Гафни Даниил, 306 группа, физический факультет МГУ

Решение двумерного уравнения теплопроводности

```
# Постановка задачи
```

Постановка задачи

Используя метод переменных направлений, решите краевую задачу:

$$\begin{cases} \frac{\partial u}{\partial t} = \Delta u + (yt)^2, \ \ 0 < x < 2, \ \ 0 < y < 1, \ \ t > 0 \\ \left.\frac{\partial u}{\partial x}\right|_{x=0} = \left. u\right|_{x=2} = 0 \\ \left. u\right|_{y=0} = \left. u\right|_{y=1} = 0 \\ \left. u\right|_{t=0} = \cos(\pi x/4) \cdot y(1-y) \end{cases}$$

Аналитическое решение

Используя Wolfram Mathematica, находим аналитическое решение:

Численное решение

Сетка

Введем в расчетной области сетку, используя фиктивные узлы в окрестности границ, чтобы получить второй порядок аппроксимации для условий Неймана:

$$\begin{cases} x_0 = 0; \;\; x_n = x_0 + nh_x, \;\; n = 0, 1, \ldots, N; \;\; x_N = 2 \longrightarrow h_x = \frac{2}{N-1} \\ y_0 = 0; \;\; y_m = y_0 + mh_y, \;\; m = 0, 1, \ldots, M; \;\; y_M = 1 \longrightarrow h_y = \frac{1}{M-1} \\ t_j = j\tau, \;\; j = 0, 1, \ldots, J; \;\; t_J = T \longrightarrow \tau = \frac{T}{J} \end{cases}$$

На данной сетке будем рассматривать сеточную функцию $w^j_{n,m} = u(x_n, y_m, t_j)$.

Аппроксимации

Оператор Лапласа

Аппроксимируем оператор Лапласа $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ разностным оператором $\Lambda w = \Lambda_x w + \Lambda_y w$, где

$$\Lambda_x w = \frac{w_{n-1,m} - 2w_{n,m} + w_{n+1,m}}{h_x^2},$$
$$\Lambda_y w = \frac{w_{n,m-1} - 2w_{n,m} + w_{n,m+1}}{h_y^2}.$$

Данная аппроксимация имеет второй порядок аппроксимации.

Здесь и далее в соответствующих ситуациях для краткости верхний индекс j, соответствующий времени, может быть негласно опущен, как и другие.

Неоднородность

$$f(y,t) = (yt)^2 \longrightarrow f^j_{n,m} = (mjh_yh_t)^2,$$
$$\text{где} \;\; m = 0, 1, \ldots, M, \;\; j = 0, 1, \ldots, J.$$

Неоднородность аппроксимируется точно.

Начальное условие

$$u|_{t=0} = \cos(\pi x/4) \cdot y(1-y) \longrightarrow w^0_{n,m} = \cos(\pi n h_x/4) \cdot m h_y (1 - m h_y)$$

Начальное условие аппроксимируется точно.

Граничное условие

- По x : $\begin{cases} w_{0,m} = w_{1,m} \\ w_{N,m} = 0 \end{cases} \qquad m = 0, 1, \ldots, M$
- По y : $\begin{cases} w_{n,0} = 0 \\ w_{n,M} = 0 \end{cases} \qquad n = 0, 1, \ldots, N$

Условие при $x = 0$ имеет первый порядок аппроксимации; остальные аппроксимируются точно.

Метод переменных направлений

В данном методе переход со слоя j на слой $j+1$ осуществляется в два этапа, с помощью вспомогательного промежуточного слоя $j+1/2$. Схема переменных направлений безусловно устойчива при любых шагах h_x, h_y, τ. При условии, что для начальных и граничных условий порядки аппроксимации будут не ниже первого, и с учетом вышеописанной аппркосимации дифференциальных операторов, которая имеет первый порядок, метод переменных направлений будет давать первый порядок аппроксимации в данном случае.

Рассмотрим подробно переход со слоя j на промежуточный слой $j+1/2$ и дальнейший переход с промежуточного слоя $j+1/2$ на слой $j+1$.

Переход $j \longrightarrow j+1/2$:

Пусть значения на слое j уже известны (на самом первом шаге значения $w\,_{n,m}^{\,0}$ известны из начального условия). Перейдем на вспомогательный промежуточный слой $j+1/2$, используя **неявную схему по переменной x и явную - по переменной y :**

- Заменим выражение $\frac{\partial^2}{\partial x^2}$ разностным аналогом, взятым на слое $j+1/2: \quad \Lambda_x w^{\,j+1/2}$.
- А выражение $\frac{\partial^2}{\partial y^2}$ разностным аналогом, взятым на слое $j: \quad \Lambda_y w^{\,j}$.

