

## **Opción 3**

**Instituto tecnológico de Oaxaca  
Ingeniería en Sistemas Computacionales**

### **Métodos Numéricos**

#### **Integrantes:**

*Alberto Ramírez Bautista  
Dafne Santiago Solano  
Jesús Edgardo López Ramírez  
Carlos Eduardo Rendón Torres  
Ángel Jesús Zorrilla Cuevas  
Eric Moisés León Cruz*

**Grupo: 4SC**

**Horario: 15:00 - 16:00**

**Fecha de entrega: 24 de junio del 2022**

## ***Índice.***

<b>Índice.</b>	<b>2</b>
<b>Introducción.</b>	<b>4</b>
<b>Desarrollo.</b>	<b>5</b>
Sistema de Ecuaciones Lineales.	5
Método de Gauss.	5
SALIDA	6
Factorización LU y PLU.	7
Ejemplo	7
Inversa de una matriz.	8
SALIDA	8
Determinantes.	8
SALIDA	9
Gauss Seidel	9
EJEMPLO	11
Método de las potencias directa/inversa.	11
SALIDA.	13
Ecuaciones no lineales.	14
Método de bisección.	14
SALIDA	15
Método de falsa posición.	16
SALIDA	17
Método de Newton/Raphson.	19
SALIDA	20
Método de Newton/Raphson Varias Variables.	20
III. Interpolación.	21
Método de lagrange.	21
SALIDA.	22
Método de Newton.	22
SALIDA.	24
Ajuste de polinomio por mínimos cuadrados.	24
Salida	25
Interpoladores cúbicos.	25
EJEMPLO	26
SALIDA	28
IV. Cálculo numérico.	28
Derivación e integración de datos tabulados.	28
Derivación e integración de funciones.	29
SALIDA	30
SALIDA	30
Integrador en cuadraturas Gaussianas.	30

Salida	31
Ecuaciones diferenciales.	31
Métodos para resolver una ecuación diferencial, problema de condiciones iniciales.	
31	
Euler izquierdo.	31
SALIDA	32
Euler centrado.	32
SALIDA	33
Euler derecho.	33
SALIDA	34
Métodos de Runge/Kutta 3er orden.	35
Salida	36
Métodos de Runge/Kutta 4to orden.	36
SALIDA	36
Métodos para resolver un sistema de ecuaciones, problema de condiciones iniciales.	
36	
Euler izquierdo.	36
Euler centrado.	36
SALIDA	37
Euler derecho.	38
Métodos de Runge/Kutta 3er orden.	38
Métodos de Runge/Kutta 4to orden.	38
SALIDA	39
Aplicaciones al problema de condiciones en la frontera.	40
Ejemplos.	40
<b>Conclusiones.</b>	<b>40</b>

## ***Introducción.***

Como se vio en clase, la necesidad de plantear métodos numéricos surge por dos principales razones, la primera es por problemas que no se pueden resolver a mano, debido a lo irrealizable e imposible que es obtener una expresión analítica. La segunda deriva de la naturaleza humana, es inviable hacer una multiplicación de matrices de gran tamaño. Es por eso que la necesidad de utilizar herramientas computacionales se ha vuelto fundamental. Fueron los métodos numéricos los que establecieron las matemáticas formales, y las matemáticas formales establecen la teoría de computación.

Los métodos numéricos son prácticamente procedimientos mediante los cuales se obtiene, casi siempre de manera aproximada, la solución de ciertos problemas realizando cálculos puramente aritméticos y lógicos como operaciones aritméticas elementales, cálculo de funciones, consulta de una tabla de valores, cálculo preposicional, etc.

Para este trabajo se decidió trabajar con el lenguaje de programación llamado **python**,

Los procedimientos que se presentan a continuación, codificados en el lenguaje **python** consisten en una lista finita de instrucciones precisas que especifican una secuencia de operaciones algebraicas y lógicas, conocida como *algoritmo*, que producen o bien una aproximación de la solución del problema o bien una gráfica.

## Desarrollo.

### I. Sistema de Ecuaciones Lineales.

#### A. Método de Gauss.

```
A = np.array([[2, -1, 1],
              [3, 1, -2],
              [-1, 2, 5]])
B = np.array([[2], [9], [-5]])

# 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)

# eliminacion 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
AB2 = np.copy(AB)

# elimina hacia atras
```

```

ultfila = n - 1
ultcolumna = m - 1
for i in range(ultfila, 0 - 1, -1):
    pivote = AB[i, i]
    atras = i - 1
    for k in range(atras, 0 - 1, -1):
        factor = AB[k, i] / pivote
        AB[k, :] = AB[k, :] - AB[i, :] * factor
    # diagonal a unos
    AB[i, :] = AB[i, :] / AB[i, i]
X = np.copy(AB[:, ultcolumna])
X = np.transpose([X])

# SALIDA
print('Matriz aumentada:')
print(AB0)
print('Pivoteo parcial por filas')
print(AB1)
print('eliminacion hacia adelante')
print(AB2)
print('eliminación hacia atrás')
print(AB)
print('solución de X: ')
print(X)

```

## SALIDA

```

Matriz aumentada:
[[ 2. -1.  1.  2.]
 [ 3.  1. -2.  9.]
 [-1.  2.  5. -5.]]
Pivoteo parcial por filas
[[ 3.  1. -2.  9.]
 [-1.  2.  5. -5.]
 [ 2. -1.  1.  2.]]
eliminación hacia adelante
[[ 3.          1.         -2.          9.          ]
 [ 0.          2.33333333  4.33333333 -2.          ]
 [ 0.          0.          5.42857143 -5.42857143]]
eliminación hacia atrás
[[ 1.  0.  0.  2.]
 [ 0.  1.  0.  1.]
 [ 0.  0.  1. -1.]]
solución de X:
[[ 2.]
 [ 1.]
 [-1.]]

