Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №8 по курсу «Численные методы»

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

Москва, 2023

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 $u(x, y, t)$. Исследовать зависимость погрешности от сеточных параметров τ и h_x, h_y.

Вариант: 16

Уравнение:

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

$$\begin{cases} u(0, y,\ t) = \sinh y \cdot e^{-3at} \\ u(\frac{\pi}{4}, y,\ t) = -2\sinh y \cdot e^{-3at} \\ u(x, 0,\ t) = \cos 2x \cdot e^{-3at} \\ u'_y(x, \ln 2,\ t) = \frac{3}{4}\cos 2x \cdot e^{-3at} \\ u(x, y, 0) = cos 2x \sinh y \end{cases}$$

Аналитическое решение:

$$u(x, y, t) = e^{-3at}\cos 2x \sinh y$$

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

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

- Считая, что значения функции $u^{k+\frac{1}{2}}_{i,j} = u(x_i, y_j, t^{k+\frac{1}{2}})$ 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: $\frac{\partial u}{\partial t}(x_i, y_j, t^{k+\frac{1}{2}}) = (1+\gamma)\frac{u^{k+1}_{i,j} - u^{k+\frac{1}{2}}_{i,j}}{\tau}$, явной аппроксимацией производной по x: $\dfrac{\partial^2 u}{\partial x^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \dfrac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2}$ и неявной аппроксимацией по y: $\dfrac{\partial^2 u}{\partial y^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \dfrac{u_{i,j-1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j+1}^{k+1}}{h_y^2}$ получим второе уравнение:

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

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.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

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.lx = args['lx']
                self.ly = args['ly']
                self.f = args['f']
                self.alpha1 = args['alpha1']
                self.alpha2 = args['alpha2']
                self.beta1 = args['beta1']
                self.beta2 = args['beta2']
                self.gamma1 = args['gamma1']
                self.gamma2 = args['gamma2']
                self.delta1 = args['delta1']
                self.delta2 = args['delta2']
                self.phi11 = args['phi11']
                self.phi21 = args['phi21']
```

```
                self.phi12 = args['phi12']
                self.phi22 = args['phi22']
                self.psi = args['psi']
                self.solution = args['solution']


class ParabolicSolver:
        def __init__(self, args, nx, ny, T, K):
                self.data = Data(args)
                self.hx = self.data.lx / nx
                self.hy = self.data.ly / ny
                self.tau = T / K
                self.x, self.y, self.t = self.prepare(nx, ny, T, K)
                self.uu = self.initalizeU(self.x, self.y, self.t)

        def getCoeffs(self, n):
                aa = np.zeros(len(n))
                bb = np.zeros(len(n))
                cc = np.zeros(len(n))
                dd = np.zeros(len(n))

                return aa, bb, cc, dd

        def computeCoeffs(self, x, y, t2, j):
                aa, bb, cc, dd = self.getCoeffs(x)
                bb[0] = self.hx * self.data.alpha2 - self.data.alpha1
                bb[-1] = self.hx * self.data.beta2 + self.data.beta1
                cc[0] = self.data.alpha1
                aa[-1] = -self.data.beta1
                dd[0] = self.data.phi11(y[j], t2) * self.hx
                dd[-1] = self.data.phi12(y[j], t2) * self.hx

                return aa, bb, cc, dd

        def prepare(self, nx, ny, T, K):
                self.hx = self.data.lx / nx
                self.hy = self.data.ly / ny
                self.tau = T / K
                x = np.arange(0, self.data.lx + self.hx, self.hx)
                y = np.arange(0, self.data.ly + self.hy, self.hy)
                t = np.arange(0, T + self.tau, self.tau)

                return x, y, t

        def initalizeU(self, x, y, t):
                u = np.zeros((len(x), len(y), len(t)))
                for i in range(len(x)):
                        for j in range(len(y)):
                                u[i][j][0] = self.data.psi(x[i], y[j])

                return u

        def analyticSolve(self, nx, ny, T, K):
                x, y, t = self.prepare(nx, ny, T, K)

                uu = np.zeros((len(x), len(y), len(t)))

                for i in range(len(x)):
                        for j in range(len(y)):
                                for k in range(len(t)):
                                        uu[i][j][k] = self.data.solution(x[i

