МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

МОДЕЛИРОВАНИЕ

ОТЧЕТ О ПРАКТИКЕ

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Мельникова Артемия Дмитриевича

Проверил
Доцент ______________ Е. П. Станкевич

Саратов 2024

СОДЕРЖАНИЕ

1 Задание №1

1.1 Условие задачи

Температура свежевыпеченного хлеба равна 150°. До отправки в магазин хлеб остывает в помещении с постоянной температурой 20°. Требуется определить длительность времени охлаждения хлеба до 40°. Результат можно получить с использованием закона теплового излучения

$$\frac{dx}{dy} = -k(x-a),$$

где $x(t)$ – температура хлеба в момент времени t, a– температура воздуха в помещении, $k > 0$– коэффициент пропорциональности. Полагаем, что $k = 0.02$.

1.2 Ход решения

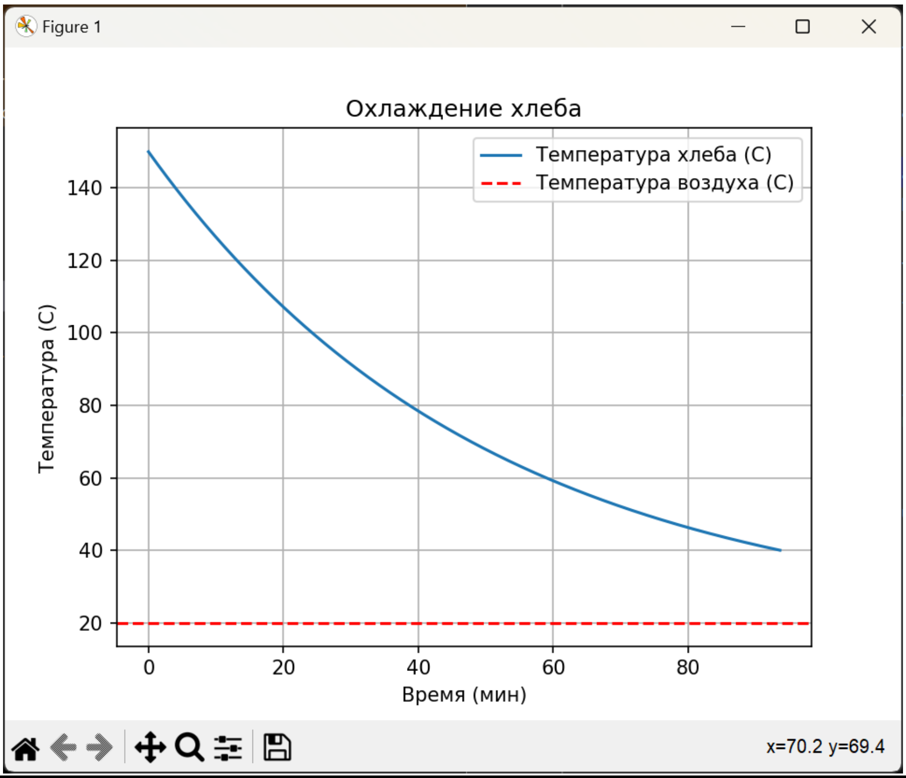
1. Инициализация параметров:
   — Задание коэффициента пропорциональности k.
   — Определение температуры воздуха в помещении a.
   — Определение начальных условий для решения дифференциального уравнения: начального времени t_0, начальной температуры хлеба x_0, шага времени dt и температуры, при которой останавливаемся $end\_temperature$.
2. Определение функций:
   — $differential\_equation(x)$: Определение функции, представляющей дифференциальное уравнение для охлаждения хлеба.
   — $euler\_method(t, x, d\_t, end\_temp)$: Определение метода Эйлера для численного решения дифференциального уравнения. Этот метод будет решать дифференциальное уравнение до тех пор, пока температура хлеба не достигнет end_temp.
3. Решение дифференциального уравнения:
   — Использование метода Эйлера для вычисления времени и температуры хлеба во времени.
   — Нахождение времени, когда температура хлеба достигнет 40 градусов ($end\_temperature$).
4. Вывод результатов

