Лабораторная работа №4. Решение СЛАУ.

1. Создадим функцию, решающую СЛАУ методом LU-разложений. Для начала создадим функции, разделяющую матрицу на L (нижняя треугольная) и U (верхняя треугольная)

```
def get_L(m):
    L = np.identity(m.shape[0])
    for i in range(1, L.shape[0]):
        for j in range(i):
            L[i, j] = m[i, j] / m[j, j]
    return L


1 usage
def get_U(m):
    U = m.copy()
    for i in range(U.shape[0]):
        for j in range(i):
            U[i, j] = 0
    return U
```

Затем реализуем саму функцию LU-разложения

```python
def LU(A, B):
    n = len(A)
    L = get_L(np.array(A))
    U = get_U(np.array(A))

    y = np.zeros(n)
    for i in range(n):
        y[i] = B[i]
        for j in range(i):
            y[i] -= L[i, j] * y[j]

    x = np.zeros(n)
    for i in reversed(range(n)):
        x[i] = y[i]
        for j in range(i+1, n):
            x[i] -= U[i, j] * x[j]
        x[i] /= U[i, i]

    return x
```

При запуске данного кода на выходе мы получим:
[0.97592014 -0.97472222 0.10291667 1.16041667]

2. Реализуем метод Якоби для решения СЛАУ

```python
def jacobi(A, B, e):
    k = 1
    begin_time = time.perf_counter_ns()
    n = len(A)
    x_next = [B[i] for i in range(n)]

    while True:
        x = x_next.copy()
        x_next = [B[i] for i in range(n)]

        for i in range(n):
            for j in range(n):
                if i != j:
                    x_next[i] -= A[i][j] * x[j]
            x_next[i] /= A[i][i]

        if math.sqrt(sum([(x_next[i] - x[i]) ** 2 for i in range(len(x))])) <= e:
            break

        k += 1

    work_time = time.perf_counter_ns() - begin_time
    return x, k, work_time / 1e6
```

3. Реализуем метод Гаусса-Зеделя для решения СЛАУ

```python
def gauss_zeudel(A, B, e):
    k = 1
    begin_time = time.perf_counter_ns()
    n = len(A)
    x = [1] * n
    x_next = [B[i] for i in range(n)]

    while True:
        for i in range(n):
            x_next[i] = B[i] - sum([A[i][j] * x_next[j] for j in range(n) if j != i]) / A[i][i]

        if math.sqrt(sum([(x_next[i] - x[i]) ** 2 for i in range(len(x))])) <= e:
            break

        k += 1
        x = x_next.copy()

    work_time = time.perf_counter_ns() - begin_time
    return x, k, work_time / 1e6
```

Существенное различие двух этих методов в том, что метод Якоби в каждом итерационном шаге вычисляет новое приближение к каждому неизвестному одновременно, используя значения предыдущего приближения для других неизвестных, а метод Гаусса-Зейделя вычисляет

