Instituto Tecnológico de Costa Rica
Ingeniería en Computadores
Nombre: Eduardo Moya Bello

CE-3102: Análisis Numérico para Ingeniería
Semestre: II - 2019
Carné: 2015096664

Catálogo del Curso

A continuación se presenta un catálogo que abarca todos los métodos numéricos estudiados a lo largo del curso de Análisis Numérico para Ingeniería. Además, los códigos fuente que se muestran en este trabajo están disponibles en Github.

Índice

1. Solución de Ecuaciones No Lineales

1.1. Consideraciones iniciales

1.1.1. Definiciones de error

Este catálogo se remite a algunos conceptos básicos de análisis numérico que no se cubrirán a excepción de las principales definiciones de error. El **error absoluto** del método viene dado por la siguiente expresión:

$$e_a = |x_{n+1} - x_n| \tag{1.1}$$

Además, el **error relativo** es:

$$e_r = \frac{|x_{n+1} - x_n|}{|x_{n+1}|} \tag{1.2}$$

Asimismo, la definición de **tolerancia** consiste en un número prefijado *tol* tal que:

$$e_r \leq tol \tag{1.3}$$

Es muy importante tomar en cuenta que **los algoritmos que se implementarán en este catálogo utilizarán como criterio de parada, la tolerancia** de la ecuación (1.3). Teniendo claros los conceptos anteriores, se puede procedes a analizar los métodos explicados en el presente catálogo.

1.2. Método de Bisección

1.2.1. Fórmula matemática

Punto medio:

$$x_1 = \frac{a_1 + b_1}{2} \tag{1.4}$$

1.2.2. Descripción breve del método

El método de bisección genera una sucesión de intervalos $I_k = [a_k, b_k]$, que satisfacen la propiedad $f(a_k)f(b_k) < 0$, donde:

$$I_{k+1} = [a_{k+1}, b_{k+1}] = \begin{cases} [a_k, x_k], & \text{si } f(a_k)f(x_k) < 0 \\ [x_k, b_k], & \text{si } f(a_k)f(x_k) > 0 \end{cases} \tag{1.5}$$

1.2.2.1 Valores iniciales

Los valores iniciales para implementar el método son el intervalo inicial (para obtener su punto medio) y una criterio de parada (iteraciones máximas, tolerancia, error).

1.2.2.2 Convergencia

Sea f una función continua en $[a, b]$ y $f(a)f(b) < 0$. El método de bisección genera una sucesión $\{x_k\}_{k=1}^{\infty}$ que converge a un valor cero $\xi \in [a_n, b_n]$ tal que $f(\xi) = 0$, donde

$$e_k = |x_k - \xi| \leq \frac{b-a}{2^k} \tag{1.6}$$
$$\lim_{k \to \infty} x_k = \xi \tag{1.7}$$

Nótese que el **error del método de bisección** se obtiene de la ecuación (1.6). Por otra parte, si se desea utilizar una tolerancia tol, entonces el valor mínimo de $iterMax$ para la iteración de bisección puede ser considerado de la forma:

$$iterMax = \left\lfloor log_2 \left(\frac{b-a}{tol} \right) \right\rfloor + 1 \tag{1.8}$$

Donde $\lfloor x \rfloor$ representa la parte entera de x.

1.2.2.3 Ventajas

- Es una opción viable y sencilla de aplicar para funciones fáciles de evaluar y de una sola raíz.
- Requiere pocos recursos a nivel computacional en comparación con otros métodos. Tal es el caso de aquellos que requieren cálculo de derivdads como el Método de Newton-Raphson (1.3).

1.2.2.4 Desventajas

- Al dividir el intervalo en partes iguales, no se toma en cuenta qué tan cerca la aproximación está de la solución, por lo que el método es ineficiente en casos donde la solución esté lejos de la mitad del intervalo.
- No es capaz de obtener varias soluciones.
- Puede ser difícil o imposible de utilizar en ciertas funciones.

