Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Информационные технологии и прикладная математика»

Лабораторные работы по дисциплине «Численные методы»

Студент: Янтиков К.А.

Группа: М80-402Б-20

Преподаватель: Пивоваров Д.Е.

Москва, 2023

Лабораторная работа №1

1. Задание

Используя явную и неявную конечно-разностные схемы, а также схему Кранка - Николсона, решить начально-краевую задачу для дифференциального уравнения параболического типа. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 8:

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + cu,\ a > 0,\ c < 0.$$

$$u_x(0,t) = \exp((c-a)t),$$

$$u(\frac{\pi}{2},t) = \exp((c-a)t),$$

$$u(x,0) = \sin x,$$

Аналитическое решение: $U(x,t) = \exp((c-a)t)\sin x.$

2. Теория

Конечно-разностная схема

Будем решать задачу на заданном промежутке от 0 до l по координате x и на промежутке от 0 до заданного параметра T по времени t.

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными парамерами l, T и параметрами насыщенности сетки N, K. Тогда размер шага по каждой из координат определяется:

$$h = \frac{l}{N},\ \tau = \frac{T}{K}$$

Считая, что значения функции $u_j^k = u(x_j, t^k)$ для всех координат $x_j = jh,\ \forall j \in \{0, ..., N\}$ на временном слое $t^k = k\tau,\ k \in \{0, ..., K-1\}$ известно, попробуем определить значения функции на временном слое t^{k+1} путем разностной апроксимации производной:

$$\frac{\partial u}{\partial t}(x_j, t^k) = \frac{u_j^{k+1} - u_j^k}{\tau}$$

И одним из методов апроксимации второй производной по x:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k)$$

Явная конечно-разностная схема

Апроксимируем вторую производную по значениям нижнего временного слоя t^k, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = a\frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2},\ \forall j \in \{1, ..., N-1\}, \forall k \in \{0, ..., K-1\}$$

Обозначим $\sigma = \frac{a\tau}{h^2}$, тогда:

$$u_j^{k+1} = \sigma u_{j-1}^k + (1 - 2\sigma)u_j^k + \sigma u_{j+1}^k$$

Граничные же значения u_0^{k+1} и u_N^{k+1} определяются граничными условиями $u_x(0, t) = \phi_0(t)$ и $u_x(l, t) = \phi_l(t)$ при помощи апроксимации производной.

Неявная конечно-разностная схема

Апроксимируем вторую производную по значениям верхнего временного слоя t^{k+1}, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = a\frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2},\ \forall j \in \{1, ..., N-1\}, \forall k \in \{0, ..., K-1\}$$

Обозначим $\sigma = \frac{a\tau}{h^2}$. Тогда значения функции на слое можно найти эффективны образом с помощью методом прогонки, где **СЛАУ**, кроме крайних двух уравнений, определяется коэффицетнами $a_j = \sigma$, $b_j = -(1 + 2\sigma)$, $c_j = \sigma$, $d_j = -u_j^k$ уравнений:

$$a_j u_{j-1}^{k+1} + b_j u_j^{k+1} + c_j u_{j+1}^{k+1} = d_j,\ \forall j \in \{1, ..., N-1\}$$

Первое и последнее уравнение системы содержащие u_0^{k+1} и u_N^{k+1} определяются граничными условиями $u_x(0, t) = \phi_0(t)$ и $u_x(l, t) = \phi_l(t)$ при помощи апроксимации производной.

Неявная схема является абсолютно устойчивой.

Схема Кранка-Николсона

Поскольку как правило решение в зависимости от времени лежит между значениями явной и неявной схемы, имеет смысл получить смешанную апроксимацию пространственных производных.

Явно-неявная схема для $\forall j \in \{1, ..., N-1\}, \forall k \in \{0, ..., K-1\}$ будет выглядеть следующим образом:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = \theta a\frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2} + (1 - \theta)a\frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2}$$

При значении параметра $\theta = \frac{1}{2}$ схема являет собой *схему Кранка-Николсона.*

Обозначим $\sigma = \frac{a\tau}{h^2}$. Тогда значения функции на слое можно найти эффективны образом с помощью методом прогонки, где **СЛАУ**, кроме крайних двух уравнений, определяется коэффицетнами $a_j = \sigma\theta$, $b_j = -(1 + 2\theta\sigma)$, $c_j = \sigma\theta$, $d_j = -(u_j^k + (1 - \theta)\sigma(u_{j-1}^k - 2u_j^k + u_{j+1}^k))$ уравнений:

$$a_j u_{j-1}^{k+1} + b_j u_j^{k+1} + c_j u_{j+1}^{k+1} = d_j,\ \forall j \in \{1, ..., N-1\}$$

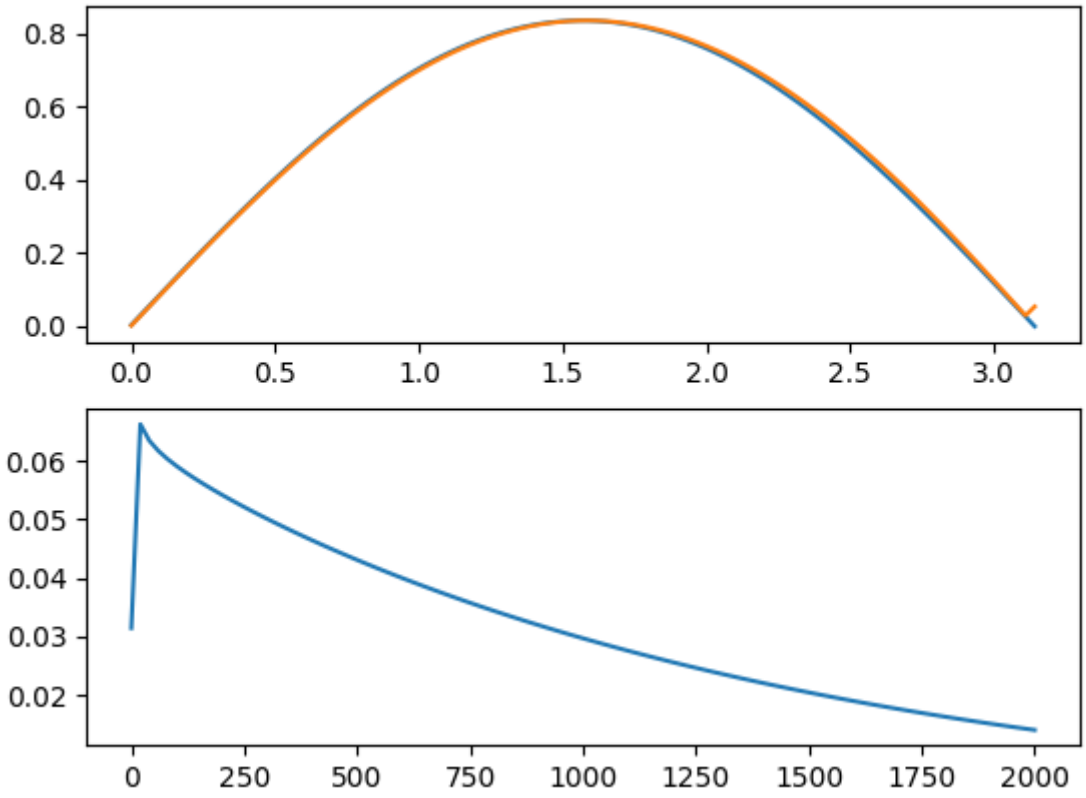
Первое и последнее уравнение системы содержащие u_0^{k+1} и u_N^{k+1} определяются граничными условиями $u_x(0, t) = \phi_0(t)$ и $u_x(l, t) = \phi_l(t)$ при помощи апроксимации производной.

Схема Кранка-Николсона является абсолютно устойчивой.

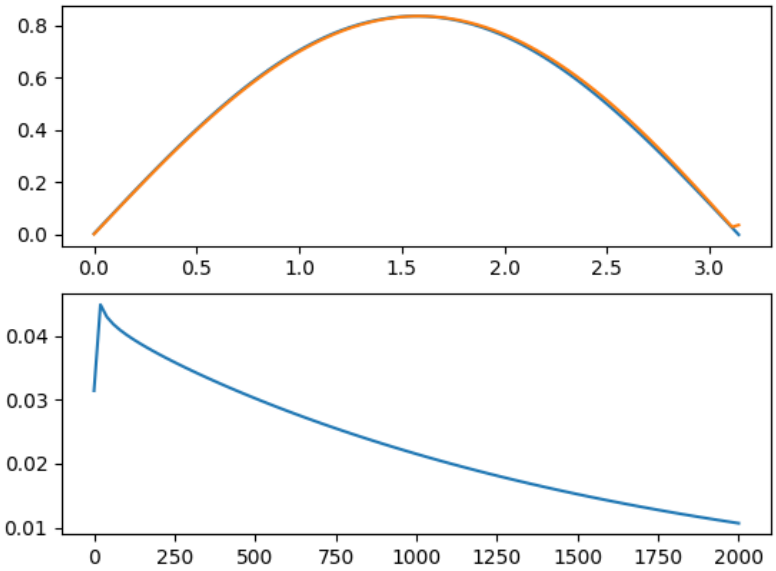
3. Результат

Явный метод:

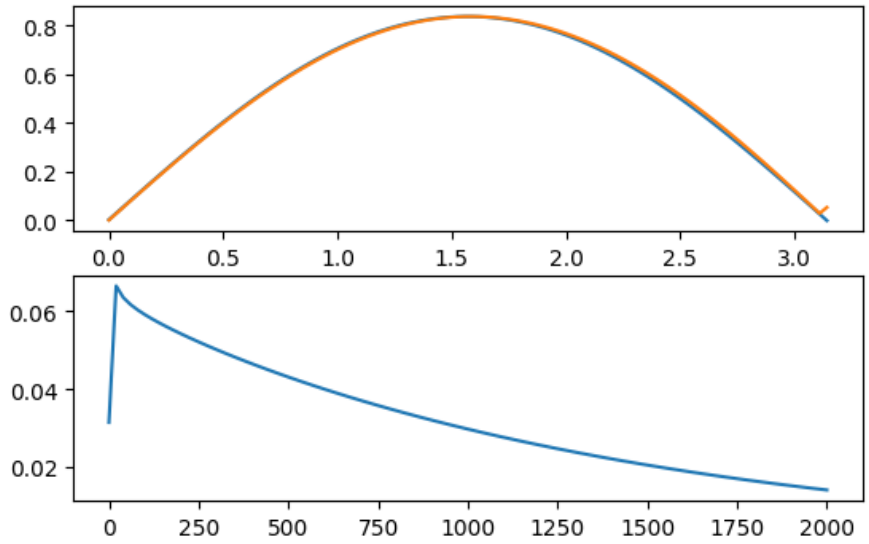
1) График численного решения явной конечно-разностной схемой с двухточечной с первым порядком апроксимацией.



2) График численного решения явной конечно-разностной схемой с трехточечной с вторым порядком апроксимацией.

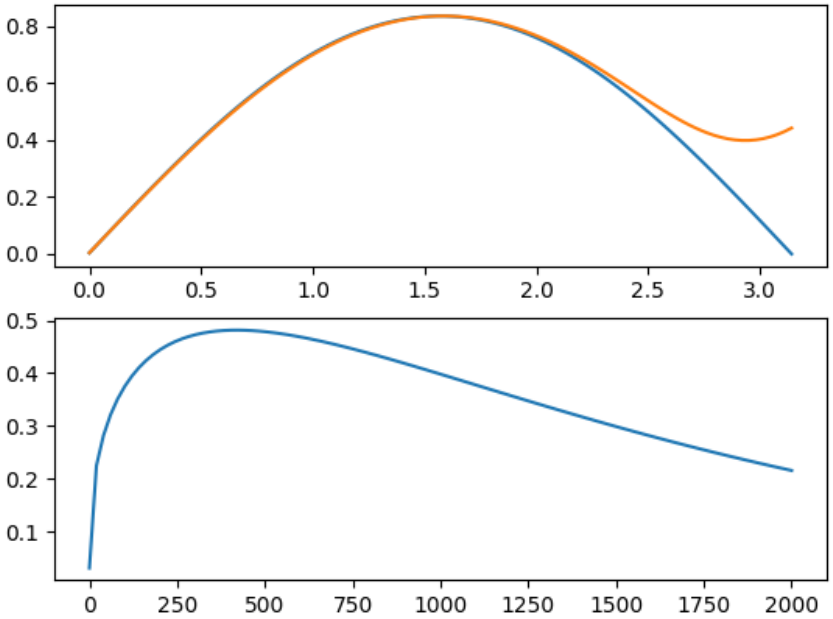


3) График численного решения явной конечно-разностной схемой с двухточечной с вторым порядком апроксимацией.