1.3 Код программы

```python
import matplotlib.pyplot as plt

# Заданные параметры
k = 0.02  # коэффициент пропорциональности
a = 20  # температура воздуха в помещении


# Функция, представляющая дифференциальное уравнение
def differential_equation(x):
    return -k * (x - a)


# Метод Эйлера для численного решения дифференциального уравнения
def euler_method(t, x, d_t, end_temp):
    t_val = [t]
    x_val = [x]

    while x_val[-1] > end_temp:
        t = t_val[-1]
        x = x_val[-1]
        x_next = x + d_t * differential_equation(x)
        t_val.append(t + d_t)
        x_val.append(x_next)

    return t_val, x_val


# Начальные условия
t0 = 0  # начальное время
x0 = 150  # начальная температура хлеба
dt = 0.001  # шаг времени
end_temperature = 40  # температура, когда останавливаемся

# Решение дифференциального уравнения методом Эйлера
t_values, x_values = euler_method(t0, x0, dt, end_temperature)

# Найдем индекс, когда температура хлеба достигнет 40 градусов
index_40_degrees = next(i for i, x in enumerate(x_values) if x <= 40)

# Выведем время в секундах, когда температура достигнет 40 градусов
time_at_40_degrees = t_values[index_40_degrees]
print("Время до достижения температуры 40 градусов:", round(time_at_40_degrees, 2), "минут")

# Визуализация результатов
plt.plot(t_values, x_values, label='Температура хлеба (C)')
plt.axhline(y=a, color='r', linestyle='--', label='Температура воздуха (C)')
plt.xlabel('Время (мин)')
```

```
plt.ylabel('Температура (C)')
plt.title('Охлаждение хлеба')
plt.legend()
plt.grid(True)
plt.show()
```

1.4 Результат работы программы




```
Время до достижения температуры 40 градусов: 93.59 минут
```

2 Задание №2

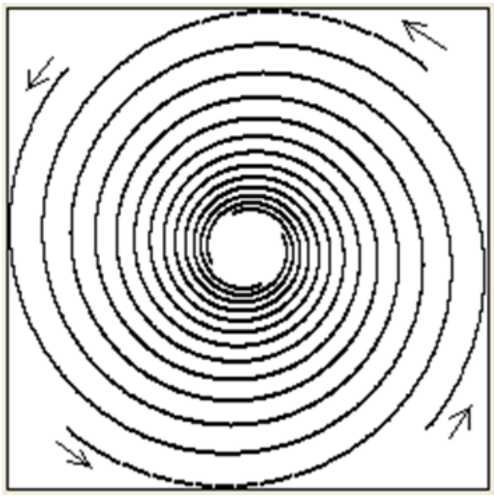
2.1 Условие задачи

Построить фазовый портрет системы дифференциальных уравнений

$$\begin{cases} \frac{dx}{dt} = ax - y, \\ \frac{dy}{dt} = x + ay \end{cases}$$

при -0.2 < a < 0. Эксперимент повторить при a > 0.

Пояснение к задаче. Схематически фазовый портрет этой системы дифференциальных уравнений имеет вид:



2.2 Ход решения

1. Определение системы уравнений: Функция $model(x, y, a)$ определяет систему уравнений в виде дифференциальных уравнений первого порядка. В данном случае у нас есть два уравнения: $\frac{dx}{dt} = ax - y$ и $\frac{dy}{dt} = x + ay$.
2. Метод Эйлера для численного решения системы:
   — Функция $euler\_method(x0, y0, a, dt, num\_steps)$ используется для численного решения системы уравнений методом Эйлера.
   — Мы начинаем с начальных условий x_0 и y_0, затем на каждом шаге вычисляем приращения $dxdt$ и $dydt$ с помощью функции $model$.
   — После этого мы обновляем значения x и y согласно методу Эйлера и сохраняем их в списки x_values и y_values.
3. Фазовый портрет системы:

— Функция $phase\_portrait(plt, x0, y0, a)$ рисует фазовый портрет системы.
— метод Эйлера используется для генерации траекторий в фазовом пространстве для различных начальных условий и значений параметра a.
— Для каждого значения параметра а вызываем функцию $phase\_portrait$ для построения соответствующего графика фазового портрета.

2.3 Код программы

```
import matplotlib.pyplot as plt


# Определение системы уравнений
def model(x, y, a):
    dxdt = a * x - y
    dydt = x + a * y
    return dxdt, dydt


# Метод Эйлера для численного решения системы
def euler_method(x0, y0, a, dt, num_steps):
    x_values = [x0]
    y_values = [y0]

    for _ in range(num_steps):
        dxdt, dydt = model(x_values[-1], y_values[-1], a)
        x_new = x_values[-1] + dt * dxdt
        y_new = y_values[-1] + dt * dydt
        x_values.append(x_new)
        y_values.append(y_new)

    return x_values, y_values


def phase_portrait(plt, x0, y0, a):
    x_values, y_values = euler_method(x0, y0, a, dt, num_steps)
    plt.plot(x_values, y_values, label=f'a={a}')
    x_values, y_values = euler_method(-x0, y0, a, dt, num_steps)
    plt.plot(x_values, y_values, label=f'a={a}')
    x_values, y_values = euler_method(-x0, -y0, a, dt, num_steps)
    plt.plot(x_values, y_values, label=f'a={a}')
    x_values, y_values = euler_method(x0, -y0, a, dt, num_steps)
    plt.plot(x_values, y_values, label=f'a={a}')
    # Настройка графика
    plt.set_xlabel('x')
```

```
    plt.set_ylabel('y')
    plt.set_title('Фазовый портрет системы')
    plt.legend()
    plt.grid(True)


# Начальные условия
x0, y0 = 1, 1

# Временной интервал
dt = 0.01
num_steps = 1000

fig1, ax1 = plt.subplots()
fig2, ax2 = plt.subplots()
fig3, ax3 = plt.subplots()

phase_portrait(ax1, x0, y0, -0.2)
phase_portrait(ax2, x0, y0, -0.1)
phase_portrait(ax3, x0, y0, 0.1)

plt.show()
```

