Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №7 по курсу «Численные методы»

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

Москва, 2023

Задание: Решить краевую задачу для дифференциального уравнения эллиптического типа. Аппроксимацию уравнения произвести с использованием центрально-разностной схемы. Для решения дискретного аналога применить следующие методы: метод простых итераций (метод Либмана), метод Зейделя, метод простых итераций с верхней релаксацией. 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: 16

Уравнение:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$\begin{cases} u'_x(0,\ y) = 0 \\ u'_x(\pi/2,\ y) = y \\ u_y(x,\ 0) = \sin x \\ u_y(x, 1) - u(\ x, 1) = 0 \end{cases}$$

Аналитическое решение:

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

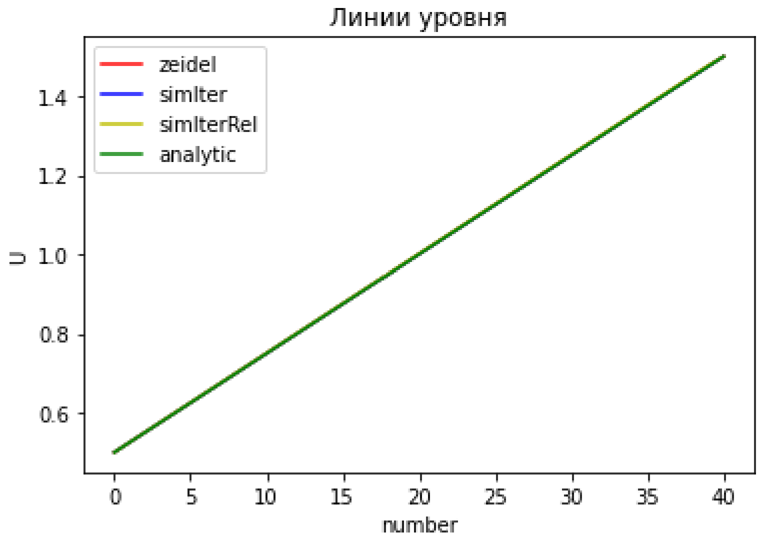
При решении эллиптических задач также используется конечно-разностная схема, однако теперь полученные СЛАУ имеют пятидиагональный вид. Для решения СЛАУ такого типа используют итерационные методы: Зейделя, простых итераций, простых итераций с верхней релаксацией. Метод простых итераций - способ численного решения математических задач. Его суть – нахождение алгоритма поиска по известному приближению (приближенному значению) искомой величины следующего, более точного приближения. Применяется в случае, когда последовательность приближений по указанному алгоритму сходится. Метод Зейделя представляет собой некоторую модификацию метода итераций. Основная его идея заключается в том, что при вычислении (k + 1)-го приближения неизвестной xi учитываются уже вычисленные ранее (k + 1)-е приближения неизвестных x1, x2, ..., xi - 1.

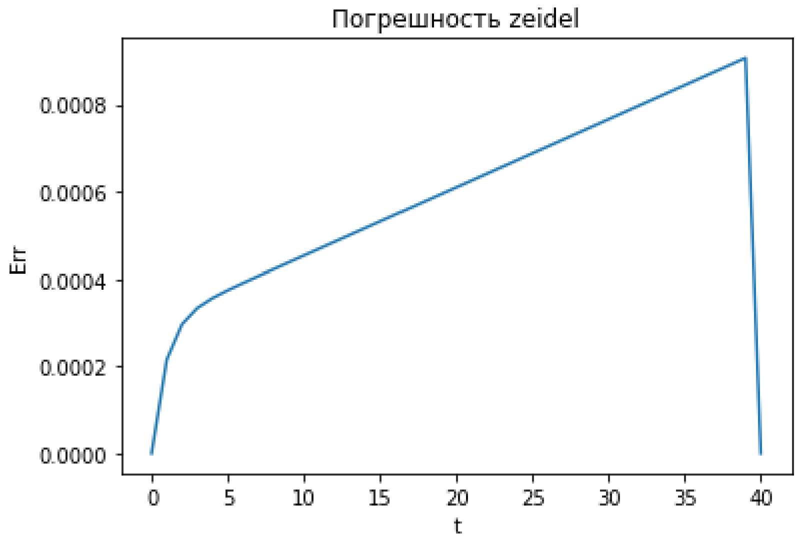
In [ ]:
```
import copy
import numpy as np
import matplotlib.pyplot as plt

def diff(L, u, nx, ny):
    mx = 0
    for i in range(nx):
        for j in range(ny):
            mx = max(mx, abs(u[i][j] - L[i][j]))
    return mx


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.w = args['w']
        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']
```

