Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №8

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ»

Вариант 3

Выполнил: Дондоков В.И.

Группа: М8О-409Б-20

Проверил: Пивоваров Д.Е.

Дата:

Оценка:

Задание: 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 *u (x, t)*. Исследовать зависимость погрешности от сеточных параметров τ, h_x, h_y.

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + a\frac{\partial^2 u}{\partial y^2},\ a > 0,$$

$$u(0, y, t) = \cosh(y)\exp(-3at),$$
$$u(\frac{\pi}{4}, y, t) = 0,$$
$$u(x,0,t) = \cos(2x)\exp(-3at),$$
$$u(x, \ln 2, t) = \frac{5}{4}\cos(2x)\exp(-3at),$$
$$u(x, y,0) = \cos(2x)\cosh(y).$$

Аналитическое решение: $U(x, y, t) = \cos(2x)\cosh(y)\exp(-3at)$.

Теоретическая часть:

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:

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

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.

Код программы:

```
ap = 8
X_MAX = np.pi / 4
Y_MAX = np.log(2)
T_MAX = 10


def ux0(y, t):
    return np.cosh(y) * np.exp(-3*ap*t)


def uxl(y, t):
    return 0


def uy0(x, t):
    return np.cos(2*x) * np.exp(-3*ap*t)


def uyl(x, t):
    return 1.25 * np.cos(2*x) * np.exp(-3*ap*t)


def psi(x, y):
    return np.cos(2*x) * np.cosh(y)


def U(x, y, t):
    return np.cos(2*x) * np.cosh(y) * np.exp(-3*ap*t)

# Метод прогонки
def equation_solve(a, b, c, d):
    size = len(a)
    p = np.zeros(size)
    q = np.zeros(size)
    p[0] = -c[0] / b[0]
    q[0] = d[0] / b[0]
    for i in range(1, size):
        p[i] = -c[i] / (b[i] + a[i] * p[i - 1])
        q[i] = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
    x = np.zeros(size)
    x[-1] = q[-1]
    for i in range(size - 2, -1, -1):
        x[i] = p[i] * x[i + 1] + q[i]
    return x


def alternating_directions(hx, hy, tau):
    x = np.arange(0, X_MAX, hx)
    y = np.arange(0, Y_MAX, hy)
    t = np.arange(0, T_MAX, tau)
    u = np.zeros((t.size, x.size, y.size))
    u[0] = np.array([[psi(xi, yj) for yj in y] for xi in x])
    u[:, 0, :] = np.array([[ux0(yj, tk) for yj in y] for tk in t])
    u[:, -1, :] = np.array([[uxl(yj, tk) for yj in y] for tk in t])
    u[:, :, 0] = np.array([[uy0(xi, tk) for xi in x] for tk in t])
    u[:, :, -1] = np.array([[uyl(xi, tk) for xi in x] for tk in t])
    for k in range(1, t.size):
        k_half = np.zeros((x.size, y.size))
        for i in range(1, x.size - 1):
            a = np.zeros_like(y)
            b = np.zeros_like(y)
            c = np.zeros_like(y)
            d = np.zeros_like(y)
```

