Задание на 20.02

Кириленко Андрей, ВШЭ

19 февраля 2019

$\boxed{1}$

Для начала, решим краевую задачу аналитически.

$u \; := u(x)$

$-u'' + u = x, \; x \in [0; \; 1], \; u(0) = u(1) = 0$

Для решения задачи рассмотрим сначала однородное уравнение $x^2 - 1 = 0$, у которого корни $x = 1, x = -1$, откуда общее решение однородного $c_1 e^x + c_2 e^{-x}$.

Частное решение общего видно на глаз, например, $u(x) = x$, тогда общее решение неоднородного: $c_1 e^x + c_2 e^{-x} + x$.

Найдем c_1, c_2. Подставляя 0 и 1 имеем: $c_1 = -c_2, \; c_1 e + c_2/e = -1$, откуда $c_1 = \frac{1}{\frac{1}{e} - e}, c_2 = \frac{1}{e - \frac{1}{e}}$.

Получили $u(x) = \frac{1}{\frac{1}{e} - e}(e^x - e^{-x}) + x$.

Далее нужно искать A, b. Для начала напишем реализацию их поиска.

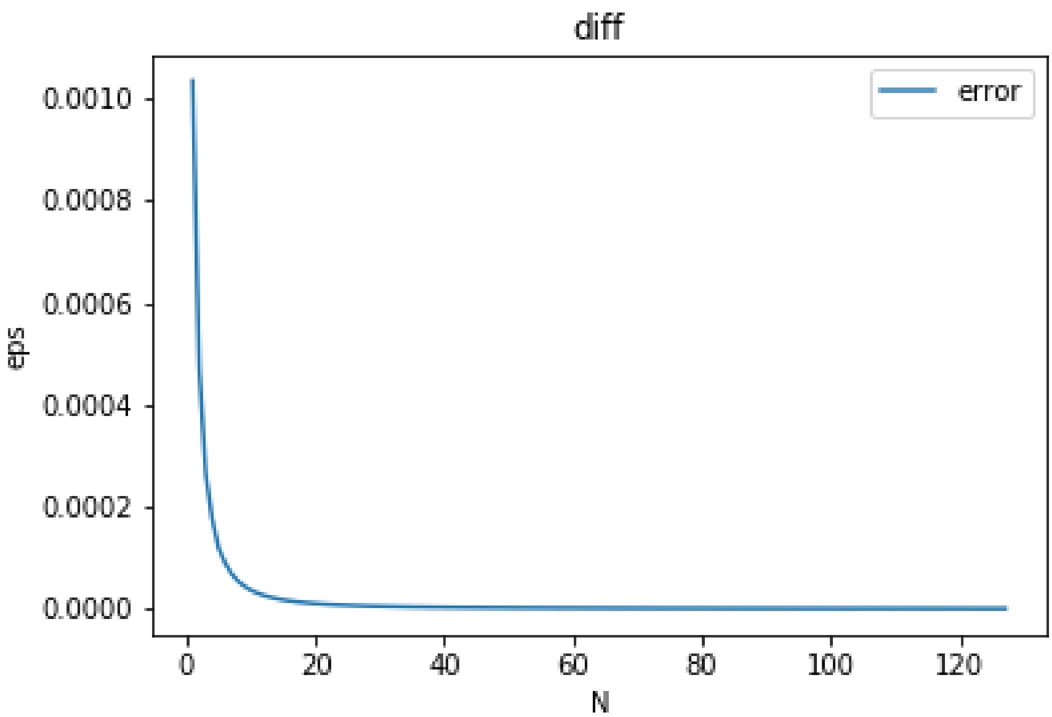
```
def u(x):
    return (np.e / (1 - np.e ** 2)) * (np.exp(x) - np.exp(-x)) + x

def A(n):
    h = 1.0 / (n + 1)
    A = []
    for i in range(n):
        row = []

        for j in range(i - 1):
            row.append(0)

        if i > 0:
            row.append(-h ** (-2))

        row.append(2 * h ** (-2) + 1)

        if i < n - 1:
            row.append(-h ** (-2))

        for j in range(n - i - 2):
            row.append(0)

        A.append(row)

    return np.array(A)

def b(n):
    h = 1.0 / (n + 1)

    res = []
    for i in range(n):
```

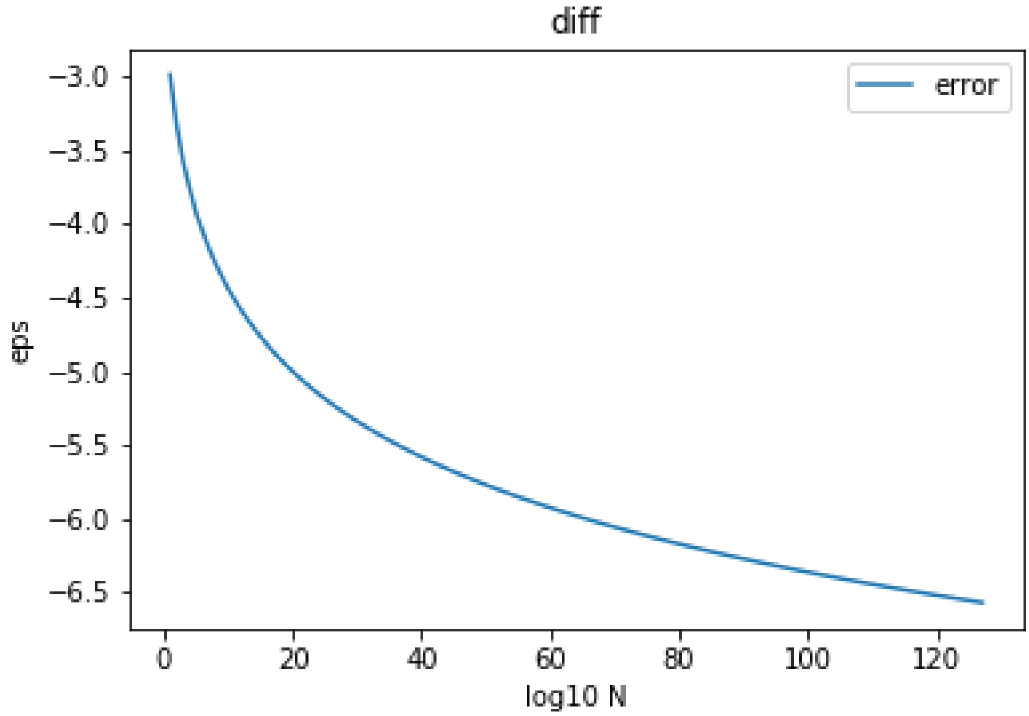
```
        res.append((i + 1) * h)

    return np.array(res)
```