1.2.3. Pseudocódigo del método

```
Bisection(f, a, b, tol) -> [xn, err, iter, fx]
  % Bisection Method
  % Inputs:
  %   - f is a polinomial expression introduced as a symbolic expression
  %   - a and b are [a, b]
  %   - tol is the tolerance
  % Outputs:
  %   - xn is the solution
  %   - err is the error
```

```
  %   - iter is the amount of completed iterations
  %   - fx is f(x)
  Handle the function
  Evaluate f(a)
  Evaluate f(b)
  If f(a) * f(b) is positive:
    End function
  Calculate the maximum amount of iterations: maxIter
  Starting at iter = 0 and ending at maxIter:
    Calculate the middle point
    Evaluate the function at the middle point: f(x_n)
    Calculate the error
    If f(x_n) == 0:
      Exact root found!
      a = xn
      b = xn
    If f(x_n) * f(b) > 0:
      b = x_n
          f(b) = f(x)
      else
          a = x_n
          f(a) = f(x)
    If the tolerance is satisfied:
      End function
```

Código 1: Pesudocódigo del Método de Bisección

1.2.4. Código OCTAVE del Método

```
function [xn, err, iter, fx] = bisection(f, a, b, tol)
  % Bisection Method
  % Inputs:
  % - f is a polinomial expression introduced as a symbolic expression
  % - a and b are [a, b]
  % - tol is the tolerance
  % Outputs:
  % - xn is the solution
  % - err is the error
  % - iter is the amount of completed iterations (-1 if IntervalError)
  % - fx is f(x)
  % Errors:
  % - IntervalError: the specified interval does not contain the zero
  f = function_handle(f);
  fa = f(a);
  fb = f(b);
  xn = 0;
  err = 0;
  iter = -1;
  fx = 0;
  if fa * fb > 0
    return
  endif
  maxIter = 1 + round((log(b - a) - log(tol))/ log(2));
  figure;
```

```
  hold on;
  for iter=0:maxIter
    xn = (a + b)/2;
    fx = f(xn);
    err = abs(b - a);
    if fx == 0
      a = xn;
      b = xn;
    elseif fb * fx > 0
      b = xn;
      fb = fx;
    else
      a = xn;
      fa = fx;
    endif
    if err <= tol
      break
    endif
    plot(iter, err, 'ro');
    title("Bisection Method");
    xlabel("Error");
    ylabel("Iterations");
  endfor
endfunction
```

Código 2: Método de Bisección en Octave

1.2.5. Código Python del Método

```
from sympy import *

x = symbols('x')


def bisection(expr, a, b, tol):
    """ Bisection Method

    Arguments:
        expr {string} -- is the polinomial expression
        a {float} -- is "a" in bisection interval [a, b]
        b {float} -- is "b" in bisection interval [a, b]
        tol {float} -- is the tolerance
    Returns:
        xn {float} -- is the solution
        err {float} -- is the error
        _iter {int} -- is the amount of iterations
        fx {float} -- is f(xn)

    """
    errReturn = [0, 0, 0, 0]
    try:
        f = sympify(expr)
        fa = f.subs(x, a)
        fb = f.subs(x, b)
```

```
        if (fa * fb > 0):
            return errReturn
        maxIter = 1 + round(N((log(b - a) - log(tol)) / log(2)))
        for _iter in range(0, maxIter):
            xn = (a + b)/2
            fx = f.subs(x, xn)
            err = abs(b - a)
            if (fx == 0):
                a = xn
                b = xn
            elif ((fb * fx) > 0):
                b = xn
                fb = fx
            else:
                a = xn
                fa = fx
            if (err <= tol):
                break
        return [xn, err, _iter, fx]
    except:
        print("There was an error.")
        return errReturn
```

Código 3: Método de Bisección en Python

1.3. Método de Newton-Raphson

1.3.1. Fórmula matemática

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad \text{para } k = 0, 1, 2, ... \tag{1.9}$$

Con $f'(x_k) \neq 0$ para todo $k \geq 0$ y x_0 algún valor inicial.

1.3.2. Descripción breve del método

1.3.2.1 Valores iniciales

Los valores iniciales para implementar el método son el valor x_0 para el inicio de las iteraciones y una criterio de parada (iteraciones máximas, tolerancia, error).

1.3.2.2 Convergencia

Suponiendo que el Método de Newton-Raphson genera una sucesión $\{p_n\}_{n=0}^{\infty}$ que converge a un cero p de la función $f(x)$. Si p es una **raíz simple**, entonces la convergencia es **cuadrática**:

$$|E_{n+1}| \approx \frac{|f''(p)|}{2|f'(p)|}|E_n|, \quad \text{para } n \text{ suficientemente grande} \tag{1.10}$$

Si p es una **raíz múltiple** de orden $M > 1$, entonces la convergencia es **lineal**:

$$|E_{n+1}| \approx \frac{M-1}{M}|E_n|, \quad \text{para } n \text{ suficientemente grande} \tag{1.11}$$

1.3.2.3 Ventajas

- Es un método ampliamente utilizado para la aproximación de raíces.
- En general, es muy eficiente ya que requiere menos iteraciones que otros métodos.

