МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе
по дисциплине «Дифференциальные уравнения»
Тема: «Определение жесткости пружин-ребер в плоском графе с известной топологией в случае двумерных продольных колебаний (в вершинах предполагаются массы).»

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Колоницкий С.Б.

Санкт-Петербург
2024

Цель работы

Целью данной работы является определение жесткости пружин-ребер в плоском графе с известной топологией в случае двумерных продольных колебаний с помощью метода оптимизации Бройдена — Флетчера — Гольдфарба — Шанно.

Задание

Дан плоский граф с известной топологией (указаны координаты вершин в декартовой системе координат), массы вершин, равновесные длины пружин-ребер, а также траектории движения масс вершин, измеренных с погрешностями (моделируется добавлением шума к выходу прямой задачи). Необходимо определить жесткости пружин-ребер.

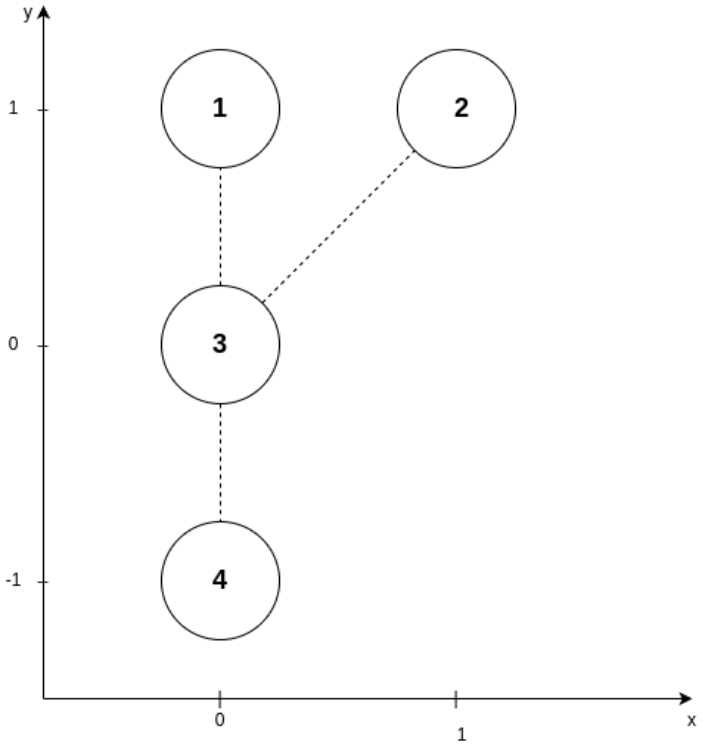

Рисунок 1 - Плоский граф с известной топологией

Описание способа решения задачи

Для решения данной задачи необходимо:

1. Применить итерационный метод численной оптимизации, предназначенный для нахождения локального минимума нелинейного функционала без ограничений, над полученными зашумленным данными для нахождения жесткости пружин-ребер в плоском графе.

2. Так как взять истинные значения траектории движения масс из натурного эксперимента невозможно, они берутся из численного эксперимента: решается соответствующая система дифференциальных уравнений методом семейства Рунге-Кутты.

3. Зашумление полученных “истинных” значений с использованием Гауссовского шума.

Решение прямой задачи

Для решения прямой задачи необходимо найти значения скорости и ускорения движения вершин с известными для них входными параметрами: массы, жесткости и равновесные длины пружин.

Для нахождения данных параметров системы используем дифференциальное уравнение второго закон Ньютона, для которого сила описана законом Гука:

$$\frac{m_i * \delta^2 u_i}{\partial t^2} = - \sum_j K_{ij}(u_i - u_j) \quad \text{, где}$$

m_i - масса вершины

u_i , u_j - координата i-ой и j-ой вершины соответственно

K_{ij} - жесткость пружины между i-ой и j-ой вершиной

Сила упругости между двумя вершинами системы рассчитывается по следующим формулам:

$$F_{\text{упр}} = - k * (r - l_{eq})$$

$$F_x = \frac{F^*(x - x_i)}{r}$$

$$F_y = \frac{F^*(y - y_i)}{r}$$, где

$r = \sqrt{(x - x_i)^2 + (y - y_i)^2}$ - расстояние между вершинами

F_x - сила, вдоль оси Ox

F_y - сила, вдоль оси Oy

l_{eq} - равновесная длина пружины

k - жесткость пружины

$x,\ y$ - координаты текущей вершины

$x_i,\ y_i$ - координаты соседней вершины

Для численного решения системы дифференциальных уравнений используется метод Рунге-Кутты. В данной задаче использован метод Рунге-Кутты четвертого порядка(РК4), который использует таблицу Бутчера для вычисления коэффициентов.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Рисунок 2 - Таблица Бутчера РК4