                return uu
```

4

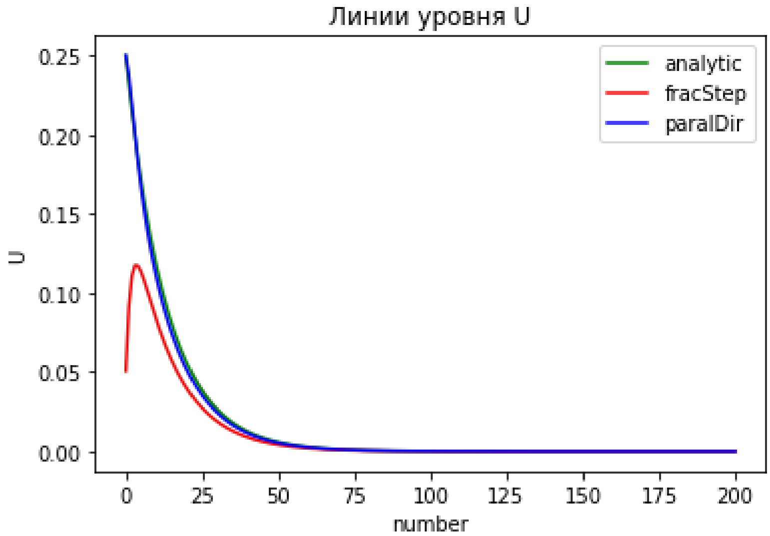
```python
    def parallelDirections_solver(self):
        for k in range(1, len(self.t)):
            u1 = np.zeros((len(self.x), len(self.y)))
            t2 = self.t[k] - self.tau / 2
            for j in range(len(self.y) - 1):
                aa, bb, cc, dd = self.computeCoeffs(self.x,
                for i in range(len(self.x) - 1):
                    aa[i] = self.data.a - self.hx * self
                    bb[i] = self.hx ** 2 - 2 * (self.hx
                    cc[i] = self.data.a + self.hx * self
                    dd[i] = -2 * (self.hx ** 2) * self.u
                    - self.data.b * (self.hx ** 2) * (se

                    - self.data.d * (self.hx ** 2) * (se
                    - (self.hx ** 2) * self.data.f(self.

                xx = tma(aa, bb, cc, dd)
                for i in range(len(self.x)):
                    u1[i][j] = xx[i]
                    u1[i][0] = (self.data.phi21(self.x[i
                                self.data.gamma2 - s
                    u1[i][-1] = (self.data.phi22(self.x[
                                self.data.delta2 + s
            for j in range(len(self.y)):
                u1[0][j] = (self.data.phi11(self.y[j], t2) -
                                self.data.alpha2 - s
                u1[-1][j] = (self.data.phi12(self.y[j], t2)
                                self.data.beta2 + se
            u2 = np.zeros((len(self.x), len(self.y)))
            for i in range(len(self.x) - 1):
                aa, bb, cc, dd = self.getCoeffs(self.y)
                bb[0] = self.hy * self.data.gamma2 - self.da
                bb[-1] = self.hy * self.data.delta2 + self.c
                cc[0] = self.data.gamma1
                aa[-1] = -self.data.delta1
                dd[0] = self.data.phi21(self.x[i], self.t[k]
                dd[-1] = self.data.phi22(self.x[i], self.t[k

                for j in range(len(self.y) - 1):
                    aa[j] = self.data.b - self.hy * self
                    bb[j] = self.hy ** 2 - 2 * (self.hy
                    cc[j] = self.data.b + self.hy * self
                    dd[j] = -2 * (self.hy ** 2) * u1[i][
                    - self.data.a * (self.hy ** 2) * (u1

