Міністерство освіти і науки України
НТУУ «КПІ ім. Ігоря Сікорського»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Лабораторна робота №3

з дисципліни «Комп’ютерне моделювання»

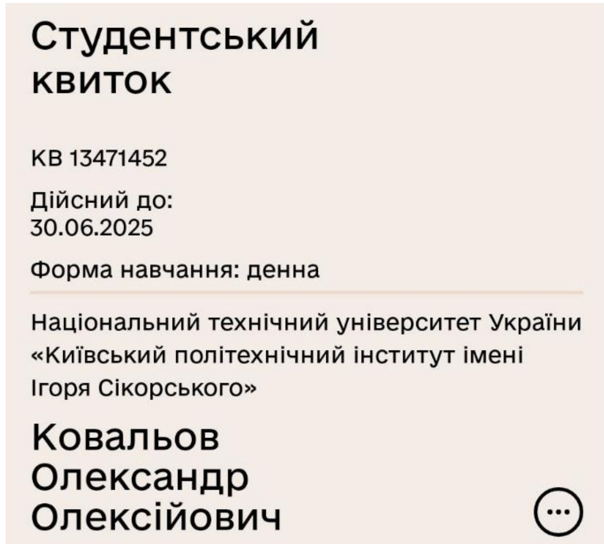
Тема «Комп’ютерне моделювання нестаціонарних процесів»

Варіант №22

Студента 3-го курсу НН ІАТЕ гр. ТР-12
Ковальова Олександра
Перевірив: д.т.н., проф. Шушура О. М.

КИЇВ 2023

Варіант. $g = 3$, $k = 6$ (де g – остання цифра у номері студентського квитка + 1, а k – передостання + 1).



Загальне завдання.

Розробити алгоритми та програмне забезпечення для розв'язку наведених задач. Алгоритми представити у вигляді блок-схем або діаграм діяльності UML. Програмне забезпечення розробити на будь-якій сучасній мові програмування.

Завдання за варіантом ($g = 3, k = 6$)

Розробити алгоритми та програмне забезпечення для розв'язку наведеної задачі Коші методом Ейлера та методом Рунге-Кутта 4-го порядку точності. Середньоквадратична загальна точність пошуку має дорівнювати 0,1. Порівняти розв'язки задачі, знайдені вказанами вище методами. Знайти розв'язок задачі бібліотечними функціями та порівняти його з отриманими результатами.

$$\begin{cases} \dfrac{dx}{dt} = k * t + x - y + g \\ \dfrac{dy}{dt} = -x + k * y \end{cases} \rightarrow \begin{cases} \dfrac{dx}{dt} = 6t + x - y + 3 \\ \dfrac{dy}{dt} = -x + 6y \end{cases}$$

Початкові умови: $x(0) = 0,\ y(0) = 0$, на відрізку $[0, 1]$.

Хід роботи

1. Метод Ейлера

Маємо систему:

$$\begin{cases} \frac{dx}{dt} = 6t + x - y + 3 \\ \frac{dy}{dt} = -x + 6y \end{cases}, \ F = \begin{pmatrix} 6t+x-y+3 \\ 6y-x \end{pmatrix}$$

Кількість кроків будемо рахувати за формулою $k = \frac{b-a}{h}$.

Значення кроку рахуємо таким чином: спочатку запускаємо алгоритм зі значенням 0.1, потім 0.1/2. Порівнюємо останні значення x та y: різниця повинна бути менше заданої точності (наприклад, 0.1). Якщо більше, то запускаємо тест вже з кроком 0.1/2 та 0.1/4. У випадку невдачі продовжуємо рекурсивно.

Результати виконання програми з використанням бібліотеки SciPy:

	STEP	METHOD	T	X	Y
[LIB]	1	Euler's	0.0000	0.0000	0.0000
[LIB]	50	Euler's	1.0000	19.3963	-55.9786

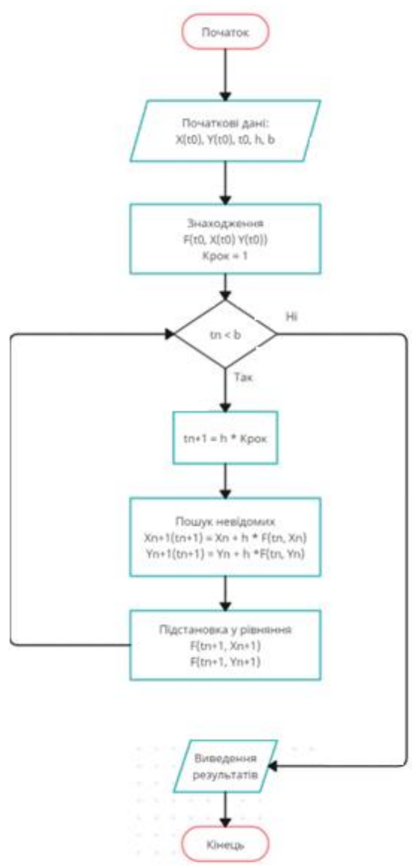
Результати виконання програми за допомогою власного методу:

	STEP	METHOD	T	X	Y
[USER]	1	Euler's	0.0000	0.0000	0.0000
[USER]	20481	Euler's	1.0000	19.4496	-56.2576

Можна зробити висновок, що в бібліотеці використовується поліпшений метод, так як кількість кроків дуже відрізняється.