При этом неоднороднось $f(x,y,t)$ в правой части уравнения аппроксимируем на промежуточным слое $j+1/2$.

В результате придем к разностному уравнению:

$$\frac{w^{\,j+1/2}-w^{\,j}}{0.5\tau}=\Lambda_x w^{\,j+1/2}+\Lambda_y w^{\,j}+f^{j+1/2}$$

Перейдем к конкретной задаче и добавим соответствующее граничное условие:

$$\begin{cases} w\,_{n,m}^{\,j+1/2}-w\,_{n,m}^{\,j}=\left(\frac{\tau}{2h_x{}^2}\,w\,_{n+1,\,m}^{\,j+1/2}-\frac{\tau}{h_x{}^2}\,w\,_{n,\,m}^{\,j+1/2}+\frac{\tau}{2h_x{}^2}\,w\,_{n-1,\,m}^{\,j+1/2}\right)+\left(\frac{\tau}{2h_y{}^2}\,w\,_{n,\,m+1}^{\,j}\right. \\ w\,_{0,m}^{\,j+1/2}=w\,_{1,m}^{\,j+1/2}, \quad w\,_{N,m}^{\,j+1/2}=w\,_{N-1,m}^{\,j+1/2} \end{cases}$$

При каждом фиксированным $n=0,1,\ldots,N-1$ можно переписать:

$$\begin{cases} \frac{\tau}{2h_y{}^2}\,w\,_{n,m-1}^{\,j+1}-\left(1+\frac{\tau}{h_y{}^2}\right)\,w\,_{n,m}^{\,j+1}+\frac{\tau}{2h_y{}^2}\,w\,_{n,\,m+1}^{\,j+1}=-\left[\,w\,_{n,\,m}^{\,j+1/2}+\frac{\tau}{2h_x{}^2}\,\left(w\,_{n+1,\,m}^{\,j+1/2}-\right.\right. \\ w\,_{0,m}^{\,j+1}=w\,_{1,m}^{\,j+1}, \quad w\,_{N,m}^{\,j+1}=0 \end{cases}$$

Введем обозначения:

$\chi_n=w\,_{n,m}^{\,j+1/2}, \quad \chi_{n-1}=0, \quad \chi_{n+1}=w\,_{n+1,m}^{\,j+1/2},$

$A^x=B^x=\frac{\tau}{2h_x{}^2}\,, \quad C^x=\left(1+\frac{\tau}{2h_x{}^2}\right),$

$F_n^x=\;w\,_{n,\,m}^{\,j}+\frac{\tau}{2h_y{}^2}\,\left(w\,_{n,\,m+1}^{\,j}-\;2w\,_{n,\,m}^{\,j}+\;w\,_{n,\,m-1}^{\,j}\right)+\frac{\tau}{2}(mjh_yh_t)^2\;.$

Получим простую систему, состоящую из уравнения, в котором неизвестные связаны рекуррентным соотношением, и граничных условий:

$$\begin{cases} A^x\chi_{n-1} - C^x\chi_n + B^x\chi_{n+1} = -F_n^x, \\ \chi_0 = \chi_1, \quad \chi_N = \chi_{N-1}. \end{cases} \quad n = 1, \ldots, N-1$$

Данную систему можно решить методом прогонки.

И снова получим простую систему уже для перехода $j + 1/2 \longrightarrow j + 1$, состоящую из уравнения, в котором неизвестные связаны рекуррентным соотношением, и граничных условий:

$$\begin{cases} A^y\gamma_{n-1} - C^y\gamma_n + B^y\gamma_{n+1} = -F_m^y, \\ \gamma_0 = \gamma_1, \quad \gamma_n = \gamma_{n-1}. \end{cases} \quad m = 1, \ldots, M-1$$

Данная система аналогично решается методом прогонки.

Метод прогонки

Рассмотрим систему для перехода $j \longrightarrow j + 1/2$:

$$\begin{cases} A^x\chi_{n-1} - C^x\chi_n + B^x\chi_{n+1} = -F_n^x, \\ \chi_0 = \chi_1, \quad \chi_N = \chi_{N-1}. \end{cases} \quad n = 1, \ldots, N-1$$

Система для перехода $j + 1/2 \longrightarrow j + 1$ будет решаться абсолютно аналогично.

Прямой ход прогонки

Идея заключается в первоначальном нахождении всех коэффицентов прогонки α_n и β_n через известные α_1 и β_1 .

