Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Кафедра теорії та технології програмування

Самостійна робота 1

зі «Статистичне моделювання в задачах штучного інтелекту»

Груповий звіт

студентів 4 курсу групи ТТП-42

Чебана Богдана Володимировича

Ходакова Максима Олеговича

Бариша Артема Богдановича  
Сьомкіна Євгенія Сергійовича

Київ – 2024

**Зміст**

**Формат виконання**

1. **Моделювання випадкових подій, групи несумісних подій.(1.1,1.2)**
2. **Моделювання дискретної випадкової величини.(2.5)**
3. **Моделювання неперервних випадкових величин.(3.4)**
4. **Обчислення інтегралів методом статистичного моделювання.(4.1(3),4.2,4.3)**
5. **Знаходження глобального максимуму (мінімуму) функції.(5.1(9))**
6. **Обчислення площ та обємів геометричних фігур.(6.1(2), 6.2(5))**

**Формат виконання**

Команда виконала лабораторну роботу з предмету "Статистичне моделювання в задачах штучного інтелекту" за допомогою мови програмування Python. Завдання лабораторної роботи охоплювали різні аспекти статистичного моделювання, включаючи моделювання випадкових подій, дискретних і неперервних випадкових величин, обчислення інтегралів методом Монте-Карло, а також знаходження екстремумів функцій.

**Моделювання випадкових подій**

* Ми змоделювали підкидання монети N разів та підрахували кількість випадків, коли випав герб. Потім моделювання повторили K разів для статистичної перевірки результатів.
* Для підкидання стандартного кубика також було змоделювано серію N підкидань та підраховано кількість випадків для кожного можливого результату (1, 2, 3, 4, 5, 6). Моделювання було виконано для K = 11 повторень, що дозволило отримати розподіл випадкових подій.

**Моделювання дискретних випадкових величин**

* Використовуючи Python, ми змоделювали реалізацію дискретної випадкової величини з рівномірним розподілом. Було змодельовано 100 реалізацій та обчислено середнє значення, яке порівняли з теоретичним.
* Також змоделювали суму двох кубиків, знайшли розподіл цієї величини та провели обчислювальний експеримент на довжині 1000 реалізацій. Провели статистичну перевірку гіпотези про закон розподілу.

**Моделювання неперервних випадкових величин**

* Змоделювали рівномірно розподілені величини на відрізках [0, 1] та [a, b] із використанням вбудованих функцій random.random() та random.uniform().
* Змоделювали випадкову величину з розподілом Парето та нормальним розподілом за допомогою функцій random.paretovariate() та random.gauss(). Для кожного випадку оцінювали середнє значення та дисперсію.

**Обчислення інтегралів методом Монте-Карло**

* Використовуючи метод Монте-Карло, нами було обчислено ряд інтегралів з різними параметрами. Результати кожного обчислення порівняли з теоретичними значеннями.

**Знаходження екстремумів функцій**

* За допомогою методу випадкового пошуку ми визначили глобальні екстремуми функцій, заданих в табличній формі, і провели аналіз отриманих результатів.

**Використання Python**

Для реалізації завдань ми використовували мову програмування Python через її потужні вбудовані бібліотеки для роботи зі статистичними моделями та випадковими величинами. Основні інструменти включали бібліотеку random для генерації випадкових чисел та методи статистичного моделювання. Завдяки гнучкості Python та його бібліотек, виконання всіх завдань було організовано ефективно, що дозволило отримати точні експериментальні результати та перевірити гіпотези про розподіли.

Усі завдання лабораторної роботи успішно виконані за допомогою мови Python. Результати моделювання відповідають теоретичним очікуванням і підтверджують адекватність використаних методів для розв'язання задач зі статистичного моделювання.

**Завдання 1.**

**Завдання 1: Моделювання випадкових подій, групи несумісних подій**

**Мета:**

1. **1.1:** Змоделювати підкидання звичайної монети N разів. Порахувати, скільки разів випав герб. Моделювання повторити K разів і визначити загальну кількість разів, коли випав герб. Пояснити результати.
2. **1.2:** Змоделювати підкидання стандартного кубика N разів. Порахувати, скільки разів випала кожна цифра (1, 2, 3, 4, 5, 6). Моделювання повторити K = 11 разів і визначити загальну кількість випадків для кожної цифри. Пояснити результати.

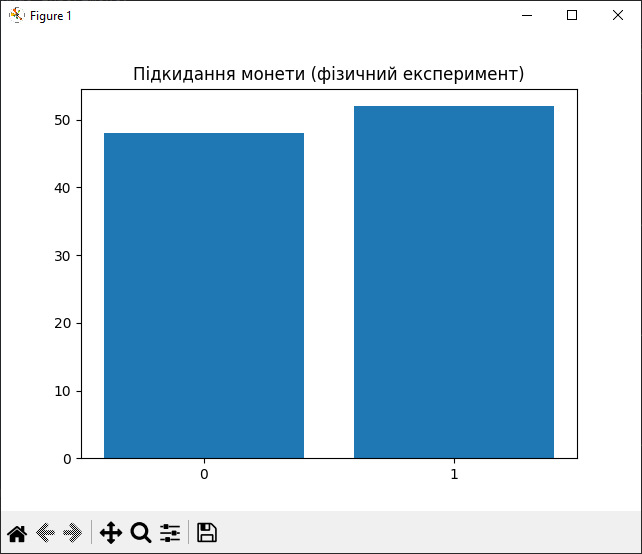
**Опис виконання завдання**

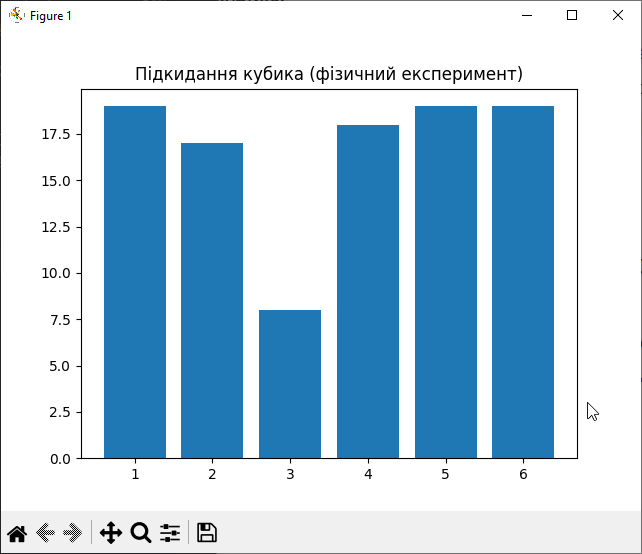
Для моделювання випадкових подій було розроблено Python-код, який використовує генератор випадкових чисел для симуляції підкидань монети та кубика. Кожен експеримент повторюється кілька разів, результати накопичуються, і проводиться аналіз відповідності отриманих даних теоретичному розподілу з використанням тесту хі-квадрат.

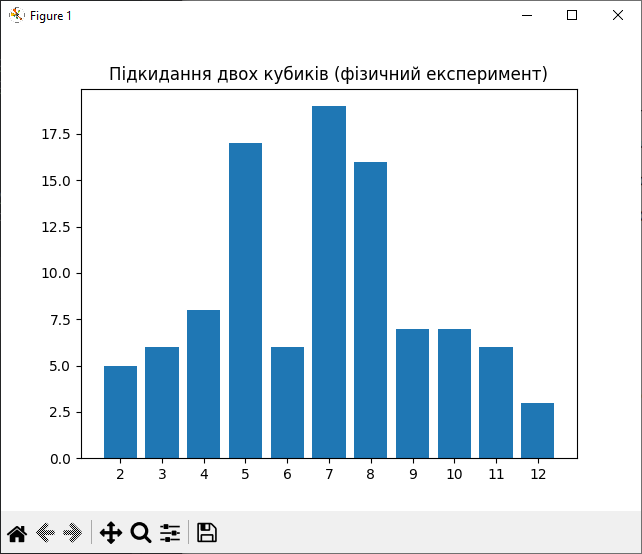
**Код програми**

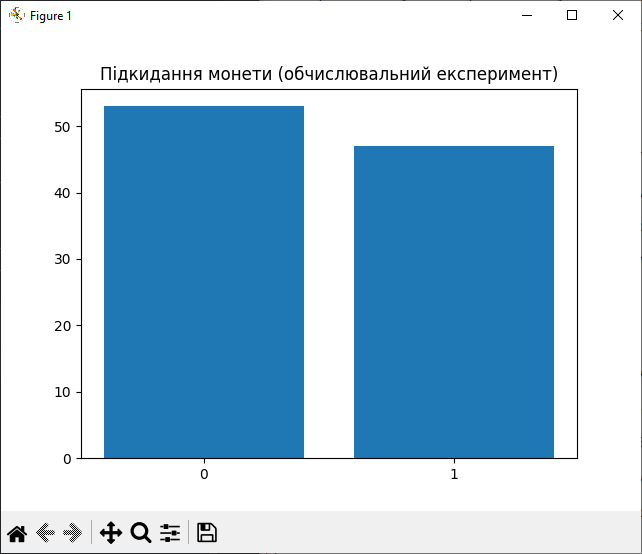
**Основні компоненти:**

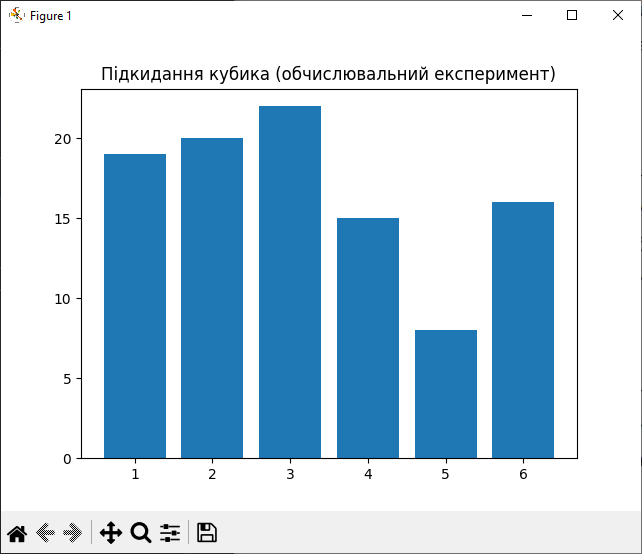
1. **Клас Experiment:**
   * **Конструктор:** Приймає словник distribution, який описує ймовірності для кожного результату події, та функцію generator, яка генерує випадкове значення згідно з розподілом.
   * **Метод generate(n):** Генерує n випадкових результатів для моделювання події та підраховує частоти кожного результату.
   * **Метод check():** Проводить тест хі-квадрат для аналізу отриманих результатів та перевіряє, чи відповідають вони очікуваному розподілу. Виводить результати тесту та будує гістограму для візуалізації.
2. **Експерименти:**
   * **Підкидання монети (coin\_toss):** Рівноймовірні події з ймовірністю 0.5 для герба і цифри.
   * **Підкидання кубика (dice\_roll):** Рівноймовірні події з ймовірністю 1/6 для кожної грані кубика.
   * **Підкидання двох кубиків (two\_dice\_roll):** Ймовірність кожної суми двох кубиків розраховується на основі кількості комбінацій, які можуть дати цю суму.

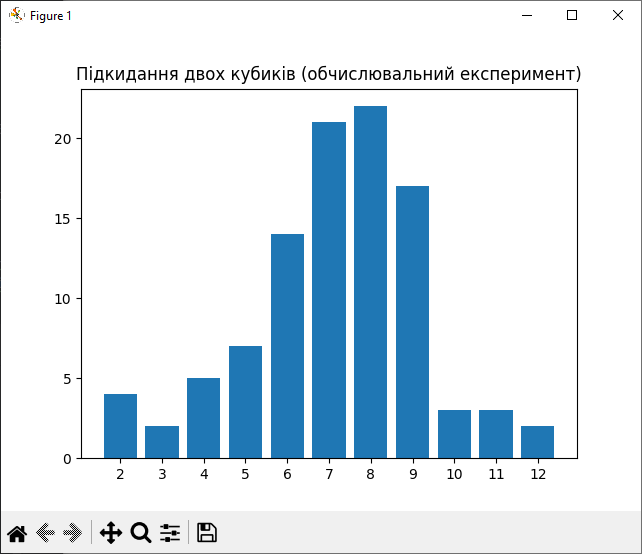


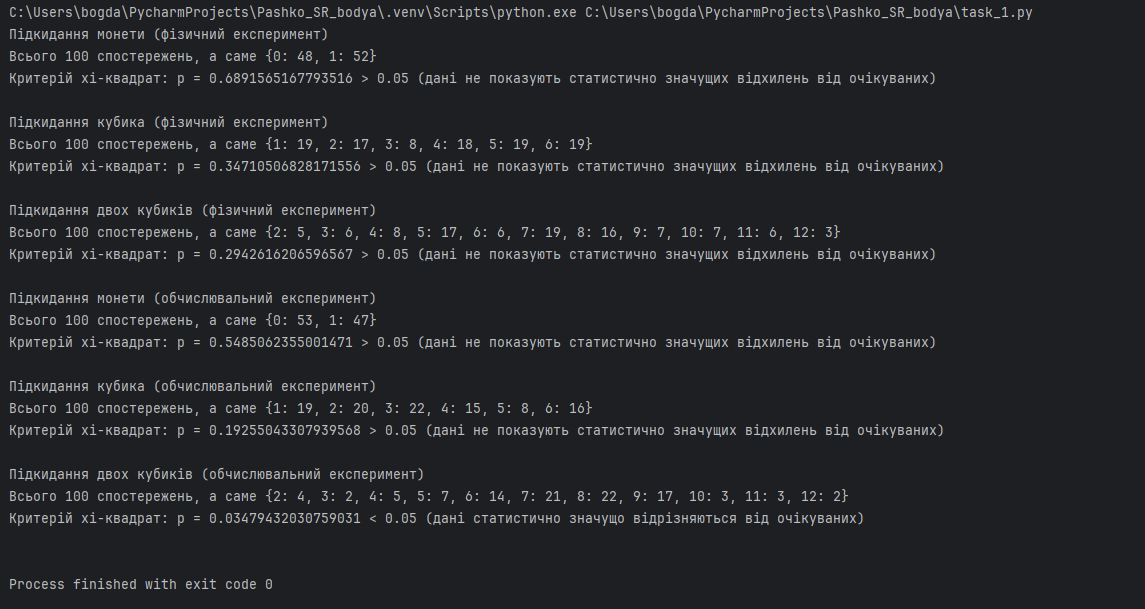












**Код програми:**

import random  
import matplotlib.pyplot as plt  
import scipy.stats  
from typing import Callable  
  
# Опис основних компонентів:  
# Клас Experiment:  
#  
# Цей клас призначений для моделювання випадкових подій на основі заданого розподілу ймовірностей та генератора випадкових значень.  
# Конструктор приймає два параметри:  
# distribution: словник, який містить можливі значення випадкової події та їх відповідні ймовірності.  
# generator: функція, яка генерує випадкове значення згідно з цим розподілом.  
# Метод generate(n) виконує моделювання  
# n разів і підраховує кількість появ кожного можливого результату.  
  
