БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

Лабораторная работа 3

# Метод секущих решения нелинейного уравнения

Вариант 7

**Выполнил:**
Журик Никита Сергеевич
2 курс, 6 группа
**Преподаватель:**
Будник Анатолий Михайлович

Минск, 2019

# Содержание

# 1. Постановка задачи

1. Отделить корень и определить отрезок $[a; b]$.
2. Решить нелинейное уравнение $f(x) = 0$ методом секущих с точностью $\epsilon = 10^{-8}$.
3. Вычислить невязку решения.
4. Проанализировать полученные результаты и сравнить с методом простой итерации и методом Ньютона.

# 2. Алгоритм решения

- Сперва отделим корни уравнения
$$f(x) = 0 \tag{1}$$
при помощи таблицы значений. Таким образом, для каждого корня получим некоторый отрезок, содержащий сам корень, причём на этом отрезке функция в левой части монотонна.

- В отличие от метода Ньютона метод секущих не требует нахождения производной функции $f(x)$ в явном виде. Вместо этого используется приближение к производной, так называемая разделённая разность
$$f'(x)|_{x^k} \approx \frac{f(x^k) - f(x^{k-1})}{x^k - x^{k-1}} \tag{2}$$
Тогда итерационный процесс для нахождения решения может быть построен следующим образом:
$$x^{k+1} = x^k - \frac{(x^k - x^{k-1})}{f(x^k) - f(x^{k-1})} f(x)\big|_{x^k} \tag{3}$$

# 3. Решение конкретного уравнения

- Рассмотрим уравнение $3ln^2x + 6lnx - 5 = 0$ и построим описанный выше итерационный процесс. Указанный итерационный процесс является двухшаговым, поэтому необходимо найти ещё одно приближение к корню перед тем, как начинать итерации.

- Для этого в качестве начальных приближений к корням используем найденные при решении методом простой итерации, а следующие приближения найдём при помощи одной итерации метода Ньютона. Получим:
$$x_1^0 = 0.14134905726406316;$$
$$x_1^1 = 0.011919072400887187;$$
$$x_2^0 = 1.8834601238122997;$$
$$x_2^1 = 1.8832389857682699.$$

# 4. Листинг программы

Для реализации алгоритма был использован Python и библиотеки numpy и matplotlib.

---

```
#Common.py

#!/usr/bin/env python
# coding: utf-8


#Plot the function

import math
import numpy as np
import matplotlib.pyplot as plt

samples = 20
```

```
left_border = 0.01
right_border = 3
delta = (right_border - left_border) / samples
eps = 10 ** -8

def f(x):
    l = math.log(x)
    return 3 * l ** 2 + 6 * l - 5

def f_prime(x):
    l = math.log(x)
    return 6 * (l + 1) / x

def phi(x):
    return math.e ** ((-3 * math.log(x) ** 2 + 5) / 6)

def phi_prime(x):
    return -phi(x) * math.log(x) / x

def calc_f_values(x_values, f):
    f_values = np.zeros(np.shape(x_values))
    for i in range(samples):
        f_values[i] = f(x_values[i])
    return f_values

#Root separation

def separate_roots(interval, f, new_samples):
    global left_border
    left_border = interval[0]
    global right_border
    right_border = interval[1]
    global samples
    samples = new_samples
    x_values = np.linspace(left_border, right_border, samples)
    f_values = calc_f_values(x_values, f)
    intervals = np.empty((1, 2))
    for i in range(samples - 1):
        if (f_values[i + 1] * f_values[i] < 0):
            if (intervals.shape[0] == 1):
                intervals = np.array([x_values[i], x_values[i + 1]])
            else:
                intervals = np.vstack((intervals, [x_values[i], x_values[i + 1]]))
    return intervals

def dichotomy(init_intervals):

    def dichotomy_single_root(interval):
        if (interval[1] - interval[0] < delta):
            return interval
        print("[{}; {}]".format(interval[0], interval[1]))
        center = (interval[0] + interval[1]) / 2
        print(f(interval[0]) * f(center))
        if (f(interval[0]) * f(center) < 0):
            return dichotomy_single_root([interval[0], center])
        else:
            return dichotomy_single_root([center, interval[1]])

    for i in range(len(init_intervals)):
        init_intervals[i] = dichotomy_single_root(init_intervals[i])
    return init_intervals

#Secant.py

#!/usr/bin/env python
# coding: utf-8
```

```python
from Common import *

def get_intervals(left, right, f, samples):
    intervals = separate_roots((left, right), f, samples)
    return intervals

#Solve the equation using secant method

def find_all_roots(intervals, f, epsilon):
    def find_root(interval, f):
        def diff_ratio(old_x, new_x):
            v = (f(new_x) - f(old_x)) / (new_x - old_x)
            return v

        x_left = interval[0]
        x_right = interval[1]
        lam = f(x_left) / f(x_right)
        x_k2 = (x_left - lam * x_right) / (1 - lam)
        x_0 = x_k2
        #old_x = (x_left + x_right) / 2
        x_k1 = x_k2 - f(x_k2) / f_prime(x_k2)
        print("x_1 after one Newton step:", x_k1)
        x_k = x_k1 - f(x_k1) / diff_ratio(x_k2, x_k1)
        iter_num = 2
        while (abs(x_k1 - x_k) >= eps):
            x_k2 = x_k1
            x_k1 = x_k
            x_k = x_k1 - f(x_k1) / diff_ratio(x_k2, x_k1)
            #print(x_k)
            iter_num += 1
        return (x_k1, x_k, iter_num, x_0)

    global eps
    eps = epsilon
    roots = np.array([])
    for interval in intervals:
        (x_k, x_k1, iter_num, x_0) = find_root(interval, f)
        print("Interval: [{}; {}]\nx^0 = {}\nRoot: x* = {}; f(x*) = {}\n|x^(k+1) - x^k| =
            {}\nNumber of iterations: {}\n"
             .format(interval[0], interval[1], x_0, x_k1, f(x_k1), abs(x_k - x_k1), iter_num))
        if len(roots) == 1:
            roots[0] = x_k1
        else:
            roots = np.append(roots, x_k1)
    return roots

if __name__ == "__main__":
    intervals = get_intervals(left_border, right_border, f, samples)
    find_all_roots(intervals, f, eps)
```