```

## B. Factorización LU y PLU.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.sparse as sp
import scipy.linalg as la
import nimfa
from sklearn.decomposition import NMF, PCA
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import pandas as pd

# graficos incrustados
%matplotlib inline

# parámetros de estilo
sns.set(style='darkgrid', palette='muted')
pd.set_option('display.mpl_style', 'default')
pd.set_option('display.notebook_repr_html', True)
plt.rcParams['figure.figsize'] = 8, 6
```

### Ejemplo

```
# Ejemplo factorización LU
A = np.array([[7, 3, -1, 2],
              [3, 8, 1, -4],
              [-1, 1, 4, -1],
              [2, -4, -1, 6]])
P, L, U = la.lu(A)
# Matriz A
A
array([[ 7,  3, -1,  2],
       [ 3,  8,  1, -4],
       [-1,  1,  4, -1],
       [ 2, -4, -1,  6]])
# Matriz de permutación
P
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
# Matriz triangular inferior
L
array([[ 1.,          0.,          0.,          0.],
       [ 0.42857143,  1.,          0.,          0.],
       [-0.14285714,  0.21276596,  1.,          0.],
       [ 0.28571429, -0.72340426,  0.08982036,  1.]])
# Matriz triangular superior
U
array([[ 7.,          3.,         -1.,          2.],
       [ 0.,          6.71428571,  1.42857143, -4.85714286],
```

```

    [ 0.          ,  0.          ,  3.55319149,  0.31914894],
    [ 0.          ,  0.          ,  0.          ,  1.88622754]])
# A = LU
L @ U
array([[ 7.,  3., -1.,  2.],
       [ 3.,  8.,  1., -4.],
       [-1.,  1.,  4., -1.],
       [ 2., -4., -1.,  6.]])

```

### C. Inversa de una matriz.

```

# MATRIZ INVERSA

# Librerias
import string
import numpy as np
from scipy import linalg

# declarando matriz A (3x3)
A = np.array([[2, -1, 1],
              [3, 1, -2],
              [-1, 2, 5]])
# Calcular la inversa B = A^(-1)
B = linalg.inv(A)

print("-----Matriz original-----")
print(A)
print()
print("-----Matriz inversa-----")
print(B)

```

#### **SALIDA**

```

-----Matriz original-----
[[ 2 -1  1]
 [ 3  1 -2]
 [-1  2  5]]

-----Matriz inversa-----
[[ 0.23684211  0.18421053  0.02631579]
 [-0.34210526  0.28947368  0.18421053]
 [ 0.18421053 -0.07894737  0.13157895]]

```

### D. Determinantes.

```

# Determinante de una matriz
import numpy as np
import functools

```



```
def det(matriz):
    orden = len(matriz)
    posdet = 0
    for i in range(orden):
        posdet += functools.reduce((lambda x, y: x * y), [matriz[(i + j) %
orden][j] for j in range(orden)])
    negdet = 0
    for i in range(orden):
        negdet += functools.reduce((lambda x, y: x*y), [matriz[(orden-i-j)%
orden][j] for j in range(orden)])
    return posdet - negdet

A = np.array([[2, -1, 1],
              [3, 1, -2],
              [-1, 2, 5]])

print(det(A))
```

### SALIDA

El determinante de la matriz es: 38

### E. Gauss Seidel

```
import numpy
m=int(input('Valor de m:'))          #Renglones = m
n=int(input('Valor de n:'))          #Columnas = n
matrix = numpy.zeros((m,n))          #Matriz de coeficientes
x=numpy.zeros((m))                   #Vector Solución

vector=numpy.zeros((n))
comp=numpy.zeros((m))
error=[]

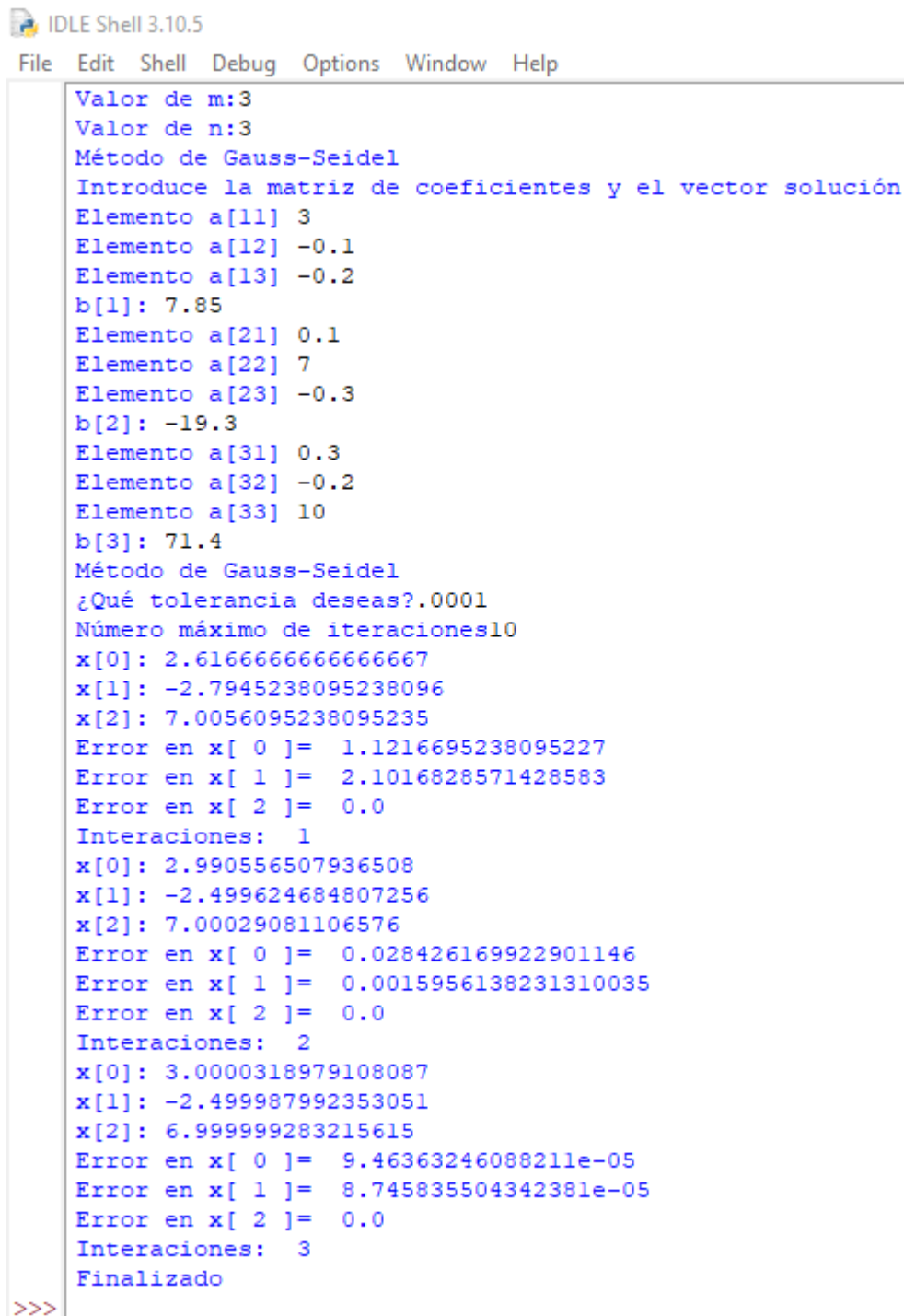
print ('Método de Gauss-Seidel')
print ('Introduce la matriz de coeficientes y el vector solución')
for r in range(0,m):
    for c in range(0,n):
        matrix[(r),(c)]=float(input("Elemento a["+str(r+1)+str(c+1)+"]
"))
    vector[(r)]=float(input('b['+str(r+1)+']: '))
print ("Método de Gauss-Seidel")
tol=float(input("¿Qué tolerancia deseas?"))
itera=int(input("Número máximo de iteraciones"))
#MÉTODO DE GAUSS SEIDEL
k=0
while k < itera:
    suma=0
    k=k+1
```

