Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №2
по дисциплине «Моделирование»

Тема: Цепи Маркова

Группа: ИУ7-73Б

Студент: Сукочева Алис

Преподаватель: Рудаков И. В.

Москва.
2021 г.

Задание:

Реализовать программу, позволяющую определить время пребывания сложной системы, работающей на базе цепей Маркова, во всех ее состояниях, определить момент достижения вероятностной константы, а также ее значение.

Теоретическая часть

Случайный процесс, протекающий в системе S, называется марковским, если он обладает следующим свойством: для каждого момента времени t_0 вероятность любого состояния системы в будущем (при $t > t_0$) зависит только от ее состояния в настоящем (при $t = t_0$) и не зависит от того, когда и каким образом система пришла в это состояние. Вероятностью i-го состояния называется вероятность $p_i(t)$ того, что в момент t система будет находиться в состоянии S_i. Для любого момента t сумма вероятностей всех состояний равна единице.

Для решения поставленной задачи, необходимо составить систему уравнений Колмогорова по следующим принципам: в левой части каждого из уравнений стоит производная вероятности i-го состояния; в правой части — сумма произведений вероятностей всех состояний (из которых идут стрелки в данное состояние), умноженная на интенсивности соответствующих потоков событий, минус суммарная интенсивность всех потоков, выводящих систему из данного состояния, умноженная на вероятность данного (i-го состояния).

Пример

Система имеет 3 состояния с матрицей интенсивностей, описанной в табл. 1.

Таблица 1 – матрица интенсивностей

0	λ_{01}	λ_{02}
λ_{10}	0	λ_{12}
λ_{20}	λ_{21}	0

$$\begin{cases} p_0' = -(\lambda_{01} + \lambda_{02})p_0 + \lambda_{10}p_1 + \lambda_{20}p_2 \\ p_1' = -(\lambda_{10} + \lambda_{12})p_1 + \lambda_{01}p_0 + \lambda_{21}p_2 \\ p_2' = -(\lambda_{20} + \lambda_{21})p_2 + \lambda_{02}p_0 + \lambda_{12}p_1 \end{cases} \qquad (1)$$

Для получения предельных вероятностей, то есть вероятностей в стационарном режиме работы при $t \to \infty$, необходимо приравнять левые части уравнений к нулю. Таким образом получается система линейных уравнений. Для решения полученной системы необходимо добавить условие нормировки ($p_0 + p_1 + p_2 = 1$).