## 5. Вывод программы

```
x_1 after one Newton step: 0.011919072400887187
Interval: [0.01; 0.1673684210526316]
x^0 = 0.14134905726406316
Root: x* = 0.07186304228911596; f(x*) = 1.290345608140342e-11
|x^(k+1) - x^k| = 3.5431435979615955e-09
Number of iterations: 10

x_1 after one Newton step: 1.8832389857682699
Interval: [1.7410526315789476; 1.8984210526315792]
x^0 = 1.8834601238122997
```

```
Root: x* = 1.883238990800978; f(x*) = 5.95967719618784e-13
|x^(k+1) - x^k| = 5.032708028096522e-09
Number of iterations: 2
```

## 6. Выводы

- Как следует из результата выполнения программы, метод секущих сошёлся для меньшего корня, как и метод Ньютона, и сделал это за 10 итераций (одна из них по формуле метода Ньютона) против 8 итераций метода Ньютона.

- В окрестности второго корня метод сошёлся за две итерации, что совпадает с числом итераций метода Ньютона.

- Стоит заметить, что невязка как одного решения, так и другого, полученного методом секущих, больше невязки, полученной в методе Ньютона. Число итераций метода секущих также больше, чем число итераций метода Ньютона. Оба результата обусловлены тем, что данный метод имеет порядок сходимости $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$ в отличие от второго порядка сходимости метода Ньютона.

- Ниже приведены результаты лишь для большего корня, так как метод простой итерации не сошёлся к меньшему:

$$\begin{aligned} k_{Secant} &= 2; \\ k_{Newton} &= 2; \\ k_{Simple} &= 24; \\ r_{Secant} &= 5.95967719618784e-13; \\ r_{Newton} &= 0.0; \\ r_{Simple} &= 1.9699690767538414e-08. \end{aligned}$$

  Сравнив результаты, можно отметить, что метод секущих значительно превосходит метод простой итерации как в плане числа итераций, так и в плане невязки решения, однако несколько уступает методу Ньютона в силу меньшей скорости сходимости, что компенсируется более широкой областью применения данного метода. В частности, в случае, когда по каким-либо причинам вычислить значение производной не представляется возможным (слишком трудоёмкая процедура по сравнению с вычислением значения самой функции, функция не является непрерывно дифференцируемой на отрезке отделения корня и т.п.), метод Ньютона плохо применим в отличие от метода секущих, что является существенным преимуществом данного метода.