Data Science Camp Test Tasks

Ігор Воробець / Ihor Vorobets

Завдання 1.

Розв'язком системи лінійних рівнянь є вектор $X^T = (0 \quad -1 \quad 1)$.
Код програми (файл «task1.py»), яка обчислює розв'язок системи 3 лінійних рівнянь із 3 невідомими для різних значень матриці A і вектора B, подано нижче.

```
import sys
import numpy as np
from numpy.linalg import LinAlgError


def main():
    if len(sys.argv) not in [2, 3]:
        sys.exit(
            "Usage: python task1.py linsys.txt [method]\n"
            "method:\n"
            "  built-in \t - uses numpy built-in function to solve a system of linear equations\n"
            "  matrix \t - computes matrix solution of a system of linear equations\n"
            "  gauss \t - uses Gaussian elimination to solve a system of linear equations\n"
            "  cramer \t - uses Cramer's rule for solution of a system of linear equations"
        )

    # Load matrix a and vector b from text file
    try:
        a, b = load_data(sys.argv[1])
    except FileNotFoundError:
        sys.exit(f"Error! There is no such file '{sys.argv[1]}'.")
    except ValueError:
        sys.exit(
            f"Error! Content of file '{sys.argv[1]}' does not "
             "correspond to number matrix with 3 rows and 4 columns."
        )

    # Get a method for solving the system of linear equations (built-in by default)
    method = sys.argv[2] if len(sys.argv) == 3 else "built-in"

    # Compute solution of the system of linear equations using the method
    if method == "built-in":
        x = np.linalg.solve(a, b)
    elif method == "matrix":
        x = solve_matrix(a, b)
    elif method == "gauss":
        x = solve_gauss(a, b)
    elif method == "cramer":
        x = solve_cramer(a, b)
    else:
        sys.exit(f"Error! '{method}' is not a correct name of method.")

    print(f"Solution of system of linear equations is {x}.")


def load_data(filename):
    """
    Loads from the text file `filename` left matrix and right column
    vector of system of three linear equations.

    Returns tuple `(a, b)`, where `a` is coefficient matrix in linear
    system (NumPy `ndarray` with shape `(3, 3)`), and `b` is vector of
    right-sides of equations (NumPy `ndarray` with shape `(3,)`).
    """

    # Use built-in NumPy function to load data
    data = np.loadtxt(filename)
```

```python
    # Data represents an array which is augmented matrix (with shape (3, 4))
    # of a coefficient matrix and a vector of right-sides of equations

    if data.shape != (3, 4):
        raise ValueError

    # Slice the augmented matrix
    a = data[:3, :3]  # Matrix a contains elements from first 3 rows and first 3 columns
    b = data[:, 3]  # Vector b contains elements from last column

    return a, b


def solve_matrix(a, b):
    """
    Computes matrix solution for a system of linear equations.

    Returns `x` (NumPy `ndarray` with shape `(n,)`), solution, such that
    `a * x = b`, where `n` is number of equations, `a` is coefficient matrix
    (NumPy `ndarray` with shape `(n, n)`), `b` is is vector of right-sides of
    equations (NumPy `ndarray` with shape `(n,)`).
    """
    if a.shape[0] != a.shape[1]:
        raise ValueError("Matrix a is not square")
    if a.shape[0] != b.shape[0]:
        raise ValueError("Number of rows in matrix a is not equal to length of vector b")

    # For matrix solution compute inverse of matrix a, and then just compute
    # product of inverse of matrix a and vector b to get a vector x (solution)

    # Compute inverse of matrix a, using row reduction (Gauss-Jordan elimination),
    # where augmented matrix consists of matrix a and identity matrix n x n

    # After the elementary row operations are performed, matrix a becomes identity
    # matrix, and identity matrix n x n becomes inverse of initial matrix a
    _, a_inv = row_reduction(a, np.eye(a.shape[0]))

    # Solution is product of inverse of matrix a and vector b
    x = a_inv @ b
    return x


def solve_gauss(a, b):
    """
    Solves a system of linear equations using Gaussian elimination.

    Returns `x` (NumPy `ndarray` with shape `(n,)`), solution, such that
    `a * x = b`, where `n` is number of equations, `a` is coefficient matrix
    (NumPy `ndarray` with shape `(n, n)`), `b` is is vector of right-sides of
    equations (NumPy `ndarray` with shape `(n,)`).
    """
    if a.shape[0] != a.shape[1]:
        raise ValueError("Matrix a is not square")
    if a.shape[0] != b.shape[0]:
        raise ValueError("Number of rows in matrix a is not equal to length of vector b")

    # Using Gaussian elimination, create augmented matrix of matrix a and vector b, and
    # perform a sequence of elementary row operations until matrix a reaches row canonical form

    # After the elementary row operations are performed, matrix a in augmented matrix