                    - self.data.c * (self.hy ** 2) * (u1
                    - (self.hy ** 2) * self.data.f(self.
                xx = tma(aa, bb, cc, dd)
                for j in range(len(self.y)):
                    u2[i][j] = xx[j]
                    u2[0][j] = (self.data.phi11(self.y[j
                                    self.data.al
                    u2[-1][j] = (self.data.phi12(self.y[
                                    self.data.be
            for i in range(len(self.x)):
                u2[i][0] = (self.data.phi21(self.x[i], self.
                                self.data.gamma2 - s
                u2[i][-1] = (self.data.phi22(self.x[i], self
                                self.data.delta2 + s
            for i in range(len(self.x)):
                for j in range(len(self.y)):
                    self.uu[i][j][k] = u2[i][j]
        return self.uu
```

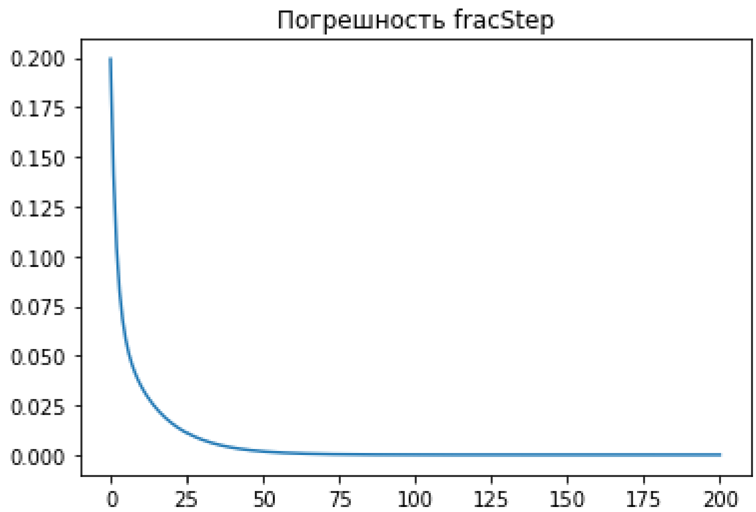
```
        def fractionalSteps_solver(self):
                for k in range(len(self.t)):
                        u1 = np.zeros((len(self.x), len(self.y)))
                        t2 = self.t[k] - self.tau / 2
                        for j in range(len(self.y) - 1):
                                aa, bb, cc, dd = self.computeCoeffs(self.x,
                                for i in range(len(self.x) - 1):
                                        aa[i] = self.data.a
                                        bb[i] = -(self.hx ** 2) / self.tau -
                                        cc[i] = self.data.a
                                        dd[i] = -(self.hx ** 2) * self.uu[i]

                                xx = tma(aa, bb, cc, dd)
                                for i in range(len(self.x)):
                                        u1[i][j] = xx[i]
                                        u1[i][0] = (self.data.phi21(self.x[i
                                                        self.data.gamma2 - s
                                        u1[i][-1] = (self.data.phi22(self.x[
                                                        self.data.delta2 + s
                        for j in range(len(self.y)):
                                u1[0][j] = (self.data.phi11(self.y[j], t2) -
                                                self.data.alpha2 - self.data
                                u1[-1][j] = (self.data.phi12(self.y[j], t2)
                                                self.data.beta2 + self.data.
                        #####
                        u2 = np.zeros((len(self.x), len(self.y)))
                        for i in range(len(self.x) - 1):
                                aa, bb, cc, dd = self.getCoeffs(self.y)
                                bb[0] = self.hy * self.data.gamma2 - self.da
                                bb[-1] = self.hy * self.data.delta2 + self.d
                                cc[0] = self.data.gamma1
                                aa[-1] = -self.data.delta1
                                dd[0] = self.data.phi21(self.x[i], self.t[k]
                                dd[-1] = self.data.phi22(self.x[i], self.t[k

                                for j in range(len(self.y) - 1):
                                        aa[j] = self.data.b
                                        bb[j] = -(self.hy ** 2) / self.tau -
                                        cc[j] = self.data.b
                                        dd[j] = -(self.hy ** 2) * u1[i][j] /
                                xx = tma(aa, bb, cc, dd)
                                for j in range(len(self.y)):
                                        u2[i][j] = xx[j]
                                        u2[0][j] = (self.data.phi11(self.y[j
                                                        self.data.alpha2 - s
                                        u2[-1][j] = (self.data.phi12(self.y[
                                                        self.data.beta2 + se
                        for i in range(len(self.x)):
                                u2[i][0] = (self.data.phi21(self.x[i], self.
                                                self.data.gamma2 - self.data
                                u2[i][-1] = (self.data.phi22(self.x[i], self
                                                self.data.delta2 + self.data
                        for i in range(len(self.x)):
                                for j in range(len(self.y)):
                                        self.uu[i][j][k] = u2[i][j]
                return self.uu

