Университет ИТМО

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.03.04 Программная инженерия

Дисциплина «Вычислительная математика»

Отчет

По лабораторной работе №2

Вариант: 2аг

Выполнил:

Совенко Е.В.

P32212

Преподаватель:

Перл Ольга Вячеславовна

Санкт-Петербург, 2023 г.

Описание метода

Решение нелинейных уравнений:

метод деления пополам

простейший численный метод для решения нелинейных уравнений вида f(x)=0. Предполагается только непрерывность функции f(x). Поиск основывается на теореме о промежуточных значениях

метод простой итерации

один из простейших численных методов решения уравнений. Метод основан на принципе сжимающего отображения, который применительно к численным методам в общем виде также может называться методом простой итерации или методом последовательных приближений. Где следующий шаг итерации вычисляется при помощи предыдущего. Выходом из итерационного процесса является условие:

x_k1 – x_k0 <= eps

Решение систем нелинейных уравнений:

метод простой итерации

Пусть дана система нелинейных уравнений специального вида

$$\left.\begin{aligned} x_1 &= \varphi_1(x_1, x_2, \ldots, x_n), \\ x_2 &= \varphi_2(x_1, x_2, \ldots, x_n), \\ &\cdots\cdots\cdots\cdots \\ x_n &= \varphi_n(x_1, x_2, \ldots, x_n), \end{aligned}\right\} \qquad (1)$$

где функции $\varphi_1, \varphi_2, \ldots, \varphi_n$ действительны, определены и непрерывны в некоторой окрестности ω изолированного решения $(x_1^*, x_2^*, \ldots, x_n^*)$ этой системы.

Введя в рассмотрение векторы

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_n) \quad \text{и} \quad \boldsymbol{\varphi}(\boldsymbol{x}) = (\varphi_1(\boldsymbol{x}), \varphi_2(\boldsymbol{x}), \ldots, \varphi_n(\boldsymbol{x})),$$

систему (1) можно записать более кратко:

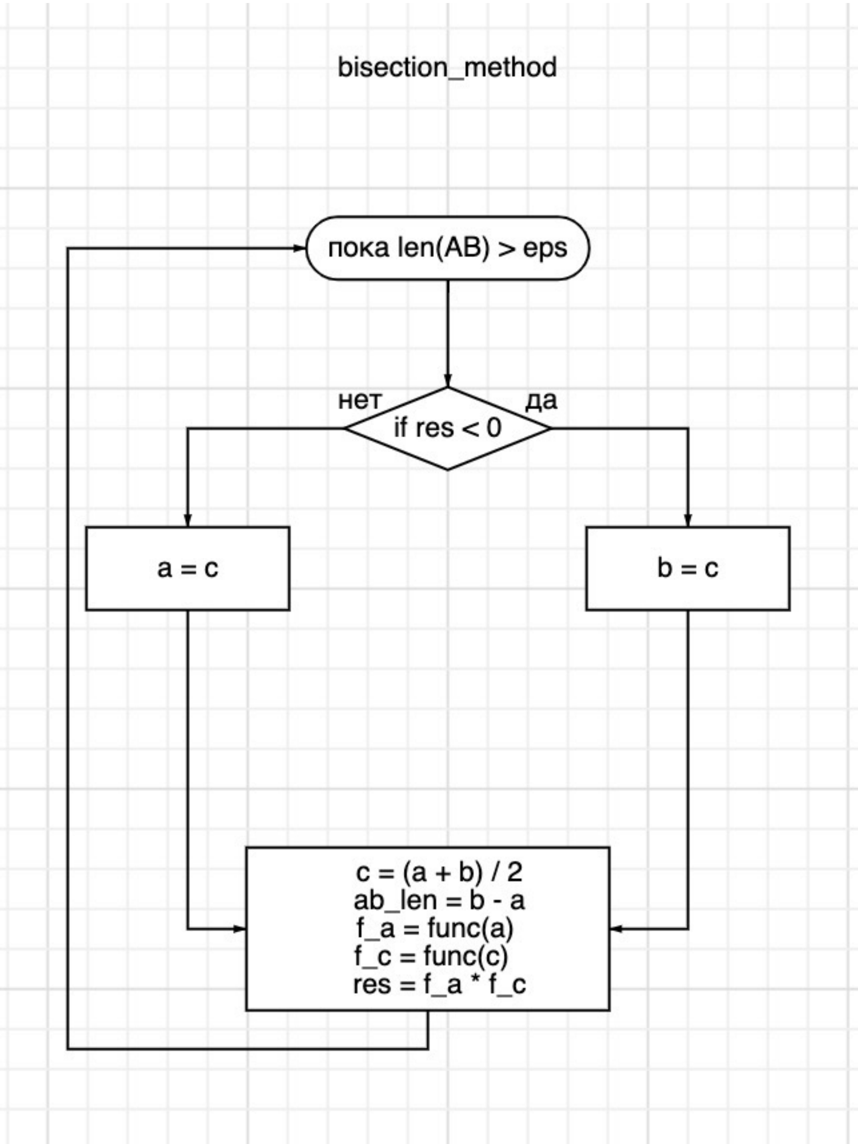
$$\boldsymbol{x} = \boldsymbol{\varphi}(\boldsymbol{x}). \qquad (2)$$

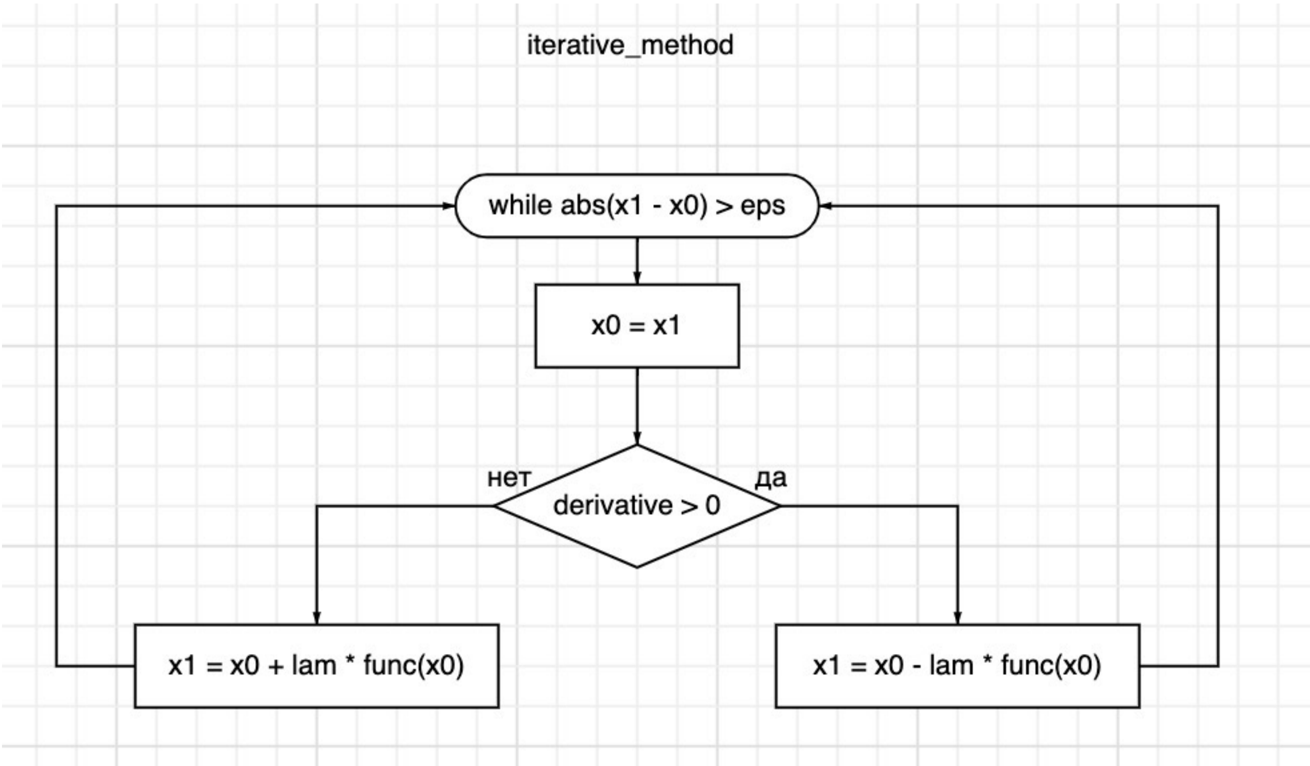
Для нахождения вектор-корня $\boldsymbol{x}^* = (x_1^*, x_2^*, \ldots, x_n^*)$ уравнения (2) часто удобно использовать *метод итерации*

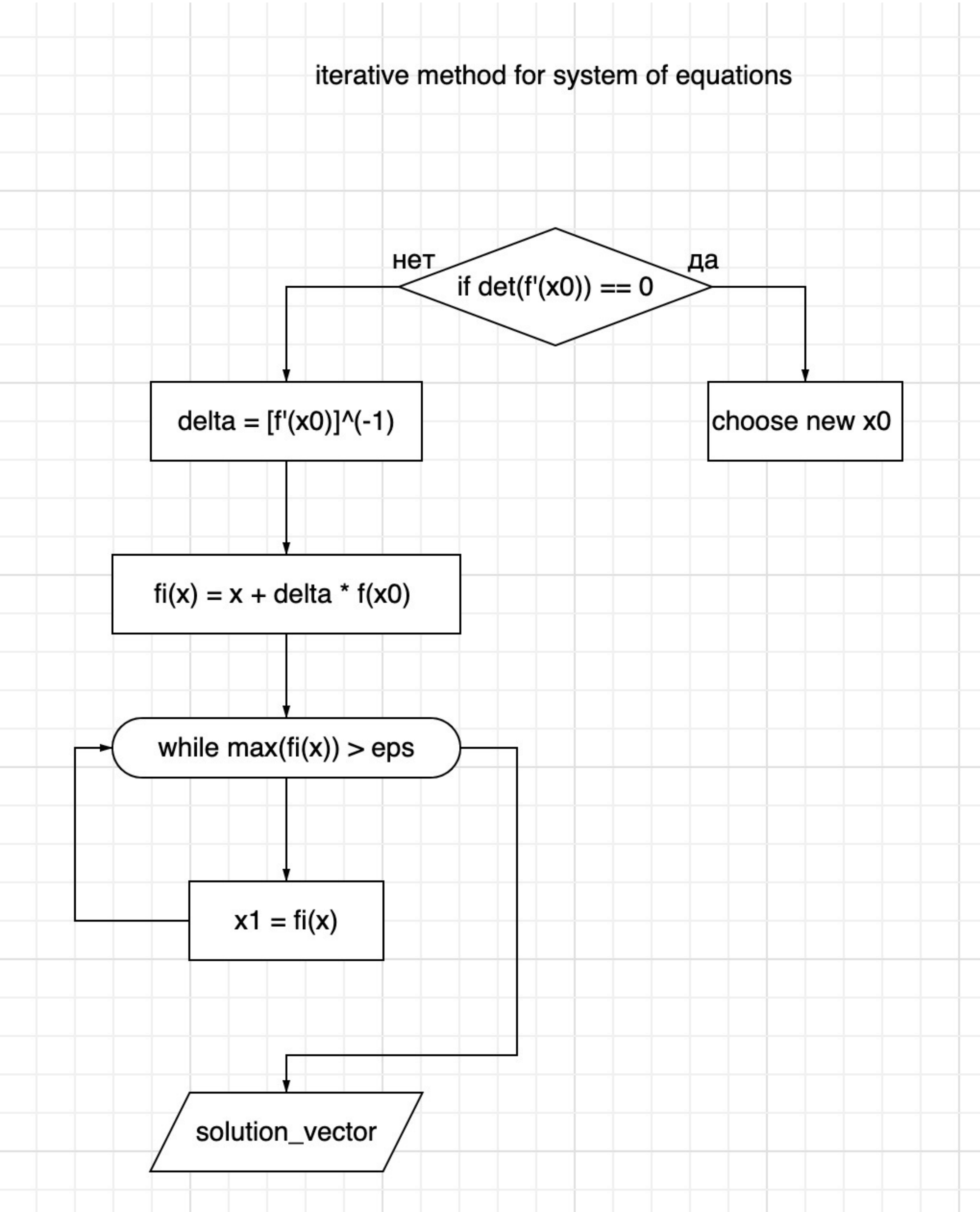
$$\boldsymbol{x}^{(p+1)} = \boldsymbol{\varphi}(\boldsymbol{x}^{(p)}) \qquad (p = 0, 1, 2, \ldots), \qquad (3)$$

Max(x_k1 – x_k0) <= eps

Блок схема


bisection_method
пока len(AB) > eps
нет
if res < 0
да
a = c
b = c
c = (a + b) / 2
ab_len = b - a
f_a = func(a)
f_c = func(c)
res = f_a * f_c


iterative_method
while abs(x1 - x0) > eps
x0 = x1
derivative > 0
нет
да
x1 = x0 + lam * func(x0)
x1 = x0 - lam * func(x0)


iterative method for system of equations
нет
if det(f'(x0)) == 0
да
delta = [f'(x0)]^(-1)
choose new x0
fi(x) = x + delta * f(x0)
while max(fi(x)) > eps
x1 = fi(x)
solution_vector

Функция, реализовывающая сам метод

Решение нелинейных уравнений:

метод деления пополам

```
c = (a + b) / 2  # middle AB
ab_len = b - a  # len AB
f_a = func(a)
f_c = func(c)
res = f_a * f_c

while ab_len >= eps:
    if res < 0:
        b = c
    else:
        a = c

    c = (a + b) / 2
    ab_len = b - a
    f_a = func(a)
    f_c = func(c)
    res = f_a * f_c

RESULT['bisection_method'] = c
return c
```

метод простой итерации

```python
def iterative_method(func, a, b, eps):
    is_monotone = check_sign_derivative_on_interval(func, a, b)
    if not is_monotone:
        RESULT['iterative_method'] = None
        print('Function must be monotone')
        return

    derivative_sign = -1 if calculate_derivative(func, a) < 0 else 1
    d_a = abs(calculate_derivative(func, a))
    d_b = abs(calculate_derivative(func, b))
    m = min(d_a, d_b)
    M = max(d_a, d_b)

    lam = 1 / M
    alpha = 1 - (m / M)
    if not 0 <= alpha < 1:
        print('The convergence condition is not met')
        RESULT['iterative_method'] = None
        return

    x0 = a
    x1 = a + 2 * eps
```

```python
    while abs(x1 - x0) > eps:
        x0 = x1
        if not a <= x1 <= b:
            print('No solutions')
            RESULT['iterative_method'] = None
            return

        if derivative_sign > 0:
            x1 = x0 - lam * func(x0)
        else:
            x1 = x0 + lam * func(x0)

    if not a <= x1 <= b:
        print('no solutions')
        RESULT['iterative_method'] = None
        return

    RESULT['iterative_method'] = x1
    return x1
```

Решение систем нелинейных уравнений:

метод простой итерации

```python
def iterative_method_matrix_nxn(matrix: list[list], free_column: list, x0, eps, debug_info: bool = None):
    if len(matrix) == 0:
        raise IOError('No matrix')

    if len(matrix) != len(matrix[0]):
        raise IOError('Matrix must be a square')

    max_error = eps + 1
    x1 = []
    count_of_iteration = 1
    while max_error > eps:
        f_x0 = []
        for i in range(len(matrix)):
            funcs_x0 = [func(x0[j]) for j, func in enumerate(matrix[i])]
            funcs_x0.append((-1) * free_column[i])
            f_x0.append(sum(funcs_x0))

        f_der_x0 = []
        for i in range(len(matrix)):
            f_der_x0.append([calculate_derivative(func, x0[j]) for j, func in enumerate(matrix[i])])

        det_f_der_x0 = calculate_determinant(f_der_x0)
        if det_f_der_x0 == 0:
            print('The special matrix was met')
            return
        inverse_matrix_f_der_x0 = list(LA.inv(f_der_x0))
        for i in range(len(f_der_x0)):
            inverse_matrix_f_der_x0[i] = list(inverse_matrix_f_der_x0[i])

        inverse_multiple_func = np.matmul(inverse_matrix_f_der_x0, f_x0)
        x1 = [x0[i] - inverse_multiple_func[i] for i in range(len(x0))]
        max_error = max([abs(x1[i] - x0[i]) for i in range(len(x0))])
        x0 = x1[:]

        count_of_iteration += 1

    if debug_info is True:
        print(f'count of iterations: {count_of_iteration}')
        print(f'max error: {max_error}')

        colors = ['r', 'b', 'g', 'y', 'c']
        if len(matrix) <= len(colors):
            dots_x = np.arange(-x0[0] * 5, x0[0] * 5 + 0.1, 0.1)

            for row in range(len(matrix)):
                dots_y = []
                for x in dots_x:
                    res = 0
                    for column in range(len(matrix[row])):
                        func = matrix[row][column]
                        res += func(x)

                    res -= free_column[row]
```

```python
                res -= free_column[row]
                dots_y.append(res)

            plt.plot(dots_x, dots_y, colors[row])

        plt.plot(dots_x, [0 for _ in range(len(dots_x))], 'g')
        plt.show()

    print()

return x1
```

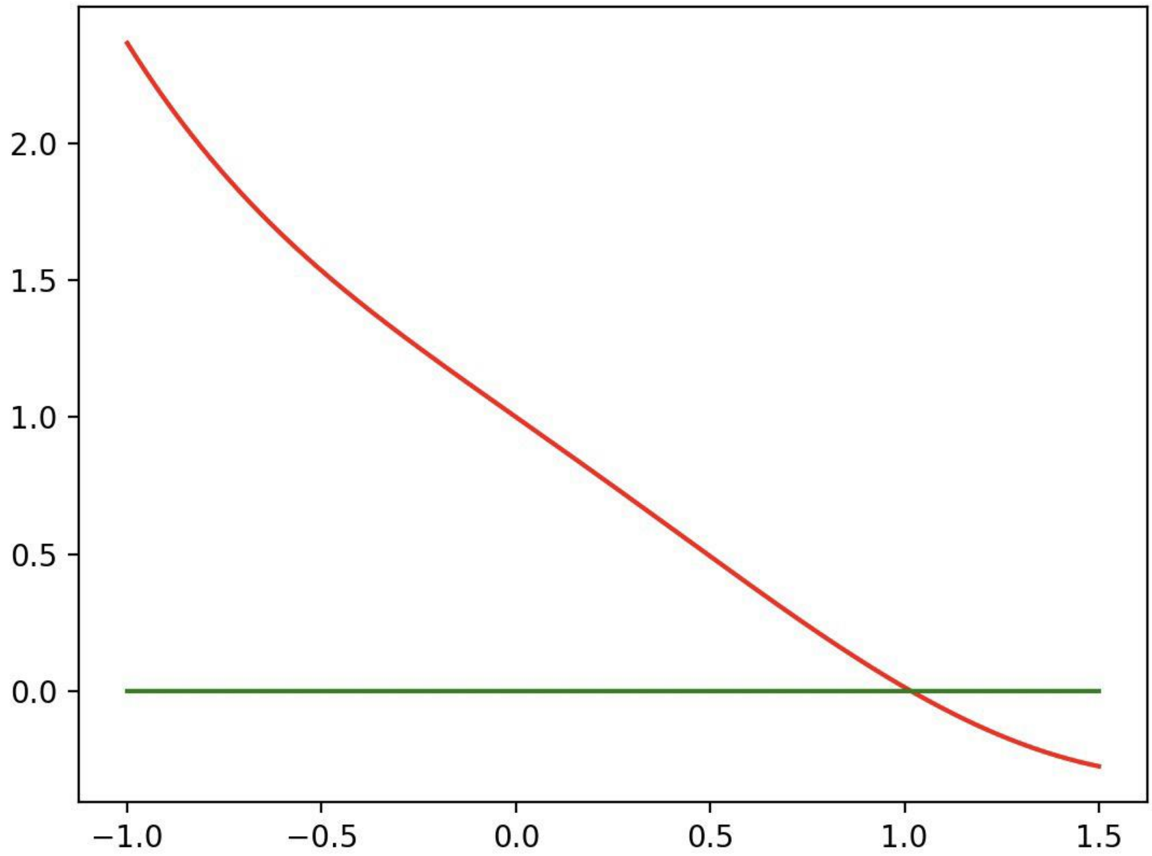
Проверка решения, путем подставления значений

Пример работы программы

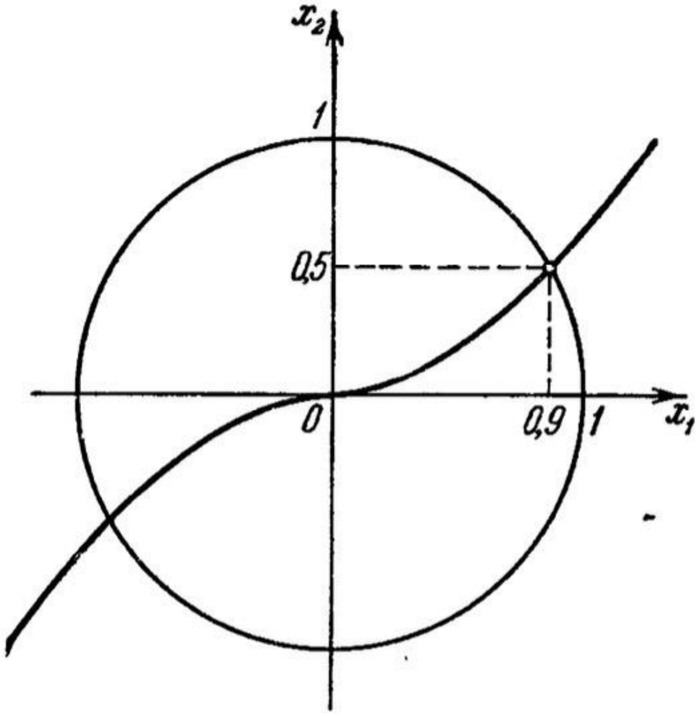
Входные данные:

```python
solve_both_methods(func=lambda x: exp(-x) - math.sin(x) ** 2 / 2,
                   interval_left=-1,
                   interval_right=1.5,
                   step=1e-5)
```

Результат работы:




```
bisection_method: 1.0169687271118164
iterative_method 1.0169527355873926
```



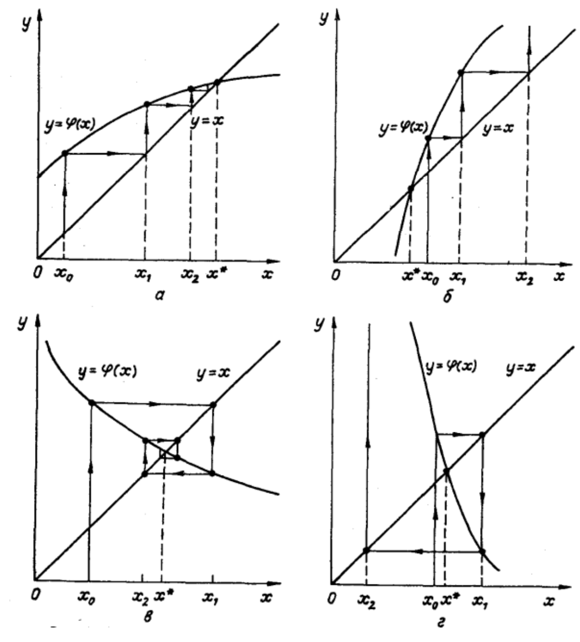

```
system_of_equations = [[lambda x: x ** 2, lambda x: x ** 2],
                       [lambda x: x ** 3, lambda x: (-1) * x]]
free_column = [1, 0]

res = iterative_method_matrix_nxn(matrix=system_of_equations,
                                  free_column=free_column,
                                  x0=[0.7, 0.4],
                                  eps=1e-5,
                                  debug_info=False)
if res:
    print_solution_vector(res)
else:
    print('No solution')
```

```
x1: 0.8260313576544946
x2: 0.563624162161103
```

Вывод

Во время выполнения лабораторной работы я изучил метод простых итераций и метод деления пополам. Метод деления пополам стремится к поиску решения на заданном отрезке, уменьшая диапазон поиска в 2 раза при каждой итерации. Соответственно предел на отрезке будет всегда, но если ответ сходится в концах отрезках, то его нужно проверить. Также необходимым условием является монотонность и f(a)*f(b) < 0. При этом если функция не монотонна, но на отрезке есть решение, он может найти его, а может не найти (при вышеуказанных условиях гарантируется поиск решения, если не соблюдены – то или-или)

Метод простых итераций, также требует монотонности. В ином случае см. картинку. Значения могут стремится к бесконечности.



В Решение систем нелинейных уравнений: метод простых итераций использует вектор начального приближения. После каждое следующее приближение зависит от вектора предыдущего. Условия схожи с методом простых итераций для решения нелинейных уравнений.