Лабораторная работа №5 по курсу "Численные методы"

Выполнил студент группы М8О-408Б-20 Меджидли Махмуд.

Преподаватель: Пивоваров Д.Е.

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

Вариант 6

$$\frac{\partial u}{\partial t}=a\frac{\partial^2 u}{\partial x^2}+\cos x(\cos t+\sin t)$$

$$\begin{cases} u(0,\, t) = sint \\ u'_x(\pi/2,\, t) = -sint \\ u(x,\, 0) = 0 \end{cases}$$

Аналитическое решение:

$$u(x,t)=\sin t\cos x$$

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

```
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


def get_zeros(N, K):
    lst = [np.zeros(N) for _ in range(0, 4)]
    lst.append(np.zeros((K, N)))
    return lst


class Data:
    def __init__(self, args):
        self.l = args['l']
        self.f = args['f']
        self.psi = args['psi']
        self.phi0 = args['phi0']
        self.phi1 = args['phi1']
        self.bound_type = args['bound_type']
        self.solve = args['solution']


class ParabolicSolver:
    def __init__(self, args, N, K, T):
```

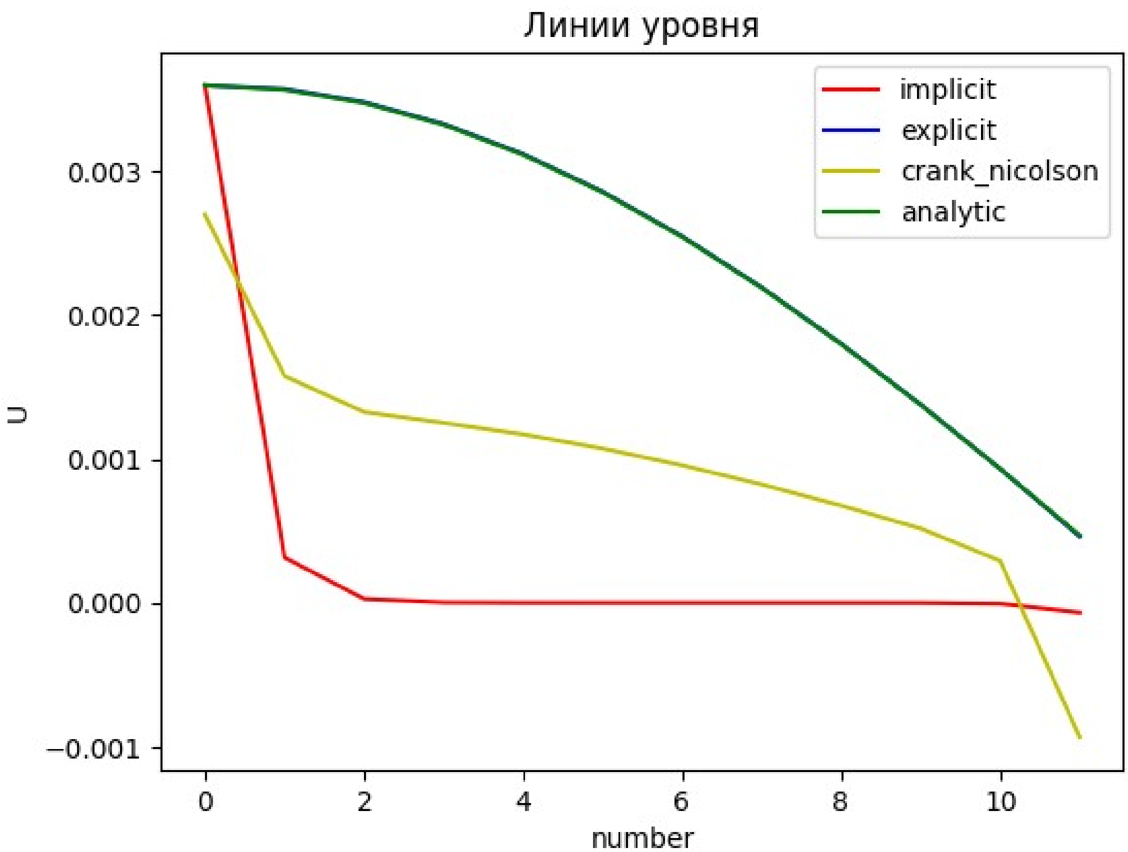
```
        self.alpha = 0
        self.beta = 1
        self.gamma = 1
        self.delta = 0
        self.data = Data(args)
        self.a = 1
        self.b = 0
        self.c = 0
        self.h = self.data.l / N
        self.tau = T / K
        self.sigma = self.a ** 2 * self.tau / (self.h ** 2)

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

    def calculate(self, a, b, c, d, u, k, N, T, K):
        t = np.arange(0, T, T / K)
        for j in range(1, N):
            a[j] = self.sigma
            b[j] = -(1 + 2 * self.sigma)
            c[j] = self.sigma
            d[j] = -u[k][j]

        if self.data.bound_type == 'a1p1':
            a[0] = 0
            b[0] = self.beta - (self.alpha / self.h)
            c[0] = self.alpha / self.h
            d[0] = self.data.phi0(t[k]) / (self.beta - self.alpha /
self.h)
            a[-1] = -self.gamma / self.h
            b[-1] = self.gamma / self.h + self.delta
            c[-1] = 0
            d[-1] = self.data.phi1(t[k]) / (self.gamma / self.h +
self.delta)
        elif self.data.bound_type == 'a1p2':
            a[0] = 0
            b[0] = -(1 + 2 * self.sigma)
            c[0] = self.sigma
            d[0] = -(u[k - 1][0] + self.sigma * self.data.phi0(k *
self.tau)) - \
                   self.tau * self.data.f(0, k * self.tau)
            a[-1] = self.sigma
            b[-1] = -(1 + 2 * self.sigma)
            c[-1] = 0
```

```
            d[-1] = -(u[k - 1][-1] + self.sigma * self.data.phi1(k *
self.tau)) - \
                    self.tau * self.data.f((N - 1) * self.h, k *
self.tau)
        elif self.data.bound_type == 'a1p3':
            a[0] = 0
            b[0] = -(1 + 2 * self.sigma)
            c[0] = self.sigma
            d[0] = -((1 - self.sigma) * u[k - 1][1] + self.sigma / 2 *
u[k - 1][0]) - self.tau \
                   * self.data.f(0, k * self.tau) - self.sigma *
self.data.phi0(
                k * self.tau)
            a[-1] = self.sigma
            b[-1] = -(1 + 2 * self.sigma)
            c[-1] = 0
            d[-1] = self.data.phi1(k * self.tau) + self.data.f((N - 1)
* self.h, k * self.tau) \
                    * self.h / (2 * self.tau) * u[k - 1][-1]

    def implicit_solver(self, N, K, T):
        lst = get_zeros(N, K)
        a = lst[0]
        b = lst[1]
        c = lst[2]
        d = lst[3]
        u = lst[4]

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

        for k in range(1, K):
            self.calculate(a, b, c, d, u, k, N, T, K)
            u[k] = tma(a, b, c, d)

        return u

    def explicit_solver(self, N, K, T):
        u = np.zeros((K, N))
        t = np.arange(0, T, T / K)
        x = np.arange(0, np.pi / 2, np.pi / 2 / N)
        for j in range(1, N - 1):
            u[0][j] = self.data.psi(j * self.h)

        for k in range(1, K):
            for j in range(1, N - 1):
                  u[k][j] = (u[k - 1][j + 1] * (self.a ** 2.0 *
self.tau / self.h ** 2.0)
                           - 2 * u[k - 1][j] * (self.a ** 2.0 *
```

