# Лабораторная работа №1
Корехов (М32031), Руковишников (М32001)

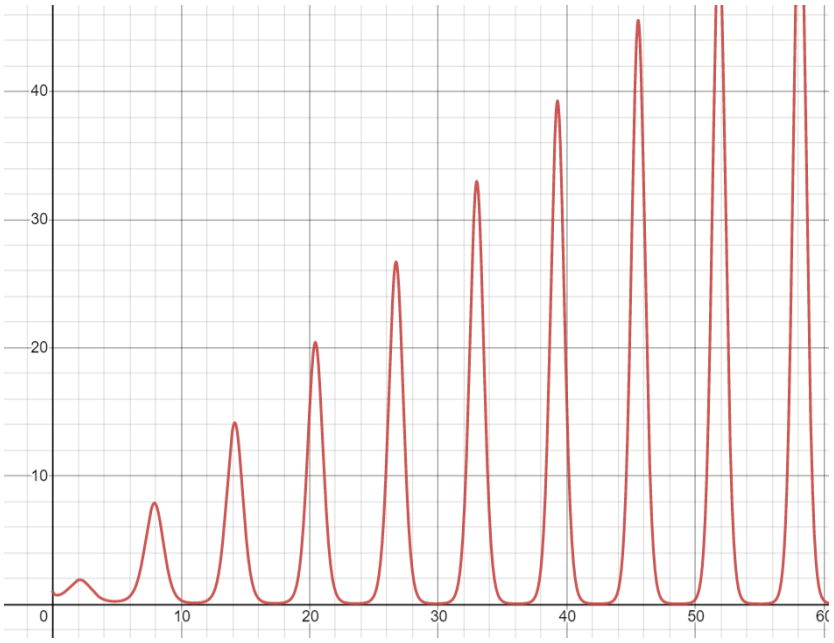
## Ссылка на реализацию:
https://github.com/kroexov/applied-math-2022

## Теоретическая база:

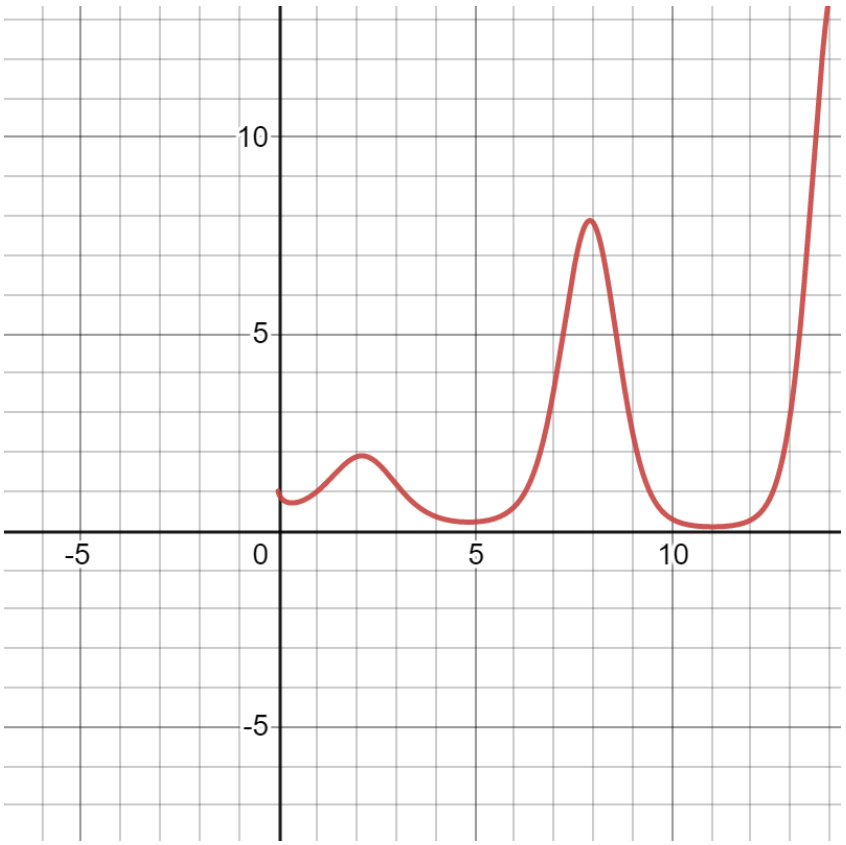
Дана функция: $y(x) = e^{sinx \cdot lnx}$