1.3.2.4 Desventajas

- Hay casos en los que se comporta de manera ineficiente.
- Su convergencia depende de la naturaleza de la función, ya que en algunos casos se dan problemas como la *periodicidad* o la *oscilación*.
- Su convergencia también depende de la exactitud del valor inicial. De hecho, para algunas funciones ningún valor inicial funciona.
- Puede suceder la división por cero.

1.3.3. Pseudocódigo del método

```
Newton(f, x_0, tol, maxIter) -> [xn, err, iter, fx]
  % Newton-Raphson Method
  % Inputs:
  %   - f is a polinomial expression introduced as a symbolic expression
  %   - x_0 is the initial value
  %   - tol is the tolerance
  %   - maxIter is the maximum amount of iterations
  % Outputs:
  %   - xn is the solution
  %   - err is the error
  %   - iter is the amount of completed iterations
  %   - fx is f(x)
  Calculate the expression of the derivative of the function
  Handle the function
  Handle the derivative
  Initialize xNext as x0
  Starting at iter = 0 and ending at maxIter:
    x_n now is what was previously defined as xNext
    Evaluate the derivative in x_n: f'(x_n)
    If f'(x_n) == 0:
      Alert division by zero
      End function
    Evaluate the function f(x)
    Calculate x_{n+1} using the Newton-Raphson's definition
    Calculate the error
    If the tolerance is satisfied:
      End function
```

Código 4: Pesudocódigo del Método de Newton-Raphson

1.3.4. Código OCTAVE del Método

```
function [xn, err, iter, fx] = newton(f, x0, tol, maxIter)
  % Newton-Raphson Method
  % Inputs:
  % - f is a polinomial expression introduced as a symbolic expression
  % - x0 is the initial value
  % - tol is the tolerance
  % - maxIter is the maximum amount of iterations
  % Outputs:
  % - xn is the solution
  % - err is the error
  % - iter is the amount of completed iterations
  % - fx is f(x)
  fd = diff(f);
  f = function_handle(f);
  fd = function_handle(fd);
  xNext = x0;
  figure
  hold on
  for iter=0:maxIter
    xn = xNext;
    fdx = fd(xn);
    if fdx == 0
      disp("Error: Division by zero");
      return
    endif
    fx = f(xn);
    xNext = xn - fx/fdx;
    err = abs(xNext - xn) / abs(xNext);
    if err <= tol
      break
    endif
    plot(iter, err, 'ro');
    title("Newton-Raphson Method");
    xlabel("Iterations")
    ylabel("Error")
  endfor
endfunction
```

Código 5: Método de Newton-Raphson en Octave

1.3.5. Código Python del Método

```
from sympy import *

x = symbols('x')


def newton(expr, x0, tol, maxIter):
    """ Newton Method

    Arguments:
        expr {string} -- is the polinomial expression
        x0 {float} -- is the initial value x_0
        tol {float} -- is the tolerance
```

```python
        maxIter {int} -- is the max amount of iterations
    Returns:
        xn {float} -- is the solution
        err {float} -- is the error
        _iter {int} -- is the amount of iterations
        fx {float} -- is f(xn)

    """
    errReturn = [0, 0, 0, 0]
    try:
        f = sympify(expr)
        fd = diff(expr, x)
        xNext = sympify(x0)
        for _iter in range(0, maxIter):
            xn = N(xNext)
            fdx = fd.subs(x, xn)
            if (fdx == 0):
                print("Error: Division by zero")
                return errReturn
            fx = f.subs(x, xn)
            xNext = xn - fx/fdx
            err = abs(xNext - xn)/abs(xNext)
            if (err <= tol):
                break
        return [N(xn), N(err), _iter, N(fx)]
    except:
        print("There was an error.")
        return errReturn
```

Código 6: Método de Newton-Raphson en Python

1.4. Método de la Secante

1.4.1. Fórmula matemática

$$x_{k+1} = x_k - \left(\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \right) f(x_k), \quad \text{para } k = 1, 2, 3, ... \tag{1.12}$$

1.4.2. Descripción breve del método

1.4.2.1 Valores iniciales

Los valores iniciales para implementar el método son los valores x_0 y x_1 para el inicio de las iteraciones y una criterio de parada (iteraciones máximas, tolerancia, error).

