Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №5

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ»

Вариант 3

Выполнил: Дондоков В.И.

Группа: М8О-409Б-20

Проверил: Пивоваров Д.Е.

Дата:

Оценка:

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.

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2}, \quad a > 0,$$
$$u_x(0,t) = \exp(-at),$$
$$u_x(\pi,t) = -\exp(-at),$$
$$u(x,0) = \sin x.$$

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

Теоретическая часть:

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.

Код програмы:

```python
def phi_0(t, a = 1.0):
    return np.exp(-a*t)

def phi_l(t, a = 1.0):
    return -np.exp(-a*t)

def u_0(x):
    return np.sin(x)

def u(x, t, a = 1.0):
    return np.exp(-a*t)*np.sin(x)


class Schema:
    def __init__(self, a=1, f0=phi_0, fl=phi_l, u0=u_0,
                 O=0.5, l0=0, l1=math.pi, T=5, aprx_cls=None):
        self.fl = lambda t: fl(t, a)
        self.f0 = lambda t: f0(t, a)
        self.u0 = u0
        self.T = T
        self.l0 = l0
        self.l1 = l1
        self.tau = None
        self.h = None
        self.a = a
        self.O = O
        self.approx = None
        if aprx_cls is not None:
            self._init_approx(aprx_cls)
        self.sigma = None

    def _init_approx(self, a_cls):
        self.approx = a_cls(self.f0, self.fl)

    def set_approx(self, aprx_cls):
        self._init_approx(self, aprx_cls)

    def set_l0_l1(self, l0, l1):
        self.l0 = l0
        self.l1 = l1

    def set_T(self, T):
        self.T = T

    def _compute_h(self, N):
        self.h = (self.l1 - self.l0) / N

    def _compute_tau(self, K):
        self.tau = self.T / K

    def _compute_sigma(self):
        self.sigma = self.a * self.tau / (self.h * self.h)

    @staticmethod
    def nparange(start, end, step=1):
        now = start
        e = 0.00000000001
        while now - e <= end:
            yield now
            now += step

    def _compute_line(self, t, x, last_line):
        pass
```