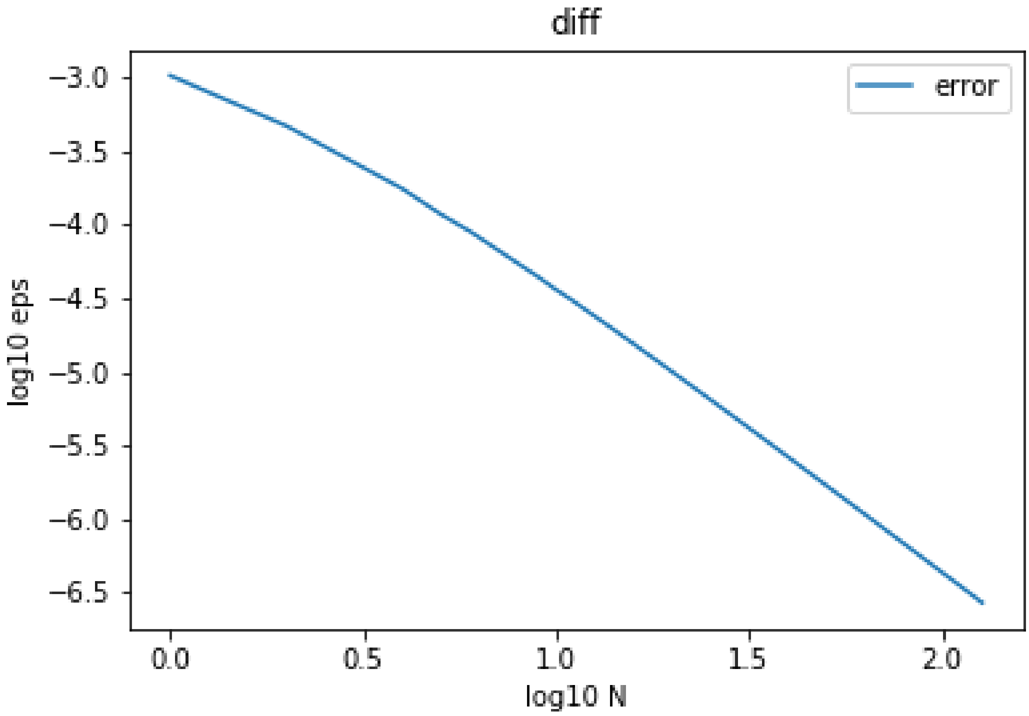
Листинг 1: Поиск A b u

```
def task1a():
    y = []
    x = []
    for n in range(1, 128):
        err = np.amax(np.abs(solve(n) - gen_u(n)))
        x.append(n)
        y.append(err)
    graph.plot(x, y, label='error')
    graph.xlabel('N')
    graph.ylabel('eps')
    graph.title('diff')
    graph.legend()
    graph.savefig("t1.png")
    graph.close()
```