Метод Рунге-Кутты 4-го порядка является мощным и универсальным инструментом для решения задач, связанных с обыкновенными дифференциальными уравнениями. Его высокая точность, простота реализации и гибкость делают его предпочтительным выбором в широком спектре прикладных задач.

Решение обратной задачи

Добавление шума

Первоначально для решения данной задачи необходимо зашумить данные, полученные на этапе численного интегрирования системы. Для этого используем Гауссовский шум. Гауссов шум моделируется как случайная величина с нормальным распределением:

$$y_{obs} = y_{true} + \varepsilon,\ \varepsilon \sim \mathcal{N}(0, \sigma^2)$$, где

y_{true} - истинные данные.

ε - шум с нулевым средним и стандартным отклонением σ, задающим уровень шума.

Оптимизация параметров с использованием метода BFGS

Для восстановления истинных значений жесткостей пружин на основе зашумленных данных используется метод BFGS. Это итерационный метод оптимизации, который аппроксимирует обратную матрицу Гессе для нахождения направления поиска.

Алгоритм BFGS

- Инициализация:

Начальное приближение параметров K и начальная аппроксимация обратной матрицы Гессе H.

- Вычисление градиента:

Градиент функции ошибки вычисляется численно с использованием метода конечных разностей.

- Направление поиска:

Направление поиска определяется как

$$p = -H * \nabla f$$

- Линейный поиск:

Выполняется линейный поиск для определения длины шага α, которая минимизирует функцию ошибки вдоль направления p.

- Обновление параметров:

Новые параметры вычисляются как

$$x_{new} = x + \alpha * p$$

- Обновление матрицы Гессе:

Матрица H обновляется с использованием векторов s и y, где

$s = x_{new} - x$ и

$y = \nabla f(x_{new}) - \nabla f(x)$

- Критерий остановки:

Итерации продолжаются до тех пор, пока норма градиента не станет меньше заданного порога tol.

Результаты работы программы и график ошибок представлены в приложении А. Так как графики траектории истинных и восстановленных значений совпадают, был рассмотрен именно график разниц траекторий, что позволяет оценить ошибку восстановления. Глядя на график, можно заметить, что ошибка не превосходит 0,002, а значит программа решает задачу с высокой точностью. Результат вывода программы это подтверждает: числа практически совпадают.

Исходный код программы представлен в приложении Б.

Код программы для симуляции представлен в приложении В.

Вывод

В ходе выполнения работы написана программа, которая определяет жесткости пружин-ребер в плоском графе с известной топологией в случае двумерных продольных колебаний с помощью метода оптимизации Бройдена — Флетчера — Гольдфарба — Шанно. Программа достигла высокой точности и выдает результат за малый промежуток времени.

ПРИЛОЖЕНИЕ А

РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

Массы пружин, поданные на вход

0.0, 1.0, 0.0, 0.0 - Масса 1 (x, y, vx, vy)

1.0, 1.0, 0.0, 0.0 - Масса 2 (x, y, vx, vy)

0.0, 0.0, 0.0, 0.0 - Масса 3 (x, y, vx, vy)

0.0, -1.0, 0.0, 0.0 - Масса 4 (x, y, vx, vy)

Равновесные длины пружин

L_eq = [1.0, 1.5, 2.0]

Истинные значения жесткостей пружин

K_true = [2.0, 3.0, 4.0]

```
/home/katya/anaconda3/bin/python /home/katya/comp_math/project/ver3.py
Истинные значения K: [2. 3. 4.]
Восстановленные значения K: [2.00037604 2.99811294 3.99870284]

Process finished with exit code 0
```