```

for r in range(0,m):
    suma=0
    for c in range(0,n):
        if (c != r):
            suma=suma+matrix[r,c]*x[c]
    x[r]=(vector[r]-suma)/matrix[r,r]
    print("x[" + str(r)+"]: "+str(x[r]))
del error[:]

#Comprobación
for r in range(0,m):
    suma=0
    for c in range(0,n):
        suma=suma+matrix[r,c]*x[c]
    comp [r]=suma
    dif=abs(comp[r]-vector[r])
    error.append(dif)
    print("Error en x[",r,"]= ",error[r])
print("Interacciones: ",k)
if all(i<=tol for i in error) == True:
    break
print("Finalizado")

```

**EJEMPLO**


```

IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
Valor de m:3
Valor de n:3
Método de Gauss-Seidel
Introduce la matriz de coeficientes y el vector solución
Elemento a[11] 3
Elemento a[12] -0.1
Elemento a[13] -0.2
b[1]: 7.85
Elemento a[21] 0.1
Elemento a[22] 7
Elemento a[23] -0.3
b[2]: -19.3
Elemento a[31] 0.3
Elemento a[32] -0.2
Elemento a[33] 10
b[3]: 71.4
Método de Gauss-Seidel
¿Qué tolerancia deseas?.0001
Número máximo de iteraciones10
x[0]: 2.6166666666666667
x[1]: -2.7945238095238096
x[2]: 7.0056095238095235
Error en x[ 0 ]= 1.1216695238095227
Error en x[ 1 ]= 2.1016828571428583
Error en x[ 2 ]= 0.0
Iteraciones: 1
x[0]: 2.990556507936508
x[1]: -2.499624684807256
x[2]: 7.00029081106576
Error en x[ 0 ]= 0.028426169922901146
Error en x[ 1 ]= 0.0015956138231310035
Error en x[ 2 ]= 0.0
Iteraciones: 2
x[0]: 3.0000318979108087
x[1]: -2.499987992353051
x[2]: 6.999999283215615
Error en x[ 0 ]= 9.46363246088211e-05
Error en x[ 1 ]= 8.745835504342381e-05
Error en x[ 2 ]= 0.0
Iteraciones: 3
Finalizado
>>>

```

**F. Método de las potencias directa/inversa.**

```
# Algoritmo Posicion Falsa
```

```
from math import *
```

```
def pol(x):
```

```
    ...
```

```
    Funcion de prueba
```

```

    ...
    return x ** 3 + 4 * x ** 2 - 10 # retorna pol(x) = x^3 + 4x^2 - 10

def trig(x):
    return x*cos(x-1) - sin(x) # retorna trig(x) = xcos(x-1) - sin(x)

def pote(x):
    return pow(7, x) -13 # retorna pote(x) = 7^x - 13

def regula(f, p0, p1, tol, n):
    ...
    :param f: funcion
    :param p0: aproximacion inicial
    :param p1: aproximacion final
    :param tol: tolerancia
    :param n: iteraciones
    :return: p aproximacion a 0 de f
    ...
    i = 0
    while i <= n:
        q0 = f(p0)
        q1 = f(p1)
        p = p1 - (q1*(p1-p0))/(q1 - q0)
        print("Iteracion => {0:<2}, p = {1:.12f}".format(i, p))
        if abs(p-p1) < tol:
            return p
        i += 1
        q = f(p)
        if q*q1 < 0:
            p0 = p1
            q0 = q1
        p1 = p
        q1 = q
    print("Iteraciones agotadas....")
    return None

print("Falsa posicion: pol(x) = x^3 + 4x^2 - 10")
regula(pol, 1, 2, 1e-8, 100)

print("\nFalsa posicin: trig(x) = xcos(x-1) - sin(x)")
regula(trig, 4, 6, 1e-8, 100)
# potencia
print("\nFalsa posicion: pote(x) = 7^x - 13")
regula(pote, 0, 2, 1e-8, 100)

```

**SALIDA.**

Falsa posicion:  $\text{pol}(x) = x^3 + 4x^2 - 10$

```

Iteracion => 0 , p = 1.263157894737
Iteracion => 1 , p = 1.338827838828
Iteracion => 2 , p = 1.358546341825
Iteracion => 3 , p = 1.363547440042
Iteracion => 4 , p = 1.364807031827
Iteracion => 5 , p = 1.365123717884
Iteracion => 6 , p = 1.365203303663
Iteracion => 7 , p = 1.365223301986
Iteracion => 8 , p = 1.365228327026
Iteracion => 9 , p = 1.365229589674
Iteracion => 10, p = 1.365229906941
Iteracion => 11, p = 1.365229986660
Iteracion => 12, p = 1.365230006692
Iteracion => 13, p = 1.365230011725

```

Falsa posicin:  $\text{trig}(x) = x\cos(x-1) - \sin(x)$