2.4 Результат работы программы

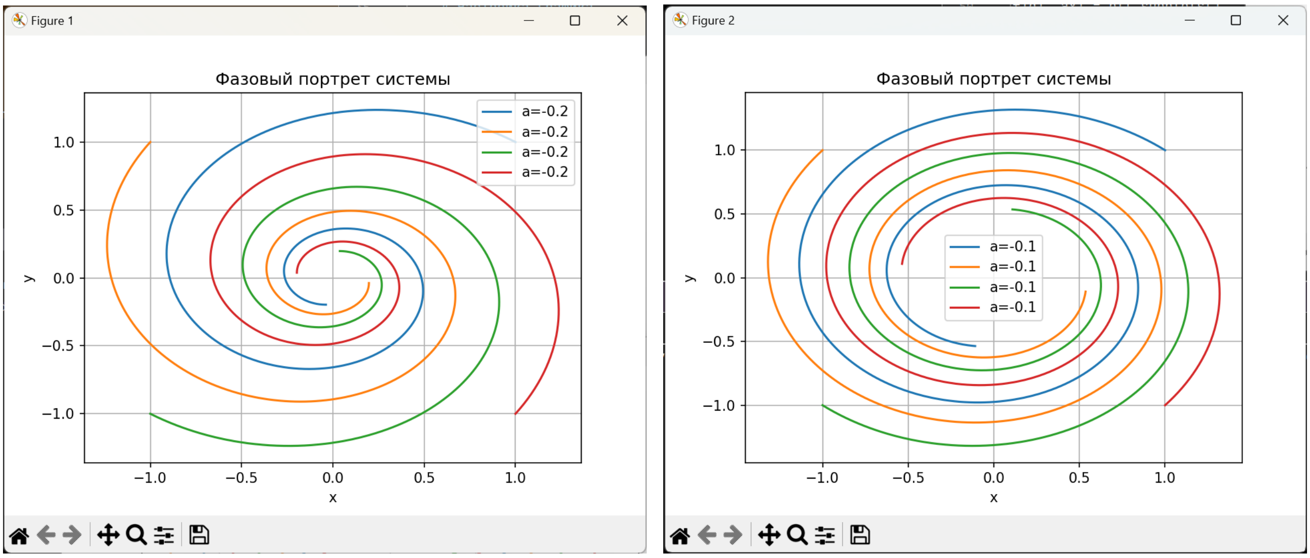


Рисунок 1 – -0.2 < a < 0

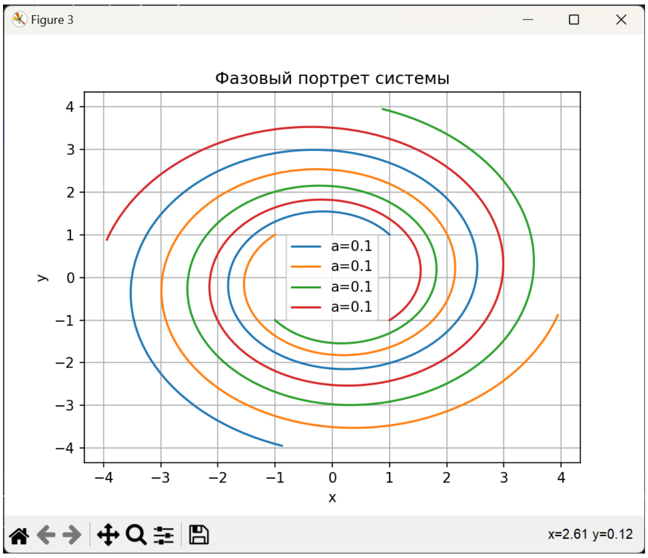


Рисунок 2 – a > 0

3 Задание №3

3.1 Условие задачи

Система состоит из трех приборов. Известна вероятность безотказной работы каждого из них в течение времени T. Приборы выходят из строя независимо друг от друга. При отказе хотя бы одного прибора вся система перестает работать. Провести 1000 испытаний с моделью системы и оценить вероятность отказа системы за время T.

