

Московский авиационный институт  
(национальный исследовательский университет)

Факультет информационных технологий и прикладной  
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Численные методы»  
Вариант №1

Студент: А. О. Дубинин  
Преподаватель: И. Э. Иванов  
Группа: М8О-306Б  
Дата:  
Оценка:  
Подпись:

Москва, 2020

# Лабораторная работа №1

**Задача:** Разработать методы решения задач линейной алгебры.

## Вариант 1

**1.1** Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

**1.2** Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

# 1 Решение

Согласно [1], численное решение нелинейных (алгебраических или трансцендентных) уравнений вида:

$$f(x) = 0$$

заключается в нахождении значений  $x$ , удовлетворяющих с заданной точностью данному уравнению, и состоит из следующих основных этапов:

- отделение (изоляция, локализация) корней уравнения;
- уточнение с помощью некоторого вычислительного алгоритма конкретного выделенного корня с заданной точностью.

Целью первого этапа является нахождение отрезков из области определения функции  $f(x)$ , внутри которых содержится только один корень решаемого уравнения. Иногда ограничиваются рассмотрением лишь какой-нибудь части области определения, вызывающей по тем или иным соображениям интерес. Для реализации данного этапа используются графические или аналитические способы. При аналитическом способе отделения корней полезна следующая теорема.

Теорема 2.1.

Непрерывная строго монотонная функция  $f(x)$  имеет единственный нуль на отрезке  $[a, b]$  тогда и только тогда, когда на его концах она принимает значения разных знаков.

Достаточным признаком монотонности функции  $f(x)$  на отрезке  $[a, b]$  является сохранение знака производной функции.

Графический способ отделения корней целесообразно использовать в случае, когда имеется возможность построения графика функции  $y = f(x)$ . Наличие графика исходной функции дает непосредственное представление о количестве и расположении нулей функции, что позволяет определить промежутки, внутри которых содержится только один корень.

Так или иначе, при завершении первого этапа должны быть определены промежутки, на каждом из которых содержится только один корень уравнения.

Для уточнения корня с требуемой погрешностью обычно применяется какой-либо итерационный метод, заключающийся в построении числовой последовательности  $x^{(k)}$ ,  $k = 0, 1, 2, \dots$ , сходящейся к искомому корню  $x^{(*)}$  уравнения.

## 1.1.1 Метод Ньютона для одного уравнения.

При нахождении корня уравнения методом Ньютона итерационный процесс определяется формулой

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Для начала вычислений требуется задание начального приближения  $x^{(0)}$ . Условия сходимости метода определяются следующей теоремой

**Теорема 2.2.**

Пусть на отрезке  $[a, b]$  функция  $f(x)$  имеет первую и вторую производные постоянного знака и пусть  $f(a) * f(b) < 0$ . Тогда, если точка  $x^{(0)}$  выбрана на  $[a, b]$  так, что

$$f(x^{(0)}) * f'(x^{(0)}) > 0$$

то начатая с нее последовательность  $x^{(k)}$ ,  $k = 0, 1, 2, \dots$ , определяемая методом Ньютона, монотонно сходится к корню  $x^{(*)} \in (a, b)$  уравнения.

В качестве условия окончания итераций в практических вычислениях часто используется правило  $|x^{(k+1)} - x^{(k)}| < \epsilon \Rightarrow x^{(*)} \approx x^{(k+1)}$ .

### 1.1.2 Метод простой итерации для одного уравнения.

При использовании метода простой итерации уравнение заменяется эквивалентным уравнением с выделенным линейным членом

$$x = \phi(x)$$

Решение ищется путем построения последовательности

$$x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2, \dots,$$

начиная с некоторого заданного значения  $x^{(0)}$ . Если  $\phi(x)$  – непрерывная функция, а  $x^{(k)}$ ,  $k = 0, 1, 2, \dots$ , – сходящаяся последовательность, то значение  $x^{(*)} = \lim_{k \rightarrow \infty} x^{(k)}$  является решением уравнения. Условия сходимости метода и оценка его погрешности определяются теоремой, доказанной выше.

**Теорема 2.3.**

Пусть функция  $\phi(x)$  определена и дифференцируема на отрезке  $[a, b]$ . Тогда, если выполняются условия

$$\phi(x) \in [a, b], \forall x \in [a, b],$$

$$\exists q : |\phi'(x)| \leq q < 1 \forall x \in [a, b],$$

то уравнение имеет единственный на  $[a, b]$  корень  $x^{(*)}$  к этому корню сходится определяемая методом простой итерации последовательность  $x^{(k)}$ ,  $k = 0, 1, 2, \dots$ , начинающаяся с любого  $x^{(0)} \in [a, b]$ . При этом справедливы оценки погрешности ( $\forall k \in N$ ):

$$|x^{(*)} - x^{(k+1)}| \leq \frac{q}{1-q} |x^{(k+1)} - x^{(k)}|,$$

$$|x^{(*)} - x^{(k+1)}| \leq \frac{q^{(k+1)}}{1-q} |x^{(1)} - x^{(0)}|,$$

Систему нелинейных уравнений с  $n$  неизвестными можно записать в виде

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0, \\ f_2(x_1, x_2, \dots, x_n) &= 0, \\ &\dots\dots\dots \\ f_n(x_1, x_2, \dots, x_n) &= 0, \end{aligned}$$

или, более коротко, в векторной форме

$$f(x) = 0$$

где  $x$  — вектор неизвестных величин,  $f$  — вектор-функция

В редких случаях для решения такой системы удастся применить метод последовательного исключения неизвестных и свести решение исходной задачи к решению одного нелинейного уравнения с одним неизвестным. Значения других неизвестных величин находятся соответствующей подстановкой в конкретные выражения. Однако в подавляющем большинстве случаев для решения систем нелинейных уравнений используются итерационные методы. В дальнейшем предполагается, что ищется изолированное решение нелинейной системы.

Замечание. Как и в случае одного нелинейного уравнения, локализация решения может осуществляться на основе специфической информации по конкретной решаемой задаче (например, по физическим соображениям) и с помощью методов математического анализа. При решении системы двух уравнений часто удобным является графический способ, когда месторасположение корней определяется как точки пересечения кривых  $f_1(x_1, x_2) = 0$ ,  $f_2(x_1, x_2) = 0$  на плоскости  $(x_1, x_2)$

### 1.2.1 Метод Ньютона для системы уравнений.

Если определено начальное приближение  $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})^T$  итерационный процесс нахождения решения системы методом Ньютона можно представить в виде

$$\begin{aligned} x_1^{(k+1)} &= x_1^{(k)} + \Delta x_1^{(k)} \\ x_2^{(k+1)} &= x_2^{(k)} + \Delta x_2^{(k)} \\ &\dots\dots\dots \\ x_n^{(k+1)} &= x_n^{(k)} + \Delta x_n^{(k)} \end{aligned}$$

где значения приращений  $\Delta x_1^{(k)}, \Delta x_2^{(k)}, \dots, \Delta x_n^{(k)}$  определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение  $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$

$$\begin{aligned} f_1(x^{(k)}) + \frac{\delta f_1(x^{(k)})}{\delta x_1} \Delta x_1^{(k)} + \frac{\delta f_1(x^{(k)})}{\delta x_2} \Delta x_2^{(k)} + \dots + \frac{\delta f_1(x^{(k)})}{\delta x_n} \Delta x_n^{(k)} &= 0 \\ f_2(x^{(k)}) + \frac{\delta f_2(x^{(k)})}{\delta x_1} \Delta x_1^{(k)} + \frac{\delta f_2(x^{(k)})}{\delta x_2} \Delta x_2^{(k)} + \dots + \frac{\delta f_2(x^{(k)})}{\delta x_n} \Delta x_n^{(k)} &= 0 \\ \dots &\dots \\ f_n(x^{(k)}) + \frac{\delta f_n(x^{(k)})}{\delta x_1} \Delta x_1^{(k)} + \frac{\delta f_n(x^{(k)})}{\delta x_2} \Delta x_2^{(k)} + \dots + \frac{\delta f_n(x^{(k)})}{\delta x_n} \Delta x_n^{(k)} &= 0 \end{aligned}$$

В векторно-матричной форме расчетные формулы имеют вид

$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$$

где вектор приращений  $\Delta x^{(k)}$  находится из решения уравнения

$$f(x^{(k)}) + J(x^{(k)})\Delta x^{(k)} = 0$$

Выражая вектор приращений и подставляя его , итерационный процесс нахождения решения можно записать в виде

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})f(x^{(k)})$$

Использование метода Ньютона предполагает дифференцируемость функций  $f_1(x), f_2(x), \dots, f_n(x)$  и невырожденность матрицы Якоби. В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная.

В практических вычислениях в качестве условия окончания итераций обычно используется критерий.

$$\|x^{(k+1)} - x^{(k)}\| \leq \epsilon$$

**1.2.2** Метод простых итераций для системы уравнений. При использовании метода простой итерации система уравнений приводится к эквивалентной системе специального вида

$$\begin{aligned} x_1 &= \phi_1(x_1, x_2, \dots, x_n) \\ x_2 &= \phi_2(x_1, x_2, \dots, x_n) \end{aligned}$$

.....

$$x_n = \phi_n(x_1, x_2, \dots, x_n)$$

или, в векторной форме

$$x = \phi(x)$$

Если выбрано некоторое начальное приближение  $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})^T$  последующие приближения в методе простой итерации находятся по формулам в векторной форме

$$x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2, \dots$$

Достаточное условие сходимости итерационного процесса формулируется следующим образом:

Теорема 2.4. Пусть вектор-функция  $\phi(x), \phi'(x)$  в ограниченной выпуклой замкнутой области  $G$  и

$$\max \|\phi'(x^{(k)})\| \leq q < 1$$

где  $q$  - постоянная. Если  $x^{(0)} \in G$  и все последовательные приближения

$$x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2, \dots$$

также содержатся в  $G$ , то процесс итерации сходится к единственному решению уравнения

$$x = \phi(x)$$

в области  $G$  и справедливы оценки погрешности  $\forall k \in N$ :

$$\begin{aligned} \|x^{(*)} - x^{(k+1)}\| &\leq \frac{q^{k+1}}{1-q} \|x^{(1)} - x^{(0)}\|, \\ \|x^{(*)} - x^{(k+1)}\| &\leq \frac{q}{1-q} \|x^{(k+1)} - x^{(k)}\|, \end{aligned}$$

## 2 Исходный код

### 1.1 Метод ньютона и простой итерации для одного уравнения.

```
1 import argparse
2 import numpy as np
3
4
5 class Solver:
6     def __init__(self, eps, output_name, log_name):
7         self.eps = eps
8         self.out_file = output_name
9         self.log_file = log_name
10        if self.log_file:
11            open(self.log_file, 'w').close()
12        self.area = (1, 3)
13        self.x0 = 1.25
14        self.lmbd = self.calc_lambda()
15        self.q = self.calc_q()
16        self.k_iter = 0
17        self.k_newton = 0
18        self.check()
19        self.iter_x = self.iter_method()
20        self.newtons_x = self.newtons_method()
21
22    def check(self):
23        with open(self.log_file, 'a') as f_log:
24            f_log.write(f'q = {self.q}\n')
25            f_log.write(f'lambda = {self.lmbd}\n')
26            x = np.linspace(self.area[0], self.area[1], 10000)
27            y = [self.phi_derivative(i) for i in x]
28            if all([i < 1 for i in y]):
29                f_log.write('phi\' < 1\n')
30            else:
31                f_log.write('phi\' >= 1\n')
32            if self.q < 1:
33                f_log.write('q < 1\n')
34            else:
35                f_log.write('q >= 1\n')
36
37    @staticmethod
38    def f(x):
39        return 2**x - x**2 - 0.5
40
41    def phi(self, x):
42        return x - self.lmbd * self.f(x)
43
44    def phi_derivative(self, x):
```



```

45         return 1 - self.lmbd * self.f_derivative(x)
46
47     staticmethod
48     def f_derivative(x):
49         return 2**x * np.log(2) - 2*x
50
51     staticmethod
52     def f_2derivative(x):
53         return 2**x * np.log(2)**2 + 2**x * 0.5 - 2
54
55     def calc_q(self):
56         x = np.linspace(self.area[0], self.area[1], 10000)
57         y = [abs(self.phi_derivative(i)) for i in x]
58         q = np.max(y)
59         return q
60
61     def calc_lambda(self):
62         flag = None
63         x = np.linspace(self.area[0], self.area[1], 10000)
64         y = [self.f_derivative(i) for i in x]
65
66         if all([np.sign(i) == -1 for i in y]):
67             flag = -1
68         elif all([np.sign(i) == 1 for i in y]):
69             flag = 1
70         else:
71             if self.log_file:
72                 with open(self.log_file, 'a') as fl:
73                     fl.write('Error: Derivative change sign\n')
74             exit(-1)
75
76         y = [abs(self.f_derivative(i)) for i in x]
77         return flag / np.max(y)
78
79     def iter_method(self):
80         x_old = self.x0
81         if self.log_file:
82             with open(self.log_file, 'a') as fl:
83                 fl.write(f'Iter:\nx{self.k_iter} = {x_old}\n')
84         while True:
85             self.k_iter += 1
86             x_new = self.phi(x_old)
87             if self.log_file:
88                 with open(self.log_file, 'a') as fl:
89                     fl.write(f'x{self.k_iter} = {x_new}\n')
90             if abs(x_new - x_old) * self.q / (1 - self.q) < self.eps:
91                 return x_new
92             else:
93                 x_old = x_new

```

```

94
95     def newtons_method(self):
96         x_old = self.x0
97         if self.log_file:
98             with open(self.log_file, 'a') as fl:
99                 fl.write(f'Newton\'s: \n{x_{self.k_iter} = {x_old}\n')
100         while True:
101             self.k_newton += 1
102             x_new = x_old - self.f(x_old) / self.f_derivative(x_old)
103             if self.log_file:
104                 with open(self.log_file, 'a') as fl:
105                     fl.write(f'x_{self.k_newton} = {x_new}\n')
106             if abs(x_new - x_old) < self.eps:
107                 return x_new
108             else:
109                 x_old = x_new
110
111     def print_solution(self):
112         with open(self.out_file, 'w') as f_out:
113             f_out.write(f'EPS = {self.eps}\n')
114             f_out.write(f'Iter: x = {self.iter_x}\n')
115             f_out.write(f'Steps = {self.k_iter}\n')
116             f_out.write(f'Newton: x = {self.newtons_x}\n')
117             f_out.write(f'Steps = {self.k_newton}\n')
118
119
120 def main():
121     parser = argparse.ArgumentParser()
122     parser.add_argument('--eps', type=float, required=True, help='Accuracy')
123     parser.add_argument('--output', required=True, help='File for answer')
124     parser.add_argument('--log', help='Logging')
125     args = parser.parse_args()
126
127     sol = Solver(args.eps, args.output, args.log)
128     sol.print_solution()
129
130 if __name__ == "__main__":
131     main()

```

## 1.2 Метод ньютона и простой итерации для системы уравнений.

```

1 import argparse
2 import numpy as np
3 from numpy.linalg import norm, solve, det
4 from itertools import product
5
6
7 class Solver:
8     def __init__(self, eps, output_name, log_name):

```

```

9         self.eps = eps
10        self.out_file = output_name
11        self.log_file = log_name
12        if self.log_file:
13            open(self.log_file, 'w').close()
14        self.area = ((2.5, 3.25), (0.5, 0.75))
15        self.x0 = [2.5, 0.5]
16        self.k_iter = 0
17        self.k_newton = 0
18        self.lmbd = self.calc_lambda()
19        self.q = self.calc_q()
20        self.iter_x = self.iter_method()
21        self.newtons_x = self.newtons_method()
22
23        staticmethod
24        def f1(x1, x2):
25            return (x1**2 + 4) * x2 - 8
26
27        staticmethod
28        def f2(x1, x2):
29            return (x1 - 1)**2 + (x2 - 1)**2 - 4
30
31        staticmethod
32        def f11(x1, x2):
33            return 2 * x1 * x2
34
35        staticmethod
36        def f12(x1):
37            return x1**2 + 4
38
39        staticmethod
40        def f21(x1):
41            return 2 * x1 - 2
42
43        staticmethod
44        def f22(x2):
45            return 2 * x2 - 2
46
47        def phi1(self, x1, x2):
48            return x1 - (self.f1(x1, x2) * self.lmbd[0, 0] + self.f2(x1, x2) *
49                        self.lmbd[0, 1])
50
51        def phi2(self, x1, x2):
52            return x2 - (self.f1(x1, x2) * self.lmbd[1, 0] + self.f2(x1, x2) *
53                        self.lmbd[1, 1])
54
55        def phi11(self, x1, x2):
56            return 1 - (self.f11(x1, x2) * self.lmbd[0, 0] + self.f21(x1) *
57                        self.lmbd[0, 1])

```

```

58
59 def phi12(self, x1, x2):
60     return -(self.f12(x1) * self.lmbd[0, 0] + self.f22(x2) *
61             self.lmbd[0, 1])
62
63 def phi21(self, x1, x2):
64     return -(self.f11(x1, x2) * self.lmbd[1, 0] + self.f21(x1) *
65             self.lmbd[1, 1])
66
67 def phi22(self, x1, x2):
68     return 1 - (self.f12(x1) * self.lmbd[1, 0] + self.f22(x2) *
69               self.lmbd[1, 1])
70
71 def phi_derivative(self, x):
72     return np.array([[self.phi11(*x), self.phi12(*x)],
73                     [self.phi21(*x), self.phi22(*x)]])
74
75 def j(self, x1, x2):
76     return [[self.f11(x1, x2), self.f12(x1)], [self.f21(x1), self.f22(x2)]]
77
78 def calc_lambda(self):
79     shape = len(self.area)
80     current_j = self.j(*self.x0)
81     inv_j = np.array([solve(current_j, i) for i in np.eye(shape)])
82     return np.transpose(inv_j)
83
84 def calc_q(self):
85     x1 = np.linspace(self.area[0][0], self.area[0][1], 100)
86     x2 = np.linspace(self.area[1][0], self.area[1][1], 100)
87     points = list(product(x1, x2))
88     vals = [norm(self.phi_derivative(point), np.inf) for point in points]
89     q = np.max(vals)
90     return q
91
92 def iter_method(self):
93     x_old = self.x0
94     if self.log_file:
95         with open(self.log_file, 'a') as fl:
96             fl.write(f'Iter:\nx{self.k_iter} = {x_old}\n')
97     while True:
98         self.k_iter += 1
99         x_new = np.array([self.phi1(*x_old), self.phi2(*x_old)])
100         if self.log_file:
101             with open(self.log_file, 'a') as fl:
102                 fl.write(f'x{self.k_iter} = {x_new}\n')
103         if norm(x_new - x_old, np.inf) * self.q / (1 - self.q) <= self.eps:
104             return x_new
105         else:
106             x_old = x_new

```

```

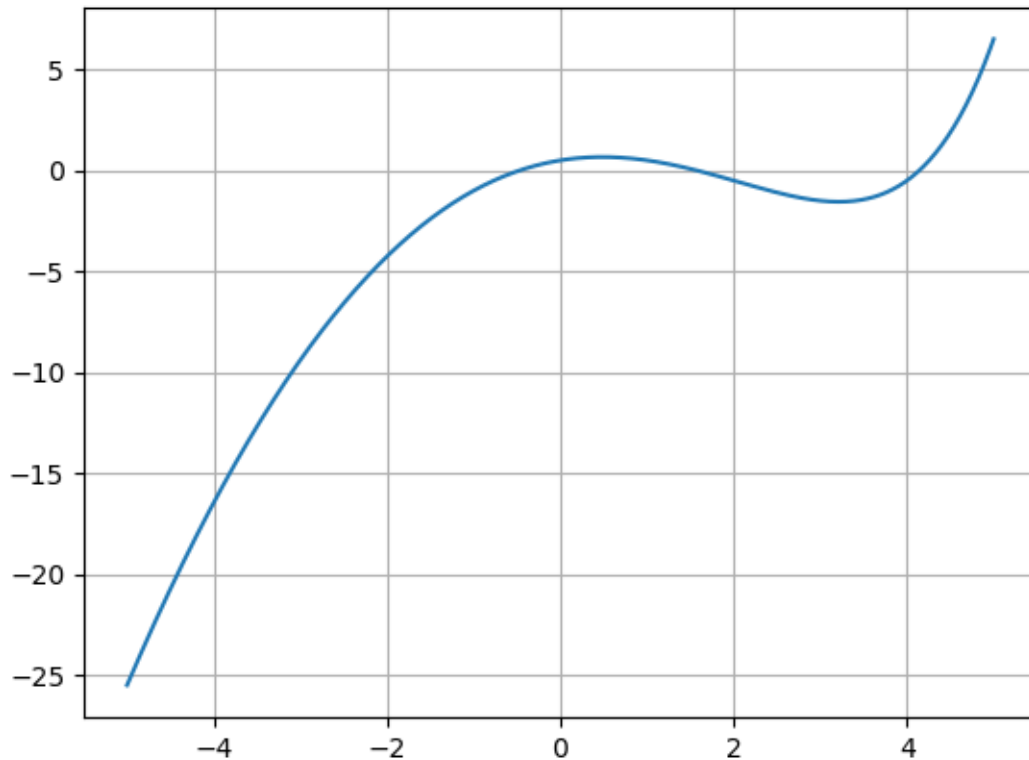
107
108 def newtons_method(self):
109     shape = len(self.area)
110     x_old = self.x0
111     if self.log_file:
112         with open(self.log_file, 'a') as fl:
113             fl.write(f'Newton\'s:\n{x_{self.k_iter} = {x_old}\n}')
114     while True:
115         current_j = self.j(*x_old)
116         if det(current_j) == 0:
117             if self.log_file:
118                 with open(self.log_file, 'a') as fl:
119                     fl.write(f'Error: detJ({self.k_newton} == 0)\n')
120             exit(-1)
121         self.k_newton += 1
122         inv_j = np.array([solve(current_j, i) for i in np.eye(shape)])
123         x_new = x_old - np.transpose(inv_j) np.array([self.f1(*x_old),
124                                                         self.f2(*x_old)])
125         if self.log_file:
126             with open(self.log_file, 'a') as fl:
127                 fl.write(f'x_{self.k_newton} = {x_new}\n')
128         if norm(x_new - x_old, np.inf) <= self.eps:
129             return x_new
130         else:
131             x_old = x_new
132
133 def print_solution(self):
134     if self.log_file:
135         with open(self.log_file, 'a') as f_log:
136             f_log.write(f'q = {self.q}\n')
137             f_log.write(f'Lambda:\n{self.lmbd}\n')
138     with open(self.out_file, 'w') as f_out:
139         f_out.write(f'EPS = {self.eps}\n')
140         f_out.write(f'Iter: x = {self.iter_x}\n')
141         f_out.write(f'Steps = {self.k_iter}\n')
142         f_out.write(f'Newton: x = {self.newtons_x}\n')
143         f_out.write(f'Steps = {self.k_newton}\n')
144
145
146 def main():
147     parser = argparse.ArgumentParser()
148     parser.add_argument('--eps', type=float, required=True, help='Accuracy')
149     parser.add_argument('--output', required=True, help='File for answer')
150     parser.add_argument('--log', help='Logging')
151     args = parser.parse_args()
152
153     sol = Solver(args.eps, args.output, args.log)
154     sol.print_solution()
155

```

```
156 || if __name__ == "__main__":  
157 ||     main()
```

### 3 Вывод программы

1.1 Метод ньютона и простой итерации для одного уравнения.  
Начальное значение определим графически.



Входные данные: Эпсилон, файл для вывода, файл для логирования.

Выходные данные: Конечное значение и кол-во итераций для обоих методов в файле output. И в файле log значения на каждой итерации.

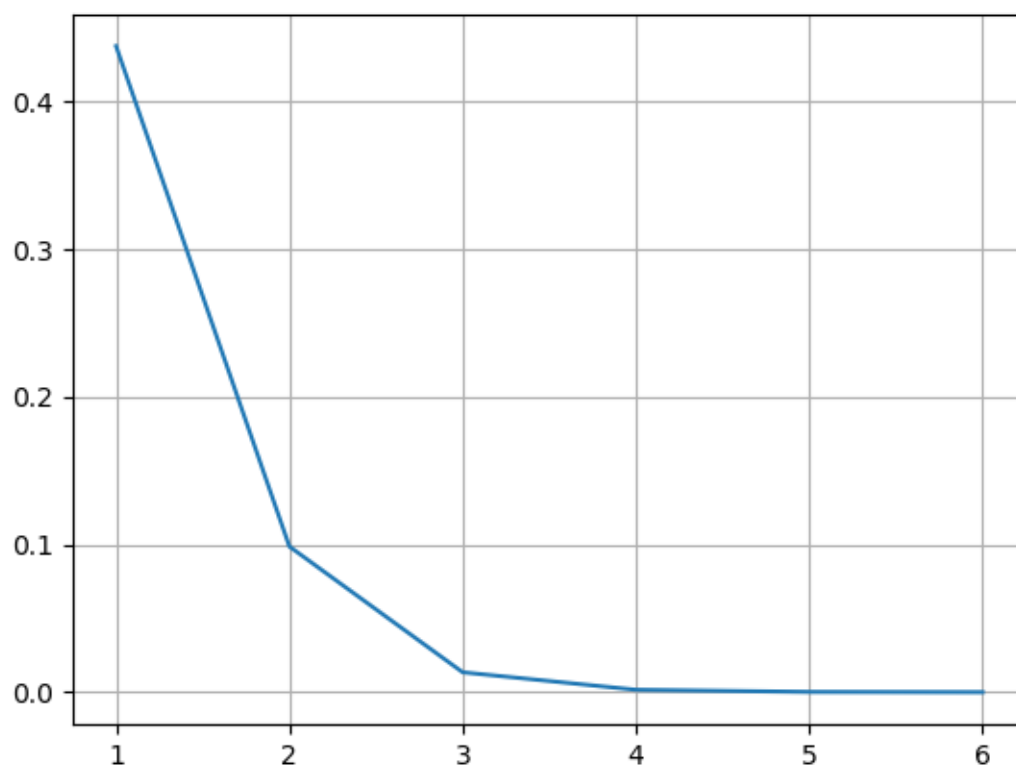
```
art@mars:~/study/NM/lab_2/p_1$ python3 Lw2_1.py --eps 0.0001 --output output
--log log
art@mars:~/study/NM/lab_2/p_1$ cat output
EPS = 0.0001
Iter: x = 1.5738271264645896
Steps = 6
Newton: x = 1.5738289236449503
Steps = 4
art@mars:~/study/NM/lab_2/p_1$ cat log
```

```
q = 0.6301279573390394
lambda = -0.8132227352659079
phi' < 1
q < 1
Iter:
x0 = 1.25
x1 = 1.5069086342344487
x2 = 1.5648348676802541
x3 = 1.5727460695657183
x4 = 1.5737005477535821
x5 = 1.573813732668428
x6 = 1.5738271264645896
Newton's:
x0 = 1.25
x1 = 1.6210487834750746
x2 = 1.574369411050435
x3 = 1.5738290003277182
x4 = 1.5738289236449503
```

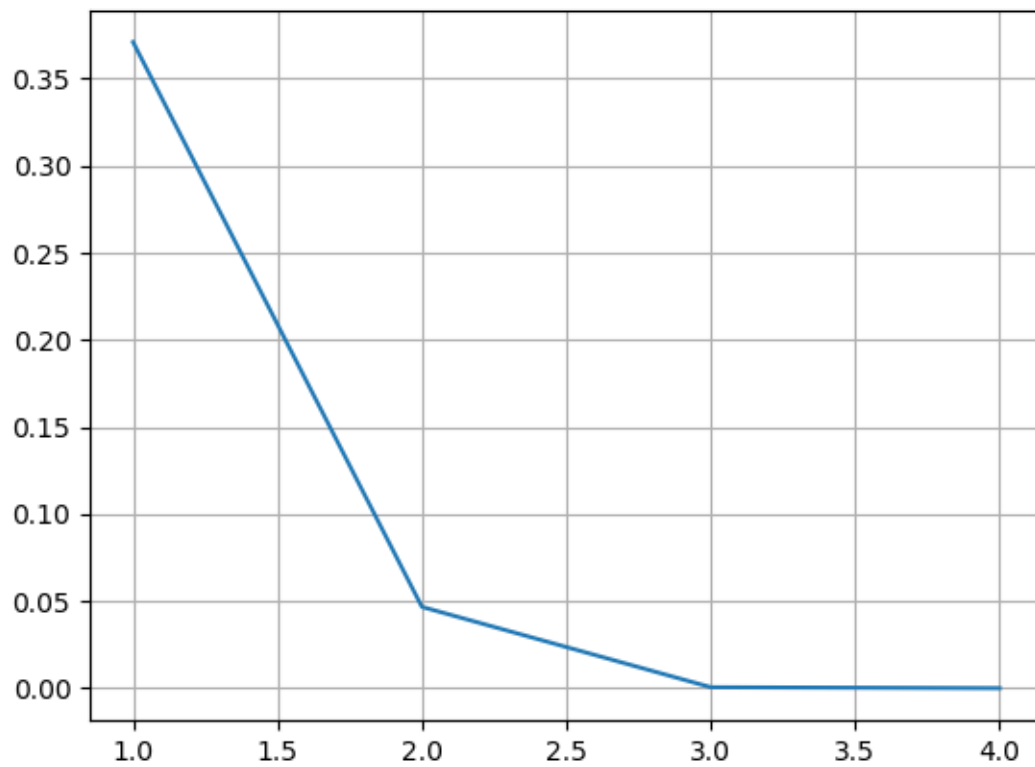
Зависимость погрешности от кол-ва итерации.

Метод простых итераций.

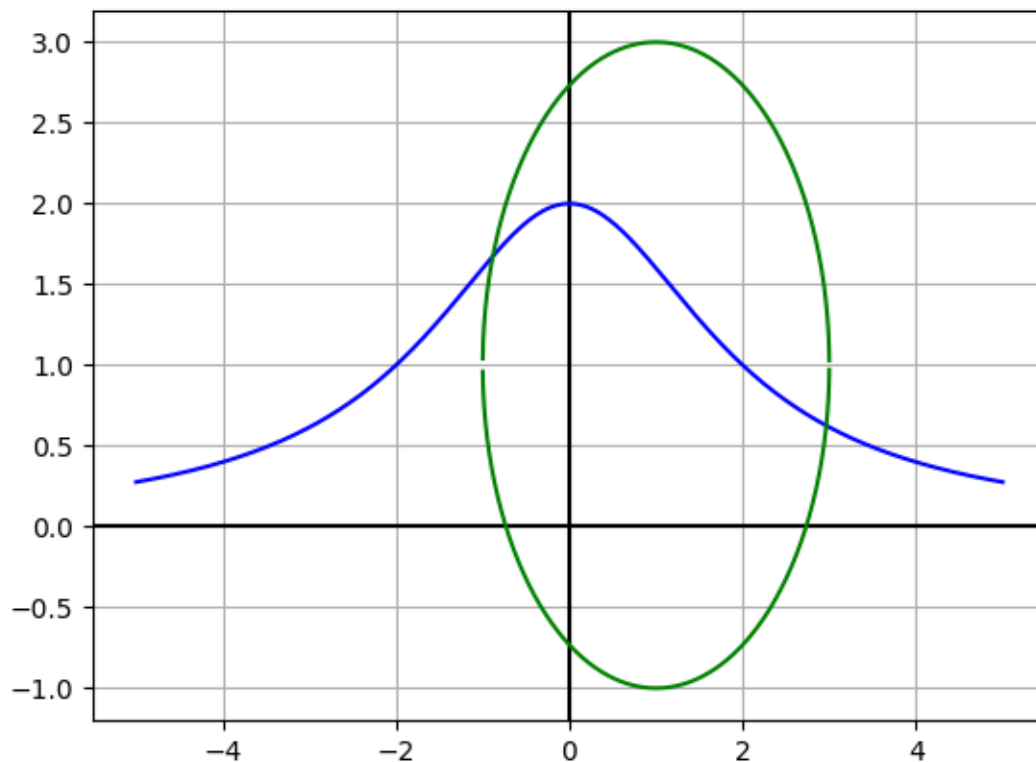




Метод ньютона.



**1.2** Метод ньютона и простой итерации для системы уравнений.  
Начальное значение определим графически.



Входные данные: Эпсилон, файл для вывода, файл для логирования.

Выходные данные: Конечное значение и кол-во итераций для обоих методов в файле output. И в файле log значения на каждой итерации.

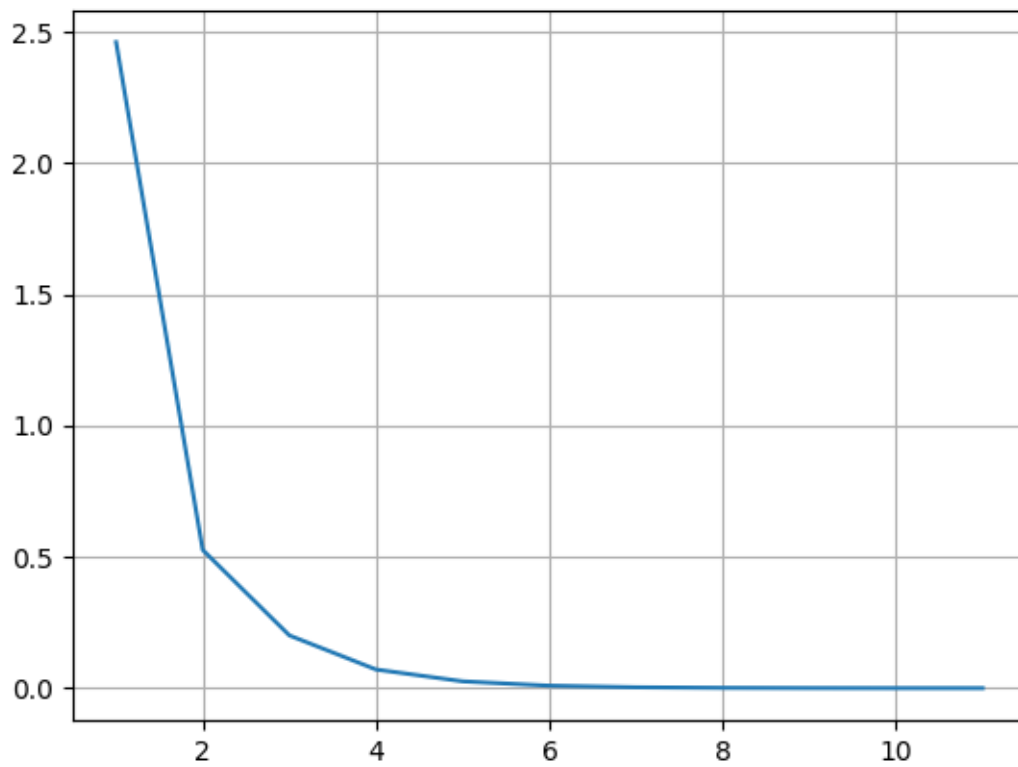
```
art@mars:~/study/NM/lab_2/p_2$ python3 Lw2_2.py --eps 0.0001 --output output
--log log
art@mars:~/study/NM/lab_2/p_2$ cat output
EPS = 0.0001
Iter: x = [2.96463486 0.62553655]
Steps = 11
Newton: x = [2.96463138 0.62553565]
Steps = 4
art@mars:~/study/NM/lab_2/p_2$ cat log
Iter:
x0 = [2.5,0.5]
x1 = [3.04887218 0.64661654]
```

```

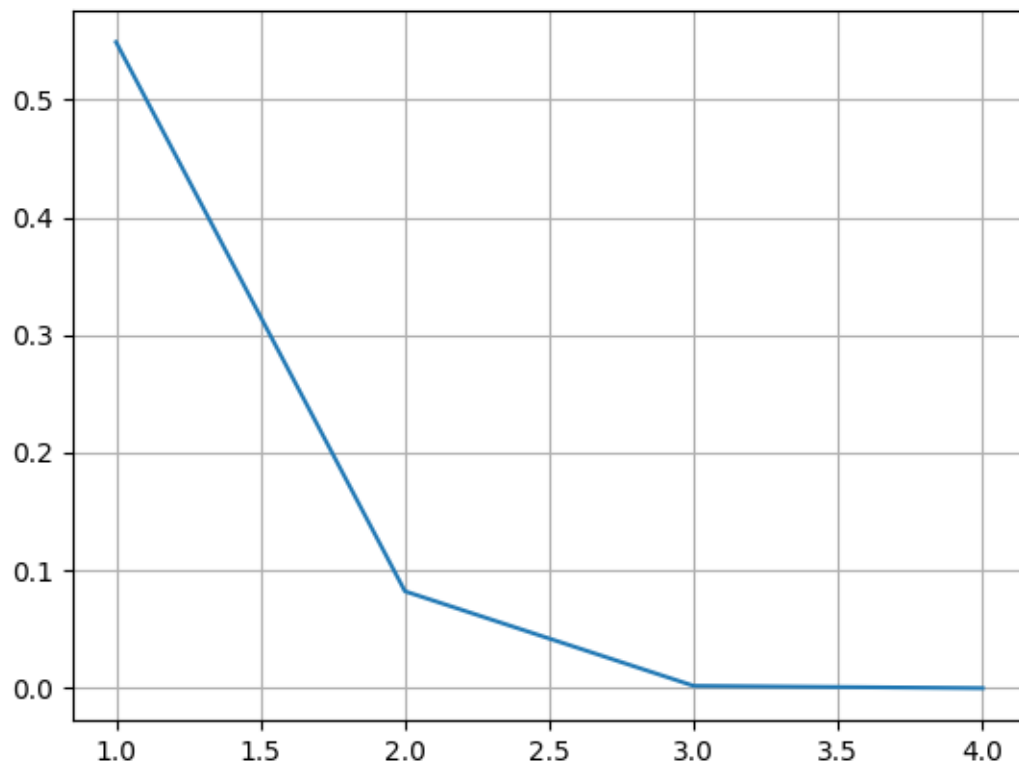
x2 = [2.93141567 0.6170041 ]
x3 = [2.97623247 0.62850686]
x4 = [2.96037651 0.62444091]
x5 = [2.96616616 0.62593055]
x6 = [2.96407427 0.62539219]
x7 = [2.96483316 0.62558762]
x8 = [2.96455823 0.62551681]
x9 = [2.96465789 0.62554248]
x10 = [2.96462177 0.62553318]
x11 = [2.96463486 0.62553655]
Newton's:
x0 = [2.5,0.5]
x1 = [3.04887218 0.64661654]
x2 = [2.96657052 0.62610893]
x3 = [2.96463249 0.62553603]
x4 = [2.96463138 0.62553565]
q = 0.8176691729323308
Lambda:
[[ 0.03007519  0.30827068]
 [ 0.09022556 -0.07518797]]

```

Зависимость погрешности от кол-ва итераций.  
Метод простых итераций.



Метод ньютона.



## 4 Выводы

Благодаря этой лабораторной работе, я узнал, что можно быстро с помощью компьютера находить решения нелинейных уравнений и систем нелинейных уравнений методом Ньютона и простой итерации.

## Список литературы

- [1] *Численные методы. Учебник*  
*Пирумов Ульян Гайкович, Гидаспов Владимир Юрьевич*  
(ISBN 978-5-534-03141-6)