```

Iteracion => 0 , p = 5.235657374722
Iteracion => 1 , p = 5.569477410510
Iteracion => 2 , p = 5.597623035312
Iteracion => 3 , p = 5.599220749873
Iteracion => 4 , p = 5.599307996036
Iteracion => 5 , p = 5.599312749633
Iteracion => 6 , p = 5.599313008600
Iteracion => 7 , p = 5.599313022708
Iteracion => 8 , p = 5.599313023477

```

Falsa posicion:  $\text{pote}(x) = 7^x - 13$

```

Iteracion => 0 , p = 0.500000000000
Iteracion => 1 , p = 0.835058241104
Iteracion => 2 , p = 1.045169783424
Iteracion => 3 , p = 1.168847080360
Iteracion => 4 , p = 1.238197008244
Iteracion => 5 , p = 1.275862101477
Iteracion => 6 , p = 1.295933755010
Iteracion => 7 , p = 1.306516642232
Iteracion => 8 , p = 1.312064439413
Iteracion => 9 , p = 1.314963818299
Iteracion => 10, p = 1.316476640694
Iteracion => 11, p = 1.317265325509
Iteracion => 12, p = 1.317676311539
Iteracion => 13, p = 1.317890428220
Iteracion => 14, p = 1.318001965936
Iteracion => 15, p = 1.318060064553
Iteracion => 16, p = 1.318090326416

```

```

Iteracion => 17, p = 1.318106088665
Iteracion => 18, p = 1.318114298545
Iteracion => 19, p = 1.318118574700
Iteracion => 20, p = 1.318120801950
Iteracion => 21, p = 1.318121962020
Iteracion => 22, p = 1.318122566245
Iteracion => 23, p = 1.318122880958
Iteracion => 24, p = 1.318123044876
Iteracion => 25, p = 1.318123130253
Iteracion => 26, p = 1.318123174722
Iteracion => 27, p = 1.318123197884
Iteracion => 28, p = 1.318123209948
Iteracion => 29, p = 1.318123216231

```

## II. Ecuaciones no lineales.

### A. Método de bisección.

```

# Método de bisección
from math import *

def pol(x):
    '''Funcion de prueba'''
    return x**3 + 4*x**2 -10    # retorna polinomio de  $pol(x) = x^3 + 4x^2 -10$ 

def trig(x):
    '''Funcion de prueba'''
    return x*cos(x-1) - sin(x) # Retorna  $trig(x) = x\cos(x-1) - \sin(x)$ 

def bisec(f, a, b, tol, n):
    '''
    Implementacion del metodo de biseccion
    Entradas:
        f    -- funcion
        a    -- inicio del intervalo
        b    -- fin del intervalo
        tol  -- tolerancia
        n    -- numero maximo de iteraciones
    Salida:
        p aproximacion a cero de f
        None en caso de iteraciones agotadas
    '''
    i = 1
    while i <= n:
        p = a + (b-a) / 2
        print("i = {0:<2}, p = {1:.12f}".format(i,p))
        if abs(f(p)) <= 1e-15 or (b-a)/2 < tol:
            return p
    
```

```

    i += 1
    if f(a)*f(p) > 0:
        a = p
    else:
        b = p
print("Iteraciones agotadas: Error!")
return None

# pol(x), a=1, b=2, Tol = 10^-8, N0 = 100
print("Biseccion Funcion pol(x) = x^3 + 4x^2 -10: ")
bisec(pol, 1, 2, 1e-8, 100)

# trig(x), a = 4, b = 6, Tol=10^-8, N0 = 100
print("Biseccino funcion trig(x) = xcos(x-1) - sen(x): ")
bisec(trig, 4, 6, 1e-8, 100)

```

### **SALIDA**

```

Biseccion Funcion pol(x) = x^3 + 4x^2 -10:
i = 1 , p = 1.500000000000
i = 2 , p = 1.250000000000
i = 3 , p = 1.375000000000
i = 4 , p = 1.312500000000
i = 5 , p = 1.343750000000
i = 6 , p = 1.359375000000
i = 7 , p = 1.367187500000
i = 8 , p = 1.363281250000
i = 9 , p = 1.365234375000
i = 10, p = 1.364257812500
i = 11, p = 1.364746093750
i = 12, p = 1.364990234375
i = 13, p = 1.365112304688
i = 14, p = 1.365173339844
i = 15, p = 1.365203857422
i = 16, p = 1.365219116211
i = 17, p = 1.365226745605
i = 18, p = 1.365230560303
i = 19, p = 1.365228652954
i = 20, p = 1.365229606628
i = 21, p = 1.365230083466
i = 22, p = 1.365229845047
i = 23, p = 1.365229964256
i = 24, p = 1.365230023861
i = 25, p = 1.365229994059
i = 26, p = 1.365230008960
i = 27, p = 1.365230016410

```

```

Bisección función trig(x) = xcos(x-1) - sen(x):

```

```

i = 1 , p = 5.000000000000
i = 2 , p = 5.500000000000
i = 3 , p = 5.750000000000
i = 4 , p = 5.625000000000
i = 5 , p = 5.562500000000
i = 6 , p = 5.593750000000
i = 7 , p = 5.609375000000
i = 8 , p = 5.601562500000
i = 9 , p = 5.597656250000
i = 10, p = 5.599609375000
i = 11, p = 5.598632812500
i = 12, p = 5.599121093750
i = 13, p = 5.599365234375
i = 14, p = 5.599243164062
i = 15, p = 5.599304199219
i = 16, p = 5.599334716797
i = 17, p = 5.599319458008
i = 18, p = 5.599311828613
i = 19, p = 5.599315643311
i = 20, p = 5.599313735962
i = 21, p = 5.599312782288
i = 22, p = 5.599313259125
i = 23, p = 5.599313020706
i = 24, p = 5.599313139915
i = 25, p = 5.599313080311
i = 26, p = 5.599313050508
i = 27, p = 5.599313035607
i = 28, p = 5.599313028157

```

## B. Método de falsa posición.

