнт розкладу $c_k = \int_a^b dx\, y(x)\, P_k(x)\rho(x)$ визначаються інтегралом якраз «підходящого» типу. Це дозволяє проводити інтерполяцію з допомогою довільного сімейства КОП.

Особливу роль в методі Гауса відіграють поліноми Лежандра, для яких відрізок скінченний, а вага дорівнює одиниці: a =-1, b=1, $\rho(x)$=1. Більшість бібліотечних функцій, що реалізують Гаусове інтегрування, використовують саме поліноми Лежандра.

Зручні також і поліноми Чебишова, для яких відомі явні вирази як для коренів $x_i = \cos(\pi(i-1/2)/N)$, так і для коефіцієнтів $a_i = (\pi/N)\sin(\pi(i-1/2)/N)$. Здається, що при використанні методу Гаусса для інтегрування по кінцевому інтервалу тільки їх слід застосовувати, оскільки немає необхідності знаходити корені і розв'язувати систему лінійніх рівнянь для a_i. На жаль, це не зовсім так. Справа в тому, що поліноми Чебишова визначені на кінцевому відрізку, але їх вага має кореневу особливість: a =-1, b=1, $\rho(x) = 1/\sqrt{1-x^2}$. Тому вони оптимальні в тому випадку, коли підінтегральна функція має вид $f(x) = g(x)/\sqrt{1-x^2}$, а якщо функція $g(x)$ добре розкладається по поліномах Чебишова. Втім, для поліномів Чебишова неважко обчислити і альтернативні ваги $\tilde{a}_l$, які дадуть точну відповідь для інтеграла від будь-якої лінійної комбінації перших N полиномів $T_0,...,T_{N-1}$, без множення на вагу. Дійсно, маючи коефіцієнти c_i розкладу функції f(x) за поліномами Чебишова

$$c_l = \sum_{k=0}^{n-1} f(x_k) T_l(x_k)/M_l,$$

де $M_l = N(1+\delta_{l0})/2$, можна знайти невизначений інтеграл від неї:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{l=0}^{N-1} d_l [T_l(1) - T_l(-1)],$$

де d_0 = const, $d_l = (c_{l-1} - c_{l+1})/(2l)$ при $l \geq 1$. Отже,

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{m=0}^{(N-1)/2} [1/(2m+1) - 1/(2m-1)] c_{2m},$$

Звідки остаточно отримуємо

$$\tilde{a}_i = \sum_{m=0}^{(N-1)/2} [1/(2m+1) - 1/(2m-1)] T_{2m}(x_i)/M_{2m}.$$

Легко перевірити, що ваги $\widetilde{a_i}$ досить близькі до канонічних для поліномів Чебишова ваг $a_i = (\pi/N)\sin(\pi(i-1/2)/N)$ і збігаються до них при N→∞. Проте, при інтегруванні полінома канонічні ваги a_i дадуть відповідь з деякою погрішністю (близько 10^{-3} при N = 20), в той час як альтернативні ваги $\widetilde{a_i}$ дадуть точну відповідь.

Тому в методі Гаусса при інтегруванні по кінцевому інтервалу іноді доцільно використовувати не поліноми Лежандра, а поліноми Чебишова з альтернативними вагами.

При вибраній кількості точок, в яких обчислюється підінтегральна функція, метод Гаусса, як правило, точніше інших методів. Формальна причина дуже проста - для N точок метод Гаусса є N-точковою схемою, яка дає точну відповідь для 2N еталонних функцій - полиномів $P_0,\ldots, P_{2N-1}$, помножених на вагу. Проте платою за це є необхідність обчислювати корені x_i і відповідні ваги a_i. Крім того, в цьому методі досить обмежений контроль точності: щоб змінити крок (кількість точок N), доведеться заново знаходити x_i та a_i. Втім, можна розбити відрізок інтегрування навпіл і на кожній з половин застосувати метод Гаусса з попередньою кількістю точок N (тим самим загальна кількість точок зросте вдвічі). Однак, слід пам'ятати, що функція, яка добре розкладається на вихідному відрізку, взагалі кажучи, не зобов'язана настільки ж добре розкладається на половинках цього відрізка.

Отже, якщо необхідно просто обчислити визначений інтеграл, то метод Гаусса краще не застосовувати. Однак, у тих випадках, коли важливо мінімізувати кількість викликів інтегрованої функції (наприклад, якщо функція обчислюється з виключно великими труднощами, або при розв'язанні інтегральних рівнянь) доцільно використовувати саме метод Гаусса.

6.5. Невласні інтеграли

Основний рецепт при взятті невласних інтегралів простий: не брати їх чисельно оскільки будь-який збіжний невласний інтеграл відповідною заміною змінних можна перетворити у регулярний.

Підхід до інтегрування функцій з особливостями визначається характером особливості:

• хибна невласність, наприклад: $\int_0^1 \sin(x)/x\ dx$, - слід доозначити функцію в точці x=0.

• інтегрувальна особливість, наприклад: $\int_0^1 1/\sqrt{x}\,dx$, - заміна змінної інтегрування $x = t^2$ перетворить невласний інтеграл в регулярний інтеграл $\int_0^1 2dt$.

У літературі зустрічаються поради застосовувати n-точкові схеми, в яких не використовується кінцева точка, в якій у інтегрованої функції особливість. Однак досвід показує, що краще ці рецепти не застосовувати оскільки вони можуть призвести до занадто великого ризику втрати точності. Краще, все ж таки, перетворити шуканий інтеграл.

При нескінченних межах інтегрування все залежить від поведінки підінтегральної функції на нескінченності:

• хибна невласність, наприклад: $\int_0^\infty \exp(-x^2)\,dx$, - інтегрована функція спадає досить швидко. Легко оцінити верхню межу інтегрування, далі якої виконувати розрахунок не має сенсу. У цьому випадку інтегрування в дійсності йде по кінцевому інтервалу. Для перевірки можна збільшити обрану верхню межу інтегрування (наприклад удвічі) і

переконатися, що зміна відповіді не перевищує замовлену точність. Якщо це не так, то слід повторити процедуру, збільшиши верхню межу, і так до тих пір, поки не буде досягнута замовлена точність.

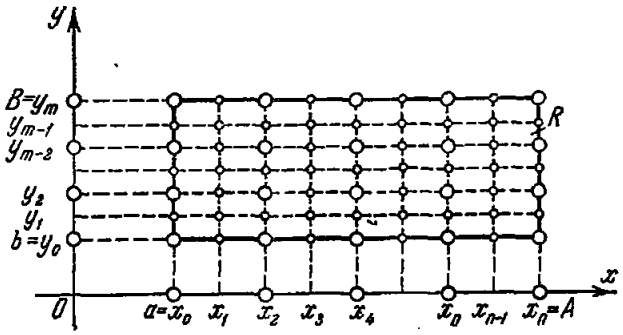
• повільно збіжний на нескінченності інтеграл, наприклад: $\int_1^\infty 1/x^2 dx$, - заміна змінної інтегрування $x = 1/t$ перетворить його в регулярний інтеграл $\int_0^1 dt$. Іншою можливістю є віднімання головної асимптотики від підінтегральної функції. Як правило, асимптотику можна проінтегрувати аналітично, а решта буде сходитися досить швидко.

Очевидно, що в разі $\int_1^\infty 1/x^2 dx$ спроба взяти велику верхню межу інтегрування, а потім збільшувати її, поки відповідь не перестане змінюватися, не ефективна, оскільки при даній верхній межі X залишковий член буде порядка 1/X і досягнути задовільної точності за кінцевий час неможливо. Для інтеграла типу $\int_1^\infty \sin(x)/x dx$ ситуація ще гірша.

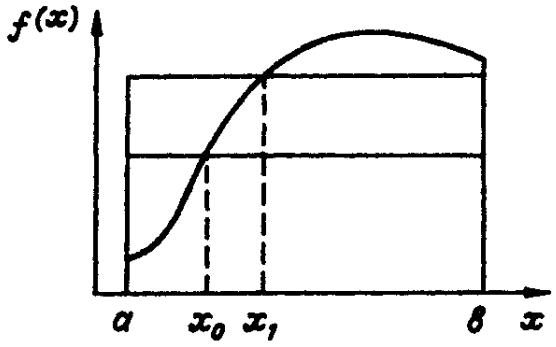
Отже, в разі невласних інтегралів найкраще привести задачу до регулярного виду.

