Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №6 по курсу «Численные методы»

Студент:	Молчанов Владислав
Группа:	М8О-408Б-20
Преподаватель:	Пивоваров Д.Е.

Москва, 2023

Задание: 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 $u(x, t)$. Исследовать зависимость погрешности от сеточных параметров τ и h.

Вариант: 16

Уравнение:

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

$$\begin{cases} u'_x(0,\ t) = cos(2t) \\ u_x(\pi/2,\ t) = 0 \\ u(x,\ 0) = \psi_1(x) = e^{-x}cos(x) \\ u_t(x, 0) = \psi_2(x) = 0 \end{cases}$$

Аналитическое решение:

$$u(x,t) = e^{-x}\cos x \cos 2t$$

Явная и неявная конечно-разностные схемы представляют собой системы уравнений, краевые решения которых заполняются из начальных данных, а значения в середине заполняются по средству вычисления уравнений с одной или несколькими неизвестными.

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

Явная схема:

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

Схема Кранка Николсона подразумевает объединение в себе двух предыдущих схем, в следствии чего при подборе необходимого коэффициента, достигается наименьшая погрешность.

Неявная схема:

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

Апроксимация первого порядка

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def tma(a, b, c, d):
    size = len(a)
    p, q = [], []
    p.append(-c[0] / b[0])
    q.append(d[0] / b[0])

    for i in range(1, size):
        p_tmp = -c[i] / (b[i] + a[i] * p[i - 1])
        q_tmp = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
        p.append(p_tmp)
        q.append(q_tmp)
```

```
    x = [0 for _ in range(size)]
    x[size - 1] = q[size - 1]

    for i in range(size - 2, -1, -1):
        x[i] = p[i] * x[i + 1] + q[i]

    return x

class Data:
    def __init__(self, args):
        self.a = args['a']
        self.b = args['b']
        self.c = args['c']
        self.d = args['d']
        self.l = args['l']
        self.f = args['f']
        self.alpha = args['alpha']
        self.beta = args['beta']
        self.gamma = args['gamma']
        self.delta = args['delta']
        self.psi1 = args['psi1']
        self.psi2 = args['psi2']
        self.psi1_dir1 = args['psi1_dir1']
        self.psi1_dir2 = args['psi1_dir2']
        self.phi0 = args['phi0']
        self.phi1 = args['phi1']
        self.bound_type = args['bound_type']
        self.approximation = args['approximation']
        self.solution = args['solution']


class HyperbolicSolver:
    def __init__(self, args, N, K, T):
        self.data = Data(args)
        self.h = self.data.l / N
        self.tau = T / K
        self.sigma = (self.tau ** 2) / (self.h ** 2)

    def analyticSolve(self, N, K, T):
        self.h = self.data.l / N
        self.tau = T / K
        self.sigma = (self.tau ** 2) / (self.h ** 2)
        u = np.zeros((K, N))
        for k in range(K):
            for j in range(N):
                u[k][j] = self.data.solution(j * self.h, k * self.tau)
        return u

    def calculate(self, N, K):
        u = np.zeros((K, N))

        for j in range(0, N - 1):
            x = j * self.h
            u[0][j] = self.data.psi1(x)

            if self.data.approximation == 'p1':
                u[1][j] = self.data.psi1(x) + self.data.psi2(x) * self.tau +
                          (self.tau ** 2 / 2)
            elif self.data.approximation == 'p2':
                u[1][j] = self.data.psi1(x) + self.data.psi2(x) * self.tau +
                          (self.data.psi1_dir2(x) + self.data.b * self.data.
                           self.data.c * self.data.psi1(x) + self.data.f())
```

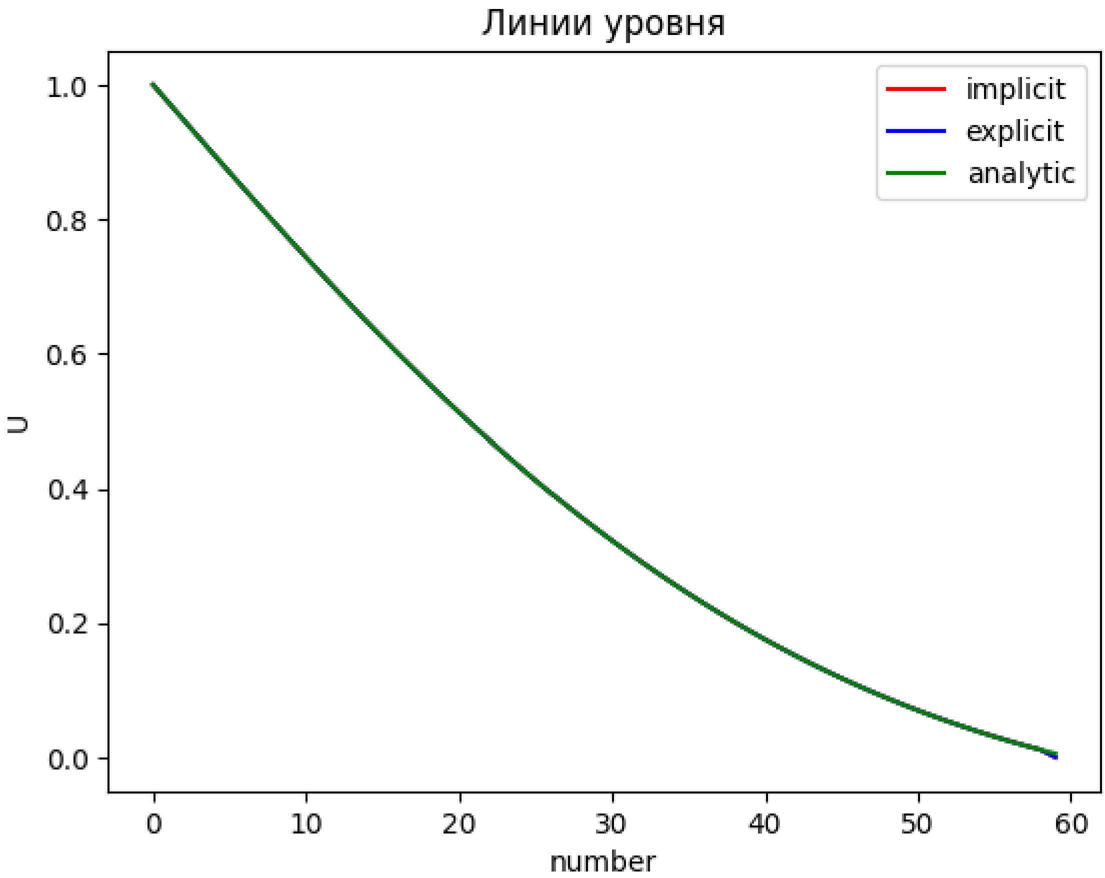
```
        return u

    def implicit_solver(self, N, K, T):
        u = self.calculate(N, K)

        a = np.zeros(N)
        b = np.zeros(N)
        c = np.zeros(N)
        d = np.zeros(N)

        for k in range(2, K):
            for j in range(1, N):
                a[j] = self.sigma
                b[j] = -(1 + 2 * self.sigma)
                c[j] = self.sigma
                d[j] = -2 * u[k - 1][j] + u[k - 2][j]

            if self.data.bound_type == 'a1p2':
                b[0] = self.data.alpha / self.h / (self.data.beta - self.dat
                c[0] = 1
                d[0] = 1 / (self.data.beta - self.data.alpha / self.h) * sel
                a[-1] = -self.data.gamma / self.h / (self.data.delta + self.
                d[-1] = 1 / (self.data.delta + self.data.gamma / self.h) * s

            elif self.data.bound_type == 'a2p3':
                k1 = 2 * self.h * self.data.beta - 3 * self.data.alpha
                omega = self.tau ** 2 * self.data.b / (2 * self.h)
                xi = self.data.d * self.tau / 2

