Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчет

по лабораторной работе №3

по дисциплине «**Вычислительная математика**»

Автор: Баянов Равиль Динарович

Факультет: ПИиКТ

Группа: P3234

Преподаватель: Перл О. В.

Санкт-Петербург, 2024

Оглавление

Описание метода

Метод Ньютона для решения СНУ (систем нелинейных уравнений) – это обычный метод Ньютона для решения уравнений, но работающий с матрицей и векторами. Метод основан на использовании разложения функции в ряд Тейлора окрестности текущего приближения. Метод Ньютона представляет собой итерационный процесс, который осуществляется по определенной формуле следующего вида:

$$(x^{(k+1)} = x^{(k)} - W^{-1}(x^{(k)}) * F(x^{k}), k = 0,1,2,3,\ldots)$$

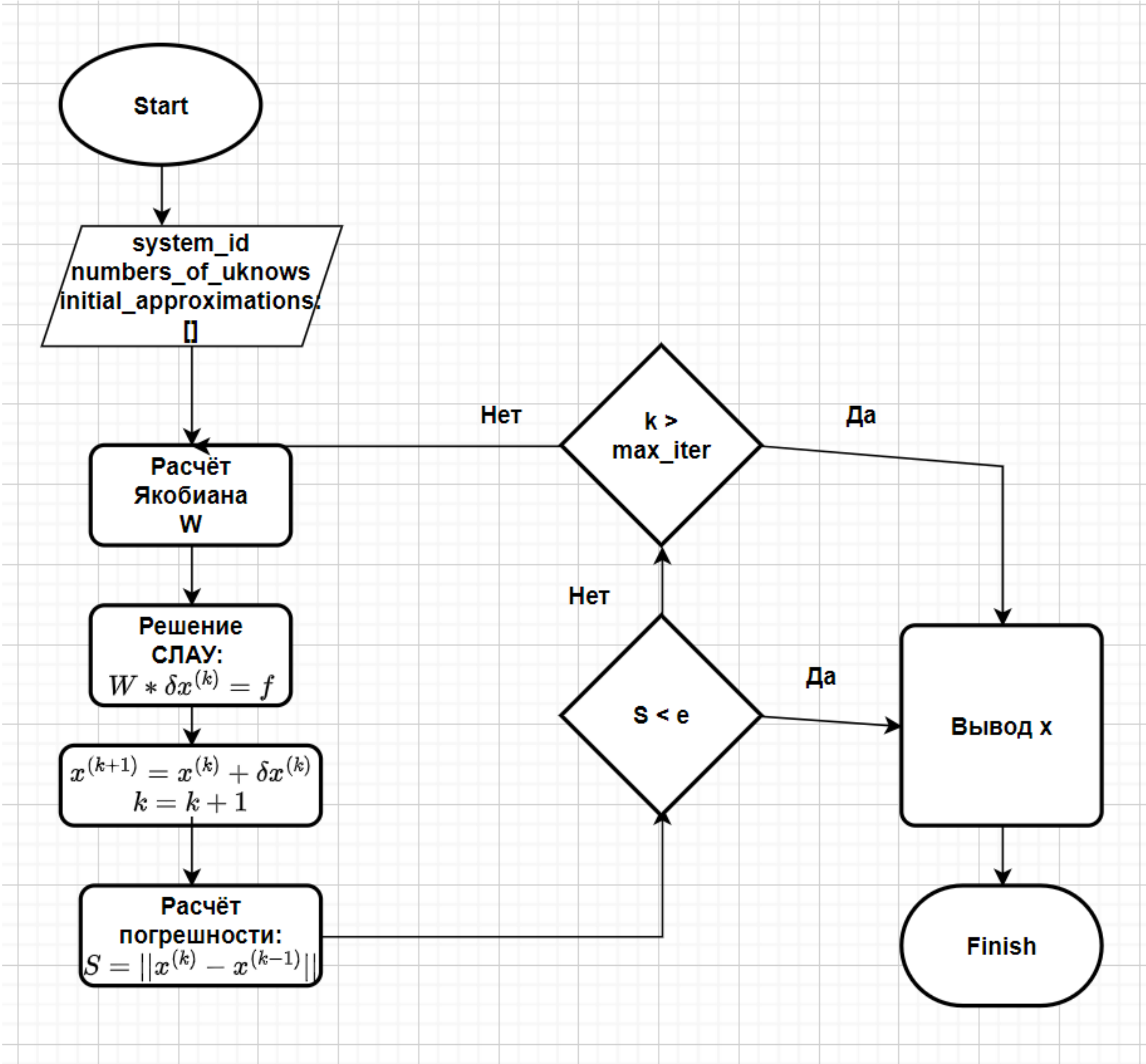
Где W – матрица Якоби, то есть матрица частных производных функций системы по каждой неизвестной.

Критерием окончания итерационного процесса является неравенство:

$$(\left\| x^{(k+1)} - x^{(k)} \right\| \leq \epsilon)$$

Метод Ньютона крайне эффективен, быстр и точен. В дальнейших разделах отчёта можно увидеть реализацию данного метода.