Рекуррентное соотношение: $\chi_n = \alpha_{n+1}\chi_{n+1} + \beta_{n+1}$

Тогда $\chi_{n-1}(\chi_n)$: $\chi_{n-1} = \alpha_n\chi_n + \beta_n = \alpha_n\alpha_{n+1}\chi_{n+1} + \alpha_n\beta_{n+1} + \beta_n$

В результате после подстановки в первое уравнение системы, получим:

$$A^x\left(\alpha_n\alpha_{n+1}\chi_{n+1} + \alpha_n\beta_{n+1} + \beta_n\right) - C^x\left(\alpha_{n+1}\chi_{n+1} + \beta_{n+1}\right) + B^x\chi_{n+1} = -F_n^x$$

Приравняв коэффициенты при одинаковых степенях χ_{n+1}:

$$\begin{aligned} \chi_{n+1}: &\quad A^x\alpha_n\alpha_{n+1} - C^x\alpha_{n+1} + B^x = 0 \\ \chi_{n+1}^0: &\quad A^x\alpha_n\beta_{n+1} + A^x\beta_n - C^x\beta_{n+1} + F_n^x = 0 \end{aligned}$$

Выразим $\alpha_{n+1}(\alpha_n)$ и $\beta_{n+1}(\beta_n)$:

$$\alpha_{n+1} = \frac{B^x}{C^x - A^x\alpha_n},\ \ \beta_{n+1} = \frac{A^x\beta_n + F_n^x}{C^x - A^x\alpha_n},\ n = 1, 2, 3, \ldots, N-1$$

Из первых граничных условий:

$$\chi_0 = k_1\chi_1 + \mu_1 = \chi_1 \Rightarrow \alpha_1 = k_1 = 1, \beta_1 = \mu_1 = 0$$

В итоге получим формулы для прямой прогонки:

$$\begin{cases} \alpha_{n+1} = \frac{B^x}{C^x - A^x\alpha_n},\ \ \beta_{n+1} = \frac{A^x\beta_n + F_n^x}{C^x - A^x\alpha_n},\ n = 1, 2, 3, \ldots, N-1 \\ \alpha_1 = 1,\ \ \beta_1 = 0 \end{cases}$$

Обратный ход прогонки

По известному χ_N и найденымм ранее коэффициентам α_n, β_n вычисляем значения χ_n.

$$\chi_n = \alpha_{n+1}\chi_{n+1} + \beta_{n+1}$$

Из вторых граничных условий:

$$\chi_N = k_2\chi_{N-1} + \mu_2 = \chi_{N-1} \Rightarrow \ \ k_2 = 1,\ \ \mu_2 = 0$$

Откуда получим:

$$\chi_N = \frac{k_2\beta_N + \mu_2}{1 - \alpha_N k_2}$$

Используем, что $k_2 = 1, \mu_2 = 0$, и получим итоговые формулы для обратной прогонки:

$$\begin{cases} \chi_n = \alpha_{n+1}\chi_{n+1} + \beta_{n+1} \\ \chi_N = \frac{\beta_N}{1-\alpha_N} \end{cases}$$

Сложность

Как видим, здесь, для прямой прогонки необходимо $0(N)$ действий для одной системы. Поскольку систем таких $M-1$, суммарная сложность будет $O(NM)$.

Аналогично для обратной прогонки: сложность $0(M)$ для одной системы, а систем $N-1$. Таким образом, для обратной прогонки сложность будет $O(MN)$.

Суммарная сложность перехода $j+1 \longrightarrow j+1/2$ будет $O(NM)$.

Такая же сложность будет и для перехода $j+1/2 \longrightarrow j+1$.

В итоге, для перехода $j \longrightarrow j+1$ сложность будет все так же $O(NM)$, а сложность всей задачи $O(NMJ)$. Именно поэтому метод переменных направлений относится к так называемым экономичным схемам.

Экономичные схемы сочетают в себе достоинства явных и неявных схем (требуют при переходе со слоя на слой числа арифметических операций, пропорционального числу узлов сетки, и являются безусловно устойчивыми, соответственно).