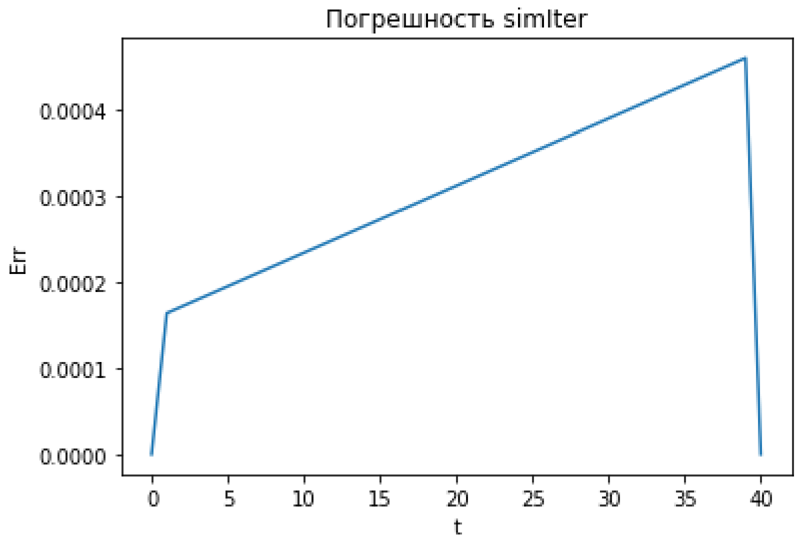
```
        self.delta1 = args['delta1']
        self.delta2 = args['delta2']
        self.phi1 = args['phi1']
        self.phi2 = args['phi2']
        self.phi3 = args['phi3']
        self.phi4 = args['phi4']
        self.solution = args['solution']


class ElepticalSolver:
    def __init__(self, args, nx, ny):
        self.data = Data(args)
        self.hx = self.data.lx / nx
        self.hy = self.data.ly / ny
        self.x = np.arange(0, self.data.lx + self.hx, self.hx)
        self.y = np.arange(0, self.data.ly + self.hy, self.hy)

        self.u = self.initalizeU(self.x, self.y)
        for i in range(1, nx):
            for j in range(1, ny):
                self.u[i][j] = self.u[0][j] + (self.x[i] - self.x[0]) * (sel
        self.iteration = 0
        self.eps = 1e-6


    def initalizeU(self, x, y):
        u = np.zeros((len(x), len(y)))
        for i in range(len(x)):
            u[i][0] = self.data.phi3(x[i]) / self.data.gamma2
            u[i][-1] = self.data.phi4(x[i]) / self.data.delta2
        for j in range(len(y)):
            u[0][j] = self.data.phi1(y[j]) / self.data.alpha2
            u[-1][j] = self.data.phi2(y[j]) / self.data.beta2

        return u

    def analyticSolve(self, nx, ny):
        self.hx = self.data.lx / nx
        self.hy = self.data.ly / ny
        x = np.arange(0, self.data.lx + self.hx, self.hx)
        y = np.arange(0, self.data.ly + self.hy, self.hy)
        u = []
        for yi in y:
            u.append([self.data.solution(xi, yi) for xi in x])
        return u

    def simpleIterationMethod_solver(self, nx, ny):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(self.u)
            u = self.initalizeU(self.x, self.y)
            for j in range(1, len(self.y) - 1):
                for i in range(1, len(self.x) - 1):
                    u[i][j] = (self.hx * self.hx * self.data.f(self.x[i], se
                               (L[i + 1][j] + L[i - 1][j]) - self.data.d * s
                               (L[i][j + 1] + L[i][j - 1]) /
                               (self.hy * self.hy) - self.data.a * self.hx *
                               (L[i + 1][j] - L[i - 1][j]) - self.data.b * s
                               (L[i][j + 1] - L[i][j - 1]) /
                               (2 * self.hy)) / (self.data.c * self.hx * sel
                                                 (self.hy * self.hy + self.c
                                                 (self.hy * self.hy))

            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
```