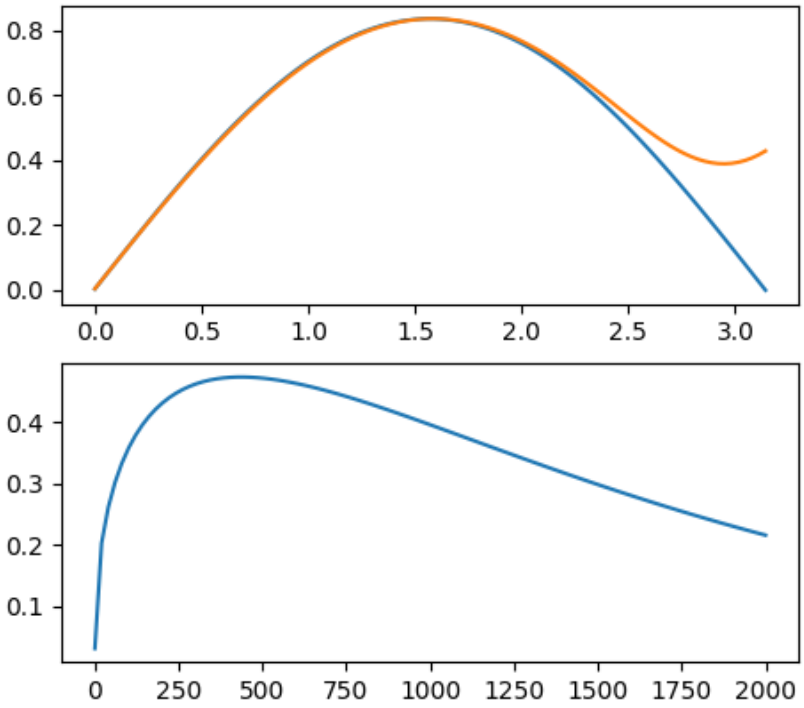


Неявный метод:

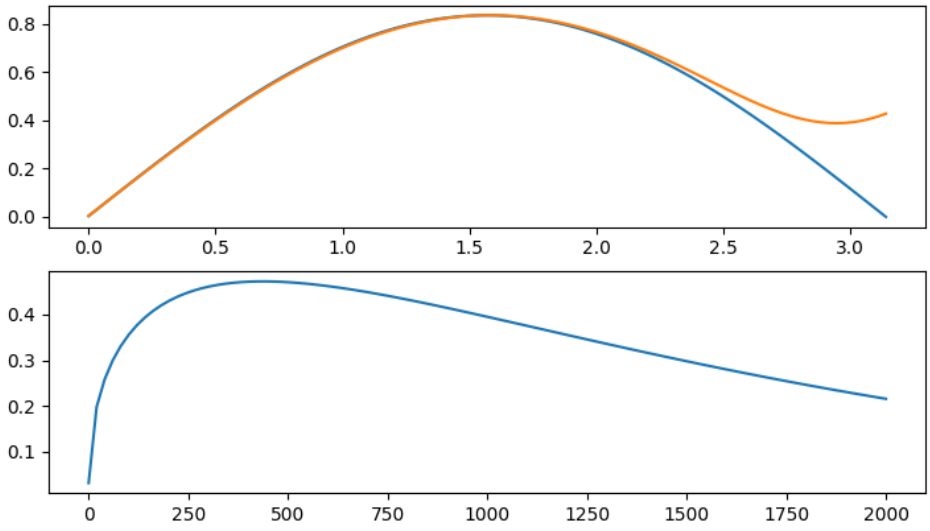
1) График численного решения неявной конечно-разностной схемой с двухточечной с первым порядком апроксимацией.



2) График численного решения неявной конечно-разностной схемой с трехточечной с вторым порядком апроксимацией.

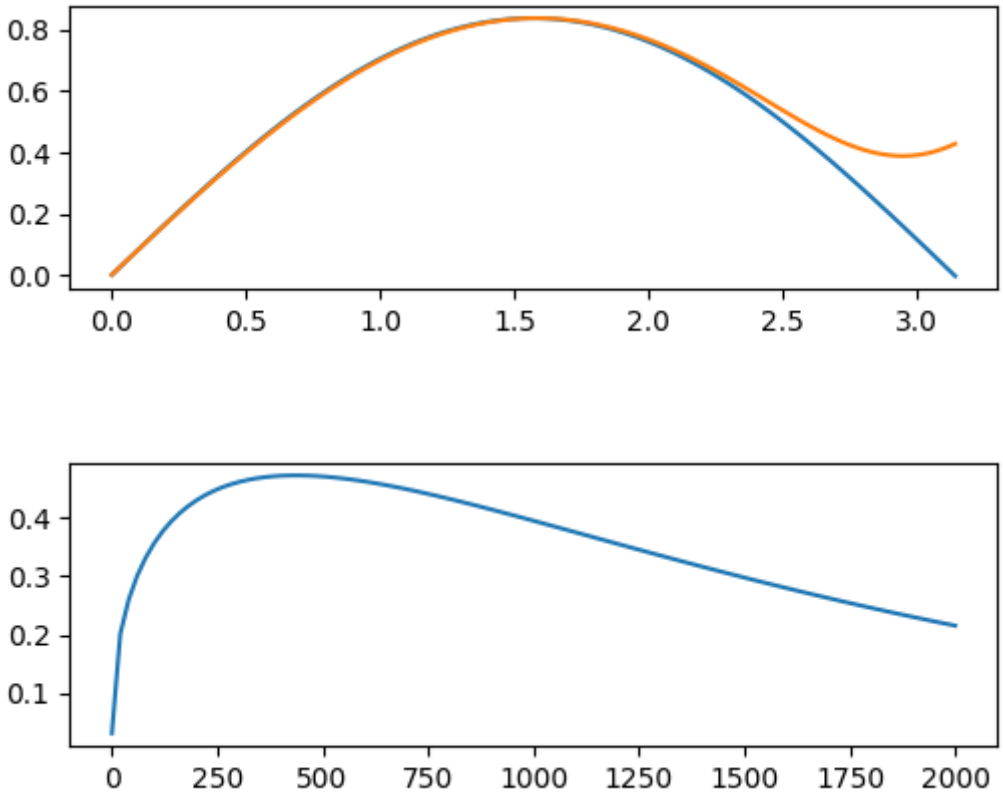


3) График численного решения неявной конечно-разностной схемой с двухточечной с вторым порядком апроксимацией.

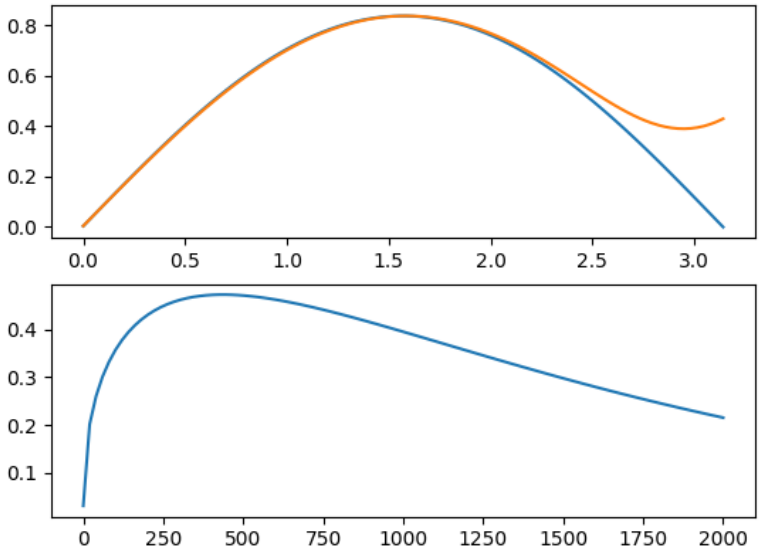


Метод Кранка-Никольсона:

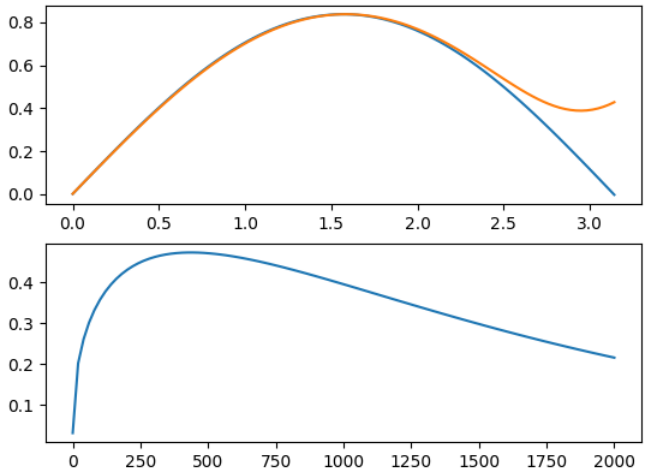
1) График численного решения методом Кранка-Никольсона с двухточечной с первым порядком апроксимацией.



2) График численного решения методом Кранка-Никольсона с трехточечной с вторым порядком апроксимацией.



3) График численного решения методом Кранка-Никольсона с двухточечной с вторым порядком апроксимацией.



4. Вывод

Выполнив данную лабораторную работу, я изучил явные и неявные конечно-разностные схемы, схему Кранка-Николсона для решения начально-краевой задачи для дифференциального уравнения параболического типа. Реализовал три варианта аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком.