3.2 Ход решения

1. Определение функции $system\_failure\_prob$:
   — Функция $system\_failure\_prob(p1, p2, p3, num\_trials)$ принимает вероятности безотказной работы каждой из трех систем $(p1, p2, p3)$ и количество испытаний (num_trials).
   — Внутри функции инициализируется переменная $failures$, которая будет отслеживать количество отказов системы.
   — Затем выполняется num_trials испытаний, генерируя случайные значения для каждой из систем и проверяя, произошел ли хотя бы один отказ.
   — Если хотя бы одно из случайных значений превышает вероятность безотказной работы $(p1, p2, p3)$, то мы увеличиваем счетчик $failures$.
   — После всех испытаний вычисляется вероятность отказа системы путем деления количества отказов на общее количество испытаний.
2. Ввод вероятностей безотказной работы для каждой из систем.
3. Вызов функции и оценка вероятности отказа системы.

3.3 Код программы

```
import random


def system_failure_prob(p1, p2, p3, num_trials):
    failures = 0

    for _ in range(num_trials):
        # Генерируем случайные значения для каждого прибора
        device1 = random.random() > p1
        device2 = random.random() > p2
        device3 = random.random() > p3
```

```
        # Проверяем, произошел ли отказ системы
        if device1 or device2 or device3:
            failures += 1

    # Вычисляем вероятность отказа системы
    failure_prob = failures / num_trials
    return failure_prob


# Указываем вероятности безотказной работы и время
print('Укажите вероятность безотказной работы для каждой из систем:')
p1 = float(input('вероятность безотказной работы системы 1 равна: '))
p2 = float(input('вероятность безотказной работы системы 2 равна: '))
p3 = float(input('вероятность безотказной работы системы 3 равна: '))

# Проводим 1000 испытаний и оцениваем вероятность отказа системы
num_trials = 1000
failure_probability = system_failure_prob(p1, p2, p3, num_trials)
print("Вероятность отказа системы:", failure_probability)
```

3.4 Результат работы программы

```
Укажите вероятность безотказной работы для каждой из систем:
вероятность безотказной работы системы 1 равна: 0.95
вероятность безотказной работы системы 2 равна: 0.9
вероятность безотказной работы системы 3 равна: 0.85
Вероятность отказа системы: 0.271
```

4 Задание №4

4.1 Условие задачи

Оценить вероятность того, что из трех взятых наудачу отрезков длиной не более L можно построить треугольник.

4.2 Ход решения

4.3 Код программы

1. Определение функции $can\_form\_triangle$:
   — Функция $can\_form\_triangle(a, b, c)$ принимает длины трех отрезков и возвращает булевое значение, указывающее, можно ли построить треугольник с такими длинами сторон.
   — Для этого функция проверяет выполнение условия треугольника: сумма длин двух сторон должна быть больше длины третьей стороны.
2. Определение функции $estimate\_probability$:
   — Функция $estimate\_probability(num\_trials, len)$ оценивает вероятность того, что случайно выбранные три отрезка с заданной максимальной длиной могут образовать треугольник.
   — Мы создаем счетчик $count\_possible\_triangles$, который будет увеличиваться каждый раз, когда три случайно выбранные стороны могут образовать треугольник.
   — Затем мы выполняем num_trials испытаний, генерируя случайные длины для каждой из трех сторон и проверяя, можно ли построить треугольник с этими длинами.
   — После всех испытаний мы вычисляем вероятность того, что три случайно выбранные стороны могут образовать треугольник, разделив количество успешных испытаний на общее количество испытаний.
3. Ввод максимальной длины стороны и запуск оценки вероятности.

```python
import random


def can_form_triangle(a, b, c):
    return a + b > c and a + c > b and b + c > a


def estimate_probability(num_trials, len):
    count_possible_triangles = 0
```

```
    for _ in range(num_trials):
        # Генерируем три случайных отрезка
        a = random.uniform(0, len)
        b = random.uniform(0, len)
        c = random.uniform(0, len)

        # Оцениваем, можно ли построить треугольник
        if can_form_triangle(a, b, c):
            count_possible_triangles += 1

    probability = count_possible_triangles / num_trials
    return probability


# Указываем количество экспериментов (больше экспериментов - более точная оценка)
l = int(input('Введите максимальную длину взятой наудачу стороны:'))
num_trials = 100000
result_probability = estimate_probability(num_trials, l)

print(f"Вероятность построить треугольник: {result_probability}")
```

4.4 Результат работы программы

```
Введите максимальную длину взятой наудачу стороны: 10
Вероятность построить треугольник: 0.49841
```

5 Задание №5

5.1 Условие задачи

Диаметр шарика для шарикоподшипника является нормально распределенной случайной величиной с параметрами $\mu = 10$ и $\sigma = 0.03$. В отделе качества диаметры шариков измеряются. Результатом измерения является нормально распределенная случайная величина со средним, равным фактическому диаметру шарика, а среднее квадратическое отклонение равно 0,02 – ошибка измерения. Если результат измерения находится в интервале [9.97; 10.03], то шарик передают в сборочный цех. В противном случае, шарик отбраковывается. Оценить вероятность того, что в сборочном цехе окажутся шарики с диаметром, не принадлежащем [9.97; 10.03]. Оценку вероятности получить на основании 1000 испытаний.