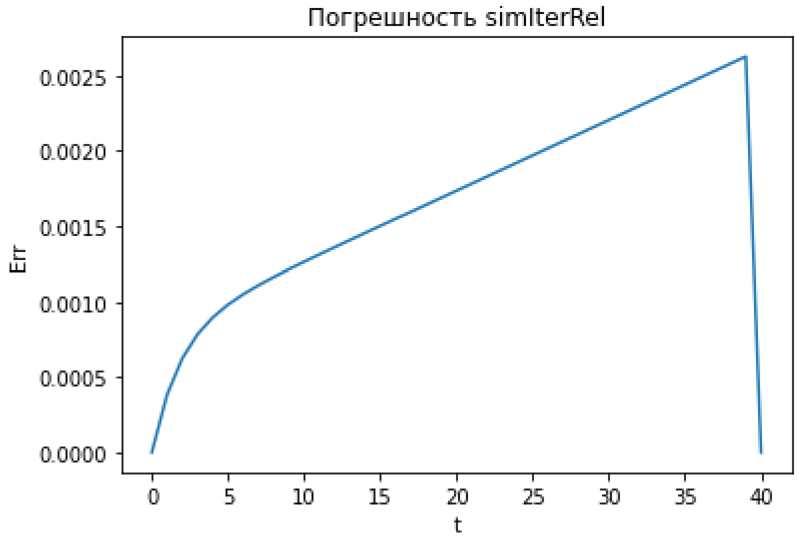
```python
            if diff(L, u, nx, ny) <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration

    def zeidelMethod_solver(self, nx, ny):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(self.u)
            u = self.initalizeU(self.x, self.y)
            for j in range(1, len(self.y) - 1):
                for i in range(1, len(self.x) - 1):
                    u[i][j] = ((self.hx ** 2) * self.data.f(self.x[i], self.
                               (L[i + 1][j] + u[i - 1][j]) - self.data.d * (
                               (L[i][j + 1] + u[i][j - 1]) / (self.hy ** 2)
                               (L[i + 1][j] - u[i - 1][j]) - self.data.b * (
                               (L[i][j + 1] - u[i][j - 1]) /
                               (2 * self.hy)) / \
                              (self.data.c * (self.hx ** 2) - 2 * (self.hy *
                               (self.hy ** 2))
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if cur_eps <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration

    def simpleIterationMethodRelaxed_solver(self, nx, ny):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(self.u)
            u = self.initalizeU(self.x, self.y)
            for j in range(1, len(self.y) - 1):
                for i in range(1, len(self.x) - 1):
                    u[i][j] = (((self.hx ** 2) * self.data.f(self.x[i], self
                                (L[i + 1][j] + u[i - 1][j]) - self.data.d *
                                (L[i][j + 1] + u[i][j - 1]) / (self.hy ** 2)
                                (L[i + 1][j] - u[i - 1][j]) - self.data.b *
                                (L[i][j + 1] - u[i][j - 1]) /
                                (2 * self.hy)) / (self.data.c * (self.hx **
                                                  (self.hy ** 2 + self.data.
                                                  (self.hy ** 2))) * self.da
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if diff(L, u, nx, ny) <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration


def compareError(a, b):
    err = 0
    lst = [abs(i - j) for i, j in zip(a, b)]
    for each in lst:
        err = max(err, each)
    return err

def presontation(dict_, time=0):
    fig = plt.figure()
    plt.title('Линии уровня')
    plt.plot(dict_['zeidel'][time], color='r', label='zeidel')
    plt.plot(dict_['simIter'][time], color='b', label='simIter')
    plt.plot(dict_['simIterRel'][time], color='y', label='simIterRel')
    plt.plot(dict_['analytic'][time], color='g', label='analytic')
```

```
    plt.legend(loc='best')
    plt.ylabel('U')
    plt.xlabel('number')
    plt.show()

    plt.title('Погрешность zeidel')
    plt.plot(abs(dict_['zeidel'][time] - dict_['analytic'][time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()

    plt.title('Погрешность simIter')
    plt.plot(abs(dict_['simIter'][time] - dict_['analytic'][time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()

    plt.title('Погрешность simIterRel')
    plt.plot(abs(dict_['simIterRel'][time] - dict_['analytic'][time]))
    plt.ylabel('Err')
    plt.xlabel('t')
    plt.show()

data = {'nx': 40, 'ny': 40}
nx, ny = int(data['nx']), int(data['ny'])
args = {
    'a': 0,
    'b': 0,
    'c': 2,
    'd': 1,
    'lx': 1,
    'ly': 1,
    'w': 1.5,
    'f': lambda x, y: 0,
    'alpha1': 0,
    'alpha2': 1,
    'beta1': 0,
    'beta2': 1,
    'gamma1': 0,
    'gamma2': 1,
    'delta1': 0,
    'delta2': 1,
    'phi1': lambda y: y,
    'phi2': lambda y: 1 + y,
    'phi3': lambda x: x,
    'phi4': lambda x: 1 + x,
    'solution': lambda x, y: x + y
}

analSolver = ElepticalSolver(args, nx, ny)
simIterSolver = ElepticalSolver(args, nx, ny)
zeidelSolver = ElepticalSolver(args, nx, ny)
simIterRelSolver = ElepticalSolver(args, nx, ny)

ans = {
    'analytic': analSolver.analyticSolve(nx, ny),
    'simIter': simIterSolver.simpleIterationMethod_solver(nx, ny)[0],
    'zeidel': zeidelSolver.zeidelMethod_solver(nx, ny)[0],
    'simIterRel': simIterRelSolver.simpleIterationMethodRelaxed_solver(nx, r
}

presontation(ans, 20)
```