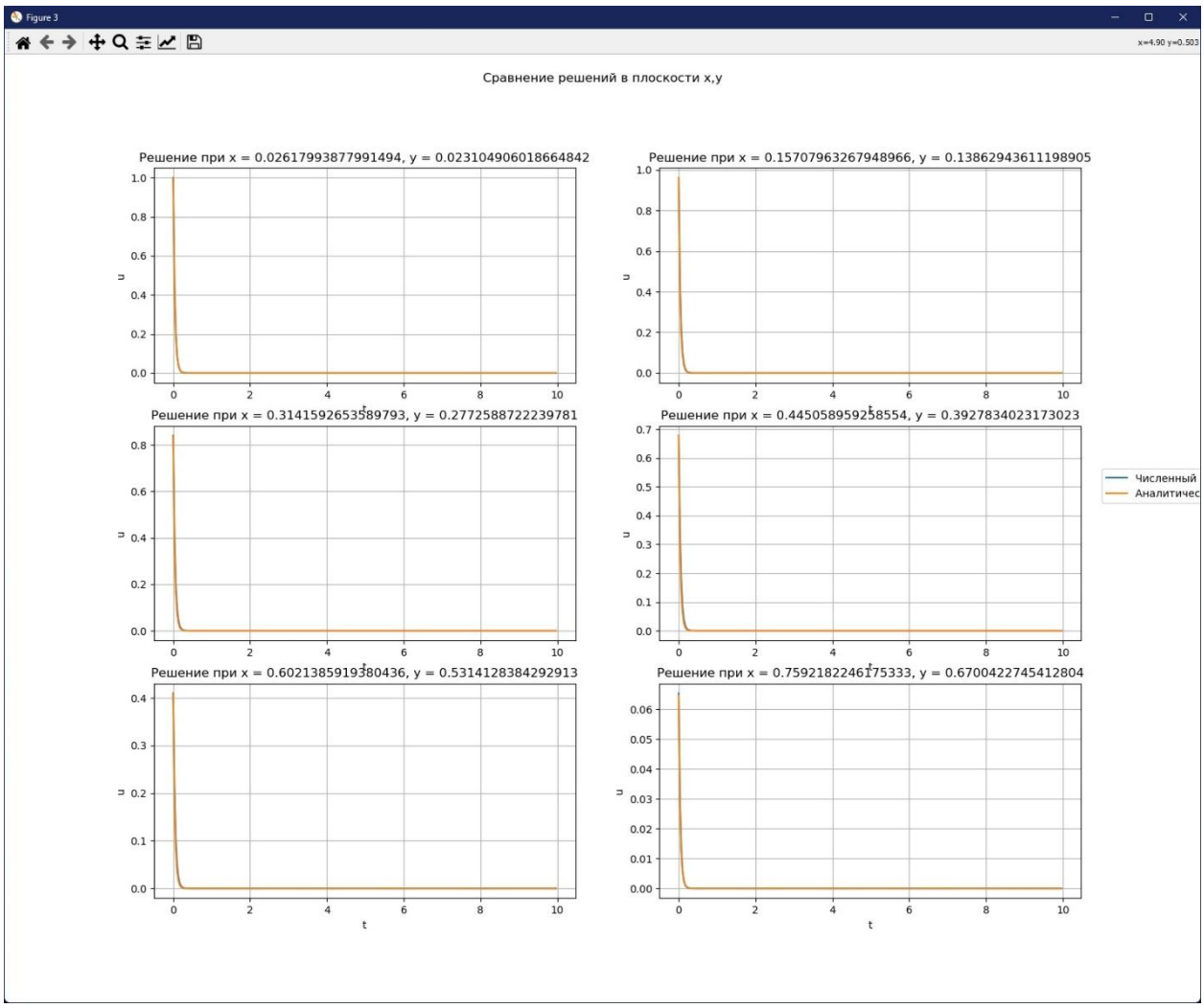
```
            s = (ap * tau) / (hx ** 2 ** 2 * 2)
            for j in range(1, y.size - 1):
                a[j] = s
                b[j] = -2 * s - 1
                c[j] = s
                d[j] = (-ap * tau / (hy ** 2 * 2)) * (u[k - 1][i][j + 1] - 2
* u[k - 1][i][j] + u[k - 1][i][j - 1]) - \
                       u[k - 1][i][j]

            alpha = 0
            betta = 1
            gamma = 1
            delta = 0
            b[0] = betta - alpha / hy
            c[0] = alpha / hy
            d[0] = uy0(x[i], t[k] - tau / 2)
            a[-1] = -gamma / hy
            b[-1] = delta + gamma / hy
            d[-1] = uyl(x[i], t[k] - tau / 2)
            k_half[i] = equation_solve(a, b, c, d)
            k_half[0] = ux0(y, t[k] - tau / 2)
            k_half[-1] = uxl(y, t[k] - tau / 2)

        for j in range(1, y.size - 1):
            a = np.zeros_like(x)
            b = np.zeros_like(x)
            c = np.zeros_like(x)
            d = np.zeros_like(x)
            s = (ap * tau) / (hx ** 2 * 2)
            for i in range(1, x.size - 1):
                a[i] = s
                b[i] = -2 * s - 1
                c[i] = s
                d[i] = (-ap * tau / (hy ** 2 * 2)) * (k_half[i][j + 1] - 2 *
k_half[i][j] + k_half[i][j - 1]) - \
                       k_half[i][j]
            alpha = 0
            betta = 1
            gamma = 0
            delta = 1
            b[0] = betta - alpha / hx
            c[0] = alpha / hx
            d[0] = ux0(y[j], t[k])
            a[-1] = -gamma / hx
            b[-1] = delta + gamma / hx
            d[-1] = uxl(y[j], t[k])

            ans = equation_solve(a, b, c, d)
            for i in range(ans.size):
                u[k][i][j] = ans[i]
            for j in range(y.size):
                u[k][0][j] = ux0(y[j], t[k])
                u[k][-1][j] = uxl(y[j], t[k])

            for i in range(x.size):
                u[k][i][0] = uy0(x[i], t[k])
                u[k][i][-1] = uyl(x[i], t[k])

    for j in range(len(y)):
        u[-1][0][j] = ux0(y[j], t[-1])
        u[-1][-1][j] = uxl(y[j], t[-1])

    for i in range(len(x)):
        u[-1][i][0] = uy0(x[i], t[-1])
```