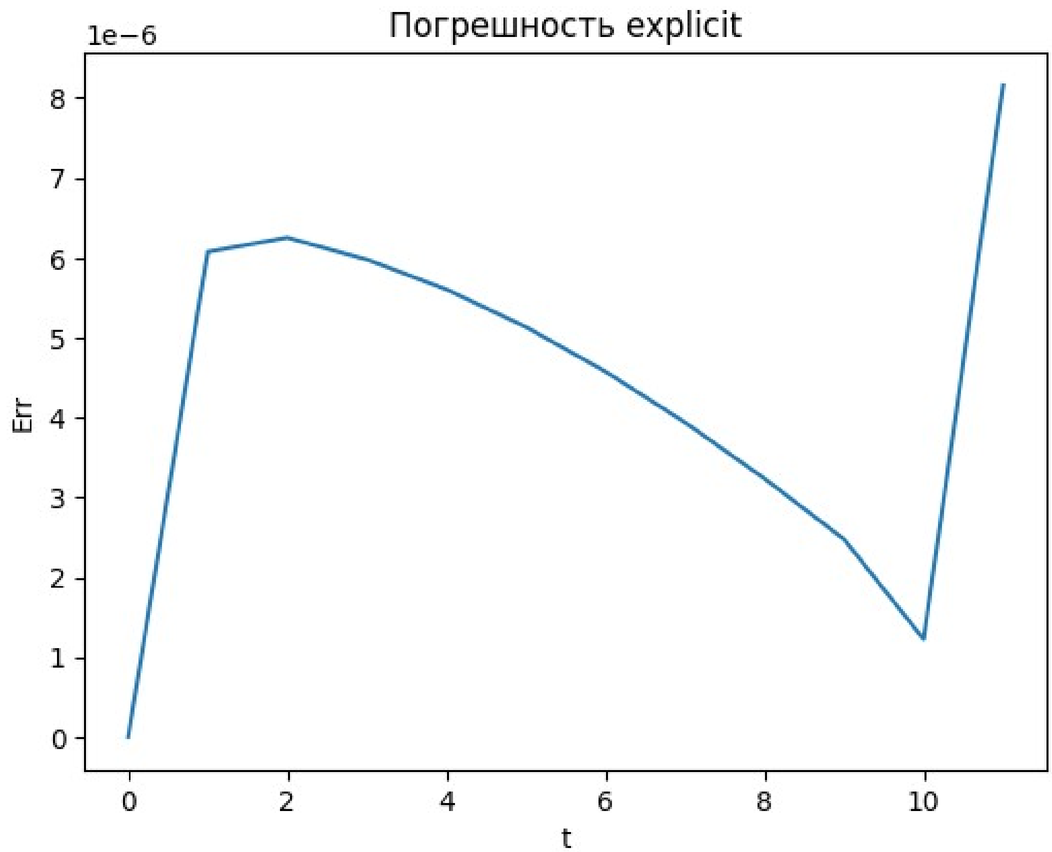
```
self.tau / self.h ** 2.0)
                           + u[k - 1][j - 1] * (self.a ** 2.0 *
self.tau / self.h ** 2.0)
                           + u[k - 1][j]
                           + self.tau * self.data.f(x[j], t[k]))

            if self.data.bound_type == 'a1p1':
                u[k][0] = self.data.phi0(t[k])
                u[k][-1] = (self.data.phi1(t[k]) + self.gamma / self.h
* u[k][-2]) / (self.delta + self.gamma / self.h)
            elif self.data.bound_type == 'a1p2':
                u[k][0] = self.data.phi0(t[k])
                u[k][-1] = (((2.0 * self.gamma * self.a / self.h /
(2.0 * self.a + self.h * self.b)) * u[k][-2] +
                             (self.gamma * self.h / self.tau / (2.0 *
self.a + self.h * self.b)) * u[k - 1][-1] +
                             (self.gamma * self.h * self.c / (2.0 *
self.a + self.h * self.b)) * self.data.f(
                            self.data.l, t[k]) + self.data.phi1(t[k]))
/ (
                                    (2.0 * self.gamma * self.a /
self.h / (2.0 * self.a + self.h * self.b)) + (
                                    self.gamma * self.h / self.tau /
(2.0 * self.a + self.h * self.b)) - (
                                            self.gamma * self.h *
self.c / (
                                            2.0 * self.a + self.h *
self.b)) * self.c + self.delta))
            elif self.data.bound_type == 'a1p3':
                u[k][0] = self.data.phi0(t[k])
                u[k][-1] = (self.data.phi1(k * self.tau) + u[k][-2] /
self.h + 2 * self.tau * u[k - 1][-1] / self.h) / \
                           (1 / self.h + 2 * self.tau / self.h)
        return u

    def crank_nicolson_solver(self, N, K, T):
        theta = 0.5
        lst = get_zeros(N, K)
        a = lst[0]
        b = lst[1]
        c = lst[2]
        d = lst[3]
        u = lst[4]
        for i in range(1, N - 1):
            u[0][i] = self.data.psi(i * self.h)

        for k in range(1, K):
            self.calculate(a, b, c, d, u, k, N, T, K)

            tmp_imp = tma(a, b, c, d)
```

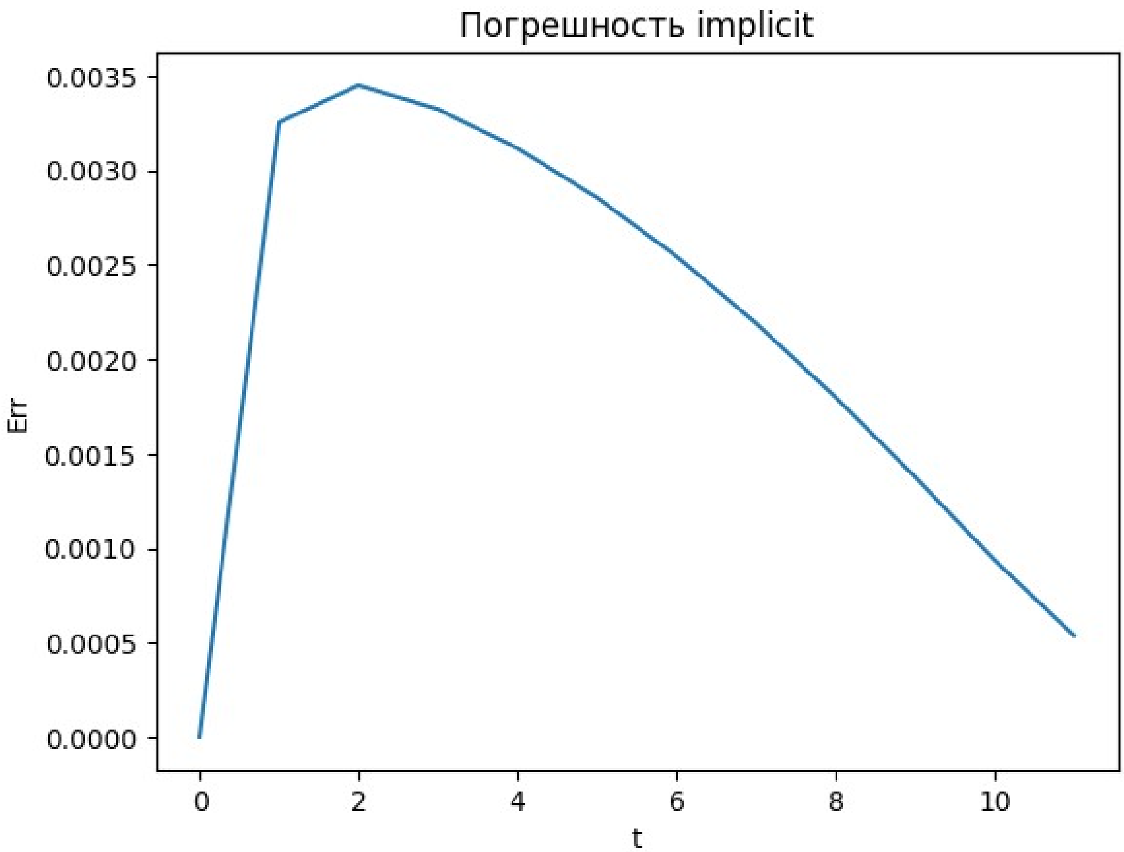
```
            tmp_exp = np.zeros(N)
            tmp_exp[0] = self.data.phi0(self.tau)
            for j in range(1, N - 1):
                tmp_exp[j] = self.sigma * u[k - 1][j + 1] + (1 - 2 *
self.sigma) * u[k - 1][j] + \
                             self.sigma * u[k - 1][j - 1] + self.tau *
self.data.f(j * self.h, k * self.tau)
            tmp_exp[-1] = self.data.phi1(self.tau)

            for j in range(N):
                u[k][j] = theta * tmp_imp[j] + (1 - theta) *
tmp_exp[j]

        return u


def compare_error(dict_):
    error = [[abs(i - j) for i, j in zip(x, y)] for x, y in
zip(dict_['numerical'], dict_['analytic'])]
    return error


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_['crank_nicolson'][time], color='y',
label='crank_nicolson')
    plt.plot(dict_['analytic'][time], color='g', label='analytic')
    plt.legend(loc='best')
    plt.ylabel('U')
    plt.xlabel('number')
    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()

    plt.title('Погрешность crank_nicolson')
    plt.plot(abs(dict_['crank_nicolson'][time] - dict_['analytic']
```

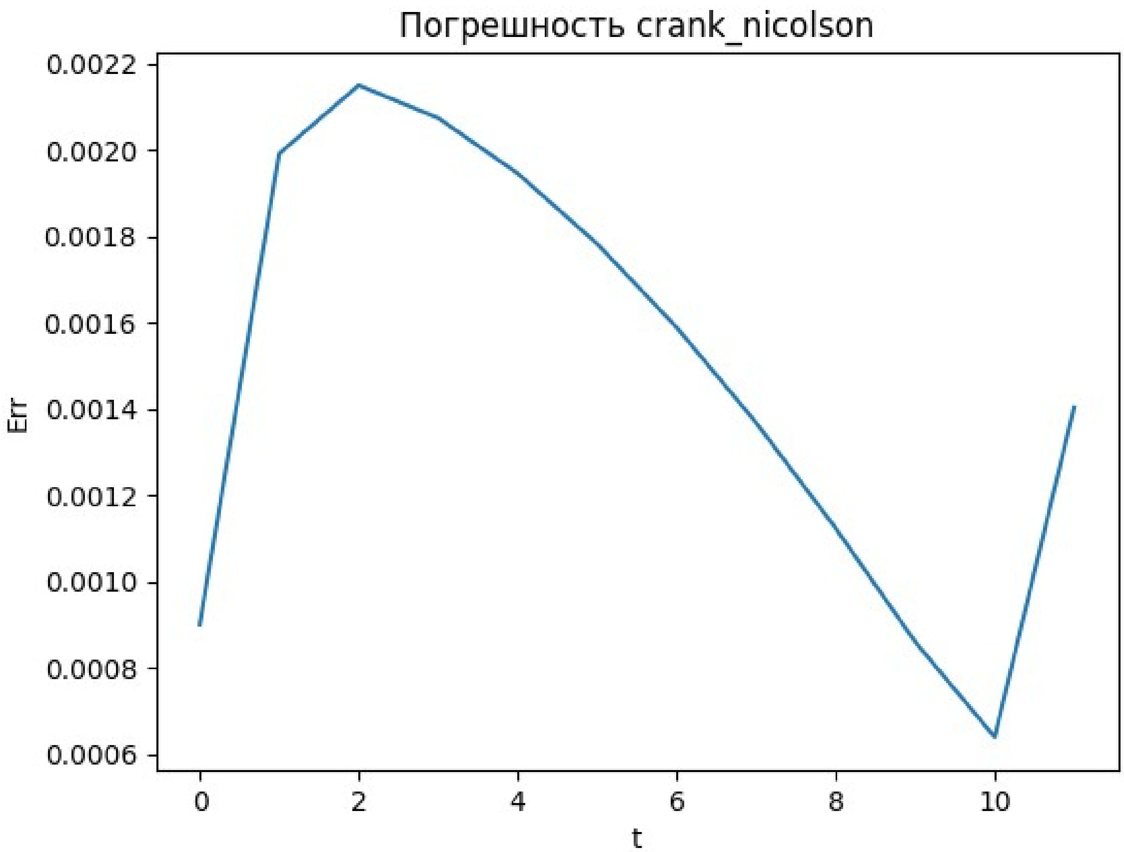
```
[time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()


N, K, T = 12, 10000, 18
args = {
    'l': np.pi / 2,
    'psi': lambda x: 0,
    'f': lambda x, t: np.cos(x) * (np.cos(t) + np.sin(t)),
    'phi0': lambda t: np.sin(t),
    'phi1': lambda t: -np.sin(t),
    'solution': lambda x, t: np.sin(t) * np.cos(x),
    'bound_type': 'a1p1'
}
solver = ParabolicSolver(args, N, K, T)
dict_ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'crank_nicolson': solver.crank_nicolson_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}
presontation(dict_ans, 2)
```



