Math Methods Web Calculator 2023-1

31 MAYO

UNIVERSIDAD EAFIT

Created by : Miguel Angel Cabrera Osorio
Jairo Andres Ruiz Machado
Gustavo Adolfo López Garcia

Nombre del logotipo

Project description

The project consists in the construction of a web calculator of mathematical methods. Which were seen in class, in order to evaluate the understanding of these methods and the ability to apply them.

The calculator was developed in Python and tools such as fastApi, javascripr, React and Java were used. This is to facilitate its development. The calculator allowed to calculate methods such as:

EQUATIONS OF ONE VARIABLE:

1. Busquedas incrementales
2. Bisección
3. Regla Falsa
4. Punto Fijo
5. Newton
6. Secante
7. Raices Multiples

ECUATION SYSTEMS:

8. Eliminación Gaussiana Sencilla
9. Eliminación Gaussiana con pivoteo parcial
10. Eliminación Gaussiana con pivoteo total
11. Factorización LU con eliminación Gaussiana
12. Doolittle
13. Cholesky
14. Crout
15. Gauss-Seidel
16. Jacobi

INTERPOLATION:

17. Vandermonde
18. Splines

User's Guide

User Manual

SetUp

clone the repository un your PC = git clone https://github.com/Miguelco23/Numerical-analysis

For Backend

Install Python = https://www.python.org/downloads/

Create a virtual enviroment (Optional)

enter to Backend foler = cd proyect_folder/Backend

Create the virtual enviroment = python -m venv env_name

Activate the virtual enviroment = source env_name/Scripts/activate

Install the dependences

Numpy = pip install numpy

FastAPI = pip install fastapi

Uvicorn = pip install uvicorn

Run API = uvicorn main:app --reload

For frontend (In a new terminal)

Install Node.js = https://nodejs.org

Enter to frontend foler = cd proyect_folder/Frontend

Run the React server = npm start

App Use

In your favorite web browser go to http://localhost:3000/

In the home page you'll see the methods divided in three categories:

One Variable equations, equation systems and Interpolation.

Each of these contains from 2 to 9 buttons with some method name

If you want to use any method, you just have to click the button with it's name.

Once you click the button, will open a modal form with the parammeters necesary for each method

You'll have to fill all the parammeters to get a correct answer

With all the inputs filled you just have to click the "calculate" button to get a response.

And finally when you want to change the method or just go back to the homepage. You just have to click the "close" button. Or just click out of the modal

Pseudocode and method code

Bisection

```
Función biseccion(f, a, b, tol)
    # Evaluar la cadena de texto como una función
    Función evaluar_funcion(x)
        Devolver eval(f)
    Fin Función

    Si evaluar_funcion(a) * evaluar_funcion(b) >= 0 entonces
        Imprimir "Error: La función no cambia de signo en el intervalo dado."
        Devolver Nulo
    Sino
        c = (a + b) / 2
        Mientras abs(evaluar_funcion(c)) > tol hacer
            Si evaluar_funcion(a) * evaluar_funcion(c) < 0 entonces
                b = c
            Sino
                a = c
            Fin Si
            c = (a + b) / 2
        Fin Mientras
        Devolver c
    Fin Si
Fin Función

# Cadena de texto que representa la función
f = "x**2 - 2"

# Intervalo y tolerancia
a = 0
b = 2
tol = 1e-6

raiz = biseccion(f, a, b, tol)
Imprimir "La raíz de la función es:", raiz
```

```
def biseccion(f, a, b, tol):

    # Evaluar la cadena de texto como una función
    def evaluar_funcion(x):
        return eval(f)

    if evaluar_funcion(a) * evaluar_funcion(b) >= 0:
        print("Error: La función no cambia de signo en el intervalo dado.")
        return None
    else:
        c = (a + b) / 2
        while abs(evaluar_funcion(c)) > tol:
            if evaluar_funcion(a) * evaluar_funcion(c) < 0:
                b = c
            else:
                a = c
            c = (a + b) / 2
        return c

# Cadena de texto que representa la función
f = "x**2 - 2"

# Intervalo y tolerancia
a = 0
b = 2
tol = 1e-6

# raiz = biseccion(f, a, b, tol)
# print("La raíz de la función es:", raiz)
```

