INSTITUTO TECNOLÓGICO DE OAXACA

Alumno: García López Irvin.

Grupo: 4SC.

Horario: 3:00Pm – 4:00Pm.

Carrera: Ing. Sistemas Computacionales.

MÉTODOS NUMÉRICOS EN PYTHON

Materia: "Métodos Numéricos".

Profesor: Pedro Antonio Peralta Regalado

Fecha 29 de junio del 2022.

INTRODUCCIÓN

Al momento de intentar resolver diversos problemas matemáticos nos vemos envuelto en diversas situaciones que llegan a afectar al resultado, esto no solo en el ámbito matemático si no en otras ramas como la física, química, finanzas, todos ellos tienen la similitud de que se llegan a presentar errores tales como aproximación, truncamiento o redondeo.

Por ello recurrimos a los métodos numéricos los cuales se pueden definir como una sucesión de operaciones matemáticas utilizadas para encontrar una solución numérica aproximada a un problema determinado. Es decir, se trata de una serie de cálculos para acercarnos lo más posible a una solución numérica con una precisión razonablemente buena.

Algo importante de mencionar sobre los métodos numéricos es que son implementados en ordenadores, dispositivos electrónicos o software especializados en ingeniería, los cuales, ya tienen incluidos los métodos numéricos en sus algoritmos de resolución, siendo vitales en el área de simulación de procesos y para dar respuestas rápidas donde una solución analítica se vuelve compleja.

En este presente trabajo se muestran algunos algoritmos de los métodos numéricos desarrollados en el lenguaje de programación PYTHON, esto con el fin de conocer como es su realización en un distinto lenguaje al que normalmente se acostumbra implementar (MATLAB U OCTAVE).

ÍNDICE.

LISTA DE CÓDIGOS.

Se entregarán los siguientes códigos programados en Python (en formato función salvo en el último tema), además deberá documentar como utilizar dichas funciones mostrando al menos un ejemplo de aplicación, considere tajantemente mostrar ejemplos relativos a los circuitos eléctricos.

1. Sistemas de ecuaciones lineales.
   1.1 Método de Gauss.

   La eliminación de Gauss es el método más familiar para resolver ecuaciones simultáneas. Consta de dos partes: la fase de eliminación y la fase de sustitución hacia atrás.

```
import numpy as np
# INGRESO
A = np.array([[4,2,5],
              [2,5,8],
              [5,4,3]])
B = np.array([[60.70],
              [92.90],
              [56.30]])
# PROCEDIMIENTO
casicero = 1e-15 # Considerar como 0
# Evitar truncamiento en operaciones
A = np.array(A,dtype=float)
# Matriz aumentada
AB  = np.concatenate((A,B),axis=1)
AB0 = np.copy(AB)
# Pivoteo parcial por filas
tamano = np.shape(AB)
n = tamano[0]
m = tamano[1]
# Para cada fila en AB
```

```
for i in range(0,n-1,1):
    # columna desde diagonal i en adelante
    columna  = abs(AB[i:,i])
    dondemax = np.argmax(columna)
    # dondemax no está en diagonal
    if (dondemax !=0):
        # intercambia filas
        temporal = np.copy(AB[i,:])
        AB[i,:] = AB[dondemax+i,:]
        AB[dondemax+i,:] = temporal
AB1 = np.copy(AB)
# eliminación hacia adelante
for i in range(0,n-1,1):
    pivote   = AB[i,i]
    adelante = i + 1
    for k in range(adelante,n,1):
        factor  = AB[k,i]/pivote
        AB[k,:] = AB[k,:] - AB[i,:]*factor

# sustitución hacia atrás
ultfila = n-1
ultcolumna = m-1
X = np.zeros(n,dtype=float)
for i in range(ultfila,0-1,-1):
    suma = 0
    for j in range(i+1,ultcolumna,1):
        suma = suma + AB[i,j]*X[j]
    b = AB[i,ultcolumna]
    X[i] = (b-suma)/AB[i,i]
X = np.transpose([X])
# SALIDA
```

```
print('Matriz aumentada:')
print(AB0)
print('Pivoteo parcial por filas')
print(AB1)
print('eliminación hacia adelante')
print(AB)
print('solución: ')
print(X)
```

```
Matriz aumentada:
[[ 4.   2.   5.  60.7]
 [ 2.   5.   8.  92.9]
 [ 5.   4.   3.  56.3]]
Pivoteo parcial por filas
[[ 5.   4.   3.  56.3]
 [ 2.   5.   8.  92.9]
 [ 4.   2.   5.  60.7]]
eliminación hacia adelante
[[ 5.    4.    3.   56.3 ]
 [ 0.    3.4   6.8  70.38]
 [ 0.    0.    5.   40.5 ]]
solución:
[[2.8]
 [4.5]
 [8.1]]
```