    # becomes an identity matrix, and vector b becomes a vector x (solution)

    # For row_reduction function convert vector b from vector to matrix n x 1
    _, x = row_reduction(a, b.reshape(b.shape[0], 1))
    return x.reshape(x.shape[0],)  # Convert vector x from matrix to vector


def solve_cramer(a, b):
    """
    Solves a system of linear equations using Cramer's rule.
    Returns `x` (NumPy `ndarray` with shape `(n,)`), solution, such that
    `a * x = b`, where `n` is number of equations, `a` is coefficient matrix
    (NumPy `ndarray` with shape `(n, n)`), `b` is is vector of right-sides of
    equations (NumPy `ndarray` with shape `(n,)`).
    """
    if a.shape[0] != a.shape[1]:
        raise ValueError("Matrix a is not square")
    if a.shape[0] != b.shape[0]:
        raise ValueError("Number of rows in matrix a is not equal to length of vector b")

    # Compute determinant of matrix a
    d = det(a)
    if d == 0:
        raise LinAlgError("Determitat of matrix a is 0 (matrix a is singular)")

    # Compute determinants of matrices formed by replacing each column of of matrix a by vector b
    dx = np.array([])
    for j in range(a.shape[1]):
        ax = np.copy(a)
        ax[:, j] = b[:]
        dx = np.append(dx, det(ax))

    # Solution is results of dividing computed determinants by determinant of matrix a
    x = dx / d
    return x


def row_reduction(matrix1, matrix2):
    """
    Perform elementary row operations on augmented matrix which contains matrices `matrix1`
    (NumPy `ndarray` with shape `(n, n)`) and `matrix2` (NumPy `ndarray` with shape `(n, m)`)
    until `matrix1` reaches row canonical form, e.g. main diagonal of `matrix1` consist of ones
    and all elements above and below the main diagonal in `matrix1` are zeros.

    Returns a tuple of `matrix1` and `matrix2` which have new values.
    """
    if matrix1.shape[0] != matrix1.shape[1]:
        raise ValueError("matrix1 is not a square matrix")
    if matrix1.shape[0] != matrix2.shape[0]:
        raise ValueError("Number of rows in matrix1 is not equal number of rows in matrix2")

    # Augment matrix1 and matrix2
    augmented = np.hstack((matrix1, matrix2))
    size = matrix1.shape[0]  # Size of squared matrix1 and number of rows in augmented matrix

    # Set raising a FloatingPointError for division by zero
    with np.errstate(divide="raise"):
        try:
            # Forward elimination
            for i in range(size - 1):
                for j in range(i + 1, size):
                    augmented[j, :] += augmented[i, :] * (-augmented[j, i] / augmented[i, i])

            # Back substitution
            for i in range(size - 1, 0, -1):
                for j in range(i - 1, -1, -1):
                    augmented[j, :] += augmented[i, :] * (-augmented[j, i] / augmented[i, i])

            # Division by pivots (elements of main diagonal in matrix1)
            for k in range(size):
                augmented[k, :] /= augmented[k, k]

        # Raise a new error, when a FloatingPointError raises
        except FloatingPointError:
            raise LinAlgError("matrix1 contains zero element in main diagonal (matrix1 is singular)")

    # Splitted matrix1 and matrix2 with new values
    return augmented[:, :size], augmented[:, size:]
```

```
def det(matrix):
    """
    Returns determinant of `matrix` (NumPy `ndarray` with shape `(n, n)`) using Laplace expansion.
    """
    if matrix.shape[0] != matrix.shape[1]:
        raise ValueError("matrix is not square")

    # For matrix 1 x 1 determinant is value of single element of matrix
    if matrix.shape[0] == 1:
        return matrix[0, 0]

    # Compute cofactor of elements in 0th row of matrix
    cofactors = np.array([

        # Cofactor equals to minor (determinant of matrix without row and column of element)
        # multiplied by -1 in power of sum of row and column (indices) of element
        det(submat(matrix, 0, j)) * (-1)**j
        for j in range(matrix.shape[1])
    ])

    # Determinant is equal to sum of products of 0th row elements and its cofactors (Laplace expansion)
    return np.dot(matrix[0, :], cofactors)


def submat(matrix, i, j):
    """
    Returns submatrix of `matrix` with `i`th row and `j`th column are eliminated.
    """
    if matrix.shape[0] != matrix.shape[1]:
        raise ValueError("matrix is not square")

    # Mask indicates that elements of i-th row and
    # j-th column are not contained in submatrix
    mask = np.ones_like(matrix, dtype=bool)
    mask[i, :] = False  # Eliminate i-th row
    mask[:, j] = False  # Eliminate j-th column

    # Eliminate i-th row and j-th column using the mask
    return matrix[mask].reshape(matrix.shape[0] - 1, matrix.shape[1] - 1)


if __name__ == "__main__":
    main()
```