                b[0] = 4 * self.data.alpha - self.data.alpha / (self.sigma +
                       (1 + xi + 2 * self.sigma - self.data.c * self.tau **
                c[0] = k1 - self.data.alpha * (omega - self.sigma) / (omega
                d[0] = 2 * self.h * self.data.phi0(k * self.tau) + self.data
                a[-1] = -self.data.gamma / (omega - self.sigma) * \
                        (1 + xi + 2 * self.sigma - self.data.c * self.tau **
                d[-1] = 2 * self.h * self.data.phi1(k * self.tau) - self.dat

            elif self.data.bound_type == 'a2p2':
                b[0] = 2 * self.data.a / self.h
                c[0] = -2 * self.data.a / self.h + self.h / self.tau ** 2 -
                       -self.data.d * self.h / (2 * self.tau) + \
                       self.data.beta / self.data.alpha * (2 * self.data.a +
                d[0] = self.h / self.tau ** 2 * (u[k - 2][0] - 2 * u[k - 1][
                       -self.data.d * self.h / (2 * self.tau) * u[k - 2][0]
                       (2 * self.data.a - self.data.b * self.h) / self.data.
                a[-1] = -b[0]
                d[-1] = self.h / self.tau ** 2 * (-u[k - 2][0] + 2 * u[k - 1
                        self.data.d * self.h / (2 * self.tau) * u[k - 2][0]
                        (2 * self.data.a + self.data.b * self.h) / self.data

            u[k] = tma(a, b, c, d)

        return u

    def _left_bound_a1p2(self, u, k, t):
        coeff = self.data.alpha / self.h
        return (-coeff * u[k - 1][1] + self.data.phi0(t)) / (self.data.beta

    def _right_bound_a1p2(self, u, k, t):
        coeff = self.data.gamma / self.h
        return (coeff * u[k - 1][-2] + self.data.phi1(t)) / (self.data.delta

    def _left_bound_a2p2(self, u, k, t):
        n = self.data.c * self.h - 2 * self.data.a / self.h - self.h / self.
```

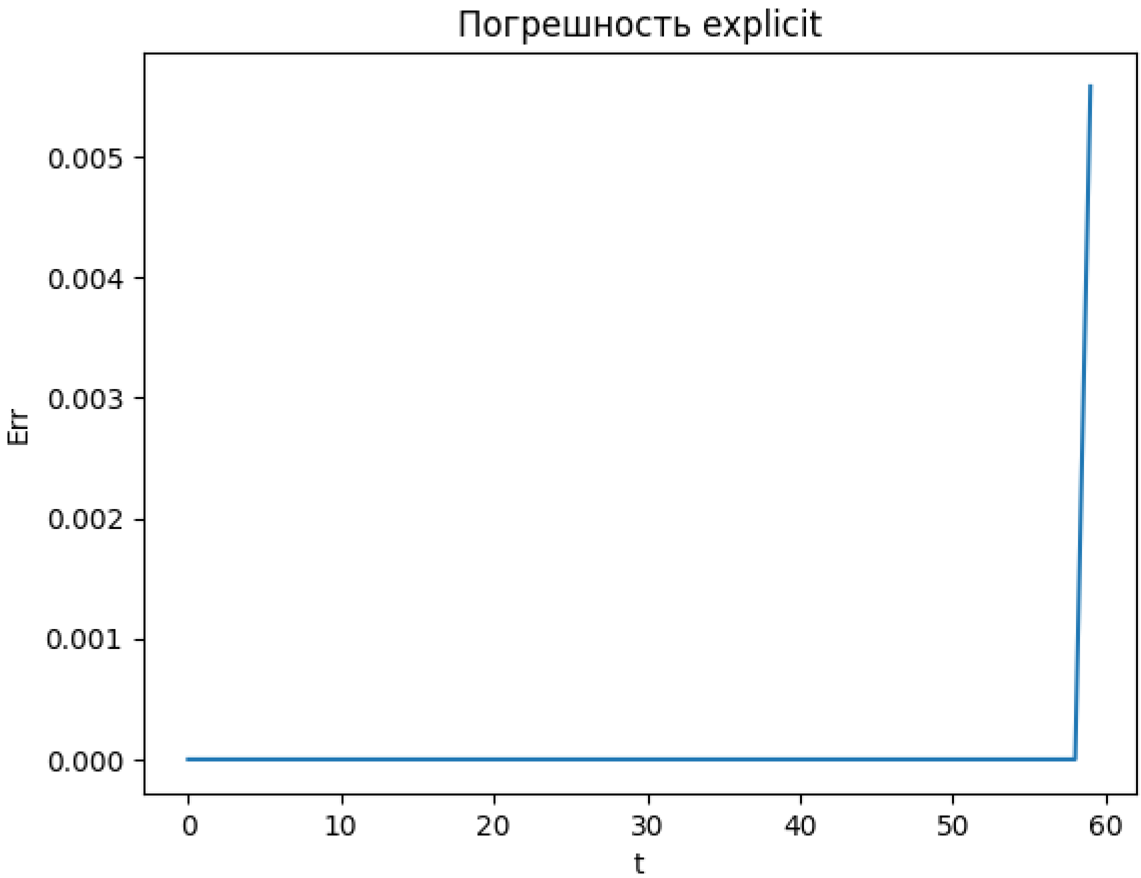
4

```
            (2 * self.tau) + self.data.beta / self.data.alpha * (2 * self.da
        return 1 / n * (- 2 * self.data.a / self.h * u[k][1] +
                        self.h / self.tau ** 2 * (u[k - 2][0] - 2 * u[k - 1]
                        -self.data.d * self.h / (2 * self.tau) * u[k - 2][0]
                        (2 * self.data.a - self.data.b * self.h) / self.data

    def _right_bound_a2p2(self, u, k, t):
        n = -self.data.c * self.h + 2 * self.data.a / self.h + self.h / self
            (2 * self.tau) + self.data.delta / self.data.gamma * (2 * self.d
        return 1 / n * (2 * self.data.a / self.h * u[k][-2] +
                        self.h / self.tau ** 2 * (2 * u[k - 1][-1] - u[k - 2
                        self.data.d * self.h / (2 * self.tau) * u[k - 2][-1]
                        (2 * self.data.a + self.data.b * self.h) / self.data

    def _left_bound_a2p3(self, u, k, t):
        denom = 2 * self.h * self.data.beta - 3 * self.data.alpha
        return self.data.alpha / denom * u[k - 1][2] - 4 * self.data.alpha /
               2 * self.h / denom * self.data.phi0(t)

    def _right_bound_a2p3(self, u, k, t):
        denom = 2 * self.h * self.data.delta + 3 * self.data.gamma
        return 4 * self.data.gamma / denom * u[k - 1][-2] - self.data.gamma
               2 * self.h / denom * self.data.phi1(t)

    def explicit_solver(self, N, K, T):
        global left_bound, right_bound
        u = self.calculate(N, K)

        if self.data.bound_type == 'a1p2':
            left_bound = self._left_bound_a1p2
            right_bound = self._right_bound_a1p2

        elif self.data.bound_type == 'a2p2':
            left_bound = self._left_bound_a2p2
            right_bound = self._right_bound_a2p2

        elif self.data.bound_type == 'a2p3':
            left_bound = self._left_bound_a2p3
            right_bound = self._right_bound_a2p3

        for k in range(2, K):
            t = k * self.tau
            for j in range(1, N - 1):
                quadr = self.tau ** 2
                tmp1 = self.sigma + self.data.b * quadr / (2 * self.h)
                tmp2 = self.sigma - self.data.b * quadr / (2 * self.h)
                u[k][j] = u[k - 1][j + 1] * tmp1 + \
                    u[k - 1][j] * (-2 * self.sigma + 2 + self.data.c * quadr
                    u[k - 1][j - 1] * tmp2 - u[k - 2][j] + quadr * self.data
            u[k][0] = left_bound(u, k, t)
            u[k][-1] = right_bound(u, k, t)