1.2 Factorización LU y PLU.

El proceso de calcular L y U para un A dado se conoce como descomposición LU o factorización LU. La descomposición LU no es única (las combinaciones de L y Upara un A prescrito son interminables), a menos que se impongan ciertas restricciones a L o U. Las restricciones distinguen un tipo de descomposición de otro.

```
import numpy as np
print ("Descoposicion LU, matrices cuadradas")
m=int (input("Introduce el numero de renglones"))
matriz= np.zeros([m,m])
u=np.zeros([m,m])
l=np.zeros([m,m])
print("Introduce los elementos de la matriz")
for r in range (0,m):
```

```
  for c in range (0,m):
      matriz [r,c]=(input("Elemento
a["+str(r+1)+","+str(c+1)+"]"))
      matriz [r,c]=float(matriz[r,c])
      u[r,c]=matriz[r,c]
  #Operaciones para hacer cers debajo
  for k in range (0,m):
    for r in range (0,m):
       if (k==r):
         l[k,r]=1
       if (k<r):
         factor=(matriz[r,k]/matriz  [k,k])
         l[r,k]=factor
         for c in range (0,m):
            matriz [r,c]=matriz[r,c]-(factor*matriz[k,c])
            u[r,c]=matriz[r,c]
print("Impresion de resultados")
print ("Matriz U")
print (u)
print ("matriz L")
print(l)
```

```
Descoposicion LU, matrices cuadradas
Introduce el numero de renglones3
Introduce los elementos de la matriz
Elemento a[1,1]4
Elemento a[1,2]-2
Elemento a[1,3]1
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:20: RuntimeWarning: invalid value encountered in double_scalars
Elemento a[2,1]20
Elemento a[2,2]-7
Elemento a[2,3]12
Elemento a[3,1]-8
Elemento a[3,2]13
Elemento a[3,3]17
Impresion de resultados
Matriz U
[[ 4. -2.  1.]
 [ 0.  3.  7.]
 [ 0.  0. -2.]]
matriz L
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [-2.  3.  1.]]
```

1.3 Inversa de una matriz.

Una matriz es inversa de otra cuando al multiplicar ambas (en cualquier orden) se obtiene la matriz identidad.

```
import numpy as np
arr = np.array([[1, 3], [5, 7]])
arr_inv = np.linalg.inv(arr)
print(arr_inv)
```

```
import numpy as np

arr = np.array([[1, 3], [5, 7]])

arr_inv = np.linalg.inv(arr)

print(arr_inv)
```

```
[[-0.875  0.375]
 [ 0.625 -0.125]]
```

1.4 Determinantes.

El determinante de una matriz siempre es igual al de su matriz traspuesta.

