Instituto Tecnológico de Buenos Aires

93.54 - Métodos Numéricos

Trabajo Práctico 3: Ecuaciones no-lineales

Grupo 8

Crespo, Elisabet del Pilar	59098
Martorell, Ariel	56209
Vekselman, Alan	59378

Profesores

Fierens, Pablo Ignacio
Grisales Campeón, Juan Pablo

Fecha de entrega: 29/04/2021

Índice

1. Introducción

En este trabajo se propuso implementar dos algoritmos que resuelvan los siguientes problemas, respectivamente:

$$y = x^p \tag{1}$$

con $x \geq 0$ y $p \in Z$, y:

$$y = x^{\frac{p}{q}} \tag{2}$$

con $x \geq 0$ y $p, q \in Z$.

Este último debe resolverse mediante el uso de ecuaciones no-lineales.

2. Newton-Raphson

El algoritmo de Newton-Raphson se utiliza para hallar los ceros de una función $f(x)$, basándose en el desarrollo de Taylor de primer orden de $f(x)$ tal que:

$$0 = f(x^*) \approx f(x) + f'(x) \cdot (x^* - x) \Rightarrow x^* \approx x - \frac{f(x)}{f'(x)} \tag{3}$$

De esta forma, se obtiene para cada iteración:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{4}$$

3. Implementación del algoritmo

3.1. Powerint

El algoritmo *powerint* resuelve $y = x^p$. Inicialmente se analiza el caso en el que $x = 0$, devolviendo 0 si $p > 0$, ∞ si $p < 0$, y 1 cuando $p = 0$.

Luego se define $res = 1$, $k = x$ cuando p es positivo o $k = \frac{1}{x}$ en caso contrario, y $p = |p|$.

Comienzan a realizarse las iteraciones comenzando con $|p|$. Si p es impar, se multiplica *res* por k. Si no directamente se cambian los valores de k a k^2 y p al resultado de su división entera por 2.

Estas iteraciones se repiten mientras que $p > 0$, obteniendo un numero de $log_2(p)$ iteraciones, haciendo de esta manera que el algoritmo funcione mas rápido que la implementación trivial en donde se haría $k \cdot k \cdot k$....

Por ultimo se devuelve *res* con el resultado final.

3.2. Powerrat

El algoritmo de *powerrat* consiste en expresar a la ecuación $r = x^{\frac{p}{q}}$ de la siguiente forma:

$$x^{\frac{p}{q}} = x^{p//q} \cdot x^{\frac{p\%q}{q}} = r \tag{5}$$

Donde el operador // significa división entera y el operador % significa resto. De esta forma, el problema se convirtió en una multiplicación entre un resultado obtenido con *powerint* y la resolución de x^m, donde m tiene módulo menor a 1.

De esta manera, se realiza el siguiente cambio de variable:

$$r_2 = \frac{r}{x^{p//q}} \tag{6}$$

Por lo tanto, ahora se debe utilizar el algoritmo de Newton-Raphson para resolver la siguiente ecuación:

$$r_2^q = x^{p\,\%q} \tag{7}$$

Finalmente, el resultado de *powerrat* será $r_2 \cdot x^{p//q}$. Cabe destacar que se utilizó una precisión de 10^{-15}, con lo cual no se podrán realizar cálculos en los cuales el resultado sea de este orden de magnitud, ya que devolverá algo incorrecto.

También se debe destacar que en la implementación real, para no tener problemas con los signos y los intervalos de convergencia, se decidió realizar todo el procedimiento con los valores absolutos de p y q, obtener el resultado final, y luego elevarlo al signo de $p \cdot q$.

Por último, se utilizó un límite máximo de 200 iteraciones, con lo cual es posible que no siempre converja bien cuando el resultado se encuentre lejos del valor inicial elegido, que fue x. Esto se debe a lo siguiente: conociendo la ecuación que se debe resolver, se puede ver que, dado que siempre $p\,\%q$ es menor a q, entonces siempre se cumplirá que $r_2 < x$. De esta manera, se decidió empezar por una cota máxima del resultado, con lo cual este último siempre estará hacia la izquierda del punto inicial. En los límites explicados en la Sec. 3.3, el algoritmo funciona bien. Sin embargo, al elevar los valores de x y q, comienzan a ser necesarias cada vez más iteraciones, ya que los valores intermedios utilizados serán cada vez más extremos (muy pequeños o muy grandes).