5. Код

```
import numpy as np
from matplotlib import pyplot as plt
import tkinter as tk
import tkinter.ttk as ttk

a = 0.5
b = 24
c = - 312
l = np.pi
N = 100
K = 2001

alpha0 = 1
alphal = 1
betta0 = 1
bettal = 1

# Граничные условия
phi0 = lambda t: np.exp((c - a) * t)
phil = lambda t: np.exp((c - a) * t)

# Начальные условия
```

```
gamma0 = lambda x: np.sin(x)

# Аналитическое решение
fResult = lambda x, t: np.exp((c - a) * t) * np.sin(x + b * t)

h = l / N
sigma = 0.4
taw = sigma * h ** 2 / a
mu = b * taw / 2 / h
U = []

def plotSlice(f, X, t):
    plt.subplot(2, 1, 1)
    plt.plot(X, f(X, t))
    plt.grid

def Error(U, fR, count):
    X = np.linspace(0, l, N)
    Y = list(map(int, np.linspace(0, U.shape[0] - 1, count)))
    plt.subplot(2, 1, 2)
    plt.plot(Y, list(map(lambda tt: np.max(np.abs(U[tt] - fR(X, taw*tt))), Y)))

def gridFun(xCond, tCond):
    xlCond,xrCond = xCond
    tlCond, trCond = tCond
    return np.zeros((xrCond, trCond))

# Явная конечно-разностная схема
def explicitMethod(iCondition, bCondition, method="1"):

    def explicitWithFirstDegree():
        # 1 порядка
        for k in range(0, K - 1):
            U[k + 1][0] = (- alpha0 / h * U[k + 1][1] + phi0(taw * (k + 1))) / (betta0 - alpha0 / h)
            U[k + 1][N - 1] = (alphal / h * U[k + 1][N - 2] + phil(taw * (k + 1))) / (bettal + alphal / h)
        pass

    def explicitWithSecondDegree():
        # 2 порядка
        for k in range(0, K - 1):
            U[k + 1][0] = (- alpha0 / h / 2 * (4 * U[k + 1][1] - U[k + 1][2]) + phi0(taw * (k + 1))) / (
            betta0 - 3 * alpha0 / h / 2)
            U[k + 1][N - 1] = (- alphal / h / 2 * (U[k + 1][N - 3] - 4 * U[k + 1][N - 2]) + phil(taw * (k + 1))) / (
            bettal + 3 * alphal / h / 2)

    def explicitWithThirdDegree():
```

```
        b0 = 2*a/h + h/taw - h*c - betta0/alpha0*(2*a - b*h)
        c0 = -2*a/h
        bN = 2*a/h + h/taw - h*c + bettal/alphal*(2*a + b*h)
        aN = -2*a/h
        for k in range(0, K - 1):
            d0 = h/taw * U[k][0] - phi0(taw*(k + 1)) * (2*a - b*h) / alpha0
            dN = h/taw * U[k][N - 1] + phil(taw*(k + 1)) * (2*a + b*h) /
   alphal
            U[k + 1][0] = (d0 - c0 * U[k + 1][1]) / b0
            U[k + 1][N - 1] = (dN - aN * U[k + 1][N - 2]) / bN

    gamma0 = iCondition
    phi0, phil = bCondition
    U = gridFun((0, K), (0, N))
    for j in range(0, N):
        U[0][j] = gamma0(j * h)
    for k in range(0, K - 1):
        for j in range(1, N - 1):
            U[k + 1][j] = (sigma + mu) * U[k][j + 1] + (1 - 2 * sigma + taw
   * c) * U[k][j] + (sigma - mu) * U[k][
                j - 1]
            pass
    if method == "1":
        explicitWithFirstDegree()
    elif method == "2":
        explicitWithSecondDegree()
    elif method == "3":
        explicitWithThirdDegree()
    else:
        pass
    return U

# Неявный метод
def implicitMethod(iCondition, bCondition, method="1"):
    def implicitWithFirstDegree():
        b0 = betta0 - alpha0 / h
        c0 = alpha0 / h
        aN = - alphal / h
        bN = bettal + alphal / h

        def gA():
            aa = np.zeros((N, N))
            aa[0][0] = b0
            aa[0][1] = c0
            for j in range(1, N - 1):
                aa[j][j - 1] = aj
                aa[j][j] = bj
                aa[j][j + 1] = cj
            aa[N - 1][N - 2] = aN
            aa[N - 1][N - 1] = bN
            return aa
```

```
123.
124.          def gB(k):
125.              bb = np.zeros((N, 1))
126.              bb[0][0] = phi0(taw * (k + 1))
127.              bb[N - 1][0] = phil(taw * (k + 1))
128.              for j in range(1, N - 1):
129.                  bb[j][0] = - U[k][j]
130.              return bb
131.          return gA, gB
132.
133.      def implicitWithSecondDegree():
134.          a0 = betta0 - 3*alpha0/ h/ 2
135.          b0 = 2 * alpha0/h
136.          c0 = - alpha0 / h /2
137.          aN = alphal / h /2
138.          bN = -2 * alphal/h
139.          cN = bettal + 3*alphal/ h/ 2
140.
141.          def gA():
142.              aa = np.zeros((N, N))
143.              aa[0][0] = a0
144.              aa[0][1] = b0
145.              aa[0][2] = c0
146.              for j in range(1, N - 1):
147.                  aa[j][j - 1] = aj
148.                  aa[j][j] = bj
149.                  aa[j][j + 1] = cj
150.              aa[N - 1][N - 2] = bN
151.              aa[N - 1][N - 1] = cN
152.              aa[N - 1][N - 3] = aN
153.              return aa
154.
155.          def gB(k):
156.              bb = np.zeros((N, 1))
157.              bb[0][0] = phi0(taw * (k + 1))
158.              bb[N - 1][0] = phil(taw * (k + 1))
159.              for j in range(1, N - 1):
160.                  bb[j][0] = - U[k][j]
161.              return bb
162.          return gA, gB
163.
164.      def implicitWithThirdDegree():
165.          b0 = 2 * a / h + h / taw - h * c - betta0 / alpha0 * (2 * a - b *
     h)
166.          c0 = -2 * a / h
167.          bN = 2 * a / h + h / taw - h * c + bettal / alphal * (2 * a + b *
     h)
168.          aN = -2 * a / h
169.          def gA():
170.              aa = np.zeros((N, N))
171.              aa[0][0] = b0
```

```
172.                 aa[0][1] = c0
173.                 for j in range(1, N - 1):
174.                     aa[j][j - 1] = aj
175.                     aa[j][j] = bj
176.                     aa[j][j + 1] = cj
177.                 aa[N - 1][N - 2] = aN
178.                 aa[N - 1][N - 1] = bN
179.                 return aa
180.
181.             def gB(k):
182.                 bb = np.zeros((N, 1))
183.                 d0 = h / taw * U[k][0] - phi0(taw * (k + 1)) * (2 * a - b *
    h) / alpha0
184.                 dN = h / taw * U[k][N - 1] + phil(taw * (k + 1)) * (2 * a + b
    * h) / alphal
185.                 bb[0][0] = d0
186.                 bb[N - 1][0] = dN
187.                 for j in range(1, N - 1):
188.                     bb[j][0] = - U[k][j]
189.                 return bb
190.
191.             return gA, gB
192.
193.         U = gridFun((0, K), (0, N))
194.         for j in range(0, N):
195.             U[0][j] = gamma0(j * h)
196.         aj = a * taw / h ** 2 - taw * b / 2 / h
197.         bj = taw * (c - 2 * a / h ** 2) - 1
198.         cj = a * taw / h ** 2 + taw * b / 2 / h
199.         if method == "1":
200.             getA, getB = implicitWithFirstDegree()
201.         elif method == "2":
202.             getA, getB = implicitWithSecondDegree()
203.         elif method == "3":
204.             getA, getB = implicitWithThirdDegree()
205.         else:
206.             return U
207.         A = getA()
208.         for k in range(0, K - 1):
209.             B = getB(k)
210.             U[k + 1] = np.linalg.solve(A, B)[:, 0]
211.
212.         return U
213.
214.     # Схема Кранка-Николсона
215.     def CrankNicolsonMethod(iCondition, bCondition, sgm):
216.         U = gridFun((0, K), (0, N))
217.         for j in range(0, N):
218.             U[0][j] = gamma0(j * h)
219.
220.         aj = sgm * a * taw / h ** 2 - taw * b / 2 / h
```

```
221.       bj = taw * (c - sgm * 2 * a / h ** 2) - 1
222.       cj = sgm * a * taw / h ** 2 + taw * b / 2 / h
223.
224.       a0 = betta0 - 3 * alpha0 / h / 2
225.       b0 = 2 * alpha0 / h
226.       c0 = - alpha0 / h / 2
227.       aN = alphal / h / 2
228.       bN = -2 * alphal / h
229.       cN = bettal + 3 * alphal / h / 2
230.       def getA():
231.           a = np.zeros((N, N))
232.           a[0][0] = a0
233.           a[0][1] = b0
234.           a[0][2] = c0
235.           for j in range(1, N - 1):
236.               a[j][j - 1] = aj
237.               a[j][j] = bj
238.               a[j][j + 1] = cj
239.           a[N - 1][N - 3] = aN
240.           a[N - 1][N - 2] = bN
241.           a[N - 1][N - 1] = cN
242.           return a
243.
244.       def getB(k):
245.           b = np.zeros((N, 1))
246.           b[0][0] = phi0(taw * (k + 1))
247.           b[N - 1][0] = phil(taw * (k + 1))
248.           for j in range(1, N - 1):
249.               b[j][0] = - U[k][j] - (1 - sgm)*(a * taw /h ** 2) * (U[k][j +
    1] - 2*U[k][j] + U[k][j - 1])
250.           return b
251.
252.       A = getA()
253.       for k in range(0, K - 1):
254.           B = getB(k)
255.           U[k + 1] = np.linalg.solve(A, B)[:, 0]
256.       return U
257.
258.   def showPostProcess(t):
259.       X = np.linspace(0, l, N)
260.       plotSlice(fResult, X, taw * t)
261.       plt.subplot(2, 1, 1)
262.       plt.plot(X, U[t])
263.
264.   def solver():
265.       global a, b, c, N, K, h, l, sigma, taw, mu, U, tt
266.       a = float(entrya.get())
267.       b = float(entryb.get())
268.       c = float(entryc.get())
269.       N = int(scaleh.get())*10
270.       tt = int(t0.get())
```

```
        sigma = float(scaleT.get())
        h = l / N
        taw = sigma * h ** 2 / a
        mu = b * taw / 2 / h
        U = []
        method = combobox.get()
        variation = combobox1.get()
        if variation == "Двухточечная с первым порядком":
            var = "1"
        elif variation == "Трехточечная со вторым порядком":
            var = "2"
        elif variation == "Двухточечная со вторым порядком":
            var = "3"
        else:
            return 0
        if method == "Явный":
            U = explicitMethod(gamma0,(phi0,phil),var)
        elif method == "Неявный":
            U = implicitMethod(gamma0, (phi0, phil), var)
        elif method == "Кранка-Николсона":
            U = CrankNicolsonMethod(gamma0, (phi0, phil), 0.5)
        else:
            return 0
        showPostProcess(tt)
        Error(U, fResult, 100)
        plt.show()


    root = tk.Tk()
    root.title("Лабораторная работа №1")
    frame = ttk.Frame(root)
    frame.grid()
    combobox = ttk.Combobox(frame, values=["Явный", "Неявный", "Кранка-
    Николсона"], height=3, width=50)
    combobox1 = ttk.Combobox(frame, values=["Двухточечная с первым порядком",
    "Трехточечная со вторым порядком", "Двухточечная со вторым порядком"],
    height=3, width=50)
    button = ttk.Button(root,text="Решить", command=solver)
    lab0 = ttk.Label(frame,text="Выберите метод")
    labgrid = ttk.Label(frame,text="Выберите параметры сетки")
    labtask = ttk.Label(frame,text="Выберите параметры
    задачи:\n\ta>0,b>0,c<0")
    sliceTask = ttk.Label(frame,text="Выберите сечение по времени")
    lab1 = ttk.Label(frame, text = "Выберите порядок \nапроксимации
    граничного условия")
    scaleh =
    tk.Scale(frame,orient=tk.HORIZONTAL,length=200,from_=0,tickinterval=20,resol
    ution=1,to=100)
    scaleT =
    tk.Scale(frame,orient=tk.HORIZONTAL,length=200,from_=0,tickinterval=0.2,reso
    lution=0.01,to=1)
```

```
312.  scaleh.set(10)
313.  scaleT.set(0.35)
314.  entrya = tk.Entry(frame,width=10,bd=10)
315.  entryb = tk.Entry(frame,width=10,bd=10)
316.  entryc = tk.Entry(frame,width=10,bd=10)
317.  t0 = tk.Entry(frame, width=10,bd=10)
318.  t0.insert(0,200)
319.  entrya.insert(0,12)
320.  entryb.insert(0,0.4)
321.  entryc.insert(0,-19)
322.  combobox.set("Явный")
323.  combobox1.set("Двухточечная с первым порядком")
324.  timeSlice = ttk.Label(frame,text="t0\t=",font="arial 20")
325.  labtaska = ttk.Label(frame,text="a\t=",font="arial 20")
326.  labtaskb = ttk.Label(frame,text="b\t=",font="arial 20")
327.  labtaskc = ttk.Label(frame,text="c\t=",font="arial 20")
328.  labgridh = ttk.Label(frame,text="N\t=",font="arial 20")
329.  labgridt = ttk.Label(frame,text="sigma\t=",font="arial 20")
330.  labelgrid0 = ttk.Label(frame,background='#cc0')
331.
332.  labtask.grid(row=1, column=0)
333.  labtaska.grid(row=1, column=1)
334.  labtaskb.grid(row=2, column=1)
335.  labtaskc.grid(row=3, column=1)
336.  entrya.grid(row=1, column=2)
337.  entryb.grid(row=2, column=2)
338.  entryc.grid(row=3, column=2)
339.  labgrid.grid(row=4,column=0)
340.  labgridh.grid(row=4, column=1)
341.  labgridt.grid(row=5, column=1)
342.  scaleh.grid(row=4, column=2)
343.  scaleT.grid(row=5, column=2)
344.  sliceTask.grid(row=6, column=0)
345.  t0.grid(row=6, column=1)
346.
347.  lab0.grid(row=8, column=0)
348.  combobox.grid(row=8, column=1)
349.
350.  lab1.grid(row=10, column=0)
351.  combobox1.grid(row=10, column=1)
352.  button.grid(row=11, column=0)
353.
354.  style = ttk.Style()
355.  style.configure("TLabel", padding=3, background='#bb2', font="arial 12",
   foreground="black")
356.  style.configure("TFrame", background='#CC0')
357.  style.configure("TButton", width=20, height=5, font="arial 20",
   foreground='red')
358.  root.mainloop()
```

Лабораторная работа №6

1. Задача

Используя явную схему крест и неявную схему, решить начально-краевую задачу для дифференциального уравнения гиперболического типа. Аппроксимацию второго начального условия произвести с первым и со вторым порядком. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 8:

$$\frac{\partial^2 u}{\partial t^2} + 2\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 2\frac{\partial u}{\partial x} - 3u,$$

$$u(0,t) = 0,$$

$$u(\pi,t) = 0,$$

$$u(x,0) = 0,$$

$$u_t(x,0) = 2\exp(-x)\sin x.$$

Аналитическое решение: $U(x,t) = \exp(-t-x)\sin x \sin(2t)$

2. Теория

Явная конечно-разностная схема

Апроксимируем вторую производную по значениям нижнего временного слоя t^k, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки:

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2} - 5u_j^k, \ \forall j \in \{1, ..., N-1\}, \forall k \in \{0, ..., K-1\}$$

Обозначим $\sigma = \frac{\tau^2}{h^2}$, тогда:

$$u_j^{k+1} = \sigma(u_{j+1}^k - 2u_j^k + u_{j-1}^k) - 5\tau^2 u_j^k + 2u_j^k - u_j^{k-1}$$

Граничные же значения u_0^{k+1} и u_N^{k+1} определяются граничными условиями $u_x(0,t)$ и $u_x(l,t)$ при помощи апроксимации производной.

Конечно-разностная схема

Будем решать задачу на заданном промежутке от 0 до l по координате x и на промежутке от 0 до заданного параметра T по времени t.

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными парамерами l, T и параметрами насыщенности сетки N, K. Тогда размер шага по каждой из координат определяется:

$$h = \frac{l}{N-1},\ \tau = \frac{T}{K-1}$$

Считая, что значения функции $u_j^k = u(x_j, t^k)$ для всех координат $x_j = jh,\ \forall j \in \{0, ..., N\}$ на предыдущих временных известно, попробуем определить значения функции на временном слое t^{k+1} путем разностной апроксимации производной:

$$\frac{\partial^2 u}{\partial t^2}(x_j, t^k) = \frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2}$$

И одним из методов апроксимации второй производной по x:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k)$$

Для расчета u_j^0 и u_j^1 можно использовать следующие формулы:

$$u_j^0 = \psi_1(x_j)$$

$$u_j^1 = \psi_1(x_j) + \tau\psi_2(x_j) + \frac{\tau^2}{2}\psi_1''(x_j) + O(\tau^2)$$

$$u_j^1 = \psi_1(x_j) + \tau\psi_2(x_j) + O(\tau^1)$$

Неявная конечно-разностная схема

Апроксимируем вторую производную по значениям верхнего временного слоя t^{k+1}, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки:

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2} = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2} - 5u_j^{k+1},\ \forall j \in \{1, ..., N-1\}, \forall k \in \{0, ..., K-1\}$$

Обозначим $\sigma = \frac{\tau^2}{h^2}$. Тогда значения функции на слое можно найти эффективны образом с помощью методом прогонки, где **СЛАУ**, кроме крайних двух уравнений, определяется коэффицетнами $a_j = 1, b_j = -(2 + 5h^2 + \frac{1}{\sigma}), c_j = 1, d_j = \frac{-2u_j^k + u_j^{k-1}}{\sigma}$ уравнений:

$$a_j u_{j-1}^{k+1} + b_j u_j^{k+1} + c_j u_{j+1}^{k+1} = d_j,\ \forall j \in \{1, ..., N-1\}$$

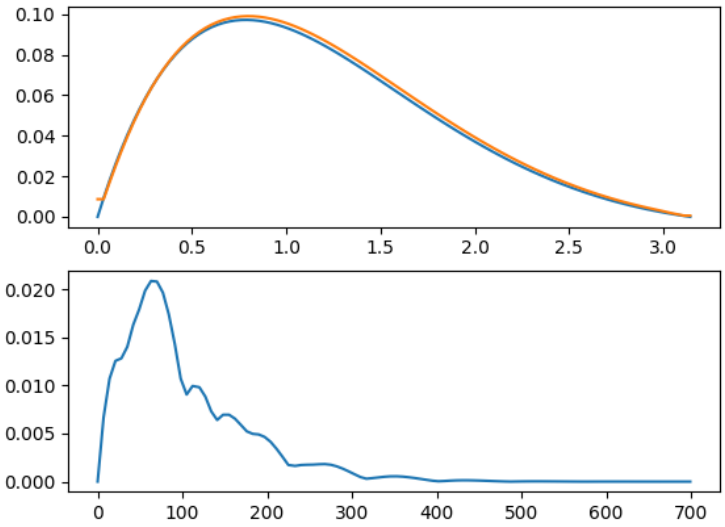
Первое и последнее уравнение системы содержащие u_0^{k+1} и u_N^{k+1} определяются граничными условиями при помощи апроксимации производной.

