```python
#! python3.7
# -*- coding: utf-8 -*-
from numpy import zeros, linspace, linalg, tanh, eye, complex64, log, sqrt, sum, NaN, inf
from matplotlib.pyplot import style, figure, axes

# 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

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

# Функция подготавливает массивы, которые содержат
# элементы диагоналей трёхдиагональной матрицы
# [E - alpha*tau*f_y]
def DiagonalsPreparation(y,t,h,N,u_left,u_right,eps,tau,alpha) :
    # Входные данные:
    # y - решение системы ОДУ в текущий момент времени
    # t - текущий момент времени t_m
    # h - шаг сетки
    # N - число интервалов сетки
    # u_left и u_right - функции, определяющие левое и правое граничные условия
    # eps - парамтр задачи
    # tau - текущий шаг по времени
    # alpha - коэффициент, определяющий численную схему

    # Выходные параметры:
    # a, b и c - диагонали трёхдиагональной матрицы
    #
    #  [ a(0)   c(0)                          ]
    #  [ b(1)   a(1)   c(1)                   ]
    #  [        b(2)   a(2)   c(2)            ]
    #  [               ...    ...    ...      ]
    #  [                      ...    ...  c(N-1) ]
    #  [                          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. - alpha*tau*(-2*eps/h**2 + (y[1] - u_left(t))/(2*h) + 3*y[0]**2)
    c[0] = - alpha*tau*(eps/h**2 + y[0]/(2*h))
    for n in range(1,N-2) :
        b[n] = - alpha*tau*(eps/h**2 - y[n]/(2*h))
        a[n] = 1. - alpha*tau*(-2*eps/h**2 + (y[n+1] - y[n-1])/(2*h) + 3*y[n]**2)
```

```python
            c[n] = - alpha*tau*(eps/h**2 + y[n]/(2*h))
        b[N-2] = - alpha*tau*(eps/h**2 - y[N-2]/(2*h))
        a[N-2] = 1. - alpha*tau*(-2*eps/h**2 + (u_right(t) - y[N-3])/(2*h) + 3*y[N-2]**2)

        return a, b, c

    def TridiagonalMatrixAlgorithm(a,b,c,B) :
        # Функция реализует метод прогонки (алгоритм Томаса)
        # для решения СЛАУ A X = B с трёхдиагональной матрицей

        # Входные параметры:
        # B - вектор правой части длины n (столбец или строка)
        # a, b, c - вектора длины n, содержащие элементы
        # диагоналей (b(1) и c(n) не используются)

        # [ a(1) c(1)                            ] [ X(1) ]    [ B(1) ]
        # [ b(2) a(2) c(2)                       ] [ X(2) ]    [ B(2) ]
        # [      b(3) a(3) c(3)                  ] [      ]    [      ]
        # [           ...  ...  ...              ] [ ...  ] =  [ ...  ]
        # [                ...  ...  c(n-1)      ] [X(n-1)]    [B(n-1)]
        # [                     b(n)   a(n)      ] [ X(n) ]    [ B(n) ]

        n = len(B)
        v = zeros(n,dtype=complex64)
        X = zeros(n,dtype=complex64)

        w = a[0]
        X[0] = B[0]/w
        for i in range(1,n) :
            v[i - 1] = c[i - 1]/w
            w = a[i] - b[i]*v[i - 1]
            X[i] = (B[i] - b[i]*X[i - 1])/w
        for j in range(n-2,-1,-1) :
            X[j] = X[j] - v[j]*X[j + 1]

        return X

    # Функция находит приближённое решение уравнения в частных производных (УрЧП/PDE)
    def PDESolving(a,b,N_0,t_0,T,M_0,u_init,u_left,u_right,eps,s,r_x,r_t,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
        # u_left и u_right - функции, определяющие левое и правое граничные условия
        # eps - парамтр задачи
        # s - номер сетки, на которой вычисляется решение
        # (если s = 0, то решение ищется на БАЗОВОЙ сетке)
        # r_x и r_t - коэффициенты сгущения сетки по x и t
        # alpha - коэффициент, определяющий численную схему

        # Выходной параметр:
        # u_basic - массив, содержащий сеточные значения решения УрЧП
        # только в узлах, совпадающих с узлами БАЗОВОЙ сетки

        # Формирование сгущённой
        # в r_x^s раз по пространственной переменной x и
        # в r_t^s раз по временной переменной t
        # сетки с индексом s:

        # Вычисление числа интервалов на сетке с номером s
        N = N_0*r_x**s; M = M_0*r_t**s
        # 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,
```

```
    # в котором будут храниться сеточные значения из узлов,
    # совпадающих с узлами БАЗОВОЙ пространственно-временной сетки
    u_basic = zeros((M_0 + 1,N_0 + 1))
    # Выделение памяти под вспомогательный массив y,
    # в котором хранятся решения системы ОДУ в текущий момент времени t = t_m
    # (система решается на сетке с N = N_0*r_x**s интервалами по пространству)
    y = zeros(N - 1)

    # Задание начального условия (на начальном временном слое)
    for n in range(N_0+1) :
        u_basic[0,n] = u_init(x[n*r_x**s])

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