def presontation(dict_, data, args, y_point, time):
```

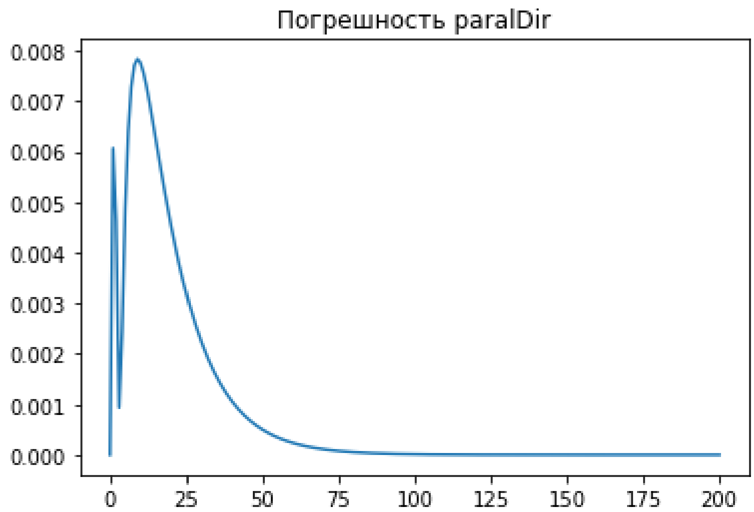


```
  plt.title('Линии уровня U')
  plt.plot(dict_['analytic'][time][y_point], color='g', label='analytic')
  plt.plot(dict_['fracStep'][time][y_point], color='r', label='fracStep')
  plt.plot(dict_['paralDir'][time][y_point], color='b', label='paralDir')
  plt.legend(loc='best')
```

```
  plt.ylabel('U')
  plt.xlabel('number')
  plt.show()


  plt.title('Погрешность fracStep')
  plt.plot(abs(dict_['analytic'][time][y_point] - dict_['fracStep'][time][y_
  plt.show()


  plt.title('Погрешность paralDir')
  plt.plot(abs(dict_['analytic'][time][y_point] - dict_['paralDir'][time][y_
  plt.show()


data = {'a': 1, 'nx': 40, 'ny': 40, 'T': 5, 'K': 200}
a, nx, ny, T, K = int(data['a']), int(data['nx']), int(data['ny']), int(data
args = {
        'a': a,
        'b': a,
        'c': 0,
        'd': 0,
        'lx': np.pi / 4,
        'ly': np.log(2),
        'f': lambda x, y, t: 0,
        'alpha1': 0,
        'alpha2': 1,
        'beta1': 1,
        'beta2': 0,
        'gamma1': 1,
        'gamma2': 0,
        'delta1': 0,
        'delta2': 1,
        'phi11': lambda y, t: np.sinh(y) * np.exp(-3 * a * t),
        'phi12': lambda y, t: -2 * np.sinh(y) * np.exp(-3 * a * t),
        'phi21': lambda x, t: np.cos(2 * x) * np.exp(-3 * a * t),
        'phi22': lambda x, t: 3 / 4 * np.cos(2 * x) * np.exp(-3 * a * t),
        'psi': lambda x, y: np.cos(2 * x) * np.sinh(y),
        'solution': lambda x, y, t: np.cos(2 * x) * np.sinh(y) * np.exp(-3 *
}

solver = ParabolicSolver(args, nx, ny, T, K)
solverFrac = ParabolicSolver(args, nx, ny, T, K)
solverParal = ParabolicSolver(args, nx, ny, T, K)
ans = {
        'fracStep': solverFrac.fractionalSteps_solver(),
        'paralDir': solverParal.parallelDirections_solver(),
        'analytic': solver.analyticSolve(nx, ny, T, K)
}