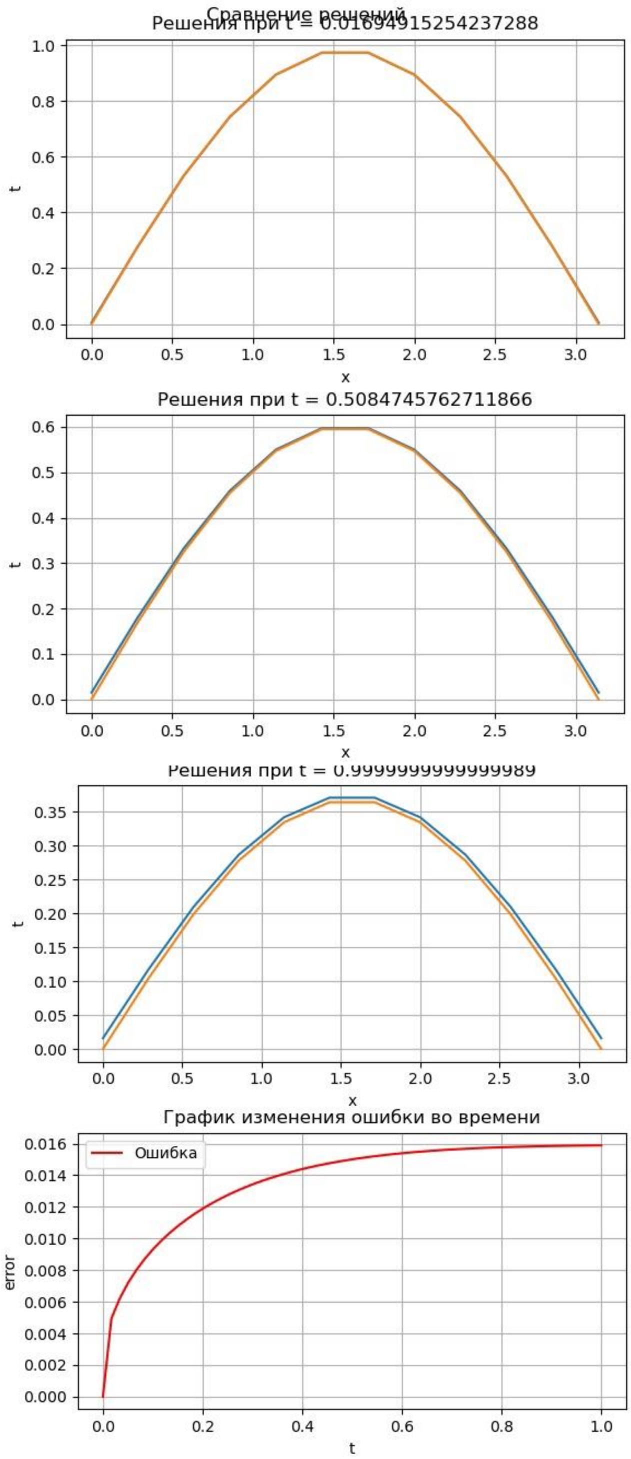
```
    def __call__(self, N=30, K=110):
        N, K = N - 1, K - 1
        self._compute_tau(K)
        self._compute_h(N)
        self._compute_sigma()
        ans = []
        x = list(self.nparange(self.l0, self.l1, self.h))
        last_line = list(map(self.u0, x))
        ans.append(list(last_line))
        X = []
        Y = []
        X.append(x)
        Y.append([0.0 for _ in x])
        for t in self.nparange(self.tau, self.T, self.tau):
            ans.append(self._compute_line(t, x, last_line))
            X.append(x)
            Y.append([t for _ in x])
            last_line = ans[-1]
        return X, Y, ans


# sigma < 0.5 - устойчивое решение

class Explict_Schema(Schema):
    def _compute_sigma(self):
        self.sigma = self.a * self.tau / (self.h * self.h)
        if self.sigma > 0.5:
            warnings.warn("Sigma > 0.5")

    def _compute_line(self, t, x, last_line):
        line = [None for _ in last_line]
        for i in range(1, len(x) - 1):
            line[i] = self.sigma * last_line[i - 1]
            line[i] += (1 - 2 * self.sigma) * last_line[i]
            line[i] += self.sigma * last_line[i + 1]
        line[0] = self.approx.explict_0(t, self.h, self.sigma,
                                        last_line, line, t - self.tau)
        line[-1] = self.approx.explict_l(t, self.h, self.sigma,
                                         last_line, line, t - self.tau)
        return line


class Explict_Implict(Schema):
    def set_O(self, O):
        self.O = O

    @staticmethod
    def race_method(A, b):
        P = [-item[2] for item in A]
        Q = [item for item in b]

        P[0] /= A[0][1]
        Q[0] /= A[0][1]

        for i in range(1, len(b)):
            z = (A[i][1] + A[i][0] * P[i - 1])
            P[i] /= z
            Q[i] -= A[i][0] * Q[i - 1]
            Q[i] /= z

        x = [item for item in Q]

        for i in range(len(x) - 2, -1, -1):
            x[i] += P[i] * x[i + 1]
```