Преобразуем к виду $y(x) = x^{sinx}$

Её график:




И ближе:




Видно, что функция имеет множество монотонных промежутков возрастания и убывания. Поскольку используемые методы работают только для унимодальных функций, мы выбрали отрезок, на котором она унимодальна. Исследовав функцию на экстремумы, мы выяснили, что отрезок [0;2] подходит по заданному условию. Реальный минимум функции находится в точке x ≈ 0.3522153992.

**Мы провели тесты с EPS от 10^-1, до 10^-8**

**Таблица зависимости количества итераций/вызовов функци от EPS:**

| Зависимость итераций и вызовов функции от EPS | | | | |
|---|---|---|---|---|
| **EPS** | **Количество итераций** | **Количество вызовов функции** | **Среднее изменение отрезка** | **Метод** |
| 10^-1 | 6 | 12 | 1,720835158 | Дихотомия |
| | 8 | 9 | 1.618033988749895 | Золотое сечение |
| | 5 | 7 | 1.643703703703704 | Фибоначчи |
| | 1 | 8 | 4.0 | Парабол |
| | 5 | 8 | 2.618033988749895 | Брента |
| 10^-2 | 9 | 18 | 1.8354064543326598 | Дихотомия |
| | 13 | 14 | 1.618033988749896 | Золотое сечение |

| | | | | |
|---|---|---|---|---|
| | 10 | 12 | 1.6484185337667057 | Фибоначчи |
| | 1 | 8 | 4.0 | Парабол |
| | 10 | 13 | 2.618033988749894 | Брента |
| 10^-4 | 16 | 32 | 1.8922643436094084 | Дихотомия |
| | 22 | 23 | 1.6180339887499107 | Золотое сечение |
| | 19 | 21 | 1.6338020792534678 | Фибоначчи |
| | 9 | 16 | 1.4479711362327734 | Парабол |
| | 20 | 23 | 2.6180339887498296 | Брента |
| 10^-6 | 22 | 44 | 1.9406799692649102 | Дихотомия |
| | 32 | 33 | 1.6180339887510267 | Золотое сечение |
| | 29 | 31 | 1.6283643690550407 | Фибоначчи |
| | 15 | 22 | 1.2688559622030877 | Парабол |
| | 29 | 32 | 2.618033988750442 | Брента |
| 10^-8 | 27 | 54 | 2,17 | Дихотомия |
| | 41 | 42 | 1,618033989 | Золотое сечение |
| | 38 | 40 | 1,6259177 | Фибоначчи |
| | 21 | 28 | 1,192040467 | Парабол |
| | 39 | 42 | 2,618033989 | Брента |

**На EPS, равном 10^-8, как на EPS с наибольшим количеством итераций, мы замерили изменение длины отрезка с каждой последующей итерацией, а в таблицу выше включили средние изменения длины отрезка**