Рисунок 3 - Результат работы программы

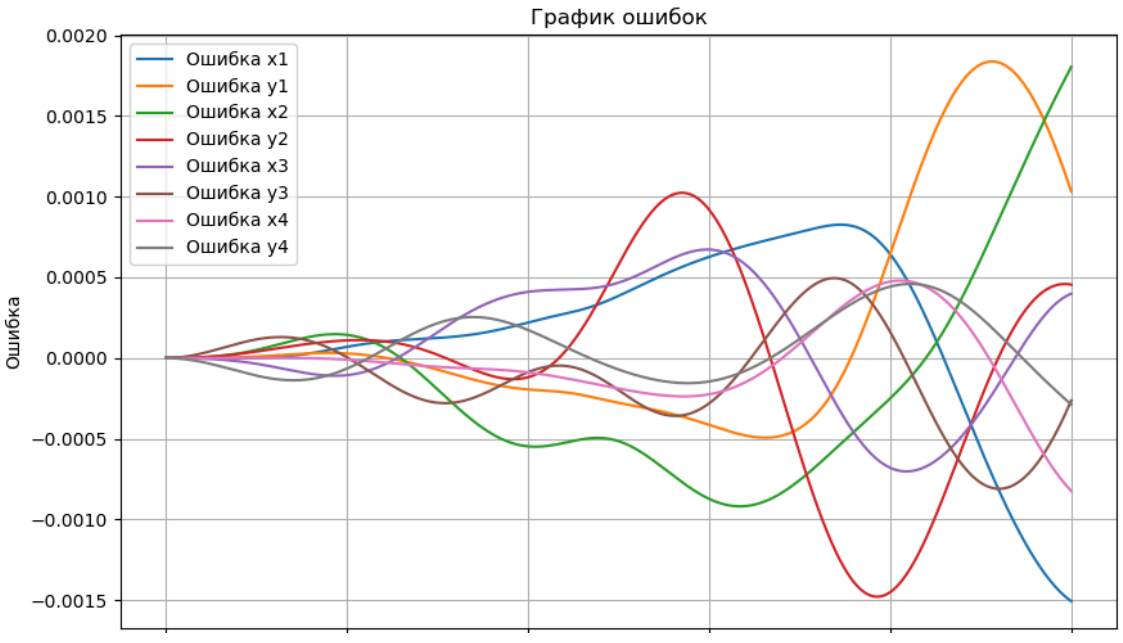

Рисунок 4 - График ошибок

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable, List, Tuple, Any
import os


# Функция для расчета силы упругости
def spring_force(x, y, x_i, y_i, k, l_eq):
      r = np.sqrt((x - x_i) ** 2 + (y - y_i) ** 2)   # Расстояние между вершинами
    if np.isclose(r, 0):  # Избегаем деления на ноль
          return 0, 0  # Нет взаимодействия для совпадающих точек
     F = -k * (r - l_eq)  # Закон Гука с учетом равновесной длины l_eq
    Fx = F * (x - x_i) / r
    Fy = F * (y - y_i) / r
    return Fx, Fy


# Функция для расчёта ускорений всех масс в системе
def compute_forces(_, y, K, L_eq):
    m1, m2, m3, m4 = 1.0, 2.0, 3.0, 4.0
     x1, y1, vx1, vy1, x2, y2, vx2, vy2, x3, y3, vx3, vy3, x4, y4, vx4, vy4 = y

    # Силы упругости между вершинами
    F1x, F1y = spring_force(x1, y1, x3, y3, K[0], L_eq[0])
    F2x, F2y = spring_force(x2, y2, x3, y3, K[1], L_eq[1])
    F3x, F3y = spring_force(x4, y4, x3, y3, K[2], L_eq[2])

    # Ускорения каждой массы
    ax1, ay1 = F1x / m1, F1y / m1
    ax2, ay2 = F2x / m2, F2y / m2
     ax3, ay3 = (-(F1x + F2x + F3x) / m3), (-(F1y + F2y + F3y) / m3)
    ax4, ay4 = F3x / m4, F3y / m4

    return np.array([vx1, vy1, ax1, ay1,
                     vx2, vy2, ax2, ay2,
                     vx3, vy3, ax3, ay3,
                     vx4, vy4, ax4, ay4])


# Функция для коэффициентов Рунге-Кутты
def get_k_coefficients(t: float, y: np.ndarray, h: float, f: Callable[[Any, Any], np.ndarray], tableau: dict) -> List[
    np.ndarray]:
    a_ = tableau['a_']
    c_ = tableau['c_']
    k = []
```

