```python
#! python3.7
# -*- coding: utf-8 -*-
from numpy import zeros, linspace, exp, 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 = -1/lambd + exp(-x**2)
    return u_init

# Определение функции, задающей левое граничное условие на производную
def mu(t) :
    mu = t
    return mu

# Определение функции, задающей правое граничное условие
def u_right(t) :
    u_right = -1/lambd
    return u_right

# Определение функции, порождающей квазиравномерную сетку,
# которая покрывает полупрямую x \in [a,+\infty]
def x(n) :
    a = 0.
    alpha = 0.; beta = 1.
    c = 1.; m = 1.
    xi_n = (beta - alpha)/N*n
    x_n = a + c*xi_n/(1. - xi_n**2)**m
    return x_n

# Функция f подготавливает массив, содержащий элементы вектор-функции,
# определяющей правую часть решаемой системы ОДУ
def f(y,x,N,t,mu,u_right) :
    f = zeros(2*N-1)
    f[0] = ((y[1] - y[0])*(2 + (x(2) - x(1))/(x(1) - x(0))) - (y[2] - y[1])*((x(1) - x(0))/(x(2) - x(1))))/(x(2) - x(0)) - mu(t)
    for n in range(1,N-1) :
        f[n] = -1/(2*(x(n+1/2) - x(n-1/2)))*((y[n+1] - y[n])/(x(n+3/4) - x(n+1/4)) - (y[n] - y[n-1])/(x(n-1/4) - x(n-3/4)))
    f[N-1] = -1/(2*(x(N-1/2) - x(N-3/2)))*((u_right(t) - y[N-1])/(x(N-1/4) - x(N-3/4)) - (y[N-1] - y[N-2])/(x(N-5/4) - x(N-7/4)))
    for n in range(N,2*N-1) :
        f[n] = y[n - N + 1]**2 - y[n]
    return f

# Функция подготавливает матрицу дифференциального оператора решаемой системы ОДУ
def D(x,N) :
    D = zeros((2*N-1,2*N-1))
    # Определениене ненулевых элементов матрицы D
    for n in range(1,N) :
        D[n,n-1] = 1/(2*(x(n+1/2) - x(n-1/2))*(x(n-1/4) - x(n-3/4)))
        D[n,n] = 1/(2*(x(n+1/2) - x(n-1/2)))*(-1/(x(n+3/4) - x(n+1/4)) - 1/(x(n-1/4) - x(n-3/4))) - lambd
    for n in range(1,N-1) :
        D[n,n+1] = 1/(2*(x(n+1/2) - x(n-1/2))*(x(n+3/4) - x(n+1/4)))
    for n in range(1,N) :
        D[n,n+N-1] = -lambd**2/2
    return D

# Функция подготавливает массив, содержащий элементы матрицы Якоби f_u
def f_y(y,x,N) :
    f_y = zeros((2*N-1,2*N-1))
    # Определениене ненулевых элементов матрицы Якоби
    f_y[0,0] = (-(2 + (x(2) - x(1))/(x(1) - x(0))))/(x(2) - x(0))
```

```
    f_y[0,1] = ((2 + (x(2) - x(1))/(x(1) - x(0))) + ((x(1) - x(0))/(x(2) - x(1))))/(x(2) - x(0))
    f_y[0,2] =(-((x(1) - x(0))/(x(2) - x(1))))/(x(2) - x(0))
    for n in range(1,N) :
        f_y[n,n-1] = -1/(2*(x(n+1/2) - x(n-1/2))*(x(n-1/4) - x(n-3/4)))
        f_y[n,n] = -1/(2*(x(n+1/2) - x(n-1/2)))*(-1/(x(n+3/4) - x(n+1/4)) - 1/(x(n-1/4) - x(n-3/4)))
    for n in range(1,N-1) :
        f_y[n,n+1] = -1/(2*(x(n+1/2) - x(n-1/2))*(x(n+3/4) - x(n+1/4)))
    for n in range(N,2*N-1) :
        f_y[n,n-N+1] = 2*y[n-N+1]
        f_y[n,n] = -1.
    return f_y

# Определение входных данных задачи
t_0 = 0.; T = 4.5
lambd = 10.

# 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 = 100; M = 500

# Определение равномерной 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 + 1,2*N - 1))

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

# Задание начального условия решаемой системы ОДУ
for n in range(N) :
    y[0,n] = u_init(x(n))
for n in range(1,N) :
    y[0,N-1+n] = u_init(x(n))**2

# Реализация схемы из семейства ROS1
# (конкретная схема определяется коэффициентом alpha)
for m in range(M) :
    w_1 = linalg.solve(D(x,N) - alpha*(t[m+1] - t[m])*f_y(y[m],x,N),f(y[m],x,N,(t[m] + t[m+1])/2,mu,u_right))
    y[m + 1] = y[m] + (t[m+1] - t[m])*w_1.real
    u[m + 1,0:N] = y[m + 1,0:N]
    u[m + 1,N] = u_right(t[m+1])

# Анимация отрисовки решения
style.use('dark_background')
fig = figure()
camera = Camera(fig)
ax = axes(xlim=(0,6), ylim=(-0.3,1.1))
ax.set_xlabel('x'); ax.set_ylabel('u')
# Отрисовываем только каждый 1-й кадр
for m in range(0,M + 1,1) :
    # Отрисовка решения в момент времени t_m
    ax.plot([x(n) for n in range(N)],u[m,0:N], color='y', ls='-', lw=2)
    camera.snap()
animation = camera.animate(interval=15, repeat=False, blit=True)
```

```
# Листинг программы, реализущей решение нелинейного уравнения
# псевдопараболического типа, описывающего нелинейные явления в полупроводниках
```