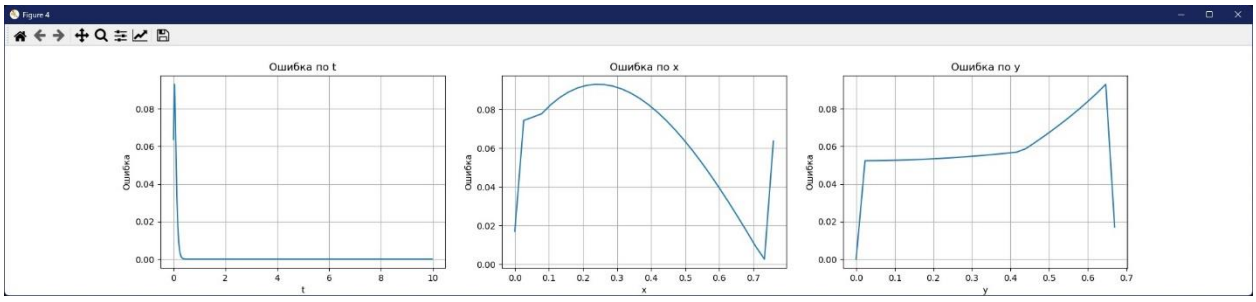
```
            u[-1][i][-1] = uyl(x[i], t[-1])

        return u


    def fractional_steps(hx, hy, tau):
        x = np.arange(0, X_MAX, hx)
        y = np.arange(0, Y_MAX, hy)
        t = np.arange(0, T_MAX, tau)
        u = np.zeros((t.size, x.size, y.size))
        u[0] = np.array([[psi(xi, yj) for yj in y] for xi in x])
        u[:, 0, :] = np.array([[ux0(yj, tk) for yj in y] for tk in t])
        u[:, -1, :] = np.array([[uxl(yj, tk) for yj in y] for tk in t])
        u[:, :, 0] = np.array([[uy0(xi, tk) for xi in x] for tk in t])
        u[:, :, -1] = np.array([[uyl(xi, tk) for xi in x] for tk in t])
        for k in range(1, t.size):
            k_half = u[k].copy()
            for j in range(1, y.size - 1):
                a = np.zeros_like(x)
                b = np.zeros_like(x)
                c = np.zeros_like(x)
                d = np.zeros_like(x)

                s = ap * tau / hx ** 2
                for i in range(1, x.size - 1):
                    a[i] = s
                    b[i] = -2 * s - 1
                    c[i] = s
                    d[i] = -u[k - 1][i][j]

                alpha = 1
                betta = 1
                gamma = 0
                delta = 1
                b[0] = betta - alpha / hx
                c[0] = alpha / hx
                d[0] = ux0(y[j], t[k] - tau / 2)

                a[-1] = - gamma / hx
                b[-1] = delta + gamma / hx
                d[-1] = uxl(y[j], t[k] - tau / 2)

                ans = equation_solve(a, b, c, d)
                for i in range(1, x.size - 1):
                    k_half[i] = ans[i]

            for j in range(y.size):
                k_half[0][j] = ux0(y[j], t[k] - tau / 2)
                k_half[-1][j] = uxl(y[j], t[k] - tau / 2)

            for i in range(1, x.size):
                a = np.zeros_like(y)
                b = np.zeros_like(y)
                c = np.zeros_like(y)
                d = np.zeros_like(y)

                tmp = ap * tau / hy ** 2
                for j in range(1, y.size - 1):
                    a[j] = s
                    b[j] = -2 * s - 1
                    c[j] = s
                    d[j] = -k_half[i][j]

                alpha = 0
```

```
            betta = 1
            gamma = 1
            delta = 0
            b[0] = betta - alpha / hy
            c[0] = alpha / hy
            d[0] = uy0(x[i], t[k])

            a[-1] = -gamma / hy
            b[-1] = delta + gamma / hy
            d[-1] = uyl(x[i], t[k])

            ans = equation_solve(a, b, c, d)
            for j in range(y.size):
                u[k][i][j] = ans[j]
        for i in range(len(x)):
            u[k][i][0] = uy0(x[i], t[k])
            u[k][i][-1] = uyl(x[i], t[k])

    return u

def analitic(nx, ny, nt):
    x = np.arange(0, X_MAX, hx)
    y = np.arange(0, Y_MAX, hy)
    t = np.arange(0, T_MAX, tau)
    return np.array([[[U(xi, yi, ti) for xi in x] for yi in y] for ti in t])


def plot_sols(nx, ny, nt, u):
    s = analitic(nx, ny, nt)
    n = 6
    x = np.arange(0, X_MAX, hx)
    y = np.arange(0, Y_MAX, hy)
