МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Радиофизический факультет
Кафедра теории колебаний и автоматического регулирования

Направление «Фундаментальная информатика и информационные технологии»

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Преддипломная практика

АППРОКСИМАЦИЯ НЕПРЕРЫВНОЙ МОДЕЛИ МУЛЬТИВИБРАТОРА МОДЕЛЬЮ С КОНЕЧНЫМ МНОЖЕСТВОМ СОСТОЯНИЙ

Научный руководитель:
доктор физико-математических наук,
профессор
Кафедра теории колебаний и
автоматического регулирования — Канаков Олег Игоревич

Студент 4-го курса — Семиков Алексей Александрович

Нижний Новгород, 2024

Оглавление

Введение

Динамическая система представляет собой такую математическую модель некоего объекта, процесса или явления, в которой пренебрегают «флуктуациями и всеми другими статистическими явлениями».

Динамическая система также может быть представлена как система, обладающая состоянием. При таком подходе динамическая система описывает (в целом) динамику некоторого процесса, а именно: процесс перехода системы из одного состояния в другое. Фазовое пространство системы — совокупность всех допустимых состояний динамической системы. Таким образом, динамическая система характеризуется своим начальным состоянием и законом, по которому система переходит из начального состояния в другое.

Различают системы с дискретным временем и системы с непрерывным временем.

В данной работе решается задача аппроксимации непрерывной динамической системы моделью с конечным множеством состояний. Аппроксимация позволяет исследовать числовые характеристики и качественные свойства объекта, сводя задачу к изучению более простых или более удобных объектов. Во-первых, это дает возможность более простого моделирования, а во-вторых, это инструмент исследования динамики исходной модели.

Методы решения задачи

Для решения задачи аппроксимации непрерывной модели мультивибратора моделью с конечным множеством состояний существуют прямые и косвенные методы [2, с. 77].

Прямые методы работают с исходной динамикой системы, начиная с набора начальных состояний и применяя оператор, который вычисляет набор состояний, достижимых из этих состояний, следуя непрерывной динамике, до тех пор, пока не будет достигнута фиксированная точка (или не будет). Данный подход работает для систем с очень простой непрерывной динамикой, однако для большинства классов проблема все еще остается неразрешимой из-за сочетания такой динамики с дискретными переходами.

Косвенные методы преобразуют исходную модель системы в абстрактную модель, принадлежащую более простому классу, проверка которой проще и зачастую разрешима. Наиболее часто используемый класс абстрактных моделей — это автоматы с конечным множеством состояний. Основное преимущество косвенного подхода состоит в том, что более простые классы моделей, например, автоматы с конечным числом состояний, допускают хорошо известные алгоритмы проверки моделей, реализуемые с помощью многочисленных зрелых инструментов, в то время как адаптация таких методов к системам с нетривиальной непрерывной динамикой значительно сложнее.

Динамическая система

Для моделирования динамической системы был применён косвенный метод. Система описывается парой нелинейных дифференциальных уравнений первого порядка. Переменные *x* и *y* представляют собой состояния системы, которые изменяются со временем, а μ является параметром системы, который может влиять на характер её поведения. Уравнения системы имеют следующий вид:

$$\begin{cases} \mu\dot{x} = -x(x^2 - 5) - y \\ \dot{y} = x \end{cases} \quad (1)$$

Вместе эти уравнения формируют систему, которая может иметь различные режимы поведения, и анализ такой системы требует применения численных методов для решения дифференциальных уравнений, таких как метод Эйлера.

Система уравнений была решена итерационным методом, используя метод Эйлера с дискретным шагом h = 0.001:

$$x_{n+1} = x_n + \frac{f_1(x,y)*\mathrm{h}}{\mu} \quad (2)$$

$$y_{n+1} = y_n + \frac{f_2(x,y)*\mathrm{h}}{\mu} \quad (3)$$

где функции f_1и f_2 имеют вид:

$$\begin{cases} f_1(x,y) = -x(x^2 - 5) - y \\ f_2(x) = x \end{cases} \quad (4)$$

Программная реализация предложенного метода для анализа рассматриваемой динамической системы заключается в поиске решений внутри заданной прямоугольной области с использованием определенного дискретного шага.

В ходе итерационного метода начальная точка последовательно изменяется на каждом этапе цикла. Расчет семейства начальных точек происходит в функции *calc_start_point(polygon, grid_size, cell_size)*. Из центральной точки каждой клетки в пределах выделенной прямоугольной области производится расчет множества решений системы (4).