Заміри були проведені з точністю = 0.1.

Блок-схема.



2. Метод Рунге-Кутта

Кількість і значення кроку вираховуємо таким же чином, як і у минулому кроці.
Результати виконання програми з використанням бібліотеки SciPy:

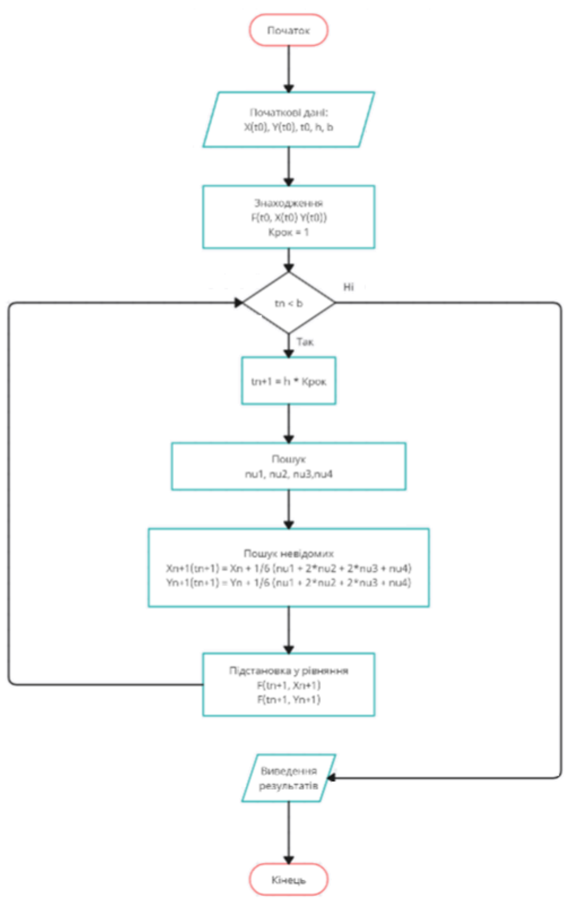
	STEP	METHOD	T	X	Y
[LIB]	1	Runge-Kutta	0.0000	0.0000	0.0000
[LIB]	50	Runge-Kutta	1.0000	19.4590	-56.3038

Результати виконання програми за допомогою власного методу:

	STEP	METHOD	T	X	Y
[USER]	1	Runge-Kutta	0.0000	0.0000	0.0000
[USER]	41	Runge-Kutta	1.0000	19.4602	-56.3105

Висновок: вручну написаний метод за результатом близький до бібліотечного. Все працює.

Блок-схема.



Висновок: Під час виконання лабораторної роботи були написані алгоритми та програмне забезпечення для вирішення задачі Коші методом Ейлера та методом Рунге-Кутта 4-го порядку. Створена программа може обчислювати розв'язки задачі за заданим кроком за початково вказаними даними. Було проведено аналіз розв'язків для власних алгоритмів та бібліотечних методів.

Програмний код

my_euler.py

```
import printer
import utils

# Consts for text formatting
LIB = "[USER]"
METHOD = "Euler's"

# ODE System Variables
G = 0
K = 0


def start(g, k, start_conditions, t_span, accuracy=0.1, SHORTENED_PRINT=False):
    init_ode_system_vars(g, k)

    h = utils.get_delta(accuracy, algorithm)
    steps = utils.get_steps(h, t_span)

    T, X, Y = algorithm(start_conditions, t_span[0], h, steps)

    printer.show(LIB, METHOD, T, X, Y, SHORTENED_PRINT)


# Define G and K in module
def init_ode_system_vars(g, k):
    global G
    global K
    G = g
    K = k


# Define the system of differential equations
def model(t, variables):
    x, y = variables
    dxdt = K * t + x - y + G
    dydt = -x + K * y
    return [dxdt, dydt]


def algorithm(base_system, t, h, steps):
    # Variables for logging
    T_LOG = []
    X_LOG = []
    Y_LOG = []

    system = base_system.copy()

    start_t = t

    F = model(t, system)

    # LOGGING
```

```
    T_LOG.append(t), X_LOG.append(system[0]), Y_LOG.append(system[1])

    for step in range(1, steps + 1):
        for counter in range(0, len(system)):
            system[counter] = system[counter] + h * F[counter]
        t = start_t + step * h
        F = model(t, system)

        # LOGGING
        T_LOG.append(t), X_LOG.append(system[0]), Y_LOG.append(system[1])

    return [T_LOG, X_LOG, Y_LOG]
```