```
# Algoritmo Posicion Falsa
```

```
from math import *
```

```
def pol(x):
```

```
    '''
```

```
    Funcion de prueba
```

```
    '''
```

```
    return x ** 3 + 4 * x ** 2 - 10 # retorna pol(x) = x^3 + 4x^2 - 10
```

```
def trig(x):
```

```
    return x*cos(x-1) - sin(x) # retorna trig(x) = xcos(x-1) - sin(x)
```



```

def pote(x):
    return pow(7, x) - 13    # retorna pote(x) = 7^x - 13

def regula(f, p0, p1, tol, n):
    """
    :param f:    funcion
    :param p0:    aproximacion inicial
    :param p1:    aproximacion final
    :param tol:    tolerancia
    :param n:    iteraciones
    :return:    p aproximacion a 0 de f
    """
    i = 0
    while i <= n:
        q0 = f(p0)
        q1 = f(p1)
        p = p1 - (q1*(p1-p0))/(q1 - q0)
        print("Iteracion => {0:<2}, p = {1:.12f}".format(i, p))
        if abs(p-p1) < tol:
            return p
        i += 1
        q = f(p)
        if q*q1 < 0:
            p0 = p1
            q0 = q1
        p1 = p
        q1 = q
    print("Iteraciones agotadas....")
    return None

print("Falsa posicion: pol(x) = x^3 + 4x^2 - 10")
regula(pol, 1, 2, 1e-8, 100)

print("Falsa posicin: trig(x) = xcos(x-1) - sin(x)")
regula(trig, 4, 6, 1e-8, 100)
# potencia
print("Falsa posicion: pote(x) = 7^x - 13")
regula(pote, 0, 2, 1e-8, 100)

```

## SALIDA

```

Falsa posicion: pol(x) = x^3 + 4x^2 - 10
Iteracion => 0 , p = 1.263157894737
Iteracion => 1 , p = 1.338827838828
Iteracion => 2 , p = 1.358546341825
Iteracion => 3 , p = 1.363547440042
Iteracion => 4 , p = 1.364807031827
Iteracion => 5 , p = 1.365123717884
Iteracion => 6 , p = 1.365203303663

```

```

Iteracion => 7 , p = 1.365223301986
Iteracion => 8 , p = 1.365228327026
Iteracion => 9 , p = 1.365229589674
Iteracion => 10, p = 1.365229906941
Iteracion => 11, p = 1.365229986660
Iteracion => 12, p = 1.365230006692
Iteracion => 13, p = 1.365230011725

```

Falsa posicin:  $\text{trig}(x) = x \cos(x-1) - \sin(x)$

```

Iteracion => 0 , p = 5.235657374722
Iteracion => 1 , p = 5.569477410510
Iteracion => 2 , p = 5.597623035312
Iteracion => 3 , p = 5.599220749873
Iteracion => 4 , p = 5.599307996036
Iteracion => 5 , p = 5.599312749633
Iteracion => 6 , p = 5.599313008600
Iteracion => 7 , p = 5.599313022708
Iteracion => 8 , p = 5.599313023477

```

Falsa posicion:  $\text{pote}(x) = 7^x - 13$

```

Iteracion => 0 , p = 0.500000000000
Iteracion => 1 , p = 0.835058241104
Iteracion => 2 , p = 1.045169783424
Iteracion => 3 , p = 1.168847080360
Iteracion => 4 , p = 1.238197008244
Iteracion => 5 , p = 1.275862101477
Iteracion => 6 , p = 1.295933755010
Iteracion => 7 , p = 1.306516642232
Iteracion => 8 , p = 1.312064439413
Iteracion => 9 , p = 1.314963818299
Iteracion => 10, p = 1.316476640694
Iteracion => 11, p = 1.317265325509
Iteracion => 12, p = 1.317676311539
Iteracion => 13, p = 1.317890428220
Iteracion => 14, p = 1.318001965936
Iteracion => 15, p = 1.318060064553
Iteración => 16, p = 1.318090326416
Iteración => 17, p = 1.318106088665
Iteración => 18, p = 1.318114298545
Iteración => 19, p = 1.318118574700
Iteracion => 20, p = 1.318120801950
Iteracion => 21, p = 1.318121962020
Iteracion => 22, p = 1.318122566245
Iteracion => 23, p = 1.318122880958
Iteracion => 24, p = 1.318123044876
Iteracion => 25, p = 1.318123130253

```

```

Iteracion => 26, p = 1.318123174722
Iteracion => 27, p = 1.318123197884
Iteracion => 28, p = 1.318123209948
Iteracion => 29, p = 1.318123216231

```

### C. Método de Newton/Raphson.

```
# Método de Newton-Raphson
```

```
import numpy as np
```

```
# INGRESO
```

```

fx = lambda x: x**3 + 4*(x**2) - 10 # ESTA ES LA EXPRESIÓN
dfx = lambda x: 3*(x**2) + 8*x          # ESTA ES LA EXPRESIÓN DE LA
DERIVADA

```

```

x0 = 2          # Extremo derecho del intervalo
tolera = 0.001  # Tolerancia

```

```
# 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)

```

**SALIDA**

```
['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
```

**D. Método de Newton/Raphson Varias Variables.**

```
import numpy as np
#Definicion de la funcion en el vector x
def F(x) :
    f1=x[0]**2+x[1]**2-1
    f2=4*x[0]**2/9+4*x[1]**2-1
    return np.array([f1,f2])
#Definicion de la derivada de la funcion
def dF(x):
    return np.array([[2*x[0],2*x[1]],
                     [8*x[0]/9,8*x[1]]])
N=100
x=np.array([1,1])
#Iteracion de newton-raphson
for k in range(N) :
    xold=x
    Jinv=np.linalg.inv(dF(x))
    x=x-np.dot(Jinv,F(x))
    e=np.linalg.norm(x-xold)
    print(k,x,e)
    if e<1e-10:
        break
```

```
In [4]: runfile('/Users/usuario/untitled0.py', wdir='/Users/usuario')
0 [0.921875 0.578125] 0.42904781930456193
1 [0.91856462 0.42419764] 0.15396295748966032
2 [0.91855865 0.39627005] 0.027927588737214515
3 [0.91855865 0.39528593] 0.0009841144731082554
4 [0.91855865 0.39528471] 1.2250389103685144e-06
5 [0.91855865 0.39528471] 1.8982038195996022e-12

In [5]:
```

### III. Interpolación.

#### A. Método de lagrange.

