

# Нелинейные уравнения и системы

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Дек 11, 2018

## Содержание

<b>1</b>	<b>Нелинейные системы и уравнения</b>	<b>1</b>
<b>2</b>	<b>Метод Ньютона</b>	<b>2</b>
2.1	Решение нелинейных уравнений . . . . .	2
2.2	Решение нелинейных систем . . . . .	5
<b>3</b>	<b>Задачи</b>	<b>6</b>
1:	Метод бисекций . . . . .	6
2:	Метод секущих . . . . .	6
3:	Метод Ньютона для систем уравнений . . . . .	6
	<b>Предметный указатель</b>	<b>7</b>

## Аннотация

Многие прикладные задачи приводят к необходимости нахождения приближенного решения нелинейных уравнений и систем нелинейных уравнений. С этой целью используются итерационные методы. Приведены алгоритмы решения нелинейных уравнений с одним неизвестным и систем нелинейных уравнений.

## 1. Нелинейные системы и уравнения

Ищется решение нелинейного уравнения

$$f(x) = 0, \tag{1}$$

где  $f(x)$  — заданная функция. Корни уравнения (1) могут быть комплексными и кратными. Выделяют как самостоятельную проблему разделения корней, когда

проводится выделение области в комплексной плоскости, содержащей один корень. После этого на основе тех или иных итерационных процессов при выбранном начальном приближении находится решение нелинейного уравнения (1).

В более общем случае мы имеем не одно уравнение (1), а систему нелинейных уравнений

$$f_i(x_1, x_2, \dots, x_n) = 0, \quad i = 1, 2, \dots, n. \quad (2)$$

Обозначим через  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  вектор неизвестных и определим вектор-функцию  $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_n(\mathbf{x}))$ . Тогда система (2) записывается в виде

$$\mathbf{F}(\mathbf{x}) = 0. \quad (3)$$

Частным случаем (3) является уравнение (1) ( $n = 1$ ). Вторым примером (3) — система линейных алгебраических уравнений, когда  $\mathbf{F}(\mathbf{x}) = A\mathbf{x} - \mathbf{f}$ .

## 2. Метод Ньютона

### 2.1. Решение нелинейных уравнений

При итерационном решении уравнений (1), (3) задается некоторое начальное приближение, достаточно близкое к искомому решению  $x^*$ . В одношаговых итерационных методах новое приближение  $x_{k+1}$  определяется по предыдущему приближению  $x_k$ . Говорят, что итерационный метод сходится с линейной скоростью, если  $x_{k+1} - x^* = O(x_k - x^*)$  и итерационный метод имеет квадратичную сходимость, если  $x_{k+1} - x^* = O(x_k - x^*)^2$ .

В итерационном методе Ньютона (методе касательных) для нового приближения имеем

$$x_{k+1} = x_k + \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, \dots, \quad (4)$$

Вычисления по (4) проводятся до тех пор, пока  $f(x_k)$  не станет близким к нулю. Более точно, до тех пор, пока  $|f(x_k)| > \varepsilon$ , где  $\varepsilon$  — малая величина.

Простейшая реализация метода Ньютона может выглядеть следующим образом:

---

```
def naive_Newton(f, dfdx, x, eps):
    while abs(f(x)) > eps:
        x = x - float(f(x))/dfdx(x)
    return x
```

---

Чтобы найти корень уравнения  $x^2 = 9$  необходимо реализовать функции

---

```
def f(x):
    return x**2 - 9

def dfdx(x):
    return 2*x
```

---

```
print naive_Newton(f, dfdx, 1000, 0.001)
```

---

Данная функция хорошо работает для приведенного примера. Однако, в общем случае могут возникать некоторые ошибки, которые нужно отлавливать. Например: пусть нужно решить уравнение  $\tanh(x) = 0$ , точное решение которого  $x = 0$ . Если  $|x_0| \leq 1.08$ , то метод сходится за шесть итераций.