| Итерация | Dichotomy | Golden Ratio | Fibonacci | Parabolas | Brents |
|---|---|---|---|---|---|
| 1 | 2,00 | 1,618033989 | 1,618033989 | 4 | 2,618033989 |
| 2 | 1,99999999 | 1,618033989 | 1,618033989 | 1,8581745 | 2,618033989 |
| 3 | 1,99999998 | 1,618033989 | 1,618033989 | 1,029419748 | 2,618033989 |
| 4 | 1,99999996 | 1,618033989 | 1,618033989 | 1,101664157 | 2,618033989 |
| 5 | 1,99999992 | 1,618033989 | 1,618033989 | 1,016755623 | 2,618033989 |
| 6 | 1,99999984 | 1,618033989 | 1,618033989 | 1,016685405 | 2,618033989 |
| 7 | 1,99999968 | 1,618033989 | 1,618033989 | 1,004785628 | 2,618033989 |
| 8 | 1,99999936 | 1,618033989 | 1,618033989 | 1,003108503 | 2,618033989 |
| 9 | 1,99999872 | 1,618033989 | 1,618033989 | 1,001146663 | 2,618033989 |
| 10 | 1,99999744 | 1,618033989 | 1,618033989 | 1,00061931 | 2,618033989 |
| 11 | 1,99999488 | 1,618033989 | 1,618033989 | 1,000257215 | 2,618033989 |
| 12 | 1,99998976 | 1,618033989 | 1,618033989 | 1,000127716 | 2,618033989 |
| 13 | 1,99997952 | 1,618033989 | 1,618033989 | 1,000056096 | 2,618033989 |
| 14 | 1,999959042 | 1,618033989 | 1,618033989 | 1,000026789 | 2,618033989 |
| 15 | 1,999918087 | 1,618033989 | 1,618033989 | 1,00001208 | 2,618033989 |
| 16 | 1,999836187 | 1,618033989 | 1,618033989 | 1,000005665 | 2,618033989 |

| | | | | | |
|---|---|---|---|---|---|
| 17 | 1,999672427 | 1,618033989 | 1,618033988 | 1,000002586 | 2,618033989 |
| 18 | 1,999345069 | 1,618033989 | 1,61803399 | 1,000001203 | 2,618033989 |
| 19 | 1,998690996 | 1,618033989 | 1,618033985 | 1,000000552 | 2,618033989 |
| 20 | 1,997385414 | 1,618033989 | 1,618033999 | 1,000000256 | 2,618033989 |
| 21 | 1,994784464 | 1,618033989 | 1,618033963 | 1,000000119 | 2,618033989 |
| 22 | 1,98962305 | 1,618033989 | 1,618034056 | | 2,618033989 |
| 23 | 1,979459251 | 1,618033989 | 1,618033813 | | 2,618033989 |
| 24 | 1,959745362 | 1,618033989 | 1,618034448 | | 2,618033989 |
| 25 | 1,922606185 | 1,618033989 | 1,618032787 | | 2,618033989 |
| 26 | 1,856331426 | 1,618033989 | 1,618037135 | | 2,618033989 |
| 27 | 6,960464503 | 1,618033989 | 1,618025751 | | 2,618033989 |
| 28 | | 1,618033989 | 1,618055556 | | 2,618033989 |
| 29 | | 1,618033989 | 1,617977528 | | 2,618033989 |
| 30 | | 1,618033989 | 1,618181818 | | 2,618033989 |
| 31 | | 1,618033989 | 1,617647059 | | 2,618033989 |
| 32 | | 1,618033989 | 1,619047619 | | 2,618033988 |
| 33 | | 1,618033989 | 1,615384616 | | 2,618033989 |
| 34 | | 1,618033989 | 1,625 | | 2,61803399 |
| 35 | | 1,618033988 | 1,6 | | 2,618033989 |
| 36 | | 1,61803399 | 1,666666666 | | 2,618033988 |
| 37 | | 1,618033988 | 1,499999999 | | 2,618033991 |
| 38 | | 1,618033991 | 1,999999997 | | 2,618033984 |
| 39 | | 1,618033984 | | | 2,618034001 |
| 40 | | 1,618033991 | | | |

## Описание алгоритмов:

### Дихотомия

Суть метода заключается в том, что мы берем точку с координатой $x = \frac{b-a}{2}$, где a и b - крайние точки, затем выбираем коэффициент $\sigma = \frac{eps}{4}$, который в несколько раз меньше допустимой погрешности. Затем мы берем 2 точки, $x_1 = x + \sigma$, $x_2 = x - \sigma$, а затем сокращаем изначальный отрезок до одной из этих точек, т.е. вдвое.