Сама програма в якості аргументів командного рядка приймає назву текстового файлу, у якому зберігаються значення A і B у вигляді:

$$\begin{matrix} a_{00} & a_{01} & a_{02} & b_0 \\ a_{10} & a_{11} & a_{12} & b_1 \\ a_{20} & a_{21} & a_{22} & b_2 \end{matrix}$$

Також програма приймає назву метода, за яким буде обчислюватися розв’язок системи. Серед можливих методів розв’язання наступні:

- built-in – використання вбудованої функції NumPy (за замовчуванням)
- matrix – матричний метод
- gauss – метод Гауса
- cramer – метод Крамера

Завдання 2.

Для заданої матриці після 7-ої ітерації зміни значень у матриці сама матриця матиме вигляд:

```
[[0, 0, 0, 0, 0, 1, 1]
 [0, 1, 1, 1, 1, 0, 1]
 [0, 1, 1, 1, 1, 0, 1]
 [0, 0, 0, 0, 1, 1, 0]
 [0, 0, 0, 0, 0, 0, 0]
 [0, 0, 1, 1, 0, 0, 0]
 [0, 1, 0, 1, 1, 0, 0]]
```

Код програми (файл «task2.py»), що ітеративно змінює значення в матриці виводить її після вказаної ітерації, подано нижче. У якості аргументів вона приймає назву текстового файлу, у якому записані значення матриці, та кількість ітерацій, протягом яких потрібно виконувати зміни значень (за замовчуванням 1).

```
import sys
import os
import numpy as np

ALIVE = 1
DEAD = 0

# Command to clear the CLI screen depends on OS type
CLEAR_CMD = "clear" if os.name == "posix" else "cls"


class MatrixSystem:
    """Representation of system which contains a cell matrix."""

    def __init__(self, matrix):
        if matrix.ndim != 2:
            raise ValueError("matrix is not a 2-D array")

        self.matrix = np.copy(matrix)
        self.height, self.width = self.matrix.shape
        self.state = 0

    def change_matrix(self):
        """
        Computes new values of cells in matrix as follows
            - if an alive cell has 2 or 3 alive neighboring cells, it keeps be alive
            - if an alive cell has 1 or 0 alive neighboring cells, it becomes dead due to "loneliness"
            - if an alive cell has 4 or more alive neighboring cells, it becomes dead due to
"overpopulation"
            - if a dead cell has exactly 3 alive neighboring cells, it becomes alive

        Computed new values of cells are stored in the cell matrix.
        """

        # Matrix that represents values of cells in the next state
        new_matrix = np.copy(self.matrix)

        # Loop over all cells of matrix
        for i in range(self.height):
            for j in range(self.width):

                # Number of alive neighboring cells
                alives = self.alive_neighbors((i, j))

                # Alive cell become dead if it has a few or many alive neighboring cells
                if self.matrix[i, j] == ALIVE and (alives <= 1 or alives >= 4):
                    new_matrix[i, j] = DEAD

                # Dead cell become alive if it has three alive neighboring cells
                elif alives == 3:
                    new_matrix[i, j] = ALIVE
        # Transit to the next state matrix and increment number of state
