Лабораторная работа №7

Студент **Лошманов Ю. А.**

Группа М8О-406Б-19

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. Вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x, y)$. Исследовать зависимость погрешности от сеточных параметров h_x, h_y.

Вариант 6

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

$$u(0,\ y) = 0$$

$$u(\tfrac{\pi}{2},\ y) = y$$

$$u_y(x,\ 0) = \sin x$$

$$u_y(x,\ 1) - u(x, 1) = 0$$

$$U(x,\ y) = y \sin x$$

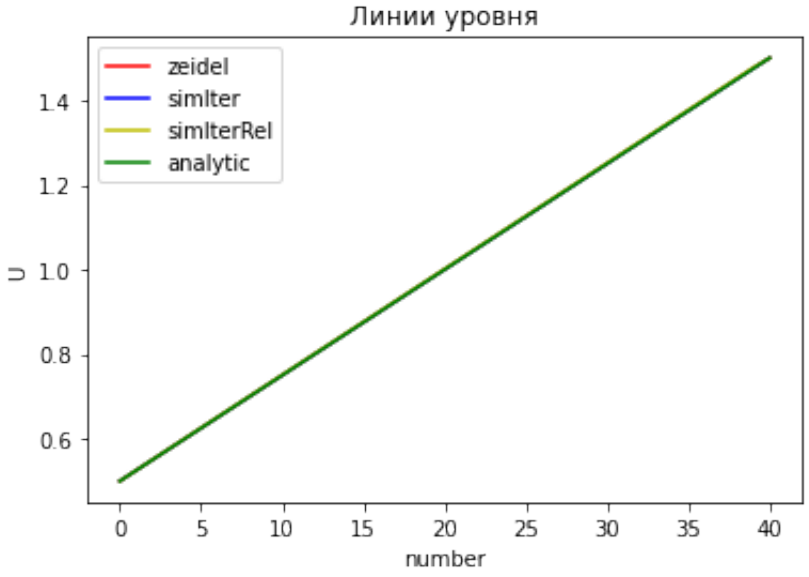
In [1]:
```
ort copy
ort numpy as np
ort matplotlib.pyplot as plt

s = {
 '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
```

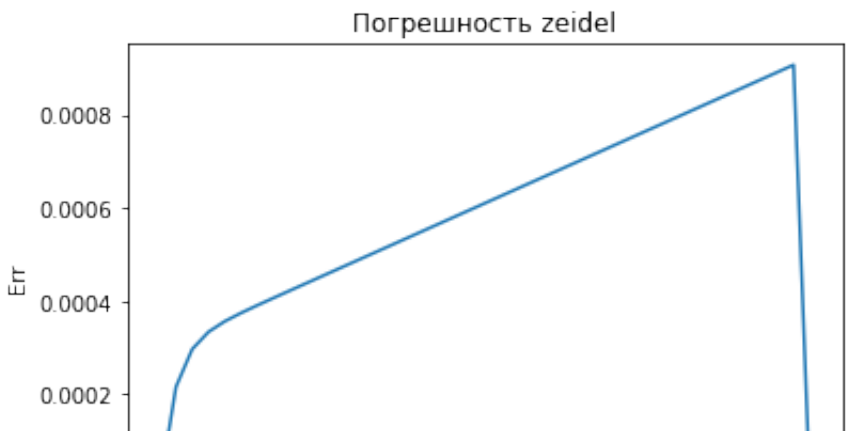
```
 'gamma2': 1,
 'delta1': 0,
 'delta2': 1,
 'phi1': lambda y: y,
 'phi2': lambda y: 1 + y,
 'phi3': lambda x: x,
 'phi4': lambda x: 1 + x,
 'solution': lambda x, y: x + y


 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


ss Eq:
 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']


ss Eleptical:
 def __init__(self, args, nx, ny):
     self.data = Eq(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.init_U(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]) *
     self.iteration = 0
     self.eps = 1e-6
```