Линии уровня
implicit
explicit
crank_nicolson
analytic
0.003
0.002
0.001
0.000
−0.001
U
0
2
4
6
8
10
number

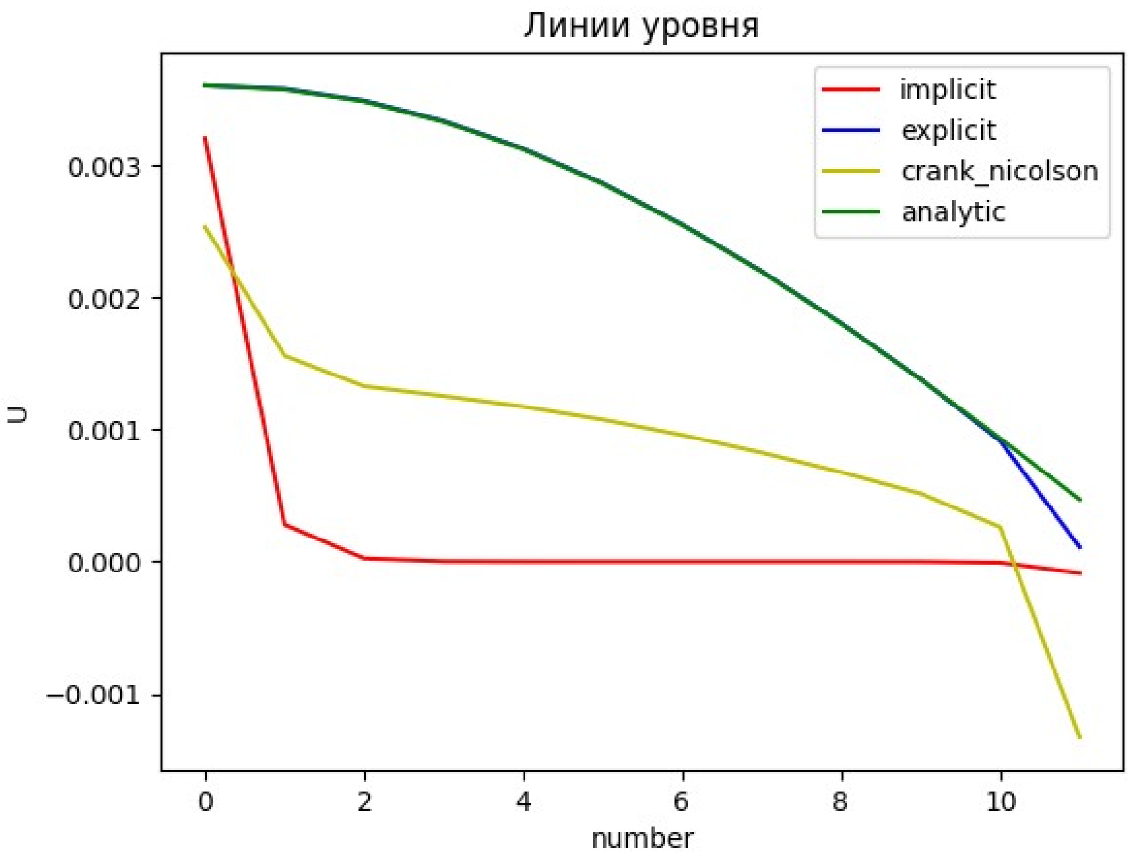

Погрешность explicit
1e−6
Err
t
0
1
2
3
4
5
6
7
8
0
2
4
6
8
10

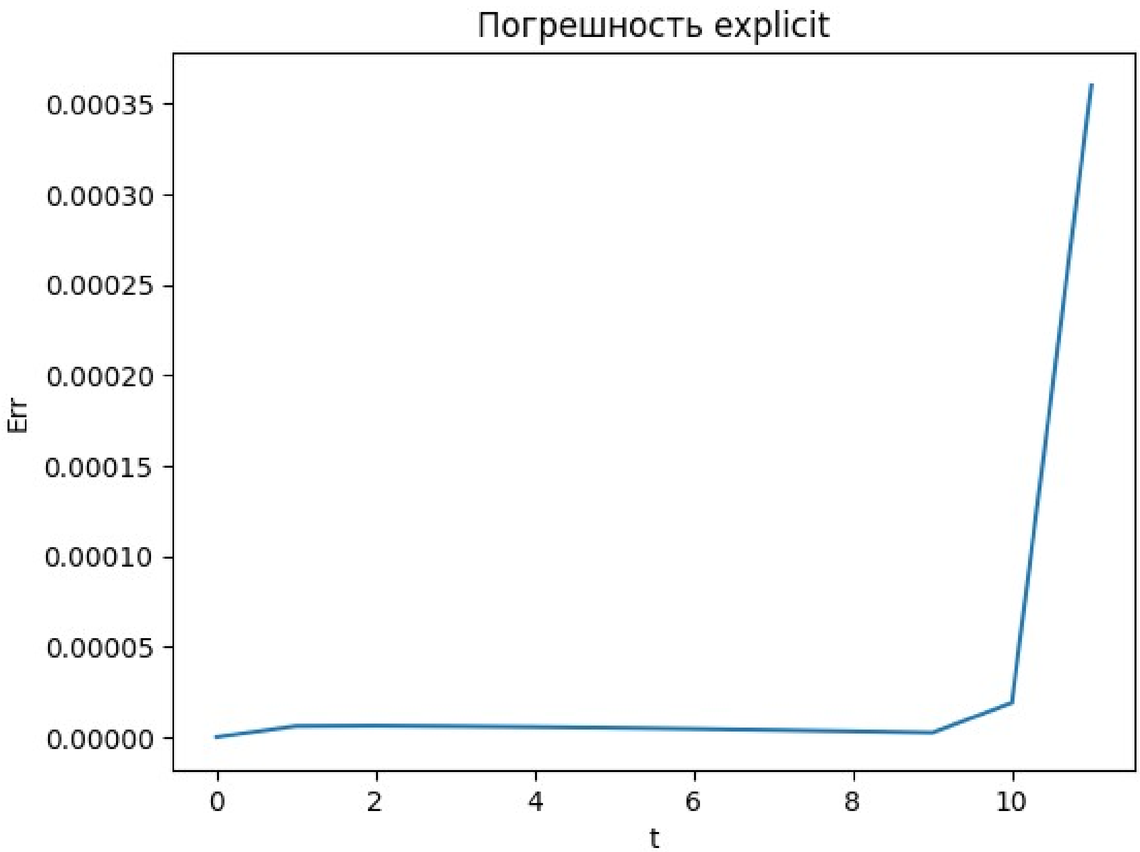

Погрешность implicit
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
Err
0
2
4
6
8
10
t