6.6. Багатовимірні інтеграли

У випадку не дуже великої кількості вимірів і не дуже складної поверхні, що обмежує область інтегрування, багатовимірний інтеграл потрібно брати як повторний. В якості додаткової перевірки можна поміняти порядок інтегрування: спочатку по x, а потім по y, і навпаки - спочатку по y, потім по x, та порівняти відповіді. При інтегруванні по кожному виміру доцільно застосовувати алгоритм Ромберга. Тобто алгоритм Ромберга застосовується спочатку для внутрішнього інтеграла $g(x_k) = \int f(x_k, y)\, dy$ в кожній точці потрібної для зовнішнього інтеграла, а потім для зовнішнього інтеграла $\int g(x)\, dx$. Єдина технічна деталь - при взятті внутрішнього інтеграла слід замовляти точність вищу, ніж необхідна точність для зовнішнього, так що при кожному переході до більш внутрішнього інтеграла точність повинна зростати в кілька разів. Іноді в літературі рекомендують збільшувати точність на порядок, але досвід показує, що це занадто багато, достатньо збільшити її в декілька разів, а у випадках гладких функцій при плоскій границі, її можна і зовсім не збільшувати.



У разі дуже великої кількості вимірів або дуже складної межі інтегрування (коли межі інтегрування для повторних інтегралів важко виписати) викладений метод не підходить. Як правило, в цьому випадку рекомендують використовувати метод статистичних випробувань (метод Монте-Карло).



Розглянемо два варіанти методу Монте-Карло для обчислення інтегралів. Перший спосіб можна інтерпретувати як статистичний варіант методу прямокутників, коли за вузел x_0 береться випадкове число, рівномірно розподілене на інтервалі інтегрування [a, b]. Внаслідок випадковості вузла x_0 похибка

інтеграла також матиме випадковий характер. Виконавши N обчислень з випадковими вузлами x_i, усереднений результат дасть наближене значення інтеграла

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i), \qquad (1)$$

Похибка інтеграла зменшуватиметься зі зростанням числа випробувань N за законом $\varepsilon \sim 1/\sqrt{N}$.

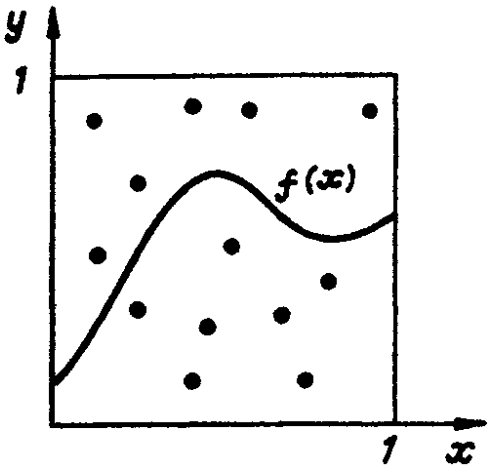
Формула (1) узагальнюється на кратні інтеграли

$$\iint \ldots \int f(x_1, x_2, \ldots, x_n)dv \approx \frac{v_c}{N}\sum_{i=1}^{N} f(x_{1i}, x_{2i}, \ldots, x_{ni}),$$

де v_c - n-мірний об'єм області інтегрування. Число вузлів, в яких доведеться обчислити підінтегральну функцію, буде пропорційно ~ ε^{-2} незалежно від кратності інтеграла.

У другому варіанті методу Монте-Карло інтеграл приводиться до вигляду

$$\int_0^1 f(x)dx$$



де $0 \le f(x) \le 1$ на інтервалі [0, 1]. Тоді дві випадкові величини x_i та y_i можна розглядати як координати точок у одиничному квадраті. При рівномірному розподілі точок у квадраті за наближене значення інтеграла приймається відношення кількості точок S, що потрапили під криву $y = f(x)$, до загального числа випробувань N

J = S / N.

Цей алгоритм також узагальнюється на кратні інтеграли.

Зауважимо, що метод Монте-Карло вимагає дуже якісного генератора випадкових чисел. Крім того, цей метод не здатний забезпечити дуже високу точність, так що його слід застосовувати тільки в тих випадках, коли нічого іншого не залишається.

6.7. Інтегрування засобами бібліотек Python

6.7.1 Чисельне інтегрування засобами бібліотеки scipy.integrate

У модулі integrate пакет sciру представлено програмне забезпечення на розв’язання двох класів задач обчислювальної математики. Розглядаються проблеми обчислення інтегралів неперервних та сіткових функцій. Другий клас задач пов'язаний з чисельним розв’язання задачі Коші для системи звичайних диференціальних рівнянь.

Основна функція обчислення визначених інтегралів від функції неперервного аргументу є quad(), яка має наступний синтаксис:

quad (func, a, b, …)

У quad() реалізовано чисельний метод із Fortran-бібліотеки QUADPACK (http://nines.cs.kuleuven.be/software/QUADPACK/). Функція приймає щонайменше три аргументи:

• func — ім'я підінтегральної функції. Якщо функція залежить від більш ніж одного аргументу, то інтегрування ведеться за першим аргументом. Для передачі підінтегральній

функції додаткових аргументів використовується опція args, якій передається кортеж з додатковими аргументами. Іноді зручно не передавати допоміжні аргументи через опцію args, а використовувати lambda – оболонку навколо підінтегральної функції;

• a - нижня межа інтегрування (число типу float);

• b – верхня межа інтегрування (число типу float).

Список та значення додаткових аргументів можна одержати за допомогою команди help(quad).

Функція quad() повертає кортеж із двох елементів, перший – значення інтеграла, другий – точність обчислення. За умовчанням функція quad обчислює наближене значення інтеграла з абсолютною та відносною похибками epsabs=1.49e-08, epsrel=1.49e-08. Тут epsabs і epsrel імена опцій, що задають ці похибки. Функція quad() дозволяє чисельно знаходити інтеграл на нескінченному інтервалі дійсної осі.

Крім quad(), у модулі scipy.integrate доступні також інші методи знаходження інтегралів:

trapz() - метод трапецій,

sims() - метод Сімпсона,

romberg() - метод Ромберга.

На відміну від quad(), у функціях trapz(), simps() та romb() сітка точок інтегрування задається користувачем, а не встановлюється автоматично.

На основі процедур інтегрування по одній змінній в бібліотеці scipy.integrate реалізовані також функції інтегрування по двох і трьох змінних (dblquad(), tplquad()).

dblquad(fun, a, b, gfun, hfun[,...])

Тут fun – це підінтегральна функція двох змінних, в якій ім'я внутрішньої змінної інтегрування має бути зазначено першим.

a, b - значення нижньої і верхньої межі інтегрування змінної x (межі зовнішнього інтеграла), gfun та hfun - імена функцій, що визначають нижню і верхню межі інтегрування змінної y (межі внутрішнього інтеграла). Зауважимо, що навіть якщо функції g і h нижньої та верхньої межі інтегрування константи їх все одно слід задавати функціями.

В dblquad() також є опції args, epsabs, epsrel, значення яких таке саме, як і для функції quad. Як і для одномірного інтеграла функція dblquad() повертає кортеж з двох чисел: значення інтеграла та абсолютну похибку.

tplquad(func,a,b,gfun,hfun,qfun,rfun[,args=(), epsabs=1.49e-08, epsrel=1.49e-08]).

Аргумент func - це підінтегральна функція трьох змінних, що задаються в порядку z, y, x. Аргументи a,b - межі інтегрування по x; gfun,hfun - нижня і верхня межі інтегрування по y і є функціями однієї змінної x; qfun,rfun - визначають нижню і верхню межі інтегрування по z і є функціями двох змінних x, y (у зазначеному порядку). Опцій args, epsabs, epsrel такі ж, як і в функції quad.

Для обчислення n-кратних інтегралів у модулі scipy.integrate є функція nquad. Вона має наступний синтаксис:

nquad(func, ranges, args=None, opts=None)

Тут func - функція, інтеграл від якої обчислюється. При створенні підінтегральної функції аргументи слід зазначати в порядку від x0 – змінної внутрішнього інтегрування, до xn – змінної зовнішнього інтегрування: def f(x0, x1, x2, x3,…, xn):… Аргумент ranges задає межі інтегрування й є послідовністю, кожен елемент якої має бути списком (або кортежем) із двох виразів (або двох чисел). Замість пари виразів можна використовувати функцію, яка повертає список із двох значень. Пара значень у ranges[0] задає межі інтегрування по x0, ranges[1] – по x1, тощо. Аргументи функцій, що задають межі інтегрування, також потрібно вказувати в наведеному порядку від x0 до xn.

Функція func, окрім змінних інтегрування, може приймати додаткові аргументи, які передаються їй через опцію args. Порядок аргументів функції func повинен відповідати порядку, в якому передаються межі інтегрування: від внутрішньої змінної до зовнішньої.