```
# implementacion de interpolacion de Lagrange
def LagrangePol(datos):
    """
    :param datos: lista de puntos (x,y) en el plano
    :return p: funcion de interpolacion
    """
    def L(k,x):
        '''Implementacion de la funcion L_k(x)'''
        out = 1
        for i, p in enumerate(datos):
            if i != k:
                out *= (x-p[0])/(datos[k][0]- p[0])
        return out

    def P(x):
        '''Implementacion del polinomio P(x)'''
        lag = 0
        for k, p in enumerate(datos):
            lag += p[1]*L(k,x)
        return lag

    return P

# datos para f(x)=1/2 con x0=2, x1=2.75 y x2=4
datosf = [(2,1/2), (11/4,4/11), (4,1/4)]
Pf = LagrangePol(datosf)
print("Polinomio de Lagrange en x=3: ")
print("{0:.12f}".format(Pf(3)))

# datos g(x)=sin(3x), x0=1, x1=1.3, x2=1.6, x3=1.9, x4=2.2
datosg = [(1, 0.1411), (1.3, -0.6878), (1.6, -0.9962), (1.9,-0.5507), (2.2,
0.3115)]
Pg = LagrangePol(datosg)
print("Polinomio de lagrange en x=1.5")
print("{0:.12f}".format(Pg(1.5)))
```

#### SALIDA.

```
Polinomio de Lagrange en x=3:
0.329545454545
Polinomio de lagrange en x=1.5
-0.977381481481.
```

#### B. Método de Newton.

```
# Metodo de Newton - Diferencias divididas de newton
```

```

import numpy as np
import sympy as sym
import matplotlib.pyplot as plt

# Ingreso de datos
xi = np.array([3.2, 3.8, 4.2, 4.5])
fi = np.array([5.12, 6.42, 7.25, 6.85])

# Diferencias divididas avanzadas
titulo = ['i', 'xi', 'fi']
n = len(xi)
ki = np.arange(0, n, 1)
tabla = np.concatenate([ki, xi, fi], axis=0)
tabla = np.transpose(tabla)

# diferencias divididas vacias
dfinita = np.zeros(shape=(n, n), dtype=float)
tabla = np.concatenate((tabla, dfinita), axis=1)

# calcular tabla inicia en columna 3
[n, m] = np.shape(tabla)
diagonal = n - 1
j = 3
while j < m:
    titulo.append('F[' + str(j - 2) + ']')
    i = 0
    paso = j - 2 # inicia en 1
    while i < diagonal:
        denominador = (xi[i + paso] - xi[i])
        numerador = tabla[i + 1, j - 1] - tabla[i, j - 1]
        tabla[i, j] = numerador / denominador
        i = i + 1
    diagonal -= 1
    j += 1

# Polinomio de diferencias divididas
# caso: puntos equidistantes en eje x
dDividida = tabla[0, 3:]
n = len(dfinita)

# expresion del polinomio con sympy
x = sym.Symbol('x')
polinomio = fi[0]
for j in range(1, n, 1):
    factor = dDividida[j - 1]
    termino = 1
    for k in range(0, j, 1):
        termino = termino * (x - xi[k])
    polinomio = polinomio + termino * factor

```

```

# Simplifica multiplicando entre (x-xi)
polisimple = polinomio.expand()

# polinomio para evaluacion numerica
px = sym.lambdify(x, polisimple)

# Puntos para la grafica

muestras = 101
a = np.min(xi)
b = np.max(xi)
pxi = np.linspace(a, b, muestras)
pfi = px(pxi)

# Salida
np.set_printoptions(precision = 4)
print('Tabla Diferencia Dividida')
print([titulo])
print(tabla)
print('dDividida: ')
print(dDividida)
print('polinomio: ')
print(polinomio)
print('polinomio simplificado: ' )
print(polisimple)

# Grafica
plt.plot(xi,fi, 'o', label = 'Puntos')
plt.plot(pxi,pfi, label = 'Polinomio')
plt.legend()
plt.xlabel('xi')
plt.ylabel('fi')
plt.title('Diferencias Divididas - Metodo de Newton')
plt.show()

```

### **SALIDA.**

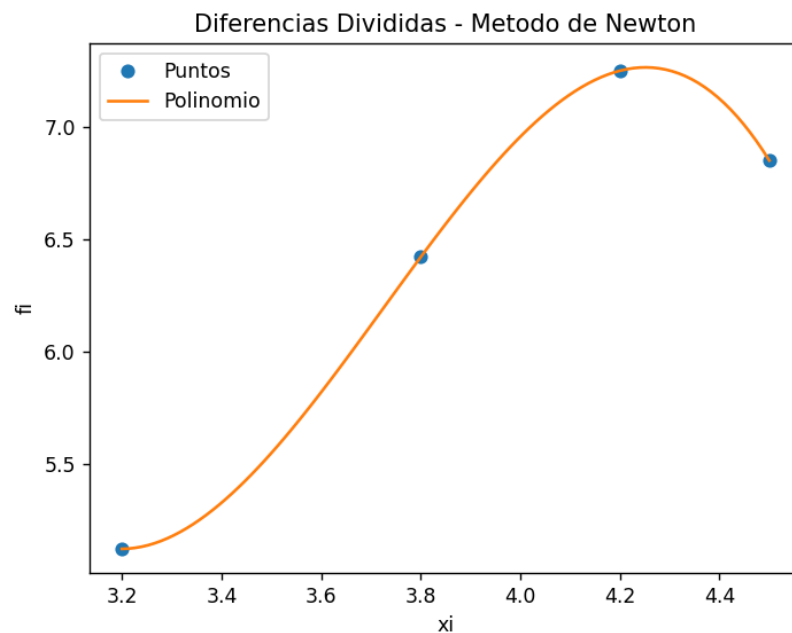
```

Tabla Diferencia Dividida
[['i', 'xi', 'fi', 'F[1]', 'F[2]', 'F[3]', 'F[4]']]
[[ 0.      3.2      5.12      2.1667 -0.0917 -3.6749  0.      ]
 [ 1.      3.8      6.42      2.075  -4.869    0.      0.      ]
 [ 2.      4.2      7.25     -1.3333  0.        0.        0.        ]
 [ 3.      4.5      6.85      0.         0.         0.         0.        ]]
dDividida:
[ 2.1667 -0.0917 -3.6749  0.      ]
polinomio:
2.16666666666667*x - 3.67490842490842*(x - 4.2)*(x - 3.8)*(x - 3.2) -
0.0916666666666694*(x - 3.8)*(x - 3.2) - 1.81333333333334
polinomio simplificado:
-3.67490842490842*x**3 + 41.0673076923077*x**2 - 149.920860805861*x +

```

184.756923076923

figure 1



### C. Ajuste de polinomio por mínimos cuadrados.