El determinante de una matriz será siempre cero (nulo) si la matriz contiene dos filas o columnas iguales, si los elementos de una fila o columna son todo ceros o si los elementos de una fila o columna son una combinación lineal de las demás.

```
import numpy as np
n_array = np.array([[50, 29], [30, 44]])
print("Numpy Matrix is:")
print(n_array)
det = np.linalg.det(n_array)
print("\nDeterminant of given 2X2 matrix:")
print(int(det))
```

```
import numpy as np

n_array = np.array([[55, 25, 15],
                    [30, 44, 2],
                    [11, 45, 77]])

print("Numpy Matrix is:")
print(n_array)

det = np.linalg.det(n_array)

print("\nDeterminant of given 3X3 square matrix:")
print(int(det))
```

```
Numpy Matrix is:
[[55 25 15]
 [30 44  2]
 [11 45 77]]

Determinant of given 3X3 square matrix:
137180
```

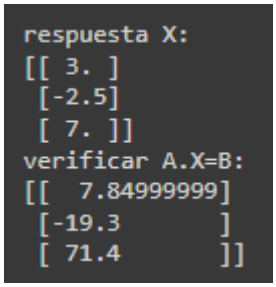
1.5 Gauss Seidel.

Los métodos iterativos de Jacobi y Gauss-Seidel son los procesos de aproximaciones sucesivas para resolver sistemas de ecuaciones lineales compatibles determinados. Ambos requieren de la verificación de un criterio de convergencia comúnmente conocido como diagonal pesada.

```
import numpy as np
# INGRESO
A = np.array([[3. , -0.1, -0.2],
              [0.1,  7  , -0.3],
              [0.3, -0.2, 10  ]])
B = np.array([7.85,-19.3,71.4])
X0  = np.array([0.,0.,0.])
tolera = 0.00001
iteramax = 100
# PROCEDIMIENTO
# Gauss-Seidel
tamano = np.shape(A)
n = tamano[0]
```

```
m = tamano[1]
#  valores iniciales
X = np.copy(X0)
diferencia = np.ones(n, dtype=float)
errado = 2*tolera
itera = 0
while not(errado<=tolera or itera>iteramax):
    # por fila
    for i in range(0,n,1):
        # por columna
        suma = 0
        for j in range(0,m,1):
            # excepto diagonal de A
            if (i!=j):
                suma = suma-A[i,j]*X[j]
        nuevo = (B[i]+suma)/A[i,i]
        diferencia[i] = np.abs(nuevo-X[i])
        X[i] = nuevo
    errado = np.max(diferencia)
    itera = itera + 1
# Respuesta X en columna
X = np.transpose([X])
# revisa si NO converge
if (itera>iteramax):
    X=0
# revisa respuesta
verifica = np.dot(A,X)
# SALIDA
print('respuesta X: ')
print(X)
print('verificar A.X=B: ')
```

print(verifica)



1.6 Método de las potencias directa/inversa.

El método de las potencias es un método iterativo que calcula sucesivas aproximaciones a los autovectores y autovalores de una matriz. El método se usa principalmente para calcular el autovector de mayor autovalor en matrices grandes.

```
#Implementacion del método de las potencias en OCTAVE
function [lambda,y] = potencias(A,x,tol,maxiter)
lambdaviejo = rand();
k = 0;
error = 1000;
while k<=maxiter && error>=tol
y = x/norm(x);
x = A*y;
lambda = y'*x;
error = abs((lambda-lambdaviejo)./lambda);
lambdaviejo = lambda;
k = k+1;
endwhile
if k>maxiter
warning('No converge tras %i iteraciones \n',maxiter);
else
fprintf('El método converge en %i iteraciones \n',k);
endif
endfunction
```

2. Ecuaciones no lineales.

2.1 Método de bisección.

El método de bisección se aplica a funciones algebraicas o trascendentes y proporciona únicamente raíces reales. Tiene su origen en un popular algoritmo de búsqueda de datos en arreglos vectoriales denominado búsqueda binaria. Es un método cerrado, es decir, requiere de un intervalo en el cual esté atrapada una raíz.

```
1;
function y=fun(x)
  y=x+exp(2*x)
endfunction
a=-1;
b=0;
tol=10^(-2);
maxiter= ceil((log(b-a)-log(tol))/log(2))-1;
n=0;
c=(a+b)/2;
yc=fun(c);
fprintf ('n     intervalo          c_n       f(c_n) \n')
fprintf ('%i      [%f,%f])        %f    %f    f%\n',n, a, b, c, yc)
ya=fun(a);
yb=fun(b);
for n=1:maxiter
  if yc==0
    a=c;
    b=c;
    fprintf ('%i      [%f,%f]      %f      %f \n', n, a, b, c, yc)
    break
   elseif yb*yc>0
    b=c;
    yb=yc;
```

```
else
  a=c;
  ya=yc;
endif
 c=(a+b)/2;
 yc=fun(c);
 fprintf ('%i     [%f,%f]     %f     %f \n', n, a, b, c, yc)
endfor
```