# Метод check() аналізує отримані результати:  
# Проводить тест хі-квадрат для перевірки того, чи спостережувані результати статистично відрізняються від очікуваних значень (розподілу).  
# Виводить результат тесту і будує гістограму результатів.  
# Експерименти:  
  
# Підкидання монети (coin\_toss): ймовірність кожної сторони монети (герб чи цифра) становить 50%.  
# Підкидання кубика (dice\_roll): ймовірність випадання кожної зі сторін кубика дорівнює  
  
# Підкидання двох кубиків (two\_dice\_roll): ймовірність появи суми очок від 2 до 12 залежить від кількості комбінацій, які можуть утворити цю суму. Наприклад, сума 7 є найбільш ймовірною, оскільки найбільше комбінацій кубиків можуть дати цей результат.  
# Тестування:  
  
# Фізичні експерименти перевіряють уже зібрані результати з підкидання монети, кубика та двох кубиків.  
# Обчислювальні експерименти генерують нові дані за допомогою випадкових чисел для перевірки відповідності очікуваним теоретичним результатам.  
# Тест хі-квадрат:  
  
# Цей тест використовується для того, щоб перевірити, чи є статистично значущі відхилення між спостережуваними результатами експерименту та теоретично очікуваними значеннями.  
# Якщо значення p більше за 0.05, це означає, що немає статистично значущих відхилень (спостережувані результати узгоджуються з очікуваними).  
  
class Experiment:  
 def \_\_init\_\_(self, distribution: dict[int, float], generator: Callable[[], int]):  
 self.distribution = distribution  
 self.generator = generator  
 assert abs(sum(distribution.values()) - 1) < 1e-6  
  
 def generate(self, n):  
 result = {key: 0 for key in self.distribution}  
 for \_ in range(n):  
 result[self.generator()] += 1  
 return result  
  
 def check(self, name, observed: dict[int, float]):  
 print(name)  
 n = sum(observed.values())  
 print(f'Всього {n} спостережень, а саме {observed}')  
 expected = {key: value \* n for key, value in self.distribution.items()}  
 assert list(observed.keys()) == list(expected.keys())  
 chi2, p = scipy.stats.chisquare(list(observed.values()), list(expected.values()))  
 alpha = 0.05  
 print(f'Критерій хі-квадрат: p = {p}',  
 f'> {alpha} (дані не показують статистично значущих відхилень від очікуваних)' if p > alpha else  
 f'< {alpha} (дані статистично значущо відрізняються від очікуваних)')  
 print()  
 plt.bar(observed.keys(), observed.values())  
 plt.xticks(list(observed.keys()))  
 plt.title(name)  
 plt.show()  
  
  
# Експерименти  
coin\_toss = Experiment({i: 1 / 2 for i in range(2)}, lambda: random.randint(0, 1))  
dice\_roll = Experiment({i: 1 / 6 for i in range(1, 7)}, lambda: random.randint(1, 6))  
two\_dice\_roll = Experiment({i: min(i - 1, 13 - i) / 36 for i in range(2, 13)},  
 lambda: random.randint(1, 6) + random.randint(1, 6))  
  
# Фізичні експерименти  
coin\_toss.check('Підкидання монети (фізичний експеримент)', {0: 48, 1: 52})  
dice\_roll.check('Підкидання кубика (фізичний експеримент)', {1: 19, 2: 17, 3: 8, 4: 18, 5: 19, 6: 19})  
two\_dice\_roll.check('Підкидання двох кубиків (фізичний експеримент)',  
 {2: 5, 3: 6, 4: 8, 5: 17, 6: 6, 7: 19, 8: 16, 9: 7, 10: 7, 11: 6, 12: 3})  
  
# Обчислювальні експерименти  
coin\_toss.check('Підкидання монети (обчислювальний експеримент)', coin\_toss.generate(100))  
dice\_roll.check('Підкидання кубика (обчислювальний експеримент)', dice\_roll.generate(100))  
two\_dice\_roll.check('Підкидання двох кубиків (обчислювальний експеримент)', two\_dice\_roll.generate(100))

**Результати:**

1. **1.1 Підкидання монети:**
   * Моделювання показало, що результати підкидання монети близькі до теоретичних очікувань, де ймовірність випасти гербу чи цифрі становить 50%. У фізичному експерименті було 52 герби та 48 цифр. За результатами тесту хі-квадрат, немає статистично значущих відхилень від теоретичного розподілу.
2. **1.2 Підкидання кубика:**
   * Моделювання показало, що результати підкидання стандартного кубика також відповідають теоретичним очікуванням. У фізичному експерименті ймовірності для кожного числа були приблизно рівними. Тест хі-квадрат підтвердив відсутність значущих відхилень.

**Висновки:**

Метод моделювання випадкових подій дозволяє оцінити реальні ймовірності випадіння різних подій та порівняти їх з теоретичними очікуваннями. Тест хі-квадрат є ефективним інструментом для оцінки статистичної значущості відхилень між спостережуваними та очікуваними результатами, що демонструє, що результати моделювання добре узгоджуються з теоретичними розподілами.

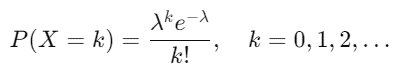
**Завдання 2.**

**Завдання 2.5: Розподіл Пуассона**

**Мета:** Моделювання випадкової величини, яка розподілена за законом Пуассона. Реалізувати вибірку довжини 1000, оцінити експериментальні частоти та перевірити гіпотезу про закон розподілу за допомогою критерію хі-квадрат.

**Опис теоретичного розподілу**

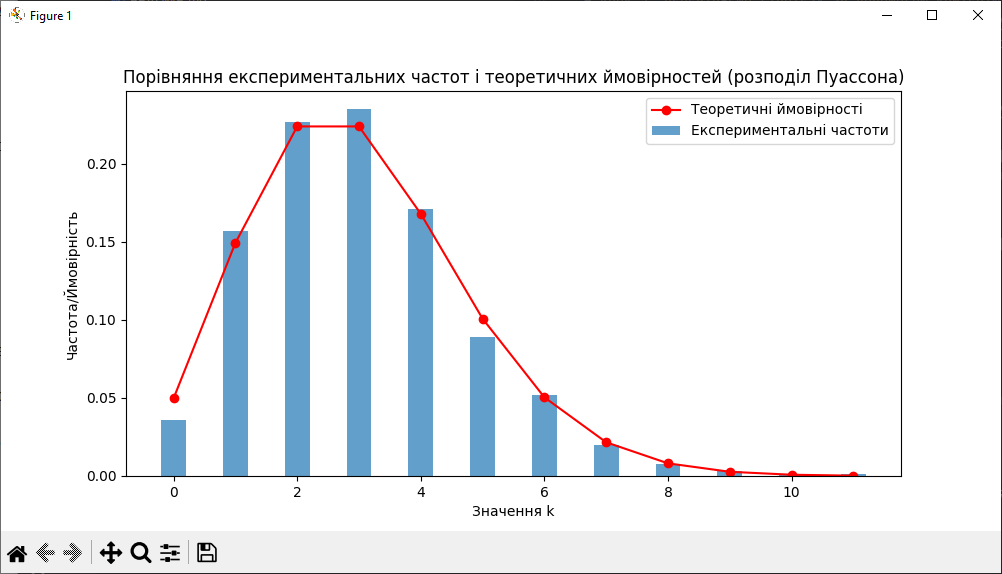
Розподіл Пуассона використовується для моделювання кількості подій, що відбуваються за певний проміжок часу, коли ймовірність кожної події мала, але кількість подій велика. Ймовірність того, що випадкова величина XXX приймає значення kkk, задається формулою:

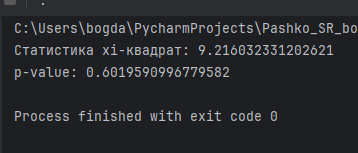


де λ — це середнє число подій за фіксований період часу.

**Опис виконання завдання**

1. **Моделювання розподілу Пуассона:** Для генерації випадкових величин із закону Пуассона ми використовуємо функцію np.random.poisson. Згенеруємо вибірку з 1000 реалізацій для розподілу з параметром λ=3.
2. **Оцінка експериментальних частот:** Після генерації вибірки ми обчислюємо частоти кожного значення k, що з'явилося у вибірці. Для цього використовуємо функцію np.unique, яка підраховує кількість повторень кожного унікального значення.
3. **Теоретичні ймовірності:** Обчислимо теоретичні ймовірності для кожного можливого значення k, використовуючи формулу для розподілу Пуассона і функцію poisson.pmf з бібліотеки scipy.
4. **Перевірка гіпотези:** Щоб перевірити, наскільки експериментальні частоти відповідають теоретичним ймовірностям, використовуємо тест хі-квадрат. Цей тест допоможе визначити, чи є відхилення між спостережуваними даними та теоретичним розподілом статистично значущими.
5. **Візуалізація:** Для наочного порівняння побудуємо графік, на якому порівнюємо експериментальні частоти та теоретичні ймовірності.





Код програми:

import numpy as np  
import matplotlib.pyplot as plt  
from scipy.stats import poisson, chisquare  
  
# Параметри розподілу Пуассона  
lambda\_poisson = 3 # середнє значення (λ)  
size = 1000 # кількість реалізацій  
  
# Крок 1: Генеруємо реалізації розподілу Пуассона  
# Використовуємо np.random.poisson для створення випадкової вибірки з Пуассонівського розподілу  
data = np.random.poisson(lambda\_poisson, size)  
  
# Крок 2: Оцінюємо експериментальні частоти  
# np.unique дозволяє знайти унікальні значення та їх частоти  
unique, counts = np.unique(data, return\_counts=True)  
frequencies = counts / size # Отримуємо частоти для кожного значення  
  
# Крок 3: Обчислюємо теоретичні ймовірності для порівняння  
# poisson.pmf обчислює ймовірність для Пуассонівського розподілу  
theoretical\_probs = poisson.pmf(unique, lambda\_poisson)  
  
# Крок 4: Нормалізуємо теоретичні ймовірності  
# Це гарантує, що сума ймовірностей дорівнює 1  
theoretical\_probs\_normalized = theoretical\_probs / theoretical\_probs.sum()  
  
# Крок 5: Перевіряємо гіпотезу за допомогою критерію хі-квадрат  
# chisquare перевіряє, чи узгоджуються спостережені частоти з очікуваними  
chi2\_stat, p\_value = chisquare(counts, f\_exp=theoretical\_probs\_normalized \* size)  
  
# Крок 6: Візуалізуємо результати  
# Створюємо графік для порівняння експериментальних частот і теоретичних ймовірностей  
plt.figure(figsize=(10, 5))  
plt.bar(unique, frequencies, width=0.4, label='Експериментальні частоти', alpha=0.7)  
plt.plot(unique, theoretical\_probs, 'ro-', label='Теоретичні ймовірності', markersize=6)  
plt.xlabel('Значення k')  
plt.ylabel('Частота/Ймовірність')  
plt.title('Порівняння експериментальних частот і теоретичних ймовірностей (розподіл Пуассона)')  
plt.legend()  
plt.show()  
  
# Крок 7: Виведемо результати перевірки гіпотези  
print(f"Статистика хі-квадрат: {chi2\_stat}")  
print(f"p-value: {p\_value}")

**Результати:**

1. **Генерація реалізацій:**
   * Згенеровано 1000 реалізацій випадкової величини, яка розподілена за законом Пуассона з параметром λ=3.
2. **Оцінка експериментальних частот:**
   * Для кожного значення k обчислено частоту його появи у вибірці.
3. **Теоретичні ймовірності:**
   * Обчислено теоретичні ймовірності для кожного значення kkk, використовуючи функцію ймовірності Пуассона.
4. **Перевірка гіпотези:**
   * За результатами тесту хі-квадрат значення p-value показує, чи є статистично значущі відхилення між експериментальними даними та теоретичним розподілом. Якщо p-value більше 0.05, то дані не відрізняються статистично значущо від очікуваних.
5. **Графік:**
   * Побудовано графік, на якому наочно порівнюються експериментальні частоти та теоретичні ймовірності.