```
        for i in range(len(a_)):
                y_temp = y + h * sum(a_[i][j] * k[j] for j in
range(i))
            k.append(f(t + c_[i] * h, y_temp))
        return k


    # Полный шаг Рунге-Кутты
    def rk_one_step(t: float, y: np.ndarray, h: float, f:
Callable[[Any, Any], np.ndarray], tableau: dict) -> Tuple[
        float, np.ndarray]:
        k = get_k_coefficients(t, y, h, f, tableau)
        b_ = tableau['b_']
            y_next = y + h * sum(b_[i] * k[i] for i in
range(len(b_)))
        return t + h, y_next


    # Решение всей задачи с помощью Рунге-Кутты
    def rk(t0: float, t_end: float, y0: np.ndarray, h: float, f:
Callable[[Any, Any], np.ndarray], tableau: dict) -> Tuple[
        np.ndarray, np.ndarray]:
        t_steps = int((t_end - t0) / h) + 1
        t = np.zeros(t_steps)
        y = np.zeros((y0.size, t_steps))

        t[0] = t0
        y[:, 0] = y0

        for step in range(t_steps - 1):
             t[step + 1], y[:, step + 1] = rk_one_step(t[step],
y[:, step], h, f, tableau)

        return t, y


    # Функция для добавления шума к данным
    def add_noise(data: np.ndarray, noise_level: float) ->
np.ndarray:
        noise = np.random.normal(0, noise_level, size=data.shape)
        return data + noise


    # Функция ошибки для оптимизации
    def error_function(K: np.ndarray, t_obs: np.ndarray, y_obs:
np.ndarray, y0: np.ndarray, h: float, L_eq: List[float],
                       tableau: dict) -> float:
        f = lambda t, y: compute_forces(t, y, K, L_eq)
        h_sim = h / 10  # Более мелкий шаг для высокой точности
        t_sim, y_sim = rk(0, t_obs[-1], y0, h_sim, f, tableau)

            # Сравнение наблюдаемых и модельных данных без
интерполяции
        y_obs_interp = np.array([
                np.interp(t_obs, t_sim, y_sim[i, :]) for i in
range(y_obs.shape[0])
```

```
        ])

        return np.mean((y_obs - y_obs_interp) ** 2)


    # Линейный поиск для адаптивного шага
    def line_search(func, x, p, grad, args, c1=1e-4, c2=0.9,
alpha_max=1.0):
        alpha = alpha_max
        phi0 = func(x, *args)
        phi_prime0 = np.dot(grad, p)

        while True:
            x_new = x + alpha * p
            phi = func(x_new, *args)
              if phi > phi0 + c1 * alpha * phi_prime0 or (phi >=
phi0 and alpha > 0.1):
                alpha *= 0.5
            else:
                break
        return alpha


    # Численное вычисление градиента
    def numerical_gradient(func: Callable[[np.ndarray], float],
params: np.ndarray, eps: float = 1e-6) -> np.ndarray:
        grad = np.zeros_like(params)
        for i in range(len(params)):
            params_eps = np.copy(params)
            params_eps[i] += eps * (np.abs(params[i]) + 1)
            grad[i] = (func(params_eps) - func(params)) / eps
        return grad


    # BFGS-оптимизация с улучшениями
    def bfgs_optimization(func: Callable, x0: List[float], args:
Tuple, bounds: List[Tuple[float, float]],
                          tol: float = 1e-5, max_iter: int = 100)
-> np.ndarray:
        x = np.array(x0)
        n = len(x)
        I = np.eye(n)
        H = I  # Начальная аппроксимация обратной матрицы Гессе

        for iteration in range(max_iter):
            # Вычисляем значение функции и градиент
            f_val = func(x, *args)
              grad = numerical_gradient(lambda p: func(p, *args),
x)

            # Проверяем критерий остановки
            if np.linalg.norm(grad) < tol:
                break

            # Направление поиска
            p = -np.dot(H, grad)
```

```
            # Линейный поиск
            alpha = line_search(func, x, p, grad, args)

            # Обновление параметров
            x_new = x + alpha * p

                grad_new = numerical_gradient(lambda p: func(p,
*args), x_new)
            s = x_new - x
            y = grad_new - grad