Часто в задачах теоретичної фізики виникає необхідність обчислювати інтеграли зі змінною верхньою межею $\int_a^x f(t)dt$. Такий інтеграл з точністю до константи збігається з первісною $\int f(x)dx$. У пакеті scipy.integrate є функція cumtrapz(), яка обчислює такий інтеграл. Вона має наступний синтаксис:

cumtrapz(y, x=None, dx=1.0, axis=-1, initial=None)

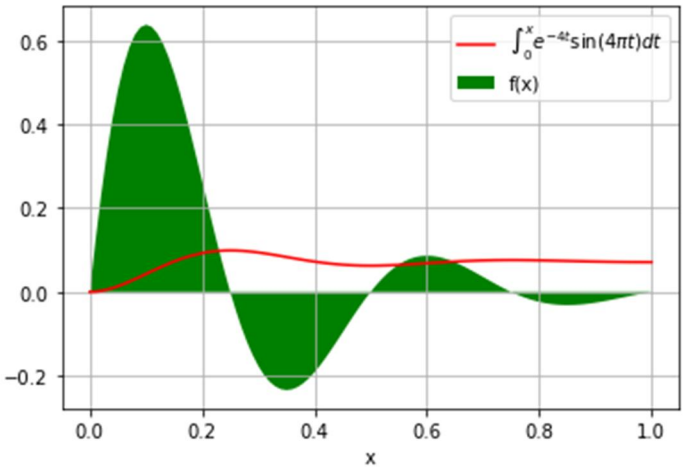
Тут y – одновимірний масив значень інтегрованої функції, x – одновимірний масив абсцис, dx – відстань між абсцисами (використовується тільки, коли x=None), axis – задає вісь інтегрування (за умовчанням використовується значення –1, тобто остання вісь). Параметр initial задає число, що використовується в якості першого значення масиву, що повертається. Зазвичай воно має дорівнювати нулю. За умовчанням йому присвоюється значення None, що означає, що значення x[0] не використовується, і масив, що повертається на одиницю коротше, ніж масив значень функції y. Якщо масив x не передається, то аргумент dx визначає відстань (осі x) між елементами масиву y. Результат не залежить від початкової абсциси x[0].

Детальніше про методи розрахунку інтегралів в модулі scipy.integrate: https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

Приклад розрахунку інтеграла $j(x) = \int_0^x e^{-4t}\sin(4\pi t)dt$

```
from numpy import exp, sin, zeros
import scipy.integrate as integrate
import matplotlib.pyplot as plt
f = lambda x: exp(-4*x) * sin(4*pi*x)

m = 101
x = zeros((m), 'float')
y = zeros((m), 'float')
y1 = zeros((m), 'float')

for i in range(0, m):
   x[i] = i/(m-1.)
   y[i] = integrate.quad(f, 0, x[i])[0]
   y1[i] = f(x[i])
```

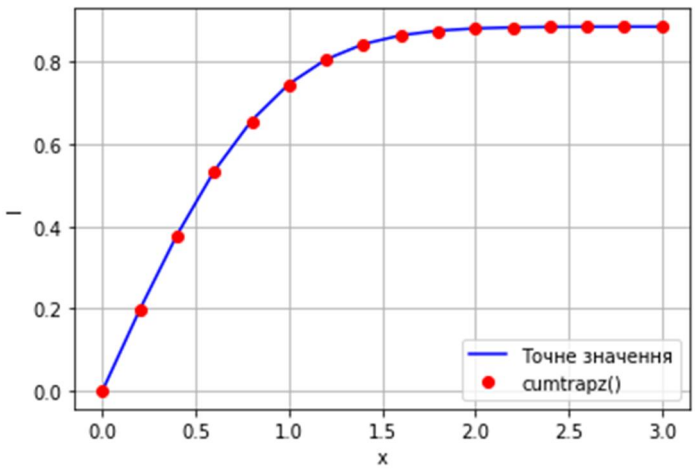
```
plt.fill(x, f(x), 'g', label='f(x)')
plt.plot(x, y, 'r', label=r'$\int_0^x e^{-4t} \sin(4 \pi t) dt$')
plt.legend(loc=0)
plt.xlabel('x')
plt.grid(True)
plt.show()

x0 = 1
I0 = integrate.quad(f, 0, x0)[0]
I1 = integrate.trapz(y1, x)
I2 = integrate.simps(y1, x)
print(f'x = {x0}\n exact value = 0.0709329...')
print(f'int(quad):  {I0}')
print(f'int(trapz): {I1}')
print(f'int(simps): {I2}')
```




```
>>x = 1
>>exact value = 0.0709329...
>>int(quad): 0.07093294896505287
>>int(trapz): 0.0708301283865725
>>int(simps): 0.0709330243013272
```

Приклад розрахунку інтеграла похибокзі зімнно межею $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$

```
from scipy.integrate import cumtrapz
from numpy import linspace, exp, pi, sqrt
from scipy.special import erf
import matplotlib.pyplot as plt

x = linspace(0, 3, num=16)
y = 2/sqrt(pi) * exp(-x**2)
yint = cumtrapz(y, x, initial=0)

plt.plot(x, erf(x), 'b-', label='Точне значення')
plt.plot(x, yint, 'ro', label='cumtrapz()')
plt.xlabel('x')
plt.ylabel('I')
plt.legend(loc=0)
plt.grid(True)
plt.show()
```



Приклад розрахунку інтеграла з нескінченною межею $\int_0^\infty e^{-x}\,dx$

```
from numpy import exp, inf
from integrate import quad
def f(x):
   return exp(-x)
quad(f2, 0.0, inf)
>>(1.0000000000000002, 5.842607038578007e-11)
```

Приклад розрахунку невласного інтеграла $\int_0^5 \frac{1}{x^2-4}dx$

```
from scipy.integrate import quad

f = lambda x: 1/(x**2 - 4)
a, b = 0, 5
point = 2
quad(f, a, b, points=point)
>>(-0.3131907421238081, 2.0317081350640365e-14)
```

6.7.2 Аналітичне інтегрування засобами бібліотеки sympy

У бібліотеці sympy для символьного інтегрування призначена функція integrate(), яка дозволяє обчислювати як невизначені інтеграли

integrate(f,x),

так і визначені

integrate(f,(x,a,b)).

Першим аргументом integrate() приймає символьний вираз - функцію, яка інтегрується, другим – змінну інтегрування або кортеж, що складається з імені змінної та її нижньої та верхньої межі. Якщо другий аргумент – ім'я, обчислюється невизначений інтеграл, тобто первісна підінтегральна функція. Якщо первісну не вдалося знайти, то повертається необчислюваний об’єкт Integral.

Функцію integrate() також можна використовувати для обчислення кратних інтегралів. При цьому внутрішні межі інтегрування потрібно задавати як функцію від змінної інтегрування зовнішнього інтеграла.

У integrate() є необчислюваний еквівалент – функція Integral(). Вона повертає об'єкт типу sympy.integrals.integrals.Integral. Щоб потім розрахувати інтеграл потрібно використати метод doit().

Дутальніше про sympy.integrate: https://docs.sympy.org/latest/modules/integrals/ integrals.html

6.8 Аналітичне та чисельне інтегрування засобами Mathematica

Для обчислення невизначеного інтеграла $\int f(x)dx$ використовується функція

Integrate[f, x].

При розрахунку інтеграла константа інтегрування не додається. Замість функції Integrate можна використовувати класичне позначення ∫. Трафарет введення знаку інтеграла можна знайти на панелі спеціальних символів "Basic Math Input". Також можна використовувати спеціальні комбінації клавіш. Для введення знака інтеграла потрібно натиснути Esc-int-Esc, потім ввести підінтегральний вираз, а далі спеціальний значок диференціала Esc-dd-Esc та ім'я змінної інтегрування. Можна інтегрувати вирази, що містять більш ніж одну змінну (інші змінні трактуватимуться як константи) та кратні інтеграли.

Mathematica вміє інтегрувати раціональні, кускові, тригонометричні та багато інших функцій. Вона знає багато інтегралів, які можуть бути визначені в термінах спеціальних функцій.

Для розрахунку визначеного інтеграла $\int_{Xmin}^{Xmax} f(x)dx$ використовується функція

Integrate[f,{x,xmin,xmax}]

При розрахунку визначеного інтеграла умову враховувати умови, накладені на змінні. Для цього використовується опція Assumptions або Assuming. У функції Assuming два аргументи: перший – список умов, другий – вираз, що обробляється.

Mathematica може обчислювати невласні інтеграли з нескінченними межами інтегрування, наприклад Integrate[f,{x,-∞, ∞}]. Також можна розраховуватие значення інтеграла у розумінні головного значення, використовуючи опцію PrincipalValue->True.

Визначен інтеграли, які залежать від параметра, можна диференціювати за цим параметром.

Для розрахунку кратного інтеграла використовується функція

Integrate[f, {xn, xnmin, xnmax},…,{x2, x2min, x2max},{x1, x1min, x2max}]

Першим вказується інтервал зміни тієї змінної, інтегрування за якою виконується в останню чергу.

Функцію Integrate можна використовувати для обчислення інтегралів по області. Для цього використовується повторний інтеграл, попередньо зануливши функцію поза цією областю.

Функція Integrate вміє обчислювати інтеграли від комплексних функцій дійсного аргументу. При цьому немає необхідності виділяти дійсну та уявну частини функції.

Для розрахунку інтеграла за ламаними в комплексній площині. використовується форма

Integrate[f[z], {z , z1, z2, … }],

де zi - комплексні числа (точки комплексної площини).

Детальніше про Integrate: https://reference.wolfram.com/language/ref/Integrate.html

Для чисельного розрахунку інтеграла використовується функція NIntegrate. Основне призначення функції NIntegrate полягає у наближеному обчисленні визначених інтегралів: одномірних

NIntegrate[f,{x, xmin, xmax}],

повторних

NIntegrate[f,{x, xmin, xmax},{y, ymin, ymax}],

або багатомірних

NIntegrate[f, {xn, xnmin, xnmax},…,{x2, x2min, x2max},{x1, x1min, x2max}].

Її можна використовувати для обчислення інтегралів по скінченому відрізку дійсної осі, на нескінченному інтервалі дійсної осі, по скінченному або безмежному інтервалу в комплексній площині.

Зазвичай другий аргумент функції NIntegrate має вигляд короткого списку {x, xmin, xmax}. Але для підінтегральних виразів з особливостями в точках (x0, x1,..., xn) його можна задавати як {x, x0, x1,..., xn}. У цьому випадку виконується перевірка наявності особливості у кожній із внутрішніх точок xi. Якщо особливостей немає, то результат еквівалентний інтегралу в межах від x0 до xn. Числа x, x0, x1,..., xn можуть бути комплексними і тоді інтеграл представлятиме інтеграл за ламаною в комплексній площині. Також можна чисельно інтегрувати за контуром на комплексній площині.

Функція NIntegrate має опції, які управляють точністю обчислень.

AccuracyGoal визначає абсолютну похибку обчислень, використовуючи кількість значущих цифр. За замовчуванням AccuracyGoal → ∞. Ця опція не повинна використовуватися як критерій припинення обчислень. Коли значення інтеграла дорівнює нулю, функція NIntegrate повертає повідомлення про повільну збіжність алгоритму. У такій ситуації потрібно встановити опцію AccuracyGoal у значення, відмінне від встановленого за умовчанням, а критерієм зупинти обчислень буде абсолютна похибка.

PrecisionGoal визначає відносну похибку обчислень і визначається кількістю значущих цифр. За замовчуванням PrecisionGoal → ∞. PrecisionGoal->n визначає значення x похибки обчислень рівну $|x|\ 10^{-n}$. Коли включені опції AccuracyGoal->m і PrecisionGoal->n, то Mathematica намагається виконати обчислення величини x з похибкою не більше $10^{-m} + |x|\ 10^{-n}$.

Опція WorkingPrecision визначає кількість значущих цифр, що використовуються у внутрішніх обчисленнях; значення цієї опції, як правило, має бути більшим за значення опції AccuracyGoal. При WorkingPrecision->MachinePrecision приводятьсяо обчислень з процесорною точністю.

Функція NIntegrate виконує розбиття інтервалу інтегрування до того часу, поки помилка обчислень обраного нею методу не стане менше похибки, визначеної опціями AccuracyGoal і PrecisionGoal. Вона використовує адаптивні алгоритми з автоматичним вибором кроку інтегрування. Ці алгоритми самі, якщо потрібно, рекурсивно дроблять відрізки розбиття області інтегрування у місцях швидкої зміни функції.

Опція MinRecursion визначає кількість кроків дроблення відрізків розбиття, з якої починаються обчислення. За замовчуванням MinRecursion->0.

Опція MaxRecursion визначає максимальну кількість кроків подрібнення. MaxRecursion→n визначає, що рекурсивно можна дробити відрізки розбиття n разів. Рекурсивний/багаторазовий поділ виконується тільки на тих ділянках, де значення функції швидко змінюються і для досягнення необхідної точності довжина інтералів розбиття повинна бути зменшена.

Опція MaxPoints обмежує кількість точок розбиття інтервалу інтегрування.

Опція Exclusions дозволяє виключити з розгляду деякі спеціальні точки (криві або поверхні для кратних інтегралів) з області інтегрування.

У функції NIntegrate реалізовано багато різних алгоритмів чисельного інтегрування. Зазвичай NIntegrate сама вибирає метод. Однак задати конкретний метод можна за допомогою опції Method.

Детальніше про NIntegrate: https://reference.wolfram.com/language/ref/NIntegrate.html?q=NIntegrate

Детальніше про методи чесельного інтегрування в Mathematica: https://reference.wolfram.com/language/tutorial/NIntegrateIntroduction.html

7. РЯДИ, ЛАНЦЮГОВІ ДРОБИ

Ситуація з нескінченними рядами і добутками нагадує ситуацію з інтегруванням на нескінченному інтервалі.

Якщо ряд сходиться досить швидко, то можна взяти досить великий індекс N в якості верхньої межі, обчислити кінцеву суму $S_N = \sum_{n=1}^{N} a_n$. Щоб оцінити похибку цієї відповіді, слід збільшити N в c раз: N → c N, і порівняти відповіді. Додаток $\sum_{n=N+1}^{c\cdot N} a_n$ розглядається як похибка для поточної відповіді $S_{c\cdot N}$. Якщо точність недостатня, то слід знову збільшити верхню межу підсумовування в c раз, і так, поки не буде досягнута потрібна точність. При цьому, як правило, можна оцінити залишковий член $\sum_{n=N+1}^{\infty} a_n \approx \int_N^{\infty} a(x)dx$ по асимптотиці членів ряду. Якщо цей простий алгоритм забезпечує задовільну швидкість збіжності, то на ньому можна зупинитися. На жаль, зазвичай швидкість виявляється недостатньою.

Якщо збіжність повільна, то можна спробувати вирахувати із членів ряду їх головну асимптотику. Ряд, відповідний асимптотиці, нерідко можна підсумувати аналітично, а залишковий ряд буде збігатися набагато швидше, ніж вихідний.

Такі основні універсальні рецепти для нескінченних рядів. Всі інші рецепти відповідають тим чи іншим окремим випадкам.

7.1. Квазігеометричний ряд

Квазігеометричний ряд - це ряд, у якого асимптотика n-го члена є $a_n \cong bq^n$. Збіжність такого ряду лінійна, тобто кількість вірних знаків пропорційна числу операцій. Дійсно, для геометричної прогресії

$$S_N = \sum_{n=1}^{N} a_n = \frac{b(q^{N+1} - 1)}{q - 1},$$

тому залишковий член дорівнює $bq^{N+1}/(1-q)$. Проте, збіжність можна прискорити додатково. Для цього слід обчислити

$$S'_N = S_N - \frac{(S_N - S_{N-1})^2}{S_N - 2S_{N-1} + S_{N-2}}.$$

Для справжньої геометричної прогресії отримаємо

$$S'_N = \frac{b(q^{N+1}-1)}{q-1} - \frac{b}{q-1}\frac{(q^{N+1}-q^N)^2}{q^{N+1}-2q^N+q^{N-1}} = \\ = \frac{b(q^{N+1}-1)}{q-1} - \frac{b}{q-1}q^{N+1} = \frac{-b}{q-1}.$$

що збігається з точною відповіддю для S_∞ при довільному N.

Для квазігеометричної прогресії величина S'_N вважається поточною відповіддю для всього ряду, а різниця між S'_N і S'_{N-1} - оцінка похибки цієї відповіді. Для геометричної прогресії при оцінці залишкового члена немає необхідності збільшувати верхню межу підсумовування оскільки залишковий член геометричної прогресії порядку $a_{N+1}/(q-1)$.

7.2. Знакопостійний ряд

Знакопостійні ряди набагато гірші знакозмінних. Досить порівняти ряди $\sum(-1)^n/n^2$ і $\sum 1/n^2$. У першому випадку кінцева сума $S_N = \sum_{n=1}^{N} a_n$ наближає шукану відповідь S_∞ з похибкою порядку поточного члена, тобто порядку $1/n^2$, що цілком прийнятно при чисельному розрахунку. У другому випадку залишковий член $\sum_{n=N+1}^{\infty} a_n \approx \int_N^\infty a(x)dx = 1/N$, так що досягти задовільної точності за скінченний час неможливо. Більш того, при підсумовуванні великої кількості доданків накопичується помилка округлення, так що точної відповіді не буде навіть при дуже тривалому розрахунку.

Якщо вирахувати асимптотику не вдається, то можна застосувати інтерполяцію. Саме, обчислювати суми $S_{N1}, S_{N2}, S_{N3}, \ldots$ (тут $N_i=N_0c^i$, а c~1.5-2). На кожному k-му кроці проводити поліноміальну інтерполяцію по k точках (x_i, y_i), i = 1,…, k (тут x_i = 1/ N_i, y_i =S_{Ni}) в точку x = 0, відповідну нескінченій верхній межі підсумовування. Тоді результат інтерполяції S'_k вважається поточною відповіддю, а різниця $|S'_k - S'_{k-1}|$ є оцінкою її похибки. На жаль, для рядів інтерполяційний алгоритм далеко не такий ефективний, як для інтегралів, проте нерідко він дає помітне прискорення.

7.3. Знакозмінний ряд

Зазвичай знакозмінний ряд, особливо якщо абсолютна величина членів ряду досить швидко спадає, сходиться швидко. Більш того, якщо абсолютна величина членів ряду зменшується монотонно, то відомий точний інтервал, в якому знаходиться відповідь: $S_\infty \in [S_{N-1}, S_N]$, Однак збіжність знакозмінного ряду можна прискорити додатково, якщо скористатися алгоритмом Ейлера:

$$\sum_{m=1}^{\infty}(-)^m a_m = \sum_{m=1}^{k-1}(-)^m a_m + \sum_{r=1}^{\infty}\frac{(-)^{k+r}}{2^{r+1}}\Delta_{k,r}.$$

Тут $\Delta_{k,r}$- різницева похідна вперед:

$$\Delta_{k,0} = a_k \quad \Delta_{k,1} = a_{k+1} - a_k \quad \Delta_{k,2} = a_{k+2} - 2a_{k+1} + a_k \quad \cdots.$$

Явна формула має вигляд:

$$\Delta_{k,r} = \sum_{p=0}^{r} C_r^p (-)^{r-p} a_{k+p},$$

що відповідає рекуррентній формулі

$$\Delta_{k,r} = \Delta_{k+1,r-1} - \Delta_{k,r-1}.$$

Виникає природне запитання: заради чого замінювати один знакозмінний ряд іншим? Справа в тому, що член нового ряду $\Delta_{k,r}$ спадає при зростанні r швидше, ніж член старого ряду a_{k+r}. Дійсно, якщо асимптотика членів ряду a_k має вигляд 1/k, то $a_{k+r} \approx 1/r$, в той же час величини $\Delta_{k,r}$ зменшуються з ростом r набагато краще, ніж $1/r$. Питання про те, як саме вибрати точку k, в якій здійснюється перехід від обчислення ряду «в лоб» до обчислення ряду з різницевих похідних $\Delta_{k,r}$, обговорюється нижче.

Для реалізації алгоритму Ейлера на практиці слід застосувати наступний алгоритм.

Алгоритм

На k+t-му кроці, тобто коли вираховані всі члени ряду аж до a_{k+t}, ми зберігаємо не різницеві похідні «вперед»:

$$\Delta_{k,0} \quad \dots \quad \Delta_{k,t},$$

а, навпаки, різницеві похідні «назад», поділені на відповідні степені двійки:

$$\Delta_{k+t,0} = a_{k+t}$$
$$\Delta_{k+t,-1}/2 = (a_{k+t} - a_{k+t-1})/2$$
$$\Delta_{k+t,-2}/4 = (a_{k+t} - 2a_{k+t-1} + a_{k+t-2})/4$$
$$\dots$$
$$\Delta_{k+t,-t}/2^t.$$

При цьому $\Delta_{k+t,-t} = \Delta_{k,t}$.

До цього моменту вже вирахований кінцевий відрізок ряду з формули Ейлера:

$$S_{k,t} = \sum_{m=1}^{\infty}(-)^m a_m = \sum_{m=1}^{k-1}(-)^m a_m + \sum_{r=1}^{t}\frac{(-)^{k+r}}{2^{r+1}}\Delta_{k,r}.$$

На наступному кроці обчислюється a_{k+t+1}. Після цього виконуються розрахунки:

$$\Delta_{k+t+1,0} = a_{k+t+1}$$
$$\Delta_{k+t+1,-1}/2 = (a_{k+t+1} - a_{k+t})/2 = (\Delta_{k+t+1,0} - \Delta_{k+t,0})/2$$
$$\Delta_{k+t+1,-2}/4 = (\Delta_{k+t+1,-1}/2 - \Delta_{k+t,-1}/2)/2$$
$$\Delta_{k+t+1,-3}/8 = (\Delta_{k+t+1,-2}/4 - \Delta_{k+t,-2}/4)/2$$
$$\dots$$
$$\Delta_{k+t+1,-t-1}/2^{t+1} = (\Delta_{k+t+1,-t}/2^t - \Delta_{k+t,-t}/2^t)/2.$$

Якщо виконується умова

$$|\Delta_{k+t+1,-t-1}|/2^{t+2} < |\Delta_{k+t,-t}|/2^{t+1},$$

тобто черговий доданок у формулі Ейлера менший за попередній, то продовжується обчислення ряду за різницевими похідним:

$$S_{k,t} \to S_{k,t+1} = S_{k,t} + (-)^{k+t+1}\Delta_{k+t+1,-t-1}/2^{t+2}$$

із запам'ятовуванням нової різницевої похідної

$$\Delta_{k+t+1,-t-1}/2^{t+1}.$$

В іншому випадку робимо висновок, що точка обрана невдало і її слід збільшити на одиницю:

$$S_{k,t} \to S_{k+1,t} = S_{k,t} + (-)^{k+t+1}\Delta_{k+t+1,-t-1}/2^{t+1}.$$

В цьому випадку різницева похідна $\Delta_{k+t+1,-t-t}/2^{t+1}$ виявляється «зайвою», тому вона не запам'ятовується.

Таким чином, в алгоритмі реалізований автоматичний вибір точки переходу k. Алгоритм Ейлера дійсно може помітно прискорити збіжність знакозмінного ряду.

7.4. Ланцюгові дроби

Ланцюговий дріб - це конструкція такого вигляду:

$$R_n = \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots \cfrac{a_{n-1}}{b_{n-1} + \cfrac{a_n}{b_n}}}}}$$

Зазвичай ланцюговий дріб умовно записують у вигляді

$$R_n = \frac{a_1}{b_1 +} \frac{a_2}{b_2 +} \frac{a_3}{b_3 +} \cdots \frac{a_n}{b_n}$$

Як правило, ланцюгові дроби виникають при обчисленні тих чи інших вищих трансцендентних функцій. Якщо функція може бути представлена у вигляді ланцюгового дробу, то цим обов'язково треба скористатися, оскільки нескінченні ланцюгові дроби збігаються набагато швидше, ніж ряди.

Звісно, дроби не слід обчислювати відповідно до їх явних виразів. Пряме обчислення ланцюгового дробу не допускає рекурсії: якщо додати до b_n доданок a_{n+1}/b_{n+1}, то весь дріб доведеться перераховувати заново.

Правильна рекурсивна процедура така:

$$A_n = b_n A_{n-1} + a_n A_{n-2},$$
$$B_n = b_n B_{n-1} + a_n B_{n-2},$$
$$R_n = \frac{A_n}{B_n},$$

Щоб почати цю рекурсію, слід покласти

$$A_0 = 0, \quad B_0 = 1, \quad A_1 = a_1, \quad B_1 = b_1.$$

Єдина тонкість полягає в тому, що в цьому алгоритмі є небезпека переповнення: при скінченому R_n величини A_n і B_n можуть одночасно спрямуватися до нескінченності, або до нуля. Тому необхідно перевіряти їх абсолютні величини на предмет переповнення (наприклад, чи не перевищують вони 10^{50}) і «недоповнення» (не стають вони менше ніж 10^{-50}) реєстру. У разі виходу за ці рамки слід поділити величини A_{n-1}, B_{n-1}, A_n і B_n на, наприклад, $|B_n|$.

7.5. Аналітичний та числовий розрахунок сум і рядів засобами бібліотеки sympy

Для символьного обчислення нескінченних і скінченних сум в sympy використовується функція summation:

summation(f, (i, imin, imax)),

яка обчислює суму виду $\sum_{i=i_{min}}^{i_{max}} f(i)$. Межі в сумі можуть бути числами, символами або ж нескінченністю (summation(f, (i, imin, oo))). Якщо знайти результат підсумовування в символьній формі не вдається, sympy повертає оператор підсумовування Sum (), який також можна використовувати для розрахунку метод doit ().