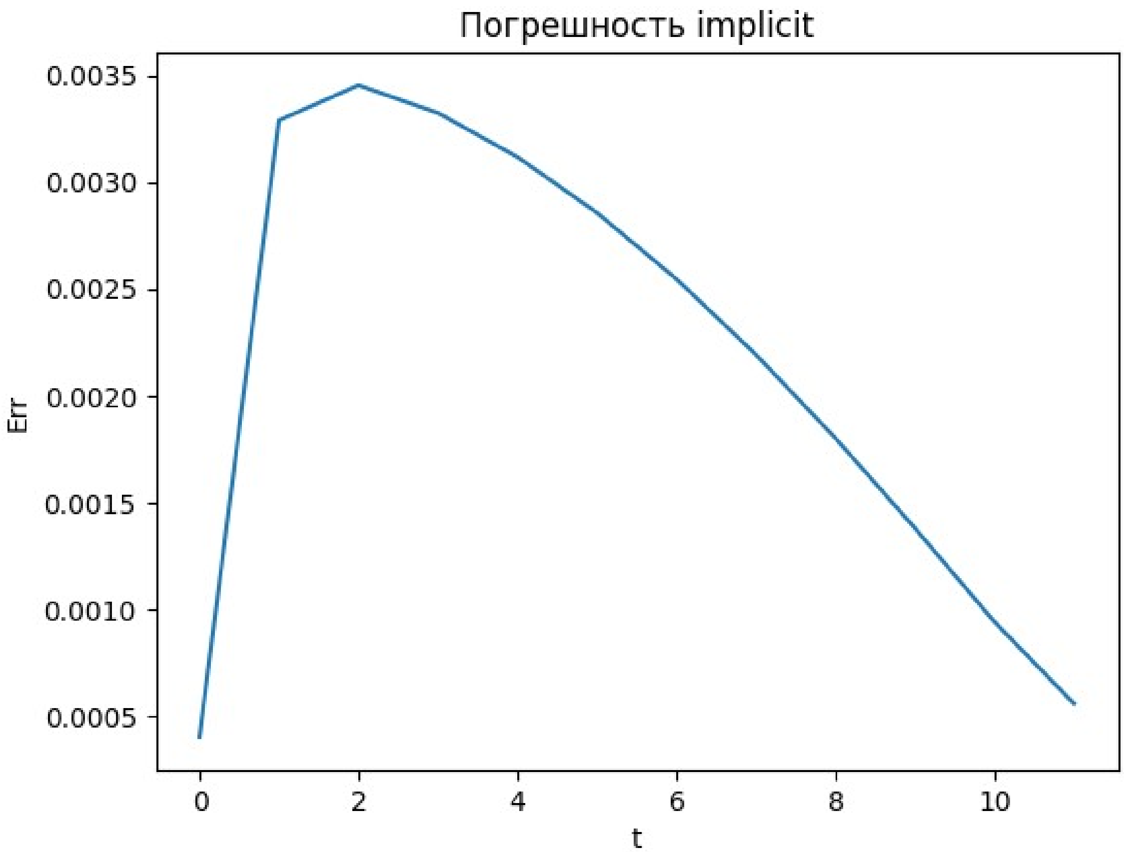


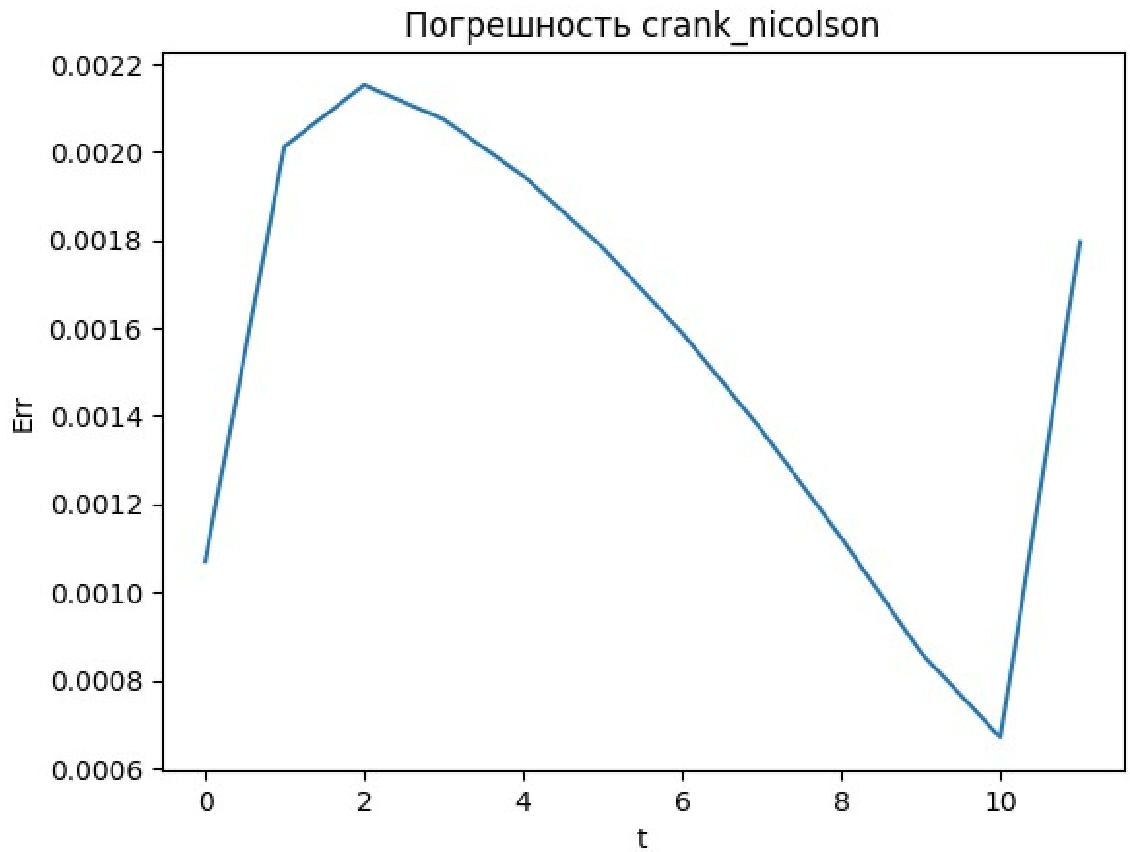
Апроксимация 3-х точечная второго порядка

```
N, K, T = 12, 10000, 18
args = {
    'l': np.pi / 2,
    'psi': lambda x: 0,
    'f': lambda x, t: np.cos(x) * (np.cos(t) + np.sin(t)),
    'phi0': lambda t: np.sin(t),
    'phi1': lambda t: -np.sin(t),
    'solution': lambda x, t: np.sin(t) * np.cos(x),
    'bound_type': 'a1p2'
}
solver = ParabolicSolver(args, N, K, T)
dict_ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'crank_nicolson': solver.crank_nicolson_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}
presontation(dict_ans, 2)
```


Линии уровня
implicit
explicit
crank_nicolson
analytic
U
0.003
0.002
0.001
0.000
−0.001
0
2
4
6
8
10
number


Погрешность explicit
Err
t
0.00035
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
2
4
6
8
10


Погрешность implicit
Err
t
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0
2
4
6
8
10



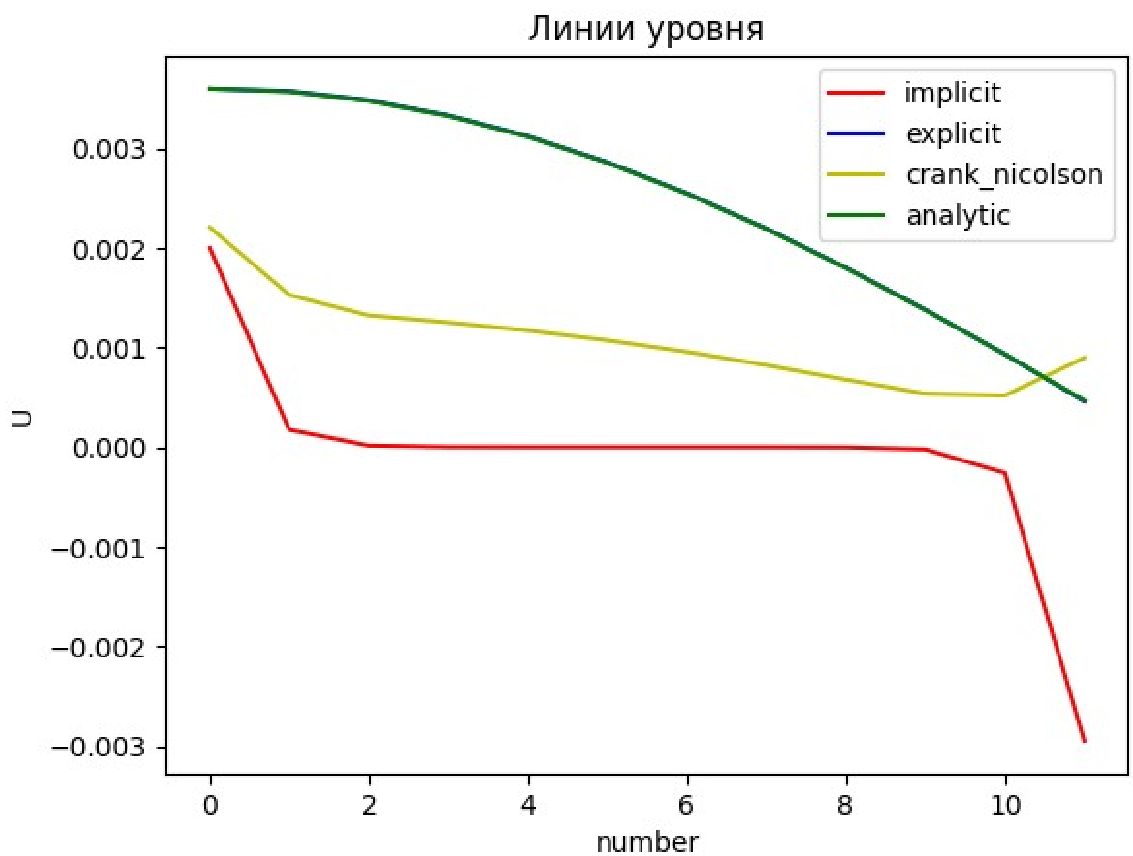
В качестве результата я получаю графики линий уровня U. Они наиболее наглядно показывают точность методов, и в каких промежутках какой метод будет эффективен, а какой нет. Также я вывожу графики модуля ошибки каждого метода. Исследование зависимости погрешности от параметров находится в одном файле с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