5.2 Ход решения

1. Задаются параметры нормального распределения: среднее значение диаметра $mean\_diameter$ и стандартное отклонение std_dev. Задается значение среднего квадратического отклонения измерения $measurement\_error$, которое представляет собой погрешность измерения. Задаются нижняя и верхняя границы интервала $lower\_bound$ и $upper\_bound$, в котором должен находиться измеренный диаметр шарика, чтобы считаться корректным. Указывается количество испытаний num_trials, которое будет проведено для оценки вероятности.
2. Моделирование испытаний: Для каждого испытания в цикле for:
   — Генерируется случайное значение диаметра шарика с учетом погрешности измерения, используя нормальное распределение $np.random.normal(mean\_diameter, measurement\_error)$.
   — Проверяется, попадает ли измеренный диаметр шарика в заданный интервал. Если не попадает, счетчик успешных испытаний $successful\_trials$ увеличивается на 1.
3. Оценка вероятности: После завершения всех испытаний вычисляется вероятность того, что измеренный диаметр шарика находится в заданном интервале, путем деления количества успешных испытаний на общее количество испытаний.

5.3 Код программы

```python
import numpy as np

# Параметры нормального распределения
mean_diameter = 10
std_dev = 0.03

# Среднее квадратическое отклонение измерения
measurement_error = 0.02

# Границы интервала
lower_bound = 9.97
upper_bound = 10.03

# Количество испытаний
num_trials = 1000

# Счетчик удачных испытаний
successful_trials = 0

# Моделирование испытаний
for _ in range(num_trials):
    # Генерация случайного диаметра шарика с учетом погрешности измерения
    measured_diameter = np.random.normal(mean_diameter, measurement_error)
    # Проверка попадания в интервал
    if measured_diameter < lower_bound or measured_diameter > upper_bound:
        successful_trials += 1

# Оценка вероятности
probability = successful_trials / num_trials
print("Оценка вероятности:", probability)
```

5.4 Результат работы программы

```
Оценка вероятности: 0.123
```

6 Задание №6

6.1 Условие задачи

Система состоит из двух новых элементов. В начальный момент времени начинают работать оба элемента. Длительность безотказной работы каждого из элементов есть экспоненциально распределенная случайная величина. При отказе обоих элементов система перестает работать. Построить модель возникновения отказов в указанной системе. Провести 1000 испытаний с моделью и оценить математическое ожидание и среднее квадратическое отклонение длительности безотказной работы системы.

6.2 Ход решения

1. Определение функции $system\_failure$:
   — Функция $system\_failure(num\_trials)$ принимает количество испытаний num_trials.
   — В цикле происходит моделирование длительности безотказной работы для каждого элемента системы.
   — Для каждого элемента генерируется случайное время безотказной работы с экспоненциальным распределением (это часто используемое распределение для моделирования времени до отказа).
   — Время до первого отказа одного из элементов определяется как минимальное время из сгенерированных для всех элементов.
   — Длительность до отказа записывается в список $durations$.
2. Вызывается функция $system\_failure$ с указанным количеством испытаний num_trials. Результат сохраняется в переменную $durations$.
3. Оценка статистик:
   — Вычисляется оценка математического ожидания $mean\_duration$ и оценка среднего квадратического отклонения $std\_deviation$ для времени безотказной работы системы, используя функции $np.mean$ и $np.std$.
   — Среднее время безотказной работы дает представление о средней продолжительности работы системы до отказа.
   — Стандартное отклонение отображает меру разброса значений времени до отказа от их среднего значения.
4. Вывод результатов.

6.3 Код программы

```python
import numpy as np


def system_failure(num_trials):
    durations = []
    for _ in range(num_trials):
        # Моделирование длительности безотказной работы для каждого элемента
        duration_elem1 = np.random.exponential(scale=1)  # Параметр scale=1 для простоты
        duration_elem2 = np.random.exponential(scale=1)

        # Определение времени до первого отказа одного из элементов
        failure_time = min(duration_elem1, duration_elem2)

        durations.append(failure_time)

    return durations


# Количество испытаний
num_trials = 1000

# Моделирование системы
durations = system_failure(num_trials)

# Оценка математического ожидания и среднего квадратического отклонения
mean_duration = np.mean(durations)
std_deviation = np.std(durations)

print("Оценка математического ожидания длительности безотказной работы системы:",
↪   mean_duration)
print("Оценка среднего квадратического отклонения длительности безотказной работы системы:",
↪   std_deviation)
```

6.4 Результат работы программы

```
Оценка математического ожидания длительности безотказной работы системы: 0.4902077863365979
Оценка среднего квадратического отклонения длительности безотказной работы системы: 0.4802884824945925
```

7 Задание №7

7.1 Условие задачи

Дана СМО типа M | M | 1 с двумя классами требований и абсолютным приоритетом. Требования 2-го класса, обслуживание которых было прервано требованиями 1-го класса, встают в хвост очереди с остаточным временем обслуживания. Построить имитационную модель системы. На основании 1000 выборочных значений оценить $\overline{u}$ и $\overline{n}$ для каждого класса требований.