Блок-схема



Start
system_id
numbers_of_uknows
initial_approximations:
[]
Расчёт
Якобиана
W
Решение
СЛАУ:
$W * \delta x^{(k)} = f$
$x^{(k+1)} = x^{(k)} + \delta x^{(k)}$
$k = k + 1$
Расчёт
погрешности:
$S = ||x^{(k)} - x^{(k-1)}||$
Нет
k >
max_iter
Да
Нет
S < e
Да
Вывод x
Finish

Исходный код метода на языке программирования Python

```
def first_derivative_1(args: []) -> float:
    return math.cos(args[0])


def first_derivative_2(args: []) -> float:
    return 1


def second_derivative_1(args: []) -> float:
    return args[1] / 2


def second_derivative_2(args: []) -> float:
    return args[0] / 2


def third_derivative_1(args: []) -> float:
    return (args[1] + k) / pow(math.cos(args[0] * args[1] + k), 2) - 2 *
args[0]


def third_derivative_2(args: []) -> float:
    return (args[0] + k) / pow(math.cos(args[0] * args[1] + k), 2)


def fourth_derivative_1(args: []) -> float:
    return 2 * a * args[0]


def fourth_derivative_2(args: []) -> float:
    return 4 * args[1]


def fifth_deriative_1(args: []) -> float:
    return 2 * args[0]


def fifth_deriative_2(args: []) -> float:
    return 2 * args[1]


def fifth_deriative_3(args: []) -> float:
    return 2 * args[2]


def six_deriative_1(args: []) -> float:
    return 4 * args[0]


def six_deriative_2(args: []) -> float:
    return 2 * args[1]


def six_deriative_3(args: []) -> float:
    return -4
```

```python
def seven_deriative_1(args: []) -> float:
    return 6 * args[0]


def seven_deriative_2(args: []) -> float:
    return -4


def seven_deriative_3(args: []) -> float:
    return 2 * args[2]


def jacobian(n: int, args: []):
    if n == 1:
        return [[first_deriative_1(args), first_deriative_2(args)],
                [second_deriative_1(args), second_deriative_2(args)]]
    elif n == 2 or n == 3:
        return [[third_deriative_1(args), third_deriative_2(args)],
                [fourth_deriative_1(args), fourth_deriative_2(args)]]
    elif n == 4:
        return [[fifth_deriative_1(args), fifth_deriative_2(args),
fifth_deriative_3(args)],
                [six_deriative_1(args), six_deriative_2(args),
six_deriative_3(args)],
                [seven_deriative_1(args), seven_deriative_2(args),
seven_deriative_3(args)]]
    else:
        return [[default_function]]


def solve_by_fixed_point_iterations(system_id, number_of_unknowns,
initial_approximations):
    if system_id > 4:
        return initial_approximations
    eps = 1e-5
    max_iter = 1000
    values = list(initial_approximations)
    funcs = get_functions(system_id)
    it = 0
    for _ in range(max_iter):
        try:
            f_vals = [funcs[i](values) for i in range(number_of_unknowns)]
        except ValueError:
            return values
        J = jacobian(system_id, values)
        J_inv = [[J[j][i] for j in range(len(J))] for i in range(len(J[0]))]
        delta = []
        for i in range(len(J)):
            s = sum(J_inv[i][k] * f_vals[k] for k in range(len(f_vals)))
            delta.append(-s)
        if all(abs(d) < eps for d in delta):
            return values
        for i in range(len(delta)):
            values[i] += delta[i]
    return values
```

Примеры работы программы

1) Первая система:

```
1
2
0.5
0.5
-6.228716899649557e-08
-0.058737839538759135
```

2) Вторая система:

```
2
2
300
400
-6.597680955208146e+246
7.801880431986798e+247
```

3) Вторая система, но с другими параметрами:

```
3
2
300
400
-6.597680955208146e+246
7.801880431986798e+247
```

4) Третья система:

```
4
3
55
66
77
2.2562493965826293e+206
8.397713021660942e+203
2.7884114507001366e+204
```

5) Ввод system_id больше, чем существует систем:

```
5
3
2
2
2
2.0
2.0
2.0
```

Просто выходим из программы с приближёнными значениями.

Вывод

При выполнении данной лабораторной работы я поработал с методом Ньютона для решения СНУ (систем нелинейных уравнений). Попробовав его на нескольких системах, я ощутил его недостатки и преимущества.

Основным преимуществом метода Ньютона является его быстрая сходимость в окрестности точного решения. Если начальное приближение достаточно близко к решению, то метод Ньютона сходится квадратично.

Однако метод Ньютона имеет ряд недостатков. Например, вычисление матрицы Якоби и её обратной матрицы может быть крайне сложным, особенно для систем с большим количеством неизвестных. С этой проблемой помогает справиться модифицированный метод Ньютона, в котором матрицу Якоби вычисляют всего один раз с приближёнными значениями. Но эта модификация проигрывает в точности и в скорости сходимости оригинальному методу. Также метод Ньютона может сходиться очень медленно или вообще может не сходиться, если приближённые значения далеки от решения.

Метод Ньютона имеют быструю сходимость, чем метод хорд или метод простых итераций, но он при этом требует гораздо большего количества вычислений на каждом проходе цикла.

Так как вычисление обратной матрицы требует O(N^3), то логично предположить, что весь метод выполняется за O(N^3). А если быть точнее, то за O(iterations * N^3).

Оценка ошибки метода Ньютона зависит от степени гладкости функции и начального приближения.

В заключение метод Ньютона является эффективным методом решения СНУ, особенно когда начальное приближение находится близко к точному. Но метод Ньютона требует выполнить много вычислений для нахождения ответа.