На графике (рис. 1) представлены результаты динамической системы для 10 000 различных начальных точек, демонстрируя разнообразие траекторий движения в состояниях системы.

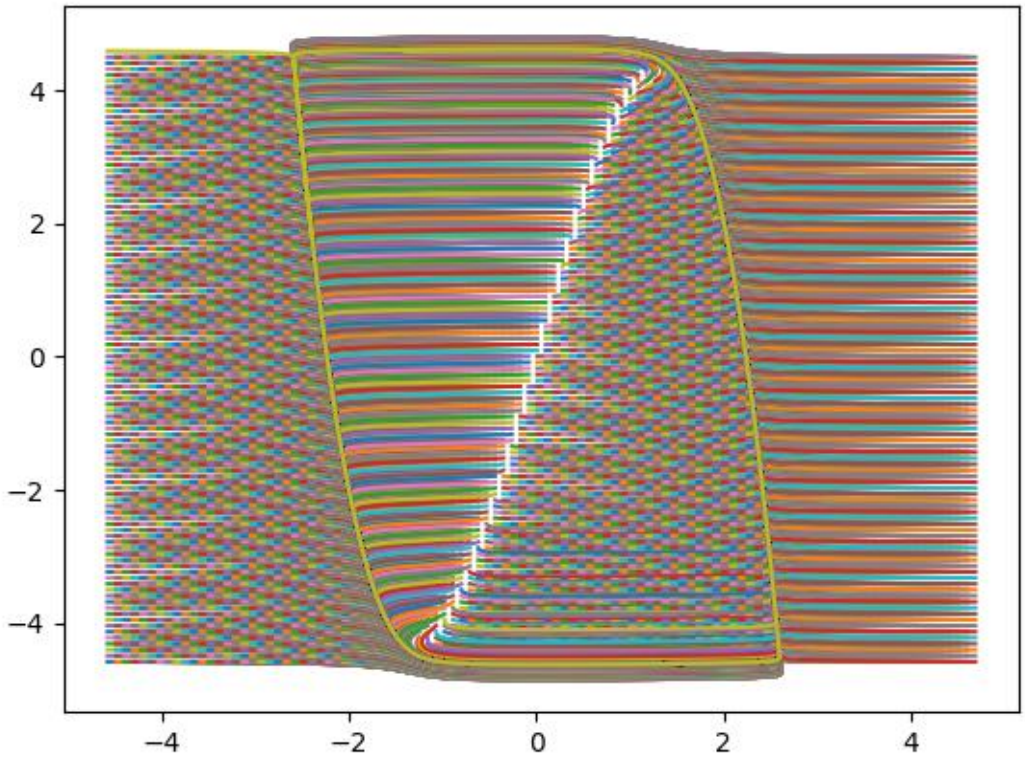

Рис. 1 Множество решений динамической системы

Построение дискретной модели

Дискретная модель – это математическая модель, которая описывает систему с помощью дискретного набора данных. Такие модели используют переменные, значения которых ограничены определенным способом, часто целыми числами или конечным набором состояний. В контексте рассматриваемой задачи данная модель служит инструментом для исследования исходной динамической системы, а также упрощает моделирование.

Для получения дискретного и конечного описания набора состояний необходимо выполнить ограничение и подразделение непрерывного и бесконечного пространства. Данная задача решается с помощью прямоугольных ячеек, или клеток. Прямоугольные ячейки не обязательно являются лучшим выбором, однако по соображениям программной реализации такие ячейки представляются гораздо более удобной структурой данных. Ограничение конечной областью достигается за счет определения пользователем начальной области, включающей в себя интересующее поведение представленной динамической системы. [3, с. 405]

Для успешного выполнения поставленной задачи была разработана программа, обладающая удобным пользовательским интерфейсом, что значительно улучшает взаимодействие с ней. Программа разделена на три основных модуля: *main.py* отвечает за интерфейс и взаимодействие пользователя с программой, *approx.py* включает в себя набор функций для проведения моделирования, *params.py* описывает начальные условия и параметры системы, которые пользователь может настраивать с помощью интерфейса программы. Соответствующие файлы представлены в Приложении.

Для создания пользовательского интерфейса была использована библиотека *prompt_toolkit*, которая позволяет создавать интерактивные диалоговые окна в командной строке. Взаимодействие представляется в виде списка для выбора действий и диалоговых окон для ввода новых значений

параметров. Интерфейс программы спроектирован таким образом, чтобы пользователь мог легко взаимодействовать с программой через командную строку без необходимости вносить изменения непосредственно в код. Это делает программу более удобной и доступной.

