Университет ИТМО

Факультет программной инженерии и компьютерных технологий

Лабораторная работа №2 по Вычислительной Математике

Выполнил: Богатов Александр Сергеевич

Группа: Р3233

Вариант: 1аб

Преподаватель: Перл Ольга Вячеславовна

Санкт-Петербург

2022

Описание математического метода:

- **Метод Ньютона**

Данный метод является обобщением метода Ньютона решения нелинейных уравнений. Он полезен для нахождения точных корней системы при наличии начального приближения. Система решается итерационном процессом. Корень следующей итерации находится по формуле $x^{k+1} = x^k - J(x^k)^{-1} F(x^k)$, где J(x) – Якобиан или матрица Якоби – матрица частных производных уравнений системы. Другим вариантом поиска корней является поиск приращений $\Delta x_1^{(k)} \dots \Delta x_n^{(k)}$ на основе системы уравнений

$$\begin{cases} f_1(\mathbf{x}^{(k)}) + \dfrac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \dfrac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_2}\Delta x_2^{(k)} + \ldots + \dfrac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \\ f_2(\mathbf{x}^{(k)}) + \dfrac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \dfrac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_2}\Delta x_2^{(k)} + \ldots + \dfrac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \\ \dotfill \\ f_n(\mathbf{x}^{(k)}) + \dfrac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \dfrac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_2}\Delta x_2^{(k)} + \ldots + \dfrac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \end{cases}$$

Метод Ньютона предполагает, что функции дифференцируемы и Якобиан – невырожденная матрица. Как условие окончание итерации используется критерий того, что разница между корнем текущей и прошлой итерации по модулю не превосходит заданную точность. Сходимость метода квадратичная при выборе начального приближения в достаточно малой окресности.

- **Метод половинного деления**

Пусть мы предварительно знаем отрезок, на котором находится корень нелинейного уравнения, функция непрерывна на этом отрезке и принимает на концах отрезка значения различных знаков. Т. е. существует такая точка x на этом отрезке, что функция обращается в ноль.

Тогда можно делить отрезок пополам, получая на каждой итерации потенциальный корень уравнения. Если в найденной точке функция в ноль не обращается, то необходимо найти знаки функции на отрезках [a, x0] и [x0, b], и выбрать для следующей итерации отрезок, подходящий изначальному условию о значениях на концах отрезка.

Критерий окончания – выполнение неравенства $b_n - a_n < 2\varepsilon$. Метод обладает безусловной линейной сходимостью.

- **Метод хорд**

Метод хорд также предполагает заранее известный отрезок, на котором находится корень, однако отрезок делится не в середине, а в точке пересечения хорды с осью X. Под хордой подразумевается отрезок через точки рассматриваемой функции по концам рассматриваемого интервала. Уравнение хорды имеет следующий вид:

$$\frac{x-a}{b-a} = \frac{y-f(a)}{f(b)-f(a)}$$

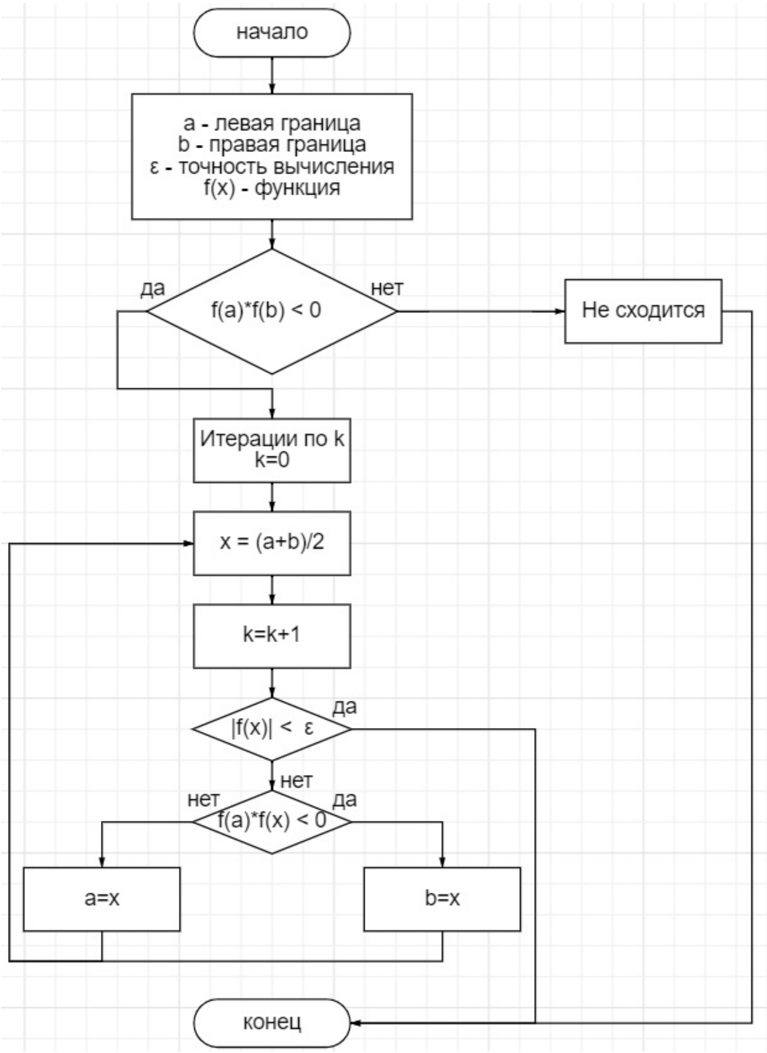
И для точки пересечения с осью X оно переписывается в виде

$$x_0 = a - \frac{f(a)}{f(b)-f(a)} \cdot (b-a)$$

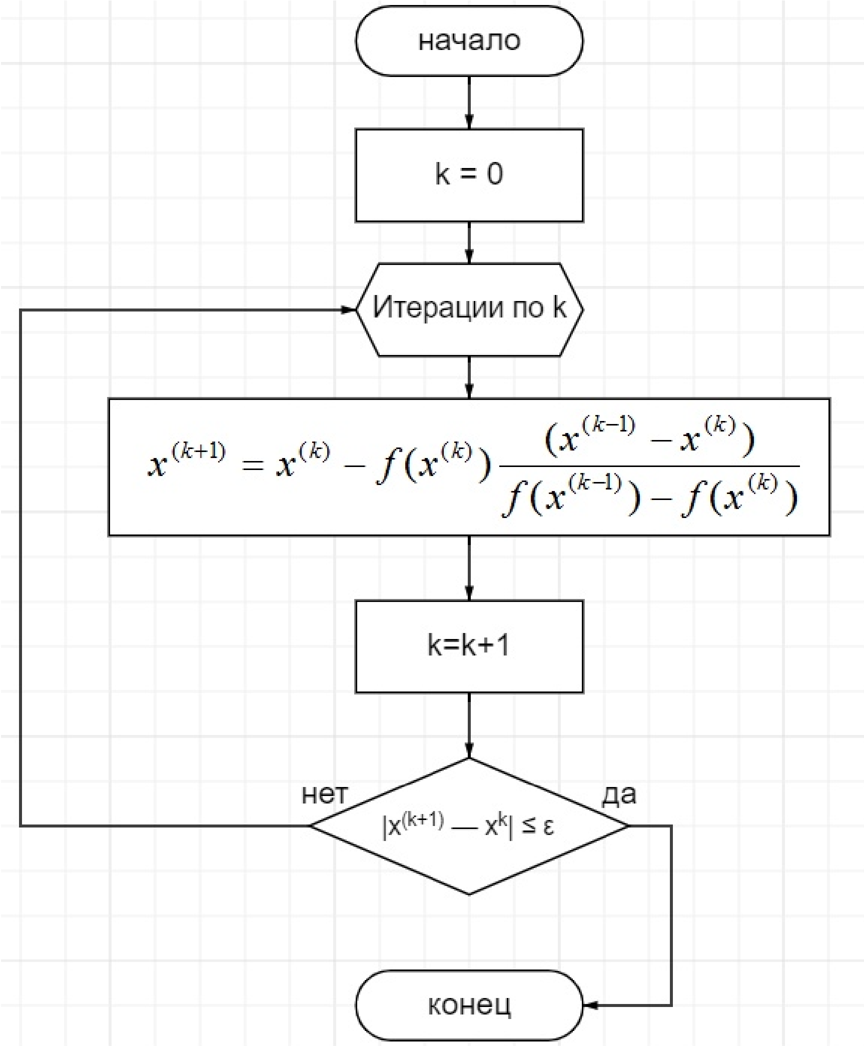
Выбор нового интервала на каждой итерации аналогичен методу бисекции. Итерационный процесс оканчивается, когда разность между корнями на текущей и прошлой итерации по модулю меньше заданной точности. Метод обладает линейной сходимостью.

