Міністерство освіти ти науки України
Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра обчислювальної математики

Звіт до лабораторної роботи №7 на тему:
"Чисельне інтегрування
диференційних рівнянь"

Виконав студент групи ОМ-3
Скибицький Нікіта

Київ, 2019

Зміст

1 Постановка задачі

Автомобіль маси M, який підтримується прижуною з демпфером, переміщується з постійною горизонтальною швидкістю. В момент часу $t = 0$ центр тяжіння автомобіля знаходиться на відстані h_0 від землі, і при цьому вертикальна швидкість відсутня. Надалі вертикальний зсув дороги від основного рівня описується функцією $x_0(t)$.

Припустимо, що пружина лінійна, з коефіцієнтом пружності k, а коефіцієнт демпфірування r є нелінійною функцією відносно швидкості двох кінці демфера:

$$r = r_0 \cdot \Big(1 + c \cdot \big|\dot{x} - \dot{x}_0\big|\Big). \tag{1.1}$$

Легко показати, що зсув $x(t)$ центру тяжіння автомобіля є розв'язком звичайного диференціального рівняння другого порядку

$$\ddot{x} = -\frac{1}{M} \cdot \Big(k \cdot \big(x - x_0\big) + r \cdot \big(\dot{x} - \dot{x}_0\big)\Big) \tag{1.2}$$

з початковими умовами

$$x(0) = \dot{x}(0) = 0. \tag{1.3}$$

Обчислити $x(t)$ на проміжку $0 \leq t \leq t_{\max}$ для контуру дороги, який описується функцією

$$x_0(t) = A \cdot (1 - \cos(\omega t)), \tag{1.4}$$

де $2A$ — максимальний зсув основного рівня.

Зауважимо, що для лінійного випадку ($c = 0$) докритичному, критичному, і позакритичному демпфіруванню відповідають значення коефіцієнта

$$\xi = \frac{r}{2\sqrt{kM}} \tag{1.5}$$

менший, рівний, і більший одиниці відповідно.

Програма повинна виводити значення: t, $x_0(t)$, $x(t)$, $\dot{x}(t)$, $\xi(t)$.

Параметри задачі:

M	A	ω	k	$t_{\max}$	r_0	c
10	2	7	640	5	160	1

2 Аналітичні маніпуляції

2.1 Попередня обробка моделі

Позначимо $y = \dot{x}$, тоді $\dot{y} = \ddot{x}$ і маємо наступну систему:

$$\begin{cases} \dot{x} = y = f(t, x, y), \\ \dot{y} = -\dfrac{1}{M} \cdot \Big(k \cdot \big(x - x_0\big) + r \cdot \big(y - \dot{x}_0\big)\Big) = g(t, x, y). \end{cases} \tag{2.1}$$

Також зауважимо, що ми використовуємо функцію $\dot{x}_0(t)$, тому наведемо тут її явний вигляд:

$$\dot{x}_0(t) = A \cdot \omega \cdot \sin(\omega t). \tag{2.2}$$

2.2 Метод Рунге-Кутти

Для моделювання будемо використовувати явний метод Рунге-Кутти четвертого порядку для системи 2×2.

Спочатку обчислюються наступні коефіцієнти:

$$\begin{aligned} k_1 &= h \cdot f(t, x, y), \\ q_1 &= h \cdot g(t, x, y), \\ k_2 &= h \cdot f(t + h/2, x + k_1/2, y + q_1/2), \\ q_2 &= h \cdot g(t + h/2, x + k_1/2, y + q_1/2), \\ k_3 &= h \cdot f(t + h/2, x + k_2/2, y + q_2/2), \\ q_3 &= h \cdot g(t + h/2, x + k_2/2, y + q_2/2), \\ k_4 &= h \cdot f(t + h, x + k_3, y + q_3), \\ q_4 &= h \cdot g(t + h, x + k_3, y + q_3), \end{aligned} \tag{2.3}$$

А потім виконуються наступні кроки за відповідними координатами:

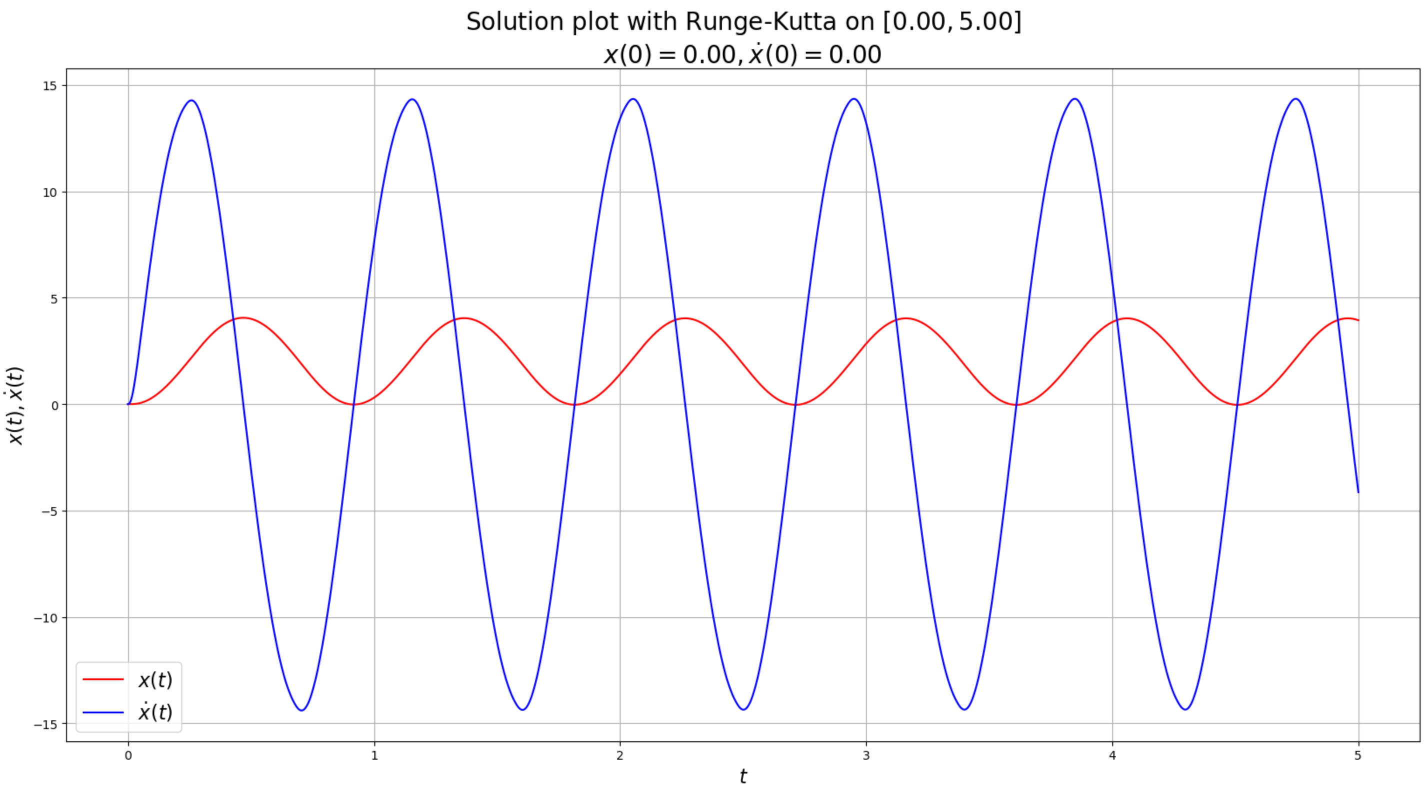
$$\begin{aligned} t &\mathrel{+}= h, \\ x &\mathrel{+}= (k_1 + 2k_2 + 2k_3 + k_4)/6, \\ y &\mathrel{+}= (q_1 + 2q_2 + 2q_3 + q_4)/6. \end{aligned} \tag{2.4}$$