Во время выполнения данного алгоритма мы берем середину отрезка и точки на минимальном расстоянии (меньше EPS) справа и слева, после чего проверяем, значение функции в какой из точек больше, и в зависимости от этого сокращаем отрезок вдвое.
Сложность алгоритма равна `ln((a0-b0)/Eps)/ln(2)` по количеству итераций и вдвое больше по количеству вызовов функции (на каждой итерации мы дважды вызываем функцию для правой и левой точки)

Алгоритм на python:

```
def dichotomy(a, b, eps):
    # complexity: ln((a0-b0)/Eps)/ln(2)
    relation = 0
    sigma = eps / 4
    iterations = 0
    while b - a > eps:
        iterations += 1
        middle = (b + a) / 2
        left = middle - sigma
        right = middle + sigma
        func_left = func(left)
        func_right = func(right)
        if func_left > func_right:
            relation += ((b - a) / (b - left))
            a = left
        elif func_left < func_right:
            relation += ((b - a) / (right - a))
            b = right
        else:
            relation += ((b - a) / (right - left))
            (a, b) = (left, right)
    print("average relation", relation / iterations)
    print("num of iterations is", iterations)
    print("num of function calls is", iterations * 2)
    return [a, b]
```

**Метод золотого сечения**

Суть метода заключается в том, что мы берем 2 точки с координатами $x_1 = 2 / (3 + \sqrt{5})$ , $x_2 = 2 / (1 + \sqrt{5})$, по правилам золотого сечения, а затем сокращаем отрезок до одной из этих точек, т.е. примерно в 1.618 раз, при этом вторая точка остается валидной на следующей итерации, т.е. нам нужно дополнительно вычислять не две, а только одну точку.

Во время выполнения данного алгоритма мы вычисляем в соответствии с правилом золотого сечения две точки внутри отрезка, затем сокращаем отрезок в зависимости от их положения, и внутри нового отрезка берем уже только одну новую точку, т.к. по правилам золотого сечения одна из двух старых точек остается валидной, что позволяет вдвое уменьшить количество вызовов функции на каждой итерации, а значит, и общую сложность алгоритма.
Сложность алгоритма равна `ln((a0-b0)/Eps)/ln(2 / (3 + math.sqrt(5)))` , или примерно `ln((a0-b0)/Eps)/ln(1.62)` по количеству итераций и примерно столько же по количеству вызовов функции (на первой итерации нам нужно дважды вызвать функцию для определения двух стартовых точек)

Алгоритм на python:

```
def golden_ratio(a, b, eps):
    # complexity: more than in dichotomy, but only one func() call
per iteration
    gr_for_left = 2 / (3 + math.sqrt(5))
    gr_for_right = 2 / (1 + math.sqrt(5))
    left = a + gr_for_left * (b - a)
    right = a + gr_for_right * (b - a)
    func_left = func(left)
    func_right = func(right)
    # in two previous lines we count func from left and right
golden ratios, so we will reduce counting time in the
    # future iterations, because we will count it only once
    iterations = 0
    relation = 0
    while b - a > eps:
        iterations += 1
        if func_left > func_right:
            relation += ((b - a) / (b - left))
            (a, left, func_left) = (left, right, func_right)
            right = a + gr_for_right * (b - a)
            func_right = func(right)
        else:
            relation += ((b - a) / (right - a))
            (b, right, func_right) = (right, left, func_left)
            left = a + gr_for_left * (b - a)
            func_left = func(left)
    print("average relation", relation / iterations)
    print("num of iterations is", iterations + 1)
    print("num of function calls is", iterations + 2)
    return [a, b]
```

**Метод Фибоначчи**

Суть метода заключается в том, что мы берем 2 точки с координатами $x_1 = (fib(n - 1) / fib(n + 1))$ , $x_2 = (fib(n) / fib(n + 1))$, по правилам чисел Фибоначчи, а затем сокращаем отрезок до одной из этих точек, т.е. примерно в 1.618 раз, при этом вторая точка остается валидной на следующей итерации, т.е. нам нужно дополнительно вычислять не две, а только одну точку.