Incremental Searches

```
input: f,x0,h,nmax

xant = x0
fant = f(xant)
xact = xant+h
fact = f(xact)

itera hasta nmax:
    si fant * fact <0:
        termine, raiz entre xant y xact
    xant = xact
    xact= xant+h
    fant = f(ant)
    fact = f(act)
```

```
from math import *
import numpy as np

def busqueda_inc(f, x0, h, nmax):
    xant = x0
    flambda = lambda x: eval(f)
    fant = flambda(xant)
    xact = xant+h
    fact = flambda(xact)
    for i in range(nmax):
        print(f"{fant} {fact}")
        if fant * fact <0:
            # print(f"hay raiz entre {xant} y {xact}")
            # print(f"iters = {i}")
            return(f"Root is between {xant} and {xact}. output with {i} iterations") # Retorna dos limites de el intervalo y las iteraciones
        xant = xact
        fant = fact
        xact = xant+h
        fact = flambda(xact)


# Ejemplo de como utilizar
# busqueda_inc("(x**3)+3*x+2",-2,0.0075,1000)
```

Cholesky

```
Input: A, b
n = tamaño de A
Inicializar L, como Matriz de orden nxn

for j = 0 hasta n:
    for i = 0 hasta j+1:
        si el elemento es la diagonal, se calcula usando raiz cuadrada con la diagonal relacioanda en A

    for i = j hasta n:

        calcular el producto punto de una columan de U con una fila de L, despejar al elemento de A correspondiente

hacer sub progresiva con Lz=b

hacer sub regresiva con L^T*x=z

retornar x
```

```python
import numpy as np

def cholesky(A, b):
    A = np.array(A)
    b = np.array(b)
    n = len(A)
    L = np.zeros((n, n))
    x = np.zeros(n)

    for i in range(n):
        for j in range(i + 1):
            if i == j:
                sum_1 = sum(L[i][k] ** 2 for k in range(j))
                L[i][i] = np.sqrt(A[i][i] - sum_1)
            else:
                sum_2 = sum(L[i][k] * L[j][k] for k in range(j))
                L[i][j] = (A[i][j] - sum_2) / L[j][j]

    # sub progresiva
    y = np.zeros(n)
    for i in range(n):
        y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i]

    # L^T * x = y sub regresiva
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - np.dot(L[i + 1:, i], x[i + 1:])) / L[i, i]

    return np.array2string(x)

# A = [[1,-1,1],
#      [-1,5,-5],
#      [1,-5,6]]

# b = [2,-6,9]
# print(cholesky(A,b)
```

Crout

```
Input: A, b
n = tamaño de A
Inicializar L,U Como matrizes de orden nxn

for j = 0 hasta n:
    Definir la diagonal de U en unos

    for i = 0 hasta j+1:
        calcular el producto punto de una columan de U con una fila de U, despejar al elemento de A correspondiente

    for i = j hasta n:

        calcular el producto punto de una columan de U con una fila de L, despejar al elemento de A correspondiente

hacer sub progresiva con Lz=b

hacer sub regresiva con Ux=z

retornar x
```

```python
import numpy as np

def crout(A, b):
    A = np.array(A)
    b = np.array(b)
    n = len(A)
    L = np.zeros((n, n))
    U = np.zeros((n, n))
    x = np.zeros(n)

    for j in range(n):
        U[j][j] = 1.0

        for i in range(j+1):
            sum_1 = sum(L[i][k] * U[k][j] for k in range(i))
            U[i][j] = A[i][j] - sum_1

        for i in range(j, n):
            sum_2 = sum(L[i][k] * U[k][j] for k in range(j))
            L[i][j] = (A[i][j] - sum_2) / U[j][j]

    # sub progresiva
    y = np.zeros(n)
    for i in range(n):
        y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i]

    # sub regresiva
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i + 1:], x[i + 1:])) / U[i, i]

    return np.array2string(x)


# A =             [[36,3,-4,5],
#                 [5,-45,10,-2],
#                 [6,8,57,5],
#                 [2,3,-8,-42]]

# b = [-20,69,96,-32]

# print(crout(A,b))
```