Код

На языке Python 3.6.8

Импорт необходимых библиотек

[1]

```
import time

import numpy as np

import plotly
import plotly.graph_objs as go
from plotly.offline import iplot

import warnings

from IPython.display import clear_output, HTML, display, Image
```

Некоторые настройки

[2]

```
plotly.offline.init_notebook_mode()
```

```
warnings.filterwarnings('ignore')
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

# Оформление графиков plotly
layout = go.Layout(
    scene = dict( aspectmode='cube', camera = dict(eye=dict(x=-2,
y=1.5, z=1)),
    xaxis = dict(
        title='x',
        gridcolor="rgb(255, 255, 255)",
        zerolinecolor="rgb(255, 255, 255)",
        showbackground=True,
        backgroundcolor="rgb(200, 200, 230)"),

    yaxis = dict(
        title='y',
        gridcolor="rgb(255, 255, 255)",
        zerolinecolor="rgb(255, 255, 255)",
        showbackground=True,
        backgroundcolor="rgb(230, 200,230)"),

    zaxis = dict(
        title='u(x, y, t)',
        gridcolor="rgb(255, 255, 255)",
        zerolinecolor="rgb(255, 255, 255)",
        showbackground=True,
        backgroundcolor="rgb(230, 230,200)",),),
    autosize=False,
    width=800, height=600,
    margin=dict(
        r=20, b=10,
        l=10, t=10),
    )
camera = dict(
    eye=dict(x=-2, y=2, z=1)
)
```

Реализуем функцию метода прогонки

```
[3]
def TDMA(data, F):
    '''
    Метод прогонки.

    Параметры:
```

```
        coeffs (numpy.array): Трехдианональная матрица
коэффициентов уравнений.
        F (numpy.array): Массив правых частей уравнений.
    Вывод:
        (numpy.array): Массив искомых значений неизвестных.
    '''

    N = len(F)
    x = [0]*(N)
    A = []
    C = [data[0][0]]
    B = [data[0][1]]
    for i in range(1, N-1):
        A.append(data[i][i-1])
        C.append(data[i][i])
        B.append(data[i][i+1])
    A.append(data[-1][-2])
    C.append(data[-1][-1])

    alpha = [-B[0]/C[0]]*(N-1)
    beta = [F[0]/C[0]]*(N-1)

    # Прямой ход прогонки
    for i in range(1, N-1):
        alpha[i] = -B[i]/(A[i-1]*alpha[i-1] + C[i])
        beta[i] = ((F[i] - A[i-1]*beta[i-1])/(A[i-1]*alpha[i-1] +
C[i]))
    x[-1] = (F[-1] - A[-1]*beta[-1]) / (C[-1] + A[-1]*alpha[-1])

    # Обратный ход прогонки
    for i in reversed(range(N-1)):
        x[i] = alpha[i]*x[i+1] + beta[i]

    return(np.array(x))
```

Демонстрация работы метода прогонки

[4]

```
data = np.array([
    [2, -1, 0, 0, 0],
    [-3, 8, -1, 0, 0],
    [0, -5, 12, 2, 0],
    [0, 0, -6, 18, -4],
    [0, 0, 0, -5, 10]
])
F = np.array([
    -25, 72, -69, -156, 20
])
TDMA(data, F)
```

```
array([-10.,   5.,  -2., -10.,  -3.])
```