---

```
-1.0589531343563485
0.9894042072982367
-0.784566773085775
0.3639981611100014
-0.03301469613719421
2.3995252668003453e-05
Число вызовов функций: 25
Решение: 3.0000000001273204
```

---

Теперь зададим  $x_0$  близким к 1.09. Возникнет переполнение

---

```
-1.0933161820201083
1.104903543244409
-1.1461555078811896
1.3030326182332865
-2.064923002377556
13.473142800575976
-126055892892.66042
```

---

Возникнет деление на ноль, так как для  $x_7 = -126055892892.66042$  значение  $\tanh(x_7)$  при машинном округлении равно 1.0 и поэтому  $f'(x_7) = 1 - \tanh(x_7)^2$  становится равной нулю в знаменателе.

Проблема заключается в том, что при таком начальном приближении метод Ньютона расходится.

Еще один недостаток функции `naive_Newton` заключается в том, что функция `f(x)` вызывается в два раза больше, чем необходимо.

Учитывая выше сказанное реализуем функцию с учетом следующего:

1. обрабатывать деление на ноль
2. задавать максимальное число итераций в случае расходимости метода
3. убрать лишний вызов функции `f(x)`

---

```
def Newton(f, dfdx, x, eps):
    f_value = f(x)
    iteration_counter = 0
    while abs(f_value) > eps and iteration_counter < 100:
        try:
            x = x - f_value/dfdx(x)
```

---

```

except ZeroDivisionError as err:
    print("Ошибка: {}".format(err))
    sys.exit(1)
f_value = f(x)
iteration_counter += 1

if abs(f_value) > eps:
    iteration_counter = -1
return x, iteration_counter

```

---

Метод Ньютона сходится быстро, если начальное приближение близко к решению. Выбор начального приближения влияет не только на скорость сходимости, но и на сходимость вообще. Т.е. при неправильном выборе начального приближения метод Ньютона может расходиться. Неплохой стратегией в случае, когда начальное приближение далеко от точного решения, может быть использование нескольких итераций по методу бисекций, а затем использовать метод Ньютона.

При реализации метода Ньютона нужно знать аналитическое выражение для производной  $f'(x)$ . Python содержит пакет SymPy, который можно использовать для создания функции `dfdx`. Для нашей задачи это можно реализовать следующим образом:

---

```

from sympy import symbol, tanh, diff, lambdify

x = symbol.Symbol('x')      # определяем математический символ x
f_expr = tanh(x)             # символьное выражение для f(x)
dfdx_expr = diff(f_expr, x)  # вычисляем символьно f'(x)

# Преобразуем f_expr и dfdx_expr в обычные функции Python
f = lambdify([x],           # аргумент f
             f_expr)
dfdx = lambdify([x], dfdx_expr)

tol = 1e-1
sol, no_iterations = Newton(f, dfdx, x=1.09, eps=tol)
if no_iterations > 0:
    print("Уравнение: tanh(x) = 0. Число итераций: {}".format(no_iterations))
    print("Решение: {}, eps = {}".format(sol, tol))
else:
    print("Решение не найдено!")

def F(x):
    y = np.zeros_like(x)
    y[0] = (3 + 2*x[0])*x[0] - 2*x[1] - 3
    y[1:-1] = (3 + 2*x[1:-1])*x[1:-1] - x[:-2] - 2*x[2:] - 2
    y[-1] = (3 + 2*x[-1])*x[-1] - x[-2] - 4
    return y

def J(x):
    n = len(x)
    jac = np.zeros((n, n))
    jac[0, 0] = 3 + 4*x[0]

```

```

jac[0, 1] = -2
for i in range(n-1):
    jac[i, i-1] = -1
    jac[i, i] = 3 + 4*x[i]
    jac[i, i+1] = -2
jac[-1, -2] = -1
jac[-1, -1] = 3 + 4*x[-1]

return jac

n = 10
guess = 3*np.ones(n)

sol, its = Newton_system(F, J, guess)

if its > 0:
    print("x = {}".format(sol))
else:
    print("Решение не найдено!")