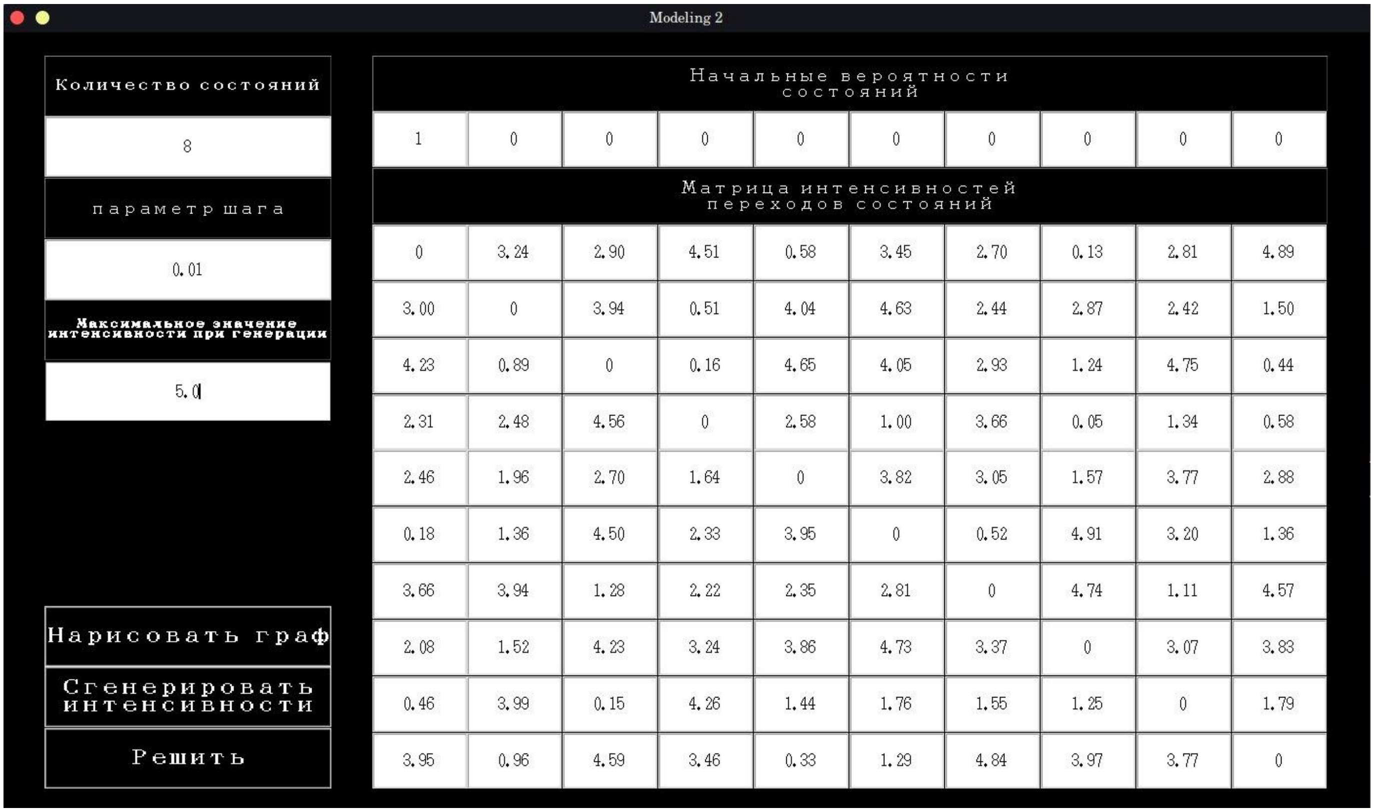
Также необходимо найти время, в которое достигается вероятностная константа. В общем случае для решения данной задачи необходимо решить систему ОДУ (1) в общем виде. Даже библиотечные функции (в ЯП Python) не находят функции как таковые, а находят только значения для функций из системы ОДУ в заданных точках, поэтому для решения данной задачи можно воспользоваться численным методом: найдем все значения вероятности p_i, как функции времени, с интервалом Δt в некотором интервале [t_0, t_N]. Когда найденная вероятность будет равна найденной (при итерации с конца!) ранее вероятностной константе с точностью до заданной погрешности, тогда можно считать искомое время найденным. На каждом шаге необходимо вычислять приращения для каждой вероятности:

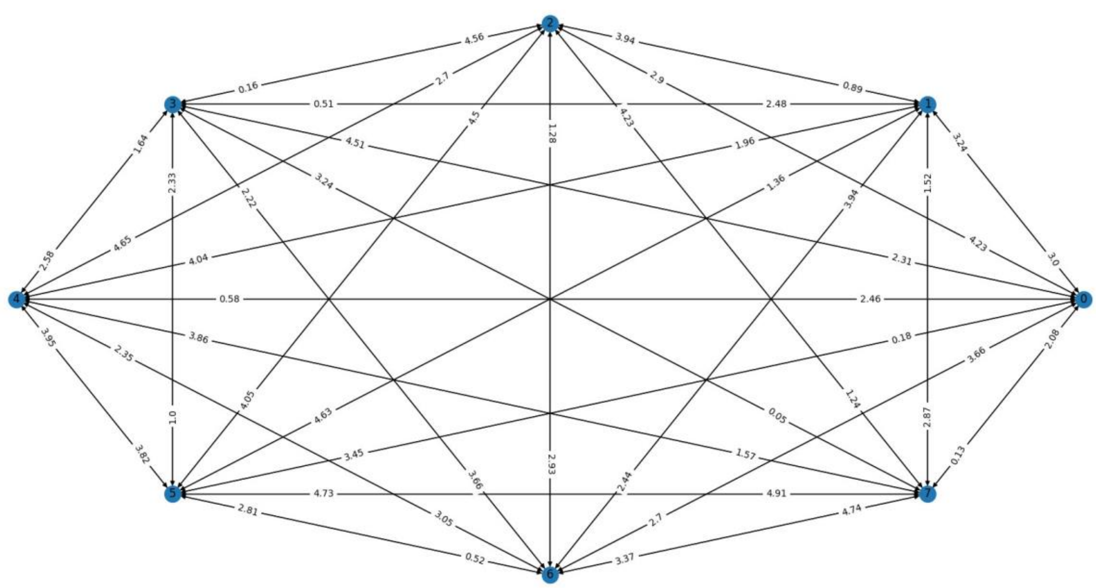
$$dp_i = \frac{-(\lambda_{01}+\lambda_{02})p_0+\lambda_{10}p_1+\lambda_{20}p_2}{\Delta t}.$$