Основной класс

```
[5]
class HeatEquationSolver_2():
    '''
    Класс для численного решения двумерного уравнения
теплопроводности.
    '''
    def __init__(self,
                 X_START=0, X_END=2,
                 Y_START=0, Y_END=1,
                 T_START=0, T_END=20,
                 N=5, M=5, J=5):

        self.X_START = X_START
        self.X_END = X_END
        self.Y_START = Y_START
        self.Y_END = Y_END
        self.T_START = T_START
        self.T_END = T_END
        self.N = N
        self.M = M
        self.J = J

        self.x = np.linspace(X_START, X_END, N)
        self.y = np.linspace(Y_START, Y_END, M)
        self.t = np.linspace(T_START, T_END, J)

        self.dx = self.x[1] - self.x[0]
        self.dy = self.y[1] - self.y[0]
        self.dt = self.t[1] - self.t[0]

    def initialize(self, a=1, f=lambda x, y, t:0, fi=lambda x,
y:0,
                   alpha1x=0, alpha2x=0, beta2x=0, beta1x=0,
mu1x=lambda y, t:0, mu2x=lambda y, t:0,
                   alpha1y=0, alpha2y=0, beta2y=0, beta1y=0,
mu1y=lambda x, t:0, mu2y=lambda x, t:0):
        '''
        Задание коэффициентов и функций конкретной задачи.
        '''

        self.a = a  # Коэффициент при операторе Лапласа
        self.f = f  # Функция "источника тепла" - неоднородность
        self.fi = fi  # Начальное условие
```

```
        self.alpha1x = alpha1x  # Коэффициент при левом условии
Неймана для x
        self.alpha2x = alpha2x  # Коэффициент при правом условии
Неймана для x
        self.beta1x = beta1x  # Коэффициент при левом условии
Дирихле для x
        self.beta2x = beta2x  # Коэффициент при правом условии
Дирихле для x
        self.mu1x = mu1x # Правая часть левого граничного условия
для x
        self.mu2x = mu2x # Правая часть правого граничного
условия для x

        self.alpha1y = alpha1y  # Коэффициент при левом условии
Неймана для y
        self.alpha2y = alpha2y  # Коэффициент при правом условии
Неймана для y
        self.beta1y = beta1y  # Коэффициент при правом условии
Дирихле для y
        self.beta2y = beta2y  # Коэффициент при левом условии
Дирихле для y
        self.mu1y = mu1y # Правая часть левого граничного условия
для y
        self.mu2y = mu2y # Правая часть правого граничного
условия для y

        self.u = np.zeros((self.J, self.N, self.M))  # Создание
трехмерного массива значений u(x, y, t)

        self.u[0] = [[self.fi(x, y) for y in self.y] for x in
self.x]  # Применение начального условия

    def calculate_layer(self, j):
        '''
        Метод, который вычисляет значения функции u(x, y, t) на
временном слое номер j,
        используя предыдущий слой номер j-1.
        '''

        step_x = self.dx
        step_y = self.dt
        step_t = self.dt/2  # Уменьшим шаг по времени в 2 раза,
чтобы получить промежуточный слой

        # Переход к промежуточному слою, задание коэффициентов
для метода прогонки
        middle_layer = [np.array([0]*self.N)]*(self.M)
        for m in range(self.M):
            data = np.zeros((self.N, self.N))
            #Учет граничных условий
            data[0][0] = self.beta1x - self.alpha1x/step_x
```

```
            data[0][1] = self.alpha1x/step_x
            data[-1][-2] = -self.alpha2x/step_x
            data[-1][-1] = self.alpha2x/step_x + self.beta2x

            for i in range(1, self.N - 1):
                data[i][i-1] = -self.a**2*step_t/step_x**2
                data[i][i] = (2*self.a**2*step_t/step_x**2 + 1)
                data[i][i+1] = -self.a**2*step_t/step_x**2

            F = [0]*self.N
            F[0], F[-1] = self.mu1x(self.y[m], self.t[j]+step_t),
self.mu2x(self.y[m], self.t[j]+step_t)
            for k in range(1, self.N-1):
                F[k] = (self.f(self.x[k], self.y[m],
self.t[j]+step_t)*step_t + self.u[j-1][k][m] +
                        self.a**2*step_t/step_y**2 * (self.u[j-1]
[k-1][m] - 2*self.u[j-1][k][m] + self.u[j-1][k+1][m]))

            middle_layer[m] = TDMA(data, F)  # Применение метода
прогонки

        middle_layer = np.array(middle_layer).T  #
Транспонирование

        # Переход к новому слою, задание коэффициентов для метода
прогонки
        new_layer = [np.array([0]*self.M)]*(self.N)

        for n in range(self.N):
            data = np.zeros((self.M, self.M))
            # Учет граничных условий
            data[0][0] = self.beta1y - self.alpha1y/step_y
            data[0][1] = self.alpha1y/step_y
            data[-1][-2] = self.alpha2y/step_y
            data[-1][-1] = self.alpha2y/step_y + self.beta2y


            for i in range(1, self.M - 1):
                data[i][i-1] = -self.a**2*step_t/step_y**2
                data[i][i] = (2*self.a**2*step_t/step_y**2 + 1)
                data[i][i+1] = -self.a**2*step_t/step_y**2

            F = [0]*self.M
            F[0], F[-1] = self.mu1y(self.x[n],
self.t[j]+2*step_t), self.mu2y(self.x[n], self.t[j]+2*step_t)
            for k in range(1, self.M-1):
                F[k] = (self.f(self.x[n], self.y[k],
self.t[j]+2*step_t)*step_t + middle_layer[n][k] +
                        self.a**2*step_t/step_x**2 *
(middle_layer[n][k-1] - 2*middle_layer[n][k] + middle_layer[n]
[k+1]))
```

```
            new_layer[n] = TDMA(data, F)  # Применение метода
прогонки

        new_layer = np.array(new_layer)

        return new_layer

    def calculate_u(self):
        print('Calculating...')
        for j in range(1, self.J):
            clear_output(wait = True)
            print(f'Progress: {round(j/(self.J-1), 2)*100}%')
            new_layer = self.calculate_layer(j)
            self.u[j] = new_layer

    def plot_state(self, n = 0, filename =
'OMM_Task_2_initial_conditions',
                   online = False):
        '''
        Метод, который строит график u(x, y, t) в момент времени
на n% от временного интервала.

         Параметры:
            n (double): Число от 0 до 100 (при n > 100 считается,
что n = 100).
        '''

        num = int(round(n/100*self.J, 0))
        if num>self.J-1:
            num = self.J-1

        data = [go.Surface(x = self.x, y = self.y, z =
self.u[num].T, colorscale = 'YlGnBu')]
        fig = go.Figure(data = data, layout = layout)

        if online:
            display(plotly.plotly.iplot(fig, filename =
filename))
        if not online:
            display(plotly.offline.iplot(fig, filename =
filename))

    def plot_initial_state(self, filename = 'OMM_Task_2_state',
                   online = True):
        '''
        Метод, который строит график начального условия.
        '''

        self.plot_state(n = 0, filename = filename,
                   online = online)

    def show_evolution(self):
```

```
        '''
        Метод, который отображает анимацию эволюции u(x, y, t).
        '''

        # Create FigureWidget and add surface trace
        fig = go.Figure(layout = layout)
        surface = fig.add_surface(z=self.u[0].T, colorscale =
'YlGnBu')

        # Set axis ranges to fixed values to keep them from
retting during animation
        fig.layout.scene.zaxis.range = [np.min(self.u),
np.max(self.u)]
        fig.layout.scene.yaxis.range = [self.Y_START, self.Y_END]
        fig.layout.scene.xaxis.range = [self.X_START, self.X_END]

        frames = []
        for j in range(self.J):
            frames.append(go.Frame(data=[{'type': 'surface','x':
self.x, 'y': self.y, 'z': self.u[j].T}]))

        fig.frames = frames

        # Настройки plotly
        fig.layout.updatemenus = [
            {
                'buttons': [
                    {
                        'args': [None, {'frame': {'duration':
100, 'redraw': False},
                                  'fromcurrent': True,
'transition': {'duration': 100}}],
                        'label': 'Play',
                        'method': 'animate'
                    },
                    {
                        'args': [[None], {'frame': {'duration':
0, 'redraw': False}, 'mode': 'immediate',
                        'transition': {'duration': 0}}],
                        'label': 'Pause',
                        'method': 'animate'
                    }
                ],
                'direction': 'left',
                'pad': {'r': 10, 't': 87},
                'showactive': False,
                'type': 'buttons',
                'x': 0.55,
                'xanchor': 'right',
                'y': 0,
                'yanchor': 'top'
            }
```