```

---

## 2.2. Решение нелинейных систем

Идея метода Ньютона для приближенного решения системы (2) заключается в следующем: имея некоторое приближение  $\mathbf{x}^{(k)}$ , мы находим следующее приближение  $\mathbf{x}^{(k+1)}$ , аппроксимируя  $\mathbf{F}(\mathbf{x}^{(k+1)})$  линейным оператором и решая систему линейных алгебраических уравнений. Аппроксимируем нелинейную задачу  $\mathbf{F}(\mathbf{x}^{(k+1)}) = 0$  линейной

$$\mathbf{F}(\mathbf{x}^{(k)}) + \mathbf{J}(\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = 0, \quad (5)$$

где  $\mathbf{J}(\mathbf{x}^{(k)})$  — матрица Якоби (якобиан):

$$\nabla \mathbf{F}(\mathbf{x}^{(k)}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_1} & \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_2} & \dots & \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_n} \\ \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_1} & \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_2} & \dots & \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_n} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_1} & \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_2} & \dots & \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_n} \end{bmatrix}$$

Уравнение (5) является линейной системой с матрицей коэффициентов  $\mathbf{J}$  и вектором правой части  $-\mathbf{F}(\mathbf{x}^{(k)})$ . Систему можно переписать в виде

$$\mathbf{J}(\mathbf{x}^{(k)})\boldsymbol{\delta} = -\mathbf{F}(\mathbf{x}^{(k)}),$$

где  $\boldsymbol{\delta} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ .

Таким образом,  $k$ -я итерация метода Ньютона состоит из двух стадий:

1. Решается система линейных уравнений (СЛАУ)  $\mathbf{J}(\mathbf{x}^{(k)})\boldsymbol{\delta} = -\mathbf{F}(\mathbf{x}^{(k)})$  относительно  $\boldsymbol{\delta}$ .

2. Находится значение вектора на следующей итерации  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}$ .

Для решения СЛАУ можно использовать приближенные методы. Можно также использовать метод Гаусса. Пакет `numpy` содержит модуль `linalg`, основанный на известной библиотеке LAPACK, в которой реализованы методы линейной

алгебры. Инструкция `x = numpy.linalg.solve(A, b)` решает систему  $Ax = b$  методом Гаусса, реализованным в библиотеке LAPACK.

Когда система нелинейных уравнений возникает при решении задач для нелинейных уравнений в частных производных, матрица Якоби часто бывает разреженной. В этом случае целесообразно использовать специальные методы для разреженных матриц или итерационные методы.

Можно также воспользоваться методами, реализованными для систем линейных уравнений.

### 3. Задачи

#### Задача 1: Метод бисекций

Написать программу для нахождения решения нелинейного уравнения  $f(x) = 0$  методом бисекций. С ее помощью найдите корни уравнения  $(1 + x^2)e^{-x} + \sin x = 0$  на интервале  $[0, 10]$ .

#### Задача 2: Метод секущих

Напишите программу для решения нелинейного уравнения  $f(x) = 0$  методом секущих. Используйте ее для решения уравнения  $4 \sin x + 1 - x = 0$  на интервале  $[-10, 10]$ .

#### Задача 3: Метод Ньютона для систем уравнений

Написать программу для нахождения решения системы нелинейных уравнений  $F(x) = 0$  методом Ньютона. С ее помощью найдите приближенное решение системы

$$\begin{aligned}(3 + 2x_1)x_1 - 2x_2 &= 3, \\ (3 + 2x_i)x_i - x_{i-1} - 2x_{i+1} &= 2, \quad i = 2, 3, \dots, n-1, \\ (3 + 2x_n)x_n - x_{n-1} &= 4.\end{aligned}$$

при  $n = 10$  и сравните его с точным решением  $x_i = 1, i = 1, 2, \dots, n$ .

## **Предметный указатель**

Матрица  
Якоби, 5

якобиан, 5