            # Проверка для обновления H
            if np.dot(y, s) > 0:
                rho = 1.0 / np.dot(y, s)
                 H = (I - rho * np.outer(s, y)) @ H @ (I - rho *
np.outer(y, s)) + rho * np.outer(s, s)

            x = x_new

        return x


    # Сохраняем и загружаем данные
    npz_file = "simulation_data.npz"


    # Функция для сохранения данных
    def save_simulation_data(filename, **data):
        np.savez_compressed(filename, **data)


    # Функция для загрузки данных
    def load_simulation_data(filename):
        if os.path.exists(filename):
            return np.load(filename)
        return None


    # Функция для построения графиков
    def plot_trajectories(t, y_true, y_obs, y_restored):
        num_bodies = y_true.shape[0] // 4

        for i in range(num_bodies):
            idx_x = i * 4  # Индекс x для текущего тела
            idx_y = idx_x + 1  # Индекс y для текущего тела

            plt.figure(figsize=(10, 6))

            # Истинная траектория
                 plt.plot(t, y_true[idx_x, :], label=f"Истинная
x{idx_x // 4 + 1}")
                 plt.plot(t, y_true[idx_y, :], label=f"Истинная
y{idx_y // 4 + 1}")

            # Наблюдаемая траектория
```

```
                            plt.plot(t,   y_obs[idx_x,   :],   '--',
label=f"Наблюдаемая x{idx_x // 4 + 1}")
                            plt.plot(t,   y_obs[idx_y,   :],   '--',
label=f"Наблюдаемая y{idx_y // 4 + 1}")

            # Восстановленная траектория
                        plt.plot(t,   y_restored[idx_x,   :],   ':',
label=f"Восстановленная x{idx_x // 4 + 1}")
                        plt.plot(t,   y_restored[idx_y,   :],   ':',
label=f"Восстановленная y{idx_y // 4 + 1}")

            plt.xlabel("Время")
            plt.ylabel("Координаты")
            plt.title(f"Траектория тела {i + 1}")
            plt.legend()
            plt.grid()
            plt.savefig(f"trajectory_body_{i + 1}.png")
            # plt.show()


    def plot_error(t, y_true, y_restored):
        num_bodies = y_true.shape[0] // 4

        plt.figure(figsize=(10, 6))

        for i in range(num_bodies):
            idx_x = i * 4  # Индекс x для текущего тела
            idx_y = idx_x + 1  # Индекс y для текущего тела

            # Вычисляем ошибку
            error_x = y_true[idx_x, :] - y_restored[idx_x, :]
            error_y = y_true[idx_y, :] - y_restored[idx_y, :]

            # Строим график ошибки
              plt.plot(t, error_x, label=f"Ошибка x{idx_x // 4 +
1}")
              plt.plot(t, error_y, label=f"Ошибка y{idx_y // 4 +
1}")

        plt.xlabel("Время")
        plt.ylabel("Ошибка")
        plt.title(f"График ошибок")
        plt.legend()
        plt.grid()
        plt.savefig(f"error_body.png")
        # plt.show()


    # Главная программа
    if __name__ == "__main__":
        data = load_simulation_data(npz_file)

        if data is not None:
            t = data["t"]
            y_true = data["y_true"]
            y_obs = data["y_obs"]
```

```
            y_restored = data["y_restored"]
            K_estimated = data["K_estimated"]
            K_true = data["K_true"]

        else:
            # Задаём таблицу Бутчера
            tableau = {
                'a_': [
                    [0, 0, 0, 0],
                    [0.5, 0, 0, 0],
                    [0, 0.5, 0, 0],
                    [0, 0, 1, 0]
                ],
                'b_': [1 / 6, 1 / 3, 1 / 3, 1 / 6],
                'c_': [0, 0.5, 0.5, 1]
            }

            # Начальные условия
             y0 = np.array([0.0, 1.0, 0.0, 0.0,  # Масса 1 (x, y,
vx, vy)
                            1.0, 1.0, 0.0, 0.0,  # Масса 2 (x, y,
vx, vy)
                            0.0, 0.0, 0.0, 0.0,  # Масса 3 (x, y,
vx, vy)
                             0.0, -1.0, 0.0, 0.0])  # Масса 4 (x,
y, vx, vy)

            t_end = 10
            h = 0.01
            L_eq = [1.0, 1.5, 2.0]  # Равновесные длины пружин
                K_true = [2.0, 3.0, 4.0]  # Истинные значения
жесткостей пружин