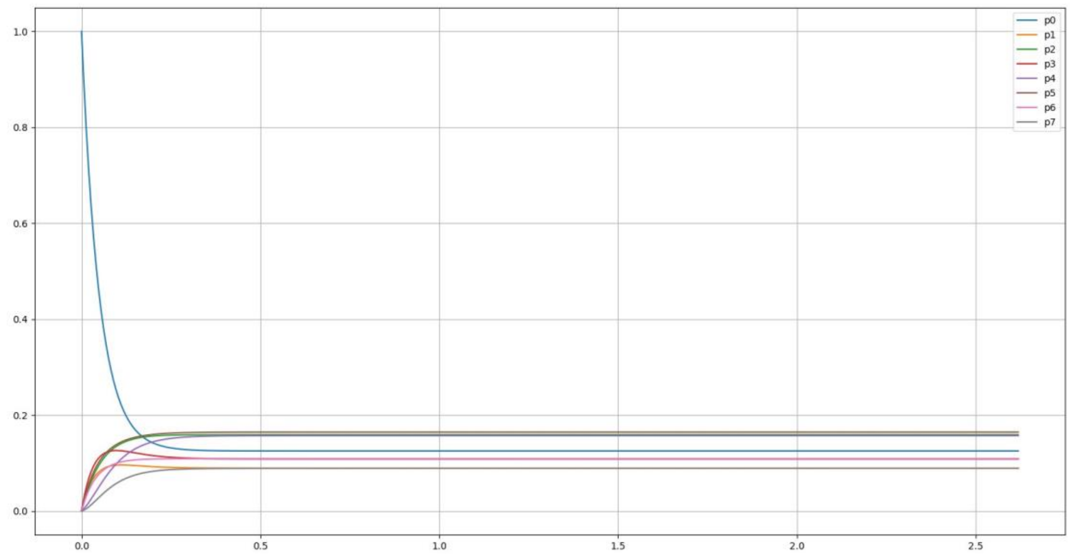
Начальные значения для Δtразумно брать на основе интенсивностей в системе, будет рассмотрено далее.