Линии уровня
zeidel
simIter
simIterRel
analytic
U
1.4
1.2
1.0
0.8
0.6
0
5
10
15
20
25
30
35
40
number


Погрешность zeidel
Err
0.0008
0.0006
0.0004
0.0002
0.0000
0
5
10
15
20
25
30
35
40
t

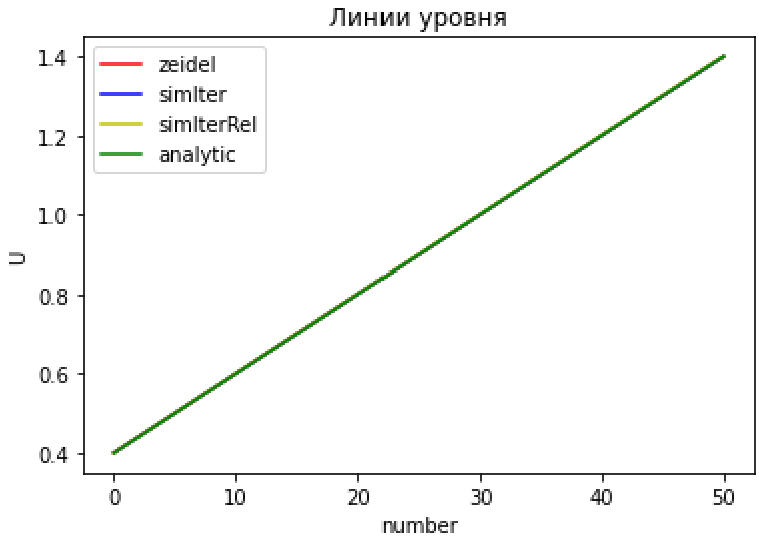

Погрешность simIter
Err
0.0004
0.0003
0.0002
0.0001
0.0000
0
5
10
15
20
25
30
35
40
t