        return u


def presontation(dict_, time=0):
    fig = plt.figure()
    plt.title('Линии уровня')
    plt.plot(dict_['implicit'][time], color='r', label='implicit')
    plt.plot(dict_['explicit'][time], color='b', label='explicit')
    plt.plot(dict_['analytic'][time], color='g', label='analytic')
    plt.legend(loc='best')
    plt.ylabel('U')
    plt.xlabel('number')
```

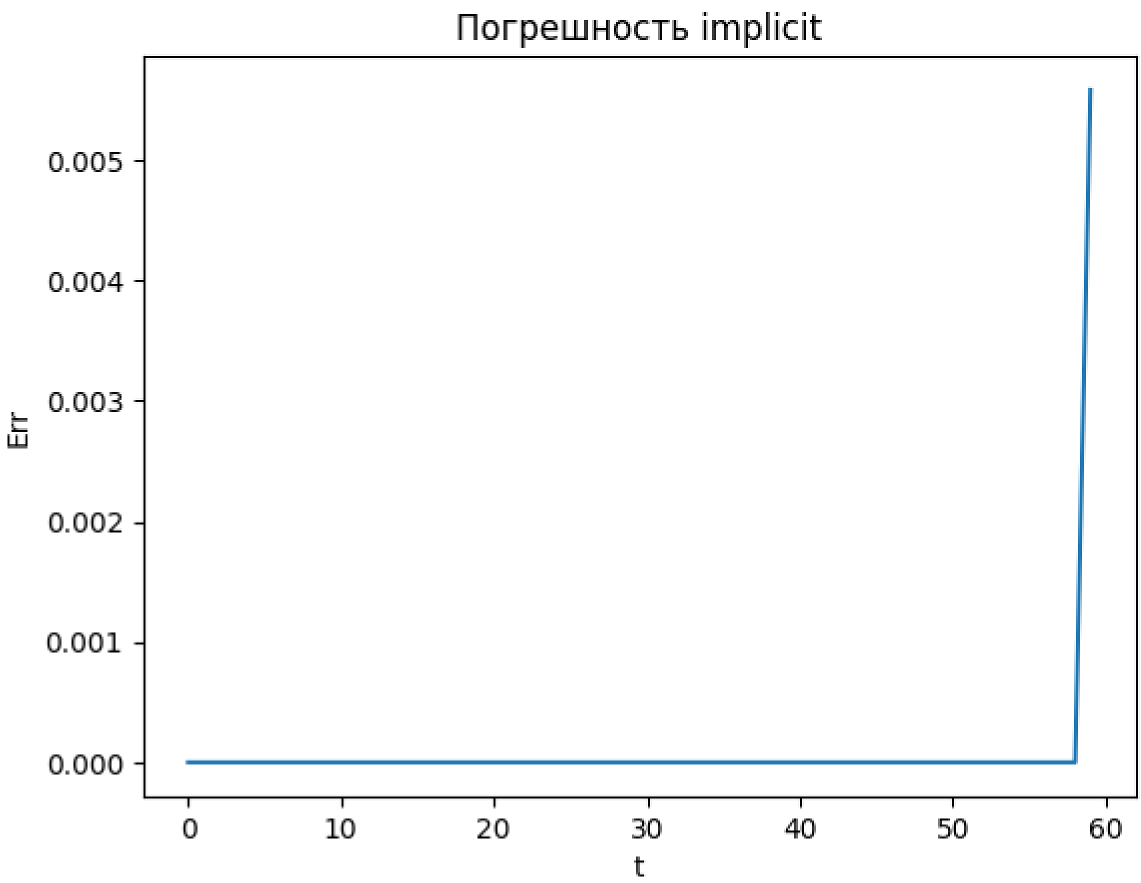
5

```
    plt.show()

    plt.title('Погрешность explicit')
    plt.plot(abs(dict_['explicit'][time] - dict_['analytic'][time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()

    plt.title('Погрешность implicit')
    plt.plot(abs(dict_['implicit'][time] - dict_['analytic'][time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()


data = {'N': 60, 'K': 100, 'T': 1}
N, K, T = int(data['N']), int(data['K']), int(data['T'])

args = {
    'a': 1,
    'b': 2,
    'c': -2,
    'd': 0,
    'l': np.pi / 2,
    'f': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'psi1': lambda x: np.exp(-x) * np.cos(x),
    'psi2': lambda x: 0,
    'psi1_dir1': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'psi1_dir2': lambda x: 2 * np.exp(-x) * np.sin(x),
    'phi0': lambda t: np.cos(2 * t),
    'phi1': lambda t: 0,
    'bound_type': 'a1p2',
    'approximation': 'p1',
    'solution': lambda x, t: np.exp(-x) * np.cos(x) * np.cos(2 * t),
}

solver = HyperbolicSolver(args, N, K, T)

ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}

presontation(ans)
```


Линии уровня
implicit
explicit
analytic
U
number
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60


Погрешность explicit
Err
t
0.005
0.004
0.003
0.002
0.001
0.000
0
10
20
30
40
50
60

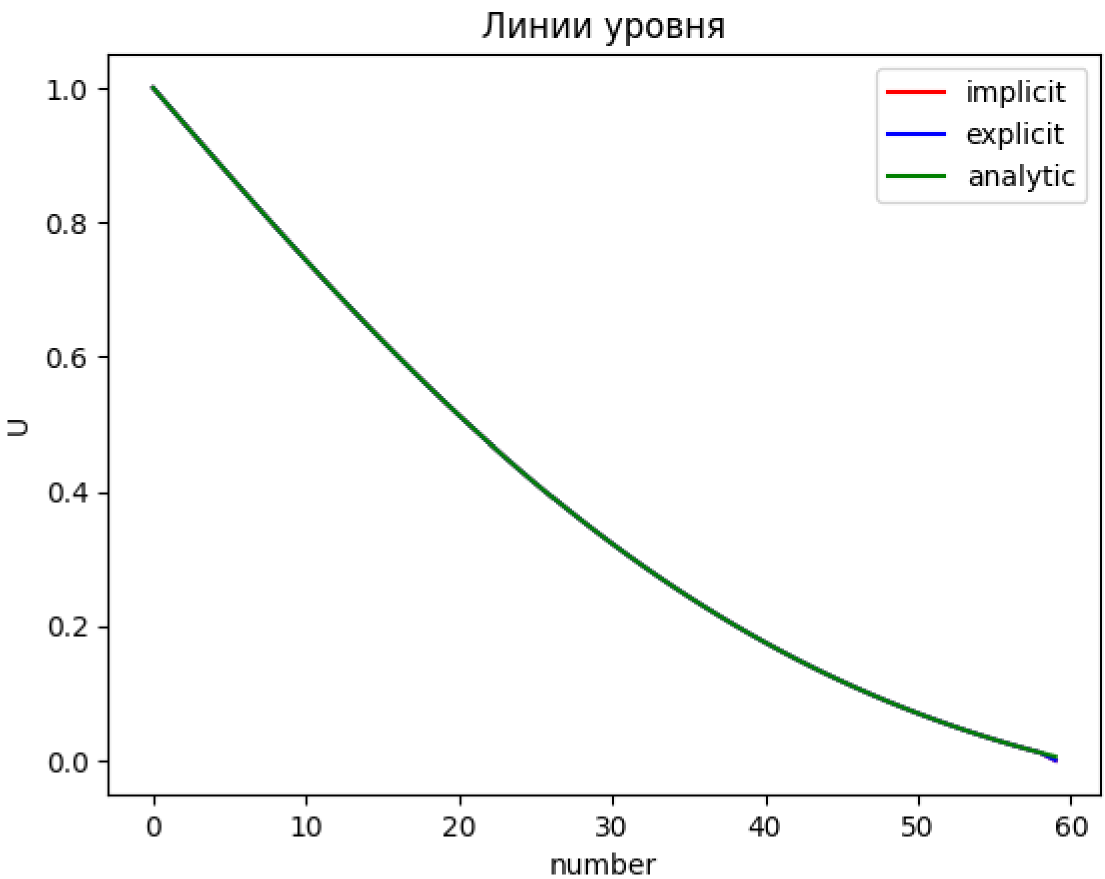
7

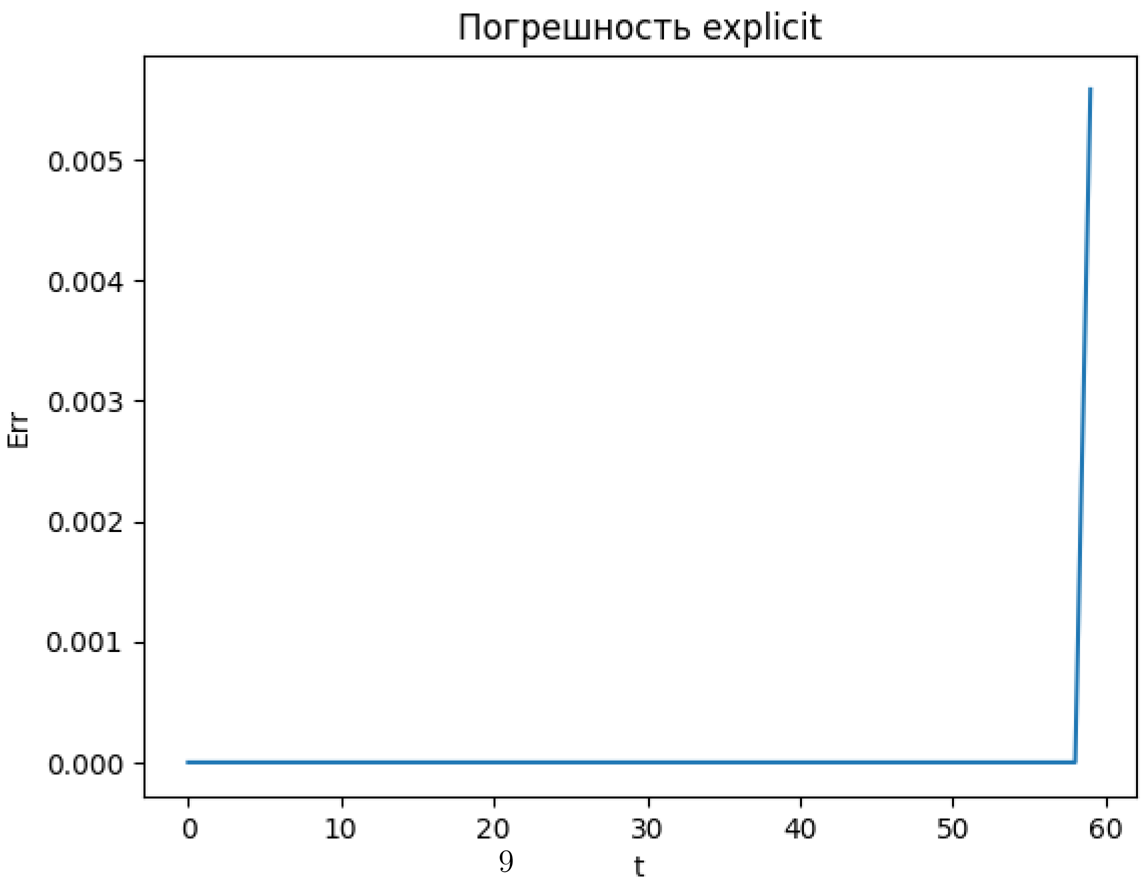


Апроксимация 3-х точечная второго порядка

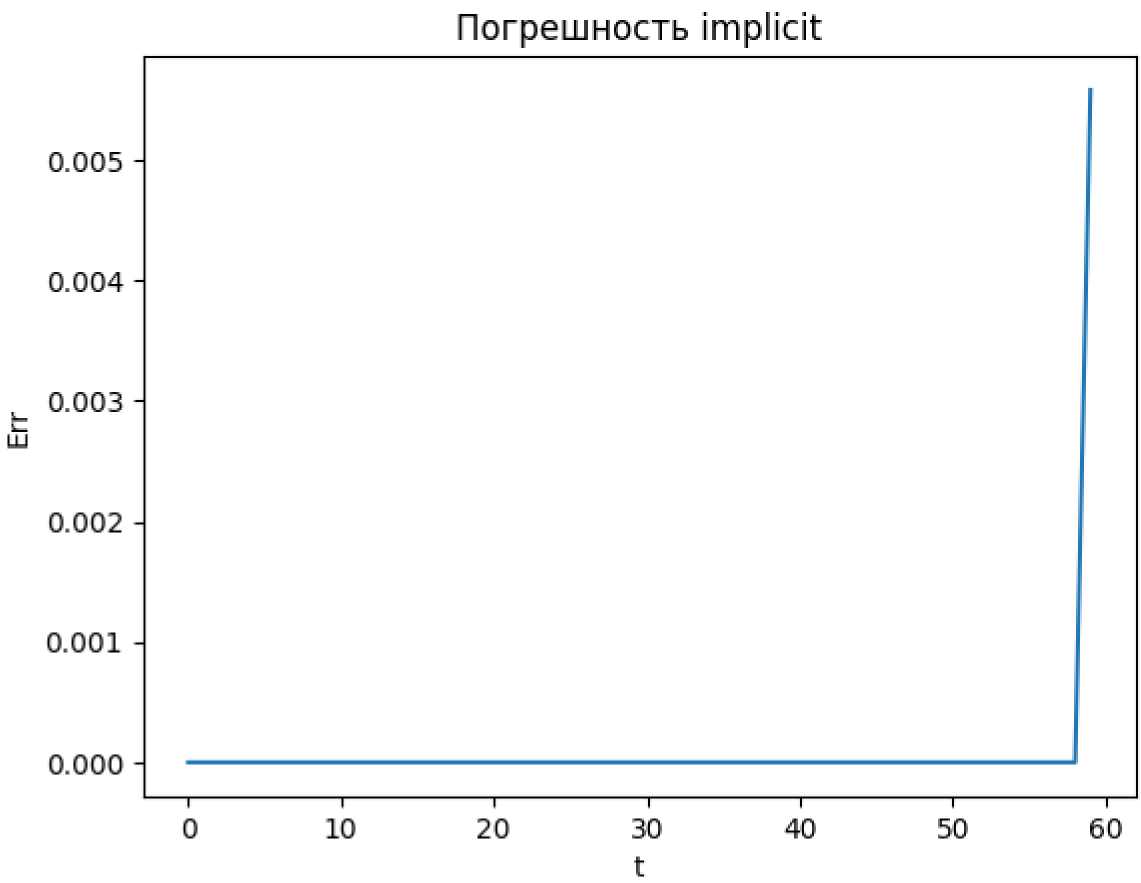
In [2]:
```
data = {'N': 60, 'K': 100, 'T': 1}
N, K, T = int(data['N']), int(data['K']), int(data['T'])

args = {
    'a': 1,
    'b': 2,
    'c': -2,
    'd': 0,
    'l': np.pi / 2,
    'f': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'psi1': lambda x: np.exp(-x) * np.cos(x),
    'psi2': lambda x: 0,
    'psi1_dir1': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'psi1_dir2': lambda x: 2 * np.exp(-x) * np.sin(x),
    'phi0': lambda t: np.cos(2 * t),
    'phi1': lambda t: 0,
    'bound_type': 'a2p3',
    'approximation': 'p2',
    'solution': lambda x, t: np.exp(-x) * np.cos(x) * np.cos(2 * t),
}

solver = HyperbolicSolver(args, N, K, T)

ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
```

```
}

presontation(ans)
```





9



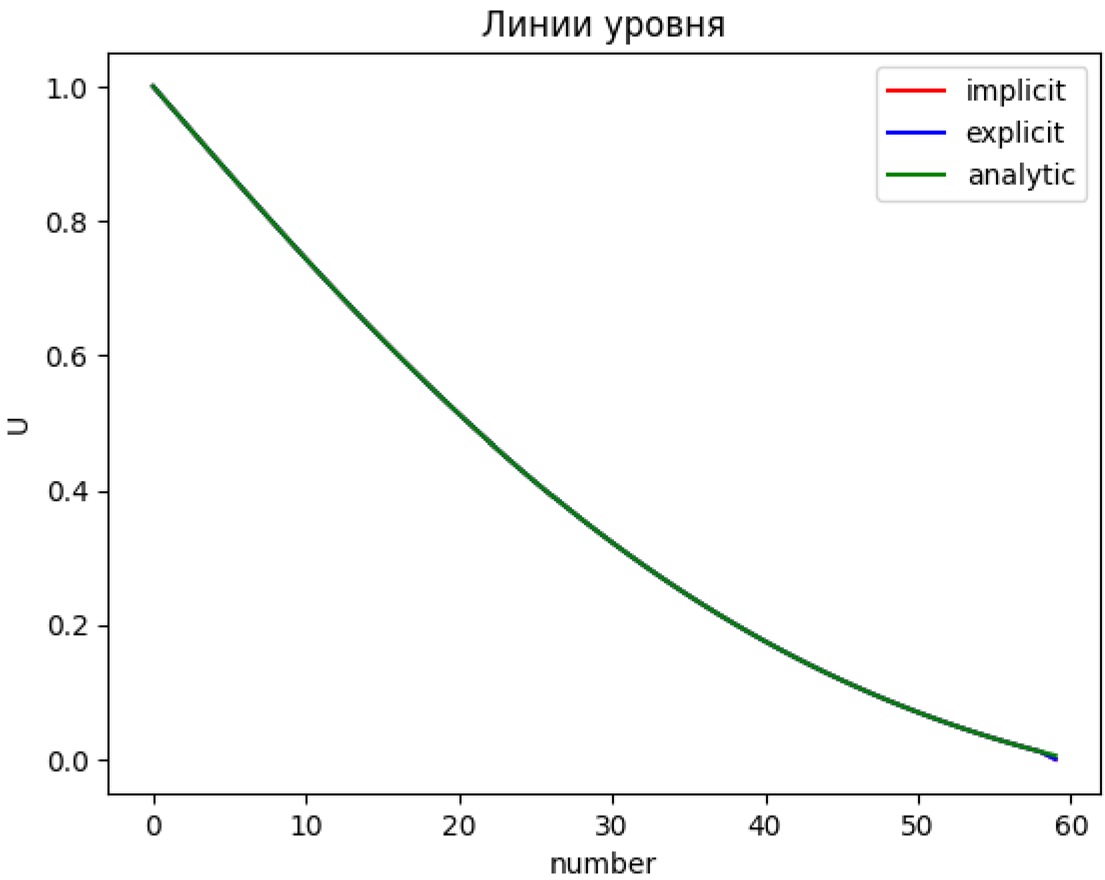
В качестве результата я получил графики линий уровня U. Они наиболее наглядно показывают точность методов, и в каких промежутках какой метод будет эффективен, а какой нет. Также я вывожу графики модуля ошибки каждого метода. Исследование зависимости погрешности от параметров находится в одном файле с исходным кодом.

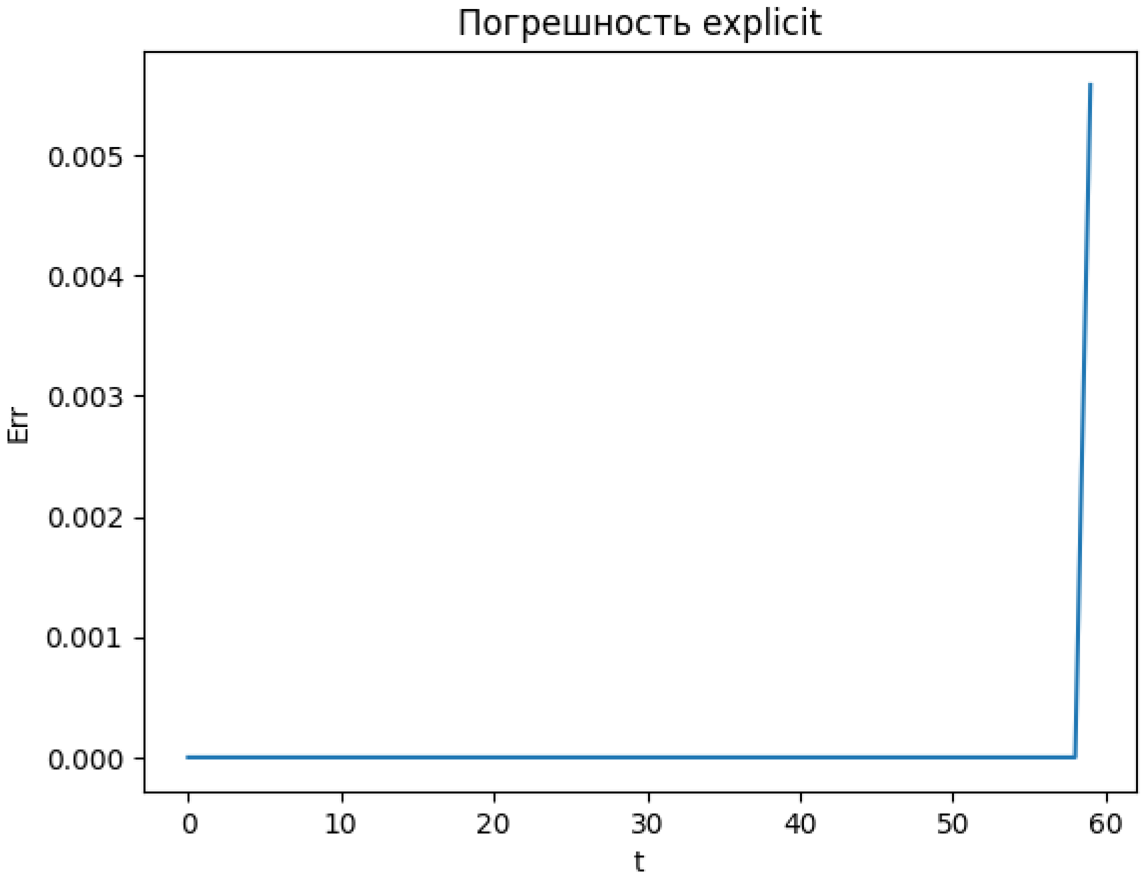
Апроксимация 2-х точечная второго порядка

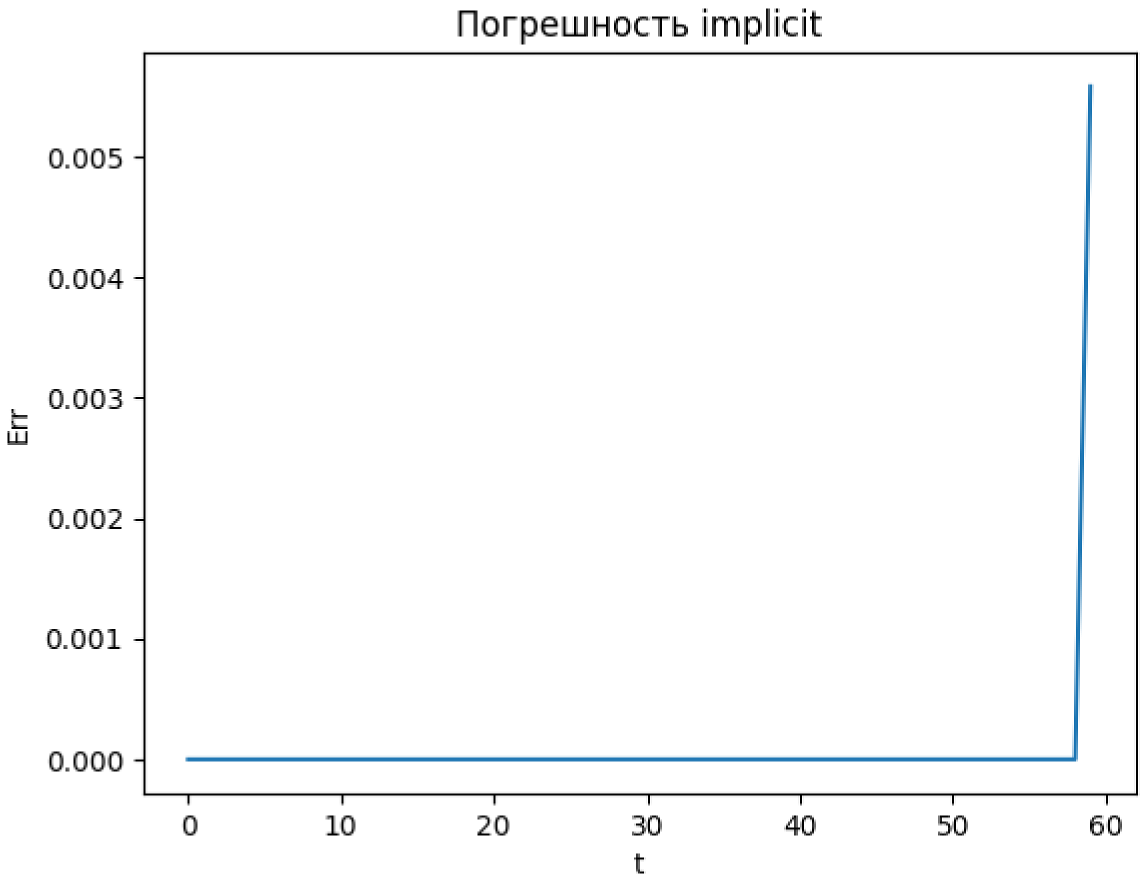
In [3]:
```
data = {'N': 60, 'K': 100, 'T': 1}
N, K, T = int(data['N']), int(data['K']), int(data['T'])