Фазовая плоскость заменяется сеткой с конечным количеством состояний. Переходы системы между состояниями будет описываться матрицей. Создания матрицы осуществляется с использованием функции *create_grid(grid, repeat_points, grid_size, repeat_points_temp, cell_size, start_point_arr, dtype)*. Функция принимает в качестве аргументов: незаполненную матрицу переходов, пустой массив для записи неоднозначного поведения в ходе функционирования модели, размер матрицы, временный массив, размер клетки, массив начальных точек, тип массива соответственно. Метод Эйлера интегрирован непосредственной в функцию для оптимизации процесса.

В рамках решения задачи для оптимизации вычислительных процессов в программном коде был применен высокопроизводительный компилятор Numba, который преобразует функции Python в оптимизированный машинный код. Используя стандартную библиотеку LLVM (Low-Level-Virtual-Machine) для компиляции и создавая специализированный код для разных типов данных и раскладок, Numba позволяет значительно ускорить работу циклов и вычислений. Также программный код включает в себя определение структурированных массивов, элементы которых имеют различный тип данных. Такой подход позволяет более эффективно обрабатывать сложные данные в вычислительных процессах. Структуры массивов определены следующим образом:

1. Массив *grid:*
    a. элемент *“array”* – массив, состоящий из четырех 16-битных целых чисел
    b. элементы *“iteration”, “pathway”* – 16-битные целые числа

2. Массив *repeat_points – [(“y”, np.int16), (“x”, np.int16), (“curr_arr”, np.int16), (“edit_arr”, np.int16)]*
    a. элементы *“x”* и *“y”* – 16-битные целые числа
    b. элементы *“curr_arr”, “edit_arr”* – массивы, состоящие из четырех 16-битных чисел

Листинг 1. Структуры массивов

```
dtype_for_matrix = np.dtype( [       # Структура для
    ('array', np.int16, (4)),        # матрицы
    ('iteration', np.int32),
    ('pathway', np.int16)
] )

dtype_for_repeat_points = np.dtype( [        # Структура для
    ('y', np.int16), ('x', np.int16),        # массива с
    ('curr_arr', np.int16, (4)),             # повторяющимися точками
    ('edited_arr', np.int16, (4))
] )


dtype_for_start_points = np.dtype( [         # Структура для
    ('x', np.float32), ('y', np.float32)     # массива стартовых точек
] )
```

Структурированные массивы в сочетании с no-python mode, являющийся строгим режимом компиляции, способствуют повышению производительности за счет ускорения доступа к данным в памяти.

Размерность матрицы для дискретизации пространства рассчитывается автоматически на основе заданного программой размера клетки и полученных решений динамической системы. Алгоритм включает в себя следующие шаги:

1. Получение минимальных и максимальных значений по осям X и Y. Эти значения определяют границы области в пространстве состояний системы;
2. Расчёт смещений (разница между максимальным и минимальным значением) по каждой оси;
3. Определение размерности матрицы путём деления максимального смещения на заданный размер клетки и последующего округления в

большую сторону для полного покрытия области решений динамической системы;

Такой подход позволяет адаптировать размер сетки к различным масштабам динамических систем и начальным условиям, обеспечивая эффективное представление данных в дискретной модели.

В главной функции программы *create_grid* происходит создание и наполнение сетки точками, которые генерируются на основе динамической системы. Она начинает с определения максимального количества точек, которые могут быть размещены в сетке. Для хранения координат этих точек инициализируется массив.

Для каждой из начальных точек системы выполняются следующие действия:

1. Интегрирование системы уравнений для получения траектории движения из этой точки;
2. Сохранение координат точек траектории в массив;
3. Сравнение новых точек с уже имеющимися в сетке для выявления неоднозначности правил перехода в дискретной модели.

Сравнение новых точек производит функция *comparison_point*. Она проходит через массив точек, определяя направление движения между двумя последовательно расположенными точками. На основе полученного вектора движения функция рассчитывает соответствующее новое значение для обновления сетки. Затем она вызывает *update_grid_and_repeat_points*.

Задачей функции *update_grid_and_repeat_points* является обновление сеточной структуры и массива неоднозначных точек системы исходя из переданных в нее параметров. Она осуществляет проверку каждой ячейки на содержание уже определенного значения. В случае отсутствия такого значения функция обновляет клетку, записывая в нее новое значение. В случае обнаружения значения, уже присутствующего в клетке, функция запускает процедуру, направленную на определение уникальности обновляемой точки. Для этого осуществляется сравнение измененной точки с уже

зарегистрированными в матрице значениями. Когда точка уникальна, она добавляется в массив неоднозначных правил перехода исследуемой матрицы. Этот процесс играет важную роль в исследовании состояний системы и может быть использован для дальнейшего углубленного изучения динамики и поведения модели.