Пример работы

1. Общий случай

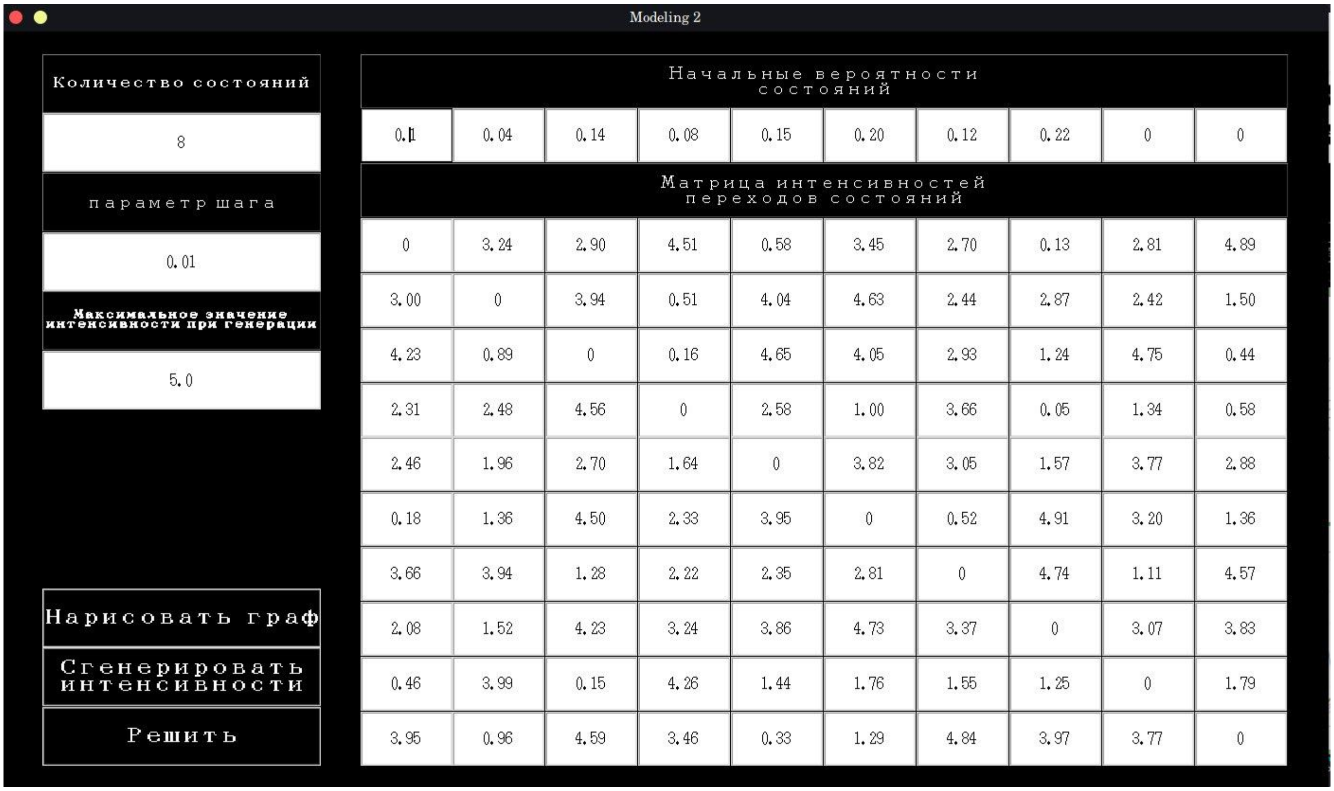


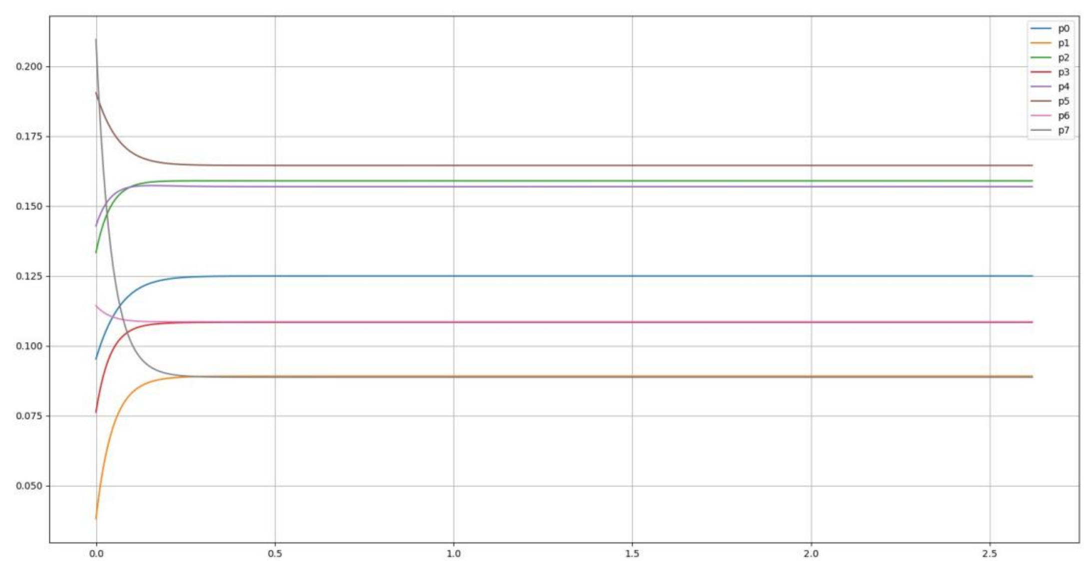





```
☁ sources [main] ⚡ python3 main.py
Стабильное состояние:
p0 = 0.12; p1 = 0.09; p2 = 0.16; p3 = 0.11; p4 = 0.16; p5 = 0.16; p6 = 0.11; p7 = 0.09;
Времена достижения стабильного состояния:
t0 = 0.35; t1 = 0.28; t2 = 0.25; t3 = 0.33; t4 = 0.35; t5 = 0.30; t6 = 0.17; t7 = 0.30;
```