Doolittle

```
Input: A, b
n = tamaño de A
Inicializar L,U Como matrizes de orden nxn

for j = 0 hasta n:
    Definir la diagonal de L en unos

    for i = 0 hasta j+1:
        calcular el producto punto de una columan de U con una fila de U, despejar al elemento de A correspondiente

    for i = j hasta n:

        calcular el producto punto de una columan de U con una fila de L, despejar al elemento de A correspondiente

hacer sub progresiva con Lz=b

hacer sub regresiva con Ux=z

retornar x
```

```python
import numpy as np

def doolittle(A, b):
    n = len(A)
    L = np.zeros((n, n))
    U = np.zeros((n, n))
    x = np.zeros(n)

    for i in range(n):
        L[i][i] = 1.0

        for j in range(i, n):
            sum_1 = sum(L[i][k] * U[k][j] for k in range(i))
            U[i][j] = A[i][j] - sum_1

        for j in range(i + 1, n):
            sum_2 = sum(L[j][k] * U[k][i] for k in range(i))
            L[j][i] = (A[j][i] - sum_2) / U[i][i]

    # Substitucion progresiva
    y = np.zeros(n)
    for i in range(n):
        y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i]

    # sub regresiva
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i + 1:], x[i + 1:])) / U[i, i]

    return np.array2string(x)


# A = np.array([[36,3,-4,5],
#               [5,-45,10,-2],
#               [6,8,57,5],
#               [2,3,-8,-42]])

# b = np.array([-20,69,96,-32])
# print(doolittle(A,b))
```

Partial Gaussian

```
Función GausPar(A_, b_)
    A = convertir_a_matriz(A_)
    b = convertir_a_matriz(b_)
    n = tamaño_fila(A)
    M = concatenar((A, reshape(b, (n, 1))), eje=1)

    Para i en rango(n-1) hacer
        aux0, aux = máximo(abs(M[i+1:n, i])), argmáximo(abs(M[i+1:n, i]))
        Si aux0 > abs(M[i, i]) entonces
            aux2 = copiar(M[i+aux, i:n+1])
            M[aux+i, i:n+1] = M[i, i:n+1]
            M[i, i:n+1] = aux2
        Fin Si

        Para j en rango(i+1, n) hacer
            Si M[j, i] != 0 entonces
                M[j, i:n+1] = M[j, i:n+1] - (M[j, i]/M[i, i]) * M[i, i:n+1]
            Fin Si
        Fin Para
    Fin Para

    x = SustitucionRegresiva(M)
    Devolver {"x": convertir_a_cadena(x)}
Fin Función

Función SustitucionRegresiva(M)
    n = tamaño_fila(M)
    x = crear_arreglo(n)
    Para i en rango(n-1, -1, -1) hacer
        x[i] = (M[i, n] - producto_punto(M[i, i+1:n], x[i+1:n])) / M[i, i]
    Fin Para
    Devolver x
Fin Función
```

```
import numpy as np

def GausPar(A_, b_):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    n = A.shape[0]
    M = np.concatenate((A, b.reshape(n, 1)), axis=1)

    # Reducción del sistema
    for i in range(n-1):
        # Cambio de filas
        aux0, aux = np.max(np.abs(M[i+1:n, i])), np.argmax(np.abs(M[i+1:n, i]))
        if aux0 > np.abs(M[i, i]):
            aux2 = M[i+aux, i:n+1].copy()
            M[aux+i, i:n+1] = M[i, i:n+1]
            M[i, i:n+1] = aux2

        for j in range(i+1, n):
            if M[j, i] != 0:
                M[j, i:n+1] = M[j, i:n+1] - (M[j, i]/M[i, i]) * M[i, i:n+1]

    # Entrega de resultados
    x = SustitucionRegresiva(M)  # Sustitución regresiva
    return {"x": np.array2string(x)}

def SustitucionRegresiva(M):
    n = M.shape[0]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (M[i, n] - np.dot(M[i, i+1:n], x[i+1:n])) / M[i, i]
    return x
```