Неявная схема является абсолютно устойчивой.

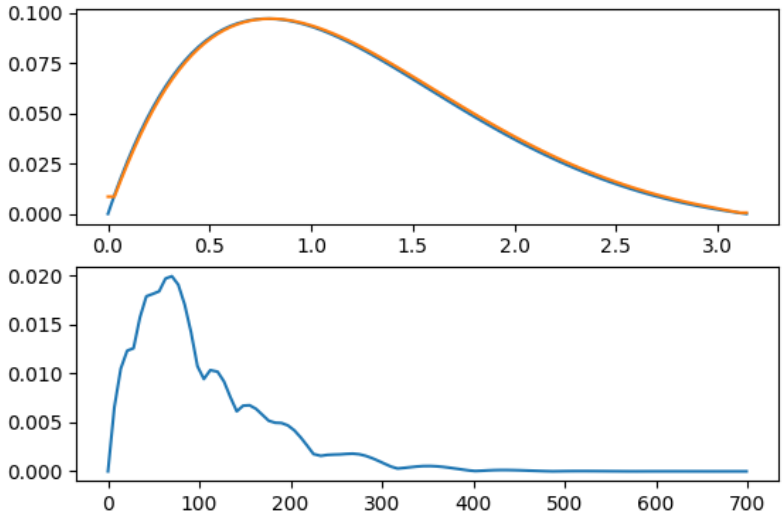
3. Результат

Явный метод:

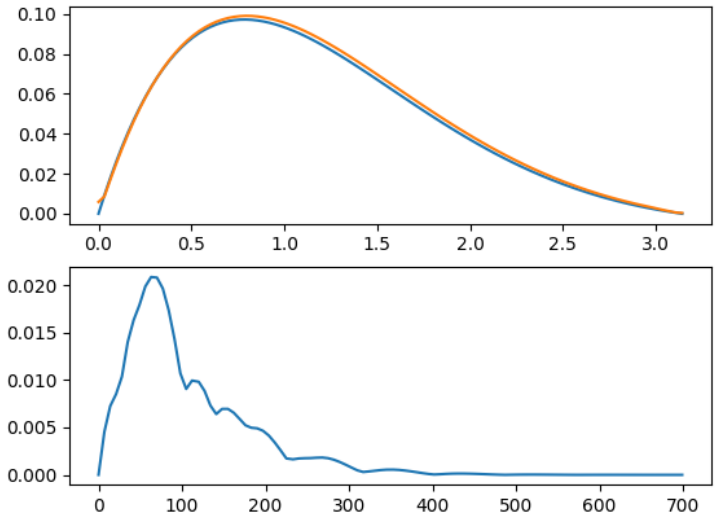
1) Порядок по пространству первый, порядок по времени первый



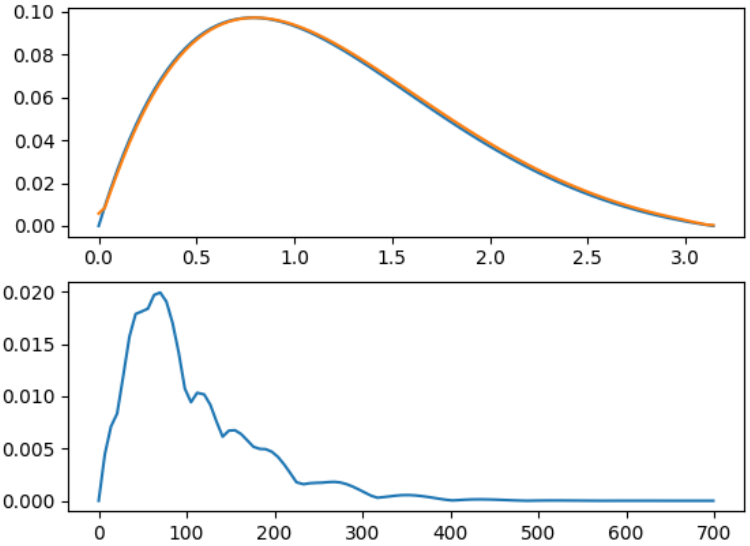
2) Порядок по пространству первый, порядок по времени второй



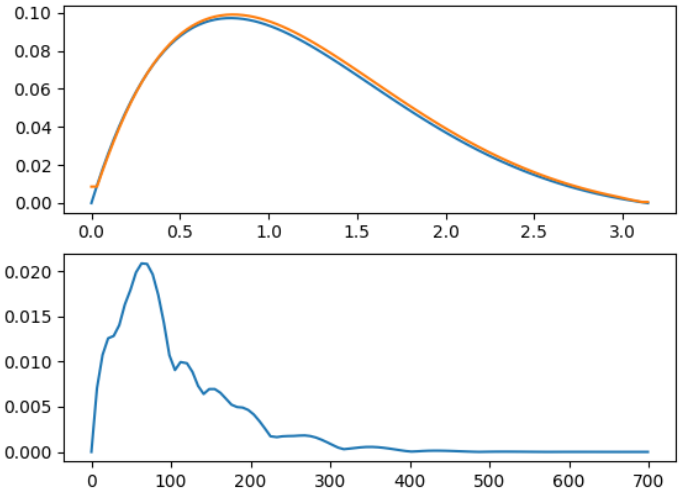
3) Порядок по пространству второй, порядок по времени первый



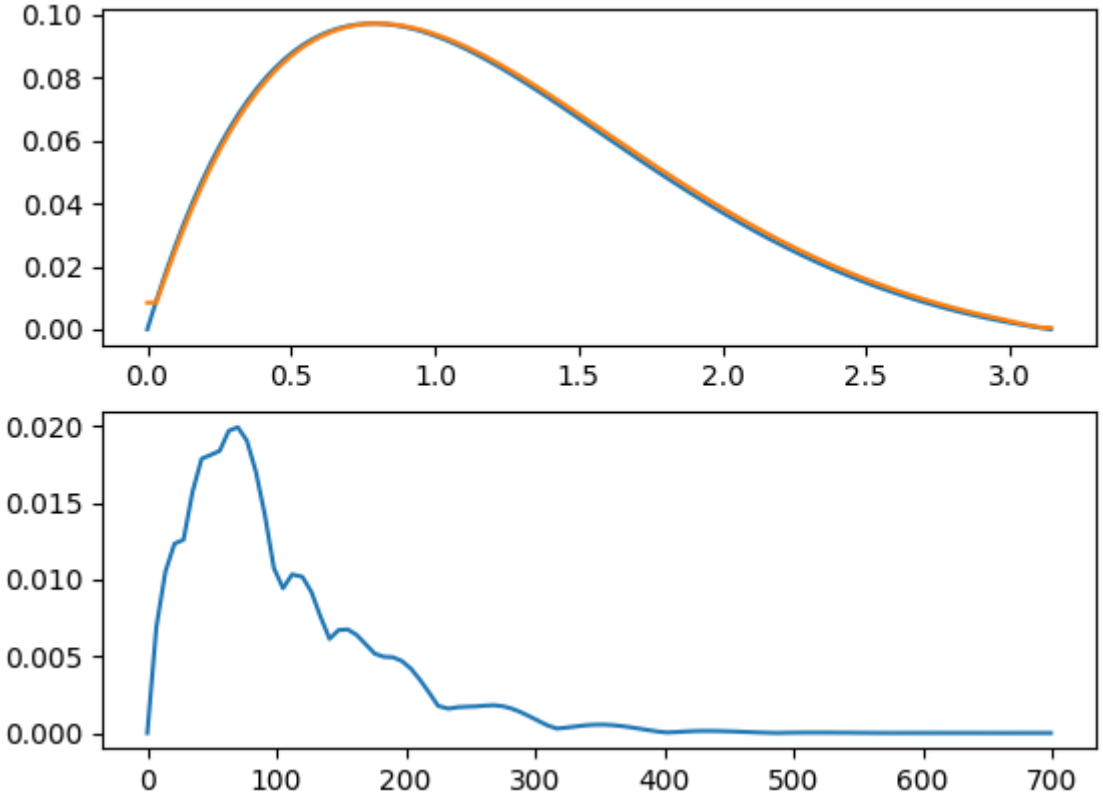
4) Порядок по пространству второй, порядок по времени второй



5) Порядок по пространству второй(по Тейлору), порядок по времени первый

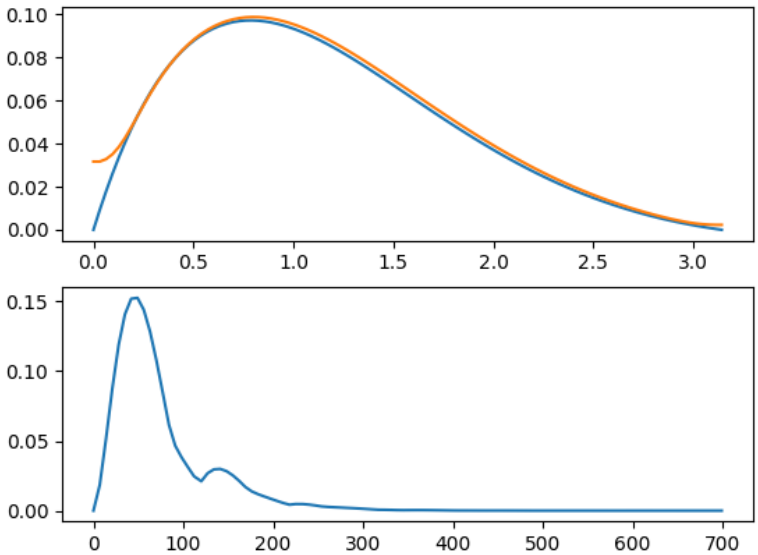


6) Порядок по пространству второй(по Тейлору), порядок по времени второй

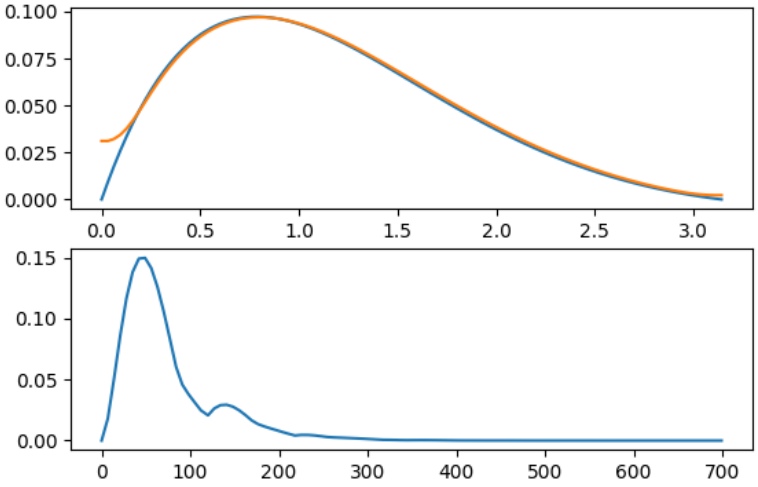


Неявный метод:

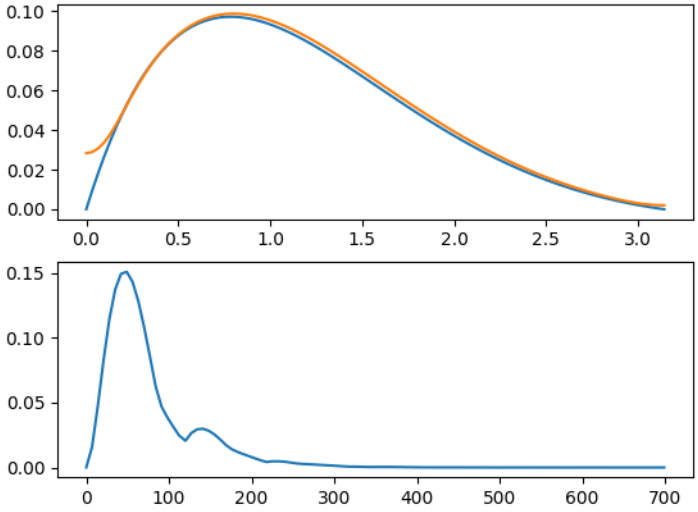
1) Порядок по пространству первый, порядок по времени первый



