МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Решение систем нелинейных уравнений

ОТЧЕТ ПО ДИСЦИПЛИНЕ

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧ»

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Сенокосова Владислава Владимировича

Преподаватель

Аспирант ___________ В.М.Шкатов

подпись, дата

Саратов 2024

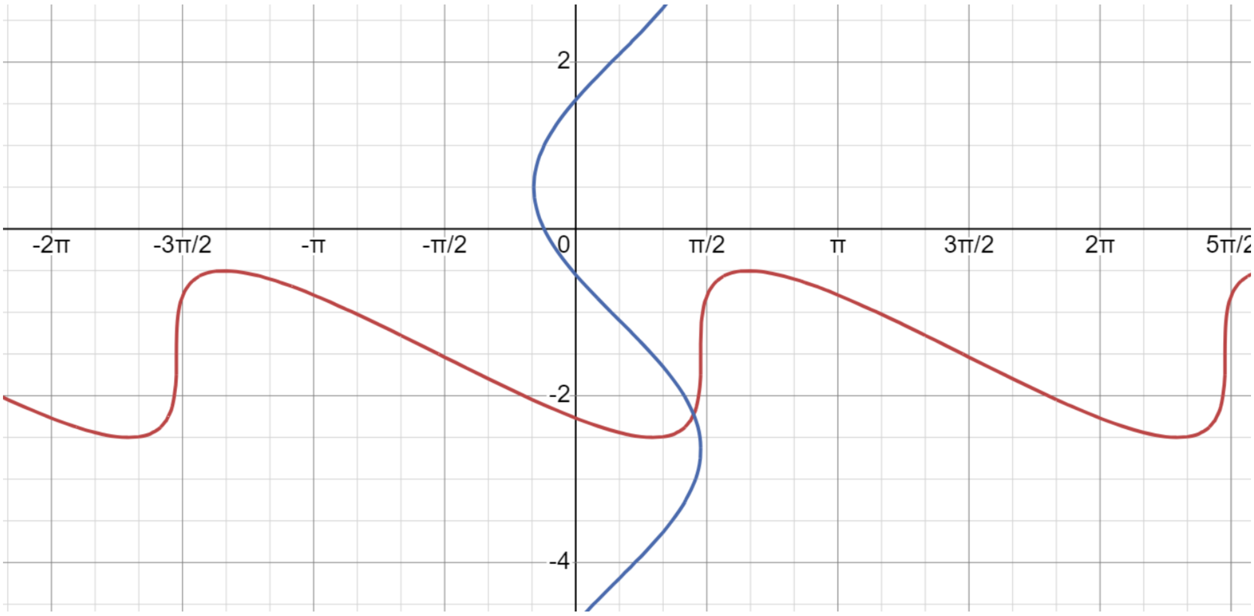
Вариант 19

19	$\sin(x_1 + x_2) - x_2 - 1.5 = 0$ $x_1 + \cos(x_2 - 0.5) - 0.5 = 0$

Задача:

Найти методом Ньютона и наискорейшего спуска все корни системы нелинейных уравнений. При реализации программного кода использовался язык Python с библиотекой sympy, для более удобной работы с линейной алгеброй.

Графическое представление функции имеет следующий вид:



Задание 1: «Метод Ньютона»

Пусть требуется решить систему вида:

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0 \\ ... \\ f_n(x_1, x_2, ..., x_n) = 0 \end{cases} \qquad (4.1)$$

где функции $f_1, f_2, ..., f_n$ – заданные нелинейные вещественнозначные функции n вещественных переменных $x_1, x_2, ..., x_n$.

Обозначим через

$$\bar{x} = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}, \; F(\bar{x}) = \begin{pmatrix} f_1(\bar{x}) \\ f_2(\bar{x}) \\ ... \\ f_n(\bar{x}) \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, ..., x_n) \\ ... \\ f_n(x_1, x_2, ..., x_n) \end{pmatrix}, \; \bar{0} = \begin{pmatrix} 0 \\ ... \\ 0 \end{pmatrix}.$$

Тогда систему (4.1) можно записать в виде

$$F(\bar{x}) = \bar{0}. \qquad (4.2)$$

Обозначим через

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}; \frac{\partial f_1}{\partial x_2}; ...; \frac{\partial f_1}{\partial x_n} \\ ... \\ \frac{\partial f_n}{\partial x_1}; \frac{\partial f_n}{\partial x_2}; ...; \frac{\partial f_n}{\partial x_n} \end{pmatrix}, \qquad (4.3)$$

J – матрица Якоби, якобиан.