Данный алгоритм похож на метод золотого сечения по реализации, разница заключается в том, что перед началом выполнения нам нужно посчитать количество итераций, последовательно деля отрезок на числа Фибоначчи, пока его размер не станет меньше EPS. После этого мы переходим к вычислениям, однако теперь мы выбираем отрезок не по правилу золотого сечения, а по отношению двух чисел Фибоначчи.

Сложность алгоритма равна примерно `ln((a0-b0)/Eps)/ln(1.618)` (так как 1.618 - это отношение соседних чисел Фибоначчи) по количеству итераций и примерно столько же по количеству вызовов функции (перед первой итерацией нам нужно дважды вызвать функцию для определения двух стартовых точек)

Алгоритм на python:

```
def fib(n):
    a = 0
    b = 1
    for __ in range(n):
        a, b = b, a + b
    return a

def fibonacci(a, b, eps):
    # complexity: precounted num of iterations | only one func()
call per iteration
    # theoretically, this is the most optimal method for guaranteed
reduce of base segment [a,b]
    # this means that by using this method, we can most optimally
find our final segment and use only one func() call!
    # in this function we will find n of iterations before start,
because we already know rules of fibonacci functions
    n = 1
    relation = 0
    while 1 / fib(n) > eps:
        n += 1
    n = n - 2
    print("num of iterations is", n)
    print("num of function calls is", n + 2)
    left = a + (b - a) * (fib(n - 1) / fib(n + 1))
    right = a + (b - a) * (fib(n) / fib(n + 1))
    func_left = func(left)
    func_right = func(right)
    for i in range(n):
        fib_for_left = (fib(n - i - 1) / fib(n - i + 1))
        fib_for_right = (fib(n - i) / fib(n - i + 1))
        if func_left > func_right:
            relation += ((b - a) / (b - left))
            (a, left, func_left) = (left, right, func_right)
            right = a + (b - a) * fib_for_right
            func_right = func(right)
        else:
            relation += ((b - a) / (right - a))
            (b, right, func_right) = (right, left, func_left)
            left = a + (b - a) * fib_for_left
            func_left = func(left)
    print("average relation", relation / n)
```

```
    return [a, b]
```

**Метод парабол**

Суть метода заключается в том, что на некоторой маленькой окрестности мы можем достаточно аппроксимировать нашу функцию параболой $y = ax^2 + bx + c$ и найти минимум этой параболы по известной формуле $x_{min} = -\frac{b}{2a}$. При этом, поскольку, формула параболы содержит 3 неизвестных коэффициента, нам необходимо 3 различные точки для их нахождения. На первой итерации эти точки выбираются произвольно, а далее мы сужаем отрезок [x1; x3] и выбираем точку x2 из предыдущего значения x2 и найденного минимума параболы.

```
def parabola_min(x, f):
    x1, x2, x3 = x
    f1, f2, f3 = f

    a1 = (f2 - f1) / (x2 - x1)
    a2 = 1 / (x3 - x2) * ((f3 - f1) / (x3 - x1) - (f2 - f1) / (x2 -
x1))
    return 1 / 2 * (x1 + x2 - a1 / a2)


def parabolas(a, b, eps):
    relation = 0
    x_min = -1
    iterations = 0
    (x1, x2, x3, function_calls) = choose_points(a, b)
    (f1, f2, f3) = map(func, [x1, x2, x3])
    function_calls += 3

    d_cur = b - a
    while True:
        x_min_prev = x_min
        x_min = parabola_min([x1, x2, x3], [f1, f2, f3])

        if abs(x_min - x_min_prev) < eps:
            print("num of iterations is", iterations)
            print("num of function calls is", function_calls)
            print("average relation", relation / iterations)
            return x_min

        f_min = func(x_min)
        function_calls += 1

        if x_min > x2 and f_min > f2:
            (x1, x2, x3) = (x1, x2, x_min)
            (f1, f2, f3) = (f1, f2, f_min)
        elif x_min < x2 and f_min < f2:
```

```
            (x1, x2, x3) = (x1, x_min, x2)
            (f1, f2, f3) = (f1, f_min, f2)
        elif x_min > x2 and f_min < f2:
            (x1, x2, x3) = (x2, x_min, x3)
            (f1, f2, f3) = (f2, f_min, f3)
        elif x_min < x2 and f_min > f2:
            (x1, x2, x3) = (x_min, x2, x3)
            (f1, f2, f3) = (f_min, f2, f3)

        d_prev = d_cur
        d_cur = x3 - x1

        relation += (d_prev / d_cur)
        iterations += 1
```