presontation(ans, data, args, 20, 20)
```







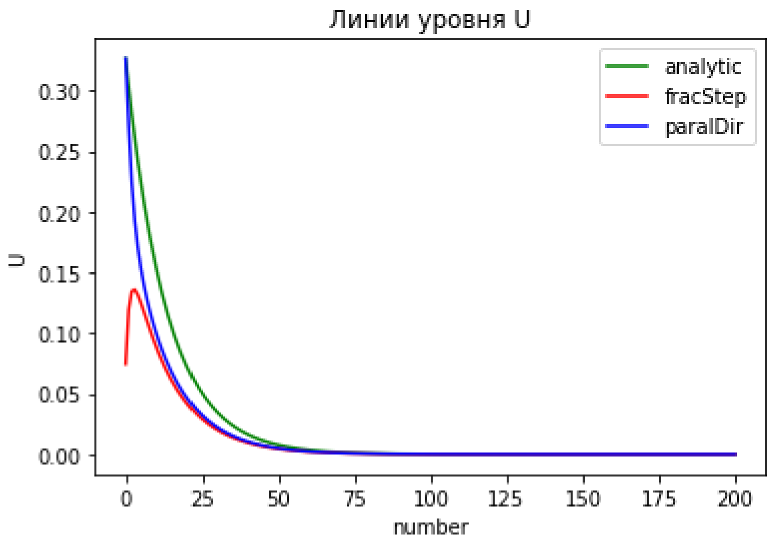
Исследование зависимости погрешности от параметров tau, hx, hy

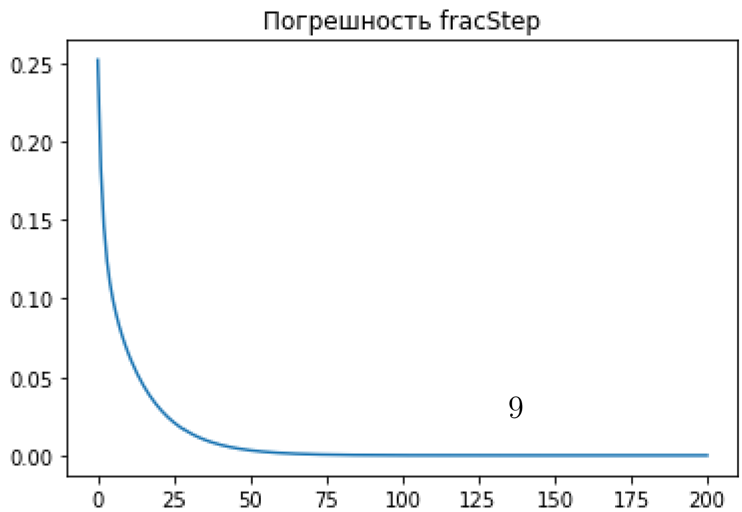
In [ ]:

```
data = {'a': 1, 'nx': 80, 'ny': 40, 'T': 5, 'K': 200}
a, nx, ny, T, K = int(data['a']), int(data['nx']), int(data['ny']), int(data
args = {
        'a': a,
        'b': a,
        'c': 0,
        'd': 0,
        'lx': np.pi / 4,
```

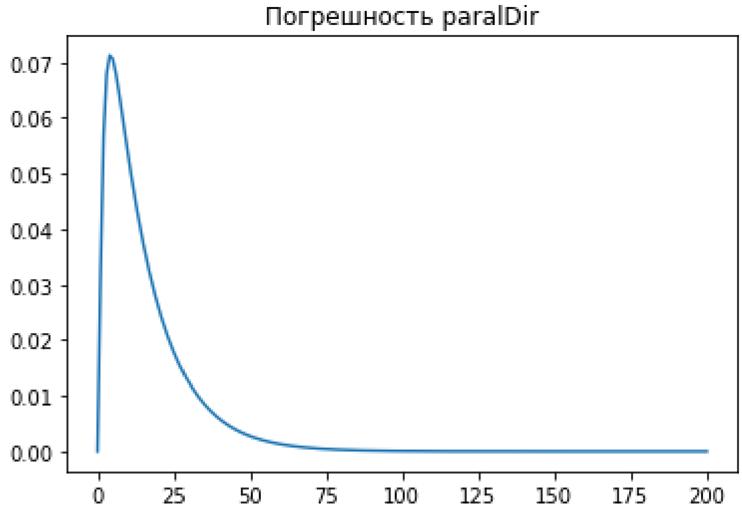
8