Gauss-Seidel

```
Función GausSeidel(A_, b_, x0_, tol, Nmax)
    A = convertir_a_matriz(A_)
    b = convertir_a_matriz(b_)
    x0 = convertir_a_matriz(x0_)
    D = diag(diagonal(A))
    L = -tril(A) + D
    U = -triu(A) + D
    T = inv(D - L) * U
    C = inv(D - L) * b
    xant = copiar(x0)
    E = 1000
    cont = 0

    Mientras E > tol y cont < Nmax hacer
        xact = T * xant + C
        E = norma(xant - xact)
        xant = xact
        cont = cont + 1
    Fin Mientras

    x = xact
    iteraciones = cont
    err = E

    Devolver {"x": x, "iterations": iteraciones, "Error": err}
Fin Función
```

```
import numpy as np

def GausSeidel(A_, b_, x0_, tol, Nmax):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    x0 = np.array(x0_)
    D = np.diag(np.diag(A))
    L = -np.tril(A) + D
    U = -np.triu(A) + D
    T = np.linalg.inv(D - L) @ U
    C = np.linalg.inv(D - L) @ b
    xant = np.array(x0)
    E = 1000
    cont = 0

    # Ciclo
    while E > tol and cont < Nmax:
        xact = T @ xant + C
        E = np.linalg.norm(xant - xact)
        xant = xact
        cont += 1

    # Entrega de resultados
    x = xact
    iterations = cont
    err = E

    return ({"x":x,"iterations":iterations,"Error":err})
```

Simple Gauss

```
Función GausSimple(A_, b_)
    A = convertir_a_matriz(A_)
    b = convertir_a_matriz(b_)
    n = tamaño_fila(A)
    M = concatenar((A, reshape(b, (n, 1))), eje=1)

    Para i en rango(n-1) hacer
        Para j en rango(i+1, n) hacer
            Si M[j, i] != 0 entonces
                M[j, i:n+1] = M[j, i:n+1] - (M[j, i]/M[i, i]) * M[i, i:n+1]
            Fin Si
        Fin Para
    Fin Para

    x = SustitucionRegresiva(M)
    Devolver convertir_a_cadena(x)
Fin Función

Función SustitucionRegresiva(M)
    n = tamaño_fila(M)
    x = crear_arreglo(n)
    Para i en rango(n-1, -1, -1) hacer
        x[i] = (M[i, n] - producto_punto(M[i, i+1:n], x[i+1:n])) / M[i, i]
    Fin Para
    Devolver x
Fin Función
```

```
import numpy as np

def GausSimple(A_, b_):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    n = A.shape[0]
    M = np.concatenate((A, b.reshape(n, 1)), axis=1)

    # Reducción del sistema
    for i in range(n-1):
        for j in range(i+1, n):
            if M[j, i] != 0:
                M[j, i:n+1] = M[j, i:n+1] - (M[j, i]/M[i, i]) * M[i, i:n+1]

    # Sustitución regresiva
    x = SustitucionRegresiva(M)

    # Entrega de resultados
    return np.array2string(x)

def SustitucionRegresiva(M):
    n = M.shape[0]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (M[i, n] - np.dot(M[i, i+1:n], x[i+1:n])) / M[i, i]
    return x
```

Total Gauss

```
Función GausTotal(A_, b_)
  A = convertir_a_matriz(A_)
  b = convertir_a_matriz(b_)
  n = tamaño_fila(A)
  M = concatenar((A, b), eje=1)
  cambi = []

  Para i en rango(n - 1) hacer
    a, b = desempaquetar(maximo_absoluto(M[i:, i:], (n - i, n - i)))

    Si b[0] + i != i entonces
      agregar_a(cambi, [i, b[0] + i])
      aux2 = copiar(M[:, b[0] + i])
      M[:, b[0] + i] = M[:, i]
      M[:, i] = aux2
    Fin Si

    Si a[0] + i != i entonces
      aux2 = copiar(M[i + a[0] - 1, i:n + 1])
      M[a[0] + i - 1, i:n + 1] = M[i, i:n + 1]
      M[i, i:n + 1] = aux2
    Fin Si

    Para j en rango(i + 1, n) hacer
      Si M[j, i] != 0 entonces
        M[j, i:n + 1] = M[j, i:n + 1] - (M[j, i] / M[i, i]) * M[i, i:n + 1]
      Fin Si
    Fin Para
  Fin Para

  x = SustitucionRegresiva(M)

  Para i en rango(longitud(cambi) - 1, -1, -1) hacer
    aux = x[cambi[i][0]]
    x[cambi[i][0]] = x[cambi[i][1]]
    x[cambi[i][1]] = aux
  Fin Para

  Devolver convertir_a_cadena(x)
Fin Función

Función SustitucionRegresiva(M)
  n = tamaño_fila(M)
  x = crear_arreglo(n)
  Para i en rango(n-1, -1, -1) hacer
    x[i] = (M[i, n] - producto_punto(M[i, i+1:n], x[i+1:n])) / M[i, i]
  Fin Para
  Devolver x
Fin Función
```

```
import numpy as np

def GausTotal(A_, b_):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    n = A.shape[0]
    M = np.hstack((A, b.reshape(n, 1)))
    cambi = []

    # Reducimos el sistema
    for i in range(n - 1):
        a, b = np.unravel_index(np.argmax(np.abs(M[i:, i:])), (n - i, n - i))

        # Cambio de columna
        if b + i != i:
            cambi.append([i, b + i])
            M[:, [b + i, i]] = M[:, [i, b + i]]

        # Cambio de filas
        if a + i != i:
            M[[a + i, i], i:] = M[[i, a + i], i:]

        for j in range(i + 1, n):
            if M[j, i] != 0:
                M[j, i:] = M[j, i:] - (M[j, i] / M[i, i]) * M[i, i:]

    # Entrega de resultados
    x = SustitucionRegresiva(M)  # Sustitución regresiva

    # Reordenamos el vector solución
    for i in range(len(cambi) - 1, -1, -1):
        aux = x[cambi[i][0]]
        x[cambi[i][0]] = x[cambi[i][1]]
        x[cambi[i][1]] = aux

    return np.array2string(x)

def SustitucionRegresiva(M):
    n = M.shape[0]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (M[i, n] - np.dot(M[i, i+1:n], x[i+1:n])) / M[i, i]
    return x
```