2.2 Método de falsa posición.

El método de la falsa posición pretende conjugar la seguridad del método de la bisección con la rapidez del método de la secante. Este método, como en el método de la bisección, parte de dos puntos que rodean a la raíz f(x) = 0, es decir, dos puntos x0 y x1tales que f(x0)f(x1) < 0.

```
# Algoritmo Posicion Falsa para raices
# busca en intervalo [a,b]
import numpy as np
# INGRESO
fx = lambda x: x**3 + 4*x**2 - 10
a = 1
b = 2
tolera = 0.001
# PROCEDIMIENTO
tramo = abs(b-a)
while not(tramo<=tolera):
      fa = fx(a)
    fb = fx(b)
    c = b - fb*(a-b)/(fa-fb)
    fc = fx(c)
    cambia = np.sign(fa)*np.sign(fc)
    if (cambia > 0):
     tramo = abs(c-a)
```

```
            a = c
           else:
        tramo = abs(b-c)
         b = c
            raiz = c
        # SALIDA
    print(raiz)
```

2.3 Método de Newton/Raphson

En análisis numérico, el método de Newton (conocido también como el método de Newton-Raphson o el método de Newton-Fourier) es un algoritmo eficiente para encontrar aproximaciones de los ceros o raíces de una función real.

```
        # Método de Newton-Raphson
# Ejemplo 1 (Burden ejemplo 1 p.51/pdf.61)
import numpy as np
# INGRESO
fx  = lambda x: x**3 + 4*(x**2) - 10
dfx = lambda x: 3*(x**2) + 8*x
x0 = 2
tolera = 0.001
# PROCEDIMIENTO
tabla = []
tramo = abs(2*tolera)
xi = x0
while (tramo>=tolera):
    xnuevo = xi - fx(xi)/dfx(xi)
    tramo  = abs(xnuevo-xi)
    tabla.append([xi,xnuevo,tramo])
    xi = xnuevo
# convierte la lista a un arreglo.
tabla = np.array(tabla)
```

```
n = len(tabla)
# SALIDA
print(['xi', 'xnuevo', 'tramo'])
np.set_printoptions(precision = 4)
print(tabla)
print('raiz en: ', xi)
print('con error de: ',tramo)
```

```
['xi', 'xnuevo', 'tramo']
[[2.0000e+00 1.5000e+00 5.0000e-01]
 [1.5000e+00 1.3733e+00 1.2667e-01]
 [1.3733e+00 1.3653e+00 8.0713e-03]
 [1.3653e+00 1.3652e+00 3.2001e-05]]
raiz en:  1.3652300139161466
con error de:  3.200095847999407e-05
```

3.Interpolación.