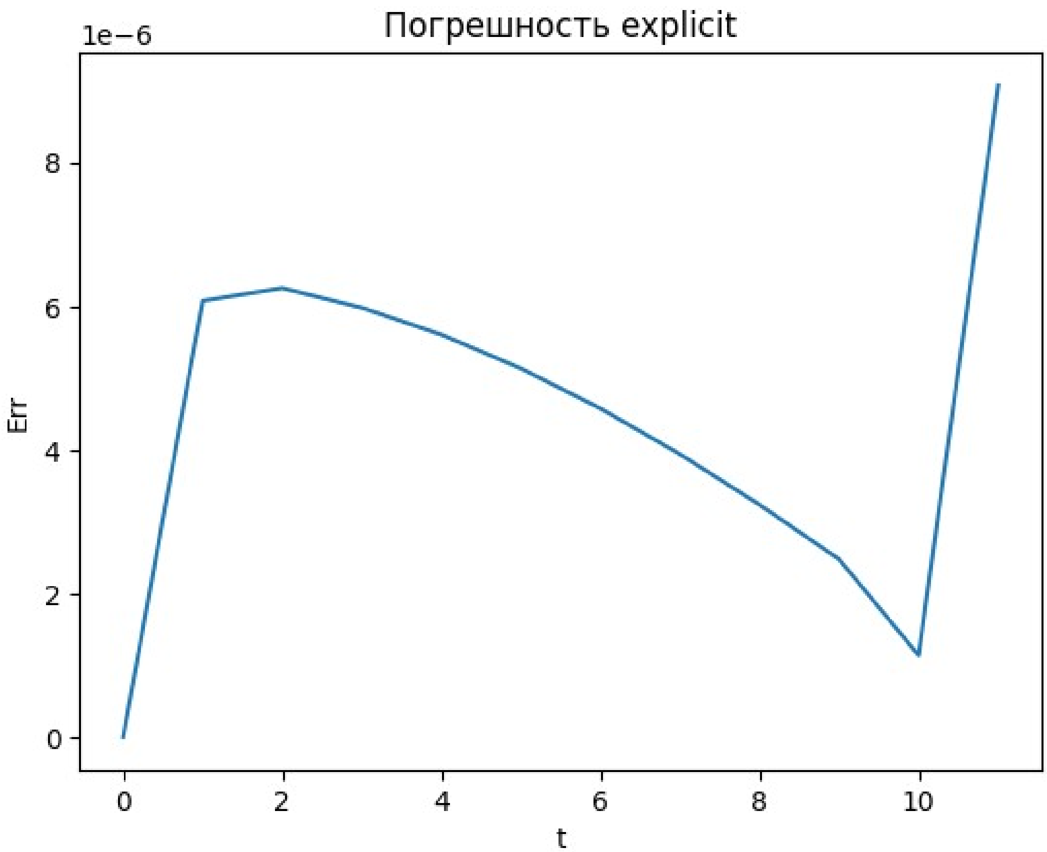
Апроксимация 2-х точечная 2 порядка

```
N, K, T = 12, 10000, 18
args = {
    'l': np.pi / 2,
    'psi': lambda x: 0,
    'f': lambda x, t: np.cos(x) * (np.cos(t) + np.sin(t)),
    'phi0': lambda t: np.sin(t),
    'phi1': lambda t: -np.sin(t),
    'solution': lambda x, t: np.sin(t) * np.cos(x),
    'bound_type': 'a1p3'
}
solver = ParabolicSolver(args, N, K, T)
dict_ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'crank_nicolson': solver.crank_nicolson_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
```

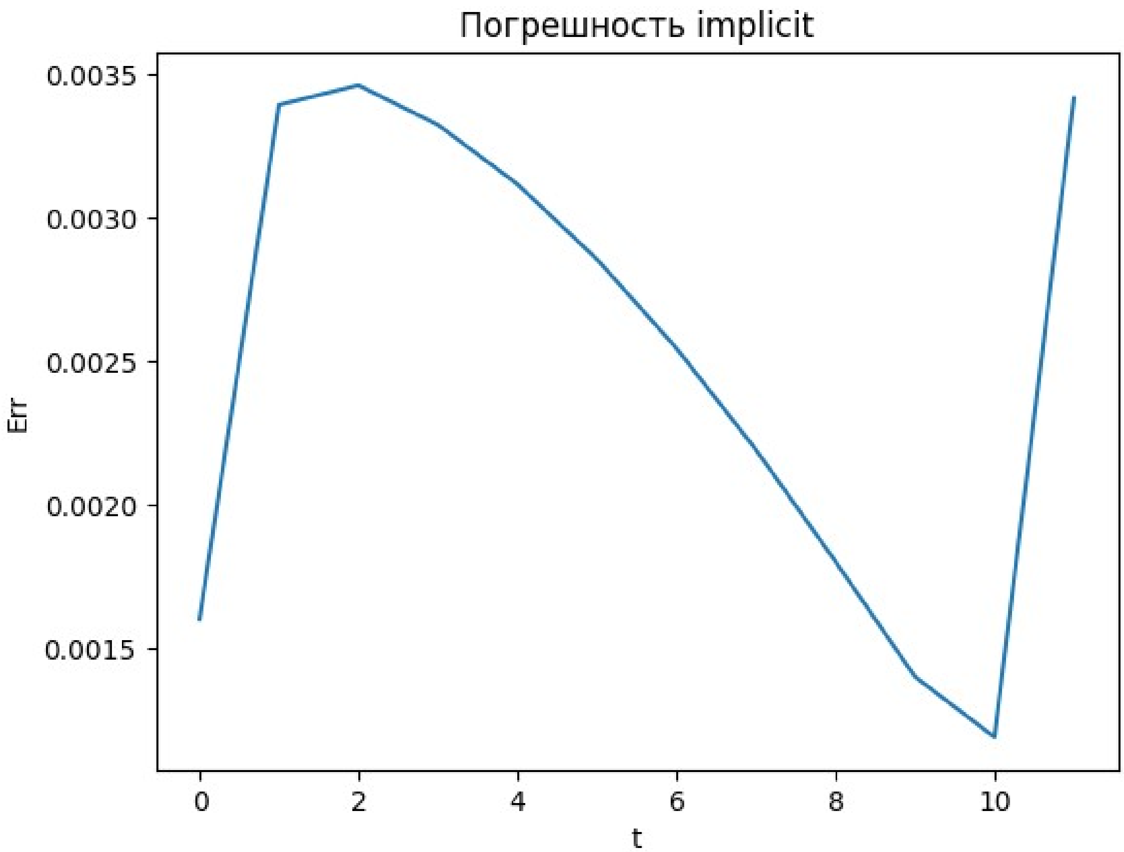
```
}
presontation(dict_ans, 2)
```

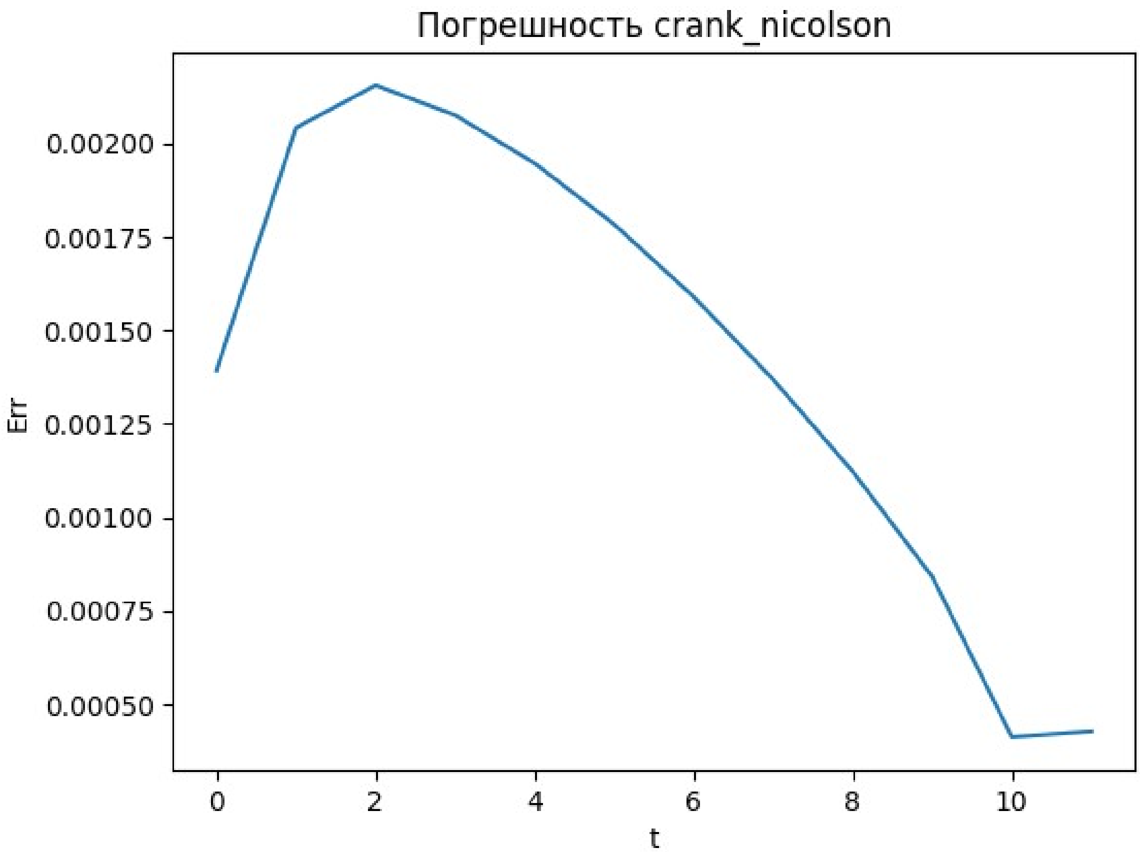
```
<ipython-input-1-d0f069d996ce>:99: RuntimeWarning: overflow
encountered in double_scalars
  d[-1] = self.data.phi1(k * self.tau) + self.data.f((N - 1) * self.h,
k * self.tau) \
<ipython-input-1-d0f069d996ce>:11: RuntimeWarning: invalid value
encountered in double_scalars
  q_tmp = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
```