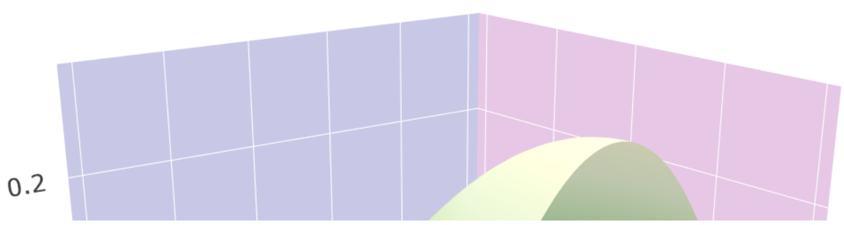
```
            ]

            display(iplot(fig))
```

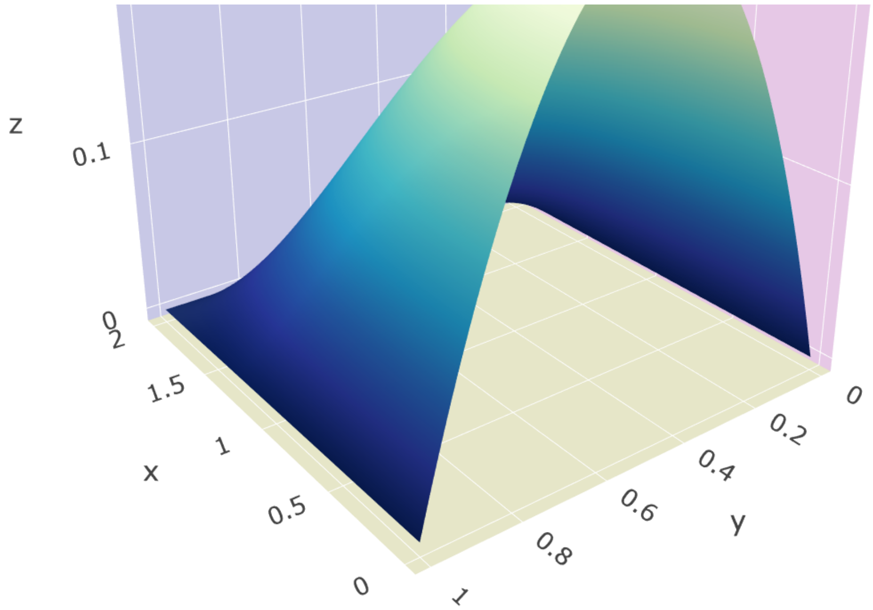
Задание параметров нашей задачи

```
[6]  solver = HeatEquationSolver_2(N = 50, M = 50, J = 50, T_END = 2)
     a = 1
     def f(x, y, t):
         return (y*t)**2
     def fi(x, y):
         return np.cos(np.pi*x/4)*y*(1-y)
     alpha1x = 1
     alpha2x = 0
     alpha1y = 0
     alpha2y = 0
     beta1x = 0
     beta2x = 1
     beta1y = 1
     beta2y = 1
     solver.initialize(a = a, f = f, fi = fi,
                       alpha1x = alpha1x, alpha2x = alpha2x, alpha1y =
     alpha1y, alpha2y = alpha2y,
                       beta1x = beta1x, beta2x = beta2x, beta1y =
     beta1y, beta2y = beta2y)
```