7.2 Ход решения

1. Определение классов $Request$ и $Server$:
   — Класс $Request$ представляет запрос с атрибутами: время поступления $arrival\_time$, время обслуживания $service\_time$, приоритет $priority$, время начала обслуживания $start\_time$ и время завершения обслуживания end_time.
   — Класс $Server$ представляет сервер с атрибутом текущего запроса $current\_request$ и методом $serve$, который устанавливает время начала и завершения запроса и обновляет текущий запрос.
2. Функция $generate\_requests$ создает список запросов, генерируя случайное количество запросов $num\_requests$, каждый с случайным временем обслуживания, определенным с помощью экспоненциального распределения. При этом устанавливается случайный приоритет.
3. Вычисление статистик:
   — Функция $calculate\_statistics$ вычисляет статистики обслуживания и ожидания для каждого класса запросов.
   — Для каждого класса вычисляются общее время обслуживания $total\_service\_time$ и общее время ожидания $total\_waiting\_time$. Затем вычисляются средние времена обслуживания и ожидания.
4. Моделирование системы:
   — Функция $simulate\_system$ моделирует работу системы.
   — Создается объект $Server$.
   — Генерируются запросы с помощью $generate\_requests$.
   — Для каждого запроса вызывается метод $serve$ сервера для его обслуживания.
5. Вызов функций и вывод результатов.