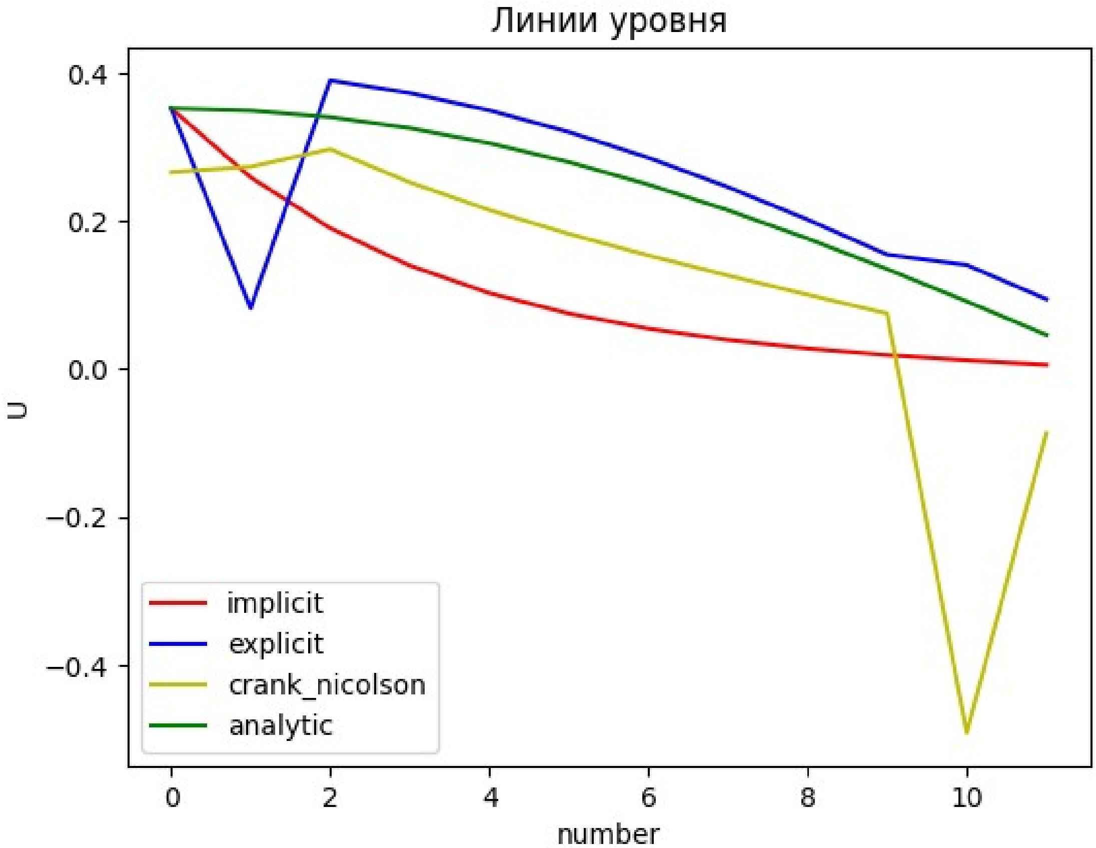
Погрешность explicit
1e−6
Err
t
0
2
4
6
8
10


Погрешность implicit
0.0035
0.0030
0.0025
0.0020
0.0015
Err
0
2
4
6
8
10
t

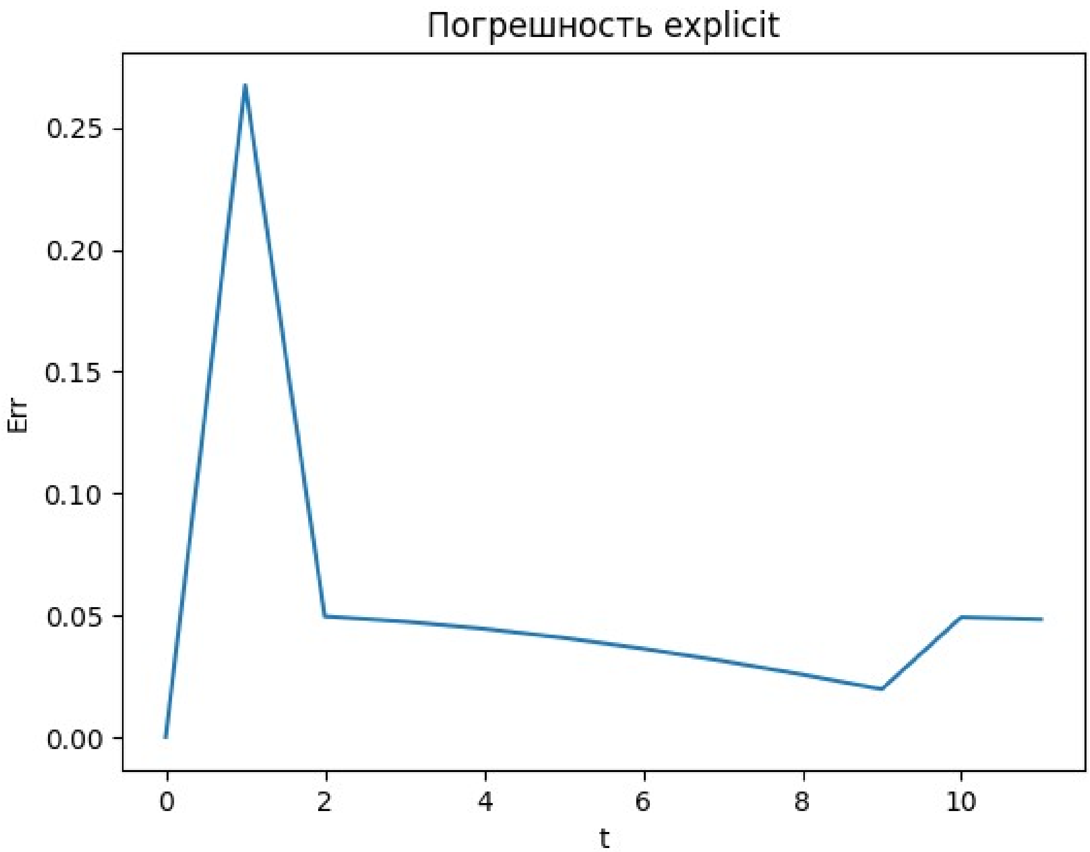


Исследование зависимости погрешности от величина tau и h

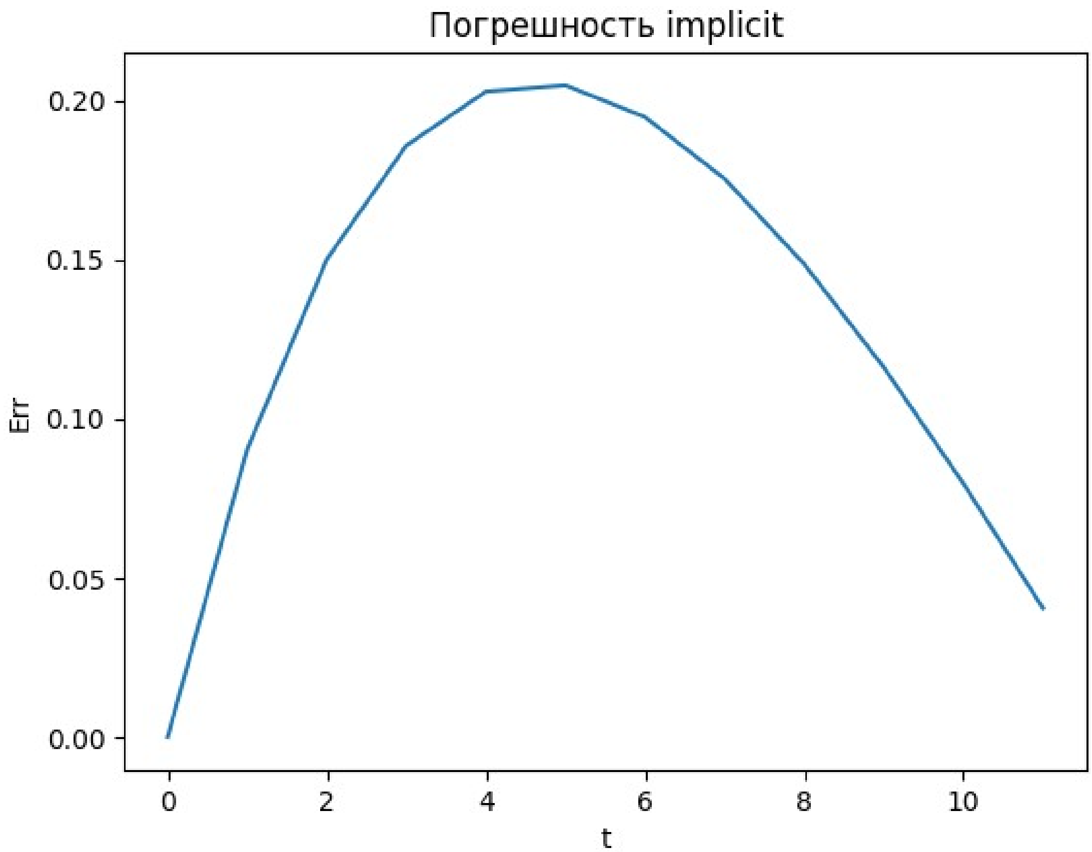
```
N, K, T = 12, 100, 18
args = {
    'l': np.pi / 2,
    'psi': lambda x: 0,
    'f': lambda x, t: np.cos(x) * (np.cos(t) + np.sin(t)),
    'phi0': lambda t: np.sin(t),
    'phi1': lambda t: -np.sin(t),
    'solution': lambda x, t: np.sin(t) * np.cos(x),
    'bound_type': 'a1p1'
}
solver = ParabolicSolver(args, N, K, T)
dict_ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'crank_nicolson': solver.crank_nicolson_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}
presontation(dict_ans, 2)
```