Начальное условие

```
[7]  solver.plot_initial_state(online = False)
```






```
None
```

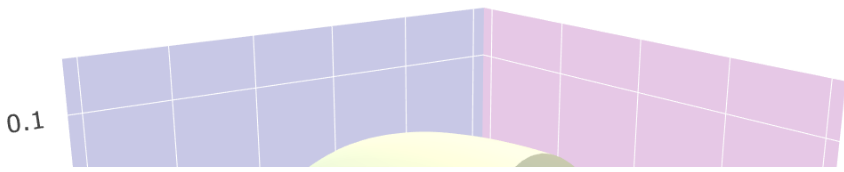
Вычисление решения

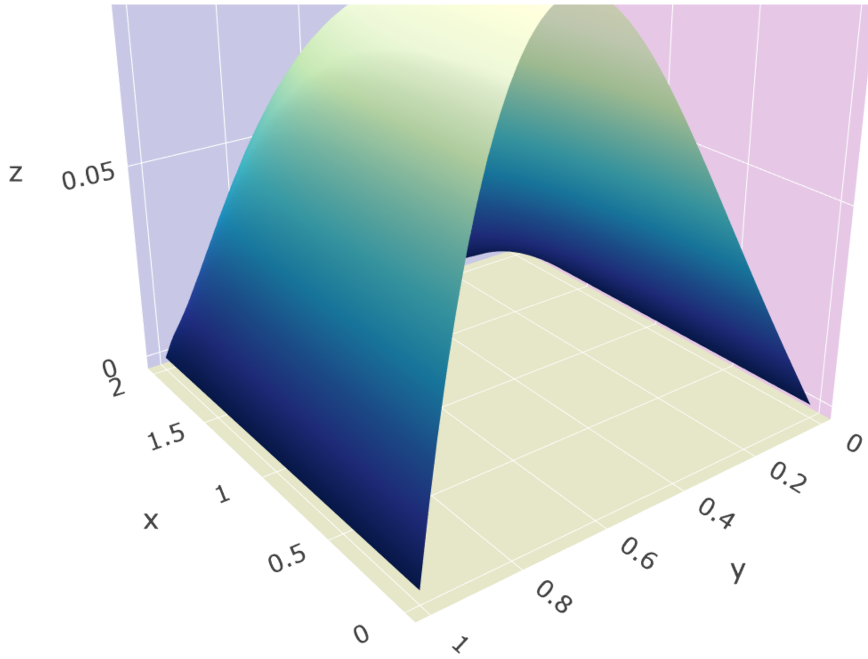
```
[ ] solver.calculate_u()
```

```
Progress: 100.0%
```

Состояние функции u(x, y, t) после того, как прошла половина времени:

```
[9] solver.plot_state(n = 50)
```