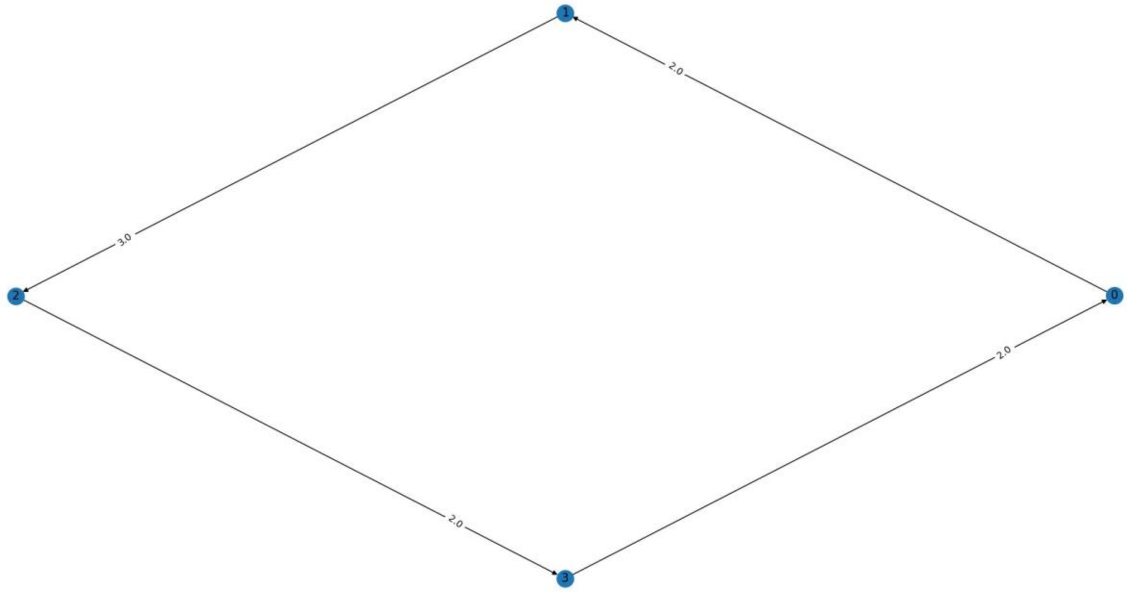
2. Общий случай с неравномерным случайным начальным состоянием

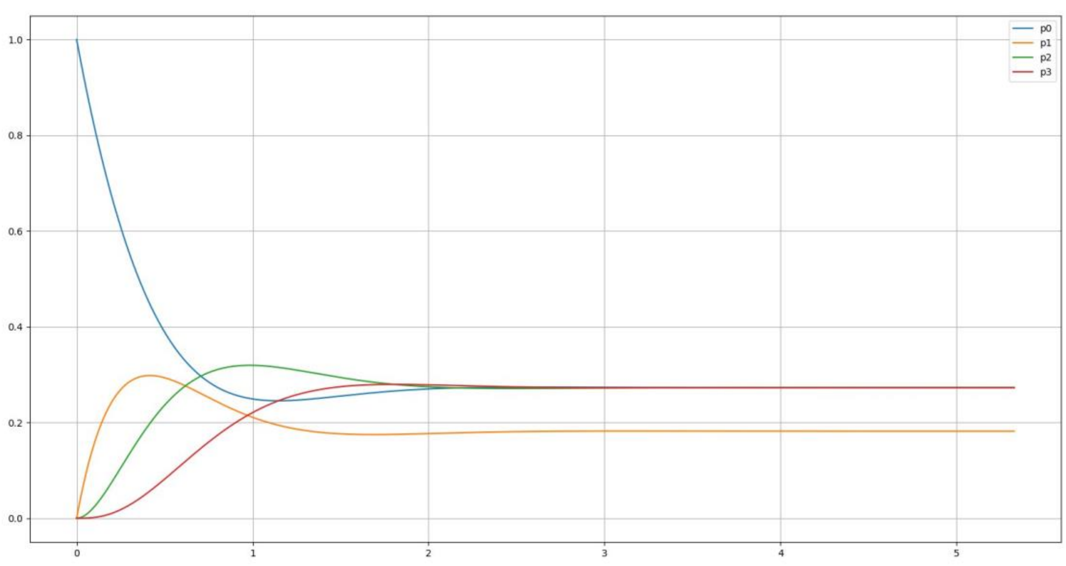




```
Стабильное состояние:
p0 = 0.12; p1 = 0.09; p2 = 0.16; p3 = 0.11; p4 = 0.16; p5 = 0.16; p6 = 0.11; p7 = 0.09;
Времена достижения стабильного состояния:
t0 = 0.20; t1 = 0.19; t2 = 0.12; t3 = 0.14; t4 = 0.07; t5 = 0.18; t6 = 0.06; t7 = 0.21;
```