Jacobi

```
Función Jacobi(A_, b_, x0_, tol, Nmax)
    A = convertir_a_matriz(A_)
    b = convertir_a_matriz(b_)
    x0 = convertir_a_matriz(x0_)
    D = diag(diagonal(A))
    L = -tril(A) + D
    U = -triu(A) + D
    T = inv(D) * (L + U)
    C = inv(D) * b
    xant = copiar(x0)
    E = 1000
    cont = 0

    Mientras E > tol y cont < Nmax hacer
        xact = T * xant + C
        E = norma(xant - xact)
        xant = xact
        cont = cont + 1
    Fin Mientras

    x = xant
    iteraciones = cont
    err = E

    Devolver {"x": x, "iterations": iteraciones, "Error": err}
Fin Función
```

```
import numpy as np

def Jacobi(A_, b_, x0_, tol, Nmax):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    x0 = np.array(x0_)
    D = np.diag(np.diag(A))
    L = -np.tril(A) + D
    U = -np.triu(A) + D
    T = np.linalg.inv(D).dot(L + U)
    C = np.linalg.inv(D).dot(b)
    xant = np.array(x0)
    E = 1000
    cont = 0

    # Ciclo
    while E > tol and cont < Nmax:
        xact = T.dot(xant) + C
        E = np.linalg.norm(xant - xact)
        xant = xact
        cont += 1

    # Entrega de resultados
    x = xant
    iterations = cont
    err = E

    return ({"x":np.array2string(x),"iterations":iterations,"Error":err})
```