```
#Primero tenemos que importar las librerías
import numpy as np
from numpy.polynomial import polynomial as P

# y ponemos la coordenada en x
x = np.linspace(-1,1,51)

# Mostramos la coordenada
print("X Coordenada...\n",x)

# y ponemos la coordenada en y
y = x**3 - x + np.random.randn(len(x))
print("\nY Coordenada...\n",y)

# usamos polynomial.polyfit() en Python Numpy para obtener el ajuste
polinomial
c, stats = P.polyfit(x,y,3,full=True)
print("\nResultado...\n",c)
print("\nResultado...\n",stats)
```



**Salida**

```
X Coordenada...
[ -1.    -0.96 -0.92 -0.88 -0.84 -0.8   -0.76 -0.72 -0.68 -0.64 -0.6   -0.56
 -0.52 -0.48 -0.44 -0.4   -0.36 -0.32 -0.28 -0.24 -0.2   -0.16 -0.12 -0.08
 -0.04  0.    0.04 0.08  0.12  0.16  0.2   0.24  0.28  0.32  0.36  0.4
  0.44  0.48  0.52 0.56  0.6   0.64  0.68  0.72  0.76  0.8   0.84  0.88
  0.92  0.96  1.   ]
```

```
Y Coordenada...
[ 0.4388579 -1.3508368  0.52616587  0.50493563 -0.41832357 -0.2853106
 -0.5068614 -0.8329285  0.76090406  0.00609587  1.12769331  0.9222503
 -0.31084757 -0.29451488  2.91848798  2.25702408  2.34855352  0.84745102
 -0.66801849  1.0340859 -0.86058488 -0.47499975 -0.7170237  2.11051398
 -0.96796432  1.51405996 -3.07453779 -0.89772079 -1.47189885  0.49479143
  0.27481024 -1.08478724 -1.08010424 -0.76667944  1.74030219 -0.97809896
  0.32968012 -0.69500776  0.36409974  0.65551068 -0.45497642 -1.00934222
 -2.35276293  1.02819382  0.90095176 -0.27077846 -0.75396276 -0.96076418
  0.00515708  0.05898137  0.3219597 ]
```

```
Resultado...
[ 0.04263471 -1.49237787 -0.1257064  1.85541669]
```

```
Resultado...
[array([61.7252889]), 4, array([1.38446749, 1.32119158, 0.50443316, 0.28853036]), 1.1
```

**D. Interpoladores cúbicos.**

```
# SPLINE CÚBICO
import numpy as np
import sympy as sym
def CubicNatural(xi, yi):
    m = xi.size
    n = m-1
    a = np.zeros(m)
    b = np.zeros(n)
    c = np.zeros(m)
    d = np.zeros(n)
    for i in range(m):
        a[i] = yi[i]
    h = np.zeros(n)
    for i in range(n):
        h[i] = xi[i+1] - xi[i]
    alfa = np.zeros(n)
    alfa[0] = 0
    for i in range(1,n):
        alfa[i] = 3*(a[i+1]-a[i])/h[i]-3*(a[i]-a[i-1])/h[i-1]
    l=np.zeros(m)
```

```

z=np.zeros(m)
u=np.zeros(n)
l[0] = 1
z[0] = 0
u[0] = 0
for i in range(1,n):
    l[i] = 2*(xi[i+1]-xi[i-1])-h[i-1]*u[i-1]
    u[i] = h[i]/l[i]
    z[i] = (alfa[i]-h[i-1]*z[i-1])/l[i]
l[m-1] = 1
z[m-1] = 0
c[m-1] = 0
for i in np.flip(np.arange(n)):
    c[i] = z[i]-u[i]*c[i+1]
    b[i] = (a[i+1]-a[i])/h[i]-h[i]*(c[i+1]+2*c[i])/3
    d[i] = (c[i+1]-c[i])/(3*h[i])

x = sym.Symbol('x')
px_tabla = []
for j in range(0,n,1):

    pxtramo = d[j] * (x-xi[j])**3 + c[j]*(x-xi[j])**2
    pxtramo = pxtramo + b[j]*(x-xi[j])+ a[j]

    px_tabla.append(pxtramo)
return a, b, c, d,px_tabla

```

### EJEMPLO

```

import numpy as np
import matplotlib.pyplot as plt
import SplineCubicoNatural as SC
import sympy as sym

x1 = np.linspace(0, 1.5, 600)
xi = np.array([0,0.5,1,1.5])
f = lambda x: np.cos(3*x**2)*np.log(x**3+1)
fi = f(xi)
f1 = f(x1)
muestras = 10
n = len(xi)
a,b,c,d,px_tabla = SC.CubicNatural(xi,fi)

print('Polinomios por tramos: ')
for tramo in range(1,n,1):
    print('x = [' +str(xi[tramo-1])+',' +str(xi[tramo])+']')
    print(str(px_tabla[tramo-1]))

```

```

xtraza = np.array([])
ytraza = np.array([])
tramo = 1
x0=float(input('Ingrese donde desea evaluar: '))
while not (tramo>=n):
    a = xi[tramo-1]
    b = xi[tramo]
    xtramo = np.linspace(a,b,muestras)
    pxtramo = px_tabla[tramo-1]
    pxt = sym.lambdify('x',pxtramo)
    ytramo = pxt(xtramo)

    xtraza = np.concatenate((xtraza,xtramo))
    ytraza = np.concatenate((ytraza,ytramo))
    tramo = tramo+1
    if a<=x0<=b:
        y0 = pxt(x0)
        print("El valor del polinomio en {} es {}".format(x0,y0))

plt.plot(xi,fi,'ro',label='puntos')
plt.plot(xtraza,ytraza,label = 'trazador', color = 'blue')
plt.plot(x1,f1,label=' funcion',color = 'green')
plt.title('Trazadores Cúbicos Naturales')
plt.xlabel('xi')
plt.ylabel('px(xi)')
plt.legend()

```

### **SALIDA**