3.1 Método de Lagrange.

El polinomio de interpolación de Lagrange es una reformulación del polinomio de interpolación de Newton, el método evita el cálculo de las diferencias divididas. El método tolera las diferencias entre las distancias x entre puntos.

```
# Interpolacion de Lagrange
# divisoresL solo para mostrar valores
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
# INGRESO , Datos de prueba
xi = np.array([0, 0.2, 0.3, 0.4])
fi = np.array([1, 1.6, 1.7, 2.0])
# PROCEDIMIENTO
# Polinomio de Lagrange
n = len(xi)
```

```
x = sym.Symbol('x')
polinomio = 0
divisorL = np.zeros(n, dtype = float)
for i in range(0,n,1):
    # Termino de Lagrange
    numerador = 1
    denominador = 1
    for j  in range(0,n,1):
        if (j!=i):
            numerador = numerador*(x-xi[j])
            denominador = denominador*(xi[i]-xi[j])
    terminoLi = numerador/denominador
    polinomio = polinomio + terminoLi*fi[i]
    divisorL[i] = denominador
# simplifica el polinomio
polisimple = polinomio.expand()
# para evaluación numérica
px = sym.lambdify(x,polisimple)
# Puntos para la gráfica
muestras = 101
a = np.min(xi)
b = np.max(xi)
pxi = np.linspace(a,b,muestras)
pfi = px(pxi)
# SALIDA
print('    valores de fi: ',fi)
print('divisores en L(i): ',divisorL)
print()
print('Polinomio de Lagrange, expresiones')
print(polinomio)
print()
```

```
print('Polinomio de Lagrange: ')
print(polisimple)
# Gráfica
plt.plot(xi,fi,'o', label = 'Puntos')
plt.plot(pxi,pfi, label = 'Polinomio')
plt.legend()
plt.xlabel('xi')
plt.ylabel('fi')
plt.title('Interpolación Lagrange')    plt.show()
```

```
    valores de fi:  [1.  1.6 1.7 2. ]
divisores en L(i):  [-0.024  0.004 -0.003  0.008]

Polinomio de Lagrange, expresiones
400.0*x*(x - 0.4)*(x - 0.3) - 566.666666666667*x*(x - 0.4)*(x - 0.2) + 250.0*x*(x - 0.3)*(x - 0.2) - 41.6666666666667*(x - 0.4)*(x - 0.3)*(x - 0.2)

Polinomio de Lagrange:
41.6666666666667*x**3 - 27.5*x**2 + 6.83333333333334*x + 1.0
```

3.2 Método de Newton.

Este método es útil para situaciones que requieran un número bajo de puntos para interpolar, ya que a medida que crece el número de puntos, también lo hace el grado del polinomio. Existen ciertas ventajas en el uso de este polinomio respecto al polinomio interpolador de Lagrange.

```
print ("------- Interpolacion Polinomica (Newton) -------")
n = int(input("Ingrese el grado del polinomio a evaluar: "))+1
matriz = [0.0] * n
for i in range(n):
    matriz[i] = [0.0] * n
vector = [0.0] * n
print (matriz)
print (vector)
for i in range(n):
    x = input("Ingrese el valor de x: ")
    y = input("Ingrese el valor de f("+x+"): ")
    vector[i]=float(x)
    matriz[i][0]=float(y)
```

```
print (vector)
print (matriz)
punto_a_evaluar = float(input("Ingrese el punto a evaluar: "))
print ("------------------------------")
print ("------- Calculando ... -------")
print ("------------------------------")
for i in range(1,n):
    for j in range(i,n):
        print ("i=",i," j=",j)
        print ("(",matriz[j][i-1],"-",matriz[j-1][i-1],")/(",vector[j],"-",vector[j-i],")")
        matriz[j][i] = ( (matriz[j][i-1]-matriz[j-1][i-1]) / (vector[j]-vector[j-i]))
        print ("matriz[",j,"][",i,"] = ",(matriz[j][i-1]-matriz[j-1][i-1])/(vector[j]-vector[j-i]))
print ("------------------------------")
print ("------------------------------")
for i in range(n):
    print (matriz[i])
    print ("------------------------------")
    print ("------------------------------")
aprx = 0
mul = 1.0
for i in range(n):
    print ("matriz[",i,"][",i,"]","=",matriz[i][i])
    mul = matriz[i][i];
    print ("mul antes del ciclo j=",mul)
for j in range(1,i+1):
    mul = mul * (punto_a_evaluar - vector[j-1])
    print ("mul en el ciclo j=",mul)
    aprx = aprx + mul
```

```
print ("-----------------------------")
print ("-----------------------------")
print ("El valor aproximado de f(",punto_a_evaluar,") es: ", aprx)
```

```
------- Interpolacion Polinomica (Newton) -------
Ingrese el grado del polinomio a evaluar: 2
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
[0.0, 0.0, 0.0]
Ingrese el valor de x: 2
Ingrese el valor de f(2): 12
Ingrese el valor de x: 3
Ingrese el valor de f(3): 13
Ingrese el valor de x: 4
Ingrese el valor de f(4): 14
[2.0, 3.0, 4.0]
[[12.0, 0.0, 0.0], [13.0, 0.0, 0.0], [14.0, 0.0, 0.0]]
Ingrese el punto a evaluar:
```