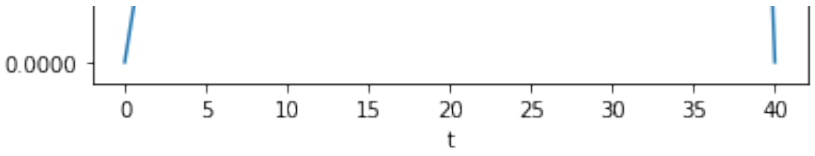
```
def init_U(self, x, y):
    u = np.zeros((len(x), len(y)))
    for i in range(len(x)):
        u[i][0] = self.data.phi3(x[i]) / self.data.gamma2
        u[i][-1] = self.data.phi4(x[i]) / self.data.delta2
    for j in range(len(y)):
        u[0][j] = self.data.phi1(y[j]) / self.data.alpha2
      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

    return u

def analytic_solve(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 simple_it(self, nx, ny):
    cur_eps = 1e9
    while self.iteration < 10000:
        L = copy.deepcopy(self.u)
        u = self.init_U(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
                           (L[i + 1][j] + L[i - 1][j]) - self.data.
                           (L[i][j + 1] + L[i][j - 1]) /
                           (self.hy * self.hy) - self.data.a * self
                           (L[i + 1][j] - L[i - 1][j]) - self.data.
                           (L[i][j + 1] - L[i][j - 1]) /
                           (2 * self.hy)) / (self.data.c * self.hx
                                             (self.hy * self.hy + s
                                             (self.hy * self.hy))
        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 zeidel_method(self, nx, ny):
    cur_eps = 1e9
    while self.iteration < 10000:
        L = copy.deepcopy(self.u)
        u = self.init_U(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],
                           (L[i + 1][j] + u[i - 1][j]) - self.data.
                           (L[i][j + 1] + u[i][j - 1]) / (self.hy *
                           (L[i + 1][j] - u[i - 1][j]) - self.data.
                           (L[i][j + 1] - u[i][j - 1]) /
                           (2 * self.hy)) / \
```

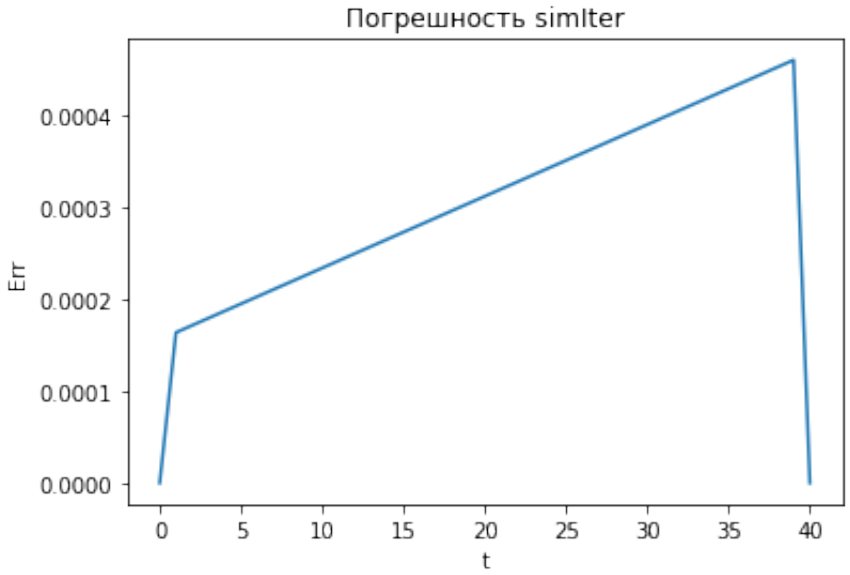
```
                                (self.data.c * (self.hx ** 2) - 2 * (self
                                 (self.hy ** 2))
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if cur_eps <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration

    def simple_it_relaxed(self, nx, ny):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(self.u)
            u = self.init_U(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],
                                (L[i + 1][j] + u[i - 1][j]) - self.data
                                (L[i][j + 1] + u[i][j - 1]) / (self.hy
                                (L[i + 1][j] - u[i - 1][j]) - self.data
                                (L[i][j + 1] - u[i][j - 1]) /
                                (2 * self.hy)) / (self.data.c * (self.h
                                                  (self.hy ** 2 + self.
                                                  (self.hy ** 2))) * se
            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


cmp_error(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

show(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()
```