Для n-мерного случая итерационный процесс Ньютона:

$$\bar{x}^{(k+1)} = \bar{x}^{(k)} - \left[J(\bar{x}^{(k)})\right]^{-1} F(\bar{x}^{(k)}). \qquad (4.4)$$

Замечание: Если начало приближения выбрано достаточно близко к решению системы, то итерационный процесс (4.4) сходится к этому решению с квадратичной скоростью.

Недостаток: Метод Ньютона достаточно трудоемкий – на каждом шаге итерационного процесса необходимо найти матрицу, обратную якобиану.

Функция, реализующая алгоритм поиска корней Ньютона представлена следующим образом:

```
# Метод Ньютона нахождения корней
def method_newton(params, system, start):
    eps = 0.000001

    # Строим якобиан от параметров x1, x2
    J = []
    for fun in system:
        buff = []
        for param in params:
            buff.append(sp.diff(fun, param))
        J.append(buff)

    # Получаем необходимые матрицы для расчета
    J_x_inv, F_x = get_data_for_newton(J, params, system, start)

    approximate_roots = []

    old_roots = sp.Matrix(deepcopy(start))
    new_roots = old_roots - J_x_inv * F_x

    approximate_roots.append((list(old_roots), None))

    while get_norm_vec(new_roots - old_roots) > eps:

        J_x_inv, F_x = get_data_for_newton(J, params, system, new_roots)

        old_roots = new_roots
        new_roots = old_roots - J_x_inv * F_x

        approximate_roots.append((list(old_roots),
                                  "{:0<20.15f}".format(get_norm_vec(new_roo
ts - old_roots))))

    return new_roots, approximate_roots
```

Для осуществления итерации с промежуточными значениями использовалась следующая функция:

```python
# Получаем необходимые данные для текущей итерации методом Ньютона
def get_data_for_newton(J, params, system, start):
    # Создаем словарь приближенных корней вида {x1: valume1, x2: valume2
...}
    lst_starts = {params[i]: start[i] for i in range(len(params))}

    # Находим значение якбиана в точках x1, x2...
    J_x = []
    for index1 in range(len(J)):
        buff = []
        for index2 in range(len(J)):
            val = J[index1][index2].subs(lst_starts).evalf()
            buff.append(round(val, 7))
        J_x.append(buff)

    # Обратная матрица
    J_x = sp.Matrix(J_x)
    J_x_inv = J_x.inv()

    # Находим значение системы функций в точках x1, x2 ....
    F_x = []
    for i in range(len(system)):
        F_x.append(system[i].subs(lst_starts).evalf())

    F_x = sp.Matrix(F_x)

    return J_x_inv, F_x
```

Результаты работы программы для двух примеров (тестового и варианта 19):

```
Методо Ньютона
-----------------------------------------------------
Из примера: [0.298712300874983, 3.87413939133413]
Из 19 варианта: [1.41423304183789, -2.22440734539974]
-----------------------------------------------------
```

Также в качестве проверки найденных корней была написана следующая функция:

```python
# Функция для проверки найденных корней
def check_roots(funcs_list, lst_roots, params):
    # Связываем параметры x1, x2 со значениями из params
    roots_dict = {params[i]: lst_roots[i] for i in range(len(params))}
    for i, func in enumerate(funcs_list):
        # Если значение = 0, то format не сможет преобразовать такое
значение
```

```
        # поэтому обрабатываем отдельно
        if func.subs(roots_dict) != 0:
            print(f"Значение в функции {i + 1}:",
"{:0<20.15f}".format(func.subs(roots_dict)))
        else:
            print(f"Значение в функции {i + 1}:", func.subs(roots_dict))
```

В результате подстановки найденных корней в систему нелинейных уравнений были получены следующие значения:

```
Проверка корней метода Ньютона
Для системы 1:
Значение в функции 1: -0.000000000000013600
Значение в функции 2: 0.000000000000017000
Для системы 2:
Значение в функции 1: 0
Значение в функции 2: 0
```

Задание 2: «Метод наискорейшего спуска»

Общим недостатком рассмотренных ранее методов является локальный характер сходимости. Когда возникают проблемы с выбором хорошего начального приближения, применяют методы спуска.