        self.matrix = new_matrix
        self.state += 1

    def alive_neighbors(self, cell):
        """
        Returns number of alive neighboring cells to `cell` which is a pair `(i, j)`.
        Each cell has eight alive neighboring cells.
        """

        # Array of neighboring cells
        neighbors = np.array([

            # Get cell in matrix by index with boundary conditions (use modulo)
            self.matrix[i % self.height, j % self.width]

            # Cells in range of one row and one colums from cell
            for i in range(cell[0] - 1, cell[0] + 2)
            for j in range(cell[1] - 1, cell[1] + 2)
            if (i, j) != cell  # Ignore cell itself
        ])

        # Count a number of neighboring cells which are alive
        return np.sum(neighbors[neighbors == ALIVE])

    def print_matrix(self, title=None):
        """
        Prints values of cell matrix in the command-line interface (CLI).
        Before print the values, runs in CLI the command to clear the screen.

        If `title` is provided, prints `title` together with values of cells.
        Othervise, prints values of cells only.
        """
        os.system(CLEAR_CMD)
        if title:
            print(title, self.matrix, sep="\n")
        else:
            print(self.matrix)


def main():
    if len(sys.argv) not in [2, 3]:
        sys.exit("Usage: python task2.py matrix.txt [iteration]")

    # Get iteration number where matrix change stops (1 by default)
    try:
        iteration = int(sys.argv[2]) if len(sys.argv) == 3 else 1
        if iteration < 1:
            sys.exit("Error! Iteration number is not positive.")
    except ValueError:
        sys.exit("Error! Iteration number is not a correct integer.")

    # Load a matrix from the text file
    try:
        data = load_data(sys.argv[1])
    except FileNotFoundError:
        sys.exit(f"Error! There is no such file '{sys.argv[1]}'.")
    except ValueError:
        sys.exit(
            f"Error! Content of file '{sys.argv[1]}' does not "
            f"correspond to number matrix with values {ALIVE} or {DEAD}."
        )

    # Cell matrix system contains loaded matrix
    system = MatrixSystem(data)

    # Repeatedly change the matrix until reach specified iteration number
    while True:
        system.change_matrix()
        if system.state == iteration:
            break

    system.print_matrix(f"Matrix after iteration {system.state}:")


def load_data(filename):
    """
    Loads cell matrix from the text file `filename`.

    Returns a NumPy `ndarray` which contains a matrix,
    where each cell has `ALIVE` or `DEAD` state.
    """

    # Use built-in NumPy function to load the matrix with int type
    data = np.loadtxt(filename, dtype=int)

    if not np.all((data == ALIVE)^(data == DEAD)):
        raise ValueError

    return data


if __name__ == "__main__":
    main()
```

Завдання 2.1.