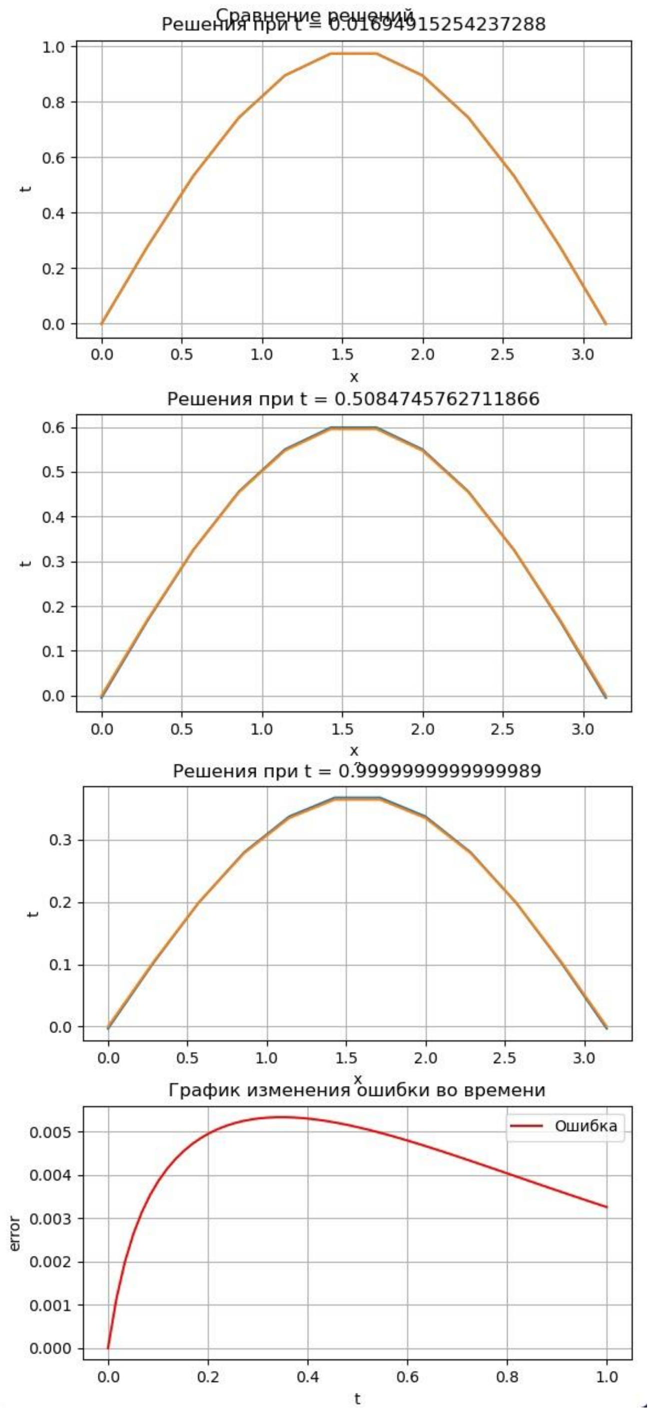
```python
        return x

    def _compute_line(self, t, x, last_line):
        a = self.sigma * self.O
        b = -1 - 2 * self.sigma * self.O

        A = [(a, b, a) for _ in range(1, len(x) - 1)]
        w = [
            -(last_line[i] +
              (1 - self.O) * self.sigma *
              (last_line[i - 1] - 2 * last_line[i] + last_line[i + 1]))
            for i in range(1, len(x) - 1)
        ]
        koeffs = self.approx.nikolson_0(t, self.h, self.sigma,
                                        last_line, self.O, t - self.tau)
        A.insert(0, koeffs[:-1])
        w.insert(0, koeffs[-1])
        koeffs = self.approx.nikolson_l(t, self.h, self.sigma,
                                        last_line, self.O, t - self.tau)
        A.append(koeffs[:-1])
        w.append(koeffs[-1])

        return self.race_method(A, w)


class Approx:
    def __init__(self, f0, fl):
        self.f0 = f0
        self.fl = fl

    def explict_0(self, t, h, sigma, l0, l1, t0):
        pass

    def explict_l(self, t, h, sigma, l0, l1, t0):
        pass

    def nikolson_0(self, t, h, sigma, l0, O, t0):
        pass

    def nikolson_l(self, t, h, sigma, l0, O, t0):
        pass


class approx_two_one(Approx):
    def explict_0(self, t, h, sigma, l0, l1, t0):
        return -h * self.f0(t) + l1[1]

    def explict_l(self, t, h, sigma, l0, l1, t0):
        return h * self.fl(t) + l1[-2]

    def nikolson_0(self, t, h, sigma, l0, O, t0):
        return 0, -1, 1, h * self.f0(t)

    def nikolson_l(self, t, h, sigma, l0, O, t0):
        return -1, 1, 0, h * self.fl(t)


class approx_three_two(Approx):
    def explict_0(self, t, h, sigma, l0, l1, t0):
        return (-2 * h * self.f0(t) + 4 * l1[1] - l1[2]) / 3

    def explict_l(self, t, h, sigma, l0, l1, t0):
        return (2 * h * self.fl(t) + 4 * l1[-2] - l1[-3]) / 3
```