**Метод Брента**

Метод Брента это по сути метод парабол, подстрахованный от неустойчивого поведения методом золотого сечения. На каждом шаге мы имеем 6 точек:
a и b - границы интервала
x - точка с минимальным значением функции
w - точка, со вторым по минимальности значением функции
v - предыдущее значение w
u - текущий минимум параболы

Изначально мы находим минимум параболы по трём точкам x, w, v. Если найденная точка подходит по условиям:

1. Входит в промежуток [a; b]
2. Разница x и u меньше половины предыдущей соответствующей разницы.

То принимаем её. Иначе берём за следующую потенциальную точку минимума золотое сечение большего из отрезков [a; x] и [b; x].

Далее меняем наблюдаемые 6 точек в соответствие с найденным значением точки.

```
def brent(a, b, eps):
    relation = 0
    r = (3 - math.sqrt(5)) / 2
    x = w = v = (a + b) / 2
    f_x, f_w, f_v = map(func, [x, w, v])
    d_cur = d_prev = b - a
    iterations = 0
    function_calls = 3

    while True:
        if max(x - a, b - x) < eps:
            print("num of iterations is", iterations)
            print("num of function calls is", function_calls)
```

```
            print("average relation", relation / iterations)
            return x

        g = d_prev / 2
        d_prev = d_cur

        if pointsDifferent(x, w, v):
            u = parabola_min([x, w, v], [f_x, f_w, f_v])

        if not pointsDifferent(x, w, v) or not (a + eps <= u <= b -
eps and abs(u - x) < g):
            if 2 * x < a + b:
                u = x + r * (b - x)
                d_prev = b - x
            else:
                u = x - r * (x - a)
                d_prev = x - a

        d_cur = abs(u - x)
        relation += (d_prev / d_cur)
        f_u = func(u)
        function_calls += 1

        if f_u < f_x:
            if u < x:
                b = x
            else:
                a = x

            (x, w, v) = (u, x, w)
            (f_x, f_w, f_v) = (f_u, f_x, f_w)
        else:
            if u < x:
                a = u
            else:
                b = u

            if f_u < f_w:
                (w, v) = (u, w)
                (f_w, f_v) = (f_u, f_w)

        iterations += 1
```

**Результат выполнения кода на многомодальной функции ( $f(x) = x^{sin(x)}$ на отрезке [0, 100]):**

========== DICHOTOMY ==========

average relation 1.945408566033278
num of iterations is 28
num of function calls is 56
[4.842557733880091, 4.84255860640912]

========== GOLDEN RATIOS ==========

average relation 1.618033988493678
num of iterations is 40
num of function calls is 41
[67.54775563840792, 67.54775634550987]

========== FIBONACCI METHOD ==========

num of iterations is 29
num of function calls is 31
average relation 1.6283643638528202
[67.54771891798923, 67.54779319735535]

========== PARABOLAS METHOD ==========

num of iterations is 12
num of function calls is 21
average relation 3.5405242006378934
29.854992010631374

========== BRENT'S METHOD ==========

num of iterations is 37
num of function calls is 40
average relation 2.618033988967102
29.85499176949643

**Можно сделать вывод, что каждый алгоритм смог найти локальный минимум и не сломаться (на нашем примере)**