1.4.2.2 Convergencia

Con certeza, únicamente se puede afirmar que que cuando la raíz es simple, el orden de convergencia corresponde a:

$$R = \frac{1 + \sqrt{5}}{2} \approx 1{,}618 \tag{1.13}$$

1.4.2.3 Ventajas

- Requiere un menor esfuerzo en los cálculos ya que es libre de derivadas.
- Incluso siendo libre de derivadas, en general es muy eficiente ya que requiere menos iteraciones que otros métodos.

1.4.2.4 Desventajas

- Al ser una aproximación, su orden de convergencia es menor que el de el Método de Newton-Raphson.
- Hay casos en los que se comporta de manera ineficiente.
- Su convergencia depende de la naturaleza de la función.
- Su convergencia también depende de la exactitud de los valores iniciales. De hecho, para algunas funciones ningún valor inicial funciona.
- Puede suceder la división por cero.

1.4.3. Pseudocódigo del método

```
Newton(f, x_0, x_1, tol, maxIter) -> [xn, err, iter, fx]
  % Secant Method
  % Inputs:
  %   - f is a polinomial expression introduced as a symbolic expression
  %   - x_0 is an initial value
  % - x_1 is an initial value
  %   - tol is the tolerance
  %   - maxIter is the maximum amount of iterations
  % Outputs:
  %   - xn is the solution
  %   - err is the error
  %   - iter is the amount of completed iterations
  %   - fx is f(x)
  Handle the function
  Initialize x_{n-1} as x_0
  Initialize x_n as x_1
  Starting at iter = 1 and ending at maxIter:
    Evaluate the divisor: div
    If div == 0:
      Alert division by zero
      End function
    Evaluate the function f(x)
    Calculate x_{n+1} using the Newton-Raphson's definition
    Calculate the error
    If the tolerance is satisfied:
      End function
    x_{n-1} = x_n
    x_n = x_{n+1}
```

Código 7: Pesudocódigo del Método de la Secante

1.4.4. Código OCTAVE del Método

```
function [xn, err, iter, fx] = secant(f, x0, x1, tol, maxIter)
  % Secant Method
  % Inputs:
  % - f is a polinomial expression introduced as a symbolic expression
  % - x0 is an initial value
  % - x1 is an initial value
  % - tol is the tolerance
  % - maxIter is the maximum amount of iterations
  % Outputs:
  % - xn is the solution
  % - err is the error
  % - iter is the amount of completed iterations
  % - fx is f(x)
  % Errors:
  % - Division by zero
  f = function_handle(f);
  xLast = x0;
  xn = x1;
  for iter=1:maxIter
    div = f(xn) - f(xLast);
    if div == 0
      disp("Error: Division by zero");
      return
    endif
    fx = f(xn);
    xNext = xn - (fx*(xn - xLast))/div;
    err = abs(xNext - xn) / abs(xNext);
    if err <= tol
      return
    endif
    xLast = xn;
    xn = xNext;
  endfor
endfunction
```

Código 8: Método de la Secante en Octave

1.4.5. Código Python del Método

```
from sympy import *

x = symbols('x')


def secant(expr, x0, x1, tol, maxIter):
    """ Secant Method

    Arguments:
        expr {string} -- is the polinomial expression
        x0 {float} -- is the initial value x_0
        x1 {float} -- is the initial value x_1
        tol {float} -- is the tolerance
```

```
        maxIter {int} -- is the max amount of iterations
    Returns:
        xn {float} -- is the solution
        err {float} -- is the error
        _iter {int} -- is the amount of iterations
        fx {float} -- is f(xn)

    """
    errReturn = [0, 0, 0, 0]
    try:
        f = sympify(expr)
        xLast = sympify(x0)
        xn = x1
        for _iter in range(1, maxIter):
            div = f.subs(x, xn) - f.subs(x, xLast)
            if (div == 0):
                print("Error: Division by zero")
                return errReturn
            fx = f.subs(x, xn)
            xNext = xn - (fx*(xn - xLast))/div
            err = abs(xNext - xn)/abs(xNext)
            if (err <= tol):
                break
            xLast = xn
            xn = xNext
        return [N(xn), N(err), _iter, N(fx)]
    except:
        print("There was an error.")
        return errReturn
```

Código 9: Método de la Secante en Python

1.5. Método de la Falsa Posición

1.5.1. Fórmula matemática

El Método de la Falsa Posición[1] utiliza el la ecuación (1.12) (fórmula del Método de la Secante) la cual se reescribe en los términos de las variables del presente método en la ecuación (1.14). Su uso se explica a continuación.

1.5.1.1 Paso 1: Procedimiento inicial

Sea $x_0 = a$ y $x_1 = b$.