3.3 Ajuste de un polinomio por mínimos cuadrados.

El ajuste de mínimos cuadrados utiliza el análisis estadístico para estimar las coordenadas más probables de los puntos conectados de una medición en una red. Las coordenadas de un nuevo punto se pueden calcular de forma única mediante un rumbo y una distancia desde un punto existente.

```
clear all
format long
printf('PORGRAMA MINIMOS CUADRADO GRADO 1 \n')
#Cambio de los valores de la matriz
x=[0 2 3 6 7];
y=[0.120 0.153 0.171 0.225 0.260];
a=polyfit(x,y,1)
fprintf ('a0=%f a1=%f\n', a(2), a(1))
Xint =4;
fprintf ('el valor de para x=%f es:\n', Xint)
Yint=a(2)+a(1)*Xint;
fprintf ('Y(%f)=%f\n',Xint,Yint)
```

```
Ventana de comandos
>> minimosCuadrados

PROGRAMA MINIMOS CUADRADO GRADO 1
a =

   1.941566265060242e-02   1.159036144578313e-01

a0=0.115904 a1=0.019416
el valor de para x=4.000000 es:
Y(4.000000)=0.193566
>>
```

3.4 Interpoladores cúbicos.

Los trazadores se utilizan cuando por ejemplo tenemos un conjunto de puntos y queremos encontrar la función que pasa por todos esos puntos y que mejor se aproxima, dicha función a calcular es lo que llamamos trazador o función spline. Esta técnica de los trazadores es muy útil en informática para dibujar figuras no regulares. Por ejemplo, se puede utilizar para dibujar un perro o un edificio con tan sólo dar unos cuantos puntos representativos de dichas figuras y buscando su trazador.

```
from Numeric import *
from math import *
from numpy.linalg import *
from pylab import *
vectorx = array((0.,1.,3.), Float)
vectory = array((1.,2.,0.), Float)
h = array((0.,0.), Float)
lamda = array((0.0, 0.0),Float)
mu = array((0.0, 0.0),Float)
d = array((0.0, 0.0,0.0),Float)
A = array(([0.0, 0.0, 0.0],[0.0, 0.0, 0.0],[0.0, 0.0, 0.0]), Float)
i = 0
n = 0
while i&lt;len(h) :
```

```
    h[i] = vectorx[i+1]-vectorx[i]
    i=i+1
i=1
while i&lt;len(h):
    lamda[i]=h[i]/(h[i-1]+h[i])
    i=i+1
i = 1
while i&lt;len(h):
    mu[i-1]=1.-lamda[i]
    i=i+1
i=1
while i&lt;len(h):
    d[i] = (6./(h[i-1]+h[i]))*((vectory[i+1]-vectory[i])/h[i]-(vectory[i]-
vectory[i-1])/h[i-1])
    i=i+1
i=0
n=1
while i&lt;len(mu):
    A[i][n]=lamda[n-1]
    i=i+1
    n=n+1
i=1
n=0
while n&lt;len(mu):
    A[i][n]=mu[i-1]
    i=i+1
    n=n+1
i=0
while i&lt;=len(mu):
    A[i][i]=2.
    i=i+1
```

```
M = solve(A,d)
def funcionS1() :
    x = arange(0,3,0.1)
    m = len(x)
    s1 = array(([0.,0.]),Float)
    s2 = array(([0.0,0.]),Float)
    S = zeros(m, Float64)
    i = 0
    l = 0
    while i&lt;=1 :
        while l&lt;m :
            s1[i]=(vectory[i+1]-vectory[i])/(h[i+1])-
(1./6.)*(2*M[i]+M[i+1])*h[i+1]
            s2[i]=(1./6.)*((M[i+1]-M[i])/h[i+1])
            S[l]  =  vectory[i]+s1[i]*(x[l]-vectorx[i])+0.5*M[i]*pow(x[l]-
vectorx[i],2.)+s2[i]*pow(x[l]-vectorx[i],3.)
            l = l+1
        i = i+1
        plot(x, S, linewidth=1.0)
        xlabel('Abcisa')
        ylabel('Ordenada')
        title('Metodo I De Los Splines ')
        grid(True)
        axhline(linewidth=1, color='r')
        axvline(linewidth=1, color='r')
        show()
funcionS1()
```