Обе функции (*comparison_point* и *update_grid_and_repeat_points*) взаимодействуют друг с другом, чтобы отслеживать изменения в сетке и регистрировать неоднозначные правила перехода, что может быть полезно для анализа траекторий в динамической системе.

Функционал, описанный в предыдущих абзацах, был усовершенствован с помощью библиотеки Numba, которая способствует ускорению процесса вычислений благодаря механизму компиляции Just-In-Time (JIT). Декоратор *@numba.njit(cache=True)* указывает на то, что функции должны быть скомпилированы, а результаты их работы кэшированы для повторного использования.

По завершении работы алгоритма каждая ячейка в рамках системы, характеризующейся уже дискретным набором данных, имеет подобный вид: [3, 1, 4, 2]. Этот массив несет в себе информацию о характере траекторий движения в динамической системе. Каждый индекс в массиве соответствует определенному направлению входа в ячейку: 0 указывает на приход из верхней клетки, 1 – из правой, 2 – из нижней, 3 – из левой. В свою очередь, числовое значение, расположенное по данному индексу, определяет направление последующего движения из этой клетки, т. е. направление выхода: 1 – указывает на движение вверх, 2 – вправо, 3 – вниз, 4 – влево.

Таким образом, матрица описывает правила перехода между состояниями в системе с конечным множеством состояний. Для того чтобы обеспечить возможность сохранения правил, описывающих ансамбль траекторий, реализовано запоминание в каждой клетке матрицы предыдущего состояния системы. Это позволяет не только проследить путь, который преодолевает та или иная траектория, но и предугадать возможные варианты её продолжения.

Матрица становится инструментом для анализа поведения системы, давая возможность выявить структурные особенности и закономерности.

Неоднозначность правил перехода

Одной из особенностей, определяющих поведение динамических систем, является характер правил перехода. В идеальных моделях каждое состояние однозначно ведет к следующему, однако в реальных системах часто появляется неоднозначность.

Неоднозначность правил перехода в динамической модели возникает, когда из одного и того же состояния системы существует несколько выходов, т. е. несколько возможных путей или результатов перехода в следующее состояние. Это означает, что поведение системы может варьироваться в зависимости от определенных условий или прошлых взаимодействий.

В рамках матрицы, описывающей систему с конечным множеством состояний, неоднозначные правила перехода представлены в виде ячеек, содержащих множество значений, каждое из которых указывает на различные возможные траектории движения из данного состояния. В программной реализации неоднозначные правила перехода записываются в массив и сохраняются в файле, обеспечивая тем самым удобство для последующего анализа и исследования.

Решение о конкретизации правила перехода в клетке с неоднозначностью осуществляется на основе изучения траекторий, которые движутся в аналогичном направлении. К примеру, рассмотрим ситуацию, когда для конкретной траектории внутри ячейки возникает неопределенность. В то же время, другие траектории, которые движутся в сходном направлении, имеют чётко определенные правила перехода. В таком случае, исходя из характеристик этих сопоставимых траекторий, возможно уточнение правила перехода.

Добавление времени

Недостаток данного подхода заключается в том, что правила могут иметь довольно много ложного поведения, поэтому на их основе может быть трудно доказать некоторые свойства.

Такая ложная транзитивность связана с тем, что соотношение перехода между соседними клетками вычисляется локально: в зависимости от времени, проведенного в данной клетке [2, с. 81]. Разницу в поведении можно увидеть на рисунке 4.