1. Verificar que $f(x_0)f(x_1) < 0$.
2. Calcular x_2 utilizando la ecuación (1.12).

1.5.1.2 Paso 2: Uso del Método de Bisección

Como $x_k \in [a_k, b_k]$, entonces el intervalo se divide en dos sub-intervalos, $[a_k, x_k]$ y $[x_k, b_k]$. Se escoge el intervalo donde se garantice que existe un cero de la función f. Es decir:

[1] También se le conoce como el *Método de Interpolación Lineal*

1. Si $f(a_k)f(x_k) < 0$, entonces se escoge el intervalo $[a_k, x_k]$.
2. Si $f(x_k)f(b_k) < 0$, entonces se escoge el intervalo $[x_k, b_k]$.

1.5.1.3 Paso 3: Uso del Método de la Secante

Ahora solo basta con aplicar la fórmula del Método de la Secante:

$$x_{k+1} = x_k - \frac{(x_k - c_k)f(x_k)}{f(x_k) - f(x_k)} \tag{1.14}$$

donde:

$$c_k = \begin{cases} a_k, & \text{si el intervalo seleccionado es } [a_k, x_k] \\ b_k, & \text{si el intervalo seleccionado es } [x_k, b_k] \end{cases} \tag{1.15}$$

Y se repite el proceso en el intervalo seleccionado y la iteración $x_k + 1$.

1.5.2. Descripción breve del método

Según la fórmula explicada anteriormente, se puede afirmar que el Método de la Falsa Posición combina el Método de Bisección y el Método de la Secante con el fin de acelerar la convergencia.

1.5.2.1 Valores iniciales

En desarrollo.

1.5.2.2 Convergencia

En desarrollo.

1.5.2.3 Ventajas

- Es una mejor aproximación que el Método de Bisección ya que toma en consideración la cercanía al cero, en lugar de usar un valor medio.

1.5.2.4 Desventajas

En desarrollo.

1.5.3. Pseudocódigo del método

En desarrollo.

1.5.4. Código OCTAVE del Método

```
function [xn, err, iter, fx] = falsePosition(f, an, bn, tol, maxIter)
  % False Position Method
  % Inputs:
  % - f is a polinomial expression introduced as a symbolic expression
  % - an and bn are [a_n, b_n]
  % - tol is the tolerance
  % - maxIter is the maximum amount of iterations
  % Outputs:
  % - xn is the solution
  % - err is the error
  % - iter is the amount of completed iterations (-1 if IntervalError)
  % - fx is f(x)
  % Errors:
  % - IntervalError: the specified interval does not contain the zero
  % - Division by zero
  f = function_handle(f);
  fa = f(an);
  fb = f(bn);
  xLast = an;
  xn = bn;
  xNext = 0;
  err = 0;
  iter = -1;
  fx = 0;
  if fa * fb > 0
    return
  endif
  for iter=0:maxIter
    div = f(xn) - f(xLast);
    if div == 0
      disp("Error: Division by zero");
      return
    endif
    fx = f(xn);
    xNext = xn - (fx*(xn - xLast))/div;
    err = abs(xNext - xn) / abs(xNext);
    if fx == 0
      a = xn;
      b = xn;
    elseif fb * fx < 0
      an = xn;
      fa = fx;
    else
      bn = xn;
      fb = fx;
    endif
    if err <= tol
      return
    endif
    xLast = xn;
    xn = xNext;
  endfor
```

```
endfunction
```

Código 10: Método de la Falsa Posición en Octave

1.5.5. Código Python del Método

En desarrollo.

1.6. Método del Punto Fijo

1.6.1. Fórmula matemática

En desarrollo.

1.6.2. Descripción breve del método

En desarrollo.

1.6.2.1 Valores iniciales

En desarrollo.

1.6.2.2 Convergencia

En desarrollo.

1.6.2.3 Ventajas

En desarrollo.

1.6.2.4 Desventajas

En desarrollo.

1.6.3. Pseudocódigo del método

En desarrollo.

1.6.4. Código OCTAVE del Método

```
function [xn, fx] = fixedPoint(f, xn, iterMax)
  % Does not validate existence
  % Does not validate uniqueness
  f = function_handle(f);
  for i=0:iterMax
    fx = f(xn);
    xn = fx;
  endfor
endfunction
```

Código 11: Método del Punto Fijo en Octave

1.6.5. Código Python del Método

En desarrollo.

1.7. Método del Müller

1.7.1. Fórmula matemática

En desarrollo.

1.7.2. Descripción breve del método

En desarrollo.

1.7.2.1 Valores iniciales

En desarrollo.