7.3 Код программы

```
import random
import numpy as np
import threading
import queue
import time

# Создаем блокировку для синхронизации доступа к стандартному выводу
print_lock = threading.Lock()
# Начальное время работы сервера
start_time = time.time()
# Определяем очередь требований
requests_queue = queue.Queue()
# Список обслуженных требований
serviced_requests_list = []
# Работа сервера
start_server = True


# класс request: запрос в системе массового обслуживания.
class Request:
    def __init__(self, request_number, arrival_time, service_time, priority):
        self.request_number = request_number  # Номер запроса для ясности
        self.arrival_time = arrival_time  # arrival_time: время прибытия запроса в систему
        ↪  (становления в очередь).
        self.service_time = service_time  # service_time: время, необходимое для
        ↪  обслуживания запроса.
        self.priority = priority  # priority: приоритет запроса.
        self.start_time = None  # start_time: время, когда требование начинает обслуживаться
        ↪  сервером.
        self.end_time = None  # время, когда обслуживание требования завершается.
        self.remaining_service_time_class2 = self.service_time  # если приоритет запроса
        ↪  равен 2,
        # тогда эта переменная используется для хранения оставшегося времени на обработку
        ↪  данного запроса

    # Функция для вывода запроса
    def print(self):
        return self.request_number, self.priority, self.arrival_time, self.service_time


requests_list: list[Request] = []


# Функция для добавления в очередь на обслуживание случайного запроса из списка
def put_in_queue():  # num, lambda_val, observation_time
    global start_time
    global requests_list
```

```
    # print(requests_list)
    current_time = 0
    start_time = time.time()
    for request in requests_list[:]:
        current_time = request.arrival_time - current_time
        time.sleep(current_time)
        current_time = request.arrival_time
        requests_queue.put(request)
        with print_lock:
            print(f'R: Запрос с параметрами {request.print()} добавлен в очередь. '
                  f'Запросов осталось: {len(requests_list) - 1}')
        requests_list.remove(request)


# Функция для получения списка из очереди для отправки его на обслуживание
def get_from_queue():
    global requests_queue
    if not requests_queue.empty():
        request = requests_queue.get()
        return request
    else:
        print('Очередь пуста')


# Класс обслуживающего прибора
class Server:
    def __init__(self):
        self.current_request = None

    # Функция дял первоначальных настроек запроса
    def request_settings(self):
        if self.current_request.priority == 2:
            """
            Если на обслуживание поступает требование 2 класса,
            то учитывается остаточное время обслуживания
            """
            self.current_request.service_time =
            ↪  self.current_request.remaining_service_time_class2
            with print_lock:
                print(f'S: Т.к. приоритет запроса №{self.current_request.request_number} '
                      f'равен {self.current_request.priority}, '
                      f'то рассматривается остаточное время обслуживания, '
                      f'равное {self.current_request.remaining_service_time_class2}, '
                      f'поэтому запрос осталось обработать
                      ↪  {self.current_request.service_time} сек')

        self.current_request.start_time = time.time() - start_time
```

```python
    self.current_request.end_time = self.current_request.start_time +
    ↪  self.current_request.service_time

    with print_lock:
        print(f'S: Запрос с параметрами {self.current_request.print()} '
              'принят на обслуживание, время начала обслуживания: ',
              ↪  self.current_request.start_time)

# Функция обслуживания требований
def serve(self):
    global start_time
    global requests_queue

    if not requests_queue.empty():
        request: Request = get_from_queue()
        with print_lock:
            print(f'S: Запрос с параметрами {request.print()} взят из очереди. '
                  f'Еще запросов в очереди: {requests_queue.qsize()}')

        if self.current_request:
            if self.current_request.end_time < time.time() - start_time:
                """
                Если текущее требование может завершиться то завершаем его
                """
                time.sleep(self.current_request.service_time)
                serviced_requests_list.append(self.current_request)

                with print_lock:
                    print(f'S: Запрос с параметрами {self.current_request.print()}
                    ↪  обслужен')

                self.current_request = request
                self.request_settings()
                return

            elif self.current_request.priority == 2 and request.priority == 1:
                """
                Если при поступлении требования 1 уровня приоритета
                на обслуживании находится требования 2 уровня,
                то требование 1 уровня сразу же поступает на обслуживание,
                а требование 2 уровня отправляется в конец очереди с сохраненным
                ↪  остаточным временем обслуживания
                """
                request.start_time = time.time() - start_time
                request.end_time = request.start_time + request.service_time
                time.sleep(abs(request.start_time - self.current_request.start_time))
                self.current_request.remaining_service_time_class2 =
                ↪  (self.current_request.end_time
```

```
                                                                    -
                                                                    ↪  request.start_time)
                requests_queue.put(self.current_request)
                with print_lock:
                    print(f'S: Т.к. во время прибытия требования
                    ↪  №{request.request_number} '
                          f'с приоритетом {request.priority} '
                          f'на обслуживании находилось требование
                          ↪  №{self.current_request.request_number} '
                          f'с приоритетом {self.current_request.priority}, '
                          f'то требование №{self.current_request.request_number}
                          ↪  отправляется в конец очереди '
                          f'с остаточным временем обслуживания, '
                          f'равным {self.current_request.remaining_service_time_class2},
                          ↪  '
                          f'а требование №{request.request_number} '
                          'отправляется на обслуживание.')

                self.current_request = request
                self.request_settings()
                return

            else:
                time.sleep(self.current_request.service_time)
                serviced_requests_list.append(self.current_request)
                with print_lock:
                    print(f'S: Запрос с параметрами {self.current_request.print()}
                    ↪  обслужен')

                self.current_request = request
                self.request_settings()
                return

        else:
            self.current_request = request
            self.request_settings()
            return

    elif not requests_list:
        time.sleep(self.current_request.service_time)