Блок-схема математического метода:

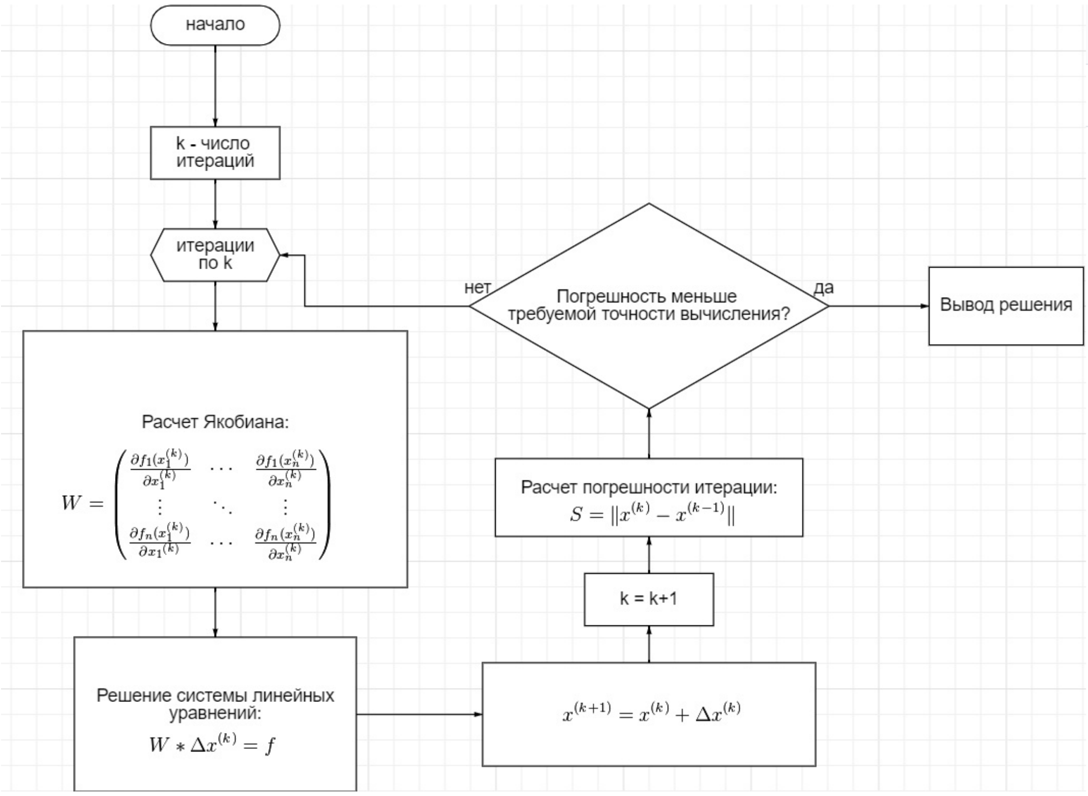
- **Метод половинного деления**



- **Метод хорд**



- **Метод Ньютона**



Листинг программы:

- **Метод половинного деления**

```
public Solution solveByBisection(Function function, double a, double b, double
e) {
            if (function.valueAtX(a)*function.valueAtX(b) > 0) {
                throw new IllegalArgumentException("Function does not fit for
this method");
            } else {
                int iters = 0;
                double variable = 0;
                do {
                    variable = (a+b)/2;
                    if (function.valueAtX(a)*function.valueAtX(variable) > 0)
                        a = variable;
                    else b = variable;
                    iters++;
                } while (Math.abs(b - a) > 2*e && iters <= MAX_ITERS);
                return new Solution(variable, iters);
            }
 }
```