```
        'ly': np.log(2),
        'f': lambda x, y, t: 0,
        'alpha1': 0,
        'alpha2': 1,
        'beta1': 1,
        'beta2': 0,
        'gamma1': 1,
        'gamma2': 0,
        'delta1': 0,
        'delta2': 1,
        'phi11': lambda y, t: np.sinh(y) * np.exp(-3 * a * t),
        'phi12': lambda y, t: -2 * np.sinh(y) * np.exp(-3 * a * t),
        'phi21': lambda x, t: np.cos(2 * x) * np.exp(-3 * a * t),
        'phi22': lambda x, t: 3 / 4 * np.cos(2 * x) * np.exp(-3 * a * t),
        'psi': lambda x, y: np.cos(2 * x) * np.sinh(y),
        'solution': lambda x, y, t: np.cos(2 * x) * np.sinh(y) * np.exp(-3 *
}

solver = ParabolicSolver(args, nx, ny, T, K)
solverFrac = ParabolicSolver(args, nx, ny, T, K)
solverParal = ParabolicSolver(args, nx, ny, T, K)
ans = {
        'fracStep': solverFrac.fractionalSteps_solver(),
        'paralDir': solverParal.parallelDirections_solver(),
        'analytic': solver.analyticSolve(nx, ny, T, K)
}

presontation(ans, data, args, 20, 20)
```

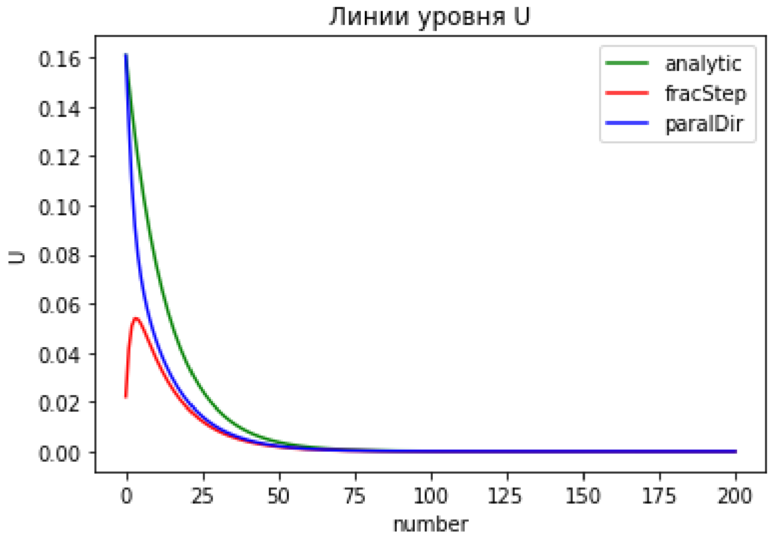


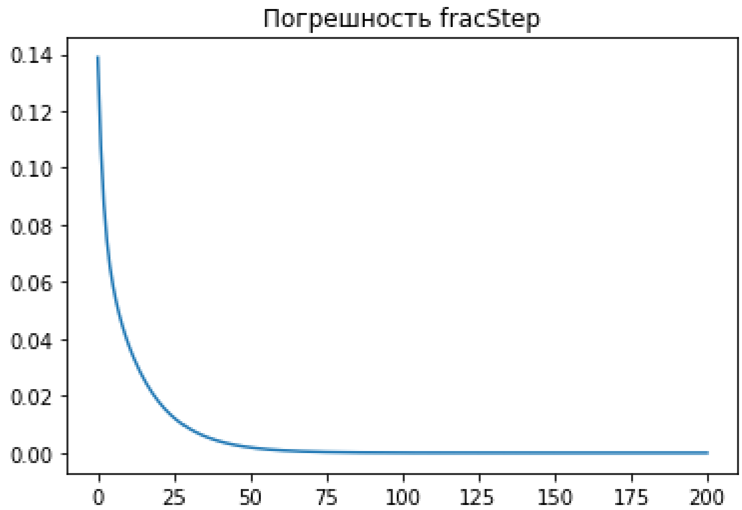


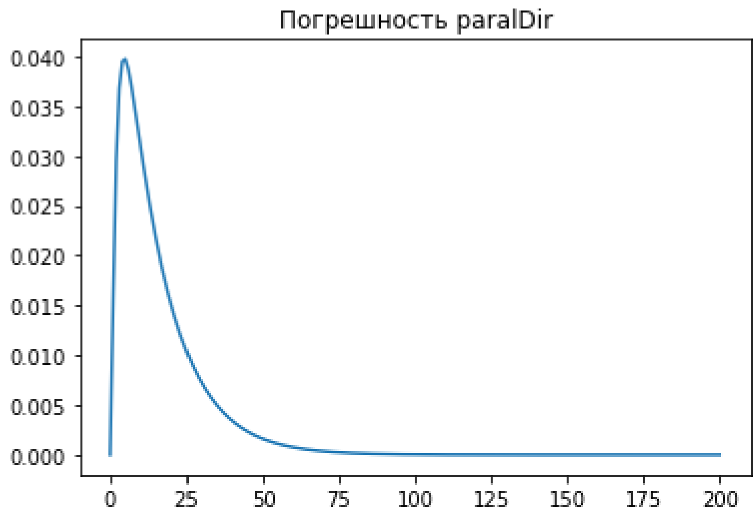
9



In [ ]:
```
data = {'a': 1, 'nx': 80, 'ny': 80, 'T': 5, 'K': 200}
a, nx, ny, T, K = int(data['a']), int(data['nx']), int(data['ny']), int(data
args = {
        'a': a,
        'b': a,
        'c': 0,
        'd': 0,
        'lx': np.pi / 4,
        'ly': np.log(2),
        'f': lambda x, y, t: 0,
        'alpha1': 0,
        'alpha2': 1,
        'beta1': 1,
        'beta2': 0,
        'gamma1': 1,
        'gamma2': 0,
        'delta1': 0,
        'delta2': 1,
        'phi11': lambda y, t: np.sinh(y) * np.exp(-3 * a * t),
        'phi12': lambda y, t: -2 * np.sinh(y) * np.exp(-3 * a * t),
        'phi21': lambda x, t: np.cos(2 * x) * np.exp(-3 * a * t),
        'phi22': lambda x, t: 3 / 4 * np.cos(2 * x) * np.exp(-3 * a * t),
        'psi': lambda x, y: np.cos(2 * x) * np.sinh(y),
        'solution': lambda x, y, t: np.cos(2 * x) * np.sinh(y) * np.exp(-3 *
}