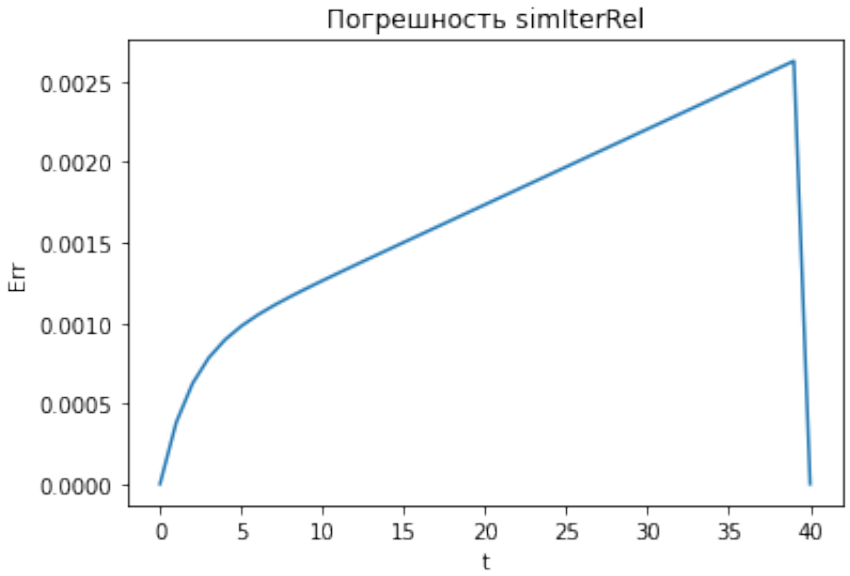
```
 plt.title('Погрешность simIter')
 plt.plot(abs(dict_['simIter'][time] - dict_['analytic'][time]))
 plt.ylabel('Err')
 plt.xlabel('t')
 plt.show()

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

a = {'nx': 40, 'ny': 40}
 ny = int(data['nx']), int(data['ny'])

lSolver = Eleptical(args, nx, ny)
IterSolver = Eleptical(args, nx, ny)
delSolver = Eleptical(args, nx, ny)
IterRelSolver = Eleptical(args, nx, ny)

 = {
 'analytic': analSolver.analytic_solve(nx, ny),
 'simIter': simIterSolver.simple_it(nx, ny)[0],
 'zeidel': zeidelSolver.zeidel_method(nx, ny)[0],
 'simIterRel': simIterRelSolver.simple_it_relaxed(nx, ny)[0]


w(ans, 20)
```







