Цель: решить выбранную СЛАУ следующими методами (постараться избежать умножения матриц, использовать поэлементные записи):

1. метод Гаусса с выбором главного элемента
2. метод LU-разложение (если применим)
3. метод Якоби
4. метод Зейделя
5. метод верхней релаксации
6. метод градиентного спуска
7. метод минимальных невязок
8. стабилизированный метод бисопряженных градиентов

*Для итерационных методов получить график убывания невязки в зависимости от итерации.

1) Метод Гаусса с выбором главного элемента.

- Прямой ход - приводим заданную матрицу A к верхнетреугольному виду:
  - находим главный (максимальный по модулю) элемент матрицы
  - меняем местами строки матрицы и элементы столбца решений так, чтобы главный элемент оказался в верхнем левом углу
  - меняем местами столбца матрицы и элементы столбца переменных так, чтобы главный элемент оказался в верхнем левом углу
  - делим нулевую строку матрицы и нулевой элемент столбца решений на главный элемент
  - вычитаем из каждого элемента нулевой в текущем столбце решений, умноженный на нулевой элемент нужной строки - получаем в начале каждой строки единицу
  - вычитаем из каждой строки матрицы нулевую, умноженную на нулевой элемент нужной строки - получаем в начале каждой строки единицу
- повторяем весь алгоритм выше для всех матриц, получаемых из исходной сдвигом левого верхнего угла на один вправо и вниз
- последний элемент столбца решений делим на последний элемент матрицы, который делаем = 1
- Обратный ход:
  - вычисляем элементы столбец переменных
  - меняем в этом столбце порядок переменных в соответствии с найденным
- возвращаем столбец решений

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import time
# формат вывода
np.set_printoptions(precision=5, suppress=True, formatter={'all':
lambda x: f'{x:0.5f}'})
```

```
#почти все формулы - из методички Томск-СЛАУ :)
```

```
'''Вариант Ж'''
a = 20 #константа - зависит от варианта задания a = 20
size = 100 #размер матриц
A = ((a-1)*np.eye(size) + np.ones(size)) #матрица A
f = np.zeros((size, 1))
for i in range(size): f[i] = i + 1 # столбец решений f

def norm3_vect(vect):
    return pow(sum(el**2 for el in vect), 0.5)

def swap_rows(A, row1, row2): #функция для смены строк в матрице
    A[[row1, row2]] = A[[row2, row1]]

def swap_columns(A, col1, col2): #функция для смены столбцов в матрице
    A[:, [col1, col2]] = A[:, [col2, col1]]

def find_max_el(A, iter): #функция для поиска главного элемента
матрицы A[iter,iter]
    size = len(A) - iter
    main_element = A[iter,iter]
    i_main, j_main = iter, iter

    for i in range(iter, size):
        for j in range(iter, size):
            if abs(A[iter:, iter:][i][j]) > abs(main_element):
                i_main, j_main = i, j
                main_element = A[iter:, iter:][i][j]
    return main_element, i_main, j_main

def gauss(A, f):
    print("Решение СЛАУ методом Гаусса с выбором макс элемента")
    size = len(A)
    if A.shape[0] != A.shape[1]:
        print("Матрица не квадратная, решение невозможно!")
        return
    x = np.arange(size) #массив с порядком корней (порядок будет
меняться при перестановке столбцов)

    #прямой ход алгоритма - приводим матрицу A к верхнетреугольному
виду
    for iter in range(len(A)):
        ''' Добавим в алгоритм проверку - если на какой-то итерации
алгоритма возникла нулевая строка - ответ будет выражаться
        через одну из переменных (её берем за константу, все остальные
переменные будут выражены через нее)'''
```

```
        if (len(A) != sum(int(np.any(el)) for el in A)):
            '''Количество переменных превышает количество уравнений,
решение не однозначн'''
            print("ERROR! Матрица не квадратная, решение невозможно!")
            return

        main_el, i_main, j_main = find_max_el(A, iter) #максимальный
элемент текущей матрицы и его местонахождение

        if (i_main != iter): #если элемент еще не в нулевой строке
            swap_rows(A, iter, i_main) #меняем в текущей матрице
нулевую строку и строку, содержащую главный элемент
            swap_rows(f, iter, i_main) #то же самое - в столбце
решений

        if (j_main != iter): #если элемент еще не в нулевом столбце
            swap_columns(A, iter, j_main) #меняем в текущей матрице
нулевой столбец и столбец, содержащий главный элемент
            swap_columns(x, iter, j_main) #то же самое - в строке
порядка переменных

        if main_el != 0:
            A[iter:, iter:][0] /= main_el #делим нулевую строку
текущей матрицы на главный элемент
            f[iter] /= main_el #делим нулевую строку текущего столбца
решений на главный элемент
        else:
            print("ERROR! main_el = 0")
            return

        for i in range(size - iter - 1):
            f[i+iter+1] -= (f[iter] * A[iter:, iter:][i+1][0])
            #вычитаем из каждого элемента нулевой в текущем столбце
решений, умноженный на нулевой элемент нужной строки - получаем в
начале каждой строки единицу
            A[iter:, iter:][i+1] -=  (A[iter:, iter:][0] * A[iter:,
iter:][i+1][0])
            #вычитаем из каждой строки нулевую, умноженную на нулевой
элемент нужной строки - получаем в начале каждой строки единицу

    f[-1] /= A[-1][-1]
    A[size-1][size-1] = 1



    #обратный ход алгоритма
    U  = np.zeros((size, 1)) #столбец решений
    for i in range(size-1, -1, -1):
        U[i] = f[i]
```

```
        for j in range(i + 1, size):
            U[i] -= U[j] * A[i][j]

    #перестановка переменных в изначальном порядке
    ans = np.zeros((size, 1))
    for i in range(size):
        ans[int(x[i])] = U[i]

    return ans

start_time = time.time()

A_gauss = np.copy(A) #сохраним исходную матрицу A в отдельной
переменной
f_gauss = np.copy(f) #и столбец решений тоже сохраним в отдельной
переменной
#это - т к иначе python расценит разные numpy матрицы как ссылающиеся
на один объект

u = gauss(A_gauss, f_gauss) #столбец решений

# Проверка - перемножаем и вычисляем норму разницы
error = norm3_vect(A@u - f)

np.set_printoptions(formatter={'float': '{: 0.5e}'.format})
print("Погрешность метода = ", error, "\n", "Проверка: ", sep = "",end
= "")
#проверяем на приблизительное равенство из-за машинных ошибок
if (error > 1e-5): print("Error")
else: print("OK")

time_gauss = 1000*(time.time() - start_time)
print("Время выполнения = {0:.4f} мс".format(time_gauss))
```

```
Решение СЛАУ методом Гаусса с выбором макс элемента
Погрешность метода = [ 2.45908e-13]
Проверка: OK
Время выполнения = 327.6000 мс
```

2) LU - разложение.

- Прямой ход - разделяем исходную матрицу A на верхнетреугольную U и нижнетреугольную L - в произведении они дают исходную
- Обратный ход:
  - из уравнения Ly = f получаем столбец y
  - из уравнения Ux = y получаем стообец x

```
def LU_decomp(A):

    size = len(A)
```

```
    L = np.zeros((size,size))
    U = np.zeros((size,size))

    for i in range(size):
        L[i][i] = 1
        for j in range(i, size):
            Uij = (A[i][j] - sum(L[i][k] * U[k][j] for k in range(i)))
            if Uij == 0:
                print("Error! devision by 0")
                return
            U[i][j] = Uij
            #вычисляем элементы матриц L и U построчно, друг через
друга
            L[j][i] = (A[j][i] - sum(L[j][k] * U[k][i] for k in
range(i))) / U[i][i]



    return L, U

def LU(A, f):
    print("Решение СЛАУ с помощью LU-разложения")
    size = len(A)

    #LU-разложение матрицы A
    LU_dec = LU_decomp(A)
    L = LU_dec[0]
    U = LU_dec[1]


    #обратный ход - решение СЛАУ
    #Ly = f
    y = np.ones((size, 1))
    for i in range(size):
        y[i] = f[i] - sum(L[i][j] * y[j] for j in range(i))

    #Ux = y
    x = np.ones((size, 1))
    for i in range(size-1, -1, -1):
        x[i] = (y[i] - sum(U[i][j] * x[j] for j in range(i+1, size)))
/ U[i][i]

    return x

start_time = time.time()

A_lu = np.copy(A)
f_lu = np.copy(f)

ans = LU(A_lu, f_lu)
```

```
# Проверка
error = norm3_vect(A@ans - f)
np.set_printoptions(formatter={'float': '{: 0.5e}'.format})
print("Погрешность метода = ", error, "\n", "Проверка: ", sep = "",end
= "")
if  (error > 1e-5): print("Error!")
else: print("OK")

time_lu = 1000*(time.time() - start_time)
print("Время выполнения = {0:.4f} мс".format(time_lu))
```

```
Решение СЛАУ с помощью LU-разложения
Погрешность метода = [ 1.96173e-13]
Проверка: OK
Время выполнения = 339.7748 мс
```

3) Метод Якоби.

- Задаем точность ε
- Пока невязка (норма) разницы произведения исходной матрицы и полученного столбца переменных x и столбца решений f больше нормы, проводим итерационный цикл:
  - вычисляем элементы нового столбца переменных через элементы старого и элементы исходной матрицы

  В зависимости от заданной точности пропорционально будет меняться количество итераций, которое понадобилось для приближения с такой точностью. Построим график зависимости количества итераций от заданной невязки.

```
def Jacobi(A, f, x0, eps):

    print("Решение СЛАУ методом Якоби")
    size = len(A)

    x_old = np.copy(x0)
    x_new = np.copy(x0)
    iter = 0

    norm = norm3_vect(A @ x_new - f)
    norms = []
    while (norm > eps):
        for i in range(size):
            x_new[i] = 0
            for j in range(size):
                if (j != i):
                    x_new[i] = x_new[i] + A[i][j] * x_old[j]
            x_new[i] = (f[i] - x_old[i]) / A[i][i]
        x_old = x_new
        iter += 1
```

```
        norm = norm3_vect(A @ x_new - f)
        norms.append(norm)


    return x_new, iter, norms


start_time = time.time()
A_j = np.copy(A)
f_j = np.copy(f)

eps = 1.00000e-5

u_0 = np.ones((size,1))
J = Jacobi(A_j, f_j, u_0, eps)
print("Количество итераций = ", J[1])

# Проверка
x = J[0]
error = norm3_vect(A@x - f)
np.set_printoptions(formatter={'float': '{: 0.5e}'.format})
print("Погрешность метода = ", error, " < eps = ", eps, "\n",
"Проверка: ", sep = "",end = "")
if  (error > eps): print("Error!")
else: print("OK")

time_jacobi = 1000*(time.time() - start_time)
print("Время выполнения = {0:.4f} мс".format(time_jacobi))

discrepancy_J = J[2]
```

```
Решение СЛАУ методом Якоби
Количество итераций =  54
Погрешность метода = [ 7.44009e-06] < eps = 1e-05
Проверка: OK
Время выполнения = 2299.8974 мс
```

4) Метод Гаусса-Зейделя.

- Задаем точность ε
- Пока невязка (норма) разницы произведения исходной матрицы и полученного столбца переменных x и столбца решений f больше нормы, проводим итерационный цикл:
  - вычисляем элементы нового столбца переменных через элементы старого и элементы исходной матрицы

  В зависимости от заданной точности пропорционально будет меняться количество итераций, которое понадобилось для приближения с такой точностью. Построим график зависимости количества итераций от заданной невязки.

```python
def Seidel(A, f, x0, eps):

    print("Решение СЛАУ методом Зейделя")
    size = len(A)

    x_old = np.copy(x0)
    x_new = np.copy(x0)
    iter = 0

    norm = norm3_vect(A @ x_new - f)
    norms = []
    while (norm > eps):
        for i in range(size):
            sig = 0
            for j in range(i):
                sig += A[i][j] * x_new[j]
            for j in range(i+1, size):
                sig += A[i][j] * x_old[j]

            x_new[i] = (f[i] - sig) / A[i][i]

        x_old = x_new
        iter += 1
        norm = norm3_vect(A @ x_new - f)
        norms.append(norm)

    return x_new, iter, norms


start_time = time.time()
A_s = np.copy(A)
f_s = np.copy(f)
eps = 1.00000e-5

u_0 = np.ones((size,1))
S = Seidel(A_s, f_s, u_0, eps)
print("Количество итераций = ", S[1])

# Проверка
x = S[0]
error = norm3_vect(A@x - f)
np.set_printoptions(formatter={'float': '{: 0.5e}'.format})
print("Погрешность метода = ", error, " < eps = ", eps, "\n",
"Проверка: ", sep = "",end = "")
if  (error > eps): print("Error!")
else: print("OK")

time_seidel = 1000*(time.time() - start_time)
print("Время выполнения = {0:.4f} мс".format(time_seidel))
```

```
discrepancy_S = S[2]
```

```
Решение СЛАУ методом Зейделя
Количество итераций =  51
Погрешность метода = [ 6.85481e-06] < eps = 1e-05
Проверка: ОК
Время выполнения = 2300.2577 мс
```

5)Метод верхней релаксации - SOR

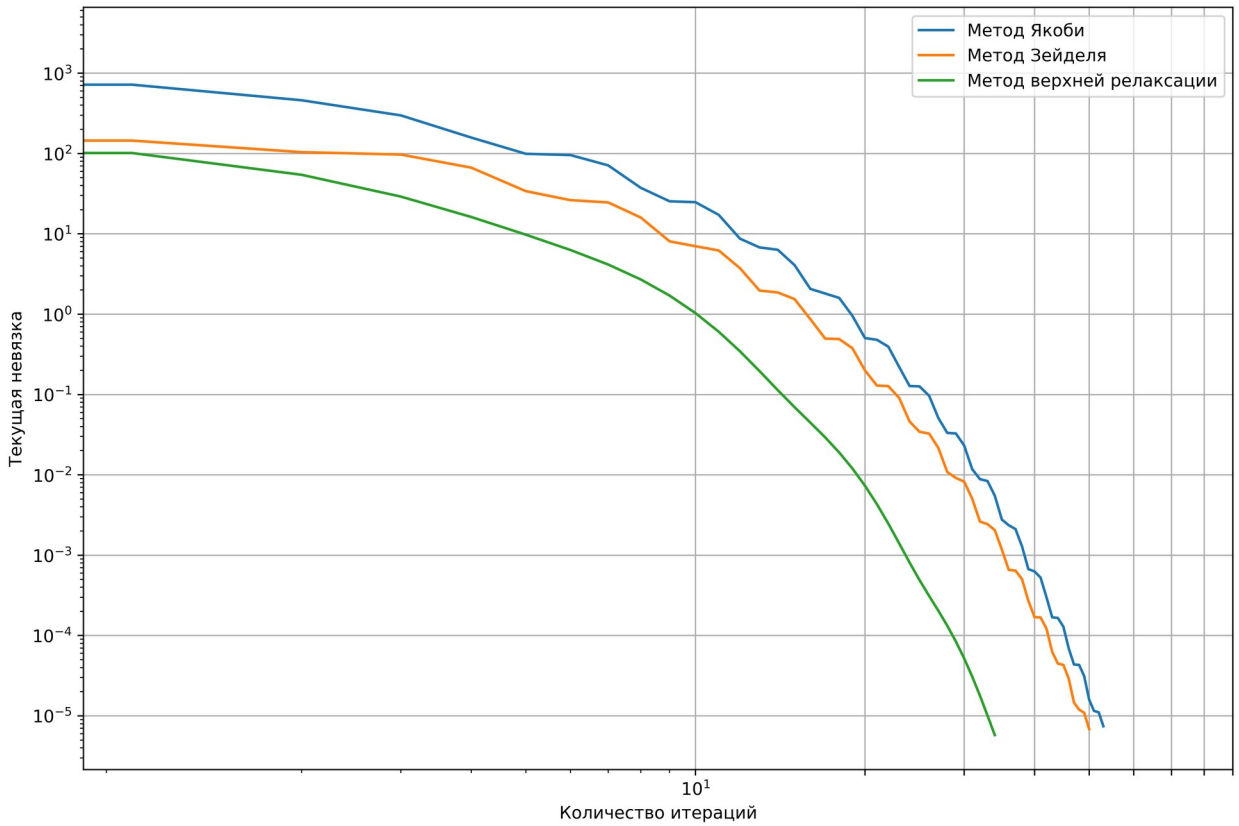
```
def SOR(A, f, x0, eps, w):

    size = len(A)

    x_old = np.copy(x0)
    x_new = np.copy(x0)
    iter = 0

    norm = norm3_vect(A @ x_new - f )
    norms = []
    while (norm > eps and iter < 1000):
        for i in range(size):
            sig = 0
            for j in range(i):
                sig += A[i][j] * x_new[j]
            for j in range(i+1, size):
                sig += A[i][j] * x_old[j]

            sig = (f[i] - sig) / A[i][i]

            x_new[i] = x_old[i] + w * (sig - x_old[i])

        x_old = x_new
        iter += 1
        norm = norm3_vect(A @ x_new - f)
        norms.append(norm)

    return x_new, iter, norms


start_time = time.time()
A_sor = np.copy(A)
f_sor = np.copy(f)
eps = 1.00000e-5

w = 0.5

u_0 = np.ones((size,1))
sor = SOR(A_sor, f_sor, u_0, eps, w)
```

```
print("Количество итераций = ", sor[1])

# Проверка
x = sor[0]
error = norm3_vect(A@x - f)
np.set_printoptions(formatter={'float': '{: 0.5e}'.format})
print("Погрешность метода = ", error, " < eps = ", eps, "\n",
"Проверка: ", sep = "",end = "")
if  (error > eps): print("Error!")
else: print("OK")

time_sor = 1000*(time.time() - start_time)
print("Время выполнения = {0:.4f} мс".format(time_sor))

discrepancy_sor = sor[2]
```

```
Количество итераций =  35
Погрешность метода = [ 5.78064e-06] < eps = 1e-05
Проверка: OK
Время выполнения = 1604.0761 мс
```

Построим для 3х итерационных методов графики зависимости невязки от количества итераций, потраченных на достижение этой невязки

```
plt.figure(figsize = [12,8], dpi = 500)


plt.plot(np.arange(0, len(discrepancy_J)),   discrepancy_J,   label =
"Метод Якоби")
plt.plot(np.arange(0, len(discrepancy_S)),   discrepancy_S,   label =
"Метод Зейделя")
plt.plot(np.arange(0, len(discrepancy_sor)), discrepancy_sor, label =
"Метод верхней релаксации")

plt.xscale('log')
plt.yscale('log')

plt.xlabel('Количество итераций')
plt.ylabel('Текущая невязка')

plt.xticks(np.arange(start=10, stop=100, step = 10))

plt.legend()
plt.grid()
plt.show()
```



Заметим: при параметре оптимальности w = 1 метод верхних релаксаций абсолютно идетничен методу Зейделя, что видно из формулы x_new[i] = x_old[i] + w * (sig - x_old[i])

```
from tabulate import tabulate
table = [['Gauss', 'LU', 'Jacobi', 'Seidel', 'SOR'], [time_gauss,
time_lu, time_jacobi, time_seidel, time_sor]]
print("Сравнение алгоритмов по скорости, мс")
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))
```

```
Сравнение алгоритмов по скорости, мс
```

Gauss	LU	Jacobi	Seidel	SOR
327.6	339.775	2299.9	2300.26	1604.08