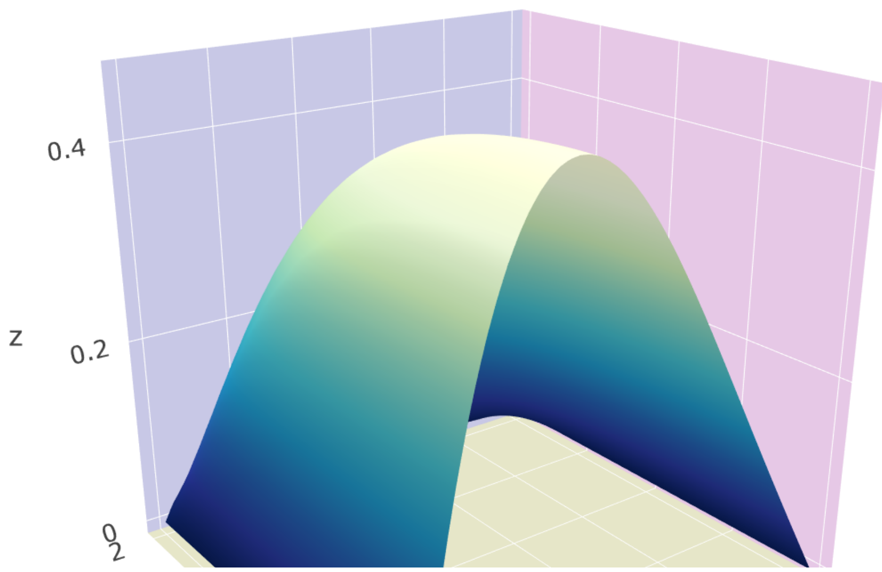


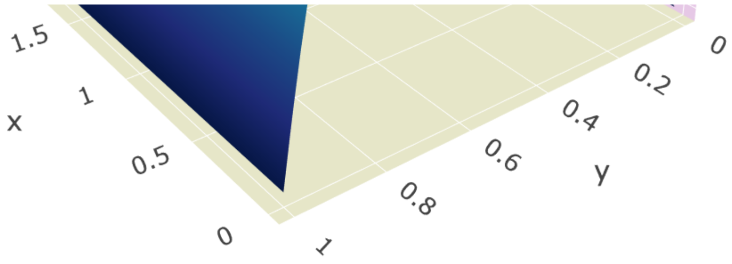


None

Конечное состояние функции u(x, y, t):

```
[10]  solver.plot_state(n = 100)
```



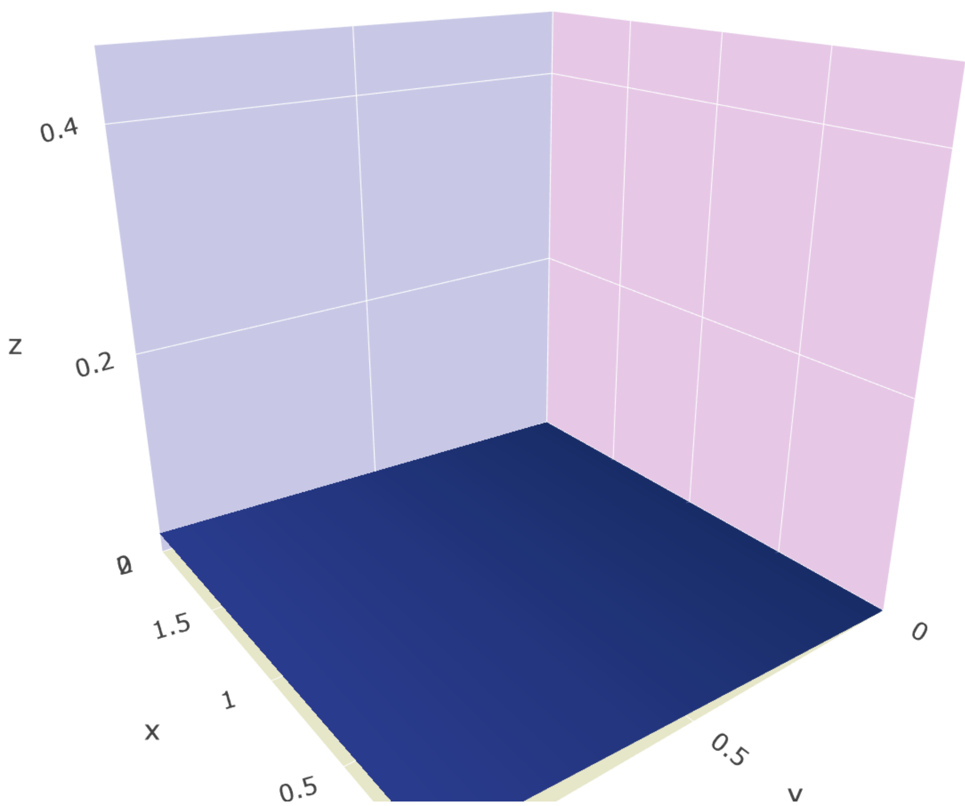


None

Анимированное численное решение

Необходимо один раз подождать полной прогрузки анимации

```
[11] solver.show_evolution()
```



Play Pause

None

😉