```python
#! python3.7
# -*- coding: utf-8 -*-
from numpy import zeros, linspace, tanh, eye, linalg
from matplotlib.pyplot import style, figure, axes
from celluloid import Camera

# Набор команд, за счёт которых анимация строится в отдельном окне
from IPython import get_ipython
get_ipython().run_line_magic('matplotlib', 'qt')

# Определение функции, задающей начальное условие
def u_init(x) :
    u_init = 0.5*tanh((x - x_0)/eps)
    return u_init

# Определение функции, задающей левое граничное условие
def u_left(t) :
    u_left = -0.5
    return u_left

# Определение функции, задающей правое граничное условие
def u_right(t) :
    u_right = 0.5
    return u_right

def k(x) :
    if x < x_special :
        k = 1.
    elif x >= x_special :
        k = 5.
    return k

# Определение функции, вычисляющей значение x_n
def x(n) :
    x = a + (b-a)/N*n
    return x

# Функция f подготавливает массив, содержащий элементы вектор-функции,
# определяющей правую часть решаемой системы ОДУ
def f(y,t):
    f = zeros(N+1)
    f[0] = y[0] - u_left(t)
    for n in range(1,N) :
        f[n] = 4*eps*(k(x(n+1/2))/(x(n+1) - x(n))*(y[n+1] - y[n]) \
            - k(x(n-1/2))/(x(n) - x(n-1))*(y[n] - y[n-1])) \
            + (y[n+1]**2 - y[n-1]**2) \
            + (y[n-1]**3 + y[n]**3)*(x(n) - x(n-1)) \
            + (y[n]**3 + y[n+1]**3)*(x(n+1) - x(n))
    f[N] = y[N] - u_right(t)
    return f

# Функция подготавливает матрицу дифференциального оператора решаемой системы ОДУ
def D() :
    D = zeros((N+1,N+1))
    # Определениене ненулевых элементов матрицы D
    for n in range(1,N) :
        D[n,n-1] = x(n) - x(n-1)
        D[n,n] = x(n+1) - x(n-1)
        D[n,n+1] = x(n+1) - x(n)
    return D

# Функция подготавливает массив, содержащий элементы матрицы Якоби f_u
def f_y(y,t):
    f_y = zeros((N+1,N+1))
    # Определениене ненулевых элементов матрицы Якоби
    f_y[0,0] = 1.
    for n in range(1,N) :
        f_y[n,n-1] = 4*eps*(k(x(n-1/2))/(x(n) - x(n-1))) - 2*y[n-1] \
            + (3*y[n-1]**2)*(x(n) - x(n-1))
```

```
            f_y[n,n] = 4*eps*(-k(x(n+1/2))/(x(n+1) - x(n)) \
                - k(x(n-1/2))/(x(n) - x(n-1))) \
                + (3*y[n]**2)*(x(n) - x(n-1)) \
                +  (3*y[n]**2)*(x(n+1) - x(n))
            f_y[n,n+1] = 4*eps*(k(x(n+1/2))/(x(n+1) - x(n))) + 2*y[n+1] \
                + (3*y[n+1]**2)*(x(n+1) - x(n))
        f_y[N,N] = 1.
        return f_y

    # Определение входных данных задачи
    a = 0.; b = 1.
    t_0 = 0.; T = 6.0

    x_0 = 0.6
    eps = 10**(-2.0)

    # Определение точки разрыва коэффициента
    x_special = 0.7

    # Определение параметра схемы (нужный раскомментировать)
    alpha = (1 + 1j)/2  # CROS1 (схема Розенброка с комплексным коэффициентом)
    # alpha = 1.          # DIRK1 (обратная схема Эйлера)

    # Определение числа интервалов пространственно-временной сетки,
    # на которой будет искаться приближённое решение
    N = 200; M = 200

    # Определение сетки по времени
    tau = (T - t_0)/M; t = linspace(t_0,T,M+1)

    # Выделение памяти под массив сеточных значений решения УЧП
    # В строке с номером m этого массива будут храниться сеточные значения решения,
    # соответствующие моменту времени t_m
    u = zeros((M + 1,N + 1))
    # Выделение памяти под вспомогательный массив y
    y = zeros((M + 1,N + 1))

    # Задание начального условия (на начальном временном слое)
    for n in range(N + 1) :
        u[0,n] = u_init(x(n))

    # Задание начального условия решаемой системы ОДУ
    y[0] = u[0]

    # Реализация схемы из семейства ROS1
    # (конкретная схема определяется коэффициентом alpha)
    for m in range(M) :
        w_1 = linalg.solve(D()- alpha*tau*f_y(y[m],t[m]),f(y[m],t[m] + tau/2))
        y[m + 1] = y[m] + tau*w_1.real
        u[m + 1] = y[m + 1]

    # Анимация отрисовки решения
    style.use('dark_background')
    fig = figure()
    camera = Camera(fig)
    ax = axes(xlim=(a,b), ylim=(-1.,1.))
    ax.set_xlabel('x'); ax.set_ylabel('u')
    for m in range(M + 1) :
        # Отрисовка решения в момент времени t_m
        ax.plot([x(n) for n in range(N+1)],u[m], color='y', ls='-', lw=2)
        # Отрисовка границы раздела сред
        ax.plot([x_special,x_special],[-1., 1.], color='r', ls='--', lw=1)
        camera.snap()
    animation = camera.animate(interval=50, repeat=False, blit=True)

    # Листинг программы, реализущей решение нелинейного уравнения
    # типа Бюргерса с помощью бикомпактной схемы
    # в случае слоистой среды
```