solver = ParabolicSolver(args, nx, ny, T, K)
solverFrac = ParabolicSolver(args, nx, ny, T, K)
solverParal = ParabolicSolver(args, nx, ny, T, K)
ans = {
        'fracStep': solverFrac.fractionalSteps_solver(),
        'paralDir': solverParal.parallelDirections_solver(),
        'analytic': solver.analyticSolve(nx, ny, T, K)
}

presontation(ans, data, args, 20, 20)
```



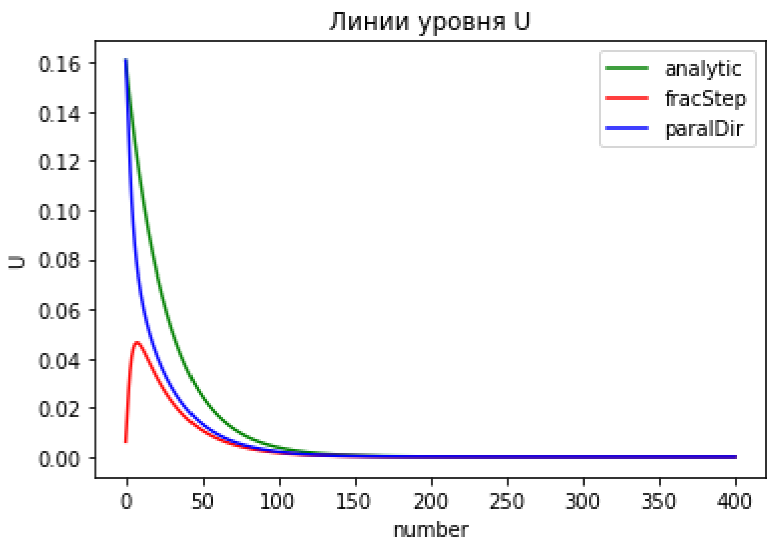


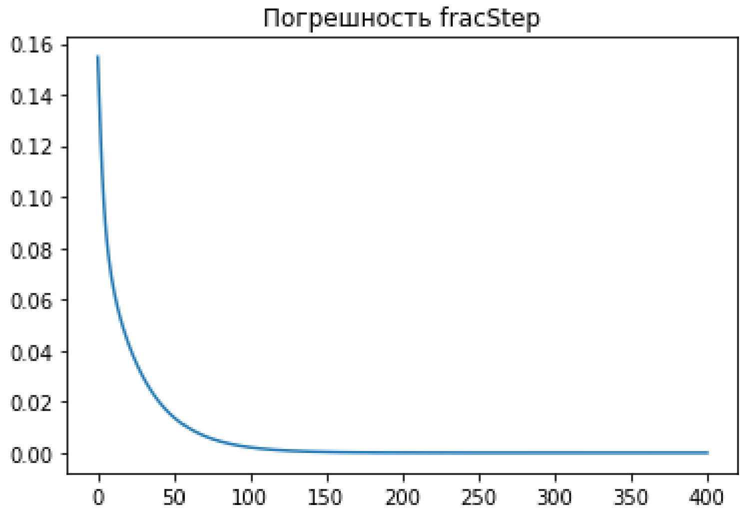


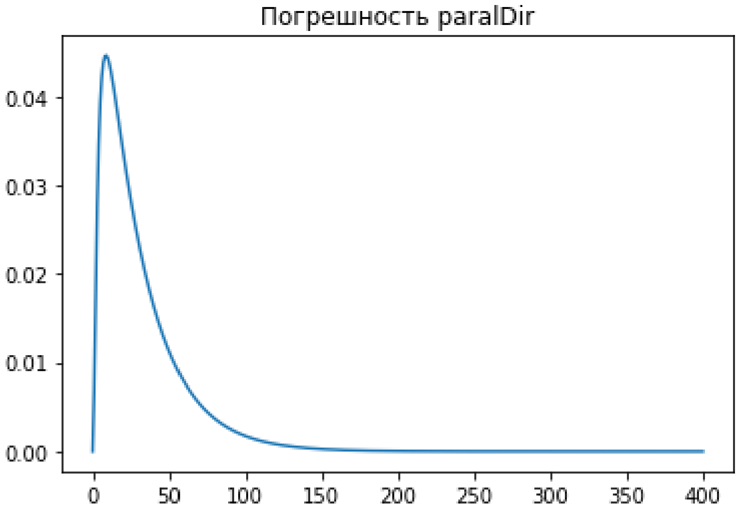
In [ ]:

```
data = {'a': 1, 'nx': 80, 'ny': 80, 'T': 5, 'K': 400}
a, nx, ny, T, K = int(data['a']), int(data['nx']), int(data['ny']), int(data
args = {
        'a': a,
        'b': a,
        'c': 0,
        'd': 0,
        'lx': np.pi / 4,
        'ly': np.log(2),
        'f': lambda x, y, t: 0,
        'alpha1': 0,
        'alpha2': 1,
        'beta1': 1,
```

```
        'beta2': 0,
        'gamma1': 1,
        'gamma2': 0,
        'delta1': 0,
        'delta2': 1,
        'phi11': lambda y, t: np.sinh(y) * np.exp(-3 * a * t),
        'phi12': lambda y, t: -2 * np.sinh(y) * np.exp(-3 * a * t),
        'phi21': lambda x, t: np.cos(2 * x) * np.exp(-3 * a * t),
        'phi22': lambda x, t: 3 / 4 * np.cos(2 * x) * np.exp(-3 * a * t),
        'psi': lambda x, y: np.cos(2 * x) * np.sinh(y),
        'solution': lambda x, y, t: np.cos(2 * x) * np.sinh(y) * np.exp(-3 *
}

solver = ParabolicSolver(args, nx, ny, T, K)
solverFrac = ParabolicSolver(args, nx, ny, T, K)
solverParal = ParabolicSolver(args, nx, ny, T, K)
ans = {
        'fracStep': solverFrac.fractionalSteps_solver(),
        'paralDir': solverParal.parallelDirections_solver(),
        'analytic': solver.analyticSolve(nx, ny, T, K)
}

presontation(ans, data, args, 20, 20)
```






Погрешность paralDir
0.04
0.03
0.02
0.01
0.00
0
50
100
150
200
250
300
350
400

Вывод Как видно на графиках погрешности, МДШ на своём старте имеет высокую погрешность, но она монотонно убывает, стремясь к 0, и уже к 0.25 пути имеет приемлемо низкий показатель. МПН же в 0 точке имеет краевое значение функций, однако уже на следующем шаге погрешность кратковременно возрастает, однако после скачка, также как и МДШ, монотонно стремится к 0. Из исследования погрешности можно сделать вывод, что она зависит от мелкости параметров $\mathrm{h}_x h_y, , 0, , .$