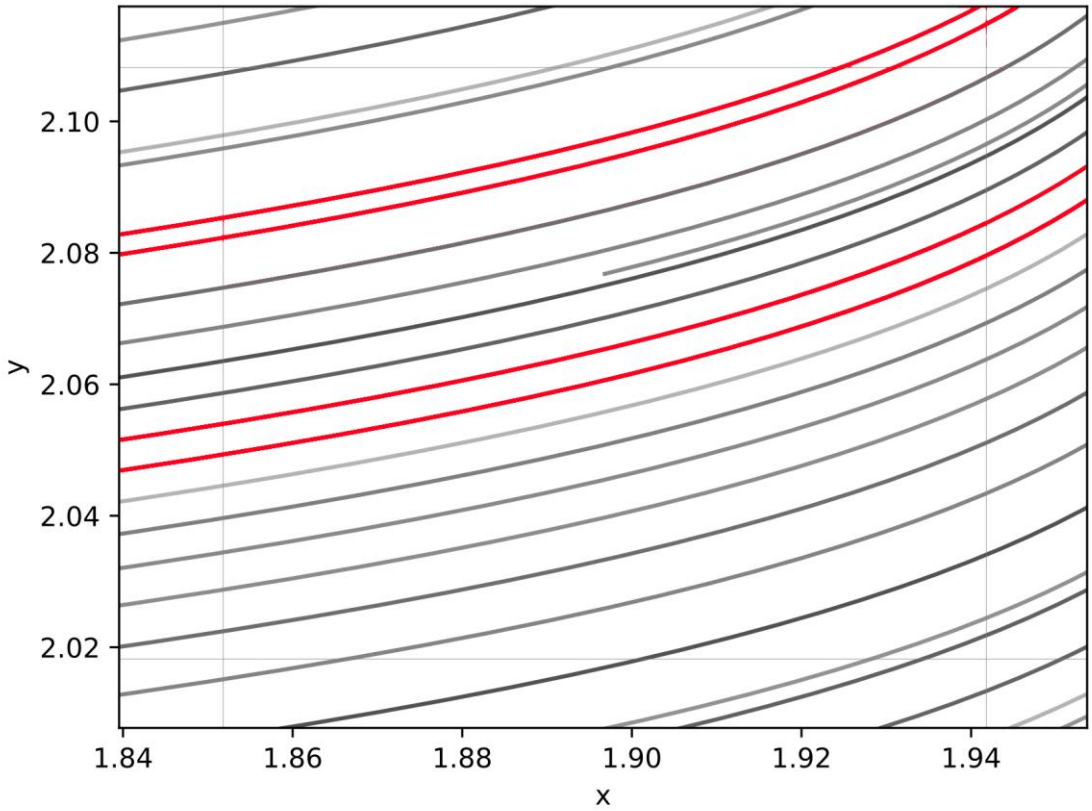

Рис. 2. Пример прохождения траекторий через клетки.

Для оценки времени, в течение которого траектория может оставаться в клетке, к правилам перехода из матрицы состояний добавляется итерационный параметр. Он будет представлять собой среднее число итераций, в течение которых траектория оставалась в клетке.

Тогда набор правил будет иметь следующий вид:

$$[3,1,2,4],t$$

где [3, 1, 2, 4] набор правил перехода по клеткам, а t – время, в течение которого траектория находиться в клетке.

Однако, учитывая, что система наблюдает за целым ансамблем траекторий, простой подсчет времени окажется недостаточным. Каждый проход траектории через ячейку увеличивает общий временной счетчик, поэтому вводится дополнительный итерационный параметр, который отслеживает количество траекторий, посетивших каждую клетку. Используя этот параметр, можно вычислить среднее время пребывания траектории в ячейке. Оно равно:

$$t_{\text{ср}} = \text{t/n}$$

где t – общее количество итераций в клетке, n – количество траекторий, посетивших эту ячейку.

После введения параметра каждая ячейка принимает вид:

$$[3,1,2,4], t_{\text{ср}}, n$$

Таким образом введение параметров времени дает обобщенное представление динамики системы, открывая возможно для более глубокого анализа и позволяя предсказать будущее поведение системы на основе ее прошлых состояний.

Заключение

Мы представили нашу динамическую систему в виде автомата с конечным числом состояний. Исследовав качественно полученный автомат, можно качественно оценить динамику системы. Использование компьютера дает приближенное решение дифференциальных уравнений на конечном отрезке времени, что позволяет качественно понять поведение фазовых траекторий в целом.

Отследив переход из одного состояния в другое с учетом предыдущего, мы добились однозначности эволюционных правил этого автомата в тех случаях, когда изменение невозможно определить только текущей клеткой. И это позволило создать правила динамики конечного автомата, который воспроизводит динамику исходной системы.

Список литературы

1. *Пухов А. А.* Лекции по колебаниям и волнам: учеб. пособие. В двух частях. Ч. 1. Колебания / А. А. Пухов. – Москва: МФТИ, 2019 – 208с.

2. *О. Малер, Г. Батт* Аппроксимация непрерывных систем временными автоматами. J. Fisher (Ed.): FMSB 2008, LNBI 5054, 2008 – 77-89с.