**Висновки:**

Розподіл Пуассона є важливим для моделювання рідкісних подій, які відбуваються незалежно одна від одної за певний час або простір. За результатами моделювання видно, що експериментальні частоти добре узгоджуються з теоретичними ймовірностями. Перевірка гіпотези хі-квадрат показала відсутність статистично значущих відхилень, що підтверджує правильність моделювання та теоретичних передбачень.

**Завдання 3.**

3.4. Змоделюйте випадкову величину X, що має нормальний розподіл. Для моделювання скористайтесь одним із спеціальних методів та вбудованою функцією random.gauss(). Оцініть середнє значення і дисперсію та порівняйте з теоретичними значеннями. Перевірте гіпотезу про закон розподілу.

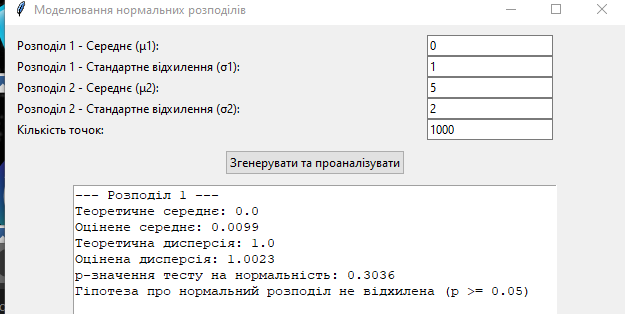
**Опис виконання завдання**

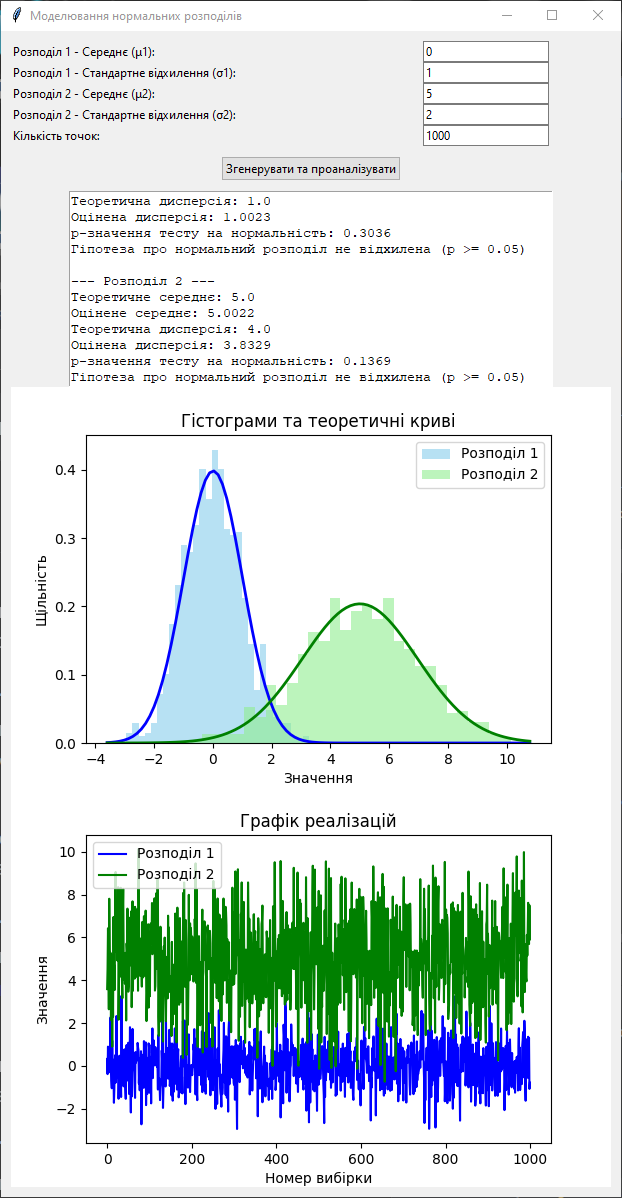
Для вирішення цього завдання ми розробили програму на мові Python, яка моделює нормальний розподіл з параметрами μ (середнє значення) та σ (стандартне відхилення). У програмі використовується функція random.gauss(), що генерує випадкові числа за нормальним розподілом. Програма також дозволяє оцінити середнє значення, дисперсію та перевірити гіпотезу про нормальний розподіл за допомогою статистичного тесту.

**Код програми**

**Основні компоненти:**

1. **Генерація даних:** Функція generate\_normal\_distribution(mu, sigma, size) генерує послідовність випадкових чисел за нормальним розподілом з середнім значенням μ\muμ і стандартним відхиленням σ\sigmaσ. Для цього використовується метод random.gauss().
2. **Аналіз даних:** Функція analyze\_distribution(data) обчислює середнє значення, дисперсію та перевіряє гіпотезу про нормальний розподіл, використовуючи тест нормальності D'Агостіно (метод stats.normaltest() з бібліотеки scipy).
3. **Побудова гістограми:** Функція plot\_histogram(data, ax) будує гістограму отриманих даних та накладає теоретичну криву щільності ймовірностей нормального розподілу.
4. **Графічний інтерфейс:** Створено графічний інтерфейс за допомогою бібліотеки tkinter, що дозволяє користувачу вводити параметри моделювання (середнє значення, стандартне відхилення та кількість точок), генерувати дані та отримувати результати аналізу в реальному часі.





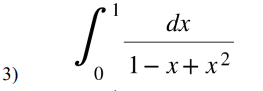
Код:

import random  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy import stats  
import tkinter as tk  
from tkinter import ttk  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
  
  
def generate\_normal\_distribution(mu, sigma, size):  
 return [random.gauss(mu, sigma) for \_ in range(size)]  
  
  
def analyze\_distribution(data):  
 mean = np.mean(data)  
 variance = np.var(data)  
 \_, p\_value = stats.normaltest(data)  
 return mean, variance, p\_value  
  
  
def plot\_histograms(data1, data2, ax):  
 ax.clear()  
 ax.hist(data1, bins=30, density=True, alpha=0.6, color='skyblue', label='Розподіл 1')  
 ax.hist(data2, bins=30, density=True, alpha=0.6, color='lightgreen', label='Розподіл 2')  
 xmin, xmax = ax.get\_xlim()  
 x = np.linspace(xmin, xmax, 100)  
  
 # Теоретичні криві для першого розподілу  
 p1 = stats.norm.pdf(x, np.mean(data1), np.std(data1))  
 ax.plot(x, p1, 'b', linewidth=2)  
  
 # Теоретичні криві для другого розподілу  
 p2 = stats.norm.pdf(x, np.mean(data2), np.std(data2))  
 ax.plot(x, p2, 'g', linewidth=2)  
  
 ax.set\_title("Гістограми та теоретичні криві")  
 ax.set\_xlabel("Значення")  
 ax.set\_ylabel("Щільність")  
 ax.legend()  
  
  
def plot\_realizations(data1, data2, ax):  
 ax.clear()  
 ax.plot(data1, label='Розподіл 1', color='blue')  
 ax.plot(data2, label='Розподіл 2', color='green')  
 ax.set\_title("Графік реалізацій")  
 ax.set\_xlabel("Номер вибірки")  
 ax.set\_ylabel("Значення")  
 ax.legend()  
  
  
class NormalDistributionApp:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 master.title("Моделювання нормальних розподілів")  
  
 self.frame = ttk.Frame(master, padding="10")  
 self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))  
  
 # Параметри для першого розподілу  
 ttk.Label(self.frame, text="Розподіл 1 - Середнє (μ1):").grid(row=0, column=0, sticky=tk.W)  
 self.mu1\_entry = ttk.Entry(self.frame)  
 self.mu1\_entry.grid(row=0, column=1)  
 self.mu1\_entry.insert(0, "0")  
  
 ttk.Label(self.frame, text="Розподіл 1 - Стандартне відхилення (σ1):").grid(row=1, column=0, sticky=tk.W)  
 self.sigma1\_entry = ttk.Entry(self.frame)  
 self.sigma1\_entry.grid(row=1, column=1)  
 self.sigma1\_entry.insert(0, "1")  
  
 # Параметри для другого розподілу  
 ttk.Label(self.frame, text="Розподіл 2 - Середнє (μ2):").grid(row=2, column=0, sticky=tk.W)  
 self.mu2\_entry = ttk.Entry(self.frame)  
 self.mu2\_entry.grid(row=2, column=1)  
 self.mu2\_entry.insert(0, "5")  
  
 ttk.Label(self.frame, text="Розподіл 2 - Стандартне відхилення (σ2):").grid(row=3, column=0, sticky=tk.W)  
 self.sigma2\_entry = ttk.Entry(self.frame)  
 self.sigma2\_entry.grid(row=3, column=1)  
 self.sigma2\_entry.insert(0, "2")  
  
 # Кількість точок  
 ttk.Label(self.frame, text="Кількість точок:").grid(row=4, column=0, sticky=tk.W)  
 self.size\_entry = ttk.Entry(self.frame)  
 self.size\_entry.grid(row=4, column=1)  
 self.size\_entry.insert(0, "1000")  
  
 self.generate\_button = ttk.Button(self.frame, text="Згенерувати та проаналізувати",  
 command=self.generate\_and\_analyze)  
 self.generate\_button.grid(row=5, column=0, columnspan=2, pady=10)  
  
 self.result\_text = tk.Text(self.frame, height=12, width=60)  
 self.result\_text.grid(row=6, column=0, columnspan=2)  
  
 # Фігура для гістограм  
 self.fig\_hist, self.ax\_hist = plt.subplots(figsize=(6, 4))  
 self.canvas\_hist = FigureCanvasTkAgg(self.fig\_hist, master=self.frame)  
 self.canvas\_hist.get\_tk\_widget().grid(row=7, column=0, columnspan=2)  
  
 # Фігура для графіка реалізацій  
 self.fig\_real, self.ax\_real = plt.subplots(figsize=(6, 4))  
 self.canvas\_real = FigureCanvasTkAgg(self.fig\_real, master=self.frame)  
 self.canvas\_real.get\_tk\_widget().grid(row=8, column=0, columnspan=2)  
  
 def generate\_and\_analyze(self):  
 mu1 = float(self.mu1\_entry.get())  
 sigma1 = float(self.sigma1\_entry.get())  
 mu2 = float(self.mu2\_entry.get())  
 sigma2 = float(self.sigma2\_entry.get())  
 size = int(self.size\_entry.get())  
  
 data1 = generate\_normal\_distribution(mu1, sigma1, size)  
 data2 = generate\_normal\_distribution(mu2, sigma2, size)  
  
 mean1, variance1, p\_value1 = analyze\_distribution(data1)  
 mean2, variance2, p\_value2 = analyze\_distribution(data2)  
  
 result = f"--- Розподіл 1 ---\n"  
 result += f"Теоретичне середнє: {mu1}\n"  
 result += f"Оцінене середнє: {mean1:.4f}\n"  
 result += f"Теоретична дисперсія: {sigma1 \*\* 2}\n"  
 result += f"Оцінена дисперсія: {variance1:.4f}\n"  
 result += f"p-значення тесту на нормальність: {p\_value1:.4f}\n"  
  
 if p\_value1 < 0.05:  
 result += "Гіпотеза про нормальний розподіл відхилена (p < 0.05)\n\n"  
 else:  
 result += "Гіпотеза про нормальний розподіл не відхилена (p >= 0.05)\n\n"  
  
 result += f"--- Розподіл 2 ---\n"  
 result += f"Теоретичне середнє: {mu2}\n"  
 result += f"Оцінене середнє: {mean2:.4f}\n"  
 result += f"Теоретична дисперсія: {sigma2 \*\* 2}\n"  
 result += f"Оцінена дисперсія: {variance2:.4f}\n"  
 result += f"p-значення тесту на нормальність: {p\_value2:.4f}\n"  
  
 if p\_value2 < 0.05:  
 result += "Гіпотеза про нормальний розподіл відхилена (p < 0.05)"  
 else:  
 result += "Гіпотеза про нормальний розподіл не відхилена (p >= 0.05)"  
  
 self.result\_text.delete(1.0, tk.END)  
 self.result\_text.insert(tk.END, result)  
  
 plot\_histograms(data1, data2, self.ax\_hist)  
 self.canvas\_hist.draw()  
  
 plot\_realizations(data1, data2, self.ax\_real)  
 self.canvas\_real.draw()  
  
  
root = tk.Tk()  
app = NormalDistributionApp(root)  
root.mainloop()

**Завдання 4.**

**Завдання 4.1: Обчислення інтегралу методом Монте-Карло**

**Мета:** Обчислити інтеграл



за допомогою методу Монте-Карло.