Simple LU

```
Función LUSimple(A_, b_)
    A = convertir_a_matriz(A_)
    b = convertir_a_matriz(b_)
    n = tamaño_fila(A)
    L = matriz_identidad(n)
    U = crear_matriz_ceros(n, n)
    M = copiar(A)

    Para i en rango(n-1) hacer
        Para j en rango(i+1, n) hacer
            Si M[j, i] != 0 entonces
                L[j, i] = M[j, i] / M[i, i]
                M[j, i:n] = M[j, i:n] - (M[j, i] / M[i, i]) * M[i, i:n]
            Fin Si
        Fin Para
        U[i, i:n] = M[i, i:n]
        U[i+1, i+1:n] = M[i+1, i+1:n]
    Fin Para
    U[n-1, n-1] = M[n-1, n-1]

    z = SustitucionProgresiva(concatenar((L, concatenar(b.reshape(n, 1)), eje=1)))
    x = SustitucionRegresiva(concatenar((U, concatenar(z.reshape(n, 1)), eje=1)))

    Devolver {"x": convertir_a_cadena(x), "L": convertir_a_cadena(L), "U": convertir_a_cadena(U)}
Fin Función

Función SustitucionProgresiva(M)
    n = tamaño_fila(M)
    z = crear_arreglo(n)
    Para i en rango(n) hacer
        z[i] = (M[i, n] - producto_punto(M[i, :i], z[:i])) / M[i, i]
    Fin Para
    Devolver z
Fin Función

Función SustitucionRegresiva(M)
    n = tamaño_fila(M)
    x = crear_arreglo(n)
    Para i en rango(n-1, -1, -1) hacer
        x[i] = (M[i, n] - producto_punto(M[i, i+1:n], x[i+1:n])) / M[i, i]
    Fin Para
    Devolver x
Fin Función
```

```
import numpy as np

def LUSimple(A_, b_):
    # Inicialización
    A=np.array(A_)
    b = np.array(b_)
    n = A.shape[0]
    L = np.eye(n)
    U = np.zeros((n, n))
    M = A.copy()

    # Factorización
    for i in range(n-1):
        for j in range(i+1, n):
            if M[j, i] != 0:
                L[j, i] = M[j, i] / M[i, i]
                M[j, i:n] = M[j, i:n] - (M[j, i] / M[i, i]) * M[i, i:n]
        U[i, i:n] = M[i, i:n]
        U[i+1, i+1:n] = M[i+1, i+1:n]
    U[n-1, n-1] = M[n-1, n-1]

    # Entrega de resultados
    z = SustitucionProgresiva(np.concatenate((L, b.reshape(n, 1)), axis=1))
    x = SustitucionRegresiva(np.concatenate((U, z.reshape(n, 1)), axis=1))
    return {"x": np.array2string(x),"L": np.array2string(L), "U":np.array2string(U)}

def SustitucionProgresiva(M):
    n = M.shape[0]
    z = np.zeros(n)
    for i in range(n):
        z[i] = (M[i, n] - np.dot(M[i, :i], z[:i])) / M[i, i]
    return z

def SustitucionRegresiva(M):
    n = M.shape[0]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (M[i, n] - np.dot(M[i, i+1:n], x[i+1:n])) / M[i, i]
    return (x)
```

Newton

```
Función Newton(f, derf, x0, tol, Nmax)
    xant = x0
    flambda = lambda x: eval(f)
    fant = flambda(xant)
    E = 1000
    cont = 0
    derflambda = lambda x: eval(derf)

    Mientras E > tol y cont < Nmax hacer
        xact = xant - fant / derflambda(xant)
        fact = flambda(xact)
        E = abs(xact - xant)
        cont = cont + 1
        xant = xact
        fant = fact

        Si E < tol entonces
            x = xact
            ite = cont
            err = E
            Devolver ("La función tiene una raíz en", x)
        Fin Si
    Fin Mientras

    Devolver ("No tiene solución")
Fin Función
```

```
from math import*
import numpy as np

def Newton(f, derf, x0, tol, Nmax):
    xant = x0
    flambda = lambda x: eval(f)
    fant = flambda(xant)
    E = 1000
    cont = 0
    derflambda = lambda x: eval(derf)
    while E > tol and cont < Nmax:
      xact = xant-fant/(derflambda(xant))
      fact = flambda(xact)
      E = abs(xact-xant)
      cont = cont+1
      xant = xact
      fant = fact
      if E < tol:
        x = xact
        ite = cont
        err = E
        return ("La función tiene una raíz en, ", x)
      else:
        return ("No tiene solucion")
```

Fixed Point

```
entradas: f, g, x0, tol, nmax

se inicializa error como un numero muy grande
iter = 0

mientras que el error sea mayor a la tol y iter menor a nmax:

    xact = g(xant)
    err = abs(xant-xact)
    iter+=1
    xant = xact

retornar xact, iter, err
```

```
from math import *
import numpy as np

def puntoFijo(f, g, x0, tol, nmax):
    iter = 0
    Err = 9999
    xant = x0
    flambda = lambda x: eval(f)
    glambda = lambda x: eval(g)

    while Err> tol and iter< nmax:
        xact = glambda(xant)
        Err  = abs(xant-xact)
        print(Err)
        iter+=1
        xant = xact


    print(f"x= {xact}")
    print(f"iters = {iter}")
    print(f"Error = {Err}")

    return({"x":xact,"iters":iter,"Error":Err})

# Ejemplo de como utilizar
# puntofijo("(e**-x)-x","(e**-x)",1,5*10**(-6),12)
```