- **Метод хорд**

```
public Solution solveByChord(Function function, double a, double b, double e) {
        double xNext = 0;
        double buffer;
        int iters = 0;
        do {
            buffer = xNext;
            xNext = b - function.valueAtX(b)*(a - b) / (function.valueAtX(a) -
function.valueAtX(b));
            a = b;
            b = buffer;
            iters++;
        } while (Math.abs(xNext - b) > e);
        return new Solution(xNext, iters);
 }
```

- **Метод Ньютона**

```
public SystemSolution solveSystemNewton(double x, double y, Function[]
functions, double e) {
        if (functions.length > 2)
            throw new IllegalArgumentException("Method not applicable!");
        else {
            double delta = 1;
            double xPrev = 0;
            double yPrev = 0;
            int iters = 0;
            do {
                xPrev = x;
                yPrev = y;
                x = xPrev - dX(functions, xPrev, yPrev) /
findJacobian(functions, xPrev, yPrev);
                y = yPrev - dY(functions, xPrev, yPrev) /
findJacobian(functions, xPrev, yPrev);
                if (Math.abs(x - xPrev) > Math.abs(y - yPrev))
                    delta = Math.abs(x - xPrev);
                else delta = Math.abs(y - yPrev);
                iters++;
            } while (delta > e && iters < MAX_ITERS);
            if (iters == MAX_ITERS)
                throw new ArithmeticException("Accuracy not reached");
            else return new SystemSolution(x, y, iters);
        }
    }

public double findJacobian(Function[] functions, double x, double y) {
```

```java
        return functions[0].getDerivativeX(x, y, 1e-9) *
functions[1].getDerivativeY(x, y, 1e-9) - functions[1].getDerivativeX(x, y, 1e-
9) * functions[0].getDerivativeY(x, y, 1e-9);
    }

    public double dX(Function[] functions, double x, double y) {
        return functions[0].valueAtXY(x, y) * functions[1].getDerivativeY(x, y,
1e-9) - functions[1].valueAtXY(x, y) * functions[0].getDerivativeY(x, y, 1e-9);
    }

    public double dY(Function[] functions, double x, double y) {
        return functions[1].valueAtXY(x, y) * functions[0].getDerivativeX(x, y,
1e-9) - functions[0].valueAtXY(x, y) * functions[1].getDerivativeX(x, y, 1e-9);
    }
```

Примеры работы программы:

```
Hello! Please enter 1 if you want to solve a non-linear equation or 2 if you want to solve a system of non-linear equations
2
Choose between these templates:
a*x + b*lg(x) - c*y^2 = 0
d*x^2 - e*x*y - fx + g = 0

a*x^2 + b*x + c*y^2 + d = 0
e*x^2 + f*y + g*y^2 + h = 0
1
Enter coefficients
1
3
-1
2
-1
-5
1
Enter accuracy
0,00001
Enter intital approximations
3,5
2,2
Solution found for x is: 3.4874427876429843
For y: 2.2616286305536475
The solution was found in 3 iterations
```

```
Hello! Please enter 1 if you want to solve a non-linear equation or 2 if you want to solve a system of non-linear equations
1
Choose between these templates:
a*x^3 + b*x + c = 0
a*sin(x)^2 + b*x^2 + c = 0
a*e^(-x) - b*sin(x)^2 = 0
1
Enter coefficients
1
-18
-83
Enter boundaries
2
10
Enter accuracy
0,00001
Solution by bisection:
Solution found is: 5.7051239013671875
The solution was found in 19 iterations
Solution by chord method:
Solution found is: 5.705115482266227
The solution was found in 15 iterations
The difference between results of methods is 8.419100960388448E-6
```

Вывод:

Методы половинного деления и хорд во многом схожи: особенность метода бисекции – в безусловной сходимости, метода хорд – в более быстрой сходимости, что видно по результатам запуска программы.

Главное преимущество метода Ньютона для решения СНАУ в его быстрой сходимости при задании достаточно хорошего начального приближения, однако метод сложен в реализации из-за необходимости вычисления матрицы Якоби.