```
#! python3.7
# -*- coding: utf-8 -*-
from numpy import zeros, linspace, sin, pi, complex64
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 = -100*sin(pi*x)**100 + 100*x**2
    return u_init

# Функция f подготавливает массив, содержащий элементы вектор-функции,
# определяющей правую часть решаемой системы ОДУ
def f(y,h,N):
    f = zeros(N-1)
    f[0] = -1/2*y[0] - h/8*y[0]**2
    f[1] = -(y[1] - 7/4*y[0]) - h/2*y[0]*(y[1] - 1/4*y[0])
    for n in range(2,N-1) :
        f[n] = -(y[n] - 2*y[n-1] + y[n-2]) - h/2*y[n-1]*(y[n] - y[n-2])
    return f

# Функция подготавливает массивы, которые содержат
# элементы диагоналей трёхдиагональной матрицы
# [E - alpha*tau*f_y]
def DiagonalsPreparation(y,h,N,tau,alpha) :
    # Входные данные:
    # y - решение системы ОДУ в текущий момент времени
    # h - шаг сетки
    # N - число интервалов сетки
    # tau - текущий шаг по времени
    # alpha - коэффициент, определяющий численную схему

    # Выходные параметры:
    # a, b и c - диагонали трёхдиагональной матрицы:
    #
    #   [ a(0)                              ]
    #   [ b(1)   a(1)                       ]
    #   [ c(2)   b(2)   a(2)                ]
    #   [        ...    ...    ...          ]
    #   [               ...    ...    ...   ]
    #   [                  c(N-2) b(N-2) a(N-2) ]

    # Выделение памяти под массивы,
    # содержащие соответствующие диагонали
    a = zeros(N-1,dtype=complex64)
    b = zeros(N-1,dtype=complex64)
    c = zeros(N-1,dtype=complex64)

    a[0] = 1/4*(2 - h**2) - alpha*tau*(-1/2 - h/4*y[0])
    a[1] = 1. - alpha*tau*(-1 - h/2*y[0])
    b[1] = -(7/4 + h**2) - alpha*tau*(7/4 - h/2*(y[1] - 1/2*y[0]))
    for n in range(2,N-1) :
        a[n] = 1. - alpha*tau*(-1 - h/2*y[n-1])
        b[n] = -(2 + h**2) - alpha*tau*(2 - h/2*(y[n] - y[n-2]))
        c[n] = 1.- alpha*tau*(-1 + h/2*y[n-1])

    return a, b, c

# Функция реализует экономичный алгоритм
# решения СЛАУ A X = B с трёхдиагональной матрицей
def SpecialMatrixAlgorithm(a,b,c,B) :
    # Входные параметры:
    # B - вектор правой части длины n
    # a, b, c - вектора длины n, содержащие элементы  диагоналей
    # (b(0), c(0) и c(1) в алгоритме не используются)
```

```

    # Структура решаемой СЛАУ:

    # [ a(0)                          ] [ X(0) ]   [ B(0) ]
    # [ b(1) a(1)                     ] [ X(1) ]   [ B(1) ]
    # [ c(2) b(2) a(2)                ] [      ]   [      ]
    # [      ...  ...  ...            ] [ ...  ] = [ ...  ]
    # [           ...  ...  ...       ] [X(n-2)]   [B(n-2)]
    # [            c(n-1) b(n-1) a(n-1) ] [X(n-1)]   [B(n-1)]

    n = len(B); X = zeros(n,dtype=complex64)
    B = B.astype(complex64)

    for i in range(n-2) :
        k = b[i+1]/a[i]
        B[i+1] = B[i+1] - k*B[i]
        k = c[i+2]/a[i]
        B[i+2] = B[i+2] - k*B[i]
    k = b[n-1]/a[n-2]
    B[n-1] = B[n-1] - k*B[n-2]

    for i in range(n) :
        X[i] = B[i]/a[i]

    return X

# Функция находит приближённое решение уравнения в частных производных (УрЧП/PDE)
def PDESolving(a,b,N,t_0,T,M,u_init,alpha) :
    # Входные параметры:
    # a, b - границы области по пространственно переменной x
    # N_0 - число интервалов базовой сетки по пространству
    # t_0, T - начальный и конечный моменты счёта
    # M_0 - число интервалов базовой сетки по времени
    # u_init - функция, определяющяя начальное условие
    # alpha - коэффициент, определяющий численную схему

    # Выходной параметр:
    # u - массив, содержащий сеточные значения решения УрЧП

    # Формирование сетки :
    # Определение сетки по пространству
    h = (b - a)/N; x = linspace(a,b,N+1)
    # Определение сетки по времени
    tau = (T - t_0)/M; t = linspace(t_0,T,M+1)

    # Выделение памяти под массив сеточных значений решения УрЧП,
    u = zeros((M + 1,N + 1))
    # Выделение памяти под вспомогательный массив y,
    # в котором хранятся решения системы ОДУ в текущий момент времени t = t_m
    y = zeros(N - 1)

    # Задание начального условия (на начальном временном слое)
    u[0] = u_init(x)

    # Задание начального условия решаемой системы ОДУ
    y = u_init(x[2:N+1])

    # Реализация схемы из семейства ROS1
    # (конкретная схема определяется коэффициентом alpha)
    for m in range(M) :
        diagonal,codiagonal_1,codiagonal_2 = DiagonalsPreparation(y,h,N,tau,alpha)
        w_1 =  SpecialMatrixAlgorithm(diagonal,codiagonal_1,codiagonal_2,f(y,h,N))
        y = y + tau*w_1.real

        u[m + 1,0] = 0
        u[m + 1,1] = 1/4*y[0]
        u[m + 1,2:N+1] = y

    return u
```

```python

# Определение входных данных задачи
a = 0.; b = 1.
t_0 = 0.; T = 0.8

# 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 = 400

u = PDESolving(a,b,N,t_0,T,M,u_init,alpha)
# Анимация отрисовки решения
style.use('dark_background')
fig = figure()
camera = Camera(fig)
ax = axes(xlim=(a,b), ylim=(-130.,130.))
ax.set_xlabel('x'); ax.set_ylabel('u')
for m in range(M + 1) :
    # Отрисовка решения в момент времени t_m
    ax.plot(linspace(a,b,N+1),u[m], color='y', ls='-', lw=2)
    camera.snap()
animation = camera.animate(interval=20, repeat=False, blit=True)

# Листинг программы, реализущей решение нелинейного уравнения
# типа Бенджамена-Бона-Махони-Бюргерса методом прямых
```