БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

Лабораторная работа 6

Метод Ньютона решения системы нелинейных уравнений

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 шар S_δ.
2. Решить систему методом Ньютона.
3. Вычислить невязку решения.
4. Проанализировать полученные результаты и сравнить с методом простой итерации и методом Гаусса-Зейделя.

2. Решение системы нелинейных уравнений

- Рассмотрим систему нелинейных уравнений:

$$\begin{cases} y - \frac{x^2}{2} + x - 0.5 = 0, \\ 2x + y - \frac{y^3}{6} - 1.6 = 0. \end{cases}$$

  Отделим корни путём выражения $y(x)$ из первого уравнения: $y(x) = \frac{x^2}{2} - x + 0.5$. Тогда подставим данное выражение во второе уравнение и отделим корни полученного нелинейного уравнения. Получим для каждого корня отрезки $[a_i, b_i]$, положим

$$x_i^0 = \left(\frac{a_i + b_i}{2}, y\left(\frac{a_i + b_i}{2}\right)\right)^T; \; S_\delta = \left\{\mathbf{x} \middle| ||\mathbf{x} - \mathbf{x}_i^0||_\infty \leq \delta, \; \delta = \max\left(\frac{y(a) + y(b)}{2}, \frac{b - a}{2}\right)\right\} \tag{1}$$

  Как видно из результата выполнения программы, для корней получаем:

$$\begin{aligned} x_1^0 &= (0.78282828, 0.02358178)^T \\ r_1 &= 0.05050505 \\ x_2^0 &= (3.81313131, 3.95686389)^T \\ r_2 &= 0.07103867 \end{aligned}$$

- Построим итерационный процесс для нахождения корней $\mathbf{x}_i$ уравнения $\mathbf{F}(\mathbf{x}) = \mathbf{0}$:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \left[\frac{d\mathbf{F}}{d\mathbf{x}}\right]^{-1} \mathbf{F}(\mathbf{x}^k); \mathbf{x}^0 = \mathbf{x}_i^0, k = 1, 2, ... \tag{2}$$

3. Листинг программы

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

```
#SystemCommon.py
import math
import numpy as np
import matplotlib.pyplot as plt

def f1(x, y):
    return y - 0.5 * x ** 2 + x - 0.5

def f1_xprime(x, y):
    return -x + 1

def f1_yprime(x, y):
    return 1

def f2(x, y):
    return 2 * x + y - (y ** 3) / 6 - 1.6

def f2_xprime(x, y):
    return 2

def f2_yprime(x, y):
```

```
    return 1 - (y ** 2) / 2

def f(x, y):
    return np.array([f1(x, y), f2(x, y)], dtype=np.double)

def df(x, y):
    return np.array([[f1_xprime(x, y), f1_yprime(x, y)], [f2_xprime(x, y), f2_yprime(x, y)]],
        dtype=np.double)

def ysub(x):
    return 0.5 * x ** 2 - x + 0.5

def ysub_prime(x):
    return x - 1

def f2_ysub(x):
    return f2(x, ysub(x))

def f2_ysub_prime(x):
    return 2 + ysub_prime(x) - (ysub(x) ** 2) * ysub_prime(x) / 2

def infNorm(x):
    return np.max(np.abs(x))

import Newton

intervals = Newton.get_intervals_table(0.0, 5.0, f2_ysub, 100)

roots = np.array([(interval[0] + interval[1]) / 2 for interval in intervals])
radii = np.array([max((ysub(interval[1]) - ysub(interval[0])) / 2, interval[1] - interval[0]) for
    interval in intervals])

print("x values for starting points: " + str(roots))
print("Sphere radia: " + str(radii))

points = np.array([(root, ysub(root)) for root in roots], dtype=np.double)

print("Starting points: " + str(points))

#SystemNewton.py

from SystemCommon import *

def systemNewton(f, df, point, outerEps):
    def invert(A):
        return np.linalg.inv(A)

    prevPoint = np.copy(point)
    jacobi = df(prevPoint[0], prevPoint[1])
    point = prevPoint - np.dot(invert(jacobi), f(prevPoint[0], prevPoint[1]))
    iterNum = 1
    while (infNorm(point - prevPoint) >= outerEps):
        prevPoint = np.copy(point)
        jacobi = df(prevPoint[0], prevPoint[1])
        point = prevPoint - np.dot(invert(jacobi), f(prevPoint[0], prevPoint[1]))
        iterNum += 1
    return (point, iterNum)

if __name__ == '__main__':
    roots = []

    for point in points:
        print("Starting point: " + str(point))
        print("Deficiency before: " + str(f(point[0], point[1])))
        (root, iterNum) = systemNewton(f, df, point, 10 ** -5)
        print("Newton root: " + str(root))
```

```
    if len(roots) == 0:
        roots = [root]
    else:
        roots.append(root)
    print("Deficiency after: " + str(f(root[0], root[1])))
    print("Number of iterations: " + str(iterNum))
    print("\n")
```

4. Вывод программы

```
x values for starting points: [0.78282828 3.81313131]
Sphere radia: [0.05050505 0.07103867]
Starting points: [[0.78282828 0.02358178]
 [3.81313131 3.95685389]]

Starting point: [0.78282828 0.02358178]
Deficiency before: [ 0.         -0.01076384]
(0.788855413987323395, 0.022291018049479104)
(0.788855413960984908, 0.022291018106793549)
Newton root: [0.78885541 0.02229102]
Deficiency after: [0. 0.]
Number of iterations: 3


Starting point: [3.81313131 3.95685389]
Deficiency before: [ 0.         -0.34209107]
(3.792799298934621532, 3.899863859260823240)
(3.792799122123207578, 3.899863468266064004)
Newton root: [3.79279912 3.89986347]
Deficiency after: [-1.55431223e-14 -2.97095681e-13]
Number of iterations: 3
```

5. Выводы

- В отличие от метода Гаусса-Зейделя, метод Ньютона сошёлся в окрестности меньшего корня за 3 итерации с невязкой, равной нулю.
- Сравним точность и скорость сходимости рассмотренных методов в окрестности большего корня:

$$
\begin{aligned}
k_{Gauss-Zeidel} &= 9;\\
k_{Newton} &= 3;\\
k_{SimpleIter} &= 4;\\
||r_{Gauss_Zeidel}||_\infty &= 6.98076505e-06;\\
||r_{Newton}||_\infty &= 2.97095681e-13;\\
||r_{SimpleIter}||_\infty &= 3.22813787e-07.
\end{aligned}
$$

- Таким образом, метод Ньютона обладает самой высокой скоростью сходимости и, как следствие, точностью среди рассмотренных методов решения систем нелинейных уравнений. Однако, как и в случае одного уравнения, он предполагает ограничения на уравнения системы: в частности, каждая функция $f_i(\mathbf{x})$ должна быть непрерывно дифференцируемой по всем переменным, что сужает область применения данного метода. Данное замечание справедливо и для рассмотренного ранее метода простой итерации в силу использования частных производных функций системы для нахождения канонического вида, в то время, как рассмотренная реализация метода Гаусса-Зейделя требует лишь непрерывных частных производных $\frac{\partial f_i}{\partial x_i}$.