Рассмотрим систему:

$$\begin{cases} f(x, y) = 0 \\ g(x, y) = 0 \end{cases}. \tag{4.12}$$

Из функций f и g системы (4.12) образуем новую функцию:

$$\Phi(x,y) = f^2(x,y) + g^2(x,y). \tag{4.13}$$

Так как функция $\Phi(x,y)$ неотрицательная, то $\exists(x^*,y^*)$:

$\Phi(x,y) \ge \Phi(x^*,y^*) \ge 0,\ \forall(x,y)\in R_2$, т. е. $(x^*,y^*) = \arg\min\limits_{x,y\in R_2} \Phi(x,y)$.

Так как $\Phi(x^*,y^*) = 0 \Rightarrow \begin{cases} f(x^*,y^*)=0 \\ g(x^*,y^*)=0 \end{cases} \Rightarrow (x^*,y^*)$ решение системы (4.12).

Последовательность точек $\{x_k\},\{y_k\}$ получим по рекуррентной формуле

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + \alpha_k \begin{pmatrix} p_k \\ q_k \end{pmatrix}, \tag{4.14}$$

где $k = 0,1,2,\ldots$; $(p_k,q_k)^T$ – вектор, определяющий направление минимизации; α_k – скалярная величина, шаговый множитель.

При этом выполняется условие релаксации: $\Phi(x_{k+1},y_{k+1}) < \Phi(x_k,y_k)$.

Вектор $\begin{pmatrix} p_k \\ q_k \end{pmatrix} = -grad\Phi(x_k,y_k) = -\begin{pmatrix} \Phi'_x(x_k,y_k) \\ \Phi'_y(x_k,y_k) \end{pmatrix}$ – антиградиент $\Phi(x,y)$.

Тогда градиентный метод имеет вид:

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \alpha_k \begin{pmatrix} \Phi'_x(x_k,y_k) \\ \Phi'_y(x_k,y_k) \end{pmatrix}, \tag{4.15}$$

где оптимальный шаг $\alpha_k = \arg\min\limits_{\alpha>0} \Phi\begin{pmatrix} x_k - \alpha\Phi'_x(x_k,y_k) \\ y_k - \alpha\Phi'_y(x_k,y_k) \end{pmatrix}$. (4.16)

Формулы (4.15) и (4.16) определяют градиентный метод, который называют методом наискорейшего спуска.

Достоинство: глобальная скорость (из любой начальной точки процесс приведет к минимальной точке).

Недостаток: медленная скорость сходимости эквивалентная линейной, причем, скорость замедляется в окрестности корня. Лучше применять совместно с другими методами (сначала – спуск, затем – метод Ньютона).

Функция, реализующая алгоритм поиска корней методом наискорейшего спуска представлена следующим образом:

```
def speed_down(params, system, start):

    eps = 0.000001

    # Строим новую функцию путем возведения в квадрат исходных функций
    # и складываем их
    new_fun = 0
    for fun in system:
        new_fun += fun ** 2

    # Находим частные производные по x1, x2....
    diff_list = []
    for par in params:
        diff_list.append(sp.diff(new_fun, par))

    # Берем фиксированный шаг
    alpha = 0.3

    # Будем собирать приближенные корни на каждом шаге
    approximate_roots = []

    # Получаем необходимые данные для итерации
    val_func = get_data_for_speed_down(params, start, diff_list)

    old_root = sp.Matrix(start)
    new_root = old_root - alpha * val_func

    approximate_roots.append((list(old_root), None))

    while get_norm_vec(new_root - old_root) > eps:

        val_func = get_data_for_speed_down(params, new_root, diff_list)
        old_root = new_root
        new_root = sp.Matrix(old_root) - alpha * val_func

        approximate_roots.append((list(new_root), (get_norm_vec(new_root -
old_root))))

    return new_root, approximate_roots
```

Для осуществления итерации с промежуточными значениями использовалась следующая функция:

```python
# Получаем необходимые данные для текущей итерации методом спуска
def get_data_for_speed_down(params, start, diff_list):

    # Создаем словарь приближенных корней вида {x1: valume1, x2: valume2 ...}
    lst_starts = {params[i]: start[i] for i in range(len(params))}

    # Вычисление значения функции от точек x1, x2, ....
    val_func = []
    for index in range(len(diff_list)):
        val = diff_list[index].subs(lst_starts).evalf()
        val_func.append(round(val, 7))

    val_func = sp.Matrix(val_func)

    return val_func
```

Результаты работы программы для двух примеров (тестового и варианта 19):

```
Методо спуска
--------------------------------------------------
Из примера: [0.298710771277548, 3.87413851097226]
Из 19 варианта: [1.41423211619258, -2.22440123790875]
--------------------------------------------------
```

В результате подстановки найденных корней в систему нелинейных уравнений были получены следующие значения:

```
Проверка корней метода спуска
Для системы 1:
Значение в функции 1: 0.000001225969901000
Значение в функции 2: -0.000000689429927000
Для системы 2:
Значение в функции 1: -0.000002535261027000
Значение в функции 2: 0.000001549058551000
```

В результате проделанной лабораторной работы были реализованы два способа нахождения корней нелинейных уравнений. Результаты совпадают с заданной точностью. Тестирование производилось по примеру и варианту 19.

