Вычмат лаба6

“Численное решение нелинейных уравнений”

Вариант 19

Б9122-02.03.01сцт 2 группа

Цель

1) Определить примерный диапазон, где может находиться корень

2) Численно найти решение с помощью трёх методов и привести таблицу, отражающую сходимость метода

3) Сделать вывод

Ход работы

▼ 1) Определение области рассмотрения

Дано уравнение $e^{-x} + x^3 - 3 = 0$

Задача сводится по факту к определению, где равные значения принимают экспонента и полином третьей степени

Экспонента с осью OX не пересекается, следовательно для оценки проверим корень полинома. Его крайне легко найти, это будет $x = \sqrt[3]{3}$, что примерно 1.44

Полином третьей степени довольно быстро возрастает, а экспонента быстро убывает в положительной своей части, следовательно положительный корень уравнения находится не многим дальше от 1.44

Значит возьмём пока что промежуток [0, 2]

На самом деле есть ещё один корень уравнения, он находится в отрицательной части. В ней полином третьей степени стремительно улетает в минус бесконечность, однако мы знаем, что показательная функция рано или поздно перегоняет любой полином по модулю. Следовательно где-то существует ещё и отрицательный корень, но его искать я не буду в рамках этой лабы

▼ 2.a) Метод хорд

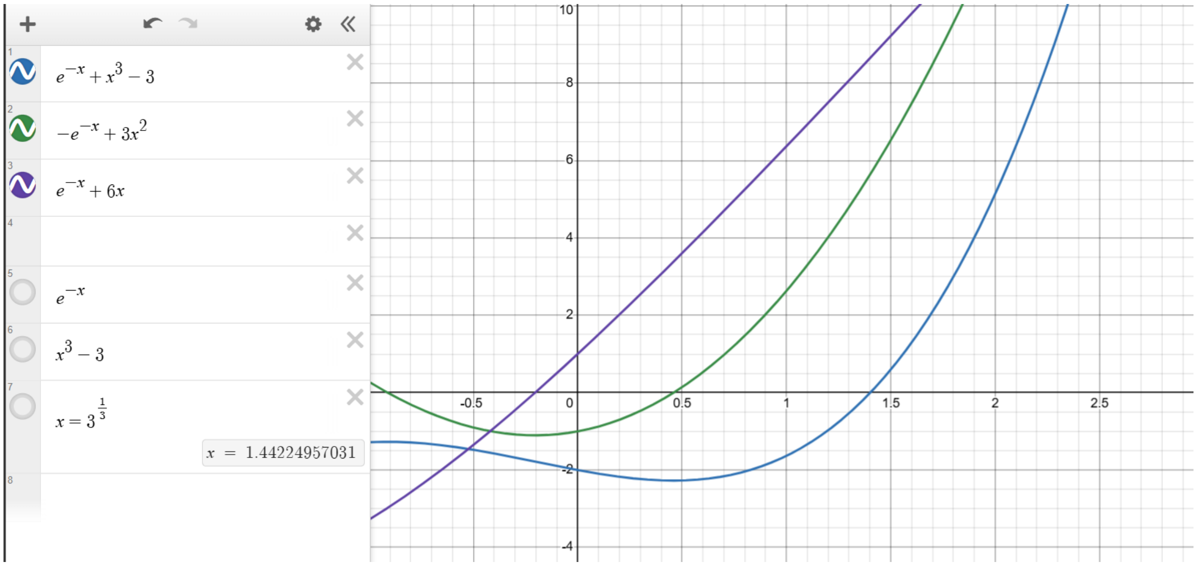
С областью рассмотрения разобрались, пора её сузить

Выберем отрезок [a, b], удовлетворяющий условиям сходимости метода:

1) На концах промежутка функция имеет разный знак

2) Производная знак не меняет

Используя замечательнейшие технологии нашего времени, можно увидеть, что точка ноль нам не подходит:



Несложно это проверить и без графиков: просто подставляем 0 в производную и получаем -1