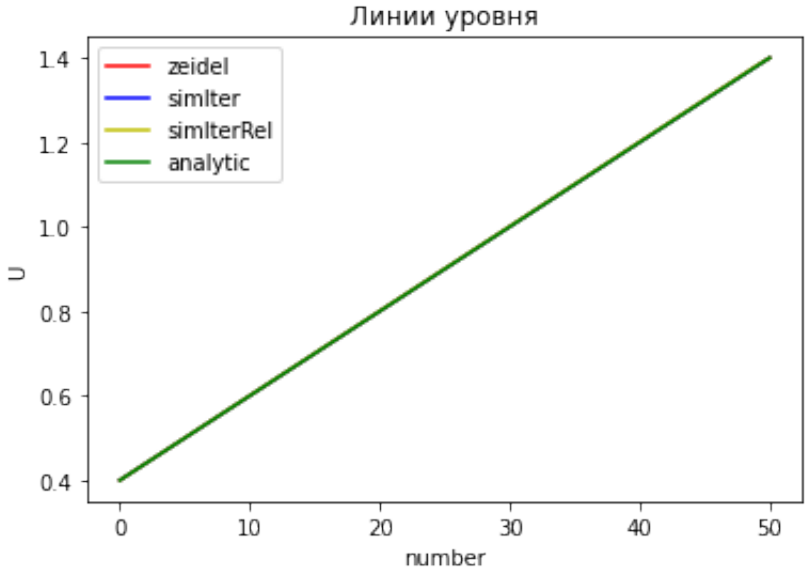


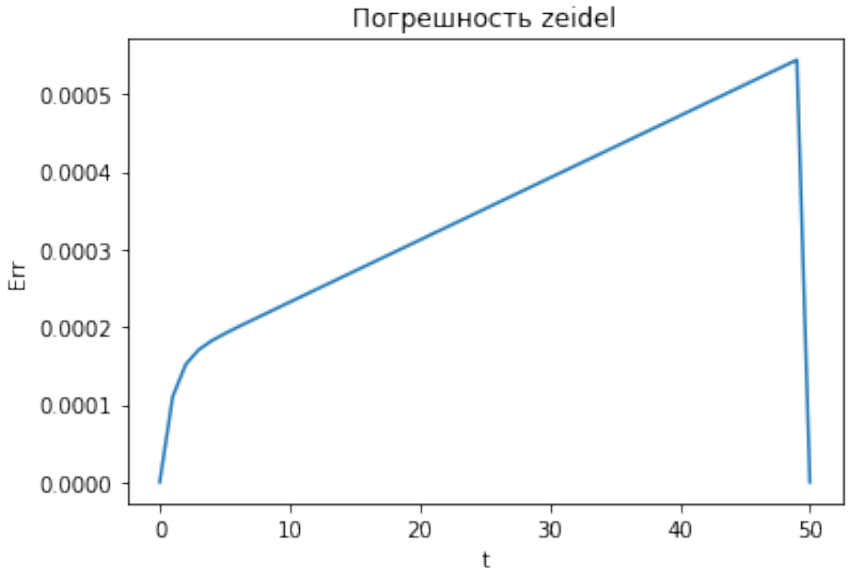


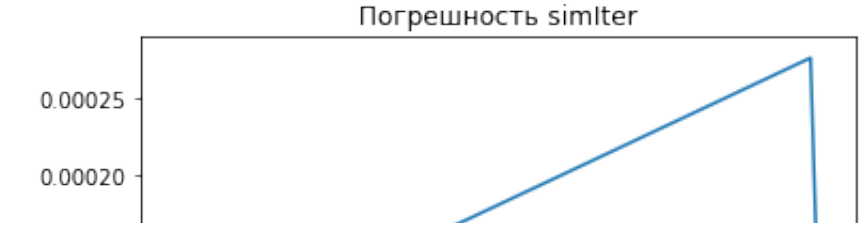
Исследование зависимости погрешности от параметров hx, hy

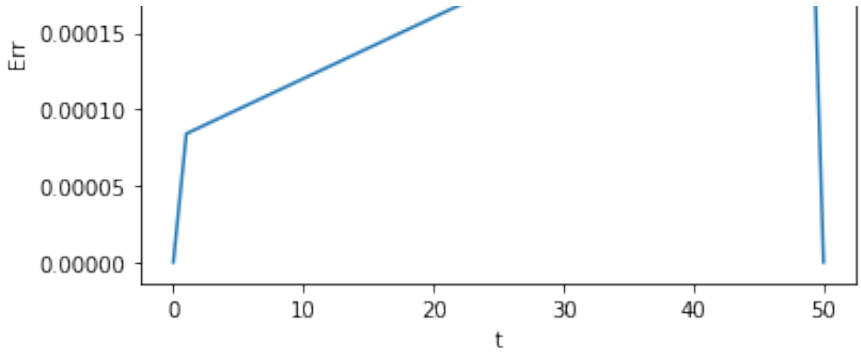
In [4]:

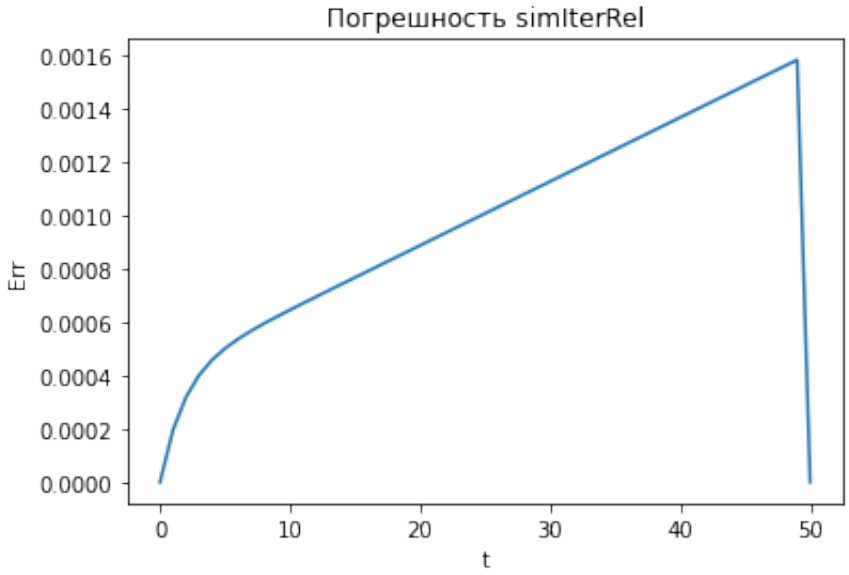
```
data = {'nx': 50, 'ny': 50}
nx, ny = int(data['nx']), int(data['ny'])

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

ans = {
    'analytic': analSolver.analytic_solve(nx, ny),
    'simIter': simIterSolver.simple_it(nx, ny)[0],
    'zeidel': zeidelSolver.zeidel_method(nx, ny)[0],
    'simIterRel': simIterRelSolver.simple_it_relaxed(nx, ny)[0]
}

show(ans, 20)
```











Выводы:

Чем меньше величина шага h_x и h_y, тем меньше погрешность у всех методов

In [ ]:

In [ ]: