```python
#! python3.7
# -*- coding: utf-8 -*-
from numpy import zeros, linspace, sin, pi, complex64, log, sqrt, sum

# Определение функции, задающей начальное условие
def u_init(x) :
    if x >= 0 and x <= pi :
        u_init = sin(x)
    else:
        u_init = 0.
    return u_init

# Определение функции, задающей левое граничное условие
def u_left(t) :
    u_left = 0.
    return u_left

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

# Функция подготавливает массивы, которые содержат
# элементы диагоналей двухдиагональной 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 - диагонали двухиагональной матрицы:
    #
    #  [ a(0)                                ]
    #  [ b(1)  a(1)                          ]
    #  [       b(2)  a(2)                    ]
    #  [             ...   ...               ]
    #  [                    ...   ...        ]
    #  [                       b(N-1) a(N-1) ]

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

    a[0] = 1. - alpha*tau*(-1/h)
    for n in range(1,N) :
        a[n] = 1. - alpha*tau*(-1/h)
        b[n] = - alpha*tau*(1/h)

    return a, b

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

    # Структура решаемой СЛАУ:

```

```
    # [ a(0)                          ] [ X(0) ]   [ B(0) ]
    # [ b(1) a(1)                     ] [ X(1) ]   [ B(1) ]
    # [      b(2) a(2)                ] [      ]   [      ]
    # [           ...  ...            ] [ ...  ] = [ ...  ]
    # [                ...  ...       ] [X(n-2)]   [B(n-2)]
    # [                  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-1) :
        k = b[i+1]/a[i]
        B[i+1] = B[i+1] - k*B[i]

    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)

    # 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
    for i in range(N) :
        y[i] = u_init(x[i+1])

    # Введение индекса, отвечающего за выбор
    # временного слоя на сетке с номером s,
    # совпадающего с соответствующим временным слоем базовой сетки.
```

```
139         # На данный момент будем отслеживать совпадение t_m на сгущённой сетке
140         # с t_m_basic на базовой сетке
141         m_basic = 1
142 
143         # Реализация схемы из семейства ROS1
144         # (конкретная схема определяется коэффициентом alpha)
145         for m in range(M) :
146             print('s={0}, m={1}'.format(s,m))
147             diagonal,codiagonal = DiagonalsPreparation(y,h,N,tau,alpha)
148             w_1 =  SpecialMatrixAlgorithm(diagonal,codiagonal,f(y,h,N,u_left,t[m] +
            tau/2))
149             y = y + tau*w_1.real
150 
151             # Выполение проверки совпадения t_{m+1}
152             # на сгущеённой сетке с t_m_basic базовой сетки
153             if (m + 1) == m_basic*r_t**s  :
154                 # Заполнение массива сеточных значений решения
155                 # исходной задачи для УрЧП
156                 u_basic[m_basic,0] = u_left(t[m+1])
157                 for n in range(1,N_0+1) :
158                     u_basic[m_basic,n] = y[n*r_x**s - 1]
159                 # Теперь будет отслеживаться совпадение t_{m+1}
160                 # на сгущённое сетке с очередным t_m_basic
161                 m_basic = m_basic + 1
162 
163     return u_basic
164 
165 # Определение входных данных задачи
166 a = 0.; b = 10.
167 t_0 = 0.; T = 6.
168 
169 # Определение параметра схемы (нужный раскомментировать)
170 alpha = (1 + 1j)/2  # CROS1 (схема Розенброка с комплексным коэффициентом)
171 # alpha = 1.          # DIRK1 (обратная схема Эйлера
172 
173 # Определение числа интервалов БАЗОВОЙ пространственно-временной сетки,
174 # на которой будет искаться приближённое решение
175 N = 50; M = 50
176 
177 # Число сеток, на которых ищется приближённое решение
178 S = 5
179 # Коэффициенты сгущения пространственно-временной сетки
180 r_x = 4; r_t = 2
181 # Теоретические параметры схемы
182 p_x = 1; p_t = 2; q = 1
183 
184 # Выделение памяти под массивы сеточных значений
185 # решений ОДУ на разных сетках с номерами s = 0,...,S-1,
186 # в которых хранятся сеточные значения решения из узлов,
187 # совпадающих с узлами базовой сетки
188 U = zeros((S,S,M + 1,N + 1))
189 
190 # "Большой цикл", который пересчитывает решение S раз
191 # на последовательности сгущающихся сеток
192 # Массив сеточных значений решения содержит только
193 # сеточные значения из узлов, совпадающих с узлами базовой сетки
194 for s in range(S) :
195     U[s,0,:,:]  = PDESolving(a,b,N,t_0,T,M,u_init,s,r_x,r_t,alpha)
196 
197 # Выделение памяти под массивы ошибок R,
198 # относительных ошибок R_rel и эффективных порядков точности p_eff
199 R = zeros((S,M + 1,N + 1))
200 R_rel = zeros(S)
201 p_eff = zeros(S)
202 
203 for s in range(1,S) :
204     R[s,:,:] = (U[s,0,:,:] - U[s-1,0,:,:])/(r_t**p_t - 1)
205     U[s,1,:,:] = U[s,0,:,:] + R[s,:,:]
206     R_rel[s] = sqrt(sum(R[s,:,:]**2))/sqrt(sum(U[s,:,:]**2))*100
```

```
207
208 for s in range(2,S) :
209     p_eff[s] = log(sqrt(sum(R[s-1,:,:]**2))/sqrt(sum(R[s,:,:]**2)))/log(r_t)
210
211 # Функция выводит форматированную таблицу эффективных порядков точности
212 print('Таблица эффективных порядков точности:')
213 print('     ',end=' ')
214 print('p_eff')
215 print()
216 for s in range(S) :
217     print('s={0:<2d}'.format(s),end=' ')
218     print('{0:5.2f}'.format(p_eff[s]),end=' ')
219     print()
220 print()
221
222 # Листинг программы, реализущей численное исследование порядка точности схемы
223 # на примере решения линейного уравнения переноса
```