3. *У. Хартонг, Л. Хедрих, Э. Барк* О дискретном моделировании и проверке моделей для нелинейных аналоговых систем. Д. Бринксма и К. Г. Ларсен (ред.): CAV 2002, LNCS 2404, 2002 – 401-414с.

Приложение

Листинг 2. Файл main.py

```
import approx as ap
from prompt_toolkit.shortcuts import radiolist_dialog, input_dialog
from params import *
import time as t
import matplotlib.pyplot as plt
import os

def approx():
    start_time = t.time()
    save_folder = 'results'
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)
    n = get_n()
    h = get_h()
    mu = get_mu()
    cell_size = get_cell()
    start_point_arr = np.array( [[1, 1]] )
    solution = ap.method_euler(start_point_arr[0], n, h, mu)
    matrix_size, start_point_polygon = ap.calc_grid_size(cell_size,
solution)
    start_point_arr = ap.calc_start_point(start_point_polygon,
matrix_size, cell_size)
    grid = np.zeros((matrix_size, matrix_size),
dtype=ap.dtype_for_matrix)
    repeat_points = np.zeros((0), dtype=dtype_for_repeat_points)
    repeat_points_temp = np.zeros((1), dtype=dtype_for_repeat_points)
    grid, repeat_points = ap.create_grid(grid, repeat_points,
matrix_size, repeat_points_temp, cell_size, start_point_arr)
    end_time = t.time()
    print(f"Время выполнения программы: {end_time - start_time}
сек.\nРазмер клетки: {cell_size}\nРазмер матрицы: {matrix_size}")
    np.save(f'{matrix_size}', grid)
    grid_filename = os.path.join(save_folder, f'{matrix_size}.txt')
    np.savetxt(grid_filename, grid[::-1], fmt='%s')
    repeat_points_filename = os.path.join(save_folder,
f'repeat_of_{matrix_size}.txt')
    with open(repeat_points_filename, "w") as my_file:
        for string in reversed(repeat_points):
            my_file.write(f'{string}\n')
    plt_filename = os.path.join(save_folder, f'{matrix_size}.svg')
    ap.draw_grid(start_point_polygon)
    plt.savefig(plt_filename)
    plt.plot()

def change_param():
    while True:
        result = radiolist_dialog(
```

```
            title='Изменить параметры',
            text='Выберите параметр для изменения:',
            values=[
                ('ch_h', f'Изменить h (Текущее значение: h =
{get_h()})'),
                ('ch_mu', f'Изменить mu (Текущее значение: mu =
{get_mu()})'),
                ('ch_cell', f'Изменить cell_size (Текущее значение:
cell_size = {get_cell()})'),
                ('back', 'Назад'),
            ],
        ).run()
        if result == 'ch_h':
            new_h = input_dialog(title="Изменить h", text="Введите
новое значение для h:").run()
            if new_h is not None:
                set_h(float(new_h))
        elif result == 'ch_mu':
            new_mu = input_dialog(title="Изменить mu", text="Введите
новое значение для mu:").run()
            if new_mu is not None:
                set_mu(float(new_mu))
        elif result == 'ch_cell':
            new_cell_size = input_dialog(title="Изменить cell_size",
text="Введите новое значение для cell_size:").run()
            if new_cell_size is not None:
                set_cell(float(new_cell_size))
        elif result == 'back':
            break
        elif result is None:
            break

def run_menu():
    result = radiolist_dialog(
        title='Аппроксимация непрерывной модели мультивибратора
моделью с конечным множеством состояний',
        text='Выберите опцию:',
        values=[
            ('approx', 'Аппроксимировать'),
            ('ch_params', 'Изменить параметры'),
            ('exit', 'Выход'),
        ],
    ).run()
    return result

def main():
    while True:
        selected = run_menu()
        if selected == 'approx':
            approx()
```

```
        elif selected == 'ch_params':
            change_param()
        elif selected == 'exit':
            break
        elif selected is None:
            break
if __name__ == '__main__':
    main()
```