ПРИЛОЖЕНИЕ

Листинг программы

```python
import sympy as sp
from copy import deepcopy

# Получаем необходимые данные для текущей итерации методом Ньютона
def get_data_for_newton(J, params, system, start):
    # Создаем словарь приближенных корней вида {x1: value1, x2: value2
...}
    lst_starts = {params[i]: start[i] for i in range(len(params))}

    # Находим значение якбиана в точках x1, x2...
    J_x = []
    for index1 in range(len(J)):
        buff = []
        for index2 in range(len(J)):
            val = J[index1][index2].subs(lst_starts).evalf()
            buff.append(round(val, 7))
        J_x.append(buff)

    # Обратная матрица
    J_x = sp.Matrix(J_x)
    J_x_inv = J_x.inv()

    # Находим значение системы функций в точках x1, x2 ....
    F_x = []
    for i in range(len(system)):
        F_x.append(system[i].subs(lst_starts).evalf())

    F_x = sp.Matrix(F_x)

    return J_x_inv, F_x

# Метод Ньютона нахождения корней
def method_newton(params, system, start):
    eps = 0.000001

    # Строим якобиан от параметров x1, x2
    J = []
    for fun in system:
        buff = []
        for param in params:
            buff.append(sp.diff(fun, param))
        J.append(buff)

    # Получаем необходимые матрицы для расчета
    J_x_inv, F_x = get_data_for_newton(J, params, system, start)
```

```
    approximate_roots = []

    old_roots = sp.Matrix(deepcopy(start))
    new_roots = old_roots - J_x_inv * F_x

    approximate_roots.append((list(old_roots), None))

    while get_norm_vec(new_roots - old_roots) > eps:

        J_x_inv, F_x = get_data_for_newton(J, params, system, new_roots)

        old_roots = new_roots
        new_roots = old_roots - J_x_inv * F_x

        approximate_roots.append((list(old_roots),
                                 "{:0<20.15f}".format(get_norm_vec(new_roo
ts - old_roots))))


    return new_roots, approximate_roots

# Находим норму вектора
def get_norm_vec(vec):
    max_val = abs(vec[0])
    for val in vec[1:]:
        abs_val = abs(val)
        if abs_val > max_val:
            max_val = abs_val
    return max_val

# Получаем необходимые данные для текущей итерации методом спуска
def get_data_for_speed_down(params, start, diff_list):

    # Создаем словарь приближенных корней вида {x1: valume1, x2: valume2
...}
    lst_starts = {params[i]: start[i] for i in range(len(params))}

    # Вычисление значения функции от точек x1, x2, ....
    val_func = []
    for index in range(len(diff_list)):
        val = diff_list[index].subs(lst_starts).evalf()
        val_func.append(round(val, 7))

    val_func = sp.Matrix(val_func)

    return val_func

# Метод спуска
def speed_damn(params, system, start):

    eps = 0.000001
```

```python
    # Строим новую функцию путем возведения в квадрат исходных функций
    # и складываем их
    new_fun = 0
    for fun in system:
        new_fun += fun ** 2

    # Находим частные производные по x1, x2....
    diff_list = []
    for par in params:
        diff_list.append(sp.diff(new_fun, par))

    # Берем фиксированный шаг
    alpha = 0.3

    # Будем собирать приближенные корни на каждом шаге
    approximate_roots = []

    # Получаем необходимые данные для итерации
    val_func = get_data_for_speed_down(params, start, diff_list)

    # Стартовые значения
    old_root = sp.Matrix(start)
    new_root = old_root - alpha * val_func

    # Список приближенных корней в процессе итераций
    approximate_roots.append((list(old_root), None))

    while get_norm_vec(new_root - old_root) > eps:

        val_func = get_data_for_speed_down(params, new_root, diff_list)
        old_root = new_root
        new_root = sp.Matrix(old_root) - alpha * val_func

        approximate_roots.append((list(new_root), (get_norm_vec(new_root -
old_root))))

    return new_root, approximate_roots

# Функция для проверки найденных корней
def check_roots(funcs_list, lst_roots, params):
    # Связываем параметры x1, x2 со значениями из params
    roots_dict = {params[i]: lst_roots[i] for i in range(len(params))}
    for i, func in enumerate(funcs_list):
        # Если значение = 0, то format не сможет преобразовать такое
значение
        # поэтому обрабатываем отдельно
        if func.subs(roots_dict) != 0:
            print(f"Значение в функции {i + 1}:",
"{:0<20.15f}".format(func.subs(roots_dict)))
        else:
```

```python
            print(f"Значение в функции {i + 1}:", func.subs(roots_dict))


if __name__ == "__main__":
    # Тестовые примеры от двух систем с двумя переменными
    x1 = sp.Symbol("x1")
    x2 = sp.Symbol("x2")

    # Система из примера
    system_1 = [
                sp.sin(x1 + 1.5) - x2 + 2.9,
                sp.cos(x2 - 2) + x1
               ]

    # Система варианта 19
    system_2 = [
                sp.sin(x1 + x2) - x2 - 1.5,
                x1 + sp.cos(x2 - 0.5) - 0.5
               ]

    #==========================================================
    print("Методо Ньютона")
    print("-----------------------------------------------------")

    root1, data1 = method_newton([x1, x2], system_1, [5, 10])
    print("Из примера:", list(root1))
    root2, data2 = method_newton([x1, x2], system_2, [5, 10])
    print("Из 19 варианта:", list(root2))


    print("-----------------------------------------------------\n")
    #==========================================================

    #==========================================================
    print("Методо спуска")
    print("-----------------------------------------------------")

    root3, data3 = speed_damn([x1, x2], system_1, [5, 10])
    print("Из примера:", list(root3))
    root4, data4 = speed_damn([x1, x2], system_2, [5, 10])
    print("Из 19 варианта:", list(root4))

    print("-----------------------------------------------------\n")
    #===========================================================

    #===========================================================
    # Проверка полученных значений
    print("Проверка корней метода Ньютона")
    print("Для системы 1:")
    check_roots(system_1, root1, [x1, x2])
    print("Для системы 2:")
```

```
    check_roots(system_2, root2, [x1, x2])
    print("Проверка корней метода спуска")
    print("Для системы 1:")
    check_roots(system_1, root3, [x1, x2])
    print("Для системы 2:")
    check_roots(system_2, root4, [x1, x2])
    #============================================================
```