lib_euler.py

```
import numpy as np
from scipy.integrate import solve_ivp

import printer

# Consts for text formatting
LIB = "[LIB]"
METHOD = "Euler's"

# ODE System Variables
G = 0
K = 0


def start(g, k, start_conditions, t_span, accuracy=0.1, SHORTENED_PRINT=False):
    init_ode_system_vars(g, k)

    # Solve using Euler's method
    solution_raw = solve_ivp(model, t_span, start_conditions, args=(G, K),
                             method='RK23', t_eval=np.linspace(0, 1))

    # Extract solutions
    t = solution_raw.t
    x = solution_raw.y[0]
    y = solution_raw.y[1]

    printer.show(LIB, METHOD, t, x, y, SHORTENED_PRINT)


# Define G and K in module
def init_ode_system_vars(g, k):
    global G
    global K
    G = g
    K = k


# Define the system of differential equations
def model(t, cond, g, k):
    x, y = cond
    dxdt = k * t + x - y + g
    dydt = -x + k * y
    return [dxdt, dydt]
```

my_runge_kutta.py

```
import printer
import utils

# Consts for text formatting
```

```python
LIB = "[USER]"
METHOD = "Runge-Kutta"

# ODE System Variables
G = 0
K = 0


def start(g, k, start_conditions, t_span, accuracy=0.1, SHORTENED_PRINT=False):
    init_ode_system_vars(g, k)

    h = utils.get_delta(accuracy, algorithm)
    steps = utils.get_steps(h, t_span)

    T, X, Y = algorithm(start_conditions, t_span[0], h, steps)

    printer.show(LIB, METHOD, T, X, Y, SHORTENED_PRINT)


# Define G and K in module
def init_ode_system_vars(g, k):
    global G
    global K
    G = g
    K = k


# Define the system of differential equations
def model(t, variables):
    x, y = variables
    dxdt = K * t + x - y + G
    dydt = -x + K * y
    return [dxdt, dydt]


def algorithm(base_system, t, h, steps):
    # Variables for logging
    T_LOG = []
    X_LOG = []
    Y_LOG = []

    system = base_system.copy()

    start_t = t

    # LOGGING
    T_LOG.append(t), X_LOG.append(system[0]), Y_LOG.append(system[1])

    for step in range(1, steps + 1):
        F = model(t, system)

        # K0
        k0 = [0] * len(system)
        for i in range(0, len(F)):
            k0[i] = h * F[i]

        # K1
        k1 = [0] * len(system)
        first_member = t + h / 2
        second_member = [None] * len(system)
        for i in range (0, len(k0)):
            second_member[i] = system[i] + k0[i] / 2
        F = model(first_member, second_member)
        for i in range(0, len(F)):
```

```
                k1[i] = h * F[i]

            # K2
            k2 = [0] * len(system)
            first_member = t + h / 2
            second_member = [None] * len(system)
            for i in range(0, len(k1)):
                second_member[i] = system[i] + k1[i] / 2
            F = model(first_member, second_member)
            for i in range(0, len(F)):
                k2[i] = h * F[i]

            # K3
            k3 = [0] * len(system)
            first_member = t + h
            second_member = [None] * len(system)
            for i in range(0, len(k2)):
                second_member[i] = system[i] + k2[i]
            F = model(first_member, second_member)
            for i in range(0, len(F)):
                k3[i] = h * F[i]

            # SYSTEM
            t = start_t + step * h
            for counter in range(0, len(system)):
                system[counter] = system[counter] + (k0[counter] + 2*k1[counter] +
2*k2[counter] + k3[counter])/6

            # LOGGING
            T_LOG.append(t), X_LOG.append(system[0]), Y_LOG.append(system[1])

        return [T_LOG, X_LOG, Y_LOG]
```

lib_runge_kutta.py

```
import numpy as np
from scipy.integrate import solve_ivp

import printer

# Consts for text formatting
LIB = "[LIB]"
METHOD = "Runge-Kutta"

# ODE System Variables
G = 0
K = 0


def start(g, k, start_conditions, t_span, accuracy=0.1, SHORTENED_PRINT=False):
    init_ode_system_vars(g, k)

    # Solve using Runge-Kutta (RK45) method
    solution_raw = solve_ivp(model, t_span, start_conditions, args=(G, K),
                             method='RK45', t_eval=np.linspace(0, 1))

    # Extract solutions
    t = solution_raw.t
    x = solution_raw.y[0]
    y = solution_raw.y[1]

    printer.show(LIB, METHOD, t, x, y, SHORTENED_PRINT)


# Define G and K in module
```

```
def init_ode_system_vars(g, k):
    global G
    global K
    G = g
    K = k


# Define the system of differential equations
def model(t, cond, g, k):
    x, y = cond
    dxdt = k * t + x - y + g
    dydt = -x + k * y
    return [dxdt, dydt]
```