Нижче подано код модифікованої програми (файл «task2_1.py»), яка випадково генерує матрицю із заданим розміром та виконує нескінченну симуляцію ітерацій з інтервалом в одну секунду. Аргументами програми є розміри матриці: її висота і ширина.

```
import sys
import os
import time
import numpy as np

ALIVE = 1
DEAD = 0

# Command to clear the CLI screen depends on OS type
CLEAR_CMD = "clear" if os.name == "posix" else "cls"

# Number of seconds to suspend execution of program
WAIT_SECS = 1


class MatrixSystem:
    """Representation of system which contains a cell matrix."""

    def __init__(self, matrix):
        if matrix.ndim != 2:
            raise ValueError("matrix is not a 2-D array")

        self.matrix = np.copy(matrix)
        self.height, self.width = self.matrix.shape
        self.state = 0

    @classmethod
    def generate(cls, shape):
        """
        Creates a `shape` (tuple `(height, width)`) size matrix which
        contains cells with randomly generated values `ALIVE` or `DEAD`.

        Returns instance of `MatrixSystem` with generated matrix.
        """

        # Get random matrix with specified shape
        random_matrix = np.random.choice([ALIVE, DEAD], size=shape)
        # Create a system with the random matrix
        return cls(random_matrix)

    def change_matrix(self):
        """
        Computes new values of cells in matrix as follows
            - if an alive cell has 2 or 3 alive neighboring cells, it keeps be alive
            - if an alive cell has 1 or 0 alive neighboring cells, it becomes dead due to "loneliness"
            - if an alive cell has 4 or more alive neighboring cells, it becomes dead due to
"overpopulation"
            - if a dead cell has exactly 3 alive neighboring cells, it becomes alive

        Computed new values of cells are stored in the cell matrix.
        """

        # Matrix that represents values of cells in the next state
        new_matrix = np.copy(self.matrix)

        # Loop over all cells of matrix
        for i in range(self.height):
            for j in range(self.width):

                # Number of alive neighboring cells
                alives = self.alive_neighbors((i, j))

                # Alive cell become dead if it has a few or many alive neighboring cells
                if self.matrix[i, j] == ALIVE and (alives <= 1 or alives >= 4):
                    new_matrix[i, j] = DEAD

                # Dead cell become alive if it has three alive neighboring cells
                elif alives == 3:
                    new_matrix[i, j] = ALIVE

        # Transit to the next state matrix and increment number of state
        self.matrix = new_matrix
        self.state += 1

    def alive_neighbors(self, cell):
        """
        Returns number of alive neighboring cells to `cell` which is a pair `(i, j)`.
        Each cell has eight alive neighboring cells.
        """

        # Array of neighboring cells
        neighbors = np.array([

            # Get cell in matrix by index with boundary conditions (use modulo)
            self.matrix[i % self.height, j % self.width]

            # Cells in range of one row and one colums from cell
            for i in range(cell[0] - 1, cell[0] + 2)
            for j in range(cell[1] - 1, cell[1] + 2)
            if (i, j) != cell  # Ignore cell itself
        ])

        # Count a number of neighboring cells which are alive
        return np.sum(neighbors[neighbors == ALIVE])

    def print_matrix(self, title=None):
        """
        Prints values of cell matrix in the command-line interface (CLI).
        Before print the values, runs in CLI the command to clear the screen.

        If `title` is provided, prints `title` together with values of cells.
        Othervise, prints values of cells only.
        """
        os.system(CLEAR_CMD)
        if title:
            print(title, self.matrix, sep="\n")
        else:
            print(self.matrix)
```

```
def main():
    if len(sys.argv) != 3:
        sys.exit("Usage: python task2_1.py height width")

    # Get height and width of matrix
    try:
        matrix_height = int(sys.argv[1])
        if matrix_height < 1:
            sys.exit("Error! Height of matrix is not positive.")
    except ValueError:
        sys.exit("Error! Height of matrix is not correct integers.")
    try:
        matrix_width = int(sys.argv[2])
        if matrix_width < 1:
            sys.exit("Error! Width of matrix is not positive.")
    except ValueError:
        sys.exit("Error! Width of matrix is not correct integers.")

    # Create randomly generated cell matrix system with specified size of matrix
    system = MatrixSystem.generate((matrix_height, matrix_width))

    # Output the initial state of matrix
    system.print_matrix("Matrix in initial state:")
    time.sleep(WAIT_SECS)

    # Simulate repeatedly changes of matrix and output values of matrix
    try:
        while True:
            system.change_matrix()
            system.print_matrix(f"Matrix after iteration {system.state}:")
            time.sleep(WAIT_SECS)

    # Exit the program when interrupt key is hitted
    except KeyboardInterrupt:
        sys.exit()


if __name__ == "__main__":
    main()
```