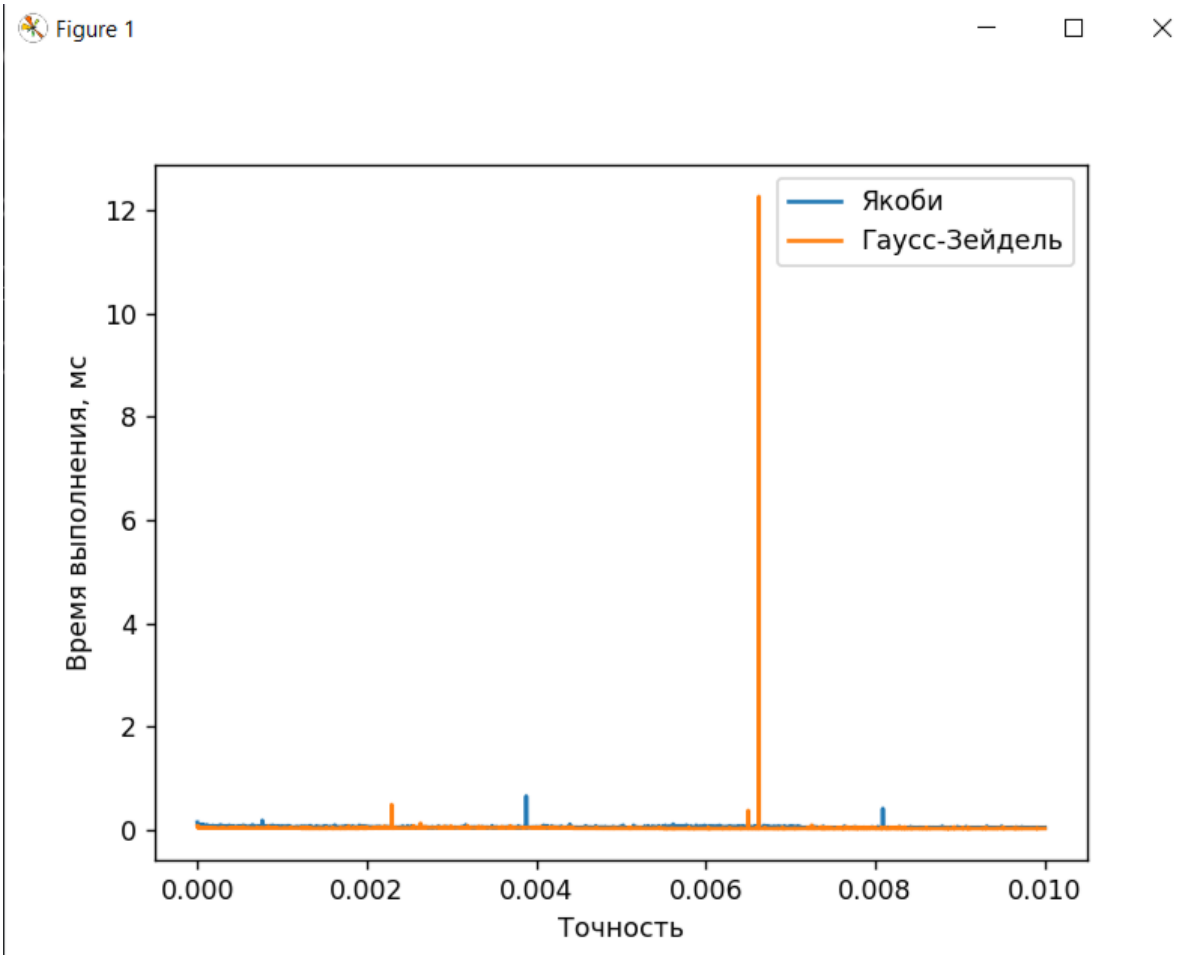
новое приближение к одному неизвестному, используя уже обновленные значения других неизвестных в текущей итерации.

4. Сравним точность этих методов в зависимости от времени и количества итераций.

```
plt.plot( *args: e_vals, [sol[1] for sol in jc], label='Якоби')
plt.plot( *args: e_vals, [sol[1] for sol in gz], label='Гаусс-Зейдель')
plt.xlabel("Точность")
plt.ylabel("Количество итераций")
plt.legend()
plt.show()

plt.plot( *args: e_vals, [sol[2] for sol in jc], label='Якоби')
plt.plot( *args: e_vals, [sol[2] for sol in gz], label='Гаусс-Зейдель')
plt.xlabel("Точность")
plt.ylabel("Время выполнения, мс")
plt.legend()
plt.show()
```

При запуске мы получаем графики, демонстрирующие сравниваемую точность



5. Реализуем решение методом Гаусса-Зейделя в матричном виде.

```python
def gauss_matrix(A, B, e):
    k = 1
    A = np.array(A)
    B = np.array(B)
    begin_time = time.perf_counter_ns()
    n = len(A)
    x = np.array([1] * n)

    while True:
        m1 = np.linalg.inv(np.diag(np.diag(A)) + np.tril(A))
        x_next = np.dot(m1, B) - np.dot(np.dot(m1, np.triu(A)), x)
        dx = sum((x_next[i] - x[i]) ** 2 for i in range(n))

        if np.sqrt(dx) <= e:
            break

        k += 1
        x = x_next.copy()

    work_time = time.perf_counter_ns() - begin_time
    return x, k, work_time / 1e6
```

6. И построим графики зависимости точности от времени выполнения для сравнения Гаусса-Зейделя в матричном и обыкновенном виде

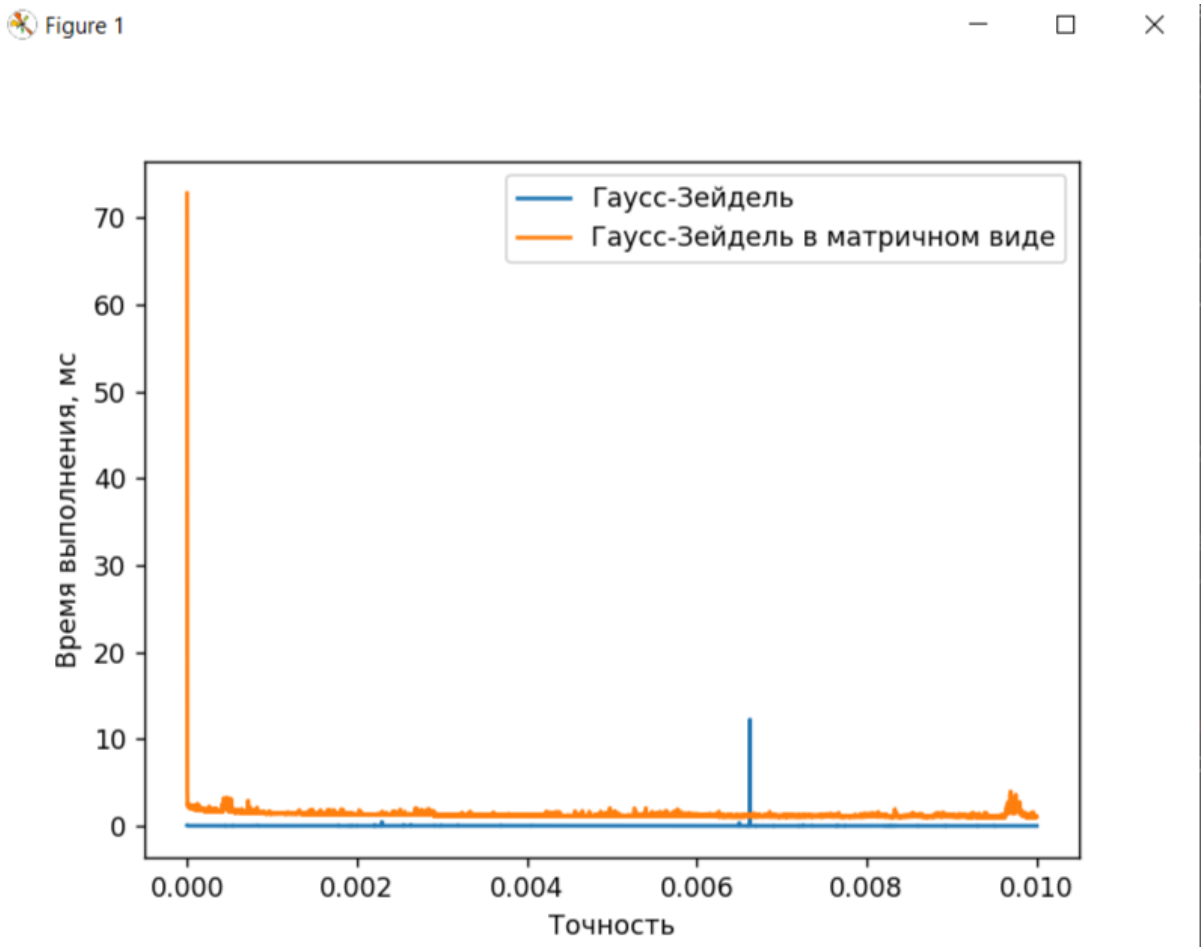


Figure 1
Гаусс-Зейдель
Гаусс-Зейдель в матричном виде
Время выполнения, мс
70
60
50
40
30
20
10
0
0.000
0.002
0.004
0.006
0.008
0.010
Точность