1.7.2.2 Convergencia

En desarrollo.

1.7.2.3 Ventajas

En desarrollo.

1.7.2.4 Desventajas

En desarrollo.

1.7.3. Pseudocódigo del método

En desarrollo.

1.7.4. Código OCTAVE del Método

```
function [x2, err, iter, fx] = muller(f, x0, x1, x2, tol, maxIter)
  % Muller Method
  % Inputs:
  % - f is a polinomial expression introduced as a symbolic expression
  % - x0 is an initial value
  % - x1 is an initial value
  % - x2 is an initial value
  % - tol is the tolerance
  % - maxIter is the maximum amount of iterations
  % Outputs:
  % - x2 is the solution
  % - err is the error
  % - iter is the amount of completed iterations
  % - fx is f(x)
```

```
  % Errors:
  % - Division by zero
  % - No real solution: no real solution found for ax**2+bx+c
  f = function_handle(f);
  for iter=1:maxIter
    div = (x0 - x1)*(x0 - x2)*(x1 - x2);
    if div == 0
      disp("Error: Division by zero");
      return
    endif
    a = ((x0 - x2)*(f(x1)-f(x2))-(x1-x2)*(f(x0)-f(x2)))/div;
    b = ((x0 - x2)^2*(f(x1)-f(x2))-(x1-x2)^2*(f(x0)-f(x2)))/div;
    c = f(x2);
    disc = b^2 - 4*a*c;
    if disc < 0
      disp("Error: No real solution");
      return
    endif
    div = b + sign(b)*sqrt(disc);
    xn = x2 - 2*c/div;
    fx = f(xn);
    err = abs(xn - x2)/abs(xn);
    if err <= tol
      return
    endif
    x0Dist = abs(xn - x0);
    x1Dist = abs(xn - x1);
    x2Dist = abs(xn - x2);
    if x0Dist > x2Dist && x0Dist > x1Dist
      x0 = x2;
    elseif x1Dist > x2Dist && x1Dist > x0Dist
      x1 = x2;
    endif
    x2 = xn;
  endfor
endfunction
```

Código 12: Método del Müller en Octave

1.7.5. Código Python del Método

En desarrollo.

2. Métodos Iterativos para Optimización

2.1. Método del Descenso Coordinado

2.1.1. Fórmula matemática

En desarrollo.

2.1.2. Descripción breve del método

En desarrollo.

2.1.2.1 Valores iniciales

En desarrollo.

2.1.2.2 Convergencia

En desarrollo.

2.1.2.3 Ventajas

En desarrollo.

2.1.2.4 Desventajas

En desarrollo.

2.1.3. Pseudocódigo del método

En desarrollo.

2.1.4. Código OCTAVE del Método

```
function [xn, fxn] = coordinateDescentXY(f, xn, maxIter)
  % Argument error: the amount of arguments of the function does not match
  % with cell length
  f = function_handle(f);
  n = length(xn);
  if  n != 2 || n != nargin(f)
    disp("Argument error");
    return
  endif
  for k=1:maxIter
    % Gauss-Seidel
    fx = f(sym('x'), xn(2));
    fx = function_handle(fx);
    xn(1) = fminsearch(fx, 0);
    fy = f(xn(1), sym('y'));
    fy = function_handle(fy);
    xn(2) = fminsearch(fy, 0);
    fxn = f(xn(1), xn(2));
  endfor
  return
endfunction
```

Código 13: Método del Descenso Coordinado (2 variables) en Octave

```
function [xn, fxn] = coordinateDescentXYZ(f, xn, maxIter)
  % Argument error: the amount of arguments of the function does not match
  % with cell length
  f = function_handle(f);
  n = length(xn);
  if  n != 3 || n != nargin(f)
    disp("Argument error");
    return
  endif
  for k=1:maxIter
    % Gauss-Seidel
    fx = f(sym('x'), xn(2), xn(3));
    fx = function_handle(fx);
    xn(1) = fminsearch(fx, 0);
    fy = f(xn(1), sym('y'), xn(3));
    fy = function_handle(fy);
    xn(2) = fminsearch(fy, 0);
    fz = f(xn(1), xn(2), sym('z'));
    fz = function_handle(fz);
    xn(3) = fminsearch(fz, 0);
    fxn = f(xn(1), xn(2), xn(3));
  endfor
  return
endfunction
```

Código 14: Método del Descenso Coordinado (3 variables) en Octave

2.1.5. Código Python del Método

En desarrollo.

2.2. Método del Gradiente Conjugado no Lineal

2.2.1. Fórmula matemática

En desarrollo.