Завдання 2.2.

Програма для візуалізації симуляцій (файл «task2_2.py») використовує модуль Matplotlib та приймає два аргументи: висоту і ширину матриці для здійснення симуляції. Код програми подано нижче.

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

from matplotlib.patches import Patch
from matplotlib.colors import ListedColormap

ALIVE = 1
DEAD = 0

# Number of seconds to pause the figure showing
WAIT_SECS = 1

# Colors of alive and dead cells
COLOR_ALIVE = "limegreen"
COLOR_DEAD = "lightgray"

class MatrixSystem:
    """Representation of system which contains a cell matrix."""

    def __init__(self, matrix):
        if matrix.ndim != 2:
            raise ValueError("matrix is not a 2-D array")
        self.matrix = np.copy(matrix)
        self.height, self.width = self.matrix.shape
        self.state = 0

    @classmethod
    def generate(cls, shape):
        """
        Creates a `shape` (tuple `(height, width)`) size matrix which
        contains cells with randomly generated values `ALIVE` or `DEAD`.

        Returns instance of `MatrixSystem` with generated matrix.
        """

        # Get random matrix with specified shape
        random_matrix = np.random.choice([ALIVE, DEAD], size=shape)

        # Create a system with the random matrix
        return cls(random_matrix)

    def change_matrix(self):
        """
        Computes new values of cells in matrix as follows
            - if an alive cell has 2 or 3 alive neighboring cells, it keeps be alive
            - if an alive cell has 1 or 0 alive neighboring cells, it becomes dead due to "loneliness"
            - if an alive cell has 4 or more alive neighboring cells, it becomes dead due to
"overpopulation"
            - if a dead cell has exactly 3 alive neighboring cells, it becomes alive

        Computed new values of cells are stored in the cell matrix.
        """

        # Matrix that represents values of cells in the next state
        new_matrix = np.copy(self.matrix)

        # Loop over all cells of matrix
        for i in range(self.height):
            for j in range(self.width):

                # Number of alive neighboring cells
                alives = self.alive_neighbors((i, j))

                # Alive cell become dead if it has a few or many alive neighboring cells
                if self.matrix[i, j] == ALIVE and (alives <= 1 or alives >= 4):
                    new_matrix[i, j] = DEAD

                # Dead cell become alive if it has three alive neighboring cells
                elif alives == 3:
                    new_matrix[i, j] = ALIVE

        # Transit to the next state matrix and increment number of state
        self.matrix = new_matrix
        self.state += 1

    def alive_neighbors(self, cell):
        """
        Returns number of alive neighboring cells to `cell` which is a pair `(i, j)`.
        Each cell has eight alive neighboring cells.
        """

        # Array of neighboring cells
        neighbors = np.array([

            # Get cell in matrix by index with boundary conditions (use modulo)
            self.matrix[i % self.height, j % self.width]

            # Cells in range of one row and one colums from cell
            for i in range(cell[0] - 1, cell[0] + 2)
            for j in range(cell[1] - 1, cell[1] + 2)
            if (i, j) != cell  # Ignore cell itself
        ])
        # Count a number of neighboring cells which are alive
        return np.sum(neighbors[neighbors == ALIVE])

    def visualize_simulation(self):
        """
        Performs simulation of matrix changes and visualizes it using Matplotlib.
        During the visualization repeatedly shows images of cell matrix
        at each iteration with a time interval.
        """
        def on_close(event):
            sys.exit()