3 Чисельне моделювання

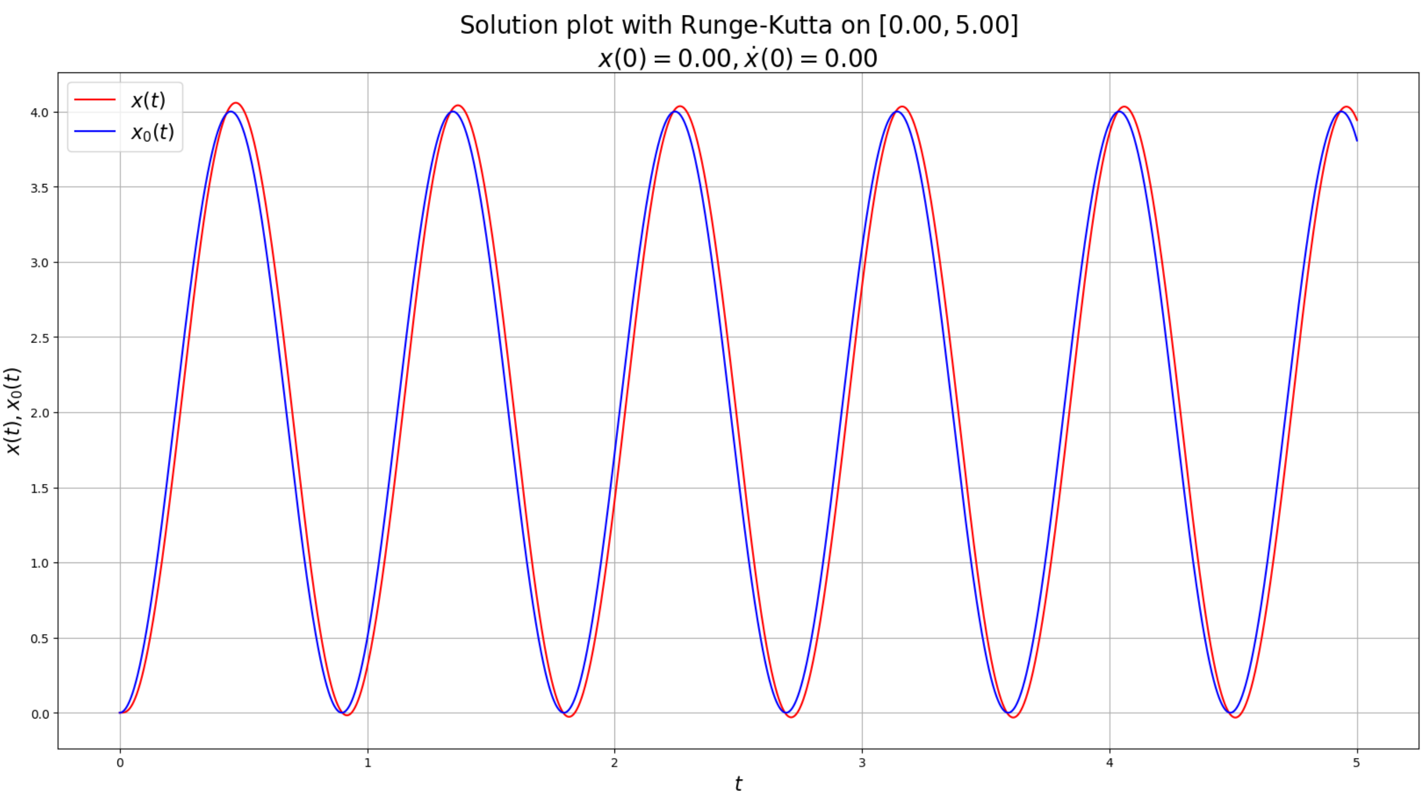
3.1 Графіки

Було отримано наступні графіки:

3.1.1 Висота центру мас і вертикальна швидкість авто

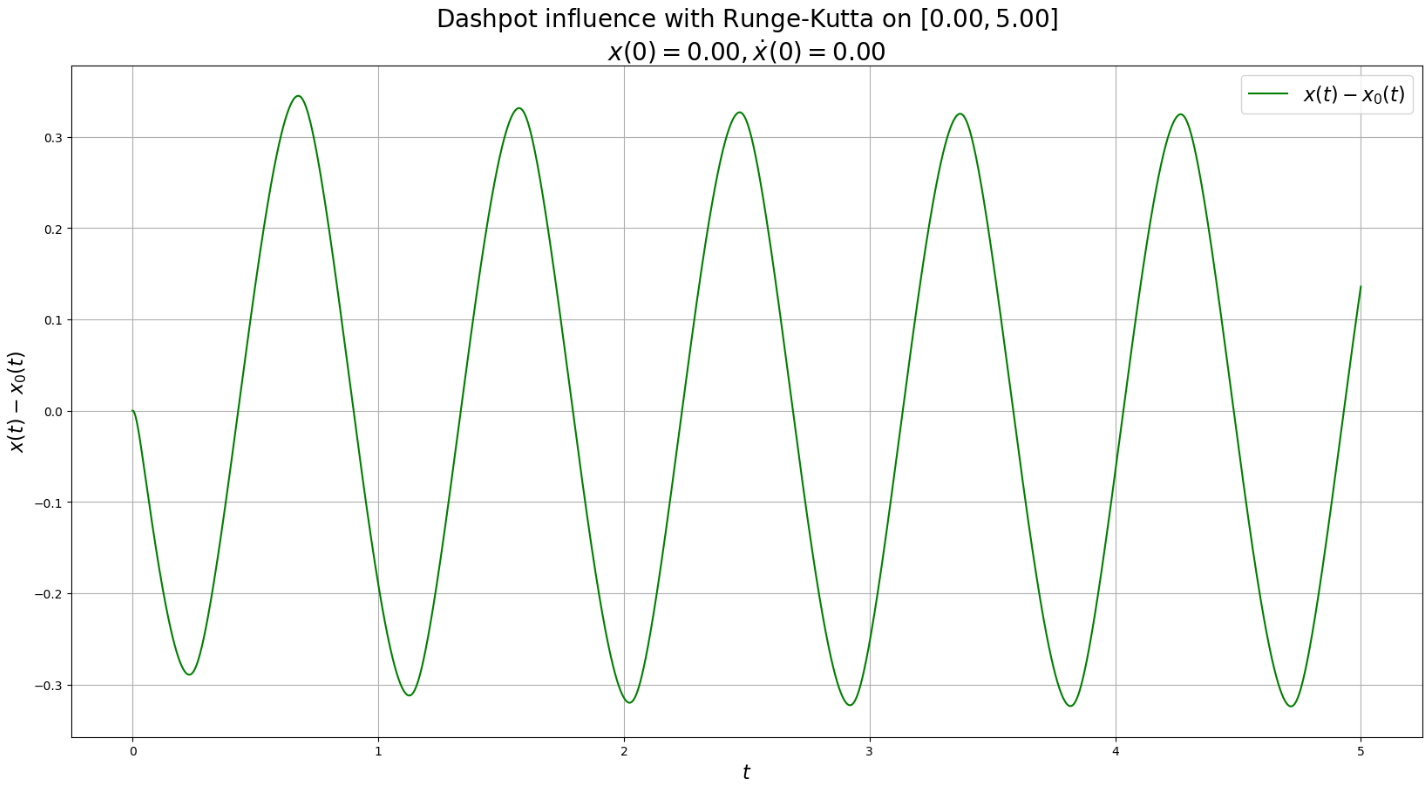


3.1.2 Висота дороги і центру мас автомобіля

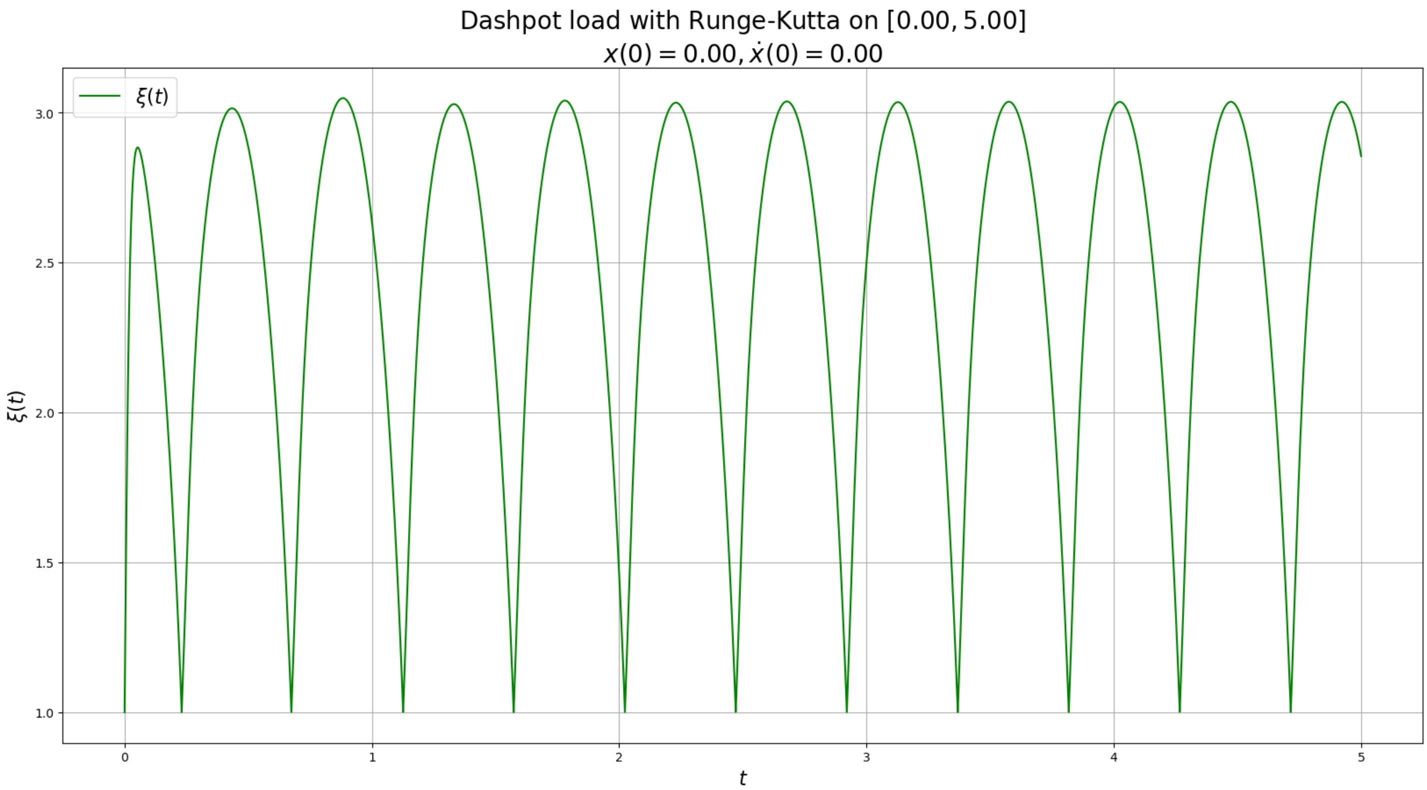


Зауважимо, що висота центру мас авто нібито буває “нижче” ніж рівень дороги, бо тут не враховується висота h_0, а варто було б.

3.1.3 Внесок демпфера у вертикальний рух



3.1.4 Критичність навантаження на демпфер



3.2 Код

3.2.1 Модель демпфера

Для моделювання демпфера було написано наступний клас:

```
class Dashpot:
    def __init__(self, k: float, r_0: float, c: float):
        assert r_0 > 0, "r_0 must be positive"
        assert k > 0, "k must be positive"
        self._k, self._r_0, self._c = k, r_0, c

    @property
    def k(self):
        return self._k

    @property
    def r_0(self):
        return self._r_0

    @property
    def c(self):
        return self._c

    def __repr__(self):
        return f'Dashpot(k={self.k}, r_0={self.r_0}, c={self.c})'

    def r(self, dot_x: float, dot_x_0: float) -> float:
        return self.r_0 * (1 + self.c * abs(dot_x - dot_x_0))

    def xi(self, r: float, m: float) -> float:
        return r / (2 * sqrt(self.k * m))

    def xi(self, dot_x: float, dot_x_0: float, m: float) -> float:
        return self.r(dot_x, dot_x_0) / (2 * sqrt(self.k * m))
```

Як бачимо, він зберігає в собі внутрішні незмінні параметри демпфера, а також надає функціональність для обчислення коефіцієнту демпфірування і значененя рівня критичності демпфірування. Обидва згадані методи потребують також відомостей про “зовнішній світ”.