Листинг 2: Строим график

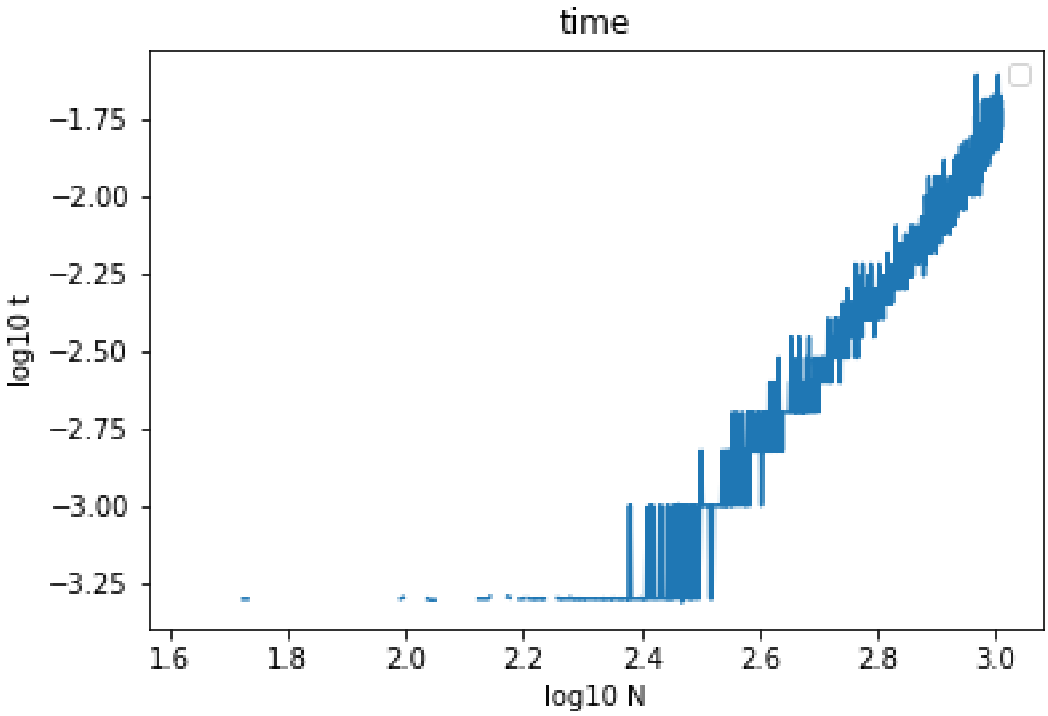






Исходя из третьего графика, зависимость получается степенной, а так как $\log a \sim -\log b \Rightarrow a \sim 1/b$, то на первом графике гипербола, убывание ошибки степенное.

Посмотрим на время работы:



Отсюда следует что время работы степенное, причем степень от 2 до 3 скорее всего. У меня интересные скачки, но возможно это связано с особенностями реализации и железа.

Напишем теперь метод прогонки, а затем сравним время работы.

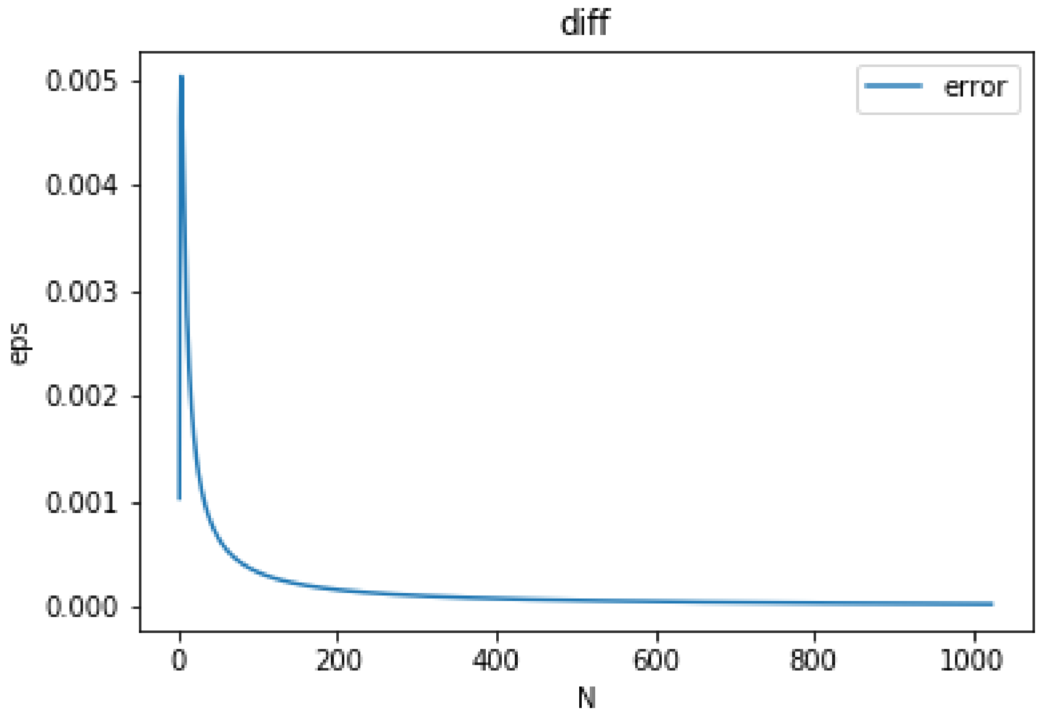
Достаточное условие выполнено, так как элемент на главной диагонали на 1 больше суммы элементов на соседних двух диагоналях.

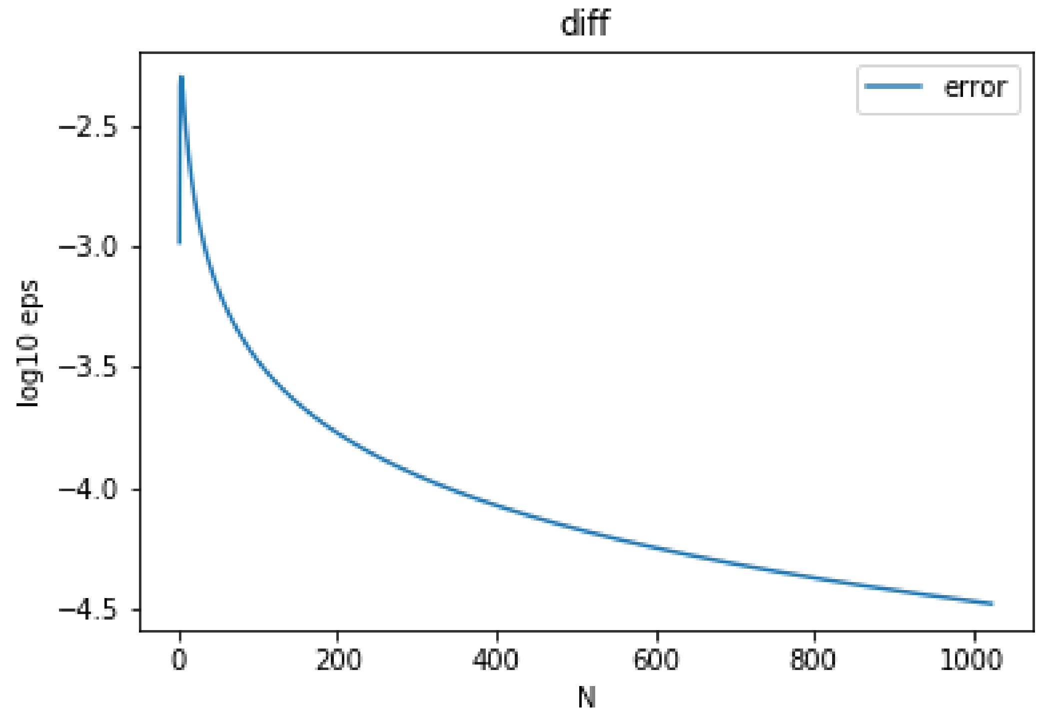
```
def tridiagonal(n):
    h = 1.0 / (n + 1)
    a = []
    b = []
    c = []
    res = []
    d = b(n)

    for _ in range(n):
        a.append(h ** (-2))
        b.append(h ** (-2))
        c.append(2 * h**(-2) + 1)
        res.append(0)

    alphas = [b[0] / c[0]]
    betas = [d[0] / c[0]]

    for i in range(1, n):
        alphas.append(b[i] / (c[i] - alphas[-1] * a[i]))
        betas.append((d[i] + betas[-1] * a[i]) / (c[i] - alphas[-1] * a[i]))
```