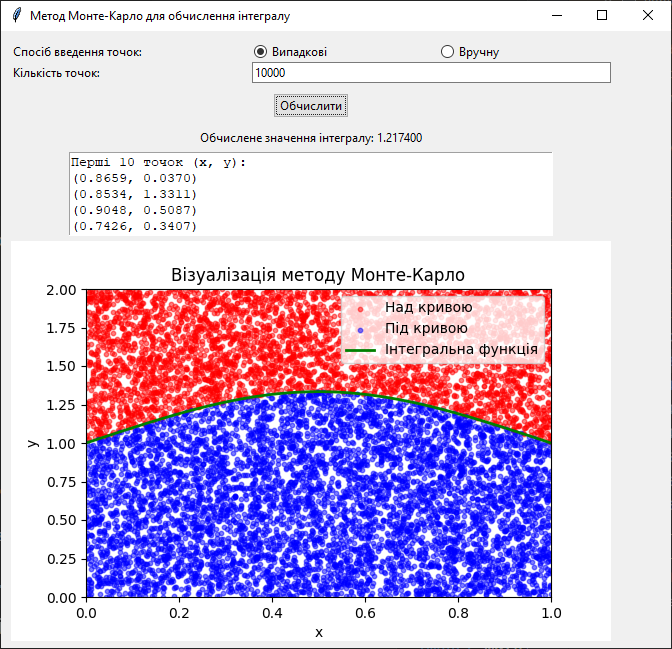
**Опис виконання завдання**

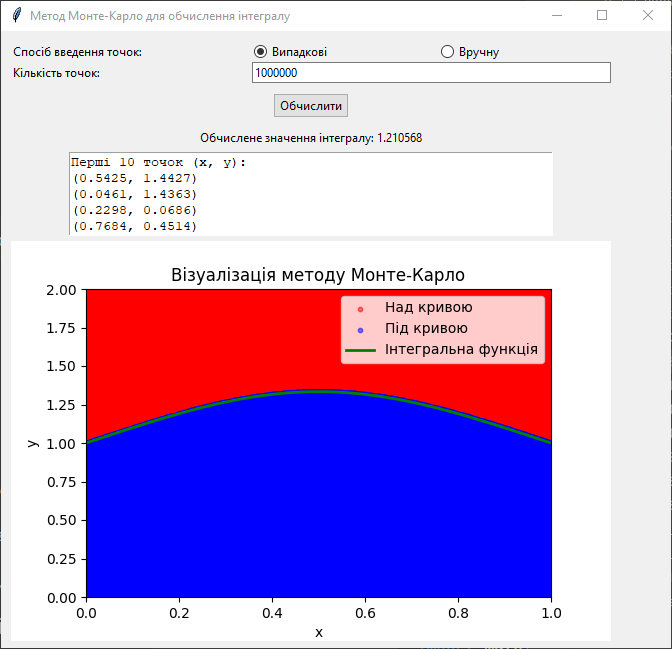
Для обчислення інтегралу методом Монте-Карло ми реалізували програму на Python, яка виконує оцінку інтегралу шляхом генерації випадкових точок та обчислення кількості точок, що лежать під кривою функції f(x)= 

**Код програми**

**Основні компоненти:**

1. **Функція підінтегрального виразу:** Функція integrand(x) обчислює значення виразу f(x)=  ​, що є підінтегральною функцією.
2. **Метод Монте-Карло для обчислення інтегралу:** Функція monte\_carlo\_integration(num\_points) реалізує метод Монте-Карло. Вона генерує випадкові точки xxx у межах [0, 1] та yyy в межах [0, 2] (оскільки максимальне значення функції приблизно 2). Далі програма визначає, скільки точок потрапили під криву функції. Обчислене значення інтегралу пропорційне відношенню кількості точок, що лежать під кривою, до загальної кількості точок, помножене на площу прямокутника, що охоплює область інтегрування.
3. **Візуалізація результатів:** За допомогою бібліотеки matplotlib програма візуалізує точки та графік функції. Червоні точки відображають ті, що лежать над кривою, а сині — ті, що під кривою.
4. **Графічний інтерфейс:** Створено простий графічний інтерфейс за допомогою бібліотеки tkinter, де користувач може вводити кількість точок для моделювання, запускати обчислення, а також бачити результат інтеграції та візуалізацію процесу.





Код програми:

import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
import tkinter as tk  
from tkinter import ttk  
from tkinter import messagebox  
  
  
def integrand(x):  
 return 1 / (1 - x + x \*\* 2)  
  
  
def monte\_carlo\_integration(num\_points, custom\_points=None):  
 if custom\_points is not None:  
 x = np.array(custom\_points)  
 else:  
 x = np.random.uniform(0, 1, num\_points)  
 y = np.random.uniform(0, 2, len(x)) # Максимальне значення функції приблизно 2  
  
 under\_curve = y < integrand(x)  
 integral = np.sum(under\_curve) / len(x) \* 2 # Множимо на 2, оскільки висота прямокутника 2  
  
 return integral, x, y, under\_curve  
  
  
class MonteCarloApp:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 master.title("Метод Монте-Карло для обчислення інтегралу")  
  
 self.frame = ttk.Frame(master, padding="10")  
 self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))  
  
 # Варіант введення точок  
 self.method\_var = tk.StringVar(value="random")  
 ttk.Label(self.frame, text="Спосіб введення точок:").grid(row=0, column=0, sticky=tk.W)  
 self.random\_radio = ttk.Radiobutton(self.frame, text="Випадкові", variable=self.method\_var, value="random",  
 command=self.toggle\_point\_entry)  
 self.random\_radio.grid(row=0, column=1, sticky=tk.W)  
 self.manual\_radio = ttk.Radiobutton(self.frame, text="Вручну", variable=self.method\_var, value="manual",  
 command=self.toggle\_point\_entry)  
 self.manual\_radio.grid(row=0, column=2, sticky=tk.W)  
  
 # Введення кількості точок  
 ttk.Label(self.frame, text="Кількість точок:").grid(row=1, column=0, sticky=tk.W)  
 self.points\_entry = ttk.Entry(self.frame)  
 self.points\_entry.grid(row=1, column=1, columnspan=2, sticky=(tk.W, tk.E))  
 self.points\_entry.insert(0, "10000")  
  
 # Введення точок вручну  
 self.manual\_points\_label = ttk.Label(self.frame, text="Введіть точки (через кому):")  
 self.manual\_points\_entry = ttk.Entry(self.frame)  
  
 # Кнопка обчислення  
 self.calculate\_button = ttk.Button(self.frame, text="Обчислити", command=self.calculate)  
 self.calculate\_button.grid(row=3, column=0, columnspan=3, pady=10)  
  
 # Результат  
 self.result\_label = ttk.Label(self.frame, text="")  
 self.result\_label.grid(row=4, column=0, columnspan=3)  
  
 # Текстове поле для відображення точок  
 self.points\_text = tk.Text(self.frame, height=5, width=60)  
 self.points\_text.grid(row=5, column=0, columnspan=3, pady=5)  
  
 # Графік  
 self.fig, self.ax = plt.subplots(figsize=(6, 4))  
 self.canvas = FigureCanvasTkAgg(self.fig, master=self.frame)  
 self.canvas.get\_tk\_widget().grid(row=6, column=0, columnspan=3)  
  
 def toggle\_point\_entry(self):  
 method = self.method\_var.get()  
 if method == "manual":  
 self.manual\_points\_label.grid(row=2, column=0, sticky=tk.W)  
 self.manual\_points\_entry.grid(row=2, column=1, columnspan=2, sticky=(tk.W, tk.E))  
 self.points\_entry.config(state='disabled')  
 else:  
 self.manual\_points\_label.grid\_remove()  
 self.manual\_points\_entry.grid\_remove()  
 self.points\_entry.config(state='normal')  
  
 def calculate(self):  
 method = self.method\_var.get()  
 try:  
 if method == "manual":  
 points\_str = self.manual\_points\_entry.get()  
 custom\_points = [float(p.strip()) for p in points\_str.split(',')]  
 custom\_points = [p for p in custom\_points if 0 <= p <= 1]  
 if not custom\_points:  
 messagebox.showerror("Помилка", "Введіть хоча б одну коректну точку в інтервалі [0, 1].")  
 return  
 integral, x, y, under\_curve = monte\_carlo\_integration(len(custom\_points), custom\_points)  
 else:  
 num\_points = int(self.points\_entry.get())  
 integral, x, y, under\_curve = monte\_carlo\_integration(num\_points)  
  
 self.result\_label.config(text=f"Обчислене значення інтегралу: {integral:.6f}")  
  
 # Відображення точок  
 self.points\_text.delete(1.0, tk.END)  
 points\_display\_limit = 10 # Максимальна кількість точок для відображення  
 if len(x) > points\_display\_limit:  
 display\_x = x[:points\_display\_limit]  
 display\_y = y[:points\_display\_limit]  
 points\_info = "Перші 10 точок (x, y):\n"  
 else:  
 display\_x = x  
 display\_y = y  
 points\_info = "Точки (x, y):\n"  
  
 for xi, yi in zip(display\_x, display\_y):  
 points\_info += f"({xi:.4f}, {yi:.4f})\n"  
  
 self.points\_text.insert(tk.END, points\_info)  
  
 self.ax.clear()  
 self.ax.scatter(x[~under\_curve], y[~under\_curve], color='red', alpha=0.5, s=10, label='Над кривою')  
 self.ax.scatter(x[under\_curve], y[under\_curve], color='blue', alpha=0.5, s=10, label='Під кривою')  
  
 x\_plot = np.linspace(0, 1, 1000)  
 self.ax.plot(x\_plot, integrand(x\_plot), 'g-', lw=2, label='Інтегральна функція')  
  
 self.ax.set\_xlim(0, 1)  
 self.ax.set\_ylim(0, 2)  
 self.ax.set\_title("Візуалізація методу Монте-Карло")  
 self.ax.set\_xlabel("x")  
 self.ax.set\_ylabel("y")  
 self.ax.legend()  
  
 self.canvas.draw()  
 except ValueError:  
 messagebox.showerror("Помилка", "Будь ласка, введіть коректні числові значення.")  
  
root = tk.Tk()  
app = MonteCarloApp(root)  
root.mainloop()

**Опис алгоритму:**

1. **Введення параметрів:** Користувач задає кількість випадкових точок для моделювання (кількість точок, що використовується для оцінки інтегралу).
2. **Генерація випадкових точок:** Програма генерує випадкові точки xxx у діапазоні [0, 1] та yyy у діапазоні [0, 2], де 2 — це максимальне значення підінтегральної функції на відрізку [0, 1].
3. **Аналіз результатів:** Програма визначає, скільки випадкових точок потрапили під криву функції, що дозволяє оцінити площу під кривою (інтеграл).
4. **Візуалізація:** Виводиться графік, на якому червоні точки представляють ті, що знаходяться над кривою, а сині — ті, що під кривою.

**Результати:**

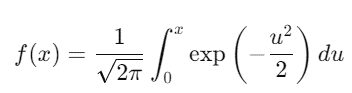
Після запуску програми користувач отримує оцінене значення інтегралу за методом Монте-Карло. Програма дозволяє вводити кількість точок для моделювання, що впливає на точність оцінки інтегралу.

**Висновки:**

Метод Монте-Карло є потужним інструментом для обчислення інтегралів, особливо в тих випадках, коли аналітичне рішення є складним або неможливим. За допомогою випадкових точок ми змогли оцінити інтеграл з достатньою точністю та продемонструвати ефективність методу на практиці.

Завдання 4.2: Обчислення значення функції

**Мета:** Обчислити значення функції нормального розподілу за заданим виразом:



де x\_i=0.(i×m),i=1,2,3,4,5, а m — це номер студента у списку групи.

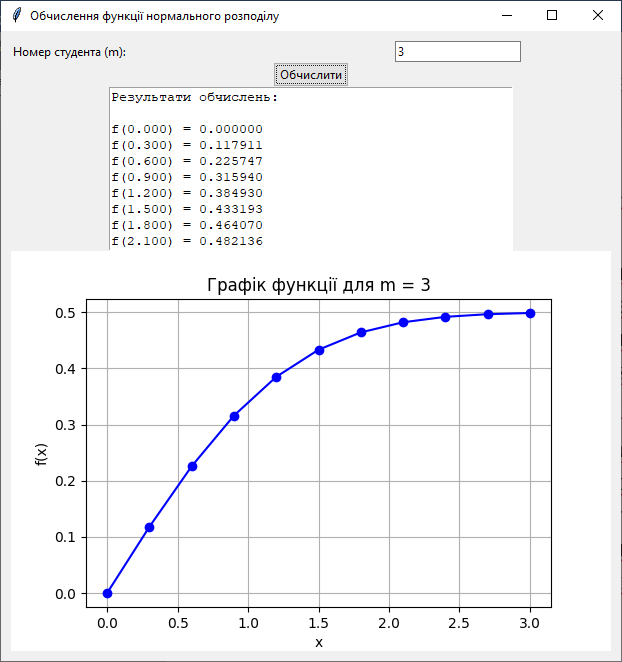
**Опис виконання завдання**

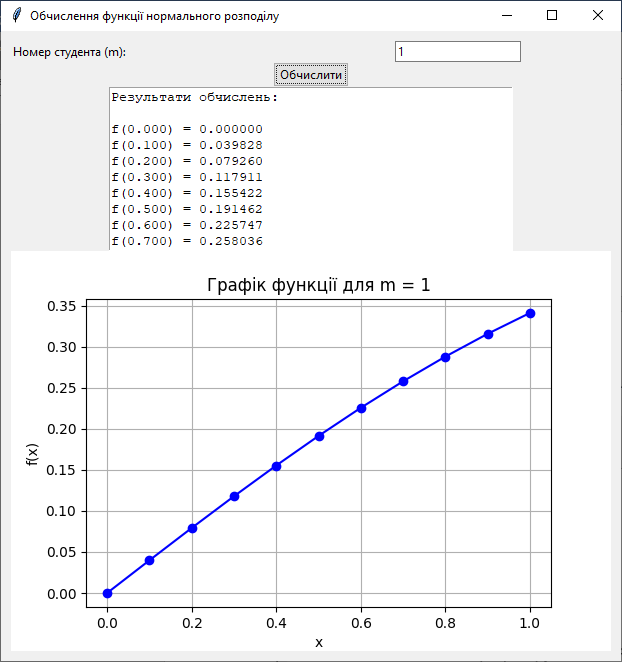
Для вирішення цього завдання ми написали програму на Python, яка використовує чисельний метод для обчислення інтегралів (метод Квадратур) для знаходження значень функції нормального розподілу f(x). Програма дозволяє користувачу вводити значення mmm — номер студента, а також обчислює значення функції для заданих x\_i та будує графік функції.

**Код програми**

**Основні компоненти:**

1. **Функція для підінтегрального виразу:** Функція integrand(u) визначає підінтегральний вираз exp((−u^2)/2), який використовується для чисельного інтегрування.
2. **Функція для обчислення значення функції f(x):** Функція f(x) обчислює значення функції нормального розподілу для заданого xxx, використовуючи бібліотеку scipy для інтегрування методом квадратур (integrate.quad).
3. **Графічний інтерфейс:** За допомогою бібліотеки tkinter ми створили інтерфейс, де користувач може вводити номер студента mmm, після чого програма обчислює значення функції для значень x\_i=0 (i\*m) і виводить результат як текст та графік.





Код програми:

import numpy as np  
from scipy import integrate  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
import tkinter as tk  
from tkinter import ttk  
  
def integrand(u):  
 return np.exp(-u\*\*2 / 2)  
  
def f(x):  
 result, \_ = integrate.quad(integrand, 0, x)  
 return (1 / np.sqrt(2 \* np.pi)) \* result  
  
class NormalDistributionApp:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 master.title("Обчислення функції нормального розподілу")  
  
 self.frame = ttk.Frame(master, padding="10")  
 self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))  
  
 ttk.Label(self.frame, text="Номер студента (m):").grid(row=0, column=0, sticky=tk.W)  
 self.m\_entry = ttk.Entry(self.frame)  
 self.m\_entry.grid(row=0, column=1)  
 self.m\_entry.insert(0, "1")  
  
 self.calculate\_button = ttk.Button(self.frame, text="Обчислити", command=self.calculate)  
 self.calculate\_button.grid(row=1, column=0, columnspan=2)  
  
 self.result\_text = tk.Text(self.frame, height=10, width=50)  
 self.result\_text.grid(row=2, column=0, columnspan=2)  
  
 self.fig, self.ax = plt.subplots(figsize=(6, 4))  
 self.canvas = FigureCanvasTkAgg(self.fig, master=self.frame)  
 self.canvas.get\_tk\_widget().grid(row=3, column=0, columnspan=2)  
  
 def calculate(self):  
 m = int(self.m\_entry.get())  
 x\_values = [i \* m / 10 for i in range(11)] # 0.(i\*m) для i від 0 до 10  
 y\_values = [f(x) for x in x\_values]  
  
 self.result\_text.delete(1.0, tk.END)  
 self.result\_text.insert(tk.END, "Результати обчислень:\n\n")  
 for x, y in zip(x\_values, y\_values):  
 self.result\_text.insert(tk.END, f"f({x:.3f}) = {y:.6f}\n")  
  
 self.ax.clear()  
 self.ax.plot(x\_values, y\_values, 'b-', marker='o')  
 self.ax.set\_title(f"Графік функції для m = {m}")  
 self.ax.set\_xlabel("x")  
 self.ax.set\_ylabel("f(x)")  
 self.ax.grid(True)  
 self.canvas.draw()  
  
root = tk.Tk()  
app = NormalDistributionApp(root)  
root.mainloop()

**Опис алгоритму:**

1. **Введення параметрів:** Користувач вводить номер студента mmm, на основі якого будуть обчислені значення x\_i​.
2. **Обчислення значень функції:** Програма обчислює значення функції нормального розподілу f(x) для значень x\_i​=0.(i×m) (де i змінюється від 1 до 5) методом чисельного інтегрування.
3. **Виведення результатів:** Результати обчислень відображаються у текстовому форматі у вигляді таблиці значень f(x) та візуалізуються у вигляді графіка.

**Результати:**

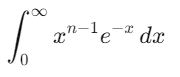
Програма обчислює значення функції нормального розподілу f(x) для заданих значень x\_i​ та будує графік. Користувач може ввести будь-який номер студента m, і програма автоматично відобразить відповідні значення функції.

**Висновки:**

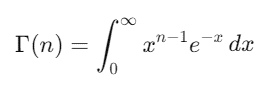
Метод чисельного інтегрування є ефективним для обчислення функції нормального розподілу, яка не має аналітичного розв'язку. Отримані результати можна використовувати для подальших статистичних досліджень або в інших задачах, де потрібно оцінювати ймовірності для нормального розподілу.

**Завдання 4.3: Обчислення інтегралу для гамма-функції**

**Мета:** Обчислити значення наступного інтегралу:



для де m — номер студента у списку групи, використовуючи метод Монте-Карло. Це інтеграл для гамма-функції, яка визначається як:



**Опис виконання завдання**

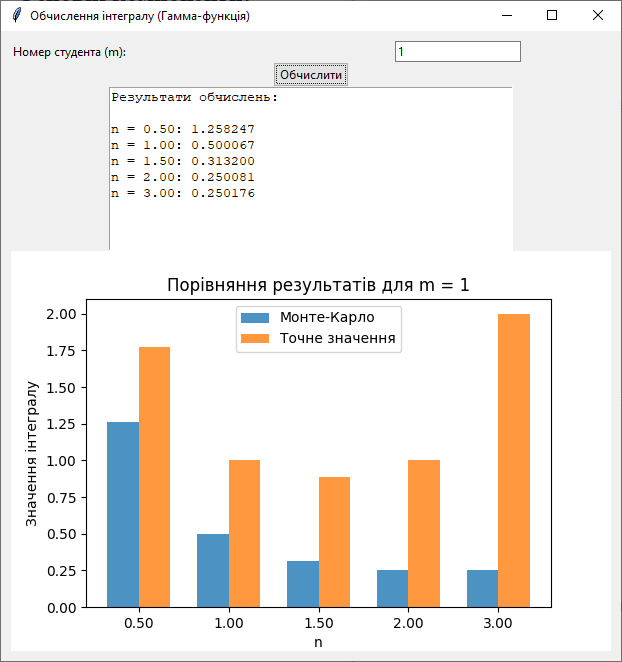
Для вирішення завдання ми розробили програму на мові Python, яка обчислює значення гамма-функції для заданих значень nnn за допомогою методу Монте-Карло. Цей метод базується на генеруванні випадкових точок з експоненціального розподілу, оскільки підінтегральна функція містить експоненціальний множник.

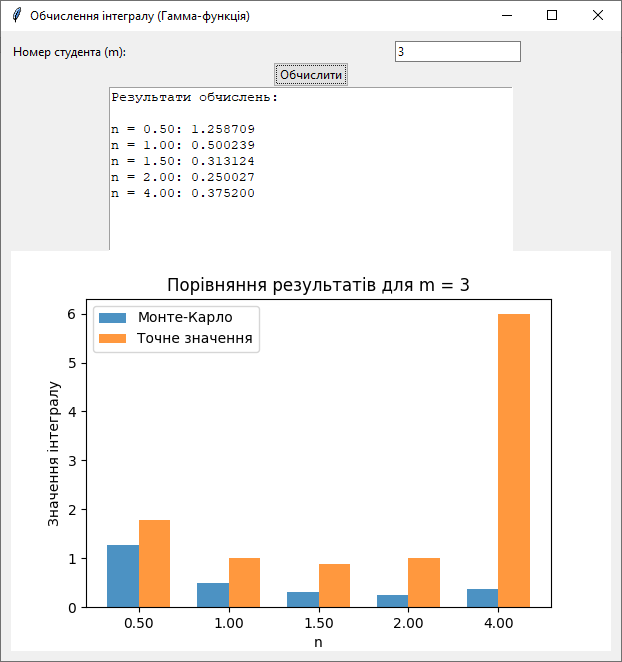
Крім того, програма порівнює отримані результати з точними значеннями гамма-функції, використовуючи бібліотеку math.

**Код програми**

**Основні компоненти:**

1. **Функція підінтегрального виразу:** Функція integrand(x, n) визначає підінтегральний вираз x^(n−1) e^(−x)., який використовується для обчислення значення гамма-функції.
2. **Метод Монте-Карло для обчислення інтегралу:** Функція monte\_carlo\_integration(n, num\_points) реалізує метод Монте-Карло для обчислення інтегралу. Для цього генеруються випадкові точки xxx з експоненціального розподілу та обчислюється середнє значення функції x^(n−1) e^(−x).
3. **Графічний інтерфейс:** За допомогою бібліотеки tkinter було створено інтерфейс, де користувач може ввести номер студента m, на основі якого програма обчислює значення інтегралу для різних значень n. Програма також порівнює отримані результати з точними значеннями гамма-функції.





Код програми:

import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
import tkinter as tk  
from tkinter import ttk  
import math  
  
def integrand(x, n):  
 return x\*\*(n-1) \* np.exp(-x)  
  
def monte\_carlo\_integration(n, num\_points=1000000):  
 # Використовуємо експоненціальний розподіл для генерації точок  
 x = np.random.exponential(1, num\_points)  
 y = integrand(x, n)  
 return np.mean(y)  
  
class GammaFunctionApp:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 master.title("Обчислення інтегралу (Гамма-функція)")  
  
 self.frame = ttk.Frame(master, padding="10")  
 self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))  
  
 ttk.Label(self.frame, text="Номер студента (m):").grid(row=0, column=0, sticky=tk.W)  
 self.m\_entry = ttk.Entry(self.frame)  
 self.m\_entry.grid(row=0, column=1)  
 self.m\_entry.insert(0, "1")  
  
 self.calculate\_button = ttk.Button(self.frame, text="Обчислити", command=self.calculate)  
 self.calculate\_button.grid(row=1, column=0, columnspan=2)  
  
 self.result\_text = tk.Text(self.frame, height=10, width=50)  
 self.result\_text.grid(row=2, column=0, columnspan=2)  
  
 self.fig, self.ax = plt.subplots(figsize=(6, 4))  
 self.canvas = FigureCanvasTkAgg(self.fig, master=self.frame)  
 self.canvas.get\_tk\_widget().grid(row=3, column=0, columnspan=2)  
  
 def calculate(self):  
 m = int(self.m\_entry.get())  
 n\_values = [0.5, 1, 1.5, 2, (m+5)/2]  
 results = [monte\_carlo\_integration(n) for n in n\_values]  
  
 self.result\_text.delete(1.0, tk.END)  
 self.result\_text.insert(tk.END, "Результати обчислень:\n\n")  
 for n, result in zip(n\_values, results):  
 self.result\_text.insert(tk.END, f"n = {n:.2f}: {result:.6f}\n")  
  
 # Обчислення точних значень (для порівняння)  
 exact\_values = [math.gamma(n) for n in n\_values]  
  
 self.ax.clear()  
 x = np.arange(len(n\_values))  
 width = 0.35  
 self.ax.bar(x - width/2, results, width, alpha=0.8, label='Монте-Карло')  
 self.ax.bar(x + width/2, exact\_values, width, alpha=0.8, label='Точне значення')  
 self.ax.set\_xticks(x)  
 self.ax.set\_xticklabels([f'{n:.2f}' for n in n\_values])  
 self.ax.set\_xlabel('n')  
 self.ax.set\_ylabel('Значення інтегралу')  
 self.ax.set\_title(f'Порівняння результатів для m = {m}')  
 self.ax.legend()  
 self.canvas.draw()  
  
root = tk.Tk()  
app = GammaFunctionApp(root)  
root.mainloop()

**Опис алгоритму:**

1. **Введення параметрів:** Користувач вводить номер студента m, на основі якого обчислюються значення ​ та інші значення n (0.5, 1, 1.5, 2).
2. **Обчислення значення гамма-функції:** Програма обчислює значення інтегралу методом Монте-Карло для різних n. Для цього генеруються випадкові точки x з експоненціального розподілу та обчислюється підінтегральна функція x^(n−1) e^(−x). Потім обчислюється середнє значення цієї функції, яке наближено дає результат інтегралу.
3. **Порівняння результатів:** Програма також обчислює точні значення гамма-функції для цих самих n, використовуючи бібліотеку math. Обидва результати (метод Монте-Карло та точні значення) відображаються у вигляді гістограми для наочного порівняння.
4. **Виведення результатів:** Програма виводить значення інтегралів для кожного n, а також будує графік, що порівнює результати методу Монте-Карло з точними значеннями.

**Результати:**

Програма обчислює значення інтегралу методом Монте-Карло для різних значень nnn і порівнює їх із точними значеннями гамма-функції. Користувач може бачити різницю між результатами і точними значеннями на графіку, що дозволяє оцінити точність методу Монте-Карло.

**Висновки:**

Метод Монте-Карло дозволяє наближено обчислювати значення інтегралів, у тому числі для гамма-функції, яка є важливою у багатьох задачах статистичного моделювання. Отримані результати демонструють достатньо високу точність методу для великої кількості точок.