args = {
    'a': 1,
    'b': 2,
    'c': -2,
    'd': 0,
    'l': np.pi / 2,
    'f': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'psi1': lambda x: np.exp(-x) * np.cos(x),
    'psi2': lambda x: 0,
    'psi1_dir1': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'psi1_dir2': lambda x: 2 * np.exp(-x) * np.sin(x),
    'phi0': lambda t: np.cos(2 * t),
    'phi1': lambda t: 0,
    'bound_type': 'a1p2',
    'approximation': 'p2',
    'solution': lambda x, t: np.exp(-x) * np.cos(x) * np.cos(2 * t),
```

10

```
}

solver = HyperbolicSolver(args, N, K, T)

ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}

presontation(ans)
```



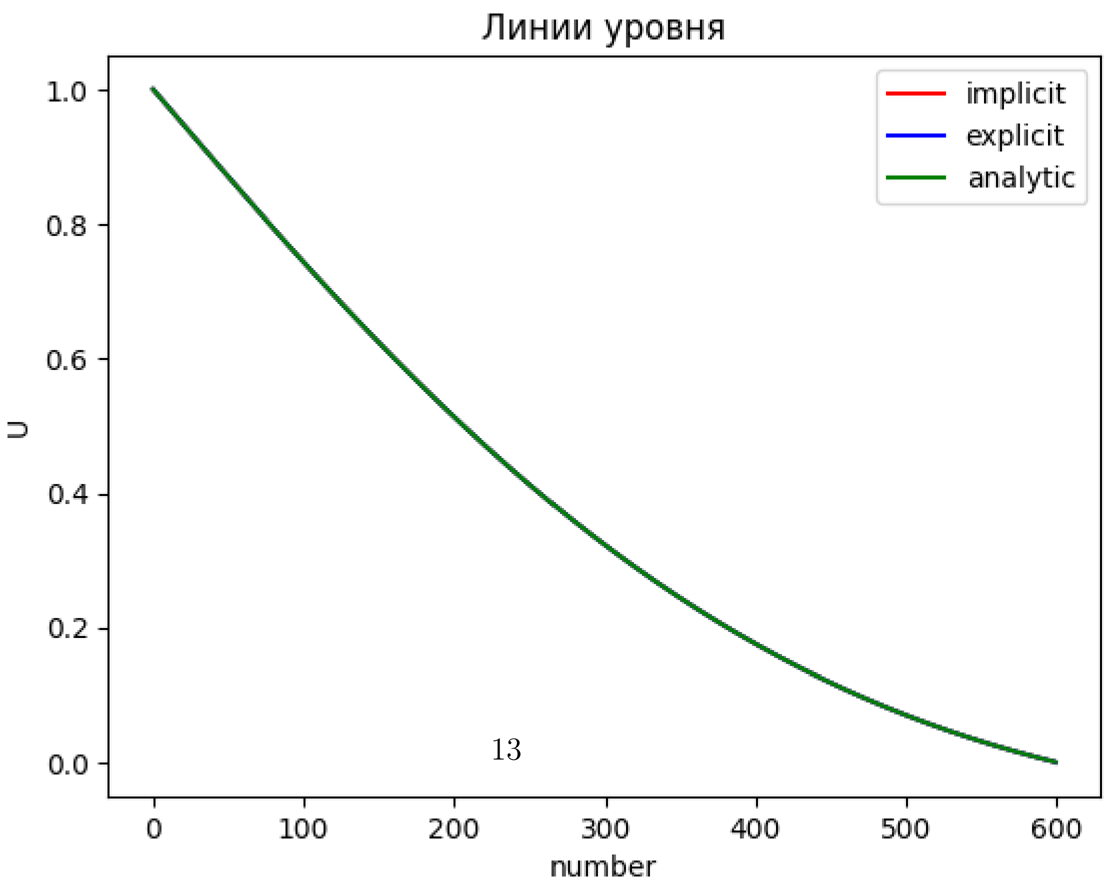


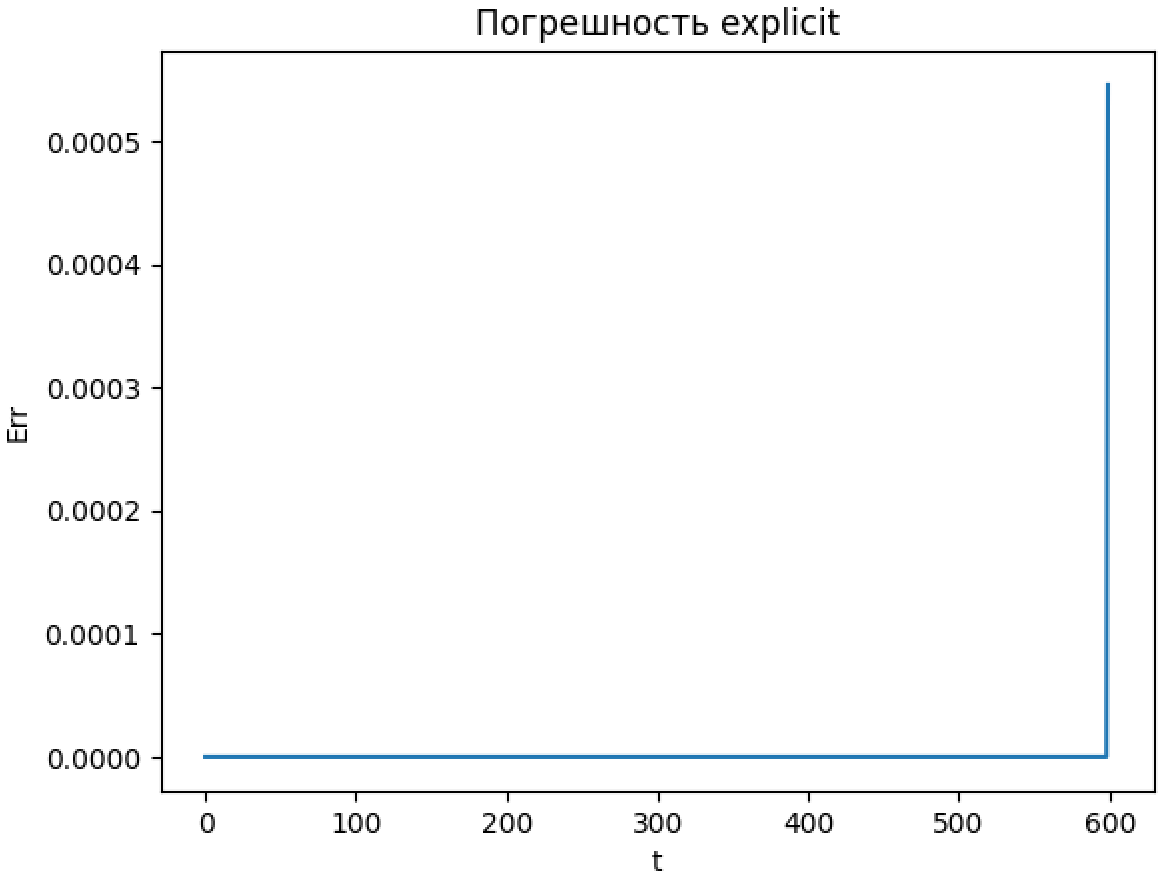


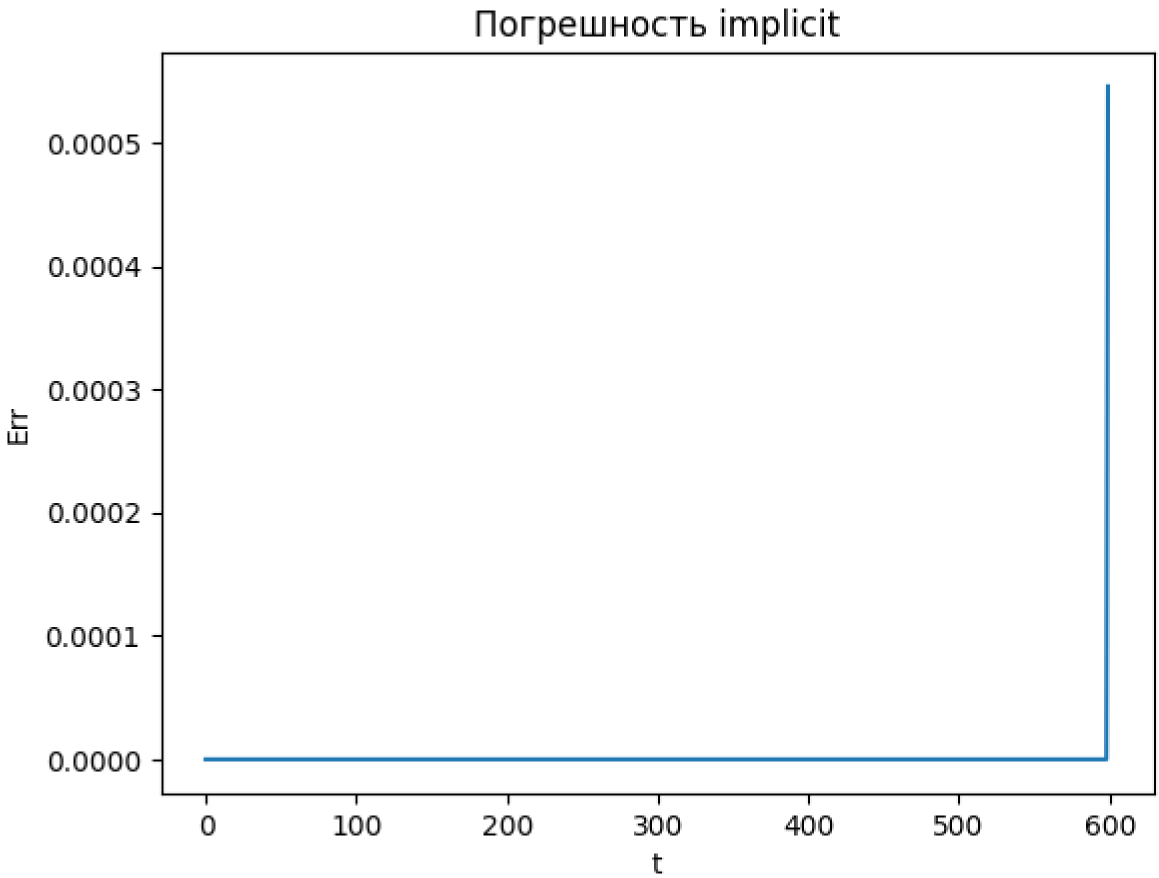
12

Исследование зависимости погрешности от параметров tau и h

In [4]:
```
data = {'N': 600, 'K': 100, 'T': 1}
N, K, T = int(data['N']), int(data['K']), int(data['T'])