2.2.2. Descripción breve del método

En desarrollo.

2.2.2.1 Valores iniciales

En desarrollo.

2.2.2.2 Convergencia

En desarrollo.

2.2.2.3 Ventajas

En desarrollo.

2.2.2.4 Desventajas

En desarrollo.

2.2.3. Pseudocódigo del método

En desarrollo.

2.2.4. Código OCTAVE del Método

```
function [xk, gNorm] = nonLinearConjugateGradient(f, xk, maxIter)
  % sigma = rand;
  sigma = 0.5;
  gf = gradient(f);
  f = function_handle(f);
  gf = function_handle(gf);
  gk = gf(num2cell(xk){:});                    % It is already a transposed matrix
  dk = -gk;
  for iter=0:maxIter
    ak = 1;
    while !(f(num2cell(xk + ak*dk){:}) - f(num2cell(xk){:}) <= sigma*ak*gk
   '*dk)
      ak = ak/2;
    endwhile
    xNext = xk + ak*dk;
    gNext = gf(num2cell(xNext){:});            % It is already a transposed matrix
    bk = norm(gNext)^2/norm(gk)^2;
    dNext = -gNext + bk*dk;
    gNorm = norm(gk);
    xk = xNext;
    gk = gNext;
    dk = dNext;
  endfor
endfunction
```

Código 15: Método del Gradiente Conjugado no Lineal en Octave

2.2.5. Código Python del Método

En desarrollo.

3. Sistemas de Ecuaciones Lineales: Métodos Directos

3.1. Método de sustitución hacia atrás

3.1.1. Fórmula matemática

En desarrollo.

3.1.2. Descripción breve del método

En desarrollo.

3.1.2.1 Valores iniciales

En desarrollo.

3.1.2.2 Convergencia

En desarrollo.

3.1.2.3 Ventajas

En desarrollo.

3.1.2.4 Desventajas

En desarrollo.

3.1.3. Pseudocódigo del método

En desarrollo.

3.1.4. Código OCTAVE del Método

```
function X = backSubstitution(A, B)
  % Back Substitution Method
  % Solves AX=B
  % Inputs:
  % - A is a NxN upper triangular matrix
  % - B is a Nx1 matrix
  % Outputs:
  % - X is the solution matrix
  n = length(B);
  X = zeros(n, 1);
  X(n) = B(n)/A(n, n);
  for k=n-1:-1:1
    div = A(k, k);
    if div != 0
      X(k) = (B(k) - A(k, k+1:n)*X(k+1:n))/A(k, k);
    else
      disp("Error: division by zero");
    endif
  endfor
endfunction
```

Código 16: Método de sustitución hacia atrás en Octave

3.1.5. Código Python del Método

En desarrollo.

3.2. Método de sustitución hacia adelante

3.2.1. Fórmula matemática

En desarrollo.

3.2.2. Descripción breve del método

En desarrollo.

3.2.2.1 Valores iniciales

En desarrollo.

3.2.2.2 Convergencia

En desarrollo.

3.2.2.3 Ventajas

En desarrollo.

3.2.2.4 Desventajas

En desarrollo.

3.2.3. Pseudocódigo del método

En desarrollo.

3.2.4. Código OCTAVE del Método

```
function X = forwardSubstitution(A, B)
  % Forward Substitution Method
  % Solves AX=B
  % Inputs:
  % - A is a NxN lower triangular matrix
  % - B is a Nx1 matrix
  % Outputs:
  % - X is the solution matrix
  n = length(B);
  X = zeros(n, 1);
  X(1) = B(1)/A(1, 1);
  for k=2:n
    div = A(k, k);
    if div != 0
      X(k) = (B(k) - A(k, 1:k-1)*X(1:k-1))/A(k, k);
    else
      disp("Error: division by zero");
    endif
```

```
  endfor
endfunction
```

Código 17: Método de sustitución hacia adelante en Octave

3.2.5. Código Python del Método

En desarrollo.

3.3. Método de Eliminación Gaussiana

3.3.1. Fórmula matemática

En desarrollo.

3.3.2. Descripción breve del método

En desarrollo.

3.3.2.1 Valores iniciales

En desarrollo.

3.3.2.2 Convergencia

En desarrollo.

3.3.2.3 Ventajas

En desarrollo.

3.3.2.4 Desventajas

En desarrollo.

3.3.3. Pseudocódigo del método

En desarrollo.