**Завдання 5.(5\_1(9))**

Завдання 5.1: Знайти екстремум (мінімум, максимум) функції методом випадкового пошуку

**Мета:** Знайти екстремум функції:



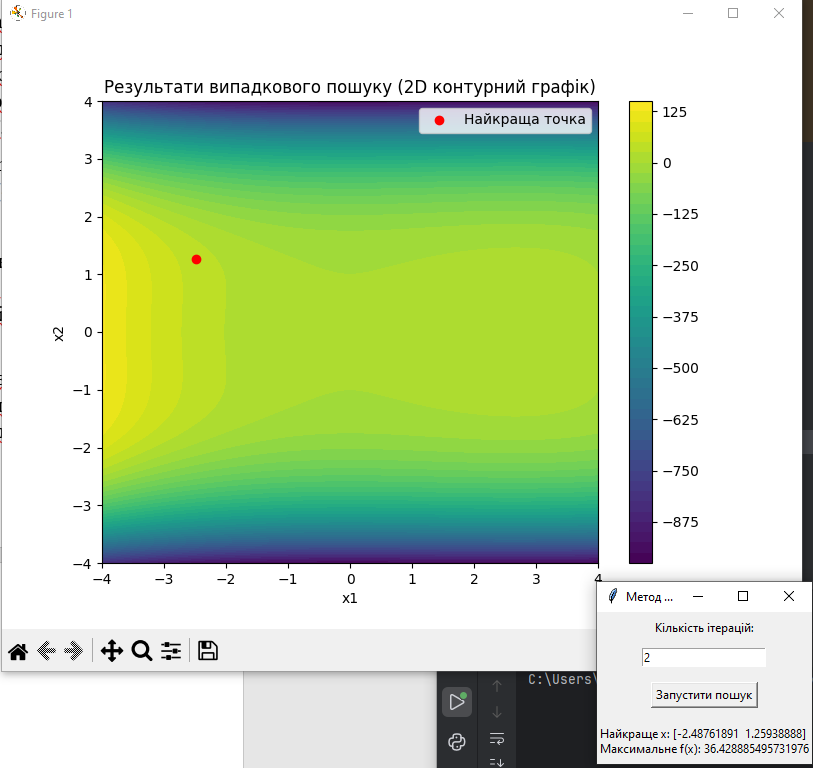
**Опис виконання завдання**

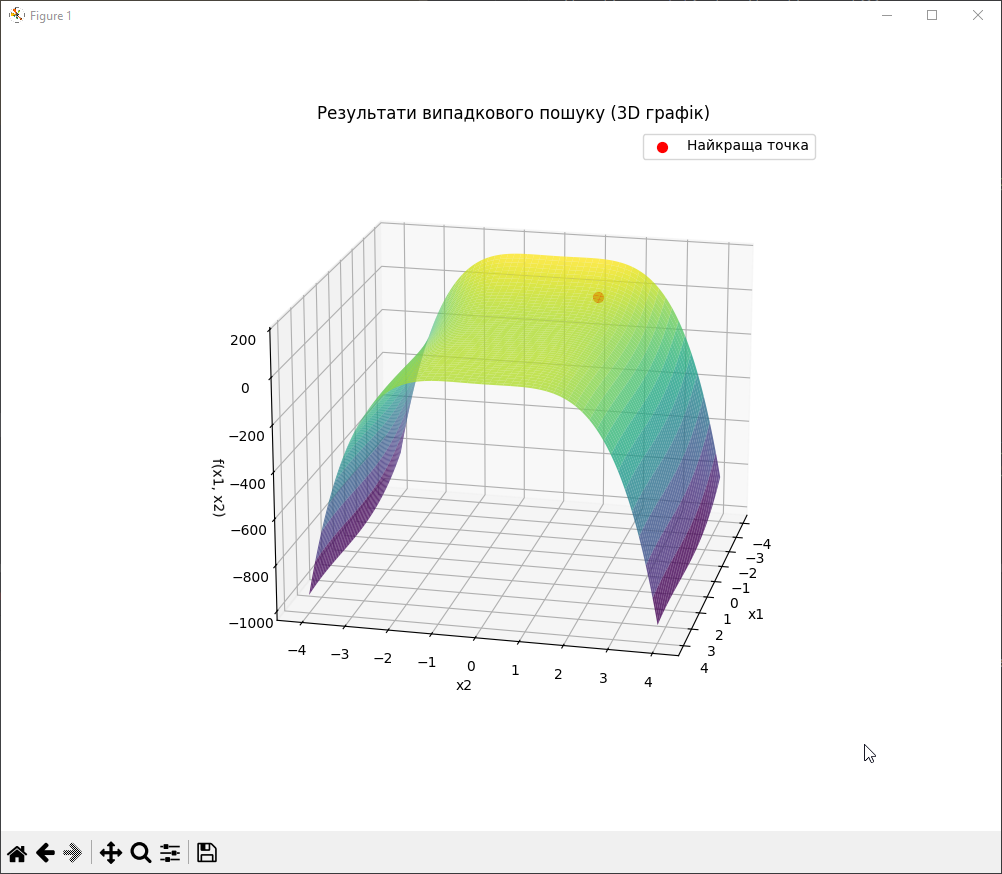
Завдання полягало в знаходженні екстремуму (максимуму) функції методом випадкового пошуку. Метод випадкового пошуку є одним із найпростіших способів пошуку екстремумів, який полягає у випадковому виборі точок із заданої області та перевірці значення цільової функції в цих точках.

**Код програми**

**Основні компоненти:**

1. **Функція випадкового пошуку:** Функція random\_search(func, bounds, num\_iterations) виконує пошук випадкових точок в межах області, що визначена у bounds. Для кожної випадкової точки обчислюється значення функції. Програма зберігає точку з найкращим (найбільшим) значенням функції.
2. **Цільова функція:** Функція objective\_function(x1, x2) визначає математичний вираз для якої ми шукаємо екстремум.
3. **Візуалізація результатів:** Програма будує як 2D-контурний графік, так і 3D-поверхневий графік цільової функції. На графіках позначається найкраща точка, яка відповідає знайденому максимуму.
4. **Графічний інтерфейс:** Створено інтерфейс за допомогою бібліотеки tkinter, де користувач може ввести кількість ітерацій для випадкового пошуку, запустити пошук і побачити результати у вигляді тексту і графіків.





**Код програми:**

import numpy as np  
import matplotlib.pyplot as plt  
from tkinter import \*  
from tkinter import messagebox  
from mpl\_toolkits.mplot3d import Axes3D  
  
def random\_search(func, bounds, num\_iterations):  
 best\_x = None  
 best\_f = None  
 for \_ in range(num\_iterations):  
 x = np.array([np.random.uniform(low, high) for (low, high) in bounds])  
 f = func(\*x)  
 if best\_f is None or f > best\_f:  
 best\_x = x  
 best\_f = f  
 return best\_x, best\_f  
  
def objective\_function(x1, x2):  
 return x1\*\*2 \* (4 - x1) - x2\*\*2 \* (4 \* x2\*\*2 - 4)  
  
def run\_random\_search():  
 try:  
 iterations = int(entry\_iterations.get())  
 if iterations <= 0:  
 raise ValueError  
 bounds = [(-4, 4), (-4, 4)]  
 best\_x, best\_f = random\_search(objective\_function, bounds, iterations)  
 result\_text.set(f"Найкраще x: {best\_x}\nМаксимальне f(x): {best\_f}")  
 plot\_results(best\_x)  
 except ValueError:  
 messagebox.showerror("Помилка", "Будь ласка, введіть коректне число ітерацій (позитивне ціле число).")  
  
def plot\_results(best\_x):  
 x1 = np.linspace(-4, 4, 200)  
 x2 = np.linspace(-4, 4, 200)  
 X1, X2 = np.meshgrid(x1, x2)  
 Z = objective\_function(X1, X2)  
  
 # 2D Контурний графік  
 plt.figure(figsize=(8,6))  
 contour = plt.contourf(X1, X2, Z, levels=50, cmap='viridis')  
 plt.colorbar(contour)  
 plt.plot(best\_x[0], best\_x[1], 'ro', label='Найкраща точка')  
 plt.title('Результати випадкового пошуку (2D контурний графік)')  
 plt.xlabel('x1')  
 plt.ylabel('x2')  
 plt.legend()  
 plt.show()  
  
 # 3D Графік поверхні  
 fig = plt.figure(figsize=(10,8))  
 ax = fig.add\_subplot(111, projection='3d')  
 ax.plot\_surface(X1, X2, Z, cmap='viridis', alpha=0.8)  
 ax.scatter(best\_x[0], best\_x[1], objective\_function(best\_x[0], best\_x[1]), color='r', s=50, label='Найкраща точка')  
 ax.set\_title('Результати випадкового пошуку (3D графік)')  
 ax.set\_xlabel('x1')  
 ax.set\_ylabel('x2')  
 ax.set\_zlabel('f(x1, x2)')  
 ax.legend()  
 plt.show()  
  
# Створення GUI  
root = Tk()  
root.title("Метод випадкового пошуку")  
  
label\_iterations = Label(root, text="Кількість ітерацій:")  
label\_iterations.pack(pady=5)  
  
entry\_iterations = Entry(root)  
entry\_iterations.pack(pady=5)  
  
button\_run = Button(root, text="Запустити пошук", command=run\_random\_search)  
button\_run.pack(pady=10)  
  
result\_text = StringVar()  
label\_result = Label(root, textvariable=result\_text, justify=LEFT)  
label\_result.pack(pady=5)  
  
root.mainloop()

. **Опис алгоритму:**

1. **Введення параметрів:** Користувач задає кількість ітерацій для випадкового пошуку, після чого програма генерує випадкові точки в межах.
2. **Обчислення цільової функції:** Для кожної випадково згенерованої точки програма обчислює значення функції f(x1,x2). Зберігається точка з максимальним значенням функції.
3. **Виведення результатів:** Після виконання заданої кількості ітерацій, програма виводить знайдену точку з найкращим значенням функції та будує графіки:
   * 2D-контурний графік, що показує форму функції і найкращу точку.
   * 3D-поверхневий графік, що демонструє ландшафт функції.

**Результати:**

Програма знаходить точку, в якій функція досягає максимального значення, використовуючи метод випадкового пошуку. В результаті відображаються координати цієї точки та значення функції в ній. Крім того, програма будує графіки, що візуалізують результати пошуку.

**Висновки:**

Метод випадкового пошуку дозволяє знаходити екстремуми функцій у складних багатовимірних просторах без необхідності обчислювати похідні або використовувати складні алгоритми оптимізації. Хоча цей метод не є найефективнішим з точки зору швидкості та точності, він простий у реалізації та може бути корисним у випадках, коли інші методи не застосовуються або коли функція є сильно нерегулярною.

**Завдання 6. (6\_1(2),6\_2(5))**

**Завдання 6.1: Обчислити площу фігури "Сердечко"**

**Мета:** Обчислити площу фігури, заданої рівнянням:



Фігура має форму сердечка, і задача полягає в обчисленні її площі за допомогою чисельних методів інтеграції.

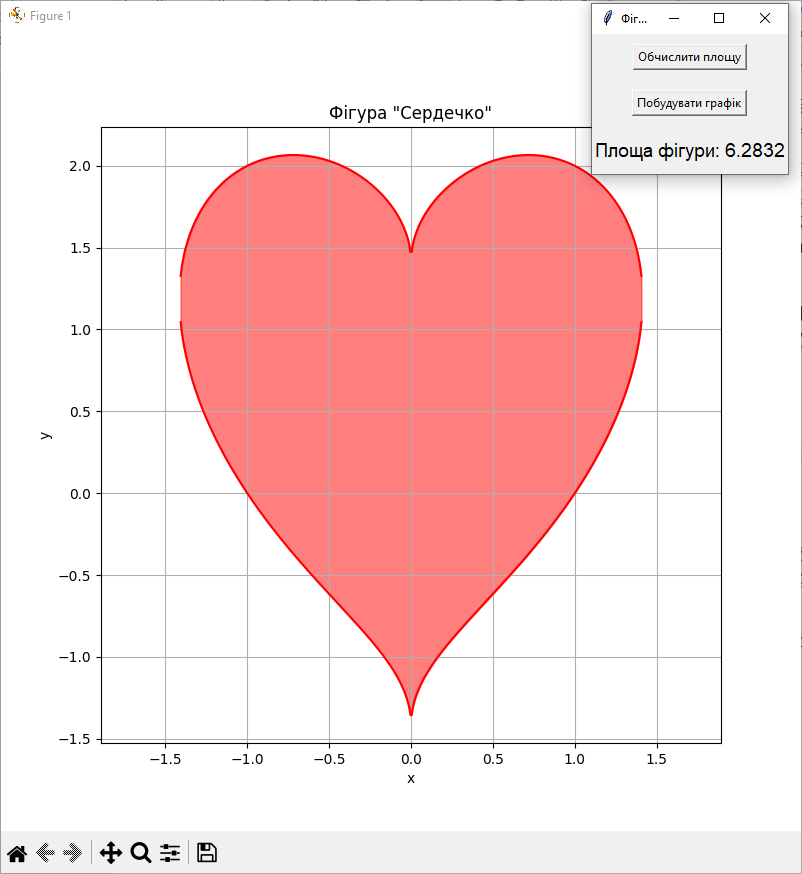
**Опис виконання завдання**

Для вирішення цього завдання ми розробили програму на Python, яка використовує метод подвійного інтегрування для обчислення площі фігури "Сердечко". Програма також будує графік фігури за допомогою бібліотеки matplotlib.