        # Set on_close function such that the program exits when the figure closes
        plt.connect("close_event", on_close)

        # Create a colormap which contains colors of alive and dead cell
        cell_colors = [COLOR_DEAD, COLOR_ALIVE] if DEAD < ALIVE else [COLOR_ALIVE, COLOR_DEAD]
        cell_cmap = ListedColormap(colors=cell_colors)

        # Create rectangle patches with specified colors
        patches = [
            Patch(color=COLOR_ALIVE, label="Alive"),
            Patch(color=COLOR_DEAD, label="Dead")
        ]

        # Define the legend and set it ub upper left of the figure
        plt.legend(handles=patches, title="Cell", bbox_to_anchor=(-0.05, 1), loc="upper right")

        # Show matrix images on creation when the interactive mode is on
        with plt.ion():

            # Show an image of the initial state of matrix
            if self.state == 0:
                plt.title("Matrix in initial state")
                plt.imshow(self.matrix, origin="upper", cmap=cell_cmap)
                plt.pause(WAIT_SECS)
                self.change_matrix()

            # Simulate repeatedly changes of matrix and show they in the figure
            while True:
                plt.title(f"Matrix after iteration {self.state}")
                plt.imshow(self.matrix, origin="upper", cmap=cell_cmap)
                plt.pause(WAIT_SECS)
                self.change_matrix()


def main():
    if len(sys.argv) != 3:
        sys.exit("Usage: python task2_2.py height width")

    # Get height and width of matrix
    try:
        matrix_height = int(sys.argv[1])
        if matrix_height < 1:
            sys.exit("Error! Height of matrix is not positive.")
    except ValueError:
        sys.exit("Error! Height of matrix is not correct integers.")
    try:
        matrix_width = int(sys.argv[2])
        if matrix_width < 1:
            sys.exit("Error! Width of matrix is not positive.")
    except ValueError:
        sys.exit("Error! Width of matrix is not correct integers.")

    # Create randomly generated cell matrix system with specified size of matrix
    system = MatrixSystem.generate((matrix_height, matrix_width))

    # Start visualization of matrix changes simulation
    system.visualize_simulation()
```

```
if __name__ == "__main__":
    main()
```

Завдання 3.

Ймовірність отримання "Н" при підкиданні монети у наступному випробуванні після проведених випробувань дорівнює $[0.69, 0.79, 0.83, 0.74, 0.80, 0.69, 0.76, 0.80]$.

Ймовірність обчислюється наступним чином. Для проведених випробувань $[\mathrm{H\ H\ H\ T\ H\ T\ H\ H}]$ спочатку обчислюється ймовірність отримання "Н" у першому випробуванні за формулою повної ймовірності. У формулі використовуються значення апріорних ймовірностей вибору монети mi $P(mi) = 1/5$, для всіх i. При визначених умовних ймовірностях отримання "Н" при виборі різних монет ймовірність отримання "Н" дорівнює

$$P(\mathrm{H}) = \sum_i P(mi)P(\mathrm{H}|mi) = 0.48$$

Далі обчислюємо апостеріорні ймовірності вибору монети mi за умови, що було отримано "Н" $P(mi|\mathrm{H})$. Вони обчислюються за формулою Баєса:

$$P(mi|\mathrm{H}) = \frac{P(mi)P(\mathrm{H}|mi)}{\sum_i P(mi)P(\mathrm{H}|mi)} = \frac{P(mi)P(\mathrm{H}|mi)}{P(\mathrm{H})}$$

Для першого випробування апостеріорні ймовірності будуть дорівнювати

$P(m1|\mathrm{H}) \approx 0.042,$

$P(m2|\mathrm{H}) \approx 0.083,$

$P(m3|\mathrm{H}) \approx 0.167,$

$P(m4|\mathrm{H}) \approx 0.333,$

$P(m5|\mathrm{H}) \approx 0.375.$

Далі виконуємо переоцінку ймовірностей вибору певної монети, а саме замінюємо апріорні ймовірності на $P(mi)$ на апостеріорні ймовірності $P(mi|\mathrm{H})$. Повторно обчислюємо ймовірність отримання "Н" у другому випробуванні $P(\mathrm{H}) \approx 0.69$.

Для другого випробування (у якому отримано "Н") обчислюємо оновлені апостеріорні ймовірності $P(mi|\mathrm{H})$, потім визначаємо ймовірність отримання "Н" у третьому випробуванні і так далі для кожного проведеного випробування. Якщо в якомусь випробуванні при підкиданні монети було отримано "Т", то тоді оновлені значення апостеріорних ймовірностей визначатимуться за формулою

$$P(mi|\mathrm{T}) = \frac{P(mi)P(\mathrm{T}|mi)}{\sum_i P(mi)P(\mathrm{T}|mi)} = \frac{P(mi)P(\mathrm{T}|mi)}{P(\mathrm{T})}$$

де ймовірність отримання "Т" при виборі різних монет дорівнює $P(\mathrm{T}|mi) = 1 - (\mathrm{H}|mi)$.

Нижче подано код програми (файл «task3.py»), яка обчислює значення отримання "Н" у наступному випробуванні для заданої послідовності випробувань. У якості аргументу вона приймає назву текстового файлу, де записані значення, отримані у випробуваннях.