Multiple Roots

```
Función RaicesMultiples(f, df, d2f, x0, tol, Nmax)
    xant = x0
    fant = f(xant)
    E = 1000
    cont = 0

    Mientras E > tol y cont < Nmax hacer
        xact = xant - (fant * df(xant)) / ((df(xant)) ** 2 - fant * d2f(xant))
        fact = f(xact)
        E = abs(xact - xant)
        cont = cont + 1
        xant = xact
        fant = fact
    Fin Mientras

    x = xact
    iteraciones = cont
    err = E

    Devolver {"x": x, "Iter": iteraciones, "Error": err}
Fin Función
```

```
def RaicesMultiples(f, df, d2f, x0, tol, Nmax):
    # Inicialización
    xant = x0
    fant = f(xant)
    E = 1000
    cont = 0

    # Ciclo
    while E > tol and cont < Nmax:
        xact = xant - (fant * df(xant)) / ((df(xant)) ** 2 - fant * d2f(xant))
        fact = f(xact)
        E = abs(xact - xant)
        cont = cont + 1
        xant = xact
        fant = fact

    # Entrega de resultados
    x = xact
    iteraciones = cont
    err = E

    return ({"x":x,"Iter":iteraciones, "Error":err})
```

False Rule

```
Función regla_falsa(f, a, b, tol, max_iter)
    i = 0
    Intentar
        fa = eval(f.replace('x', 'a'))
        fb = eval(f.replace('x', 'b'))
        Mientras abs(b - a) > tol y i < max_iter hacer
            c = (a * fb - b * fa) / (fb - fa)
            fc = eval(f.replace('x', 'c'))
            Si fa * fc < 0 entonces
                b = c
                fb = fc
            Sino
                a = c
                fa = fc
            Fin Si
            i += 1
        Fin Mientras
        Si abs(b - a) > tol entonces
            Imprimir "El método no converge después de %d iteraciones." % max_iter
            Devolver Nulo
        Devolver "La raiz de la funcion es: " + str(c)
    Excepto ZeroDivisionError
        Imprimir "Error: División por cero."
        Devolver Nulo
Fin Función

# intervalo y tolerancia
f = "x**2 - 2"
a = 1
b = 2
tol = 1e-6
max_iter = 100

raiz = regla_falsa(f, a, b, tol, max_iter)
Imprimir raiz
```

```
def regla_falsa(f, a, b, tol, max_iter):
    i = 0
    try:
        fa = eval(f.replace('x', 'a'))
        fb = eval(f.replace('x', 'b'))
        while abs(b - a) > tol and i < max_iter:
            c = (a * fb - b * fa) / (fb - fa)
            fc = eval(f.replace('x', 'c'))
            if fa * fc < 0:
                b = c
                fb = fc
            else:
                a = c
                fa = fc
            i += 1
        if abs(b - a) > tol:
            print("The method does not converge after %d iterations." % max_iter)
            return None
        return "The root of the function is:" + str(c)
    except ZeroDivisionError:
        print("Error: Division by zero.")
        return None



# # intervalo y toleracia
# f = "x**2 -2"
# a = 1
# b = 2
# tol = 1e-6
# max_iter = 100

# raiz = regla_falsa(f, a, b, tol, max_iter)
# print(raiz)
```

Secant

```
Función secante(f, x0, x1, tol, max_iter)
    i = 0
    Mientras i < max_iter hacer
        Intentar
            fx0 = eval(f.replace('x', str(x0)))
            fx1 = eval(f.replace('x', str(x1)))
            dx = (x1 - x0) / (fx1 - fx0)
            x0 = x1
            fx0 = fx1
            x1 = x1 - fx1 * dx
            fx1 = eval(f.replace('x', str(x1)))
            Si abs(fx1) <= tol entonces
                Devolver "La raíz de la función es: " + str(x1)
            Fin Si
            i += 1
        Excepto ZeroDivisionError
            Devolver "Error: División por cero."
    Devolver "El método no converge después de %d iteraciones." % max_iter
Fin Función

# intervalo y tolerancia
f = "x**2 - 2"
x0 = 1
x1 = 2
tol = 1e-6
max_iter = 5

raiz = secante(f, x0, x1, tol, max_iter)
Imprimir raiz
```

```
def secante(f, x0, x1, tol, max_iter):
    i = 0
    while i < max_iter:
        try:
            fx0 = eval(f.replace('x', str(x0)))
            fx1 = eval(f.replace('x', str(x1)))
            dx = (x1 - x0) / (fx1 - fx0)
            x0 = x1
            fx0 = fx1
            x1 = x1 - fx1 * dx
            fx1 = eval(f.replace('x', str(x1)))
            if abs(fx1) <= tol:
                return("The root of the function is: "+ str(x1))
            i += 1
        except ZeroDivisionError:
            return("Error: Division by zero.")
    return("The method does not converge after %d iterations." % max_iter)


# intervalo y toleracia
# f = "x**2 - 2"
# x0 = 1
# x1 = 2
# tol = 1e-6
# max_iter = 5

# raiz = secante(f, x0, x1, tol, max_iter)
# print(raiz)
```