3.2.2 Модель машини

Для моделювання машини було написано наступний клас:

```
class Car:
    def __init__(self, m: float, dashpot: Dashpot):
        assert m > 0, "m must be positive"
        self._m, self._dashpot = m, dashpot
        self._x, self._dot_x = 0, 0

    @property
    def m(self):
        return self._m
```

```
    @property
    def dashpot(self):
        return self._dashpot

    @property
    def x(self):
        return self._x

    @property
    def dot_x(self):
        return self._dot_x

    def __repr__(self):
        return f'Car(x={self.x:.7f}, dot_x={self.dot_x:.7f}, ' + \
            f'm={self.m}, dashpot={self.dashpot})'

    def r(self, dot_x_0: float) -> float:
        return self.dashpot.r(self.dot_x, dot_x_0)

    def xi(self, dot_x_0: float) -> float:
        return self.dashpot.xi(self.dot_x, dot_x_0, self.m)
```

Як бачимо, він зберігає в собі внутрішні незмінні параметри машини (включаючи положення машини, і параметри демпфера цієї машини), а також надає функціональність для обчислення коефіцієнту демпфірування і значененя рівня критичності демпфірування. Обидва згадані методи потребують також відомостей про "зовнішній світ".

3.2.3 Модель дороги

Для моделювання дороги було написано наступний клас:

```
class Road:
    def __init__(self, a: float, omega: float):
        assert a >= 0, "a must be positive"
        assert omega >= 0, "omega must be positive"
        self._a, self._omega = a, omega

    @property
    def a(self):
        return self._a

    @property
    def omega(self):
        return self._omega

    def __repr__(self):
        return f'Road(a={self.a}, omega={self.omega})'

    def x_0(self, t: float) -> float:
        return self.a * (1 - cos(self.omega * t))
```

```
    def dot_x_0(self, t: float) -> float:
        return self.a * self.omega * sin(self.omega * t)
```

Як бачимо, він зберігає в собі внутрішні незмінні параметри дороги, а також надає функціональність для обчислення положення дороги під машиною у певний момент часу.

3.2.4 Модель сцени (фізичного світу)

Сценою (eng. *stage*) у програмуванні прийнято називати клас, що містить і контролює життя усіх об'єктів які моделюються.

Для моделювання сцени було написано наступний клас:

```
class Stage:
    def __init__(self, car: Car, road: Road):
        self._car, self._road = car, road
        self._t = 0

    @property
    def car(self):
        return self._car

    @property
    def x(self):
        return self.car.x

    @property
    def dot_x(self):
        return self.car.dot_x

    @property
    def road(self):
        return self._road

    @property
    def t(self):
        return self._t

    @property
    def r(self) -> float:
        return self.car.r(self.road.dot_x_0(self.t))

    @property
    def xi(self) -> float:
        return self.car.xi(self.road.dot_x_0(self.t))

    @property
    def x_0(self) -> float:
        return self.road.x_0(self.t)

    @property
```

```
    def dot_x_0(self) -> float:
        return self.road.dot_x_0(self.t)

    def __repr__(self):
        return f'Stage(t={self.t:.7f}, car={self.car}, road={self.road})'

    def move(self, dt: float):
        def f(t: float, x: float, y: float) -> float:
            return y

        def g(t: float, x: float, y: float) -> float:
            return - 1 / self.car.m * (
                self.car.dashpot.k * (x - self.x_0) + self.r * (y - self.dot_x_0)
            )

        k1 = dt * f(self.t, self.x, self.dot_x)
        q1 = dt * g(self.t, self.x, self.dot_x)

        k2 = dt * f(self.t + dt / 2, self.x + k1 / 2, self.dot_x + q1 / 2)
        q2 = dt * g(self.t + dt / 2, self.x + k1 / 2, self.dot_x + q1 / 2)

        k3 = dt * f(self.t + dt / 2, self.x + k2 / 2, self.dot_x + q2 / 2)
        q3 = dt * g(self.t + dt / 2, self.x + k2 / 2, self.dot_x + q2 / 2)

        k4 = dt * f(self.t + dt, self.x + k3, self.dot_x + q3)
        q4 = dt * g(self.t + dt, self.x + k3, self.dot_x + q3)

        self._t, self.car._x, self.car._dot_x = self.t + dt, \
            self.x + (k1 + 2 * k2 + 2 * k3 + k4) / 6, \
            self.dot_x + (q1 + 2 * q2 + 2 * q3 + q4) / 6
```

Як бачимо, він містить метод `move` який і відповідає за моделювання поведінки усіх об'єктів що перебувають на сцені.

3.2.5 Програма-драйвер

Окремо від вищезгаданих класів було написано програму-драйвер:

```
def splot(save=False):
    if save:
        plt.figure(figsize=(20,10))

    plt.xlabel('$t$', fontsize=16)


def fplot(save=False):
    global _IMG
    _IMG += 1
    plt.legend(fontsize=16)
    plt.grid(True)
```

```python
    if save:
        plt.savefig(f'../tex/{_IMG}.png', bbox_inches='tight')
    else:
        plt.get_current_fig_manager().full_screen_toggle()
        plt.show()


if __name__ == '__main__':
    _IMG = 0
    T_MAX = 5 + 1e-7
    h_t, h_x, h_x_0, h_dot_x, h_xi = [], [], [], [], []
    dt = .0001
    stage = Stage(
        car=Car(m=10, dashpot=Dashpot(k=640, r_0=160, c=1)),
        road=Road(a=2, omega=7)
    )

    while stage.t <= T_MAX:
        h_t.append(stage.t)
        h_x.append(stage.x)
        h_x_0.append(stage.x_0)
        h_dot_x.append(stage.dot_x)
        h_xi.append(stage.xi)
        print(stage)
        stage.move(dt)

    splot(save=False)
    plt.title(
        f'Solution plot with Runge-Kutta on $[{h_t[0]:.2f}, {h_t[-1]:.2f}]$\n'
        f'$x(0) = {h_x[0]:.2f}, \\dot x(0) = {h_dot_x[0]:.2f}$', fontsize=20
    )
    plt.ylabel('$x(t), \\dot x(t)$', fontsize=16)
    plt.plot(h_t, h_x, 'r-', label='$x(t)$')
    plt.plot(h_t, h_dot_x, 'b-', label='$\\dot x(t)$')
    fplot(save=False)

    splot(save=False)
    plt.title(
        f'Solution plot with Runge-Kutta on $[{h_t[0]:.2f}, {h_t[-1]:.2f}]$\n'
        f'$x(0) = {h_x[0]:.2f}, \\dot x(0) = {h_dot_x[0]:.2f}$', fontsize=20
    )
    plt.ylabel('$x(t), x_0(t)$', fontsize=16)
    plt.plot(h_t, h_x, 'r-', label='$x(t)$')
    plt.plot(h_t, h_x_0, 'b-', label='$x_0(t)$')
    fplot(save=False)

    splot(save=False)
    plt.title(
        f'Dashpot influence with Runge-Kutta on $[{h_t[0]:.2f}, {h_t[-1]:.2f}]$\n'
        f'$x(0) = {h_x[0]:.2f}, \\dot x(0) = {h_dot_x[0]:.2f}$', fontsize=20
```

```
    )
    plt.ylabel('$x(t) - x_0(t)$', fontsize=16)
    plt.plot(h_t, [h_x[i] - h_x_0[i] for i in range(len(h_t))],
        'g-', label='$x(t) - x_0(t)$')
    fplot(save=False)

    splot(save=False)
    plt.title(
        f'Dashpot load with Runge-Kutta on $[{h_t[0]:.2f}, {h_t[-1]:.2f}]$\n'
        f'$x(0) = {h_x[0]:.2f}, \\dot x(0) = {h_dot_x[0]:.2f}$', fontsize=20
    )
    plt.ylabel('$\\xi(t)$', fontsize=16)
    plt.plot(h_t, h_xi, 'g-', label='$\\xi(t)$')
    fplot(save=False)
```

Як бачимо, вона не містить жодної складної логіки (у чому і була суть допоміжних класів), тобто зі сторони клієнту робота з реалізованою класами функціональністю дуже проста.

Єдиною хоча б трохи цікавою особливістю коду, яка вспливає тілкьи у програмі-драйвері є можливість друкувати у людському форматі стан сцени (а також машини, демпфера, і дороги), приблизно у такому форматі:

```
Stage(t=0,
        car=Car(
                x=0, dot_x=0, m=10,
                dashpot=Dashpot(k=640, r_0=160, c=1)
        ),
        road=Road(a=2, omega=7)
)
```

або

```
Stage(t=1.2345000,
        car=Car(
                x=3.2203857, dot_x=11.6096103, m=10,
                dashpot=Dashpot(k=640, r_0=160, c=1)
        ),
        road=Road(a=2, omega=7)
)
```