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

# 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
# [D - 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 тирёхдиагональной 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_0,t_0,T,M_0,u_init,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
    # 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[2:N+1])

    # 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_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

        # 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] = 0
            if s == 0 :
                u_basic[m_basic,1] = 1/4*y[0]
            else :
                u_basic[m_basic,1] = y[r_x**s-2]
            for n in range(2,N_0+1) :
                u_basic[m_basic,n] = y[n*r_x**s - 2]
            # 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 = 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 БАЗОВОЙ пространственно-временной сетки,
# на которой будет искаться приближённое решение
N = 50; M = 50

# Число сеток, на которых ищется приближённое решение
S = 4
# Коэффициенты сгущения пространственно-временной сетки
r_x = 2; r_t = 2
# Теоретические параметры схемы
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,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,M + 1,N + 1))
R_rel = zeros(S)
p_eff = zeros(S)
```

```python
for s in range(1,S) :
    R[s,:,:] = (U[s,0,:,:] - U[s-1,0,:,:])/(r_t**p_t - 1)
    U[s,1,:,:] = U[s,0,:,:] + R[s,:,:]
    R_rel[s] = sqrt(sum(R[s,:,:]**2))/sqrt(sum(U[s,:,:]**2))*100

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

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

print('Таблица оценок относительных ошибок (в процентах):')
PrintResults(R_rel)
print('Таблица эффективных порядков точности:')
PrintResults(p_eff)

# 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,m,:]**2))/sqrt(sum(R[s,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 Бенджамена-Бона-Махони-Бюргерса методом прямых с контролем точности по Ричардсону
# (с вычислением эффективных порядков точности)
```