Розклад функції (виразу) f(x) в ряд в околиці точки x = x0 до величин порядку $(x-x0)^n$ включно здійснюється функцією

series(expr [, x = None, x0 = 0, n = 6, ...])

та методом

f(x).series(x [, x0 = 0, n = 6]).

Аргументи x0 та n не обов'язкові. Якщо вони не вказані, то за замовчуванням x0 = 0, n = 6. При застосуванні функції series() або методу f(x).series результат виводиться у вигляді виразу з залишковим членом, який можна прибрати використавши метод

expr.series(...).removeO().

Функція series() також може використовувати для розкладу полінома за ступенями (x-a) при будь-якому a.

Отримані ряди можна диференціювати та виконувати над ними деякі арифметичні операції.

Детальніше про суми та ряди у sympy: https://docs.sympy.org/latest/modules/series/index.html#contents

7.6. Аналітичний та числовий розрахунок сум і рядів засобами Mathematica

Для символьного обчислення скінченних та нескінченних сум у Mathematica застосовується функція Sum:

Sum [f, {i, imax}]

обчислюється сума виду $\sum_{i=0}^{\text{imax}} f$,

Sum[f,{i,imin,imax}]

сума обчислюється аналогічно до попереднього, але починаючи з початкового значення i=imin,

Sum[f,{i,imin,imax,di}]

задається крок di зміни i,

Sum[f,{i,imin,imax},{j,jmin,jmax}…]

розраховується n-кратна сума,

Sum[f,i]

розраховується невизначена сума.

Mathematica вміє обхислювати ряди (суми з нескінченними межами підсумовування). Якщо результат підсумовування у символьній формі отримати не вдається, то Mathematica повертає відповідь у необчислюваній формі. Однак, для таких рядів можна розрахувати наближене значення суми за допомогою функції

NSum[f,{i,imin,imax}]

NSum[f,{i,imin,imax,di}]

Функція NSum має опції, які управляють точністю обчислень. AccuracyGoal визначає абсолютну похибку обчислень, використовуючи кількість значущих цифр. PrecisionGoal→n визначає значення x похибки обчислення рівну $|x|\ 10^{-n}$. Опція WorkingPrecision визначає кількість значущих цифр, що використовуються у внутрішніх

обчисленнях. При WorkingPrecision→ MachinePrecision обчислення здійснюються із процесорною точністю.

За замовчуванням функція NSum підсумовує перші 15 членів, перш ніж переходить до апроксимації залишкового члена ряду. Проте це не завжди коректно. Щоб підсумувати більшу кількість елементів, використовується опція NSumTerms.

За замовчуванням функція NSum перевіряє збіжність ряду. Якщо відомо, що ряд збігається, то відключивши перевірку збіжності опцією VerifyConvergence→False можна прискорити обчислення. Якщо ряд не збігається, то функція NSum як результат повертає ComplexInfinity. Для виведення на екран умов, за яких результат сумування вірний, слід використовувати опцію GenerateConditions→True.

Функція

Series[f,{x,x0,n}]

виконує розклад в ряд Телора функції f в околі точки x0 з точністю до $(x\text{-}x0)^n$. З отриманими рядами можна виконувати звичайні арифметичні операції.

Функцію Series також можна використовувати для перерозкладання полінома за степенями x- $x0$ при будь-якому $x0$. Функція CoefficientList [poly, var] повертає список коефіцієнтів полінома poly при степенях змінної var. Функція SeriesCoefficient[series,n] повертає коефіцієнт при n-тому члені степеневого ряду.

У пакеті Mathematica є декілька функцій, що виконують розклад в ряд Фур'є:

FourierTrigSeries[expr, t, n]

розкладає вираз expr, визначений на відрізку [-π, π], що залежить від змінної t, у тригонометричний ряд Фур'є порядку n. Опція Assumptions уточнює діапазон зміни символьних змінних, що фігурують у виразі expr.

Функції FourierCosSeries та FourierSinSeries здійснюють розклад за косинусами та синусами відповідно. Функція FourierSeries використовується для побудови комплексних рядів Фур'є.

Функція FourierCoefficient повертає n-й коефіцієнт розкладу функції в ряд Фур'є. Аналогічне призначення мають функції FourierSinCoefficient та FourierCosCoefficient.

Розрахунок асимптотичної апроксимації суми здійснює функція AsymptoticSum.

8. СИСТЕМИ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ

8.1. Метод Гауса з вибором головного елемента

Якщо потрібно просто розв'язати систему лінійних рівнянь (СЛР): $\hat{A}\vec{x} = \vec{b}$ для однієї або декількох правих частин $\overrightarrow{b_i}$, то найпростішим і досить швидким алгоритмом є тріангуляція. Якщо правих частин кілька (i = 1 ... m), то весь набір СЛР можна записати як $\widehat{\mathrm{A}}\,\hat{\mathrm{x}} = \hat{b}$, де матриці $\hat{\mathrm{x}}$ і $\hat{b}$ маю розмірність n×m.

Алгоритм

Спочатку для 1-го стовпця матриці A шукаємо максимальний по модулю елемент, нехай це $a_{k,1}$.

Міняємо місцями 1-й і k-й рядки матриць A і b.

Ділимо 1-й рядок матриць A і b на a_{11}.

Для всіх k = 2. . . n:

віднімаємо від k-го рядка матриці A 1-й рядок, помножений на a_{k1}, аналогічно, від k-го рядка матриці b віднімаємо 1-й рядок, також помножений на a_{k1}.

У результаті вся поддіагональная частина 1-го стовпчика зануляється, а на діагоналі стоїть одиниця.

Після цього забуваємо про 1-й рядок матриць A і b і про 1-й стовпець матриці A. Повторюємо викладений алгоритм для урізаної матриці (2 ... n) x (2 ... n). І так продовжуємо до кінця матриці. Результатом буде верхня трикутна матриця з одиницями на діагоналі:

$$\begin{pmatrix} 1 & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ 0 & 1 & a_{23} & a_{24} & \dots & a_{2n} \\ 0 & 0 & 1 & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 & a_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Тепер $\hat{x}$ легко знайти зворотною підстановкою: $x_{nk} = b_{nk}$ (для всіх k=1…m); після чого $x_{ik} = b_{ik} - \sum_{j=i+1}^{n} a_{ij} x_{jk}$, де i=n-1, n-2,…,1 (також для всіх k = 1 ... m).

Для несингулярних матриць цей алгоритм не дає збоїв. З точки зору чисельного розрахунку, сингулярна матриця – це матриця, у власних значеннях якої занадто великий перепад, що робить помилки округлення при відніманні двох «великих» власних значень більше, ніж «маленьке» власне значення. Число операцій при тріангуляції пропорційно n^3, що, на жаль, дуже багато.

Зауважимо, що при тріангуляції вираховується детермінант матриці A, який є добутком тих чисел $a_{i,i}$, на які ми ділили, помножене на $(-1)^s$, де s - число фактично виконаних перестановок.

Приклад розв'язання методом Гауса (без вибора головного елемента) системи алгебраїчних рівнянь

$$\begin{cases} 5x_1 + x_2 + 2x_3 = 10 \\ 6x_1 + 18x_2 + 6x_3 = 54 \\ 10x_1 + 20x_2 + 402x_3 = 160 \end{cases}$$

```
def solveGaus(A, b):
  """Метод Гауса"""
   n = len(b)
   x = zeros(n)
   for k in range(n):
      f = A[k,k]
      b[k] = b[k] / f
      for i in range(k, n):
         A[k, i] = A[k, i] / f
      for i in range(k+1, n):
         f = A[i, k]
         for j in range(k, n):
            A[i, j] = A[i, j] - A[k, j] * f
         b[i] = b[i] - b[k] * f
```

```
    x[n-1] = b[n-1]
    for i in range(n-2, -1, -1):
        sum = 0
        for j in range(i+1, n):
            sum +=   A[i, j] * x[j]
        x[i] = b[i] - sum

    return x

from numpy import array, zeros

A = array([[5., 1., 2.], [6., 18., 6.], [10., 20., 40.]])
b = array([10., 54., 160.])

solveGaus(A, b)

>>array([0.44444444, 1.86666667, 2.95555556])
```

8.2. LU-розклад

Цей спосіб дає можливість розв'язати СЛР для даної матриці один раз для будь-яких правих частин b.