args = {
    'a': 1,
    'b': 2,
    'c': -2,
    'd': 0,
    'l': np.pi / 2,
    'f': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'psi1': lambda x: np.exp(-x) * np.cos(x),
    'psi2': lambda x: 0,
    'psi1_dir1': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'psi1_dir2': lambda x: 2 * np.exp(-x) * np.sin(x),
    'phi0': lambda t: np.cos(2 * t),
    'phi1': lambda t: 0,
    'bound_type': 'a1p2',
    'approximation': 'p1',
    'solution': lambda x, t: np.exp(-x) * np.cos(x) * np.cos(2 * t),
}

solver = HyperbolicSolver(args, N, K, T)

ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}

presontation(ans)
```



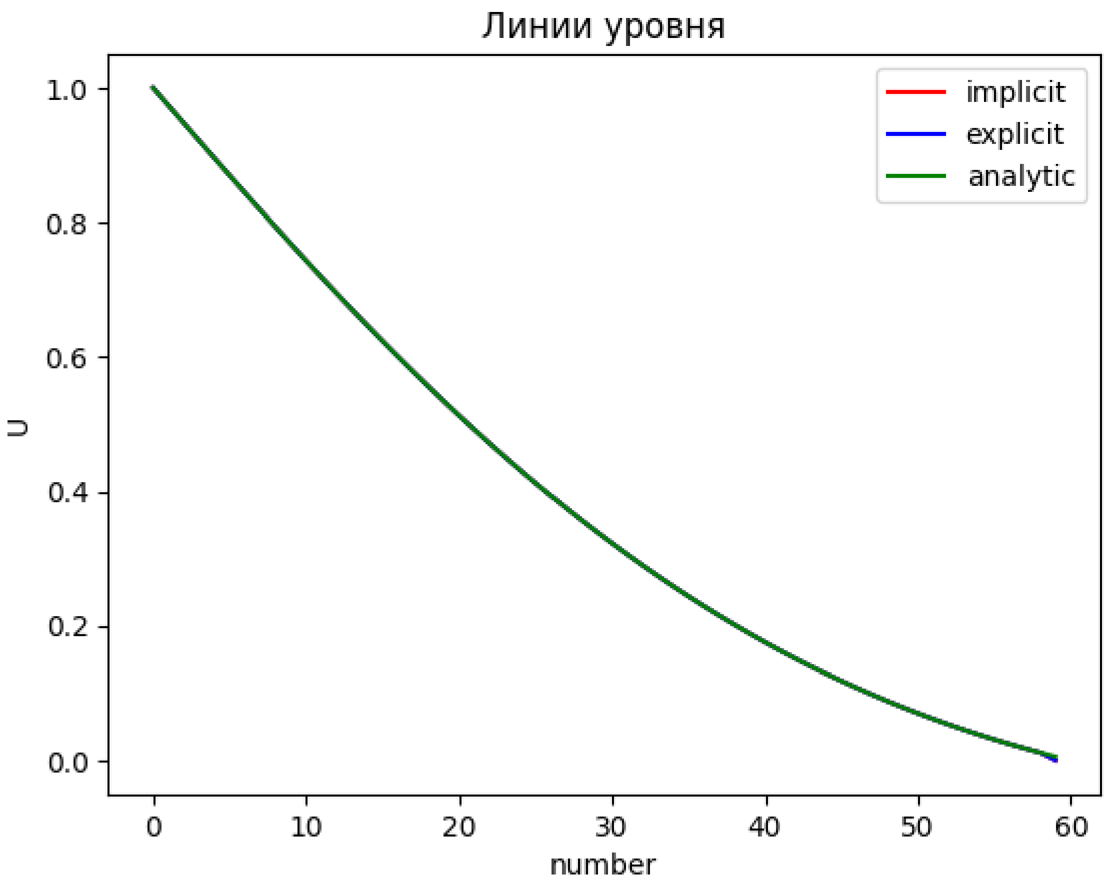




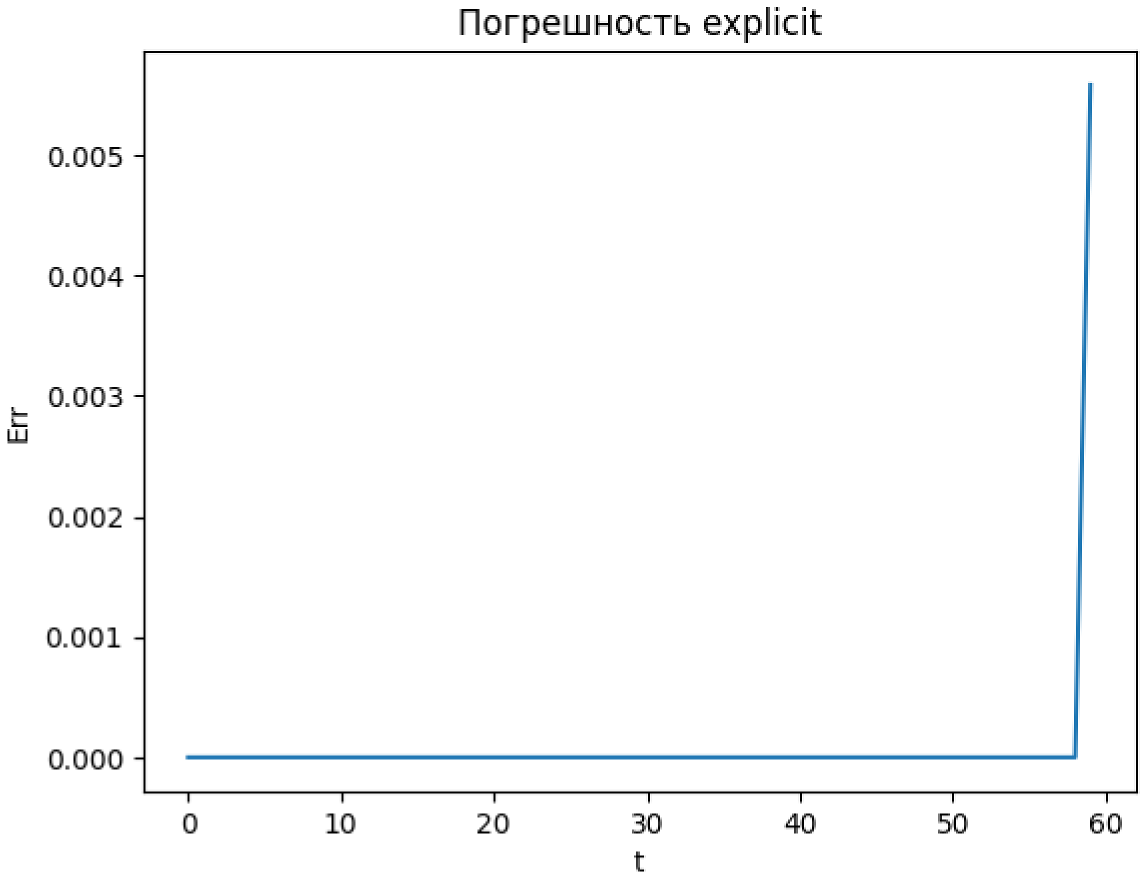
In [5]:

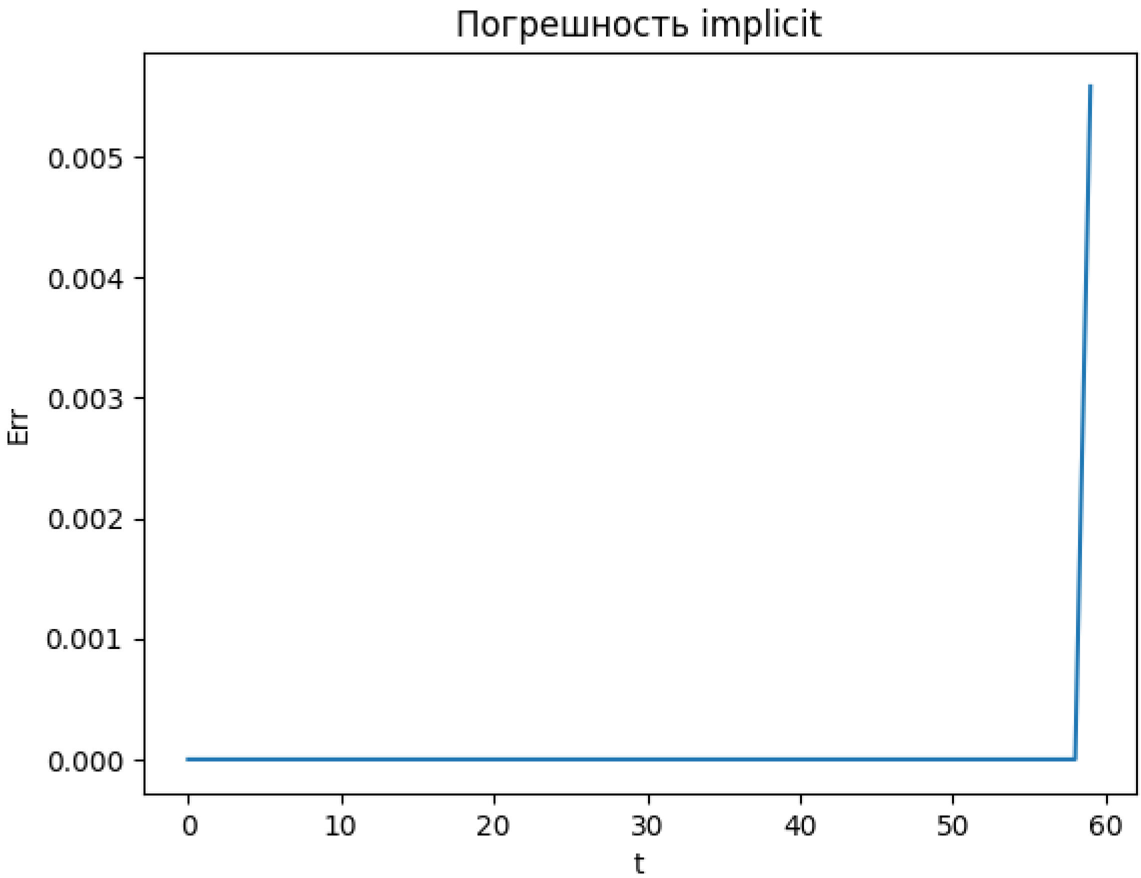
```
data = {'N': 60, 'K': 10000, 'T': 1}
N, K, T = int(data['N']), int(data['K']), int(data['T'])

args = {
    'a': 1,
    'b': 2,
    'c': -2,
```

```
    'd': 0,
    'l': np.pi / 2,
    'f': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'psi1': lambda x: np.exp(-x) * np.cos(x),
    'psi2': lambda x: 0,
    'psi1_dir1': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'psi1_dir2': lambda x: 2 * np.exp(-x) * np.sin(x),
    'phi0': lambda t: np.cos(2 * t),
    'phi1': lambda t: 0,
    'bound_type': 'a1p2',
    'approximation': 'p1',
    'solution': lambda x, t: np.exp(-x) * np.cos(x) * np.cos(2 * t),
}

solver = HyperbolicSolver(args, N, K, T)

ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}

presontation(ans)
```



15





При решение этого типа задач шаг h имеет больший вес при подсчёте погрешности, уменьшив его в сто раз, можно уменьшить погрешность в 10 раз.

Вывод

Схемч крест для решений уравнений гиперболичесого типа имеют высокую точность и, при достаточной мелкости tau, способны достигать настолько маленькую погрешность, что ей можно будет пренебречь при решении реальных задач математической физики.