3. Кольцо.






```
☁ sources [main] ⚡ python3 main.py
Стабильное состояние:
p0 = 0.27; p1 = 0.18; p2 = 0.27; p3 = 0.27;
Времена достижения стабильного состояния:
t0 = 2.74; t1 = 2.49; t2 = 2.86; t3 = 2.59;
```

Текст программы

main.py :

```
import random
import math
import tkinter as tk
import config as cfg
import tkinter.messagebox as mb
import matplotlib.pyplot as plt
import numpy as np
from algorithm import solve
import networkx as nx


root = tk.Tk()
root.title("Modeling 2")
root["bg"] = cfg.MAIN_COLOUR
root.geometry(str(cfg.WINDOW_WIDTH) + "x" + str(cfg.WINDOW_HEIGHT))
root.resizable(height=False, width=False)


data_frame = tk.Frame(root)
data_frame["bg"] = cfg.MAIN_COLOUR


matrix_frame = tk.Frame(root)
matrix_frame["bg"] = cfg.MAIN_COLOUR


data_frame.place(x=int(cfg.BORDERS_WIDTH), y=int(cfg.BORDERS_HEIGHT),
                 width=cfg.DATA_WIDTH,
                 height=cfg.DATA_HEIGHT)


matrix_frame.place(x=int(cfg.BORDERS_WIDTH * 2 + cfg.DATA_WIDTH),
y=int(cfg.BORDERS_HEIGHT),
                 width=cfg.MATRIX_WIDTH,
                 height=cfg.DATA_HEIGHT)


def process():
    global matrix_entries, dt_entry, matrix_size_entry, start_probs_entries
    size = int(matrix_size_entry.get())
    matrix = [[float(matrix_entries[i][j].get()) for j in range(size)] for i in
```

```
range(size)]
    start_probs = [float(start_probs_entries[i].get()) for i in range(size)]
    dt = float(dt_entry.get())
    solve(matrix, start_probs, dt)




def draw_graph():
    global matrix_entries, matrix_size_entry
    size = int(matrix_size_entry.get())
    matrix = [[float(matrix_entries[i][j].get()) for j in range(size)] for i in
range(size)]


    G = nx.from_numpy_matrix(np.matrix(matrix), create_using=nx.DiGraph)
    layout = nx.circular_layout(G)
    nx.draw(G, layout, with_labels=True)
    nx.draw_networkx_edge_labels(G, pos=layout,
edge_labels=nx.get_edge_attributes(G,'weight'), label_pos=0.2)
    plt.show()




start_probs_entries = [
    tk.Entry(matrix_frame, bg=cfg.ADD_COLOUR, font=("Arial", 12), fg=cfg.MAIN_COLOUR,
justify="center")
    for i in range(cfg.MATRIX_MAX_SIZE)
]


for i in range(1, cfg.MATRIX_MAX_SIZE):
    start_probs_entries[i].insert(0, '0')
start_probs_entries[0].insert(0, '1')


matrix_entries = [
    [
        tk.Entry(matrix_frame, bg=cfg.ADD_COLOUR, font=("Arial", 12),
fg=cfg.MAIN_COLOUR, justify="center")
        for i in range(cfg.MATRIX_MAX_SIZE)
    ]
    for j in range(cfg.MATRIX_MAX_SIZE)
]
```

```
for i in range(cfg.MATRIX_MAX_SIZE):
    for j in range(cfg.MATRIX_MAX_SIZE):
        matrix_entries[i][j].insert(0, '0')


def generate_random():
    global lambda_limit_entry, matrix_entries
    limit = float(lambda_limit_entry.get())
    for i in range(cfg.MATRIX_MAX_SIZE):
        for j in range(cfg.MATRIX_MAX_SIZE):
            matrix_entries[i][j].delete(0, len(matrix_entries[i][j].get()))
            matrix_entries[i][j].insert(0, '0' if i == j else f"{random.random() * lim-
it:.2f}")


matrix_size_entry = tk.Entry(data_frame, bg=cfg.ADD_COLOUR, font=("Arial", 12),
                      fg=cfg.MAIN_COLOUR, justify="center")
matrix_size_entry.insert(0, '2')


dt_entry = tk.Entry(data_frame, bg=cfg.ADD_COLOUR, font=("Arial", 12),
                    fg=cfg.MAIN_COLOUR, justify="center")
dt_entry.insert(0, '0.1')


lambda_limit_entry = tk.Entry(data_frame, bg=cfg.ADD_COLOUR, font=("Arial", 12),
                    fg=cfg.MAIN_COLOUR, justify="center")
lambda_limit_entry.insert(0, '1.0')


lambda_limit_label = tk.Label(data_frame, text="Максимальное значение\nинтенсивности
при генерации",
                       font=("Arial", 8), bg=cfg.MAIN_COLOUR,
                       fg=cfg.ADD_COLOUR, relief=tk.GROOVE)