Отже, матриця $\hat{A}$ представляється у вигляді добутку $\hat{A} = \hat{L}\hat{U}$:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ L_{21} & 1 & 0 & \dots & 0 \\ L_{31} & L_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & L_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & \dots & U_{1n} \\ 0 & U_{22} & U_{23} & \dots & U_{2n} \\ 0 & 0 & U_{33} & \dots & U_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & U_{nn} \end{pmatrix}$$

Очевидно, що будь-яка СЛР, записана в такому вигляді $Ax = LUx = y$, вже розв'язана: якщо ввести позначення z для розв'язку системи $Lz = y$, то x є розв'язком системи $Ux = z$. Обидві системи трикутні, тобто розв'язуютьсяться підстановкою: $Lz = y$ - прямою, а $Ux = z$ - зворотною. Дійсно, оскільки $\sum_{i=1}^{k-1} L_{ki}\, z_i + z_k = y_k$ то z_1 визначається відразу, а всі інші z_k виражаються через раніше знайдені $z_j (j > k)$.

Аналогічно, оскільки $\sum_{i=k+1}^{n} U_{ki}\, x_i + U_{kk} x_k = z_k$, то x_n визначається відразу, а інші x_k знаходяться через раніше знайдені $x_j (j > k)$.

Всі рівняння, що зв'язують матричні елементи A з матричними елементами L і U, діляться на три групи:

1. Наддіагональні елементи A:

$$A_{ik} = \sum_{j=1}^{i-1} L_{ij}\, U_{jk} + U_{ik} \qquad (k > i)$$

Це співвідношення дозволяє визначити елемент U_{ik} за умови, що відомі всі елементи L, що містяться в стовпцях 1... k-1, i, крім того, всі елементи k-го стовпця U, що лежать вище U_{ik}, тобто всі U_{jk} при $j<i$.

2. Діагональні елементи A:

$$A_{ii} = \sum_{j=1}^{i-1} L_{ij} U_{ji} + U_{ii}$$

Це співвідношення дозволяє визначити діагональний елементU_{ii} при тих же умовах, що і наведені вище.

3. Піддіагональні елементи A:

$$A_{ik} = \sum_{j=1}^{k-1} L_{ij} U_{jk} + L_{ik} U_{kk} \qquad (k < i)$$

Це співвідношення дозволяє визначити елемент L_{ik} за умови, що вже відомі всі елементи U_{jk} (при $j \le k$) для даного стовпця, а також всі елементи L, що містяться в стовпцях 1... k - 1.

З пунктів 1-3 випливає, що при використанні наведених співвідношень спочатку для 1-го стовпця для елементів $a_{11}, a_{21}\ldots a_{n1}$ в зазначеному порядку, далі для другого стовпця в тому ж порядку $a_{12}, a_{22}\ldots a_{n2}$, і так далі, аж до n-го стовпця в порядку $a_{1n}, a_{2n}\ldots a_{nn}$, умова можливості їх розв'язання кожного разу буде виконуватися. Єдина модифікація, яку необхідно внести в цю послідовність дій, полягає в тому, що виникає необхідність переставляти рядки в матрицях. При визначенні елементів L_{ik} доводиться ділити на діагональні елементи U_{kk}, тобто переконатися, що не виникає ділення на нуль. Ясно, що на будь-якому етапі маніпуляцій можемо переставити рядки в A і в L.

Порівнюючи пункти 2 і 3 видно, що в обох випадках потрібно обчислити абсолютно однотипні суми: $A_{ik} - \sum_{j=1}^{k-1} L_{ij} U_{jk}$. При i = k отримаємо просто U_{kk}, а при i > k результат треба розділити на U_{kk}, що дасть L_{ik}. Отже, при визначенні діагонального елемента U_{kk} для k-го стовпця треба почати з того, що обчислити всі ці суми для $k \le i \le n$ і вибрати найбільшу за абсолютною величиною (скажімо, при $i = i_0$). Це і буде майбутній діагональний елемент U_{kk}. Щоб він потрапив на діагональ, треба переставити рядок i_0 з k-м рядком в матрицях A і L. Така перестановка нічого не зіпсує. Після перестановки, в результаті якої U_{kk} потрапить на діагональ, всі інші обчислені суми треба на нього поділити, обчисливши тим самим L_{ik}. При цьому необхідно запам'ятати, яка саме перестановка проводилася для k-го стовпчика.

Результатом усіх цих маніпуляцій з урахуванням перестановок буде тотожність

$$P_n P_{n-1} \ldots P_2 P_1 A = LU$$

де $P_к$ - це оператор перестановки i_0-го і k-го рядків, виконаної при роботі з k-м стовпцем (якщо $i_0 = k$, то $P_к = E$).

Перестановки рядків в A приведуть лише до необхідності переставити в тому ж порядку і таким же чином елементи у правій частині СЛР: $LUx = P_n P_{n-1}\ldots P_2 P_1 y$. Обидві матриці L і U поміщаються в один масив.

8.3. Тридіагональні системи

В окремому випадку тридіагональної матриці LU розклад приводить до лінійного за розміром матриці n алгоритму розв'язання СЛР. Дійсно, тридіагональную матрицю можна розкласти як

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \ldots & 0 \\ a_2 & b_2 & c_2 & 0 & \ldots & 0 \\ 0 & a_3 & b_3 & c_3 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & b_n \end{pmatrix} = \begin{pmatrix} v_1 & 0 & 0 & 0 & \ldots & 0 \\ u_2 & v_2 & 0 & 0 & \ldots & 0 \\ 0 & u_3 & v_3 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & v_n \end{pmatrix} \begin{pmatrix} 1 & w_1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & w_2 & 0 & \ldots & 0 \\ 0 & 0 & 1 & w_3 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \end{pmatrix}$$

При цьому як процедура розкладання, так і пряма і зворотна підстановки потребують числа операцій, пропорційного $O(n^1)$.

Співвідношення, що пов'язують a_i, b_i, c_i з u_i, v_i, w_i, мають вигляд:

$$b_k = u_k w_{k-1} + v_k \qquad a_k = u_k \qquad c_k = v_k * w_k$$

(тут $w_0 = 0$). Вважаючи $Lz=y$, $Ux=z$, отримаємо

$$u_k z_{k-1} + v_k z_k = y_k \qquad x_k + w_k x_{k+1} = z_k$$

(тут $z_0 = 0$ і $x_{n+1}=0$).

Видно, що процедури LU-розкладу і прямої підстановки (обчислення z) можна проводити зверху вниз:

$$v_k = b_k - u_k w_{k-1} \qquad w_k = c_k / v_k \qquad u_k = a_k$$

$$z_k = (y_k - u_k z_{k-1}) / v_k.$$

Більш того, їх можна проводити одночасно, причому в результаті отримаємо значення z_i і w_i, необхідні для зворотної підстановки. Це - перший етап алгоритму. Значення u_i і v_i для зворотної підстановки абсолютно не потрібні, тобто можна й невідводити пам'ять під масиви u_i і v_i; достатньо обчислювати поточні значення u_i і v_i в ході прямої підстановки. Саме тому в матриці U, а не в матриці L, на діагональ одинички - це дозволяє запам'ятовувати на один масив менше.

Зворотна підстановка

$$x_k = z_k - w_k x_{k+1}$$

становить другий етап алгоритму.

Зауважимо, що у викладеній процедурі відбувається ділення на діагональні елементи v_k. Тому слід додати перевірку рівності v_k машинному нулю, тобто перевірку на (майже) рівність b_k і $u_k w_{k-1}$. Якщо v_k (майже) зануляються, то доведеться користуватися звичайною тріангуляцією.

8.4. Розв'язання систем лінійних алгебраїчних рівнянь засобами бібліотек Python

8.4.1 Чисельне розв'язання систем лінійних алгебраїчних рівнянь засобами бібліотеки numpy.linalg

У пакеті numpy.linalg реалізовані базові операції лінійної алгебри, де є функції розкладу матриці на множники. Для симетричної позитивно визначеної матриці A має місце розклад Холсцького, коли A = LL*, де L – нижньотрикутна матриця (функція cholesky()). Для загальних матриць використовується QT-розклад, коли A = QR, де Q - ортогональна матриця, а R - верхньотрикутна матриця (функція qr()). Сингулярний розклад матриці проводиться функцією svd().

Особлива увага приділяється розв'язанню систем лінійних рівнянь. Для обчислення зворотної матриці використається функція inv(). Обчислення псевдозворотних матриць здійснюється за допомогою pinv(). Функція solve() призначена на розв'язання системи рівнянь.

