Численное решение нелинейных уравнений

In [3]:

```
import numpy as np
from scipy import linalg as LA
import math
import matplotlib.pyplot as plt
```

Задание 1

Методом деления отрезка пополам найдите корни уравнения $e^{-x} - 1 + \sin(x) = 0$ с точностью 10^{-3}.

In [2]:

```
# точность
eps_1 = 0.001
```

In [3]:

```
# функция
def func_1(x):
    res = np.exp(0 - x) - 1 + np.sin(x)
    return res
```

In [8]:

```
# поиск концов a, b - где f принимает противоположные знаки
# работает именно для данной задачи, так как, очевидно при x<0 значения положительны, а при
def get_contr_a_b(f, a = -1, b = 1):
    #a = - 1   # можно взять любое a<0, так как f(x)>0 для x<0 в нашей задаче
    #b = 1 - тоже модем взять любое положительное приближение
    while (f(a) * f(b) >= 0):
        #a -= 0.1
        b += 0.1
    return a, b
```

In [9]:

```
a0, b0 = get_contr_a_b(func_1)
print("a =", a0, " b =", b0)
```

```
a = -1  b = 2.100000000000001
```

In [10]:

```
def solve_method(f, a, b, tol):
    res = 0
    m = (a + b) / 2
    if (abs(f(b) - f(a)) < tol):
        return m     # вообще можем тут вернуть любую точку отрезка [a,b], я возвращаю сери
    else:
        if (f(a)* f(m) < 0):
            return solve_method(f, a, m, tol)
        if (f(m) * f(b) < 0):
            return solve_method(f, m, b, tol)
```

In [12]:

```
print("Найденное решение: ", solve_method(func_1, a0, b0, eps_1))
```

```
Найденное решение:  2.0765380859375013
```

Подставив это значение, поймем, что это действительно приближенное решение уравнения.

К слову, этот метод очень сильно зависит от начального отрезка (что очевидно, так как мы ищем решение именно внутри него). Так, при другом выборе начального отрезка мы получим совсем другое решение:

In [13]:

```
a01, b01 = get_contr_a_b(func_1, a = 7.7, b = 7.75)
print("a =", a01, " b =", b01)
print("Другое найденное решение: ", solve_method(func_1, a01, b01, eps_1))
```

```
a = 7.7  b = 7.85
Другое найденное решение:  7.83125
```

Как мы видим, решение мы нашли совсем другое.

Задание 2

Определить число отрицательных корней многочлена $f(x) = 32x^3 + 20x^2 - 11x + 3$, указать для одного из них такой отрезок локализации, на котором выполняются условия теоремы о сходимости метода Ньютона, указать нулевое начальное приближение и оценить число итераций необходимых для достижения заданной точности $\epsilon = 10^{-3}$.

In [15]:

```
eps_2 = 0.001
```

In [50]:

```
def func_2(x):
    res = 32 * x**3 + 20 * x**2 - 11 * x + 3
    return res
```

In [16]:

```
def func_2_der1(x):
    return 96 * x**2 + 40 * x - 11

def func_2_der2(x):
    return 8 * (24 * x + 5)

def func_2_der3(x):
    return 192
```

Всего данное уравнение (так как оно степнеи 3) имеет три корня: один из них будет отрицательный (около -1), также будет 2 комплексных корня.

In [22]:

```
# условие сходимости метода Ньютона: |f'(x)|<1
def is_newton_ok(f1, a, b):
    xes = np.linspace(a, b, 10000)
    is_ok = True
    for x in xes:
        if(abs(f1(x)) >= 1):
            is_ok = False
    return is_ok
```

In [38]:

```
is_newton_ok(func_2_der1, -2, -0.1)
```

Out[38]:

```
False
```

In [39]:

```
# ищем, где будет выполняться:
def find_sides(f1, a, b):
    while abs(f1(a) >= 1):
        a = a + 0.005
        #find_sides(f1, a, b)
    while(abs(f1(b)) >= 1):
        b -= 0.005
        #find_sides(f1, a, b)
    return a, b
```

In [40]:

```
aa, bb = find_sides(func_2_der1, -1, -0.5)
print("Отрезок локализации: [", aa, bb, "]")

print(is_newton_ok(func_2_der1, aa, bb))
```

```
Отрезок локализации: [ -0.6149999999999997 -0.5950000000000001 ]
True
```

In [58]:

```
# шаг метода Ньютона: возьму слагаемое только с первой производной
def newton_step(f, f1, f2, f3, x):
    res = x - f(x)/f1(x) #- (f2(x)*f(x)*f(x))/(2*f1(x) **3) - (f2(x)**2 * f(x)**3)/(2*f1(x)
    return res
```

In [59]:

```
def solve_newton(f, f1, f2, f3, x, tol):
    iters = 0
    x_prev = x + 2 * tol # просто чтобы не остановиться на первом же шаге
    while(abs(x - x_prev) > tol):
        x_prev = x
        x = newton_step(f, f1, f2, f3, x)
        iters += 1
    return x, iters
```

In [60]:

```
solution2, num_iters = solve_newton(func_2, func_2_der1, func_2_der2, func_2_der3, -0.614,
```

In [61]:

```
print("Решение: ", solution2)
print('Число итераций:', num_iters)
```

```
Решение:  -1.0414868358533398
Число итераций: 12
```

Итак, за 12 итераций мы сошлись к отрицательному решению с нужной точностью (за начальное приближение я брала 0.0614).