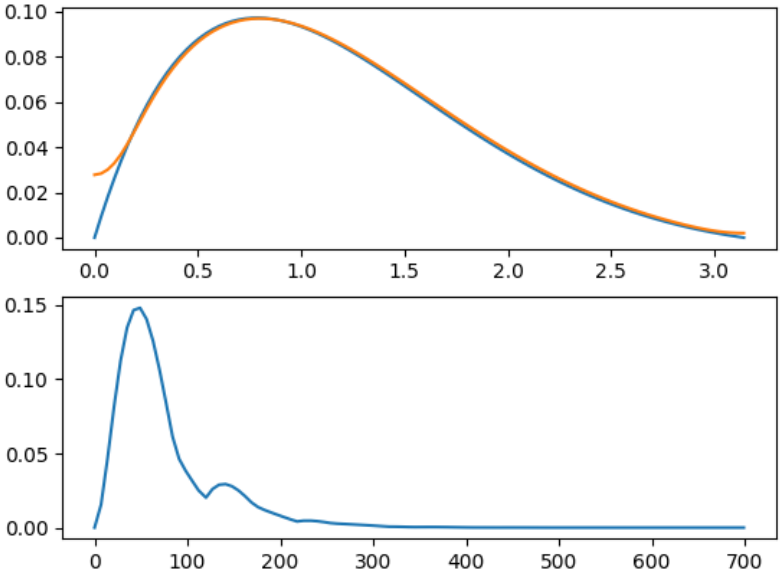
2) Порядок по пространству первый, порядок по времени второй



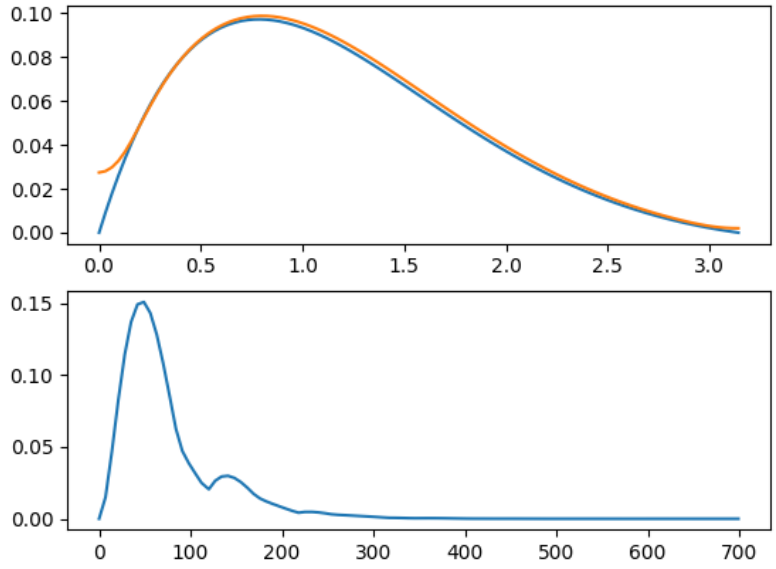
3) Порядок по пространству второй, порядок по времени первый



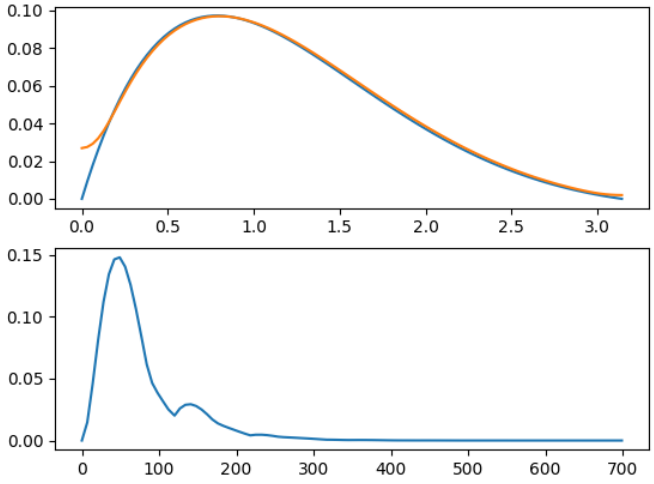
4) Порядок по пространству второй, порядок по времени второй



5) Порядок по пространству второй(по Тейлору), порядок по времени первый



6) Порядок по пространству второй(по Тейлору), порядок по времени второй



4. Вывод

Выполнив данную лабораторную работу, изучил явную схему крест и неявную схему для решения начально-краевой задачи для дифференциального уравнения гиперболического типа. Выполнил три варианта аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком и двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислил погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением.

5. Код

```
import numpy as np
from matplotlib import pyplot as plt
import tkinter as tk
import tkinter.ttk as ttk

a = 1
b = 1
c = -3
d = 2

l = np.pi
K = 700
N = 100



# Граничные условия
phi0 = lambda t: 0
phil = lambda t: 0

# Начальные условия
gamma0 = lambda x: 0
gamma1 = lambda x: 2*np.exp(-x)*np.sin(x)

f = lambda x, t: 0

# Аналитическое решение
fResult = lambda x, t: np.exp(-t-x) * np.sin(x)*np.sin(2*t)

h = l / N
sigma = 0.4
taw = np.sqrt(sigma * h ** 2 / a)
mu = b * taw**2 / 2 / h
U = []

def gridFun(xCond, tCond):
    xlCond,xrCond = xCond
    tlCond, trCond = tCond
    return np.zeros((xrCond, trCond))


def plotSlice(f, X, t):
    plt.subplot(2, 1, 1)
    plt.plot(X, f(X, t))
    plt.grid


def Error(U, fResult, count):
```

```
    X = np.linspace(0, l, N)
    Y = list(map(int, np.linspace(0, U.shape[0] - 1, count)))
    plt.subplot(2, 1, 2)
    plt.plot(Y, list(map(lambda tt: np.max(np.abs(U[tt] - fResult(X, taw*tt))), Y)))

def showPostProcess(t):
    X = np.linspace(0, l, N)
    plotSlice(fResult, X, taw * t)
    plt.subplot(2, 1, 1)
    plt.plot(X, U[t])

def D(f):
    dx = 0.000001
    return lambda x: (f(x + dx)-f(x)) / dx


# Явная конечно-разностная схема
def explicitMethod(iCondition, bCondition, f, order=(1, 1)):
    def explicitWithFirstDegreeB():
        # 1 порядка
        for k in range(1, K - 1):
            U[k + 1][0] = (- alpha0 / h * U[k + 1][1] + phi0(taw * (k + 1))) /
(betta0 - alpha0 / h)
            U[k + 1][N - 1] = (alphal / h * U[k + 1][N - 2] + phil(taw * (k + 1))) /
(bettal + alphal / h)
        pass

    def explicitWithSecondDegreeB():
        # 2 порядка
        for k in range(1, K - 1):
            U[k + 1][0] = (- alpha0 / h / 2 * (4 * U[k + 1][1] - U[k + 1][2]) +
phi0(taw * (k + 1))) / (
            betta0 - 3 * alpha0 / h / 2)
            U[k + 1][N - 1] = (- alphal / h / 2 * (U[k + 1][N - 3] - 4 * U[k + 1][N -
2]) + phil(taw * (k + 1))) / (
            bettal + 3 * alphal / h / 2)

    def explicitWithThirdDegreeB():
        b0 = 2*a/h + h/taw**2 - h*c - betta0/alpha0*(2*a - b*h) + d*h/2/taw
        c0 = -2*a/h
        bN = 2*a/h + h/taw**2 - h*c + bettal/alphal*(2*a + b*h) + d*h/2/taw
        aN = -2*a/h
        for k in range(1, K - 1):
            d0 = 2*h/taw**2 * U[k][0] + (- h/taw**2 + d*h/2/taw)*U[k - 1][0] -
phi0(taw*(k + 1)) * (2*a - b*h) / alpha0 + h*f(0, taw * (k + 1))
            dN = 2*h/taw**2 * U[k][N - 1] + (-h/taw**2 + d*h/2/taw)*U[k - 1][N - 1] +
phil(taw*(k + 1)) * (2*a + b*h) / alphal - h*f(l, taw*(k + 1))
            U[k + 1][0] = (d0 - c0 * U[k + 1][1]) / b0
            U[k + 1][N - 1] = (dN - aN * U[k + 1][N - 2]) / bN
            # проверить знаки!!!!
```

```
    def explicitWithFirstDegreeI():
        for j in range(0, N):
            U[1][j] = gamma0(j * h) + gamma1(j * h) * taw

    def explicitWithSecondDegreeI():
        for j in range(0, N):
            U[1][j] = gamma0(j * h) + gamma1(j * h) * (taw - d*taw**2/2) + (a *
D(D(gamma0))(j * h) + b * D(gamma0)(j * h) +
            c * gamma0(j * h) + f(j * h, taw))*taw**2/2

    alpha0 = 1
    betta0 = 0
    alphal = 1
    bettal = 0

    bOrd, iOrd = order
    gamma0,gamma1 = iCondition
    phi0, phil = bCondition
    U = gridFun((0, K), (0, N))

    for j in range(0, N):
        U[0][j] = gamma0(j * h)
    if iOrd == 1:
        explicitWithFirstDegreeI()
    elif iOrd == 2:
        explicitWithSecondDegreeI()
    else:
        pass
    for k in range(1, K - 1):
        for j in range(1, N - 1):
            U[k + 1][j] = ((sigma + mu) * U[k][j + 1] + (-2 * sigma + 2 + c*taw**2) *
U[k][j] + (sigma - mu) * U[k][
                j - 1] + (-1 + d*taw/2) * U[k - 1][j] + taw**2 * f(j*h, k*taw))/(1 +
d*taw/2)
            pass
    if bOrd == 1:
        explicitWithFirstDegreeB()
    elif bOrd == 2:
        explicitWithSecondDegreeB()
    elif bOrd == 3:
        explicitWithThirdDegreeB()
    else:
        pass
    return U


# Неявный метод
def implicitMethod(iCondition, bCondition, f, order=(1,1)):

    def implicitWithFirstDegreeI():
        for j in range(0, N):
```

```
            U[1][j] = gamma0(j * h) + gamma1(j * h) * taw

    def implicitWithSecondDegreeI():
        for j in range(0, N):
            U[1][j] = gamma0(j * h) + gamma1(j * h) * (taw - d*taw**2/2) + (a *
D(D(gamma0))(j * h) + b * D(gamma0)(j * h) +
            c * gamma0(j * h) + f(j * h, taw))*taw**2/2

    def implicitWithFirstDegree():
        b0 = betta0 - alpha0 / h
        c0 = alpha0 / h
        aN = - alphal / h
        bN = bettal + alphal / h

        def gA():
            aa = np.zeros((N, N))
            aa[0][0] = b0
            aa[0][1] = c0
            for j in range(1, N - 1):
                aa[j][j - 1] = aj
                aa[j][j] = bj
                aa[j][j + 1] = cj
            aa[N - 1][N - 2] = aN
            aa[N - 1][N - 1] = bN
            return aa

        def gB(k):
            bb = np.zeros((N, 1))
            bb[0][0] = phi0(taw * (k + 1))
            bb[N - 1][0] = phil(taw * (k + 1))
            for j in range(1, N - 1):
                bb[j][0] = U[k - 1][j]*(-t00+d00)+U[k][j]*(2*t00 + c) + f(j*h, k*taw)
            return bb
        return gA, gB

    def implicitWithSecondDegree():
        a0 = betta0 - 3*alpha0/ h/ 2
        b0 = 2 * alpha0/h
        c0 = - alpha0 / h /2
        aN = alphal / h /2
        bN = -2 * alphal/h
        cN = bettal + 3*alphal/ h/ 2

        def gA():
            aa = np.zeros((N, N))
            aa[0][0] = a0
            aa[0][1] = b0
            aa[0][2] = c0
            for j in range(1, N - 1):
                aa[j][j - 1] = aj
                aa[j][j] = bj
```

```
                aa[j][j + 1] = cj
            aa[N - 1][N - 2] = bN
            aa[N - 1][N - 1] = cN
            aa[N - 1][N - 3] = aN
            return aa

        def gB(k):
            bb = np.zeros((N, 1))
            bb[0][0] = phi0(taw * (k + 1))
            bb[N - 1][0] = phil(taw * (k + 1))
            for j in range(1, N - 1):
                bb[j][0] = U[k - 1][j]*(-t00+d00)+U[k][j]*(2*t00 + c) + f(j*h, k*taw)
            return bb
        return gA, gB

    def implicitWithThirdDegree():
        b0 = 2 * a / h + h / taw ** 2 - h * c - betta0 / alpha0 * (2 * a - b * h) + d
* h / 2 / taw
        c0 = -2 * a / h
        bN = 2 * a / h + h / taw ** 2 - h * c + bettal / alphal * (2 * a + b * h) + d
* h / 2 / taw
        aN = -2 * a / h
        def gA():
            aa = np.zeros((N, N))
            aa[0][0] = b0
            aa[0][1] = c0
            for j in range(1, N - 1):
                aa[j][j - 1] = aj
                aa[j][j] = bj
                aa[j][j + 1] = cj
            aa[N - 1][N - 2] = aN
            aa[N - 1][N - 1] = bN
            return aa

        def gB(k):
            bb = np.zeros((N, 1))
            d0 = 2 * h / taw ** 2 * U[k][0] + (- h / taw ** 2 + d * h / 2 / taw) *
U[k - 1][0] - phi0(taw * (k + 1)) * (
            2 * a - b * h) / alpha0 + h * f(0, taw * (k + 1))
            dN = 2 * h / taw ** 2 * U[k][N - 1] + (-h / taw ** 2 + d * h / 2 / taw) *
U[k - 1][N - 1] + phil(
                taw * (k + 1)) * (2 * a + b * h) / alphal - h * f(l, taw * (k + 1))
            bb[0][0] = d0
            bb[N - 1][0] = dN
            for j in range(1, N - 1):
                bb[j][0] = U[k - 1][j]*(-t00+d00)+U[k][j]*(2*t00 + c) + f(j*h, k*taw)
            return bb

        return gA, gB

    alpha0 = 1
```

```
    betta0 = 0
    alphal = 1
    bettal = 0

    a00 = a/h**2
    b00 = b/2/h
    t00 = 1/taw**2
    d00 = d/2/taw

    aj = b00 - a00
    bj = t00 + d00 + 2 *a00
    cj = - b00 - a00

    bOrd, iOrd = order
    gamma0, gamma1 = iCondition
    phi0, phil = bCondition
    U = gridFun((0, K), (0, N))
    for j in range(0, N):
        U[0][j] = gamma0(j * h)

    if iOrd == 1:
        implicitWithFirstDegreeI()
    elif iOrd == 2:
        implicitWithSecondDegreeI()
    else:
        pass

    if bOrd == 1:
        getA, getB = implicitWithFirstDegree()
    elif bOrd == 2:
        getA, getB = implicitWithSecondDegree()
    elif bOrd == 3:
        getA, getB = implicitWithThirdDegree()
    else:
        return U
    A = getA()
    for k in range(1, K - 1):
        B = getB(k)
        U[k + 1] = np.linalg.solve(A, B)[:, 0]

    return U



def solver():
    global a, b, c, N, K, h, l, sigma, taw, mu, U, tt
    a = float(entrya.get())
    b = float(entryb.get())
    c = float(entryc.get())
    d = float(entryd.get())
    N = int(scaleh.get()) * 10
```

```
    tt = int(t0.get())
    sigma = float(scaleT.get())
    h = l / N
    taw = np.sqrt(sigma * h ** 2 / a)
    mu = b * taw**2 / 2 / h
    U = []
    method = combobox.get()
    iOrder = combobox1.get()
    bOrder = combobox2.get()

    if iOrder == "Первый":
        iOrd = 1
    elif iOrder == "Второй":
        iOrd = 2
    elif iOrder == "Второй (по Тейлору)":
        iOrd = 3
    else:
        pass

    if bOrder == "Первый":
        bOrd = 1
    elif bOrder == "Второй":
        bOrd = 2
    else:
        pass

    currentOrder = (iOrd, bOrd)
    if method == "Явный":
        U = explicitMethod((gamma0, gamma1), (phi0, phi1), f, order=currentOrder)
    elif method == "Неявный":
        U = implicitMethod((gamma0, gamma1), (phi0, phi1), f,order=currentOrder)
    else:
        pass
    showPostProcess(tt)
    Error(U, fResult, 100)
    plt.show()


root = tk.Tk()
root.title("Лабораторная работа №2")
frame = ttk.Frame(root)
frame.grid()
combobox = ttk.Combobox(frame, values=["Явный", "Неявный"], height=3, width=50)
combobox2 = ttk.Combobox(frame, values=["Первый", "Второй"], height=3, width=50)
combobox1 = ttk.Combobox(frame, values=["Первый", "Второй", "Второй (по Тейлору)"],
height=3, width=50)
button = ttk.Button(root,text="Решить", command=solver)
# image = tk.PhotoImage(file="task.PNG")
# lab = ttk.Label(frame,image=image)
lab0 = ttk.Label(frame,text="Выберите метод")
labgrid = ttk.Label(frame,text="Выберите параметры сетки")
```

```
labtask = ttk.Label(frame,text="Выберите параметры задачи:\n\ta>0,b>0,c<0,d>0")
sliceTask = ttk.Label(frame,text="Выберите сечение по времени")
lab1 = ttk.Label(frame, text = "Выберите порядок \n(по пространству)")
lab2 = ttk.Label(frame, text = "Выберите порядок \n(по времени)")
scaleh =
tk.Scale(frame,orient=tk.HORIZONTAL,length=200,from_=0,tickinterval=20,resolution=1,t
o=100)
scaleT =
tk.Scale(frame,orient=tk.HORIZONTAL,length=200,from_=0,tickinterval=0.4,resolution=0.
01,to=2)
scaleh.set(10)
scaleT.set(0.35)

entrya = tk.Entry(frame,width=10,bd=10)
entryb = tk.Entry(frame,width=10,bd=10)
entryc = tk.Entry(frame,width=10,bd=10)
entryd = tk.Entry(frame,width=10,bd=10)

t0 = tk.Entry(frame, width=10,bd=10)
t0.insert(0,10)

entrya.insert(0,a)
entryb.insert(0,b)
entryc.insert(0,c)
entryd.insert(0,d)

combobox.set("Явный")
combobox1.set("Первый")
combobox2.set("Первый")

timeSlice = ttk.Label(frame,text="t0\t=",font="arial 20")
labtaska = ttk.Label(frame,text="a\t=",font="arial 20")
labtaskb = ttk.Label(frame,text="b\t=",font="arial 20")
labtaskc = ttk.Label(frame,text="c\t=",font="arial 20")
labtaskd = ttk.Label(frame,text="d\t=",font="arial 20")
labgridh = ttk.Label(frame,text="N\t=",font="arial 20")
labgridt = ttk.Label(frame,text="sigma\t=",font="arial 20")
labelgrid0 = ttk.Label(frame,background='#cc0')

# lab.grid(row=0, column=0, columnspan=3)
labtask.grid(row=1, column=0)
labtaska.grid(row=1, column=1)
labtaskb.grid(row=2, column=1)
labtaskc.grid(row=3, column=1)
labtaskd.grid(row=4, column=1)

entrya.grid(row=1, column=2)
entryb.grid(row=2, column=2)
entryc.grid(row=3, column=2)
entryd.grid(row=4, column=2)
```

```
labgrid.grid(row=5,column=0)
labgridh.grid(row=5, column=1)
labgridt.grid(row=6, column=1)

scaleh.grid(row=5, column=2)
scaleT.grid(row=6, column=2)
sliceTask.grid(row=7, column=0)
t0.grid(row=7, column=1)

lab0.grid(row=9, column=0)
combobox.grid(row=9, column=1)

lab1.grid(row=11, column=0)
combobox1.grid(row=11, column=1)
lab2.grid(row=12,column=0)
combobox2.grid(row=12,column=1)
button.grid(row=13, column=0)

style = ttk.Style()
style.configure("TLabel", padding=3, background='#bb2', font="arial 12",
foreground="black")
style.configure("TFrame", background='#CC0')
style.configure("TButton", width=20, height=5, font="arial 20", foreground='red')
root.mainloop()
```