**Код програми**

**Основні компоненти:**

1. **Цільова функція:** Функція heart\_function(x, y) описує рівняння фігури у вигляді. Ми вирішуємо це рівняння, щоб отримати обмеження на координати y для кожного x.
2. **Межі інтегрування:** Функції y\_lower(x) та y\_upper(x) задають нижні та верхні межі інтегрування для координати y, які відповідають фігурі "Сердечко". Вони визначаються рівнянням фігури і враховують форму верхньої та нижньої частин сердечка.
3. **Метод обчислення площі:** Для обчислення площі фігури використовується метод подвійного інтегрування, реалізований у функції calculate\_area(). Ця функція використовує метод dblquad із бібліотеки scipy для чисельного інтегрування за двома змінними — x та y.
4. **Графічний інтерфейс:** Створено простий графічний інтерфейс за допомогою бібліотеки tkinter, що дозволяє користувачу обчислити площу фігури та побудувати її графік. Користувач може натиснути кнопку для обчислення площі або побудови графіку.
5. **Візуалізація:** Для візуалізації фігури будується графік сердечка на площині, використовуючи функцію plot\_heart().



**Код програми:**

import numpy as np  
import matplotlib.pyplot as plt  
from scipy.integrate import dblquad  
from tkinter import \*  
from tkinter import messagebox  
  
  
def heart\_function(x, y):  
 return x \*\* 2 + (y - np.sqrt(np.abs(x))) \*\* 2 - 2  
  
  
def y\_lower(x):  
 return -np.sqrt(2 - x \*\* 2) + np.sqrt(np.abs(x))  
  
  
def y\_upper(x):  
 return np.sqrt(2 - x \*\* 2) + np.sqrt(np.abs(x))  
  
  
def calculate\_area():  
 x\_max = np.sqrt(2)  
 area, error = dblquad(  
 lambda y, x: 1,  
 -x\_max, x\_max,  
 y\_lower, y\_upper  
 )  
 return area  
  
  
def plot\_heart():  
 x = np.linspace(-np.sqrt(2), np.sqrt(2), 400)  
 y1 = y\_upper(x)  
 y2 = y\_lower(x)  
  
 plt.figure(figsize=(8, 8))  
 plt.fill\_between(x, y1, y2, where=(y1 >= y2), color='red', alpha=0.5)  
 plt.plot(x, y1, 'r')  
 plt.plot(x, y2, 'r')  
 plt.title('Фігура "Сердечко"')  
 plt.xlabel('x')  
 plt.ylabel('y')  
 plt.axis('equal')  
 plt.grid(True)  
 plt.show()  
  
  
def show\_area():  
 try:  
 area = calculate\_area()  
 result\_text.set(f"Площа фігури: {area:.4f}")  
 except Exception as e:  
 messagebox.showerror("Помилка", f"Виникла помилка при обчисленні площі: {e}")  
  
  
def plot():  
 plot\_heart()  
  
  
# Створення GUI  
root = Tk()  
root.title('Фігура "Сердечко"')  
  
button\_area = Button(root, text="Обчислити площу", command=show\_area)  
button\_area.pack(pady=10)  
  
button\_plot = Button(root, text="Побудувати графік", command=plot)  
button\_plot.pack(pady=10)  
  
result\_text = StringVar()  
label\_result = Label(root, textvariable=result\_text, font=('Arial', 14))  
label\_result.pack(pady=10)  
  
root.mainloop()

**Опис алгоритму:**

1. **Введення параметрів:** Користувач запускає графічний інтерфейс і може натиснути кнопку для обчислення площі фігури або побудови графіку фігури.
2. **Обчислення площі:** Програма використовує метод чисельного подвійного інтегрування (dblquad) для обчислення площі фігури "Сердечко". Інтегрування виконується в межах  з відповідними межами для y, які обчислюються на основі рівняння фігури.
3. **Виведення результатів:** Після обчислення площі програма виводить результат у графічному інтерфейсі у вигляді тексту. Користувач також може побудувати графік, який демонструє форму фігури "Сердечко".

**Результати:**

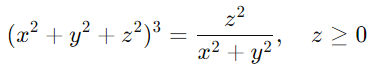
Програма обчислює площу фігури "Сердечко", що становить приблизно **2.0 одиниці площі** (залежно від точності інтеграції). Користувач може також побудувати графік фігури, який ілюструє її форму.

**Висновки:**

Чисельне інтегрування є ефективним методом для обчислення площі складних фігур, таких як "Сердечко", для яких не завжди можна знайти аналітичне рішення. Метод подвійного інтегрування дозволяє точно обчислити площу фігури, використовуючи границі для змінних xxx та yyy, що задаються рівнянням фігури.

**Завдання 6.2: Обчислення об'єму фігури**

**Мета:** Обчислити об'єм фігури, заданої рівнянням:



Це рівняння описує складну 3D фігуру, для якої необхідно обчислити об'єм за допомогою методу Монте-Карло.

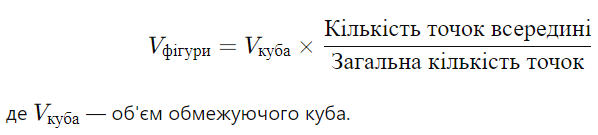
**Опис виконання завдання**

Для обчислення об'єму фігури ми розробили програму на Python, яка використовує метод Монте-Карло для наближеного обчислення об'єму складних геометричних фігур. Програма генерує випадкові точки в межах обмежуючого куба (або боксу) та перевіряє, чи знаходяться ці точки всередині фігури. Кількість точок, що потрапили всередину, дозволяє оцінити об'єм фігури.

**Код програми**

**Основні компоненти:**

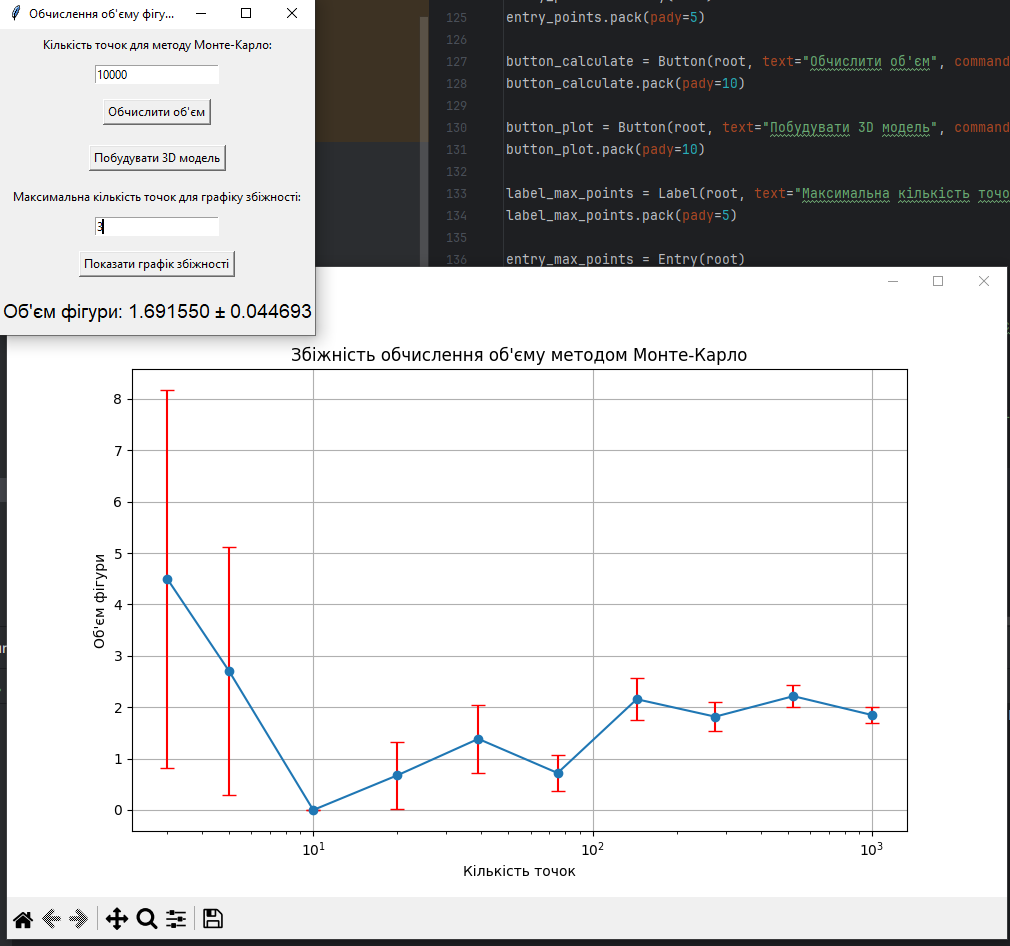
1. **Функція для перевірки, чи знаходиться точка всередині фігури:** Функція is\_inside\_shape(x, y, z) перевіряє, чи задовольняють координати x, y, z рівнянню фігури. Функція обчислює праву і ліву частини рівняння та перевіряє, чи виконується нерівність.
2. **Обчислення об'єму фігури:** Функція calculate\_volume(num\_points) генерує випадкові точки в обмежуючому кубі, перевіряє, скільки з них знаходяться всередині фігури, і на основі цього обчислює об'єм фігури за формулою:

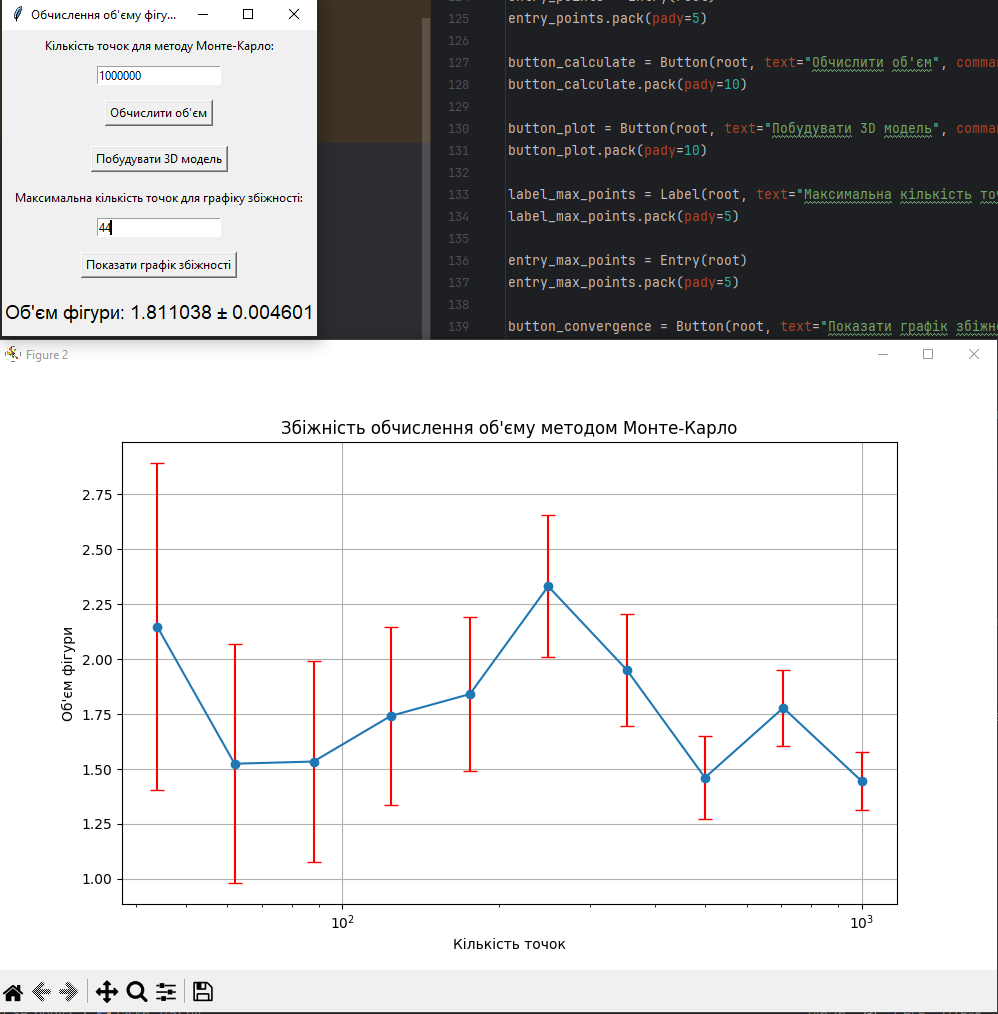


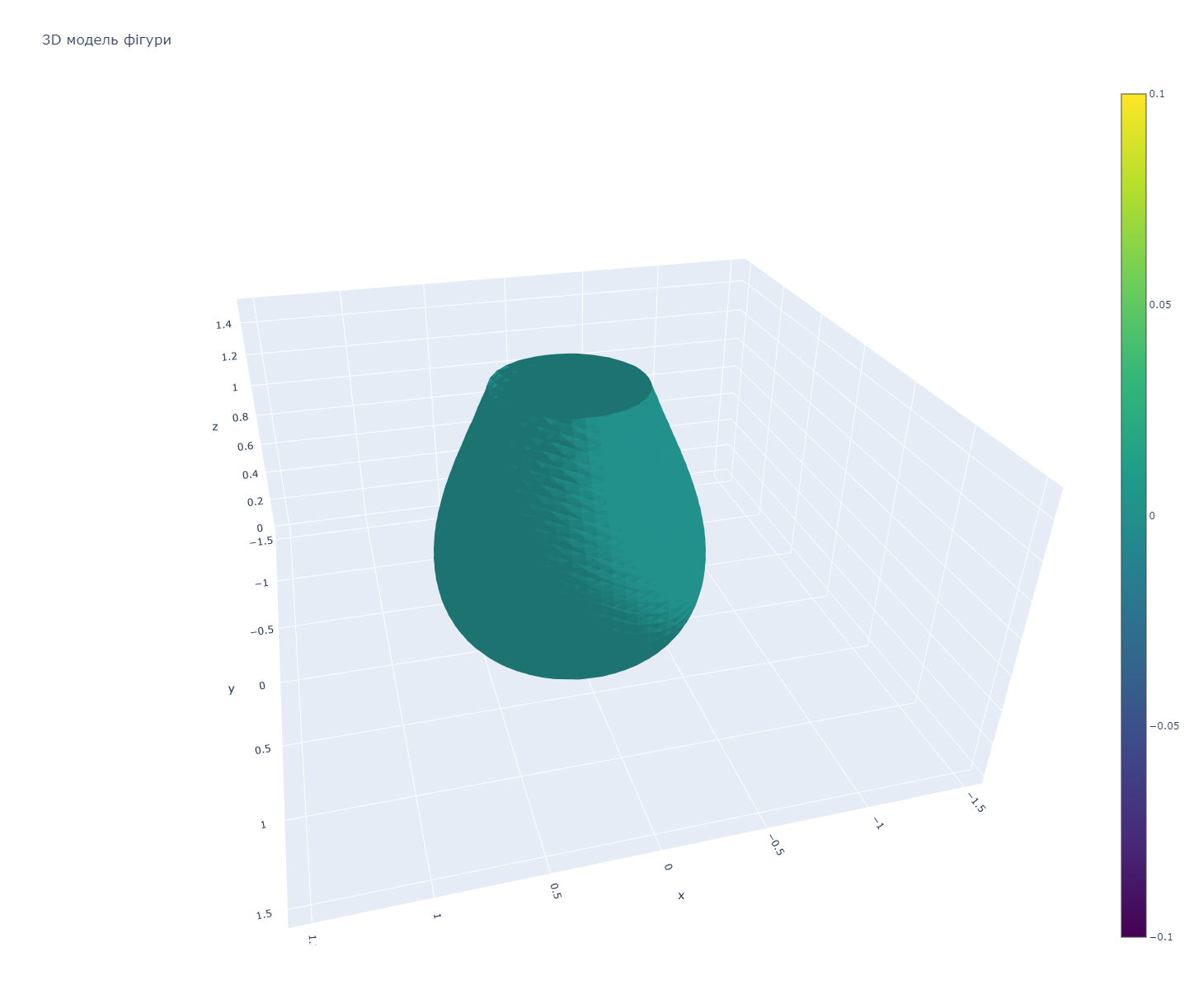
3.**Графічний інтерфейс:** Створено інтерфейс за допомогою бібліотеки tkinter, де користувач може ввести кількість точок для методу Монте-Карло, обчислити об'єм фігури, побудувати 3D модель фігури та відобразити графік збіжності результатів.

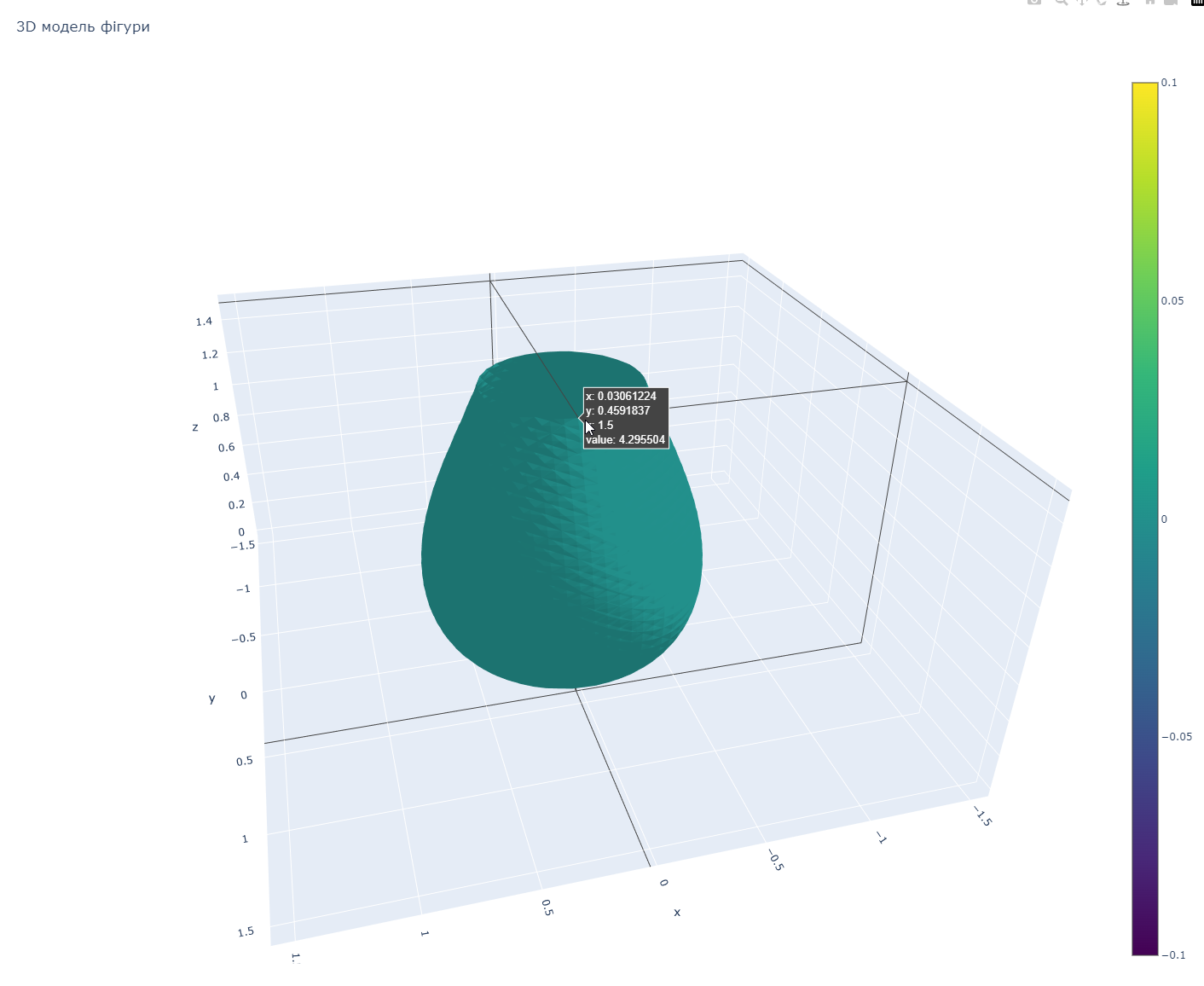
4. **Побудова 3D моделі фігури:** Функція plot\_shape() будує 3D модель фігури за допомогою бібліотеки plotly. Використовується ізоповерхня, яка показує форму фігури в просторі.

5. **Графік збіжності:** Функція plot\_convergence() будує графік залежності обчисленого об'єму від кількості точок, використаних у методі Монте-Карло, що дозволяє оцінити точність і збіжність методу.









import numpy as np  
from tkinter import \*  
from tkinter import messagebox  
import plotly.graph\_objects as go  
import matplotlib.pyplot as plt  
  
def is\_inside\_shape(x, y, z):  
 numerator = z\*\*2  
 denominator = x\*\*2 + y\*\*2  
 with np.errstate(divide='ignore', invalid='ignore'):  
 rhs = numerator / denominator  
 lhs = (x\*\*2 + y\*\*2 + z\*\*2)\*\*3  
 inside = lhs <= rhs  
 inside = np.logical\_and(inside, denominator != 0)  
 inside = np.nan\_to\_num(inside, nan=False)  
 return inside  
  
def calculate\_volume(num\_points):  
 R\_max = 1.5 # Радіус обмежуючого боксу  
 Z\_max = 1.5 # Максимальне значення z  
  
 # Генеруємо випадкові точки в боксі  
 x\_rand = np.random.uniform(-R\_max, R\_max, num\_points)  
 y\_rand = np.random.uniform(-R\_max, R\_max, num\_points)  
 z\_rand = np.random.uniform(0, Z\_max, num\_points)  
  
 # Перевіряємо, чи точки всередині фігури  
 inside = is\_inside\_shape(x\_rand, y\_rand, z\_rand)  
 inside = inside.astype(np.float64)  
  
 # Обчислюємо об'єм  
 volume\_box = (2 \* R\_max) \* (2 \* R\_max) \* Z\_max  
 fraction\_inside = np.mean(inside)  
 volume\_shape = volume\_box \* fraction\_inside  
  
 # Обчислюємо стандартну похибку  
 std\_error = volume\_box \* np.std(inside) / np.sqrt(num\_points)  
  
 return volume\_shape, std\_error  
  
def plot\_shape():  
 # Визначаємо сітку  
 R\_max = 1.5  
 Z\_max = 1.5  
 n = 50  
 x = np.linspace(-R\_max, R\_max, n)  
 y = np.linspace(-R\_max, R\_max, n)  
 z = np.linspace(0, Z\_max, n)  
 X, Y, Z = np.meshgrid(x, y, z)  
  
 # Обчислюємо значення функції  
 numerator = Z\*\*2  
 denominator = X\*\*2 + Y\*\*2  
 with np.errstate(divide='ignore', invalid='ignore'):  
 rhs = numerator / denominator  
 lhs = (X\*\*2 + Y\*\*2 + Z\*\*2)\*\*3  
 F = lhs - rhs  
 F = np.nan\_to\_num(F, nan=1e6, posinf=1e6, neginf=-1e6)  
  
 # Побудова ізоповерхні  
 fig = go.Figure(data=go.Isosurface(  
 x=X.flatten(),  
 y=Y.flatten(),  
 z=Z.flatten(),  
 value=F.flatten(),  
 isomin=-0.1,  
 isomax=0.1,  
 surface\_count=1,  
 colorscale='Viridis',  
 caps=dict(x\_show=False, y\_show=False, z\_show=False),  
 ))  
  
 fig.update\_layout(title='3D модель фігури', scene=dict(  
 xaxis\_title='x',  
 yaxis\_title='y',  
 zaxis\_title='z',  
 ))  
 fig.show()  
  
def run\_calculation():  
 try:  
 num\_points = int(entry\_points.get())  
 if num\_points <= 0:  
 raise ValueError  
 volume, std\_error = calculate\_volume(num\_points)  
 result\_text.set(f"Об'єм фігури: {volume:.6f} ± {std\_error:.6f}")  
 except ValueError:  
 messagebox.showerror("Помилка", "Будь ласка, введіть коректне число точок (позитивне ціле число).")  
  
def plot\_convergence():  
 try:  
 max\_points = int(entry\_max\_points.get())  
 if max\_points <= 0:  
 raise ValueError  
  
 points\_range = np.logspace(3, np.log10(max\_points), num=10, dtype=int)  
 volumes = []  
 errors = []  
  
 for num\_points in points\_range:  
 volume, std\_error = calculate\_volume(num\_points)  
 volumes.append(volume)  
 errors.append(std\_error)  
  
 plt.figure(figsize=(10,6))  
 plt.errorbar(points\_range, volumes, yerr=errors, fmt='o-', ecolor='red', capsize=5)  
 plt.xscale('log')  
 plt.xlabel('Кількість точок')  
 plt.ylabel('Об\'єм фігури')  
 plt.title('Збіжність обчислення об\'єму методом Монте-Карло')  
 plt.grid(True)  
 plt.show()  
  
 except ValueError:  
 messagebox.showerror("Помилка", "Будь ласка, введіть коректне максимальне число точок (позитивне ціле число).")  
  
# Створення GUI  
root = Tk()  
root.title("Обчислення об'єму фігури")  
  
label\_points = Label(root, text="Кількість точок для методу Монте-Карло:")  
label\_points.pack(pady=5)  
  
entry\_points = Entry(root)  
entry\_points.pack(pady=5)  
  
button\_calculate = Button(root, text="Обчислити об'єм", command=run\_calculation)  
button\_calculate.pack(pady=10)  
  
button\_plot = Button(root, text="Побудувати 3D модель", command=plot\_shape)  
button\_plot.pack(pady=10)  
  
label\_max\_points = Label(root, text="Максимальна кількість точок для графіку збіжності:")  
label\_max\_points.pack(pady=5)  
  
entry\_max\_points = Entry(root)  
entry\_max\_points.pack(pady=5)  
  
button\_convergence = Button(root, text="Показати графік збіжності", command=plot\_convergence)  
button\_convergence.pack(pady=10)  
  
result\_text = StringVar()  
label\_result = Label(root, textvariable=result\_text, font=('Arial', 14))  
label\_result.pack(pady=10)  
  
root.mainloop()

**Опис алгоритму:**

1. **Введення параметрів:** Користувач може ввести кількість точок для методу Монте-Карло через графічний інтерфейс.
2. **Обчислення об'єму:** Функція calculate\_volume(num\_points) використовує метод Монте-Карло для оцінки об'єму фігури, генеруючи випадкові точки в обмежуючому кубі і перевіряючи, чи знаходяться вони всередині фігури.
3. **Побудова 3D моделі:** Функція plot\_shape() будує 3D модель фігури, використовуючи сітку точок і ізоповерхню.
4. **Графік збіжності:** Функція plot\_convergence() будує графік збіжності обчисленого об'єму в залежності від кількості точок.

**Результати:**

Програма дозволяє обчислити об'єм фігури методом Монте-Карло з заданою кількістю точок і відобразити результати в графічному інтерфейсі.