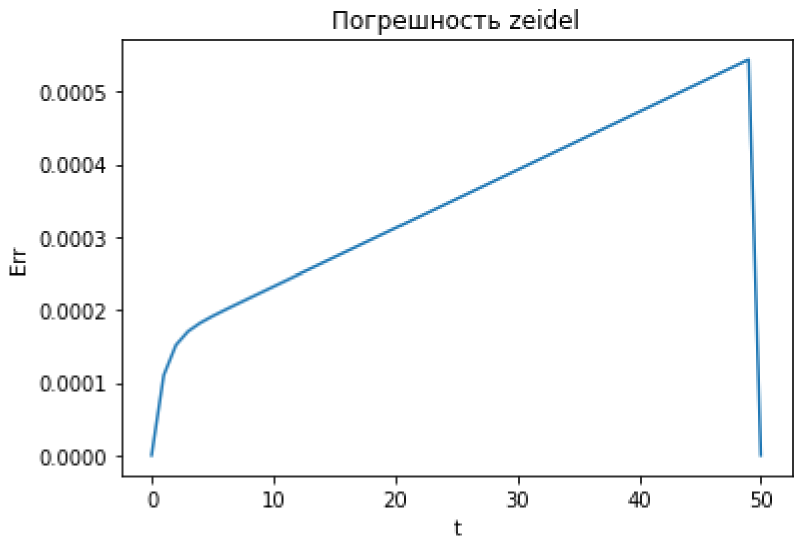


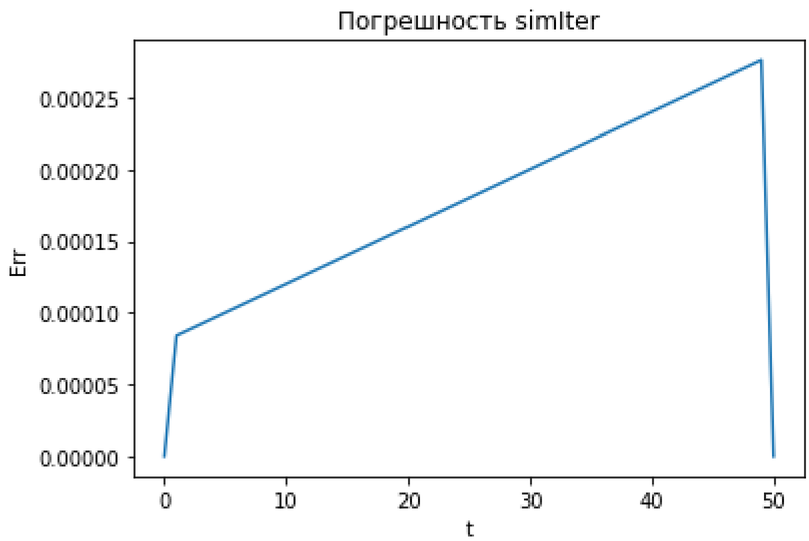
Исследование зависимости погрешности от параметров hx, hy

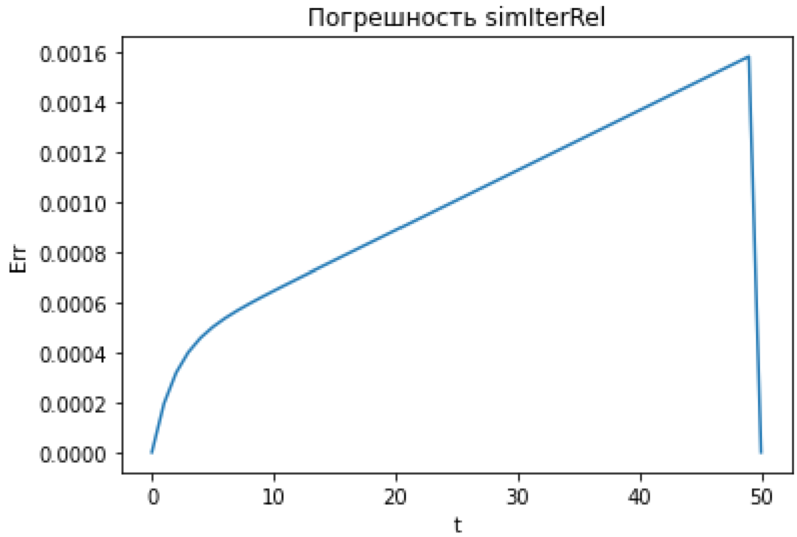
In [ ]:

```
data = {'nx': 50, 'ny': 50}
nx, ny = int(data['nx']), int(data['ny'])
args = {
    'a': 0,
    'b': 0,
    'c': 2,
    'd': 1,
    'lx': 1,
    'ly': 1,
    'w': 1.5,
    'f': lambda x, y: 0,
    'alpha1': 0,
    'alpha2': 1,
    'beta1': 0,
    'beta2': 1,
    'gamma1': 0,
    'gamma2': 1,
    'delta1': 0,
    'delta2': 1,
    'phi1': lambda y: y,
    'phi2': lambda y: 1 + y,
    'phi3': lambda x: x,
    'phi4': lambda x: 1 + x,
    'solution': lambda x, y: x + y
}

analSolver = ElepticalSolver(args, nx, ny)
simIterSolver = ElepticalSolver(args, nx, ny)
zeidelSolver = ElepticalSolver(args, nx, ny)
simIterRelSolver = ElepticalSolver(args, nx, ny)

ans = {
    'analytic': analSolver.analyticSolve(nx, ny),
    'simIter': simIterSolver.simpleIterationMethod_solver(nx, ny)[0],
    'zeidel': zeidelSolver.zeidelMethod_solver(nx, ny)[0],
    'simIterRel': simIterRelSolver.simpleIterationMethodRelaxed_solver(nx, n
}

presontation(ans, 20)
```


Линии уровня
zeidel
simIter
simIterRel
analytic
U
number
0.4
0.6
0.8
1.0
1.2
1.4
0
10
20
30
40
50


Погрешность zeidel
Err
t
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
0
10
20
30
40
50


Погрешность simIter
Err
t
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0
10
20
30
40
50



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.

Вывод Погрешность итерационных методов задаётся не только мелкостью шагов, но и желаемой точностью пользователя (eps), до которой за максимальное n количество итераций должны прийти алгоритмы. Исследование зависимости погрешности от мелкости hx и hy также приводит к уменьшению погрешности при уменьшению мелкости шагов.