```
import sys
import numpy as np
import pandas as pd

HEAD = "H"
TAIL = "T"

# Probabilities to obtain head for each coin to be chosen
HEAD_PROBS = {"m1": 0.1, "m2": 0.2, "m3": 0.4, "m4": 0.8, "m5": 0.9}


def main():
    if len(sys.argv) != 2:
        sys.exit("Usage: python task3.py observations.txt")
    # Load observations array from text file
    try:
        observations = load_data(sys.argv[1])
    except FileNotFoundError:
        sys.exit(f"Error! There is no such file '{sys.argv[1]}'.")
    except ValueError as e:
        sys.exit(f"Error! {e}.")

    # Probability distributions for head and tail for each coin to be chosen
    coin = pd.DataFrame()
    coin[HEAD] = HEAD_PROBS
    coin[TAIL] = 1 - coin[HEAD]  # Each probability of tail equals (1 - probability of head)

    # Prior probabilities to choose a certain coin is equal for all coins
    choose = pd.Series({c: 1 for c in HEAD_PROBS.keys()})
    choose = normalize(choose)  # Each prior probability equals 1 / (number of coins)

    heads = []  # Will contain probabilies to obtain head in next observation
    for observation in observations:

        # Calculate posterior probabilities using Bayes' theorem and update prior probabilities
        choose = normalize(choose * coin[observation])

        # Calculate new probability to obtain head based on previous observations (law of total
probability)
        h = np.sum(choose * coin[HEAD])
        heads.append(round(h, 2))  # Probability is rounded to 2 decimal digits

    print(
        f"For {len(observations)} sequensive coin tosses probabilities "
        f"to obtain head ('{HEAD}') in next observation: {heads}."
    )


def load_data(filename):
    """
    Loads observations data from the text file `filename`.

    Returns a NumPy `ndarray` which contains all observations as strings.
    """

    # Use built-in NumPy function to load the array with string type
    data = np.loadtxt(filename, dtype=str)

    if data.ndim != 1:
        raise ValueError("All observations must be placed in one row")
    if not np.all((data == HEAD)^(data == TAIL)):
        raise ValueError(f"All observations must be heads ('{HEAD}') or tails ('{TAIL}')")

    return data


def normalize(probabilities):
    """
    Normalizes the probability distribution `probabilities`
    (Pandas `Series`) such that all probabilities sum to 1.

    Returns a normalized probability distribution as Pandas `Series`.
    """

    # Normalize by dividing probabilities by their sum
    return probabilities / np.sum(probabilities)


if __name__ == "__main__":
    main()
```