Splines

```
Función spline(x, y)
    n = longitud de x
    poli = lista vacía
    internos = lista vacía
    outer = lista vacía
    crit3 = lista vacía
    coef_matrix = matriz de ceros de tamaño 3*(n-1) x 3*(n-1)

    Si longitud de y es diferente a n
        Lanzar ValueError con el mensaje "Las entradas deben tener la misma longitud"

    # Calcular polinomios internos
    Para i en rango(2, n)
        Agregar [x[i-1]^2, x[i-1], y[i]] a la lista internos
        Agregar [x[i-1]^2, x[i-1], y[i]] a la lista internos

    # Calcular polinomios externos
    Agregar [x[0]^2, x[0], y[0]] a la lista outer
    Agregar [x[n-1]^2, x[n-1], y[n-1]] a la lista outer

    # Cálculos adicionales
    Para i en rango(2, n)
        Agregar [x[i-1]^2, x[i-1], y[i]] a la lista crit3
        Agregar [x[i-1]^2, x[i-1], y[i]] a la lista internos
        # Realizar cálculos adicionales específicos aquí

    # Devolver resultados calculados
    Devolver los resultados calculados (poli, internos, outer, crit3, coef_matrix)
Fin de la función
```

```
import numpy as np


def spline(x,y):
    n = len(x)
    poli=[]
    internos=[]
    outer=[]
    crit3 = []
    coef_matrix = np.zeros(3*(n-1),3*(n-1))
    if len(y) != n:
        raise ValueError("Las entradas deben tener la misma longitud")

    #Calcular polinomios internos

    for i in range(2,n):
        internos.append([x[i-1]**2,x[i-1],y[i]])
        internos.append([x[i-1]**2,x[i-1],y[i]])

    #Calcular 2 polinomios externos
    outer.append([x[0]**2,x[0],y[0]],[x[n-1]**2,x[n-1],y[n-1]])

    for i in range(2,n):
        crit3.append([x[i-1]**2,x[i-1],y[i]])
        internos.append([x[i-1]**2,x[i-1],y[i]])




# Example usage:
x = [3, 4.5, 7, 9]
y = [2.5, 1, 2.5, 0.5]
# query_points = [1.5, 2.5, 4.5]

interpolated_values = spline(x, y)
print(interpolated_values)
```

Vandermonde

```
entradas: x, y, grado

verificar que x y y sean de la misma longitud

vander_matrix se inicializa como una lista vacia

for i= 0 hasta n:
    row se inicializa como una lista vacia
    for j=0 hasta degree+1:
        agregar a row x[i]**j
    agregar row a vander_matrix

#tenemos ahora la matriz de vandermonde, a resolver el sistema para los coeficientes con algun metodo

coefficients = resolver_sistema_lineal(vander_matrix, y)

retorna coefficients
```

```python
import numpy as np
from .Crout import crout

def vandermonde(x, y, degree):
    n = len(x)
    if len(y) != n:
        raise ValueError("Las entradas deben tener la misma longitud"

    # Crear la matriz de vandermonde
    vander_matrix = []

    for i in range(n):
        row = []
        for j in range(degree + 1):
            row.append(x[i] ** j)
        vander_matrix.append(row)

    # Resolver el sistema lineal
    coefficients = crout(vander_matrix, y)

    return coefficients

# Como usar:
x = [-2, -1, 2, 3]
y = [12.13533528, 6.367879441, -4.610943901, 2.085536923]
degree = 3

coefficients = vandermonde(x, y, degree)
print(coefficients)
```