Листинг 3. Файл approx.py

```
import numpy as np
import numba as nb
import matplotlib.pyplot as plt
from params import *

""" Глобальные переменны """
n = get_n()
h = get_h()
mu = get_mu()
cell_size = get_cell()

def draw_grid(start_point_polygon):
    running = True
    i = 0
    while running:
        line_x = (start_point_polygon[0, 0] - cell_size) + i *
cell_size
        line_y = (start_point_polygon[1, 1] + cell_size) - i *
cell_size
        i += 1
        if line_x < start_point_polygon[0, 1]:
            plt.plot([line_x, line_x], [start_point_polygon[1, 0] -
1, start_point_polygon[1, 1] + 1], '-', linewidth=0.2, color='grey')
        if line_y > start_point_polygon[1, 0]:
            plt.plot([start_point_polygon[0, 0] - 1,
start_point_polygon[0, 1] + 1], [line_y, line_y], '-', linewidth=0.2,
color="grey")
        if line_x > start_point_polygon[0, 1] + 1 and line_y <
start_point_polygon[1, 0] - 1:
            running = True
            break

def draw_grafic(point):
    X = point[0]
    Y = point[1]
    plt.xlabel('x')
    plt.ylabel('y')
    plt.plot(X, Y)
```

```
@nb.njit(cache=True)
def f_1(x, y):
  return -x*(x**2 - 5) - y

@nb.njit(cache=True)
def f_2(x, y):
  return x

@nb.njit(cache=True)
def method_euler(start_point, n, h, mu):
    """ Решение методом Эйлера """
    point = np.zeros( (2, n+1) )
    x, y = start_point[0], start_point[1]
    point[0, 0], point[1, 0] = x, y
    for i in nb.prange(n):
        dx = f_1(x, y) * h / mu
        dy = f_2(x, y) * h
        x += dx
        y += dy
        point[0, i+1], point[1, i+1] = x, y
    return point

@nb.njit(cache=True)
def calc_grid_size(cell_size, solution):
    x_min, x_max = min(solution[0]), max(solution[0])
    y_min, y_max = min(solution[1]), max(solution[1])
    x_offset, y_offset = x_max - x_min, y_max - y_min
    grid_size = int(max(x_offset, y_offset) // cell_size + 1)
    if max(x_offset, y_offset) == x_offset:
        start_point_polygon = np.array([ [x_min, x_max], [x_min,
x_max] ])
    elif max(x_offset, y_offset) == y_offset:
        start_point_polygon = np.array([ [y_min, y_max], [y_min,
y_max] ])
    return grid_size, start_point_polygon

@nb.njit(cache=True)
def calc_middle_iteration(grid):
    for row in range(grid.shape[0]):
        for col in range(grid.shape[1]):
            cell = grid[row, col]
            if cell['iteration'] != 0 and cell['pathway'] != 0:
                cell['iteration'] //= cell['pathway']

@nb.njit(cache=True)
def calc_start_point(polygon, grid_size, cell_size):
    quantity = grid_size * grid_size
    x_min, x_max = polygon[0, 0], polygon[0, 1]
    y_min, y_max = polygon[1, 0], polygon[1, 1]
    start_point = np.empty( (quantity, 2) )
```

```
    x_offset, y_offset = 0, 0 # индекс смещения по x и по y
    for i in range(quantity):
        x = x_min + (cell_size / 2) + x_offset * cell_size
        y = y_min + (cell_size / 2) + y_offset * cell_size
        x_offset += 1
        if x > x_max:
            x_offset = 0
            y_offset += 1
        start_point[i, 0] = x
        start_point[i, 1] = y
    return start_point

@nb.njit(cache=True)
def error_len(x, y, grid_size):
    return 0 <= x < grid_size and 0 <= y < grid_size
@nb.njit(cache=True)
def save_coord_point(coord_point_arr, grid, grid_size, cell_size,
point, n):
    prev_arr = np.array([[
        int( (grid_size - 1) / 2 ) + int( point[0, 0] / cell_size ),
        int( (grid_size - 1) / 2 ) + int( point[1, 0] / cell_size )
    ]])
    for i in nb.prange(1, n):
        curr_x = int( (grid_size + 1) / 2 ) + int( point[0, i] /
cell_size )
        curr_y = int( (grid_size + 1) / 2 ) + int( point[1, i] /
cell_size )
        if(error_len(prev_arr[0][0], prev_arr[0][1], grid_size) and
error_len(curr_x, curr_y, grid_size)):
            if prev_arr[0][0] != curr_x or prev_arr[0][1] != curr_y:
                x, y = prev_arr[0, 0], prev_arr[0, 1]
                coord_point_arr = np.vstack((coord_point_arr,
prev_arr.reshape(1, 2)))
                grid[y, x]['iteration'] += 1
            else:
                grid[y, x]['iteration'] += 1
            prev_arr[0][0], prev_arr[0][1] = curr_x, curr_y