        with print_lock:
            print(f'S: Запрос с параметрами {self.current_request.print()} обслужен')
        serviced_requests_list.append(self.current_request)
        self.current_request = None
        time.sleep(3)
        return

    else:
```

```python
            return


# Функция для генерации времени поступления запросов с помощью пуассоновского процесса
def generate_poisson_process(lambda_val, observation_time):
    """
    Генерирует пуассоновский поток требований.

    Аргументы:
    lambda_val : float
        Интенсивность пуассоновского потока.
    T : float
        Время наблюдения.

    Возвращает:
    list
        Список моментов времени, в которые произошли события.
    """
    time_points = []
    current_time = 0

    while current_time < observation_time:
        # Генерируем интервал времени до следующего события с экспоненциальным
        ↪  распределением
        dt = np.random.exponential(1 / lambda_val)
        current_time += dt

        # Если следующее событие произойдет в пределах времени наблюдения, добавляем его в
        ↪  список
        if current_time < observation_time:
            time_points.append(current_time)

    return time_points


# Функция для генерации запросов
def generate_requests(number_of_requests, lambda_val, observation_time):
    cur_time = 0
    time_array = []

    # Создание списка со временем поступления запросов на обслуживание
    for i in range(int(number_of_requests / lambda_val / observation_time)):
        poisson_process = generate_poisson_process(lambda_val=lambda_val,
        ↪  observation_time=observation_time)
        # print(poisson_process)
        for j in range(len(poisson_process)):
            poisson_process[j] += cur_time
        time_array.extend(poisson_process)
```

```
        cur_time = time_array[-1]

    # num_requests: количество запросов, которые нужно создать.
    global requests_list  # Создание пустого списка requests, который будет заполнен
    ↪  сгенерированными запросами.
    number = 0

    # Заполнение списка запросами
    for request in time_array:
        number += 1
        priority = random.choice([1, 2])
        arrival_time = request
        service_time = random.random()
        requests_list.append(Request(request_number=number, arrival_time=arrival_time,
        ↪  service_time=service_time,
                                     priority=priority))


# Функция, отвечающая за начало работы сервера
def server_start_work(server: Server):
    global start_server
    while start_server:
        server.serve()


# Функция, контролирующая работу сервера, исходя из количества оставшихся запросов
def requests_control(server: Server):
    global requests_queue
    global requests_list
    global start_server
    while len(requests_list) > 0 or requests_queue.qsize() > 0 or server.current_request is
    ↪  not None:
        start_server = True
    else:
        start_server = False


# Функция для подсчета математического ожидания числа требований для запроса каждого
↪  приоритета
def pk_statistics(server: Server, priority, length):  # функция для подсчета pkg
    global requests_queue
    global start_server
    global start_time
    r_list = [0]*length
    t_list = [0]*length
    p_list: list[float] = [0]*length
    k = 0
    k_time = time.time()
```

```python
    while start_server:
        k_current = 0
        for item in list(requests_queue.queue):
            if item.priority == priority:
                k_current += 1

        if server.current_request and server.current_request.priority == priority:
            k_current += 1

        if k_current != k:
            t_list[k] += time.time() - k_time
            r_list[k] += 1
            k_time = time.time()
            k = k_current
    else:
        all_time = time.time() - start_time
        """
        with print_lock:
            print(f'r{priority}_list = {r_list}')
            print(f't{priority}_list = {t_list}')
        """
        n = 0
        for i in range(0, len(p_list)):
            p_list[i] = t_list[i]/all_time
            n += (i * p_list[i])
        with print_lock:
            # print(f'p{priority}_list = {p_list}')
            print(f'математическое ожидание числа требований с приоритетом {priority}
            ↪  n{priority} равно {n}')
    return


# Функция для подсчета математического ожидания длительности пребывания требования для
↪  запроса каждого приоритета
def calculate_statistics():
    u1 = k1 = u2 = k2 = 0

    for request in serviced_requests_list:
        if request.priority == 1:
            u1 += request.end_time - request.arrival_time
            k1 += 1
        else:
            u2 += request.end_time - request.arrival_time
            k2 += 1

    u1 /= k1
    u2 /= k2
```

```python
    with print_lock:
        print(f'математическое ожидание длительности пребывания требования с приоритетом 1
        ↪ u1 равно {u1}')
        print(f'математическое ожидание длительности пребывания требования с приоритетом 2
        ↪ u2 равно {u2}')
    return


def main():
    global requests_list
    global serviced_requests_list

    num = 1000  # Число запросов
    lambda_val = 2  # Интенсивность пуассоновского потока.
    observation_time = 5  # Время наблюдения

    # Генерация запросов
    generate_requests(number_of_requests=num, lambda_val=lambda_val,
    ↪ observation_time=observation_time)
    print(len(requests_list))
    for request in requests_list:
        print(request.request_number, request.priority, request.arrival_time,
        ↪ request.service_time)

    server = Server()  # Идентификация сервера

    # Создания потока для контроля работы сервера
    control_thread = threading.Thread(target=requests_control, args=(server,))
    request_thread = threading.Thread(target=put_in_queue)  # Создание потока для
    ↪ отправления запросов в очередь
    server_thread = threading.Thread(target=server_start_work, args=(server,))  # Создание
    ↪ потока для работы сервера
    # Создание потока для подсчета математического ожидания числа требований для запроса с
    ↪ приоритетом 1
    n1_thread = threading.Thread(target=pk_statistics, args=(server, 1, num,))
    # Создание потока для подсчета математического ожидания числа требований для запроса с
    ↪ приоритетом 2
    n2_thread = threading.Thread(target=pk_statistics, args=(server, 2, num,))

    # Запуск потоков
    control_thread.start()
    n1_thread.start()
    n2_thread.start()
    request_thread.start()
    server_thread.start()

    # Остановка потоков
    request_thread.join()
```

```python
    server_thread.join()
    control_thread.join()
    n1_thread.join()
    n2_thread.join()

    # for request in serviced_requests_list:
    #     print(request.request_number, request.priority, request.arrival_time, request.service_time)
    # print(len(serviced_requests_list))

    # Вывод посчитанного математического ожидания длительности пребывания требования для запроса каждого приоритета
    calculate_statistics()


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

7.4 Результат работы программы

```
математическое ожидание числа требований с приоритетом 1 n1 равно 52.66404577897455
математическое ожидание числа требований с приоритетом 2 n2 равно 82.30856471973952
математическое ожидание длительности пребывания требования с приоритетом 1 u1 равно 113.19999021379294
математическое ожидание длительности пребывания требования с приоритетом 2 u2 равно 154.41865309580263
```