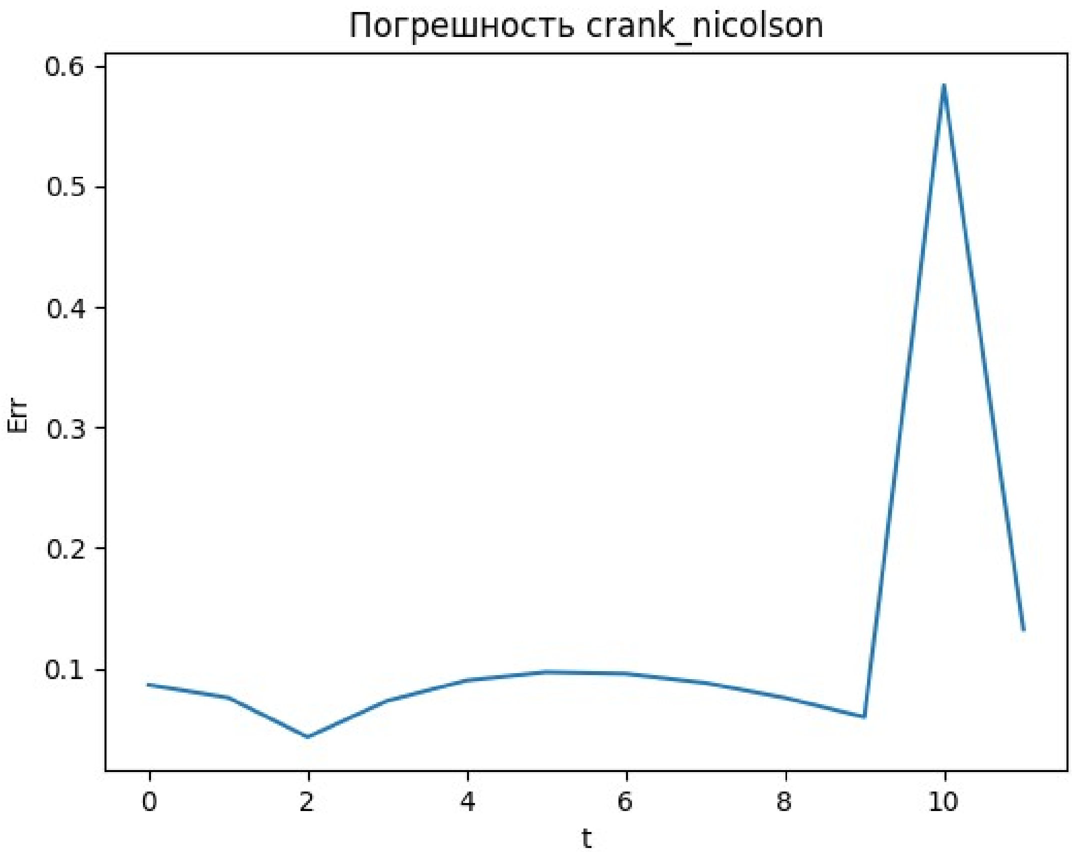


Линии уровня
0.4
0.2
0.0
−0.2
−0.4
U
0
2
4
6
8
10
number
implicit
explicit
crank_nicolson
analytic