Лабораторная работа №7

1. Задача

Решить краевую задачу для дифференциального уравнения эллиптического типа. Аппроксимацию уравнения произвести с использованием центрально-разностной схемы. Для решения дискретного аналога применить следующие методы: метод простых итераций (метод Либмана), метод Зейделя, метод простых итераций с верхней релаксацией. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 8:

8.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2\frac{\partial u}{\partial x} - 3u,$$

$$u(0, y) = \cos y,$$

$$u(\frac{\pi}{2}, y) = 0,$$

$$u(x, 0) = \exp(-x)\cos x,$$

$$u(x, \frac{\pi}{2}) = 0.$$

Аналитическое решение: $U(x, y) = \exp(-x)\cos x \cos y$.

2. Теория

Конечно-разностная схема

Будем решать задачу на заданном промежутке от 0 до l_x по координате x и на промежутке от 0 до l_y по координате y.

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными парамерами l_x, l_y и параметрами насыщенности сетки N_x, N_y. Тогда размер шага по каждой из координат определяется:

$$h_x = \frac{l_x}{N_x - 1},\ h_y = \frac{N_y}{N_y - 1}$$

Попробуем определить связь между дискретными значениями функции путем разностной апроксимации производной:

$$\frac{\partial^2 u}{\partial x^2}(x_j, y_i) + \frac{\partial^2 u}{\partial y^2}(x_j, y_i) = \frac{u_{j-1,i} - 2u_{j,i} + u_{j+1,i}}{h_x^2} + \frac{u_{j,i-1} - 2u_{j,i} + u_{j,i+1}}{h_y^2}$$

Тогда выражая из искомого уравнения значение $u_{i,j} = \frac{h_y^2(u_{j-1,i} + u_{j+1,i}) + h_x^2(u_{j,i-1} + u_{j,i+1})}{2(h_x^2 + h_y^2)}$, мы получаем основу для применения иттерационных методов решения *СЛАУ*.

Для расчета $u_{j,0}$ и $u_{0,i}$ следует использовать граничные условия.

Начальная инициализация

Поскольку в нашем варианте известны граничные значения $u(x, l_{y0})$ и $u(x, l_{y1})$, то для начальной инициаизации значений в сетке можно использовать линейную интерполяцию при фиксированном $x = x_j$ для улучшения сходимости:

$$u_{j,i} = \frac{u(x_j, l_{y1}) - u(x_j, l_{y0})}{l_{y1} - l_{y0}} \cdot (y_i - l_{y0}) + u(x_j, l_{y0})$$

Граничные значения

Для границ по y координате значения заданы явно граничным условием, и мы можем определить их на начальном этапе при инициализации.

Для границ по x координате аппроксимируем значение производной из граничного условия с помощью трёхточечной аппроксимации в точках $x = 0$ и $x = l$ и получаем 2 новых уравнения в *СЛАУ* соответственно:

$$\frac{-3u_{0,i} + 4u_{1,i} - u_{2,i}}{2h_x} = \phi_0(y_i)$$

$$\frac{3u_{N,i} - 4u_{N-1,i} + u_{N-2,i}}{2h_x} = \phi_1(y_i)$$

Тогда основа для иттерационного метода:

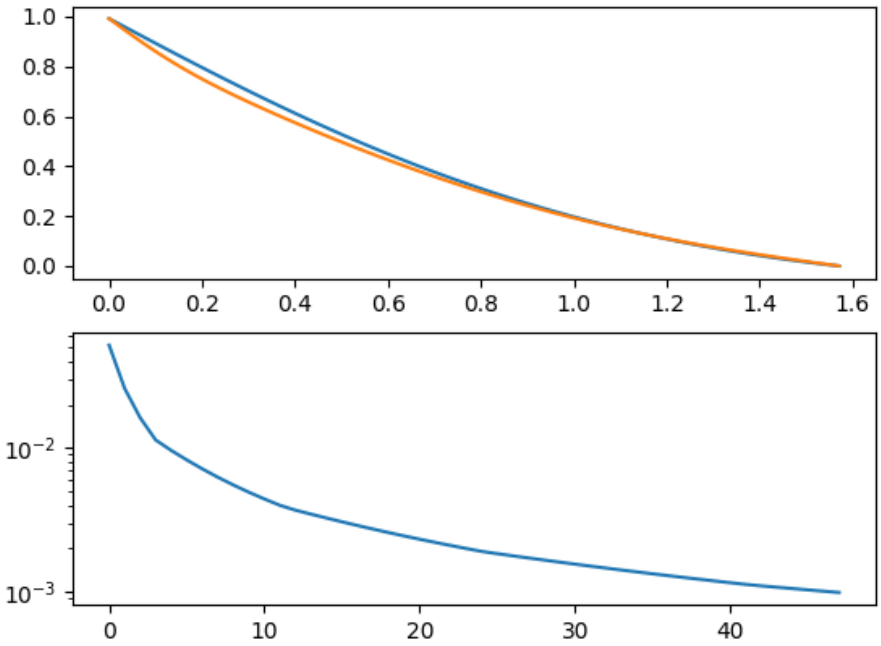
$$u_{0,i} = \frac{-2h_x\phi_0(y_i) + 4u_{1,i} - u_{2,i}}{3}$$