Implementación de la interpolación cubica en Octave.

```
function p=difdiv(x,y)
  n=length(y)-1;
  d=y;
  p=d(1);
  for k=1:n
    d=(d(2:end)-d(1:end-1))./(x(k+1:end)-x(1:end-k));
    p=[0,p]+d(1)*poly(x(1:k));
  endfor
  endfunction
```

Ventana de comandos

```
>> p = difdiv(x,y);
>> polyout(p,'x')
0.00013333*x^4 - 0.00155*x^3 + 0.0057167*x^2 + 0.0102*x^1 + 0.12
>>
```

4. Cálculo numérico.

a. Derivación e integración de los datos tabulados.

```
% Fichero Derivadas_ex_1.m para simular derivación de e^x
h = 1; e=exp(1);
f=fopen('Salida_derivadas_1','w');
for i=1:14
h=h/10;
der1 = (exp(1+h)-e)/h; err1=abs(der1-e); der2=(e-exp(1-h))/h;
der3 = (exp(1+h)-exp(1-h))/2/h; err2=abs(der3-e);
```

```
der4 = (exp(1+h)-2*e+exp(1-h))/(h*h);

fprintf(f,'%6.0e %18.15f %18.15f %18.15f %18.15f %18.15f
%18.15f %18.15f\n',...

h,e,der1,err1,der2,der3,err2,der4);

end

fclose('all');

h=0.5; R=zeros(32,3);

for i=1:32

fd=(exp(1+h)-e)/h;

cd=(exp(1+h)-exp(1-h))/(2*h);

R(i,:)=[h abs(fd-e) abs(cd-e)];

h=h/2;

end

loglog(R(:,1),R(:,2),'+',R(:,1),R(:,3),'or')

legend('Derivadas avanzadas','Derivadas
centradas','Location','NorthWest');

xlabel('h');

ylabel('error');
```

b. Derivación e integración de funciones.

```
function [x,df] = derinum(f,a,b,N)

h = (ba)/(N1);

x = linspace(a,b,N);

df = (feval(f,a+h*(1:N))feval(f,a+h*(1:(N2))))/(2*h);
```

```
endfunction
```

c. Integrador en cuadradas Gaussianas.

```
function int = rectang(f,a,b,N = 150)
% Función que aproxima la integral de f en el
% intervalo [a,b] mediante la fórmula compuesta de
% rectángulos (punto medio) con N subintervalos
h = (ba)/N;
int = h*sum(feval(f,a+h*(1:N)h/2));
endfunction
```

CONCLUSIÓN.

Para concluir, la implementación de los métodos numéricos en otro lenguaje nos ayuda a comprender mejor su funcionamiento por ello es una gran practica el desarrollar este tipo de proyectos, aunque estos suelen complicarse ya que, aunque el lenguaje sea sencillo de trabajar el implementar llega a ser tedioso por la sintaxis, en este caso Python es un lenguaje sencillo de aprender. Otro punto importante a mencionar es que como los métodos numéricos normalmente se implementan en MATHLAB u OCTAVE lo cual son softwares que están enfocados principalmente al desarrollo de estos programas matemáticos.

Como observación sobre el trabajo en algunos casos solo se presenta solamente el código y no se muestra un ejemplo de su ejecución, debido a problemas que llegan a presentarse por el desarrollo de funciones o el propio IDE por ello se muestra cuál es su implementación en el lenguaje Octave y otros que si se lograron implementar en Python solo muestra el código como función, pero falta un ejemplo práctico.