У методі найменших квадратів для матриці A розміром $n \times m (n \geq m)$ і правої частини b є вектор x (псевдорозв'язок), який мінімізує норму нев'язки r = b - Ax. В numpy.linalg для реалізації методу найменших квадратів використовується функція lstsq().

Детальніше про numpy.linalg: https://numpy.org/doc/stable/reference/routines.linalg.html

Приклад розв'язання за допомогою функції solve системи алгебраїчних рівнянь

$$\begin{cases} 5x_1 + x_2 + 2x_3 = 10 \\ 6x_1 + 18x_2 + 6x_3 = 54 \\ 10x_1 + 20x_2 + 402x_3 = 160 \end{cases}$$

```
from numpy import array
from numpy.linalg import solve

A = array([[5, 1, 2], [6, 18, 6], [10, 20, 40]])
b = array([10, 54, 160])

solve(A, b)
```

```
>>array([0.44444444, 1.86666667, 2.95555556])
```

Приклад розв'язання за допомогою функції lstsq системи алгебраїчних рівнянь

$$\begin{cases} 5x_1 + x_2 + 2x_3 = 10 \\ 6x_1 + 18x_2 + 6x_3 = 54 \\ 10x_1 + 20x_2 + 402x_3 = 160 \end{cases}$$

```
from numpy import array
from numpy.linalg import lstsq

A = array([[5, 1, 2], [6, 18, 6], [10, 20, 40]])
b = array([10, 54, 160])

lstsq(A, b, rcond=-1)
```

```
>> (array([0.44444444, 1.86666667, 2.95555556]), array([], dtype=float64), 3, array([48.34268369, 13.0656916 , 4.27465043]))
```

8.4.2 Чисельне розв'язання систем лінійних алгебраїчних рівнянь засобами бібліотеки scipy.linalg

Пакет scipy надає широкі можливості розв'язання задач лінійної алгебри. Модуль scipy.linalg містить функції для обчислення норм матриць та векторів, факторизації матриць, розв'язання систем лінійних рівнянь, дозволяє знаходити власні значення та власні вектори, обчислювати функції матриць. Для розв'язання задач лінійної алгебри з розрідженими матрицями, які особливо важливі при наукових обчисленнях, призначений модуль sparse.linalg.

Scipy, як і numpy, вміє працювати як з масивами, так і з матрицями. Під час задання матриць підтримується синтаксис MATLAB (ініціалізація за рядком).

Для обчислення визначника матриці є функція det(). Для знаходження норми матриці або вектора призначена функція norm().

Функція

solve()

призначена для розв'язання лінійних алгебраїчних рівнянь. Розв'язання систем рівнянь зі стрічковою матрицею здійснюється функцією

solve_banded()

Для більш загальних матриць у модулі scipy.linalg є функції LU-розкладу. Більше того, для роботи з будь-якими невиродженими матрицями реалізований загальний розклад A = PLU — функція

1u()

Тут L - нижня трикутна матриця з діагональними елементами, рівними одиниці, U - верхня трикутна матриця, P - матриця перестановок Інші можливості LU-розкладу реалізовані у функціях lu_factor(), lu.solve().

Є також функції інших розкладів матриць:

• розклад Холського - функції

cholesky(), cholesky_banded(), cho_factor(), cho_solve();

• QR-розклад - функція

qr();

• сингулярний розклад - функції

svd(), svdvals(), diagsvd(), diagsvd();

• розкладання Шура – функції

schur(), rsf2csf().

Реалізація методу найменших квадратів реалізована функцією

lstsq()

Замість функції lstsq() можна використовувати pinv() або pinvy2(), які дозволяють знайти псевдорозв’язки системи лінійних рівнянь.

Детальніше про scipy.linalg: https://docs.scipy.org/doc/scipy/reference/linalg.html

8.5. Розв’язання систем лінійних алгебраїчних рівнянь засобами Mathematica

З розділу 4 «Пошук коренів нелінійних рівнянь» вже відомо, що для символьного розв'язання рівнянь або систем рівнянь в Mathematica використовуються функції

Solve[expr, var]

або

Roots[lhs==rhs,var].

Детальніше про Solve: https://reference.wolfram.com/language/ref/Solve.html?q=Solve

Детальніше про Roots: https://reference.wolfram.com/language/ref/Roots.html?q=Roots

Функція

NSolve[expr, var]

знаходить наближено всі корені рівняння або системи рівнянь. Детальніше про NSolve: https://reference.wolfram.com/language/ref/NSolve.html?q=NSolve

Для знаходження розв’язків лінійних систем рівнянь також можна використовувати функцію

LinearSolve[M, b],

яка знаходить вектор x, що відповідає матричному рівнянню M·x=b, де M – матриця коефіцієнтів, b – вектор вільних членів. Аргумент b може бути вектором або матрицею. Якщо b - матриця, то визначаються розв’язки, що відповідають кожному стовпцю цієї матриці. Детальніше про LinearSolve: https://reference.wolfram.com/language/ref/LinearSolve.html

Не завжди функції Mathematica можуть знайти точні розв’язки. Тоді їм потрібно допомогти, заздалегідь виконавши деякі перетворення рівняння або системи рівнянь.

Функція

Reduce[eqns,vars]

спрощує рівняння, задані списком eqns, та намагається розв’язати їх відносно змінних, заданих списком vars. Рівняння, що повертаються функцією Reduce, еквівалентні початковим та містять усі можливі розв’язки. Результуючі рівняння поєднуються з використанням операцій логічного 'Та' (&&) або логічного 'АБО' (||).

У форматі

Reduce[eqns,vars, elims]

функція спрощує рівняння та намагається виключити невідомі elims. Детальніше про Reduce: https://reference.wolfram.com/language/ref/Reduce.html?q=Reduce

Функція

Eliminate[eqns, vars]

спрощує систему рівнянь шляхом виключення невідомих vars. Вона працює добре з лінійними та поліноміальними рівняннями. Детальніше про Eliminate: https://reference.wolfram.com/language/ref/Eliminate.html?q=Eliminate

Функція

SolveAlways[eqns, vars]

повертає список значень параметрів, підстановка яких перетворює рівняння на тотожність відносно змінних vars. Детальніше про Solve: https://reference.wolfram.com/language/ref/Eliminate.html?q=Eliminate

Література

1. Калиткин Н.Н. Численные методы. М., Наука, 1978.
2. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. М.: Наука, 1973, 1987.
3. Самарский А. А. Введение в численные методы, М.: «Наука», 1982, 1987.
4. Самарский А. А., Гулин А. В. Численные методы. - М. : Наука, 1989.
5. Гулд Х., Тобочник Я. Компьютерное моделирование в физике. Ч. 1. М. : Мир, 1990.
6. Гулд Х., Тобочник Я. Компьютерное моделирование в физике. Ч. 2. М. : Мир, 1990.
7. Мудров А.Е. Численные методы для пэвм на языках бейсик, фортран и паскаль. - Томск: Раско, 1991.
8. Лященко М. Я., Головань М. С. Чисельні методи. Київ : Либідь, 1996.
9. Ильина В.А., Силаев П.К. Численные методы для физиков-теоретиков. Т1. – Москва-Ижевск: Институт компьютерных исследований, 2003. – 132 с.
10. Ильина В.А., Силаев П.К. Численные методы для физиков-теоретиков. Т2. – Москва-Ижевск: Институт компьютерных исследований, 2004. – 118 с.
11. Цегелик Г.Г. Чисельні методи. Львів. нац. ун-т ім. І.Франка, 2004.
12. Головацький В.А. Система комп’ютерної алгебри Mathematica 5. Чернівці, Рута, 2008.
13. Вабищевич П.Н. Численные методы: Вычислительный практикум. — М.: Книжный дом «Либроком», 2010.
14. Шахно С. М., Дудикевич А. Т., Левицька С. М. Практикум з чисельних методів. Львів : ЛНУ ім. Івана Франка, 2013.
15. Задачин В. М., Конюшенко І. Г. Чисельні методи. Харків: ХНЕУ ім. С.Кузнеця, 2014.
16. Доля П.Г.Введение в научный Python. Харків, 2016.
17. Доля П.Г., Антоненко Г.М. Розв'язання задач вищої математики на комп'ютері. Харків, 2017.
18. Хайбрахманов, С. А. Основы научных расчётов на языке программирования Python. Челябинск : Изд- во Челяб. гос. ун-та, 2019.

СУЧАСНІ МЕТОДИ АНАЛІТИЧНОГО І ЧИСЛОВОГО ПРОГРАМУВАННЯ В ЗАДАЧАХ ТЕОРЕТИЧНОЇ ФІЗИКИ

Методичні рекомендації

Укладач ***Сеті Юлія Олександрівна***

Відповідальний за випуск ***Ткач М.В.***