$$u_{N,i} = \frac{2h_x\phi_1(y_i) + 4u_{N-1,i} - u_{N-2,i}}{3}$$

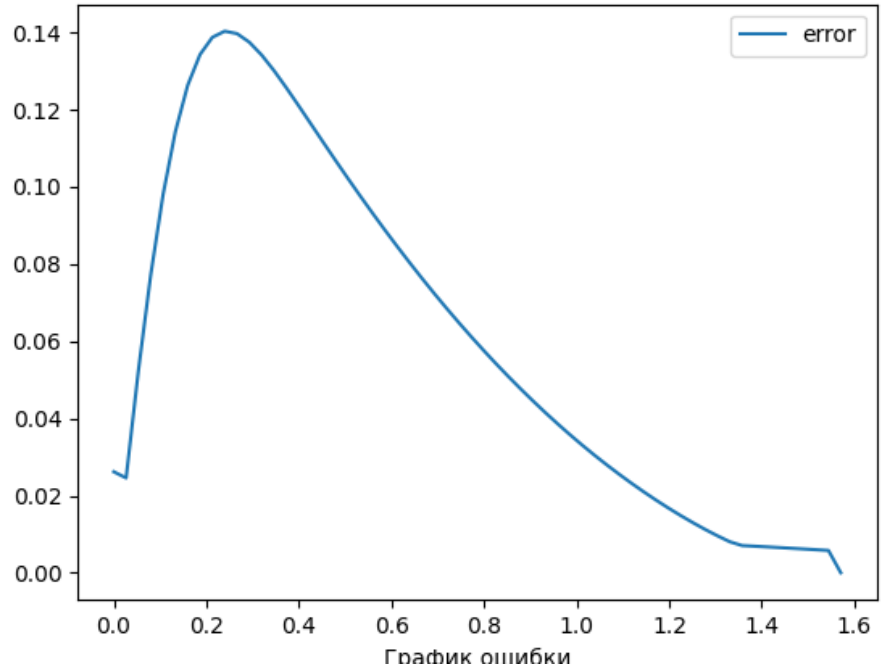
3. Результат

1) Метод Либмана

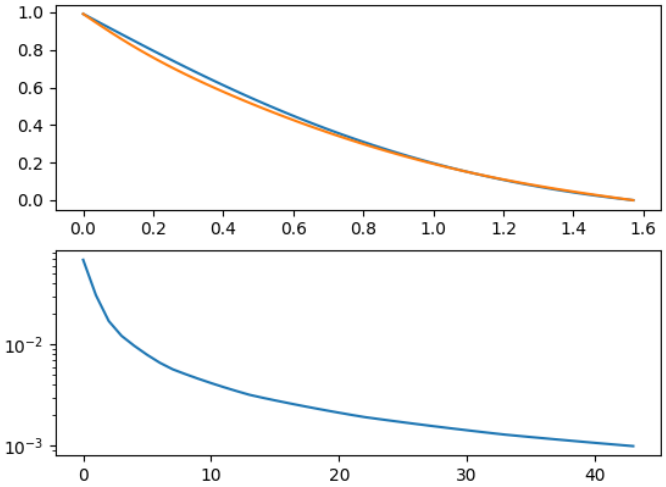
Libman - 48



максимальная ошибка = 0.1403483774364367

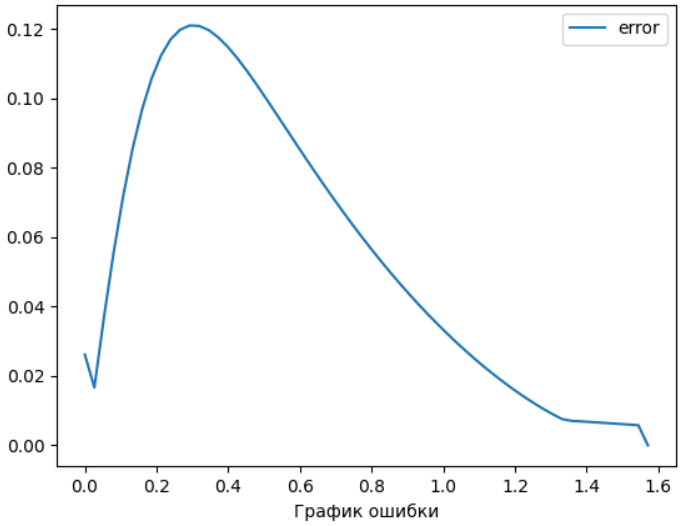


2) Метод Зейделя

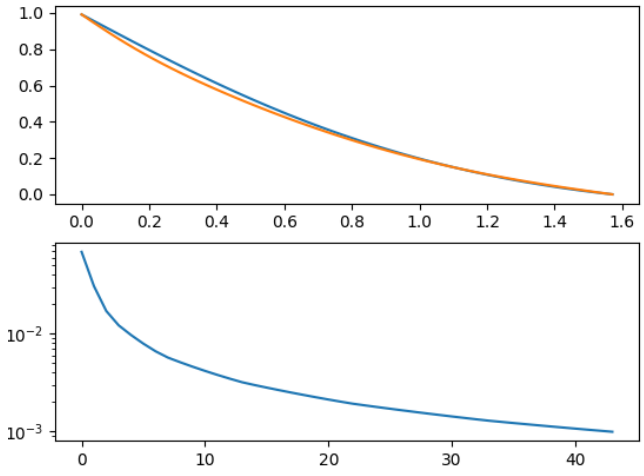


Seidel - 44

максимальная ошибка = 0.12104484264164794



3) Метод простых итераций с верхней релаксацией upperRelaxation - 44



максимальная ошибка = 0.12104484264164794

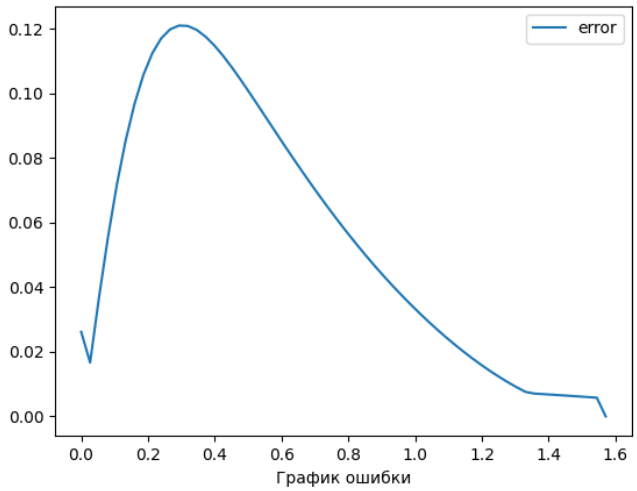


График ошибки

4. Вывод

Для выполнения данной лабораторной работы нужно было решить краевую задачу для дифференциального уравнения эллиптического типа. Также нужно было аппроксимировать уравнения с использованием центрально-разностной схемы. Для решения дискретного аналога пришлось вспомнить метод простых итераций (метод Либмана), метод Зейделя и применить новый для меня метод простых итераций с верхней релаксацией.

5. Код

```
import numpy as np
from matplotlib import pyplot as plt

import warnings
warnings.filterwarnings("ignore")

def gridFun(xCond, tCond):
    xlCond,xrCond = xCond
    tlCond, trCond = tCond
    return np.zeros((xrCond, trCond))


def plotSlice(f, X, t):
    plt.subplot(2, 1, 1)
    plt.plot(X, f(X, t))
    plt.grid


def showPostProcess(y, err):
    X = np.linspace(0, lx, K)
    plotSlice(fResult, X, hy * y)
    plt.subplot(2, 1, 1)
    plt.plot(X, U[:, y])
    plt.subplot(2, 1, 2)
    plt.semilogy(err)
```

```
    plt.show()

def error(U):
    x1, y1 = np.mgrid[0:np.pi/2:60j, 0:np.pi/2:60j]
    z1 = np.exp(-x1)*np.cos(x1)*np.cos(y1)
    X = np.linspace(0, np.pi/2, 60)
    Y = []
    for t in range(60):
        Y.append(np.max(np.abs(U[t]-z1[t])))
    print("максимальная ошибка = {}".format(np.max(Y)))
    plt.plot(X, Y, label = 'error')
    plt.xlabel('График ошибки')
    plt.legend()
    plt.show()

def ellipticEquation(xConds, yConds, method):
    def interpolation():
        for i in range(1, K - 1):
            for j in range(1, N - 1):
                alpha = (j * hy) / ly
                U[i][j] = ux0(i * hx)*(1 - alpha) + uxl(i * hx) * alpha
        return U


    def Libman(U, epsil):
        n = 0
        errors = []
        while True:
            n += 1
            Uold = U.copy()
            for i in range(1, K - 1):
                for j in range(1, N - 1):
                    U[i][j] = delta * ((hhx + ahx) * Uold[i - 1][j] +
                                       (hhx - ahx) * Uold[i + 1][j] +
                                       (hhy + bhy) * Uold[i][j - 1] +
                                       (hhy - bhy) * Uold[i][j + 1])
            err = np.max(np.abs(Uold - U))
            errors.append(err)
            if (err < epsil):
                break
        print("Libman - ", n)
        return U, errors

    def Seidel(U, epsil):
        n = 0
        errors = []
        while True:
            n += 1
            Uold = U.copy()
            for i in range(1, K - 1):
                for j in range(1, N - 1):
```

```
                U[i][j] = delta * ((hhx + ahx) * U[i - 1][j] +
                                   (hhx - ahx) * U[i + 1][j] +
                                   (hhy + bhy) * U[i][j - 1] +
                                   (hhy - bhy) * U[i][j + 1])
        err = np.max(np.abs(Uold - U))
        errors.append(err)
        if (err < epsil):
            break
    print("Seidel - ", n)
    return U, errors

def upperRelaxation(U, epsil, omega):
    n = 0
    good = False
    errors = []
    while (n < 1000):
        n += 1
        Uold = U.copy()
        for i in range(1, K - 1):
            for j in range(1, N - 1):
                U[i][j] = U[i][j] + omega * (delta * ((hhx + ahx) * U[i - 1][j] +
                                   (hhx - ahx) * Uold[i + 1][j] +
                                   (hhy + bhy) * U[i][j - 1] +
                                   (hhy - bhy) * Uold[i][j + 1]) - U[i][j])
        err = np.max(np.abs(Uold - U))
        errors.append(err)
        if (err < epsil):
            good = True
            break
    if (not good):
        print("Расходится!!!")
    print("upperRelaxation - ", n)
    return U, errors

ux0, uxl = xConds
u0y, uly = yConds
U = gridFun((0, K), (0, N))

for i in range(0, K):
    U[i][0] = ux0(hx * i)
    U[i][N-1] = uxl(hx * i)

for j in range(0, N):
    U[0][j] = u0y(hy * j)
    U[K-1][j] = uly(hy * j)

delta = 1/(2/hx**2 + 2/hy**2 + c)
hhx = 1/hx**2
ahx = a/2/hx
hhy = 1/hy**2
bhy = b/2/hy
```

```
    err = []
    U = interpolation()
    if method == 1:
        U, err = Libman(U, epsil)
    elif method == 2:
        U, err = Seidel(U, epsil)
    elif method == 3:
        U, err = upperRelaxation(U, epsil, omega)
    else:
        pass
    return U, err

def solver():
    global a, b, c, N, K, h, hx, hy, U, omega,epsil
    a = a
    b = b
    c = c
    omega = omega
    K = 100
    N = 100
    tt = 5
    hy = ly / N
    hx = lx / K
    U = []
    err = []
    epsil = 0.0001

a = -1
b = -1
c = -2
lx = np.pi/2
ly = np.pi/2
omega = 1
K = 60
N = 60
epsil = 0.001
tt = 5

u0y = lambda y: np.cos(y)
uly = lambda y: 0
ux0 = lambda x: np.exp(-x)*np.cos(x)
uxl = lambda x: 0
fResult = lambda x, y: np.exp(-x)*np.cos(x)*np.cos(y)

hx = lx / K
hy = ly / N
U = []

U, err = ellipticEquation((ux0, uxl), (u0y, uly), 1)
showPostProcess(tt, err)
error(U)
```

Лабораторная работа №8

1. Задача

Используя схемы переменных направлений и дробных шагов, решить двумерную начально-краевую задачу для дифференциального уравнения параболического типа. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 8:

8.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - xy\sin t,$$

$$u(0, y, t) = 0,$$

$$u(1, y, t) - u_x(1, y, t) = 0,$$

$$u(x, 0, t) = 0,$$

$$u(x, 1, t) - u_y(x, 1, t) = 0,$$

$$u(x, y, 0) = xy.$$

Аналитическое решение: $U(x, y, t) = xy\cos t$.

2. Теория

Конечно-разностная схема

Будем решать задачу на заданной площади от 0 до l_x по координате x, от 0 до l_y по координате y и на промежутке от 0 до заданного параметра T по времени t.

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными парамерами l_x, l_y, T и параметрами насыщенности сетки N_x, N_y, K. Тогда размер шага по каждой из координат определяется:

$$h_x = \frac{l_x}{N_x - 1},\ h_y = \frac{l_y}{N_y - 1},\ \tau = \frac{T}{K - 1}$$

Конечно-разностная схема решения параболического типа в сетке на временном слое t^{k+1} определяется с помощью 2-ух этапов, на каждом из которых решается трёхдиагональное уравнение с помощью метода прогонки:

- Считая, что значения функции $u_{i,j}^k = u(x_i, y_j, t^k)$ на временном слое t^k известно, попробуем определить значения функции на временном слое $t^{k+\frac{1}{2}}$ путем разностной апроксимации производной по времени: $\frac{\partial u}{\partial t}(x_i, y_j, t^k) = (1+\gamma)\frac{u_{i,j}^{k+\frac{1}{2}} - u_{i,j}^k}{\tau}$, неявной аппроксимацией производной по x: $\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t^k) = \frac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2}$ и явной аппроксимацией по y: $\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t^k) = \frac{u_{i,j-1}^k - 2u_{i,j}^k + u_{i,j+1}^k}{h_y^2}$ получаем уравнение:

$$-a\tau h_x^2\gamma u_{i,j-1}^k - ((1+\gamma)h_x^2h_y^2 - 2a\tau h_x^2\gamma)u_{i,j}^k - a\tau h_x^2\gamma u_{i,j+1}^k = a\tau h_y^2 u_{i-1,j}^{k+\frac{1}{2}} - (2a\tau h_y^2 + (1+\gamma)h_x^2h_y^2)u_{i,j}^{k+\frac{1}{2}} + a\tau h_y^2 u_{i+1,j}^{k+\frac{1}{2}}$$

- Считая, что значения функции $u_{i,j}^{k+\frac{1}{2}} = u(x_i, y_j, t^{k+\frac{1}{2}})$ на временном слое $t^{k+\frac{1}{2}}$ известно из прошлого этапа, попробуем определить значения функции на временном слое t^{k+1} путем разностной апроксимации производной по времени: $\frac{\partial u}{\partial t}(x_i, y_j, t^{k+\frac{1}{2}}) = (1+\gamma)\frac{u_{i,j}^{k+1} - u_{i,j}^{k+\frac{1}{2}}}{\tau}$, явной аппроксимацией производной по x: $\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \frac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2}$ и неявной аппроксимацией по y: $\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \frac{u_{i,j-1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j+1}^{k+1}}{h_y^2}$ получим второе уравнение:

$$-a\tau h_y^2\gamma u_{i-1,j}^{k+\frac{1}{2}} - ((1+\gamma)h_x^2h_y^2 - 2a\tau h_y^2\gamma)u_{i,j}^{k+\frac{1}{2}} - a\tau h_y^2\gamma u_{i+1,j}^{k+\frac{1}{2}} = a\tau h_x^2 u_{i,j-1}^{k+1} - (2a\tau h_x^2 + (1+\gamma)h_x^2h_y^2)u_{i,j}^{k+1} + a\tau h_x^2 u_{i,j+1}^{k+1}$$

При $\gamma = 1$ получаем метод переменных направлений, когда как при $\gamma = 0$ - метод дробных шагов.