Задание 3

Уравнение $e^x = 3x^2$ среди своих корней имеет корень $x^* = 0.91$. Для нахождения корня предлагается использовать один из следующих методов простой итерации:

3.1 $x_{n+1} = 2\ln(\sqrt(3)x_n)$

3.2 $x_{n+1} = \sqrt{(exp(x_n)/3)}$

3.3 $x_{n+1} = \exp(x_n)/(3x_n)$

3.4 $x_{n+1} = 3\ln(\sqrt(3)x_n) - x_n/2$

Оцените скорость сходимости через оценку производной и оцените число итераций необходимых для достижения требуемой точности $\varepsilon = 10^{-2}$.

In [10]:

```
eps_3 = 0.01
```

In [1]:

```
def func_3(x):
    res = np.exp(x) - 3 * x * x
    return res
```

Рассмотрим методы выше:

3.1) $f(x) = 2\ln(\sqrt{(3)}x)$, тогда $f'(x) = \frac{2}{x}$. У нас есть условие: $|f'(x)| < 1$, то есть $|\frac{2}{x}| < 1$, но $\frac{2}{0.91} = 2.1978$ - в этот диапазон не входит. Значит, этот метод не подходит, смотрим следующий.

3.2) $f(x) = \sqrt{(e^x/3)}$, тогда $f'(x) = \frac{\sqrt{e^x}}{\sqrt{3}}$. У нас есть условие: $|f'(x)| < 1$, то есть $|\frac{\sqrt{e^x}}{\sqrt{3}}| < 1$, тогда $\frac{\sqrt{e^{0.91}}}{\sqrt{3}} = 0.91 < 1$ - в этот диапазон входит. Значит, этот метод можем использовать.

3.3) $f(x) = e^x/(3x)$, тогда $f'(x) = \frac{e^x(x-1)}{3x^2}$. У нас есть условие: $|f'(x)| < 1$, то есть $|\frac{e^x(x-1)}{3x^2}| < 1$, тогда $\frac{e^{0.91}(0.91-1)}{3 \cdot 0.91^2} = -0.09$ - в этот диапазон входит. Значит, этот метод можем использовать.

3.4) $f(x) = 3\ln(\sqrt{(3)}x) - x/2$, тогда $f'(x) = \frac{3}{x} - \frac{1}{2}$. У нас есть условие: $|f'(x)| < 1$, то есть $|\frac{3}{x} - \frac{1}{2}| < 1$, но $\frac{3}{0.91} - \frac{1}{2} = 2.797 > 1$ - в этот диапазон не входит. Значит, этот метод не подходит.

Итак, нам подходят только 2-й и 3-й методы (у них линейная скорость сходимости). Рассмотрим их:

In [7]:

```
def func_met2(x):
    return np.sqrt(np.exp(x) / 3)

def func_met3(x):
    return np.exp(x) / (3 * x)
```

In [13]:

```
def solve_mpi(y_func, method_func, x, tol):
    # y_func: какое уравнение решаем
    # method_func: функция из метода
    # x0: начальное приближение
    iters = 0
    x_prev = x + 2 * tol # просто чтобы сразу не остановиться

    while(abs(y_func(x) - y_func(x_prev)) > tol):
        x_prev = x
        x = method_func(x)
        iters += 1

    return x, iters
```

В качестве начальной точки в обоих случаях возьму 1.

In [16]:

```
ss2, ii2 = solve_mpi(func_3, func_met2, 1, eps_3)
print("Решение при помощи функции метода 2: ", ss2)
print("Сошлись за ", ii2, " шагов.")
```

```
Решение при помощи функции метода 2:  0.9118362136581822
Сошлись за  5  шагов.
```

In [17]:

```
ss3, ii3 = solve_mpi(func_3, func_met3, 1, eps_3)
print("Решение при помощи функции метода 3: ", ss3)
print("Сошлись за ", ii3, " шагов.")
```

```
Решение при помощи функции метода 3:  0.9099751835195202
Сошлись за  3  шагов.
```

При другом начальном приближении сходиться можем дольше:

In [25]:

```
ss4, ii4 = solve_mpi(func_3, func_met3, 2.2, eps_3)
print("Решение при помощи функции метода 2: ", ss4)
print("Сошлись за ", ii4, " шагов.")
```

```
Решение при помощи функции метода 2:  0.9102866821699753
Сошлись за  4  шагов.
```