3.2.1. Newton-Raphson especializado

Dado que ya se conocía la forma del problema a resolver, se decidió hacer una implementación especializada del algoritmo para resolver dicho problema. Éste será de la siguiente forma:

$$E(r) = r^q - m = 0 \tag{8}$$

Donde r es la incógnita, y m y q son conocidos. De esta forma, su derivada será:

$$\frac{dE(r)}{dr} = q \cdot r^{q-1} \tag{9}$$

Reemplazando en la fórmula Eq. 4, se tendrá:

$$x_{k+1} = x_k - \frac{x_k^q - m}{q \cdot x_k^{q-1}} \tag{10}$$

Distribuyendo el cociente, se tendrá:

$$x_{k+1} = x_k - \frac{x_k}{q} + \frac{m}{q} \cdot x_k^{1-q} \tag{11}$$

Finalmente, sacando factor común $\frac{x_k}{q}$, se llegará a:

$$x_{k+1} = \frac{x_k}{q} \cdot (q - 1 + m \cdot x_k^{-q}) \tag{12}$$

Este último es el resultado utilizado en la implementación del algoritmo. Se iterará hasta que ocurra alguna de las dos condiciones siguientes:

- En caso de que el límite de iteraciones haya superado al máximo provisto como parámetro, el programa cortará y devolverá el valor actual de x_{k+1}.
- En caso de que la diferencia entre x_k y x_{k+1} sea menor al error provisto como parámetro, el programa cortará y devolverá el valor actual de x_{k+1}.

3.3. Verificación del algoritmo

Para verificar el correcto funcionamiento del algoritmo, se implementó la función *test*. Ésta no recibe parámetros.

Se decidió realizar una cantidad arbitraria de 500 intentos. En cada uno de ellos, se generan números aleatorios x, p y q. Sus límites serán:

- $0 < x < 50$
- $-100 \leq p < 100$
- $-50 < q < 50$ & $q \neq 0$

Luego, se procederá a obtener en la variable *calc* el resultado de llamar a *powerrat* con los valores de x, p y q generados. En la variable *real* se guardará el resultado real obtenido de hacer $x^{\frac{p}{q}}$. Si el resultado obtenido con *powerrat* difiere del real en menos de un 0.1 % de éste, entonces se considerará que el resultado fue correcto y se continuará a la siguiente iteración, imprimiendo *Worked* en caso de haber pasado todos los intentos. En caso de fallar en alguno, se imprimirá *Failed*, y se saldrá de la función.

4. Anexo

4.1. Powerint

```python
import numpy as np

# Exponentiation by squaring
def powerint(x : float, p : int) -> float:
    if not x: return 0 if p > 0 else np.inf if p < 0 else 1

    res, k, p = 1, x if p > 0 else 1/x, abs(p)
    while p > 0:
        if p%2: res *= k
        k, p = k * k, p//2

    return res
```

4.2. Newton-Raphson especializado

```python
def newt_raph_power(q, k, maxiter, x_ini, tol = 1e-15):
    """ Solves x**q - k = 0 """

    for i in range(maxiter):
        x_k1 = x_ini / q * (q - 1 + k * powerint(x_ini, -q))
        if abs(x_k1 - x_ini) <= tol: return x_k1
        x_ini = x_k1

    return x_ini
```

4.3. Powerrat

```python
# Para no usar np.sign
def sign(x): return 0 if not x else -1 if x < 0 else 1

def powerrat(x : float, p : int, q : int) -> float:
    if x == 1 or not p: return 1
    if not q: return np.inf if p > 0 else 0

    newp, newq = abs(p), abs(q)

    if newq == 1: return powerint(x, p * sign(q))
    if newp == newq: return powerint(x, sign(p * q))

    k = powerint(x, newp % newq)
    k2 = powerint(x, newp//newq)

    res = newt_raph_power(newq, k, 200, x) * k2
    return powerint(res, sign(p * q))
```

4.4. Test

```python
def test():
    tot = 500
    for i in range(tot):
        x = np.random.randint(1, 50)
        p = np.random.randint(-100, 100)
        q = np.random.randint(1, 50) * (-1)**np.random.randint(0, 2)

        real = x**(p/q)
        calc = powerrat(x, p, q)
        worked = abs(calc - real) / real <= 1e-3

        if not worked:
            print('Failed')
            return

    print('Worked')
```