    return coord_point_arr[1:]

@nb.njit(cache=True)
def is_in(e, arr):
    len_arr = len(arr)
    if(len_arr > 0):
        for i in nb.prange(len_arr):
            if e['y'] == arr[i]['y'] and e['x'] == arr[i]['x'] and \
                np.all(e['curr_arr'] == arr[i]['curr_arr']) and
np.all(e['edited_arr'] == arr[i]['edited_arr']):
                return True
    return False
```

```
@nb.njit(cache=True)
def numba_vstack(arr1, arr2):
    combined_length = len(arr1) + len(arr2)
    result = np.empty(combined_length, dtype=dtype_for_repeat_points)
    result[:len(arr1)] = arr1
    result[len(arr1):] = arr2
    return result

@nb.njit(cache=True)
def update_grid_and_repeat_points(grid, point_coords, k, new_value,
repeat_points, repeat_points_temp, pathway_counter):
    y, x = int(point_coords[1]), int(point_coords[0])
    cell = grid[y, x]
    if cell['array'][k] in (0, new_value):
        cell['array'][k] = new_value
        if cell['pathway'] <= pathway_counter:
            cell['pathway'] += 1
    else:
        point_temp = cell['array'].copy()
        point_temp[k] = new_value
        is_unique = True
        for rp in repeat_points:
            if np.array_equal(point_temp, rp['edited_arr']):
                is_unique = False
                break
        if is_unique:
            repeat_points_temp[0]['y'], repeat_points_temp[0]['x'] =
y, x
            repeat_points_temp[0]['curr_arr'] = cell['array'].copy()
            repeat_points_temp[0]['edited_arr'] = point_temp
            if not is_in(repeat_points_temp[0], repeat_points):
                repeat_points = numba_vstack(repeat_points,
repeat_points_temp)
    return repeat_points

@nb.njit(cache=True)
def comparison_point(arr, grid, repeat_points, repeat_points_temp,
pathway_counter):
    for i in nb.prange(1, len(arr) - 1):
        prev_x, prev_y = arr[i - 1]
        curr_x, curr_y = arr[i]
        next_x, next_y = arr[i + 1]
        k = 0
        if prev_x < curr_x:
            k = 3
        elif prev_x > curr_x:
            k = 1
        if prev_y < curr_y:
            k = 2
        elif prev_y > curr_y:
```

```
            k = 0
        new_value = 0
        if next_y < curr_y:
            new_value = 3
        elif next_y > curr_y:
            new_value = 1
        elif next_x < curr_x:
            new_value = 4
        elif next_x > curr_x:
            new_value = 2
        repeat_points = update_grid_and_repeat_points(grid, arr[i],
k, new_value, repeat_points, repeat_points_temp, pathway_counter)
    return repeat_points


def create_grid(grid, repeat_points, grid_size, repeat_points_temp,
cell_size, start_point_arr):
    coord_point_arr = np.empty((0, 2), dtype=np.float64)
    for i in range(len(start_point_arr)):
        start_point = start_point_arr[i]
        pathway_counter = i
        point = method_euler(start_point, n, h, mu)
        draw_grafic(point)
        new_points = save_coord_point(coord_point_arr, grid,
grid_size, cell_size, point, n)
        repeat_points = comparison_point(new_points, grid,
repeat_points, repeat_points_temp, pathway_counter)
    calc_middle_iteration(grid)
    return grid, repeat_points
```

Листинг 4. Файл params.py

```
import numpy as np

def set_h(temp):
    global _h
    _h = temp

def set_mu(temp):
    global _mu
    _mu = temp

def set_cell(temp):
    global _cell_size
    _cell_size = temp

def get_h():
    global _h
    return _h
```

```
def get_mu():
    global _mu
    return _mu

def get_cell():
    global _cell_size
    return _cell_size

def get_n():
    global _n
    return _n

_segment = np.array([0, 10])
_h = 0.001 # Шаг
_mu = 0.1 # Параметр
_n = int( (_segment[1] - _segment[0]) / _h )
_cell_size = 1.02 # Размер клетки

dtype_for_matrix = np.dtype( [      # Тип для
    ('array', np.int16, (4)),       # матрицы
    ('iteration', np.int32),
    ('pathway', np.int16)
] )
dtype_for_repeat_points = np.dtype( [       # Тип для
    ('y', np.int16), ('x', np.int16),       # массива с
    ('curr_arr', np.int16, (4)),            # повторяющимися точками
    ('edited_arr', np.int16, (4))
] )
dtype_for_start_points = np.dtype( [
    ('x', np.float32), ('y', np.float32)
] )
```