Лабораторная работа 7 по курсу «Вычислительные задачи прикладной математики». Отчёт.

Владислав Соврасов
2-о-051318

1 Сравнение наивного и Gillespie методов для моделирования процесса случайного распада вещества

В первой части работы рассматривается уравнение $\dot{x} = -x, x(0) = x_0$, описывающее поведение среднего количества молекул в процессе распада вещества. Это уравнение было смоделировано с помощью наивного алгоритма, алгоритма Gillespie а также решено методом Рунге-Кутты 4го порядка. Полученные численные решения представлены на рис. 1. Видно, что две стохастические реализации процесса распада отличаются от детерминированного решения и друг от друга. В наивном алгоритме $\Delta t = 0.01/a(0)$, что обеспечивает для данного уравнения выполнение требования $\Delta t \ll 1/a(x)$ в течение всего процесса решения. График решения, полученного наивным алгоритмом, имеет характерные ступеньки, которые означают выполнение нескольких итераций подряд без уменьшения количества молекул. Алгоритм Gillespie совершает итерации только в моменты распада, что позволяет ему работать на порядок быстрее (см. табл. 1).

Метод	Время, с
Naive	0.0257
Gillespie	0.0031

Таблица 1: Время выполнения различных методов

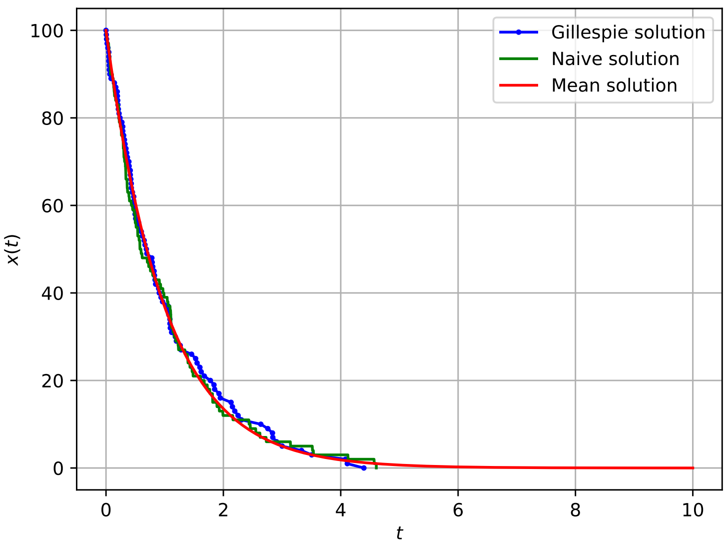


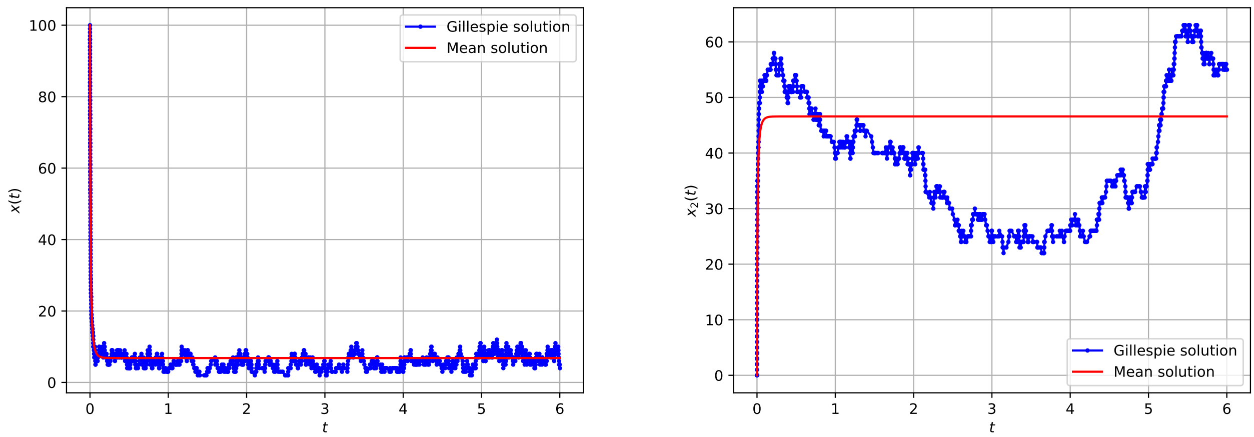
Рис. 1: Решения уравнения распада, полученные различными методами

2 Решение системы уравнений

Во второй части работы следующая система была смоделирована как случайный процесс:

$$\begin{cases} \dot{x} = 2x_2 - 2x^2 \\ \dot{x}_2 = x^2 - x_2 \end{cases}$$

Начальные условия: $x(0) = 100$, $x_2(0) = 0$. Также данная система была решена методом Рунге-Кутты 4го порядка. Все решения получены для промежутка времени $t \in [0, 6]$. Детерминированное решение очень быстро приходит к одному из расположенных на кривой $x_2 = x^2$ состояний равновесия, в то время как стохастическая реализация демонстрирует колебания относительно состояния равновесия.