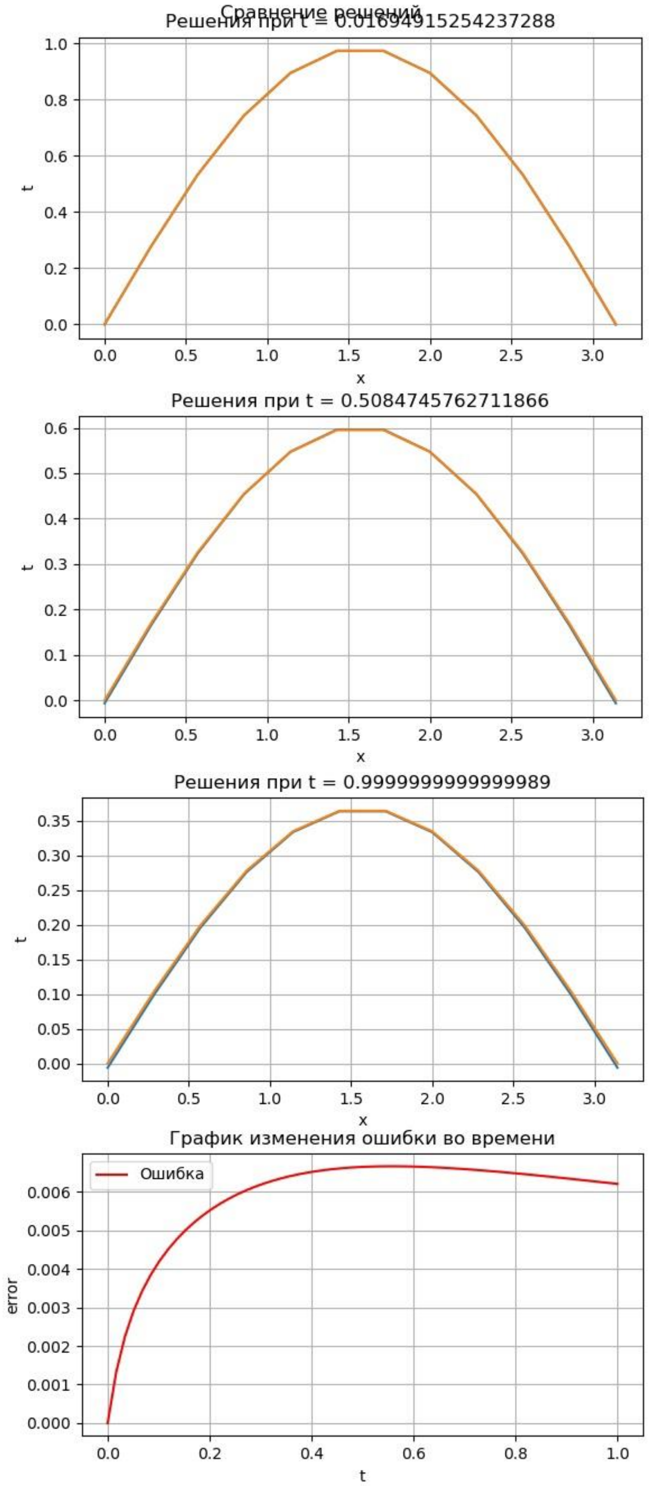
```
    def nikolson_0(self, t, h, sigma, l0, O, t0):
        d = 2 * sigma * O * h * self.f0(t)
        d -= l0[1] + (1 - O) * sigma * (l0[0] - 2 * l0[1] + l0[2])
        return 0, -2 * sigma * O, 2 * sigma * O - 1, d

    def nikolson_l(self, t, h, sigma, l0, O, t0):
        d = 2 * sigma * O * h * self.fl(t)
        d += l0[-2] + (1 - O) * sigma * (l0[-3] - 2 * l0[-2] + l0[-1])
        return 1 - 2 * sigma * O, 2 * sigma * O, 0, d


class approx_two_two(Approx):
    def explict_0(self, t, h, sigma, l0, l1, t0):
        return -2 * sigma * h * self.f0(t0) + \
               2 * sigma * l0[1] + (1 - 2 * sigma) * l0[0]

    def explict_l(self, t, h, sigma, l0, l1, t0):
        return 2 * sigma * h * self.fl(t0) + \
               2 * sigma * l0[-2] + (1 - 2 * sigma) * l0[-1]

    def nikolson_0(self, t, h, sigma, l0, O, t0):
        d = 2 * sigma * O * h * self.f0(t) - l0[0]
        d -= 2 * (1 - O) * sigma * (l0[1] - l0[0] - h * self.f0(t0))
        return 0, -(2 * sigma * O + 1), 2 * sigma * O, d

    def nikolson_l(self, t, h, sigma, l0, O, t0):
        d = -2 * sigma * O * h * self.fl(t) - l0[-1]
        d -= 2 * (1 - O) * sigma * (l0[-2] - l0[-1] + h * self.fl(t0))
        return 2 * sigma * O, -(2 * sigma * O + 1), 0, d


def plot_graphs(x, t, sol, a=1):
    fig, ax = plt.subplots(4, 1)
    fig.suptitle('Сравнение решений')
    fig.set_figheight(16)
    fig.set_figwidth(6)

    times = [t[1][0], t[len(t) // 2][0], t[len(t) - 1][0]]
    solutions = [sol[1], sol[len(t) // 2], sol[len(t) - 1]]

    for i in range(3):
        time = times[i]
        ax[i].plot(x[0], solutions[i], label='Численный метод')
        ax[i].plot(x[0], [u(xi, times[i], a) for xi in x[0]],