Значения на слое $u_{i,j}^0$ и на границах сетки определяются с помощью заданных граничных условий и их аппроксимаций.

Апроксимация первых производных

Для того, чтобы получить 2-ой порядок аппроксимации будем аппроксимировать верхнюю границу по y трёхточечной аппроксимацией в явном виде и двухточечной второго порядка в неявном методе.

Трёхточечная аппроксимация второго порядка

Трёхточечная апроксимация второго порядка в точке $y = l_y$ равна соответственно:

$$\frac{3u_{i,N_y}^{k+\frac{1}{2}} - 4u_{i,N_y-1}^{k+\frac{1}{2}} + u_{i,N_y-2}^{k+\frac{1}{2}}}{2h_y} = \psi_1(x_i, t^{k+\frac{1}{2}})$$

Тогда, поскольку мы знаем значения для внутренних узлов, получаем выражение для граничного значения при явном методе:

$$u_{i,N_y}^{k+\frac{1}{2}} = \frac{2h_y\psi_1(x_i, t^{k+\frac{1}{2}}) + 4u_{i,N_y-1}^{k+\frac{1}{2}} - u_{i,N_y-2}^{k+\frac{1}{2}}}{3}$$

Двухточечная аппроксимация второго порядка

Двухточечная апроксимация второго порядка в точке $y = l_y$ равна соответственно:

$$\frac{u_{i,N_y+1}^{k+1} - u_{i,N_y-1}^{k+1}}{2h_y} = \psi_1(x_i, t^{k+1})$$

Тогда, поскольку мы знаем значения для внутренних узлов, получаем выражение для граничного значения при неявном методе:

$$-a\tau h_y^2\gamma u_{i-1,N_y}^{k+\frac{1}{2}} - ((1+\gamma)h_x^2h_y^2 - 2a\tau h_y^2\gamma)u_{i,N_y}^{k+\frac{1}{2}} - a\tau h_y^2\gamma u_{i+1,N_y}^{k+\frac{1}{2}} - 2a\tau h_x^2h_y\psi_1(x_i, t^{k+1}) = 2a\tau h_x^2u_{i,N_y-1}^{k+1} - (2a\tau h_x^2 + (1+\gamma)h_x^2h_y^2)u_{i,N_y}^{k+1}$$

Двухточечная аппроксимация первого порядка

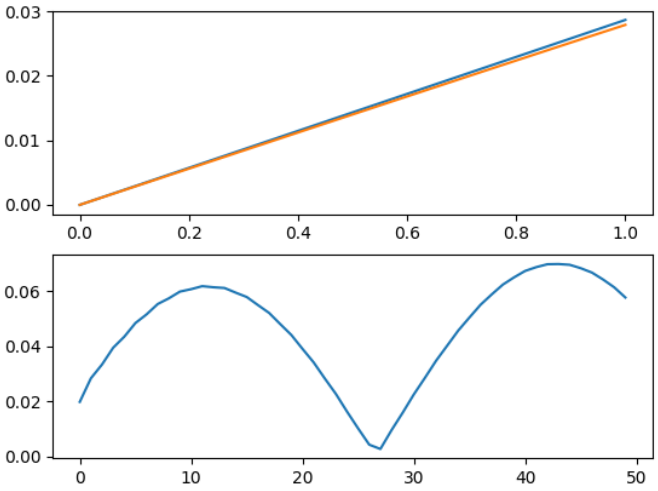
Впрочем можно аппроксировать граничное условие в обоих случаях двухточечной аппроксимацией первого порядка:

$$\frac{u_{i,N_y}^{k+1} - u_{i,N_y-1}^{k+1}}{h_y} = \psi_1(x_i, t^{k+1})$$

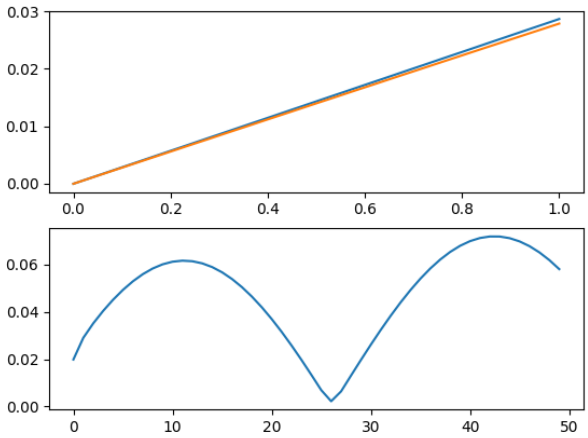
Тогда очевидны формулы для определения значений функции при $y = l_y$ в обоих направлениях прогонки.

3. Результат

1) Метод Переменных направлений



2) Метод Дробных шагов



4. Вывод

Используя схемы переменных направлений и дробных шагов, научился решать двумерную начально-краевую задачу для дифференциального уравнения параболического типа. Вычислил погрешности в различные моменты времени и исследовал зависимость погрешность от различных параметров.

5. Код

```
import numpy as np
from matplotlib import pyplot as plt

import warnings
warnings.filterwarnings("ignore")

def plotSlice(f, X, y, t):
    plt.subplot(2, 1, 1)
    plt.plot(X, f(X, y, t))
    plt.grid

def showPostProcess(t, y):
    i = 2
    X = np.linspace(0, lx, nx)
    plotSlice(fResult, X, y * hy, ht * t * 2)
    plt.subplot(2, 1, 1)
    plt.plot(X, U[i*t, :, y])

def Error(U, fR):
    i = 2
    X = np.linspace(0, lx, nx)
    Y = np.linspace(0, ly, ny)
    T = list(range(0, int(U.shape[0]/2)))
    plt.subplot(2, 1, 2)
    XX, YY = np.meshgrid(X, Y)
```

```
    plt.plot(T, list(map(lambda t: np.max(np.abs(U[i*t] - np.transpose(fR(XX, YY, ht * t * i)))), T)))

def gridFun(iCond, bCond):
    xCond, yCond = bCond
    xlCond, xrCond = xCond
    ylCond, yrCond = yCond
    tlCond, trCond = iCond
    return np.zeros((trCond, xrCond, yrCond))

def parabolicEquation2D(iCond, bCond, method):

    def init():
        for m in range(nx):
            for n in range(ny):
                U[0][m][n] = uxy0(m * hx, n * hy)

    def solve():
        for k in range(0, nt - i, i):
            for n in range(1, ny - 1):
                A1 = getA1(k, n)
                B1 = getB1(k, n)
                U[int(k + i / 2), :, n] = np.linalg.solve(A1, B1)[:, 0]
            for m in range(1, nx - 1):
                A2 = getA2(k, m)
                B2 = getB2(k, m)
                U[k + i, m, :] = np.linalg.solve(A2, B2)[:, 0]
            for n in range(0, ny):
                U[k + i][0][n] = (u0yt(n * hy, (k + i) * ht) - c1 * U[k + i][1][n]) / b1
                U[k + i][nx - 1][n] = (ulyt(n * hy, (k + i) * ht) - aN1 * U[k + i][nx - 2][n]) / bN1

    def FractionSteps():

        aj1 = -a/hx**2
        bj1 = 1/ht/2 + 2*a/hx**2
        cj1 = -a/hx**2

        aj2 = -b/hy**2
        bj2 = 1/ht/2 + 2*b/hy**2
        cj2 = -b/hy**2

        def gA1(k, n):
            aa = np.zeros((nx, nx))
            aa[0][0] = b1
            aa[0][1] = c1
            for j in range(1, nx - 1):
                aa[j][j - 1] = aj1
                aa[j][j] = bj1
                aa[j][j + 1] = cj1
```

```
            aa[nx - 1][nx - 2] = aN1
            aa[nx - 1][nx - 1] = bN1
            return aa

        def gB1(k, n):
            bb = np.zeros((nx, 1))
            bb[0][0] = u0yt(n * hy, ht * (k + i))
            bb[nx - 1][0] = ulyt(n * hy, ht * (k + i))
            for m in range(1, nx - 1):
                bb[m][0] = f(m * hx, n * hy, k * ht)/2 + U[k][m][n] / ht / 2
            return bb

        def gA2(k, m):
            aa = np.zeros((ny, ny))
            aa[0][0] = b2
            aa[0][1] = c2
            for j in range(1, ny - 1):
                aa[j][j - 1] = aj2
                aa[j][j] = bj2
                aa[j][j + 1] = cj2
            aa[ny - 1][ny - 2] = aN2
                  aa[ny - 1][ny - 1] = bN2
                  return aa

              def gB2(k, m):
                  bb = np.zeros((ny, 1))
                  bb[0][0] = ux0t(m * hx, ht * (k + i))
                  bb[ny - 1][0] = uxlt(m * hx, ht * (k + i))
                  for n in range(1, ny - 1):
                      bb[n][0] = f(m * hx, n * hy, (k + i) * ht)/2 + U[int(k +
   i/2)][m][n] / ht / 2
                  return bb
              return (gA1, gB1), (gA2, gB2)

          def AlternatingDirection():

              aj1 = -a / hx ** 2
              bj1 = 1 / ht + 2 * a / hx ** 2
              cj1 = -a / hx ** 2

              aj2 = -b / hy ** 2
              bj2 = 1 / ht + 2 * b / hy ** 2
              cj2 = -b / hy ** 2

              def gA1(k, n):
                  aa = np.zeros((nx, nx))
                  aa[0][0] = b1
                  aa[0][1] = c1
                  for j in range(1, nx - 1):
                      aa[j][j - 1] = aj1
                      aa[j][j] = bj1
```

```
                    aa[j][j + 1] = cj1
                aa[nx - 1][nx - 2] = aN1
                aa[nx - 1][nx - 1] = bN1
                return aa

            def gB1(k, n):
                bb = np.zeros((nx, 1))
                bb[0][0] = u0yt(n * hy, ht * (k + i))
                bb[nx - 1][0] = ulyt(n * hy, ht * (k + i))
                for m in range(1, nx - 1):
                    bb[m][0] = f(m * hx, n * hy, int(k + i/2) * ht) + (1 / ht - 2*b/ hy**2) * U[k][m][n] + (b/hy**2) * (U[k][m][n + 1] + U[k][m][n - 1])
                return bb

            def gA2(k, m):
                aa = np.zeros((ny, ny))
                aa[0][0] = b2
                aa[0][1] = c2
                for j in range(1, ny - 1):
                    aa[j][j - 1] = aj2
                    aa[j][j] = bj2
                    aa[j][j + 1] = cj2
                aa[ny - 1][ny - 2] = aN2
                aa[ny - 1][ny - 1] = bN2
                return aa

            def gB2(k, m):
                bb = np.zeros((ny, 1))
                bb[0][0] = ux0t(m * hx, ht * (k + i))
                bb[ny - 1][0] = uxlt(m * hx, ht * (k + i))
                for n in range(1, ny - 1):
                    bb[n][0] = f(m * hx, n * hy, int(k + i/2) * ht) + (1 / ht - 2*a/ hx**2) * U[int(k + i/2)][m][n] + (a/hx**2) * (U[int(k + i/2)][m + 1][n] + U[int(k + i/2)][m - 1][n])
                return bb

            return (gA1, gB1), (gA2, gB2)
        if method == 1:
            (getA1, getB1), (getA2, getB2) = AlternatingDirection()
        elif method == 2:
            (getA1, getB1), (getA2, getB2) = FractionSteps()
        else:
            pass
        b1 = 1
        c1 = 0
        b2 = 1
        c2 = 0

        aN1 = -betta1 / hx
        bN1 = alpha1 + betta1 / hx
        aN2 = -betta2 / hy
```

```
177.         bN2 = alpha2 + betta2 / hy
178.
179.         uxy0 = iCond
180.         xCond, yCond = bCond
181.         u0yt, ulyt = xCond
182.         ux0t, uxlt = yCond
183.         U = gridFun((0, nt), ((0, nx), (0, ny)))
184.         i = 2
185.         init()
186.         solve()
187.         return U
188.
189.     def solver():
190.         global a, b, nx, ny, nt, hx, hy, ht, U, tt, yy
191.         i = 2
192.         a = a
193.         b = b
194.         nx = nx
195.         ny = ny
196.         nt = 2 * nt
197.         tt = tt
198.         yy = yy
199.         hx = lx / nx
200.         hy = ly / ny
201.         ht = lt / nt
202.         nx += 1
203.         ny += 1
204.         nt += i
205.         U = []
206.
207.     a = 1
208.     b = 1
209.
210.     lx = 1
211.     ly = 1
212.     lt = 5
213.     omega = 1.5
214.     nx = 100
215.     ny = 100
216.     nt = 100
217.     eps = 0.001
218.
219.     u0yt = lambda y, t: 0
220.     ulyt = lambda y, t: 0
221.     ux0t = lambda x, t: 0
222.     uxlt = lambda x, t: 0
223.     uxy0 = lambda x, y: x*y
224.     f = lambda x, y, t: -x*y*np.sin(t)
225.
226.     fResult = lambda x, y, t: x*y*np.cos(t)
227.
```

```
    alpha1 = 1
    betta1 = -1
    alpha2 = 1
    betta2 = -1

    hx = lx / nx
    hy = ly / ny
    ht = lt / nt

    tt = 3
    yy = 3
    U = []

    U = parabolicEquation2D(uxy0, ((ux0t, uxlt), (u0yt, ulyt)), 1)
    showPostProcess(tt, yy)
    Error(U, fResult)
    plt.show()

    U = parabolicEquation2D(uxy0, ((ux0t, uxlt), (u0yt, ulyt)), 2)
    showPostProcess(tt, yy)
    Error(U, fResult)
    plt.show()
```