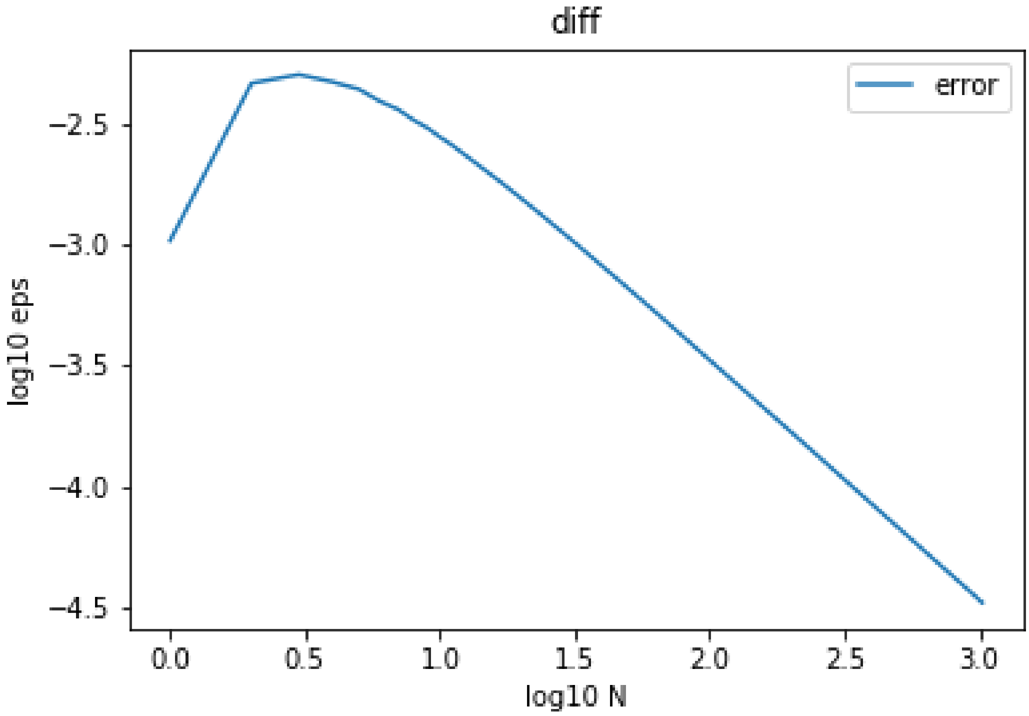
```
    res[n - 1] = betas[-1]
    for i in range(n - 2, -1, -1):
        res[i] = res[i + 1] * alphas[i] + betas[i]
    return np.array(res)
```

Листинг 3: Прогонка

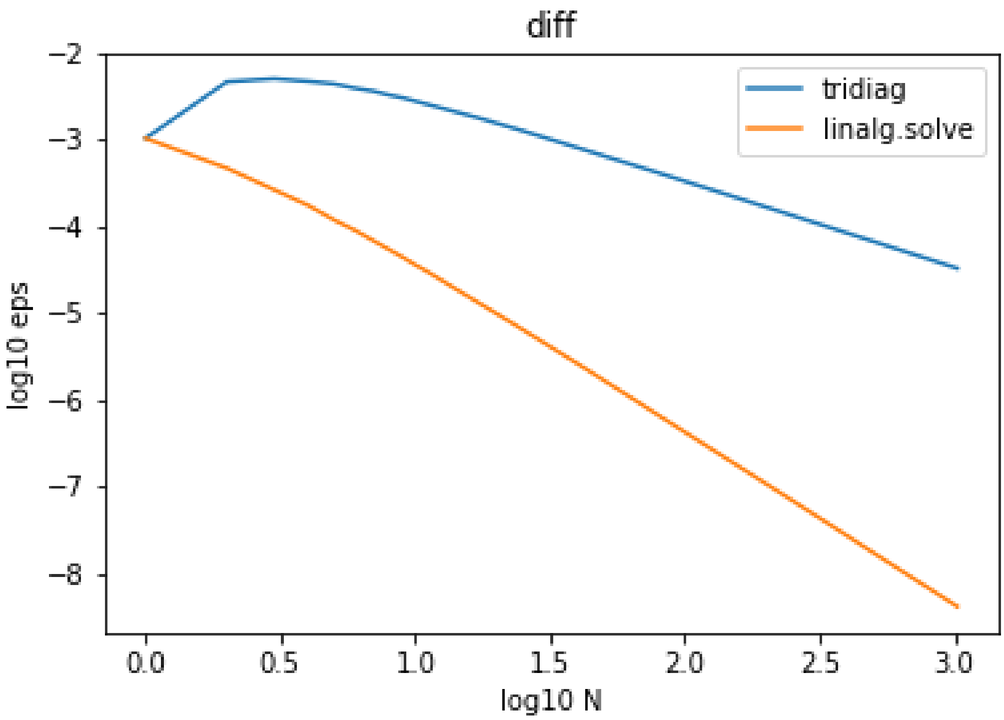
Получится на первый взгляд похожая история:



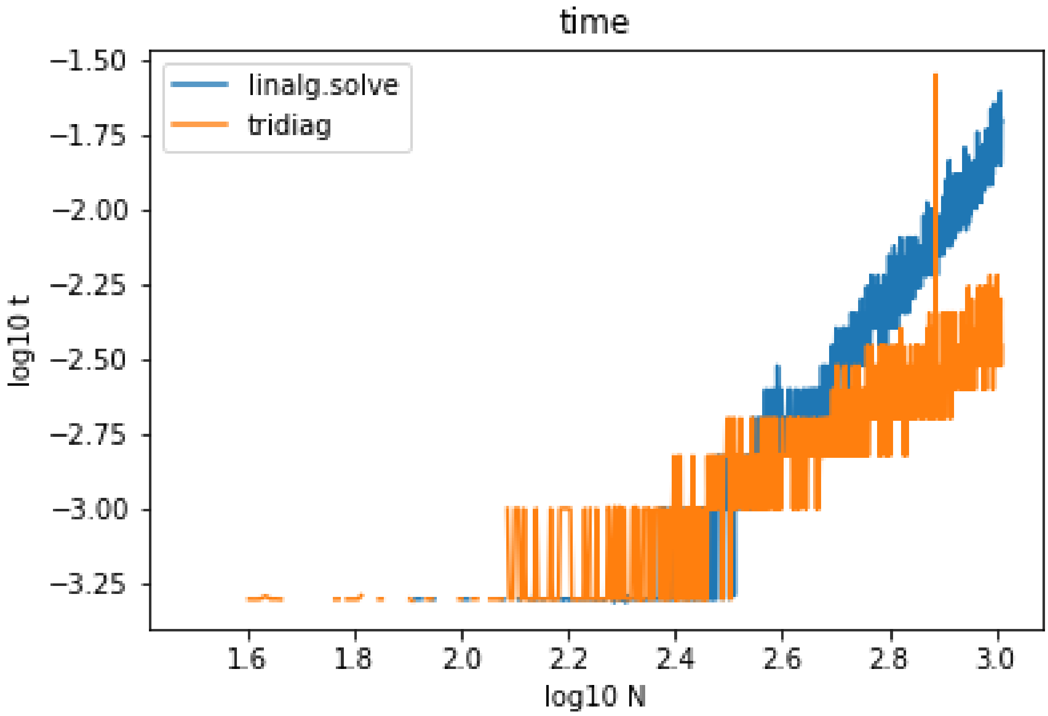




Сравним на одном графике:



Здесь уже становится понятно, что точность степенная, но при этом значительно хуже. Стало быть, должно быстрее работать. Посмотрим на график времени работы:



И действительно, получается гораздо быстрее, метод прогонки работает за линейное время и память, получается что мы обмениваем скорость на точность.

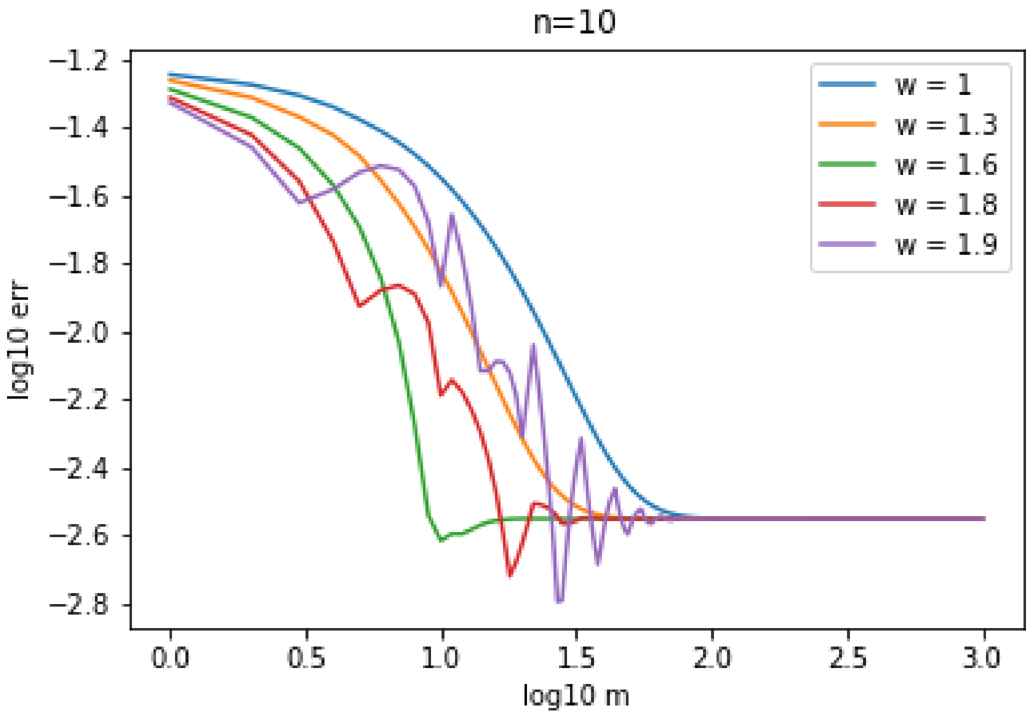
$\boxed{2}$

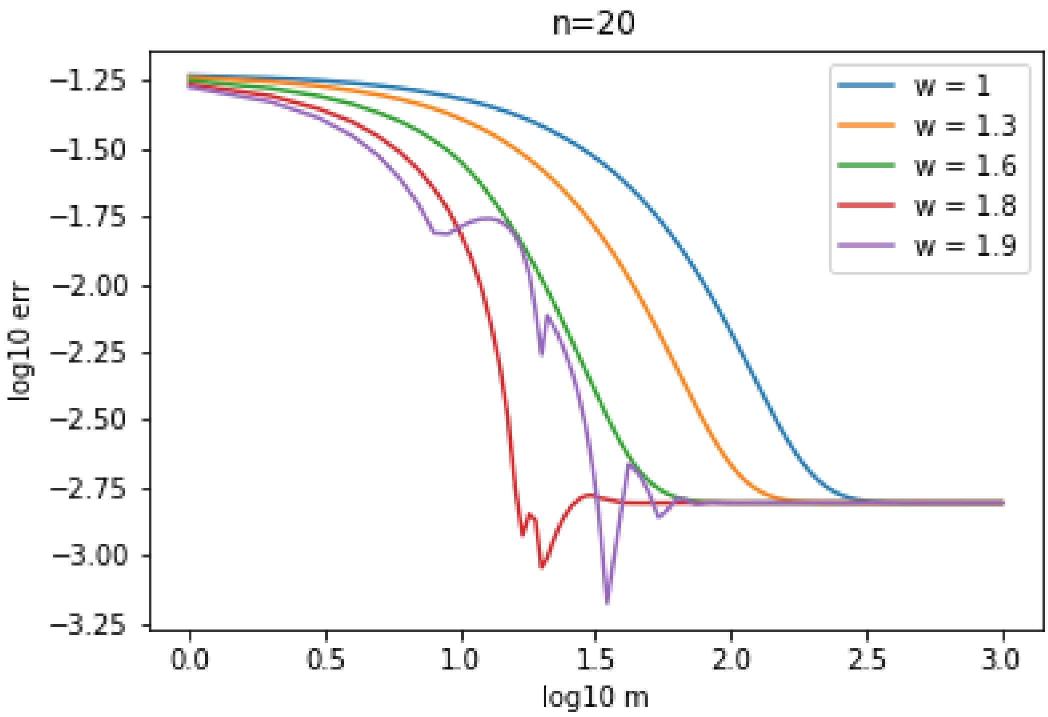
Реализуем вычисление SOR по формуле с лекции:

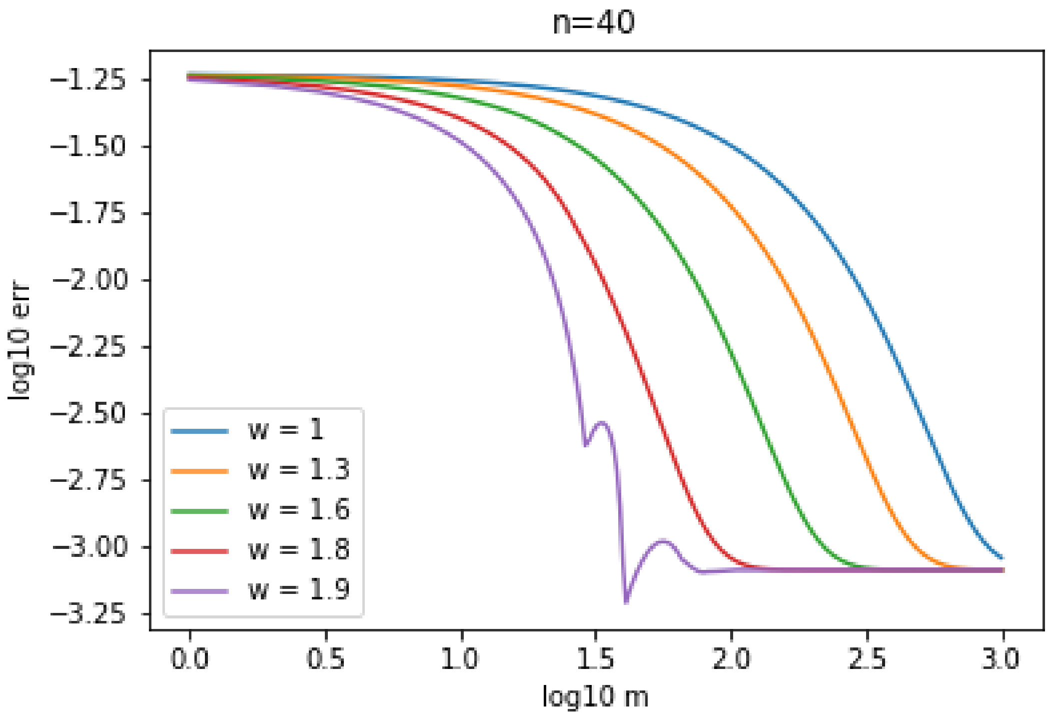
```
def sor(An, bn, n, m, w):
    res = []
    cur = [0] * n
    for _ in range(m):
        nxt = []
        for i in range(n):
            sum1 = sum([An[i][j] * nxt[j] for j in range(i)])
            sum2 = sum([An[i][j] * cur[j] for j in range(i + 1, n)])
            nxt.append((1 - w) * cur[i] + w * (bn[i] - sum1 - sum2) / An[i][i])
        res.append(nxt)

        cur = nxt.copy()
    return res
```

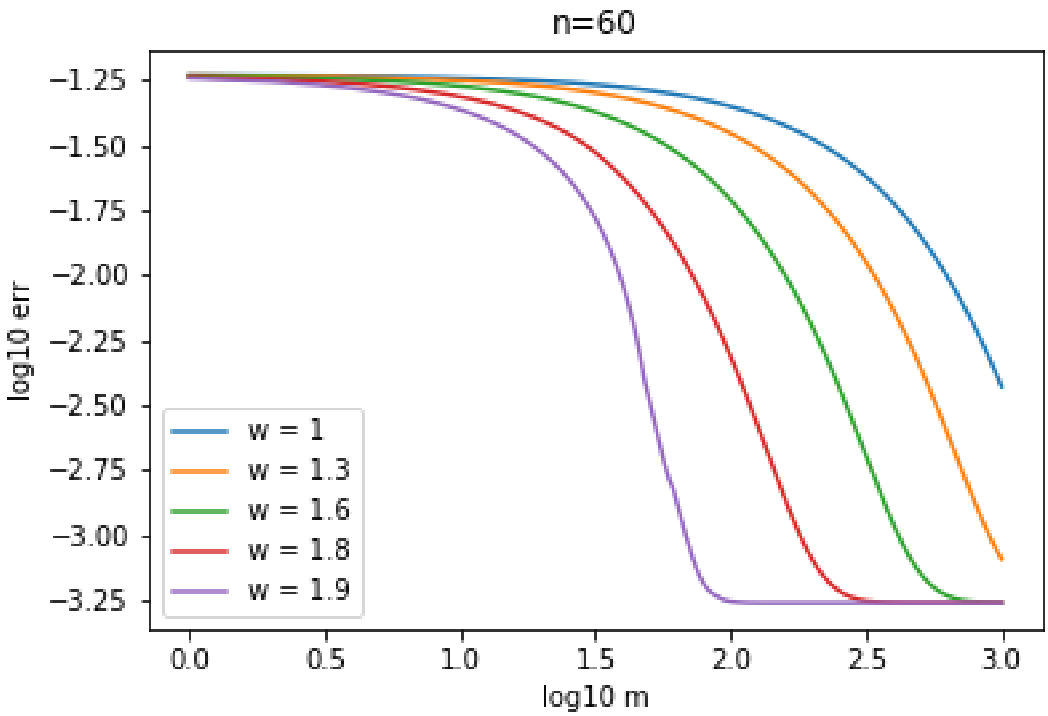
Листинг 4: SOR

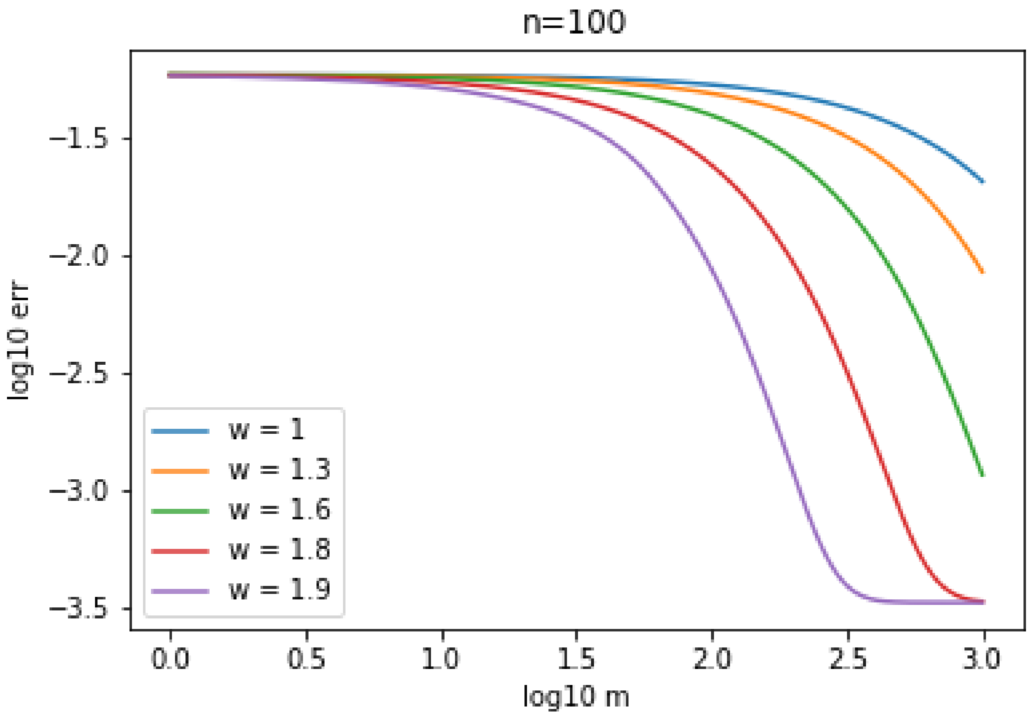
Построим графики зависимости погрешности от количества итераций в логарифмическом масштабе:

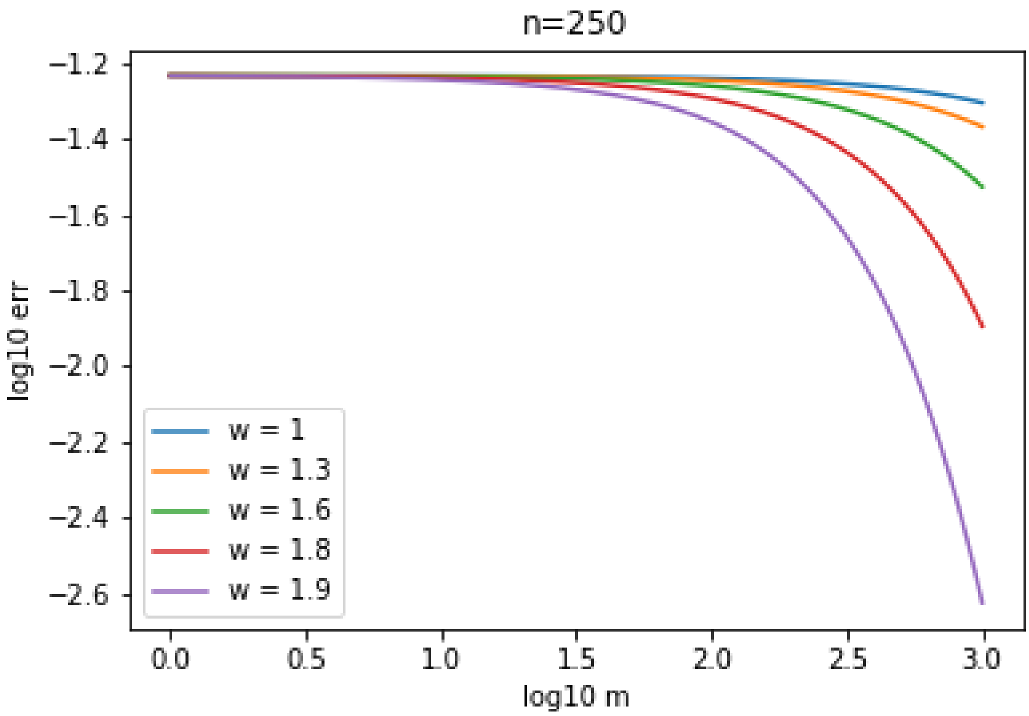

n=10
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9
log10 err
log10 m
−1.2
−1.4
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
−2.8
0.0
0.5
1.0
1.5
2.0
2.5
3.0

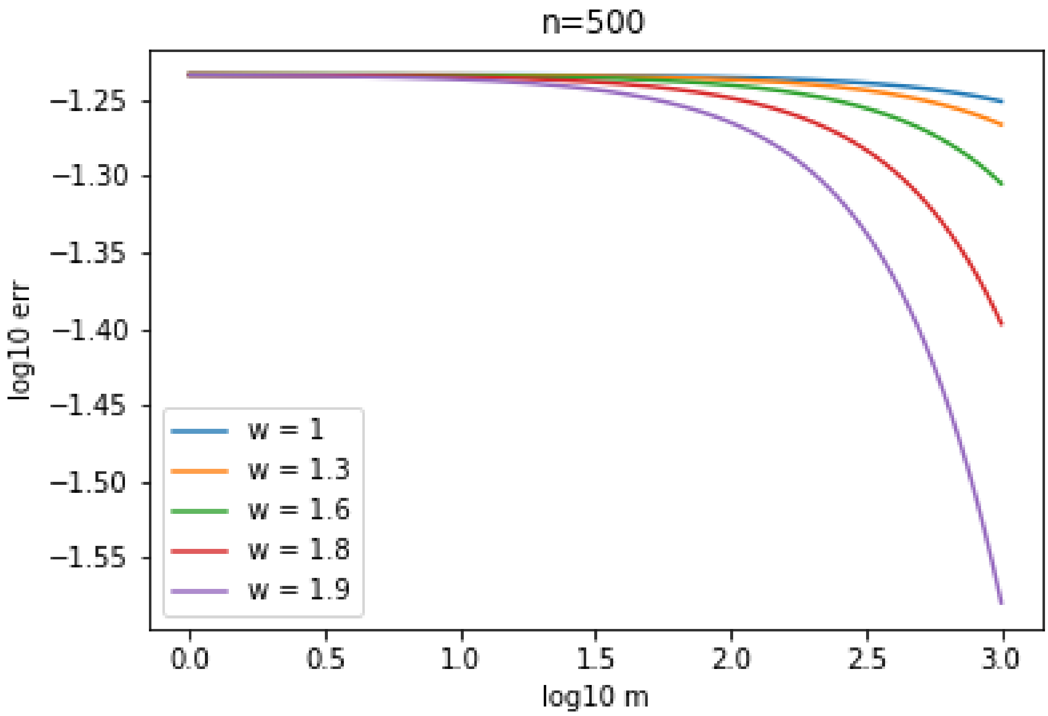

n=20
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9
log10 err
log10 m
−1.25
−1.50
−1.75
−2.00
−2.25
−2.50
−2.75
−3.00
−3.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0


n=40
log10 err
log10 m
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9
−1.25
−1.50
−1.75
−2.00
−2.25
−2.50
−2.75
−3.00
−3.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0


n=60
log10 err
log10 m
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9
−1.25
−1.50
−1.75
−2.00
−2.25
−2.50
−2.75
−3.00
−3.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0


n=100
−1.5
−2.0
−2.5
−3.0
−3.5
log10 err
0.0
0.5
1.0
1.5
2.0
2.5
3.0
log10 m
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9


n=250
−1.2
−1.4
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
log10 err
0.0
0.5
1.0
1.5
2.0
2.5
3.0
log10 m
w = 1
w = 1.3
w = 1.6
w = 1.8
w = 1.9



Получается экспоненциальная зависимость, а оптимальное значение омеги зависит от N, однако при больших N лучше подходят значения, близкие к 2. Кроме того, с ростом размера матрицы начинает расти ошибка. При этом с ростом числа итераций она убывает экспоненциально, как было сказано выше.