Под условия подходит например 0.5, возьму это значение как начало отрезка

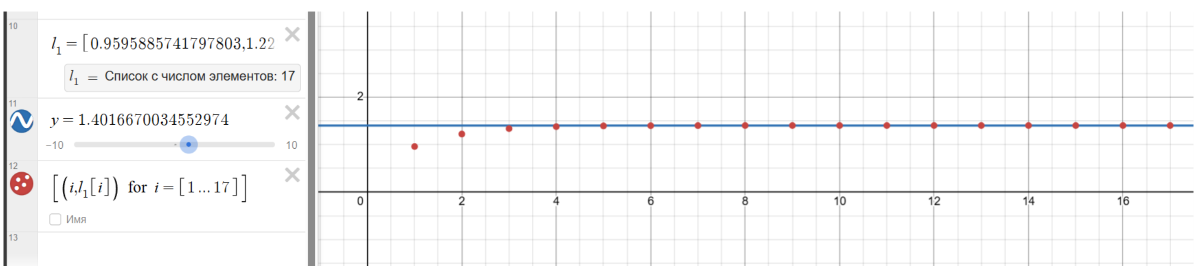
Код самого метода:

```python
import numpy as np

def f(x): return np.exp(-x) + x**3 - 3


def chord_method(eps, lst: list):
    counter = 1
    b = lst[1]
    xprev = lst[0]
    xnext = xprev - (b-xprev)*f(xprev) / (f(b) - f(xprev))
    print(counter, xnext)

    while abs(xnext - xprev) > eps:
        xprev = xnext
        xnext = xprev - (b-xprev)*f(xprev) / (f(b) - f(xprev))
        counter += 1
        print(counter, xnext)

    return xnext


chord_method(0.0000001, [0.5, 2])
```
✓ 0.0s

Таблица значений:

```
1 0.9595885741797803
2 1.2221421074581298
3 1.3359339753248611
4 1.378625141767487
5 1.3937192299042407
6 1.3989411176058297
7 1.4007339323901347
8 1.4013478366163927
9 1.401557863022297
10 1.4016296942056725
11 1.4016542586107394
12 1.4016626586977485
13 1.4016655311707253
14 1.4016665134303286
15 1.4016668493194882
16 1.4016669641786061
17 1.4016670034552974
```

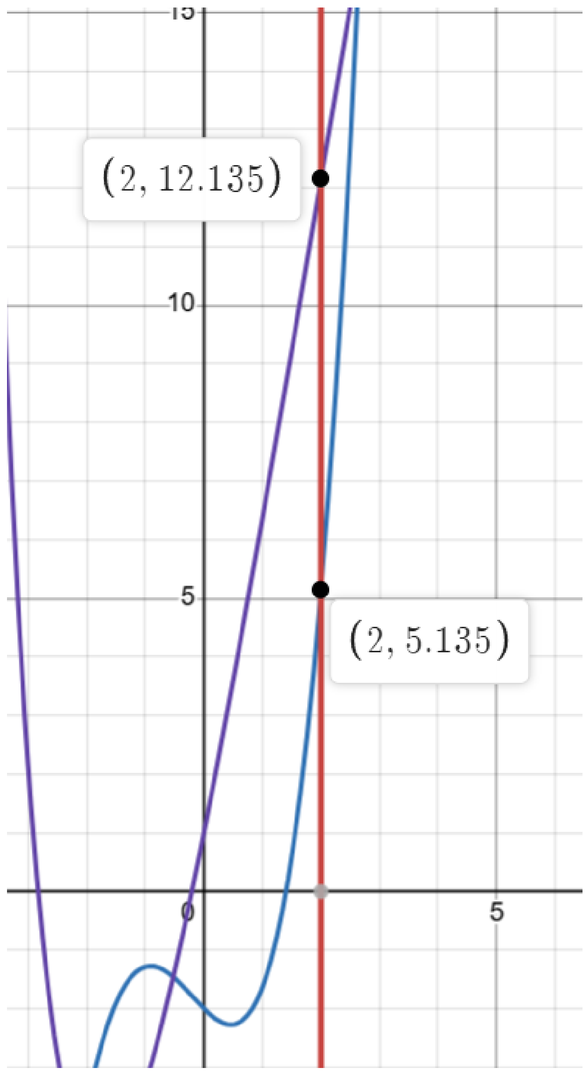
График сходимости:



Видно что по факту метод сошелся ещё на 5-ой итерации, но вычисления продолжились по причине точности до одной миллионной, которую я указал в коде, потому точное значение было достигнуто за 17 итераций

▼ 2.b) Метод Ньютона

Для метода Ньютона достаточно одной точки, в которой знаки исходной функции и её второй производной равны. Точка 2 отлично подойдёт в качестве начального приближения:



Код:

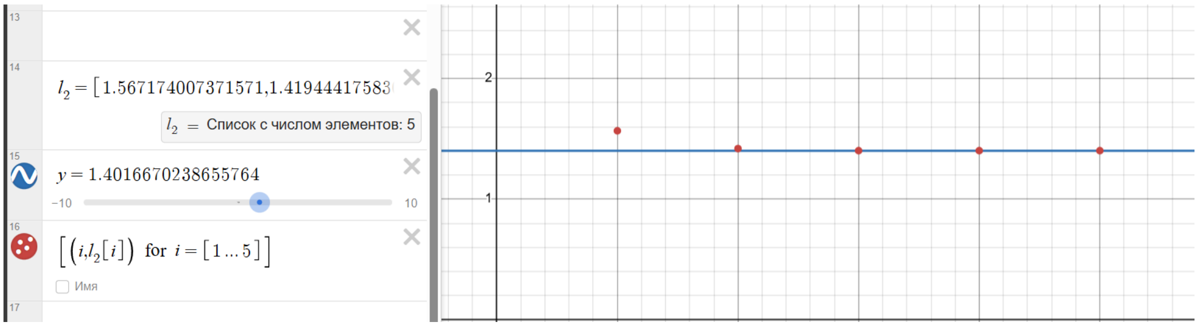
```
def Newton_method(eps, x0):
    counter = 1
    xprev = x0
    xnext = xprev - f(xprev) / df(xprev)
    print(counter, xnext)

    while abs(xnext - xprev) > eps:
        counter += 1
        xprev = xnext
        xnext = xprev - f(xprev) / df(xprev)
        print(counter, xnext)

    return xnext

Newton_method(0.0000001, 2)
```

✓ 0.0s

Таблица значений:

```
1 1.567174007371571
2 1.4194441758304381
3 1.4019046017324523
4 1.4016670671084956
5 1.4016670238655764
```

График сходимости:



Даже при такой большой выставленной точности метод Ньютона сошелся всего за 5 итераций

2.c) Метод бисекции

Из ограничений здесь нам важно только различие знака на концах интервала

В предыдущих пунктах не раз было показан сам график функции, потому я просто возьму отрезок [0.5, 2]

Код:

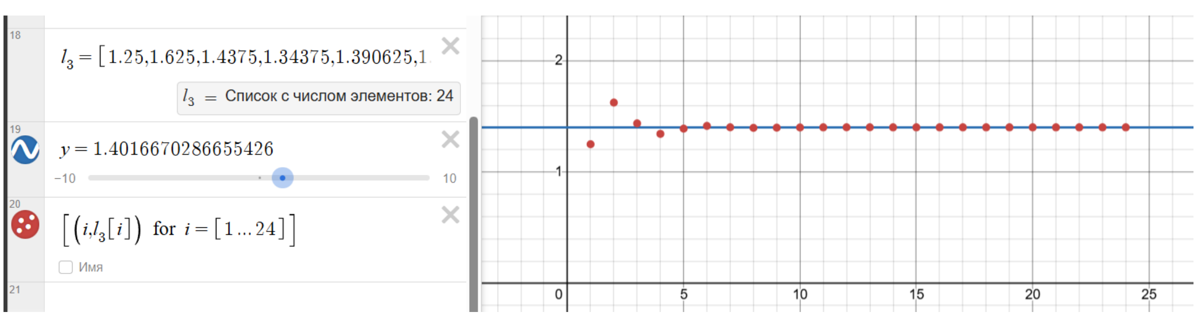
```python
def bisection_method(eps, lst: list):
    l = lst[0]
    r = lst[1]
    c = 0
    counter = 1

    xprev = r
    xnext = c

    while abs(xnext - xprev) > eps:
        c = (r + l)/2

        if f(c)*f(l) > 0: l = c
        elif f(c)*f(r) > 0: r = c

        xprev = xnext
        xnext = c

        print(counter, xnext)
        counter += 1

    return xnext

bisection_method(0.0000001, [0.5, 2])
```

✓ 0.0s

Таблица значений:

```
1 1.25
2 1.625
3 1.4375
4 1.34375
5 1.390625
6 1.4140625
7 1.40234375
8 1.396484375
9 1.3994140625
10 1.40087890625
11 1.401611328125
12 1.4019775390625
13 1.40179443359375
14 1.401702880859375
15 1.4016571044921875
16 1.4016799926757812
17 1.4016685485839844
18 1.401662826538086
19 1.4016656875610352
20 1.4016671180725098
21 1.4016664028167725
22 1.4016667604446411
23 1.4016669392585754
24 1.4016670286655426
```

График сходимости:



На моё удивление, метод дихотомии сошелся не за 50 итераций, а всего за 24

▼ 3) Вывод

Из проведенных мною вычислений следует, что метод Ньютона сходится быстрее всех: всего 5 итераций при точности в 1 миллионная

На втором месте - метод хорд, который сошелся за 17 итераций
И на третьем месте - метод бисекции, не далеко ушедший от второго места: 24 итерации

не знаю зачем, но ссылочка на десмос:

Desmos | Graphing Calculator

Explore math with our beautiful, free online graphing calculator. Graph functions, plot points, visualize algebraic equations, add sliders, animate graphs, and more.

https://www.desmos.com/calculator/kbacrgvkxl