    # Введение индекса, отвечающего за выбор
    # временного слоя на сетке с номером s,
    # совпадающего с соответствующим временным слоем базовой сетки.
    # На данный момент будем отслеживать совпадение t_m на сгущённой сетке
    # с t_m_basic на базовой сетке
    m_basic = 1

    # Реализация схемы из семейства ROS1
    # (конкретная схема определяется коэффициентом alpha)
    for m in range(M) :
        print('s={0}, m={1}'.format(s,m))
        diagonal,codiagonal_down,codiagonal_up = DiagonalsPreparation(y,t[m],h,N,u_left,u_right,eps,tau,alpha)
        w_1 = TridiagonalMatrixAlgorithm(diagonal,codiagonal_down,codiagonal_up,f(y,t[m] + tau/2,h,N,u_left,u_right,eps))
        # Переопределение y в новый момент времени t_{m+1}
        y = y + tau*w_1.real

        # Выполение проверки совпадения t_{m+1}
        # на сгущеённой сетке с t_m_basic базовой сетки
        if (m + 1) == m_basic*r_t**s  :
            # Заполнение массива сеточных значений решения
            # исходной задачи для УрЧП
            u_basic[m_basic,0] = u_left(t[m+1])
            for n in range(1,N_0) :
                u_basic[m_basic,n] = y[n*r_x**s - 1]
            u_basic[m_basic,N_0] = u_right(t[m+1])
            # Теперь будет отслеживаться совпадение t_{m+1}
            # на сгущённое сетке с очередным t_m_basic
            m_basic = m_basic + 1

    return u_basic

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

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

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

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

# Число сеток, на которых ищется приближённое решение
S = 3
# Коэффициенты сгущения пространственно-временной сетки
r_x = 2; r_t = 2
# Теоретические параметры схемы
```

```python
p_x = 2; p_t = 2; q = 1

# Выделение памяти под массивы сеточных значений
# решений ОДУ на разных сетках с номерами s = 0,...,S-1,
# в которых хранятся сеточные значения решения из узлов,
# совпадающих с узлами базовой сетки
U = zeros((S,S,M + 1,N + 1))

# "Большой цикл", который пересчитывает решение S раз
# на последовательности сгущающихся сеток
# Массив сеточных значений решения содержит только
# сеточные значения из узлов, совпадающих с узлами базовой сетки
for s in range(S) :
    U[s,0,:,:]  = PDESolving(a,b,N,t_0,T,M,u_init,u_left,u_right,eps,s,r_x,r_t,alpha)

# Выделение памяти под массивы ошибок R,
# относительных ошибок R_rel и эффективных порядков точности p_eff
R = zeros((S,S,M + 1,N + 1))
R_rel = zeros((S,S))
p_eff = zeros((S,S))

for s in range(1,S) :
    for l in range(s) :
        R[s,l,:,:] = (U[s,l,:,:] - U[s-1,l,:,:])/(r_t**(p_t + l*q) - 1)
        U[s,l+1,:,:] = U[s,l,:,:] + R[s,l,:,:]
        R_rel[s,l] = sqrt(sum(R[s,l,:,:]**2))/sqrt(sum(U[s,l+1,:,:]**2))*100

for s in range(2,S) :
    for l in range(s-1) :
        p_eff[s,l] = log(sqrt(sum(R[s-1,l,:,:]**2))/sqrt(sum(R[s,l,:,:]**2)))/log(r_t)

# Функция выводит форматированную таблицу
def PrintTriangular(A,i) :
    print('      ',end=' ')
    for l in range(len(A)) :
        print(' p={0:<4d}'.format(p_t + l*q),end=' ')
    print()
    for m in range(len(A)) :
        print('s={0:<2d}'.format(m),end=' ')
        for l in range(m + 1 - i) :
            print('{0:5.2f}'.format(A[m,l]),end=' ')
        print()
    print()

print('Таблица оценок относительных ошибок (в процентах):')
PrintTriangular(R_rel,1)
print('Таблица эффективных порядков точности:')
PrintTriangular(p_eff,2)

# Выделение памяти под массив значений эффективных
# порядков точности расчёта приближённого решения
# в каждом узле t_m, 1 <= m <= M (второй индекс массива),
# кроме t_0, так как в нём решение задано точно,
# и на разных сетках (первый индекс массива)
p_eff_ForEveryTime = zeros((S,M + 1));

# Вычисление эффективных порядков точности
for m  in range(1,M+1) :
    # Вычисление p^{eff}_{(0)}(t_m) и p^{eff}_{(1)}(t_m) невозможно
    p_eff_ForEveryTime[0,m] = NaN
    p_eff_ForEveryTime[1,m] = NaN
    for s in range(2,S) :
        p_eff_ForEveryTime[s,0] = inf
        p_eff_ForEveryTime[s,m] = log(sqrt(sum(R[s-1,0,m,:]**2))/sqrt(sum(R[s,0,m,:]**2)))/log(r_t)

# Отрисовка результатов расчётов для сетки с номером S-1
style.use('dark_background')
```

```
fig = figure()
ax = axes(xlim=(t_0,T), ylim=(-2.,3.))
ax.set_xlabel('t'); ax.set_ylabel('$p^{eff}$')
# Рисуется зависимость теоретического порядка точности p_theor от узла базовой сетки t_m
t = linspace(t_0,T,M+1)
ax.plot(t,t*0 + p_t,color='g', ls='--', lw=2)
# Рисуется зависимость эффективного порядка точности от узла базовой сетки
ax.plot(t[1:M+1],p_eff_ForEveryTime[S-1,1:M + 1],color='y', ls='-', lw=3)

# Листинг программы, реализущей решение нелинейного уравнения
# типа Бюргерса методом прямых с контролем точности по по Ричардсону
# (с вычислением эффективных порядков точности)
```