    t = np.arange(0, T_MAX, tau)
    px = np.linspace(x.size//nx, nx-1, n, dtype=np.int32)
    py = np.linspace(y.size//ny, ny-1, n, dtype=np.int32)
    pt = np.linspace(t.size//nt, nt-1, n, dtype=np.int32)
    xy = np.array(list(zip(px, py)))
    xt = np.array(list(zip(px, pt)))
    yt = np.array(list(zip(py, pt)))
    fig, ax = plt.subplots(3, 2)
    fig.suptitle('Сравнение решений в плоскости x,y')
    fig.set_figheight(14)
    fig.set_figwidth(16)
    k = 0
    for i in range(3):
        for j in range(2):
            ax[i][j].set_title(f'Решение при x = {x[xy[k][0]]}, y =
{y[xy[k][1]]}')
            ax[i][j].plot(t, u[:,xy[k][0],xy[k][1]], label='Численный метод')
            ax[i][j].plot(t, s[:,xy[k][0],xy[k][1]], label='Аналитическое
решение')
            ax[i][j].grid(True)
            ax[i][j].set_xlabel('t')
            ax[i][j].set_ylabel('u')
            k += 1
    plt.legend(bbox_to_anchor=(1.05, 2), loc='upper left', borderaxespad=0.)

nx = 30
ny = 30
nt = 300
hx = X_MAX / nx
hy = Y_MAX / ny
tau = T_MAX / nt
res = alternating_directions(hx, hy, tau)
```

```
x = np.arange(0, X_MAX, hx)
y = np.arange(0, Y_MAX, hy)
t = np.arange(0, T_MAX, tau)
sol = np.array([[[U(xi, yi, ti) for yi in y] for xi in x] for ti in t])
plot_sols(nx, ny, nt, res)

fig, ax = plt.subplots(1,3)
fig.set_figheight(4)
fig.set_figwidth(20)
ax[0].set_title(f'Ошибка по t')
ax[1].set_title(f'Ошибка по x')
ax[2].set_title(f'Ошибка по y')
ax[0].set_xlabel('t')
ax[1].set_xlabel('x')
ax[2].set_xlabel('y')
ax[0].set_ylabel('Ошибка')
ax[1].set_ylabel('Ошибка')
ax[2].set_ylabel('Ошибка')
ax[0].plot(t, [np.max(abs(sol[i] - res[i])) for i in range(t.size)])
ax[1].plot(x, [np.max(abs(sol[:,i] - res[:,i])) for i in range(x.size)])
ax[2].plot(y, [np.max(abs(sol[:,:,i] - res[:,:,i])) for i in range(y.size)])
ax[0].grid(True)
ax[1].grid(True)
ax[2].grid(True)

nx = 30
ny = 30
nt = 300
hx = X_MAX / nx
hy = Y_MAX / ny
tau = T_MAX / nt
res = fractional_steps(hx, hy, tau)

x = np.arange(0, X_MAX, hx)
y = np.arange(0, Y_MAX, hy)
t = np.arange(0, T_MAX, tau)
sol = np.array([[[U(xi, yi, ti) for yi in y] for xi in x] for ti in t])
plot_sols(nx, ny, nt, res)

fig, ax = plt.subplots(1,3)
fig.set_figheight(4)
fig.set_figwidth(20)
ax[0].set_title(f'Ошибка по t')
ax[1].set_title(f'Ошибка по x')
ax[2].set_title(f'Ошибка по y')
ax[0].set_xlabel('t')
ax[1].set_xlabel('x')
ax[2].set_xlabel('y')
ax[0].set_ylabel('Ошибка')
ax[1].set_ylabel('Ошибка')
ax[2].set_ylabel('Ошибка')
ax[0].plot(t, [np.max(abs(sol[i] - res[i])) for i in range(t.size)])
ax[1].plot(x, [np.max(abs(sol[:,i] - res[:,i])) for i in range(x.size)])
ax[2].plot(y, [np.max(abs(sol[:,:,i] - res[:,:,i])) for i in range(y.size)])
ax[0].grid(True)
ax[1].grid(True)
ax[2].grid(True)
```

Результат:





Вывод:

В ходе лабораторной работы, используя *схемы переменных направлений* и *дробных шагов*, решил двумерную начально-краевую задачу для дифференциального уравнения параболического типа. В различные моменты времени вычислил погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением *u (x, t)*. Исследовал зависимость погрешности от сеточных параметров τ, h_x, h_y.