3.3.4. Código OCTAVE del Método

```
function X = gaussianElimination(A, B)
  n = length(A);
  X = [A, B];
  % For each row of augmented matrix
  for i=1:n
    pivot = X(i, i);
    pivotRow = X(i, :);
    % Multipliers' vector
    M = zeros(1, n - i);
```

```
    m = length(M);
    % Get each row multiplier
    for k=1:m
      M(k) = X(i + k, i) / pivot;
    endfor
    % Modify each row
    for k=1:m
      X(i + k, :) = X(i + k, :) - pivotRow*M(k);
    endfor
  endfor
  X = backSubstitution(X(1:n, 1:n), X(:,n+1));
endfunction
```

Código 18: Método de Eliminación Gaussiana en Octave

```
function X = backSubstitution(A, B)
  % Back Substitution Method
  % Solves AX=B
  % Inputs:
  % - A is a NxN upper triangular matrix
  % - B is a Nx1 matrix
  % Outputs:
  % - X is the solution matrix
  n = length(B);
  X = zeros(n, 1);
  X(n) = B(n)/A(n, n);
  for k=n-1:-1:1
    div = A(k, k);
    if div != 0
      X(k) = (B(k) - A(k, k+1:n)*X(k+1:n))/A(k, k);
    else
      disp("Error: division by zero");
    endif
  endfor
endfunction
```

Código 19: Método de sustitución hacia atrás

3.3.5. Código Python del Método

En desarrollo.

3.4. Método de Factorización LU

3.4.1. Fórmula matemática

En desarrollo.

3.4.2. Descripción breve del método

En desarrollo.

3.4.2.1 Valores iniciales

En desarrollo.

3.4.2.2 Convergencia

En desarrollo.

3.4.2.3 Ventajas

En desarrollo.

3.4.2.4 Desventajas

En desarrollo.

3.4.3. Pseudocódigo del método

En desarrollo.

3.4.4. Código OCTAVE del Método

```
function [L, U] = lu(X)
  n = length(X);
  L = eye(n);
  U = X;
  % For each row of matrix X
  for i=1:n
    pivot = U(i, i);
    pivotRow = U(i, :);
    % Multipliers' vector
    M = zeros(1, n - i);
    m = length(M);
    % Get each row multiplier
    for k=1:m
      M(k) = U(i + k, i) / pivot;
    endfor
    % Modify each row and each L subcolumn
    for k=1:m
      U(i + k, :) = U(i + k, :) - pivotRow*M(k);
      L(i + k, i) = M(k);
    endfor
  endfor
endfunction
```

Código 20: Función auxiliar para el Método de Factorización LU en Octave

```
function [X] = luDecomposition(A, B)
  [L, U] = lu(A);
  Y = forwardSubstitution(L, B);
  X = backSubstitution(U, Y);
```

```
endfunction
```

Código 21: Método de Factorización LU en Octave

3.4.5. Código Python del Método

En desarrollo.

3.5. Método de Factorización de Cholesky

3.5.1. Fórmula matemática

En desarrollo.

3.5.2. Descripción breve del método

En desarrollo.

3.5.2.1 Valores iniciales

En desarrollo.

3.5.2.2 Convergencia

En desarrollo.

3.5.2.3 Ventajas

En desarrollo.

3.5.2.4 Desventajas

En desarrollo.

3.5.3. Pseudocódigo del método

```
function L = cholesky(A)
  n = length(A);
  L = zeros(n);
  for i=1:n
    for j=1:i
      if j==i
        sum = 0;
        for k=1:j-1
          sum = sum + L(j, k)^2;
        endfor
        L(j,j) = sqrt(A(j, j) - sum);
      else
        sum = 0;
        for k=1:j-1
          sum = sum + L(i, k)*L(j, k);
```

```
        endfor
        L(i, j) = (1/L(j, j))*(A(i, j) - sum);
      endif
    endfor
  endfor
endfunction
```

Código 22: Función auxiliar para el Método de Factorización de Cholesky en Octave

```
function X = choleskyDecomposition(A, B)
  L = cholesky(A);
  Y = forwardSubstitution(L, B);
  X = backSubstitution(L', Y);
endfunction
```

Código 23: Método de Factorización de Cholesky en Octave

3.5.4. Código OCTAVE del Método

En desarrollo.

3.5.5. Código Python del Método

En desarrollo.

4. Sistemas de Ecuaciones Lineales: Métodos Iterativos

5. Interpolación

6. Regresión Numérica

7. Diferenciación Numérica

8. Integración Numérica

9. Ecuaciones Diferenciales Numéricas

10. Método Iterativos para Calcular Valores Propios