Погрешность explicit
0.25
0.20
0.15
0.10
0.05
0.00
Err
0
2
4
6
8
10
t

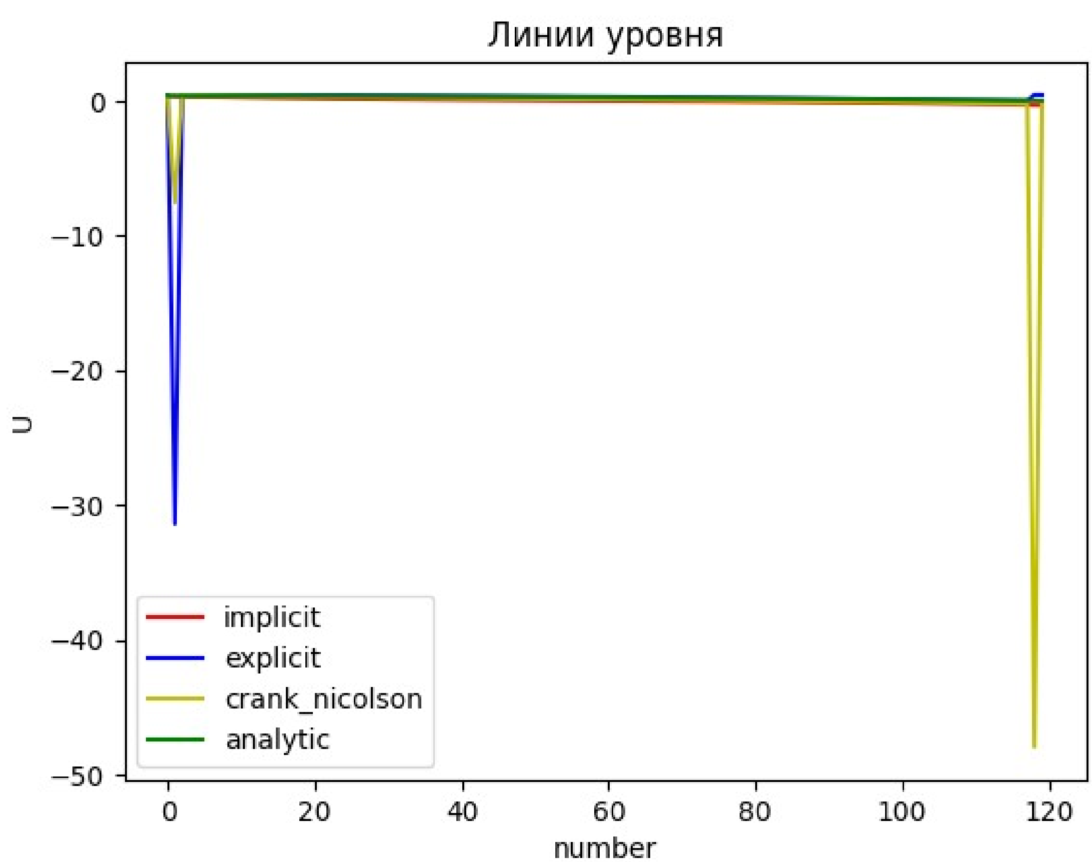

Погрешность implicit
Err
t
0.00
0.05
0.10
0.15
0.20
0
2
4
6
8
10

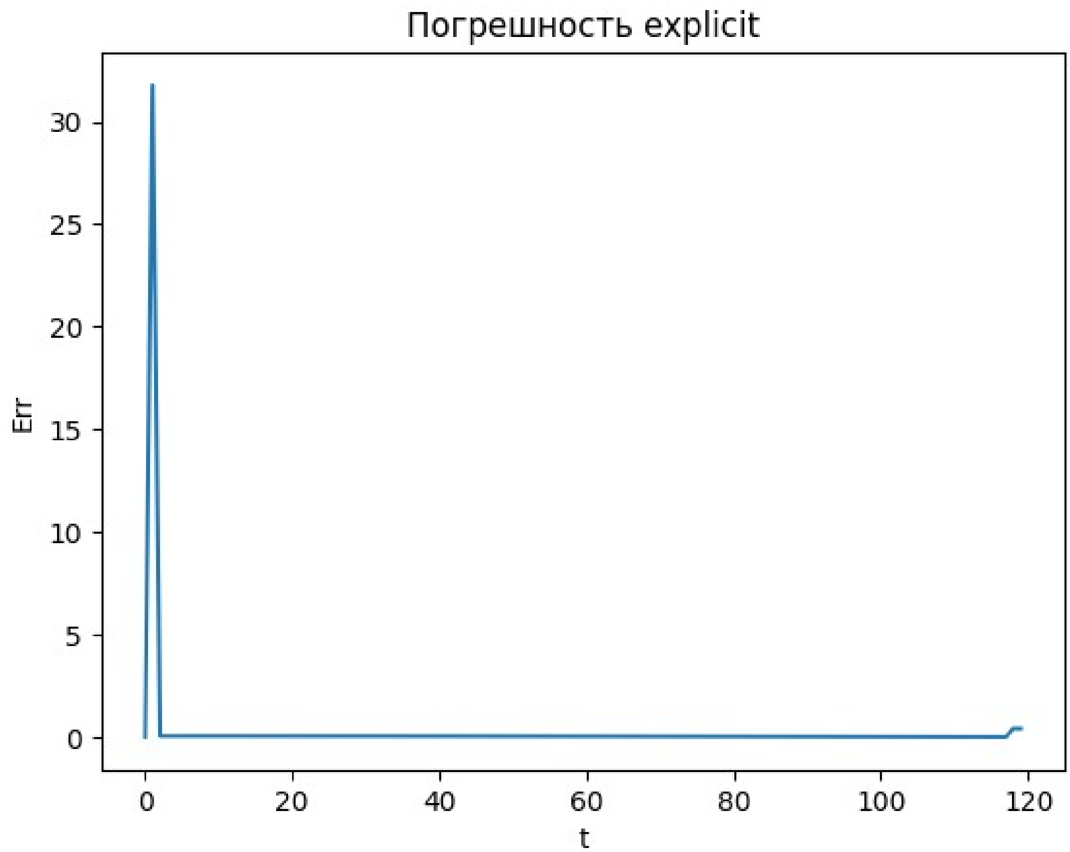


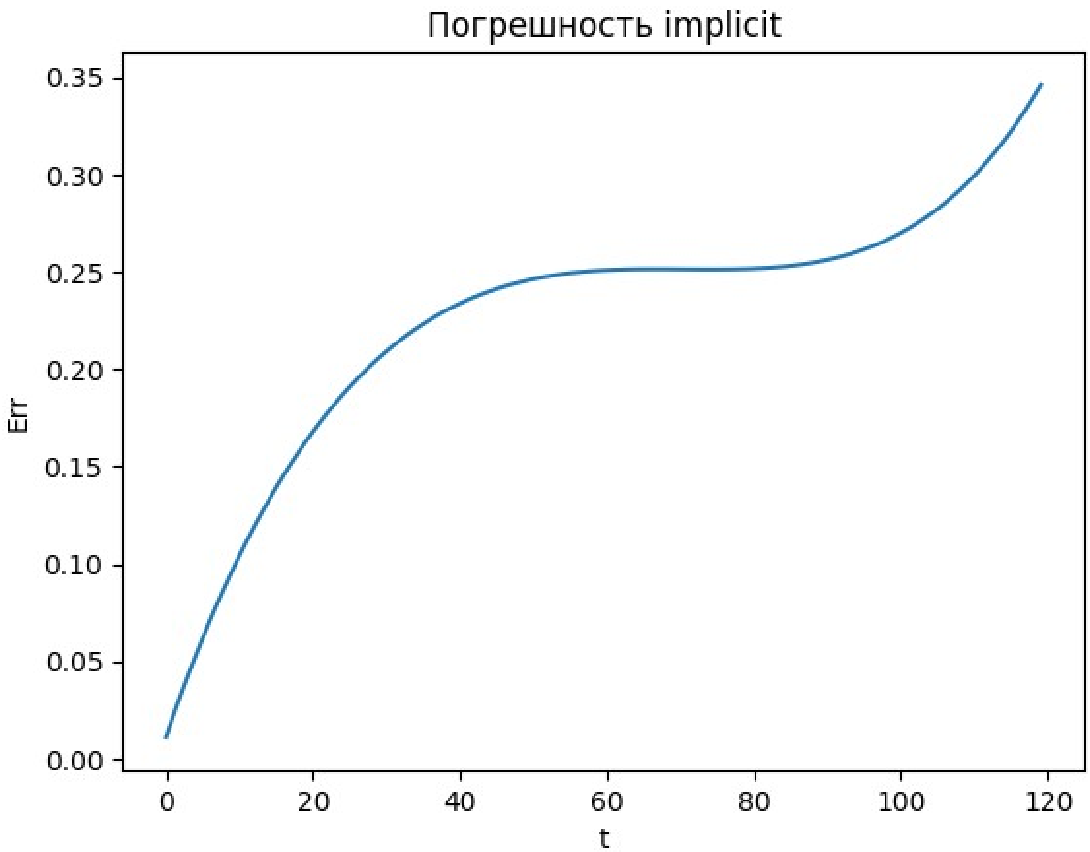

```python
N, K, T = 120, 100, 18
args = {
    'l': np.pi / 2,
    'psi': lambda x: 0,
    'f': lambda x, t: np.cos(x) * (np.cos(t) + np.sin(t)),
    'phi0': lambda t: np.sin(t),
    'phi1': lambda t: -np.sin(t),
    'solution': lambda x, t: np.sin(t) * np.cos(x),
    'bound_type': 'a1p2'
}
solver = ParabolicSolver(args, N, K, T)
dict_ans = {
    'implicit': solver.implicit_solver(N, K, T),
    'explicit': solver.explicit_solver(N, K, T),
    'crank_nicolson': solver.crank_nicolson_solver(N, K, T),
    'analytic': solver.analyticSolve(N, K, T)
}
presontation(dict_ans, 2)
```

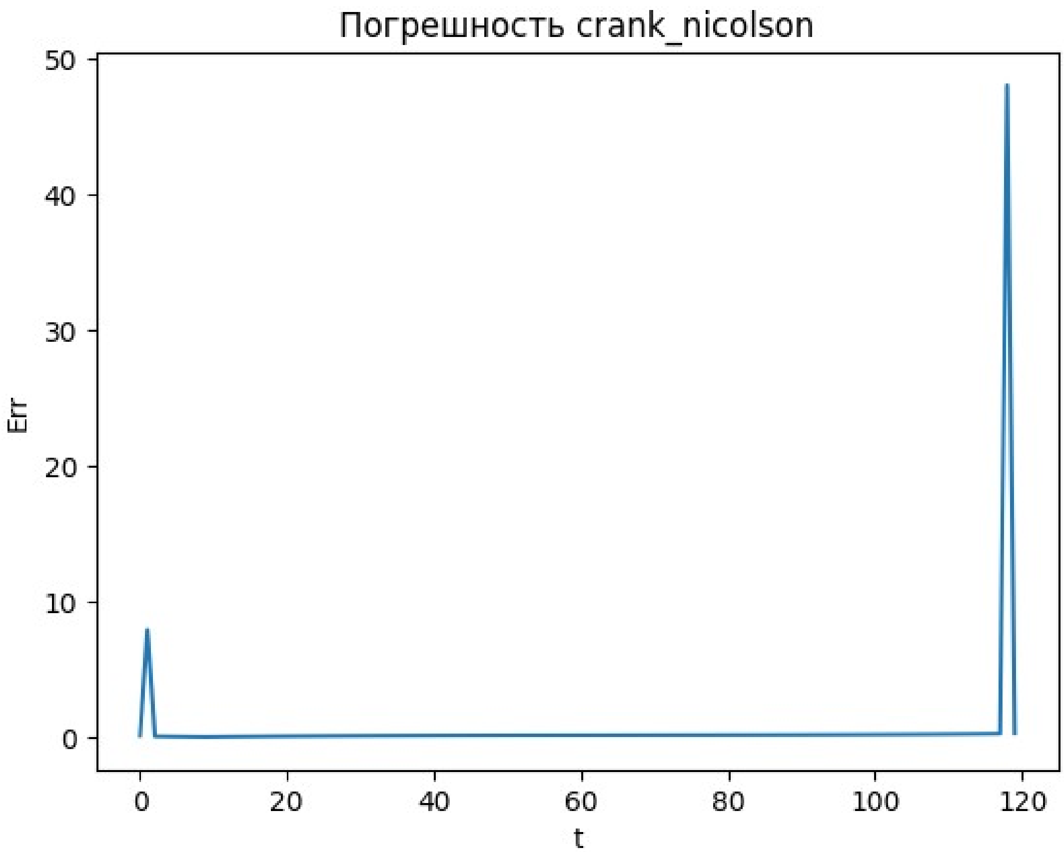
```
<ipython-input-1-d0f069d996ce>:129: RuntimeWarning: overflow
encountered in double_scalars
  u[k][j] = (u[k - 1][j + 1] * (self.a ** 2.0 * self.tau / self.h **
2.0)
```

```
<ipython-input-1-d0f069d996ce>:130: RuntimeWarning: overflow
encountered in double_scalars
  - 2 * u[k - 1][j] * (self.a ** 2.0 * self.tau / self.h ** 2.0)
<ipython-input-1-d0f069d996ce>:131: RuntimeWarning: overflow
encountered in double_scalars
  + u[k - 1][j - 1] * (self.a ** 2.0 * self.tau / self.h ** 2.0)
<ipython-input-1-d0f069d996ce>:140: RuntimeWarning: overflow
encountered in double_scalars
  u[k][-1] = (((2.0 * self.gamma * self.a / self.h / (2.0 * self.a +
self.h * self.b)) * u[k][-2] +
<ipython-input-1-d0f069d996ce>:129: RuntimeWarning: invalid value
encountered in double_scalars
  u[k][j] = (u[k - 1][j + 1] * (self.a ** 2.0 * self.tau / self.h **
2.0)
<ipython-input-1-d0f069d996ce>:173: RuntimeWarning: overflow
encountered in double_scalars
  tmp_exp[j] = self.sigma * u[k - 1][j + 1] + (1 - 2 * self.sigma) *
u[k - 1][j] + \
<ipython-input-1-d0f069d996ce>:174: RuntimeWarning: overflow
encountered in double_scalars
  self.sigma * u[k - 1][j - 1] + self.tau * self.data.f(j * self.h, k
* self.tau)
```




Погрешность explicit
Err
t
0
5
10
15
20
25
30
0
20
40
60
80
100
120


Погрешность implicit
Err
t
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
20
40
60
80
100
120



Вывод:

Из графиков погрешности можно сделать вывод, что параметр tau оказывает наибольшее влияние на погрешность (чем он меньше, тем погрешность ниже), а количество шагов h увеличивает погрешность многократно. Важно отметить, что конечно-разностные схемы для решения уравнений параболического типа обладают высокой точностью и при достаточно мелком значении tau могут достичь настолько небольшой погрешности, что ее можно пренебречь при решении реальных задач математической физики.