label='Аналитическое решение')
        ax[i].grid(True)
        ax[i].set_xlabel('x')
        ax[i].set_ylabel('t')
        ax[i].set_title(f'Решения при t = {times[i]}')

    error = np.zeros(len(t))
    for i in range(len(t)):
        error[i] = np.max(np.abs(sol[i] - np.array([u(xi, t[i][0], a) for xi
in x[0]])))
    ax[3].plot([i[0] for i in t], error, 'red', label='Ошибка')
    ax[3].set_title('График изменения ошибки во времени')
    ax[3].set_xlabel('t')
    ax[3].set_ylabel('error')

    fig.tight_layout()
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
a = 1
schema = Explict_Schema(T = 1, aprx_cls=approx_three_two, a=a)
x, t, sol = schema(N = 12, K = 60)
plot_graphs(x, t, sol, a)

implict = Explict_Implict(T = 1, aprx_cls=approx_two_two, O=1)
x, t, sol = implict(N = 12, K = 60)
plot_graphs(x, t, sol)

krank = Explict_Implict(T = 1, aprx_cls=approx_two_two)
x, t, sol = krank(N = 12, K = 60)
plot_graphs(x, t, sol)
```

Результат:


Сравнение решений
Решения при t = 0.01694915254237288
t
x
Решения при t = 0.5084745762711866
t
x
Решения при t = 0.9999999999999989
t
x
График изменения ошибки во времени
Ошибка
error
t


Сравнение решений
Решения при t = 0.01694915254237288
Решения при t = 0.5084745762711866
Решения при t = 0.9999999999999989
График изменения ошибки во времени
Ошибка
x
t
error



Вывод:

В ходе выполнения лабораторной работы были изучены явная, неявная и гибридная

схемы решений начально-краевой задачи для дифференциального уравнения

параболического типа. Также были изучены три варианта аппроксимации граничных условий. Были получены результаты в графическом представлении и подсчитаны

погрешности для каждого варианта решения.