start_probs_label = tk.Label(matrix_frame, text="Начальные вероятности\nсостояний",
                       font=("Arial", 12), bg=cfg.MAIN_COLOUR,
                       fg=cfg.ADD_COLOUR, relief=tk.GROOVE)


matrix_label = tk.Label(matrix_frame, text="Матрица интенсивностей\nпереходов
состояний", font=("Arial", 12),
```

```
                        bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, relief=tk.GROOVE)


matrix_size_label = tk.Label(data_frame, text="Количество состояний", font=("Arial",
10),
                        bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, relief=tk.GROOVE)


dt_label = tk.Label(data_frame, text="параметр шага", font=("Arial", 12),
                        bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, relief=tk.GROOVE)


lambda_gen_button = tk.Button(data_frame, text="Сгенерировать\nинтенсивности",
font=("Consolas", 14),
                      bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, command=generate_random,
                      activebackground=cfg.ADD_COLOUR, activefore-
ground=cfg.MAIN_COLOUR)
draw_graph_button = tk.Button(data_frame, text="Нарисовать граф", font=("Consolas",
14),
                      bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, command=draw_graph,
                      activebackground=cfg.ADD_COLOUR, activefore-
ground=cfg.MAIN_COLOUR)
solve_button = tk.Button(data_frame, text="Решить", font=("Consolas", 14),
                      bg=cfg.MAIN_COLOUR, fg=cfg.ADD_COLOUR, command=process,
                      activebackground=cfg.ADD_COLOUR, activefore-
ground=cfg.MAIN_COLOUR)






offset = 0
matrix_size_label.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
matrix_size_entry.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
dt_label.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS, width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
dt_entry.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS, width=cfg.DATA_WIDTH,
```

```
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
lambda_limit_label.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
lambda_limit_entry.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)


offset = cfg.ROWS - 3
draw_graph_button.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
lambda_gen_button.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS,
width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)
offset += 1
solve_button.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.ROWS, width=cfg.DATA_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.ROWS)


offset = 0


start_probs_label.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.MATRIX_FRAME_ROWS,
width=cfg.MATRIX_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.MATRIX_FRAME_ROWS)


offset += 1


for i in range(cfg.MATRIX_MAX_SIZE):
    start_probs_entries[i].place(x=int(i / cfg.MATRIX_MAX_SIZE * cfg.MATRIX_WIDTH),
y=cfg.DATA_HEIGHT * offset // cfg.MATRIX_FRAME_ROWS,
                                 width=cfg.MATRIX_WIDTH // cfg.MATRIX_MAX_SIZE,
height=cfg.DATA_HEIGHT // cfg.MATRIX_FRAME_ROWS)


offset += 1
```

```
matrix_label.place(x=0, y=cfg.DATA_HEIGHT * offset // cfg.MATRIX_FRAME_ROWS,
width=cfg.MATRIX_WIDTH,
                  height=cfg.DATA_HEIGHT // cfg.MATRIX_FRAME_ROWS)


offset += 1


for i in range(cfg.MATRIX_MAX_SIZE):
    for j in range(cfg.MATRIX_MAX_SIZE):
        matrix_entries[i][j].place(x=int(j / cfg.MATRIX_MAX_SIZE * cfg.MATRIX_WIDTH),
                                   y=cfg.DATA_HEIGHT * (offset + i) //
cfg.MATRIX_FRAME_ROWS,
                                   width=cfg.MATRIX_WIDTH // cfg.MATRIX_MAX_SIZE,
                                   height=cfg.DATA_HEIGHT // cfg.MATRIX_FRAME_ROWS)
root.mainloop()
```