(a) (b)

Рис. 2: Решения системы уравнений

3 Исходный код

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time

import numpy as np
import matplotlib.pyplot as plt

from lab3 import rungeKuttaMethod


def naive_solver(initial_point, max_time, inc_rules, dec_rules):
    assert len(inc_rules) == len(dec_rules) == 1
    assert len(initial_point) == len(dec_rules)
    time = 0.
    x = initial_point.copy()
    x_s = [initial_point]
    step = 1 / dec_rules[0](x) / 100
    times = [0.]
    while time < max_time:
        a = dec_rules[0](x)
        r = np.random.random()
        if r < a*step:
            x[0] -= 1
        time += step
```

```
            x_s.append(x.copy())
            times.append(time)
            if x[0] <= 0:
                break

    return times, x_s


def gillespie_solver(initial_point, max_time, inc_rules, dec_rules):
    assert len(inc_rules) == len(dec_rules)
    assert len(initial_point) == len(dec_rules)
    time = 0.
    x = initial_point.copy()
    x_s = [initial_point]
    times = [0.]
    while time < max_time:
        v_plus = []
        for rule in inc_rules:
            v_plus.append(rule(x))
        v_minus = []
        for rule in dec_rules:
            v_minus.append(rule(x))
        a_0 = np.sum(np.array(v_plus + v_minus))
        if a_0 <= 0:
            break
        probs = np.array(v_plus + v_minus) / a_0
        r1 = max(np.random.random(), 1e-12)
        tao = np.log(1 / r1) / a_0
        time += tao
        idx = np.random.choice(2*len(initial_point), p=probs)
        if idx >= len(inc_rules):
            x[idx % len(inc_rules)] -= 1
        else:
            x[idx] += 1
        x_s.append(x.copy())
        times.append(time)

    return times, x_s


def main():

    np.random.seed(1)

    t_r, x_r = rungeKuttaMethod(lambda x, t: -x, 0., 10., 100., 1e-4)

    start = time.time()
    t_n, x_n = naive_solver([100], 10, [lambda x: 0], [lambda x: x[0]])
    end = time.time()
```

```
    print('Naive solver time, 1d ', end - start)

    start = time.time()
    t, x = gillespie_solver([100], 10, [lambda x: 0], [lambda x: x[0]])
    end = time.time()
    print('Gillespie time, 1d ', end - start)

    plt.xlabel('$t$')
    plt.ylabel('$x(t)$')
    plt.plot(t, np.array(x).reshape(-1), 'b-o', markersize=2,
             label='Gillespie solution')
    plt.plot(t_n, np.array(x_n).reshape(-1), 'g-', markersize=2,
             label='Naive solution')
    plt.plot(t_r, x_r, 'r-', label='Mean solution')
    plt.grid()
    plt.legend()
    plt.savefig('../pictures/lab7_eq_1.pdf', format = 'pdf')
    plt.clf()

    t_r, x_r = rungeKuttaMethod(lambda x, t: np.array([2*(x[1] - x[0]**2),
                                                         x[0]**2 - x[1]]),
                                0., 6., np.array([100., 0]), 1e-4)

    start = time.time()
    t, x = gillespie_solver([100, 0], 6, [lambda x: 2*x[1], lambda x: x[0]**2],
                            [lambda x: 2*x[0]**2, lambda x: x[1]])
    end = time.time()
    print('Gillespie time, 2d ', end - start)

    plt.xlabel('$t$')
    plt.ylabel('$x(t)$')
    plt.plot(t, np.array(x)[:, 0], 'b-o', markersize=2, label='Gillespie solution')
    plt.plot(t_r, np.array(x_r)[:, 0], 'r-', label='Mean solution')
    plt.grid()
    plt.legend()
    plt.savefig('../pictures/lab7_eq_2_1.pdf', format = 'pdf')
    plt.clf()

    plt.xlabel('$t$')
    plt.ylabel('$x_2(t)$')
    plt.plot(t, np.array(x)[:, 1], 'b-o', markersize=2, label='Gillespie solution')
    plt.plot(t_r, np.array(x_r)[:, 1], 'r-', label='Mean solution')
    plt.grid()
    plt.legend()
    plt.savefig('../pictures/lab7_eq_2_2.pdf', format = 'pdf')


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