            # Генерация "истинных" данных
              f_true = lambda t, y: compute_forces(t, y, K_true,
L_eq)
            t, y_true = rk(0, t_end, y0, h, f_true, tableau)

            # Добавление шума к данным
            noise_level = 0.05
            y_obs = add_noise(y_true, noise_level)

            # Оптимизация для восстановления жесткостей пружин
                K_initial_guess = [1.0, 1.0, 1.0]  # Начальное
приближение
             bounds = [(0.1, 10.0), (0.1, 10.0), (0.1, 10.0)]  #
Ограничения на жесткости

            # Выполняем оптимизацию
            K_estimated = bfgs_optimization(
                error_function,
                K_initial_guess,
                args=(t, y_obs[:4, :], y0, h, L_eq, tableau),
                bounds=bounds
            )
```

```
            # Восстановленные данные
                _, y_restored = rk(0, t_end, y0, h, lambda t, y:
compute_forces(t, y, K_estimated, L_eq), tableau)

            # Завершающий блок сохранения данных
                save_simulation_data(npz_file, t=t, y_true=y_true,
y_obs=y_obs, y_restored=y_restored, K_estimated=K_estimated,
                                     K_true=K_true)

        # Построение графиков
        plot_trajectories(t, y_true, y_obs, y_restored)
        plot_error(t, y_true, y_restored)

        print("Истинные значения K:", K_true)
        print("Восстановленные значения K:", K_estimated)
```

ПРИЛОЖЕНИЕ В

КОД ПРОГРАММЫ ДЛЯ СИМУЛЯЦИИ

```
from matplotlib.animation import FuncAnimation
from matplotlib import animation
import matplotlib.pyplot as plt
import numpy as np
from ver3 import load_simulation_data


def animate_bodies(filename, t, y_true):
    fig, ax = plt.subplots(figsize=(8, 8))
    num_bodies = y_true.shape[0] // 4

                x_min,   x_max   =   np.min(y_true[0::4,   :]),
np.max(y_true[0::4, :])
                y_min,   y_max   =   np.min(y_true[1::4,   :]),
np.max(y_true[1::4, :])
    ax.set_xlim(x_min - 1, x_max + 1)
    ax.set_ylim(y_min - 1, y_max + 1)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_title("Движение тел в системе")
    ax.grid()

        lines  =  [ax.plot([],  [],  'o-',  markersize=5,  lw=2,
label=f"Тело массой {i + 1}")[0] for i in range(num_bodies)]

    connections = [(0, 2), (1, 2), (2, 3)]
     springs = [ax.plot([], [], '-', color='gray', lw=1.5)[0]
for _ in range(len(connections))]

    ax.legend()

    # Функция инициализации
    def init():
        for line in lines:
            line.set_data([], [])
        for spring in springs:
            spring.set_data([], [])
        return lines + springs

    # Функция обновления анимации
    def update(frame):
        for i, line in enumerate(lines):
            x_idx = i * 4
               x, y = y_true[x_idx, frame], y_true[x_idx + 1,
frame]
            line.set_data([x], [y])

        for i, (a, b) in enumerate(connections):
              spring_x = [y_true[a * 4, frame], y_true[b * 4,
frame]]
```

```
                    spring_y = [y_true[a * 4 + 1, frame], y_true[b *
4 + 1, frame]]
                springs[i].set_data(spring_x, spring_y)

            return lines + springs

             ani = FuncAnimation(fig, update, frames=len(t),
init_func=init, blit=True, interval=50)

        writergif = animation.PillowWriter(fps=30)
        ani.save(f'{filename}.gif', writer=writergif)

        plt.show()


    if __name__ == "__main__":
        npz_file = "simulation_data.npz"
        data = load_simulation_data(npz_file)

        if data is not None:
            t = data["t"]
            y_true = data["y_true"]
            y_obs = data["y_obs"]
            y_restored = data["y_restored"]
            K_estimated = data["K_estimated"]
            K_true = data["K_true"]

        animate_bodies("true_bodies_motion",t, y_true)
        animate_bodies("restored_bodies_motion", t, y_restored)
```