algorithm.py :

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt


eps = 1e-3


def normalize(probs):
    s = sum(probs)
    for i in range(len(probs)):
        probs[i] /= s
```

```
def solve_ode(matrix, start_probs, dt, steady_states):
    matrix_to_solve = [
        [-sum(matrix[i]) if j == i else matrix[j][i] for j in range(len(matrix))]
        for i in range(len(matrix))]
    ts = np.arange(0, dt * 1000, dt)
    results = odeint(vectorfields, start_probs, ts,
                     args=(matrix_to_solve,), atol=1.0e-8, rtol=1.0e-6)
    results = np.transpose(results)
    steady_ts = []


    for i in range(len(results)):
        plt.plot(ts, results[i], label='p' + str(i))
        row = results[i]
        flag = True
        for j in range(len(row) - 1, -1, -1):
            if abs(steady_states[i] - row[j]) > eps:
                steady_ts.append(ts[j])
                flag = False
                break
        if flag:
            steady_ts.append(0)


    print("Времена достижения стабильного состояния:")
    for i in range(len(steady_ts)):
        print(f"t{str(i)} = {steady_ts[i]:.2f}; ", end='')
    print()


    plt.legend()
    plt.grid()
    plt.show()


def vectorfields(w, _, matrix_to_solve):
    """
    Defines the differential equations for the coupled spring-mass system.
    Arguments:
        w :  vector of the state variables:
                  w = [p1, p2, p3...]
        _ :  time
        matrix_to_solve :  vector of the parameters:
                  lambdas = [[lambda_ji for i in range(len(matrix))] for j in
```

```
range(len(matrix))]
    """
    f = []
    for i in range(len(w)):
        f.append(0)
        for p, lambda_coeff in zip(w, matrix_to_solve[i]):
            f[i] += p * lambda_coeff


    return f


def solve(matrix, start_probs, dt):
    normalize(start_probs)
    b = [0 for _ in range(len(matrix) - 1)]
    b.append(1)
    matrix_to_solve = [
        [-sum(matrix[i]) if j == i else matrix[j][i] for j in range(len(matrix))]
        for i in range(len(matrix) - 1)]
    matrix_to_solve.append([1 for _ in range(len(matrix))])


    ps = np.linalg.solve(matrix_to_solve, b)
    print("Стабильное состояние:")
    for i in range(len(ps)):
        print(f"p{str(i)} = {ps[i]:.2f}; ", end='')
    print()
    max_lambda = max([max(matrix[i]) for i in range(len(matrix))])
    avg_lambda = sum([sum(matrix[i]) for i in range(len(matrix))]) / len(matrix) /
(len(matrix) - 1)
    dt = (1 / (max_lambda + avg_lambda) * 2) * dt
    solve_ode(matrix, start_probs, dt, ps)
```