```

Polinomios por tramos:
x = [0.0,0.5]
-3.31247027109868*x**3 + 1.00047864003856*x
x = [0.5,1.0]
-1.48387406328545*x + 9.69377871502949*(x - 0.5)**3 - 4.96870540664802*(x - 0.5)**2 + 0
x = [1.0,1.5]
0.817754566338649*x - 6.38130844393081*(x - 1.0)**3 + 9.57196266589622*(x - 1.0)**2 - 1
Ingrese donde desea evaluar: 0.8
El valor del polinomio en 0.8 es -0.5444331441462155

```

## **IV. Cálculo numérico.**

### **A. Derivación e integración de datos tabulados.**

#### **Derivación**

```

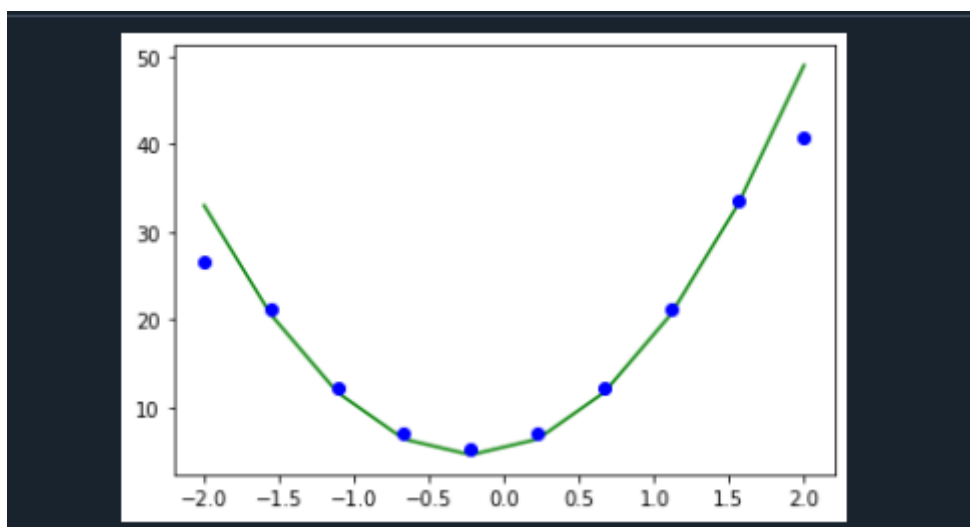
import numpy as np
import matplotlib.pyplot as plt
#Definicion de la funcion f de x

```

```

def f(x) :
    return 3*x**3+2*x**2+5*x+1
N=10
a=-2
b=2
#Vector de x que va de a - b y toma N datos
x=np.linspace(a,b,N)
dx=(b-a)/(N-1)
y=f(x)
#Llena de ceros el arreglo
yp=np.zeros_like(x)
#Condicion del rango que tiene N
for i in range(N) :
    if i==0:
        yp[i]=(y[i+1]-y[i])/dx
    elif i==N-1:
        yp[i]=(y[i]-y[i-1])/dx
    else:
        yp[i]=(y[i+1]-y[i-1])/(2*dx)
#Definición de la funcion fp de x
def fp(x):
    return 9*x**2+4*x+5
#Creacion de la grafica
plt.plot(x,fp(x),
'g-')
plt.plot(x,yp,"bo")
plt. show()

```



### Integración

```

import numpy as np

```

```

import matplotlib.pyplot as plt
from scipy import integrate
#Definimos f de x
def f(x) :
    return 1-x**2
a=10
b=45

N=9
x=np.linspace(a,b,N)
dx=(b-a)/(N-1)
y=f(x)
#Integración
I=integrate.simps(y,x)
#
print(I)

```

```

In [46]: runfile('C:/Users/ERIC/.spyder-py3/untitled0.py', wdir='C:/Users/ERIC/.spyder-py3')
0.6604938271604939

In [47]: runfile('C:/Users/ERIC/.spyder-py3/untitled0.py', wdir='C:/Users/ERIC/.spyder-py3')
-30006.666666666664

```

## E. Derivación e integración de funciones.

### Derivación de una función

```

# -*- coding: utf-8 -*-
"""
Created on Wed Jun 29 13:50:17 2022

@author: carlo
"""

from sympy import Symbol #Manipular las expresiones de forma algebraica
from scipy.misc import derivative
x=Symbol('x') #x
y=2*x**3 #ponemos nuestra expresion
derivada= y.diff(x) #Derivamos respecto a x
print(derivada) #imprimimos la derivada
f=lambda x: 2*x**3 #Calculamos derivada numérica
print(derivative(f,1.0,dx= 1e-8))

```

**SALIDA**

```
6*x**2
5.999999985739635
```

**Integración de una función**

```
# -*- coding: utf-8 -*-
"""
This is a temporary script file.
"""

from sympy import Symbol #importamos Las Librerias
from sympy import integrate#Es la parte que me va hacer la integral de forma simbolica

from scipy.integrate import quad
x=Symbol('x') #X va ser igual a un Simbolo
print(integrate(x**3+x**2+1,x)) #vamos a imprimir el resultado de la
integral que necesitamos integral

                                #el ** significa una potencia
                                #después la variable a la que se va a

integrar que es la x
f=lambda x:x**3+x**2+1          #crear funciones anonimas
print(quad(f,1,2))              #integrales de forma númerica
```

**SALIDA**

```
x**4/4 + x**3/3 + x
(7.08333333333333, 7.864079757761525e-14)
```

**F. Integrador en cuadraturas Gaussianas.**

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return 1-x**2

a=0
b=1
```

```

x=np.array([(1/3)**0.5,-(1/3)**0.5])
w=np.array([1,1])

u=(b-a)*x/2+(b+a)/2
#Con este cambio de variable la integral queda de 1 a 1

I=(b-a)*np.sum(2*f(u))/2

print(I)

```

### Salida

```
1.3333333333333333
```

## II. Ecuaciones diferenciales.

### A. Métodos para resolver una ecuación diferencial, problema de condiciones iniciales.

#### 1. Euler izquierdo.

```

import numpy as np
import matplotlib.pyplot as plt

#Funcion:  $x'(t)=4*x(t)/t$ ,  $x(1)=1$ 
t0=1      #valor inicial de t
tn=2      #valor final de t
x0=1      #valor inicial de x
n=20      #numero de pasos
def EulerI(t0,tn,x0,n):
    t=np.linspace(t0, tn,n+1)
    x=np.zeros(n+1)
    x[0]=x0
    h=(tn-t0)/n      #Tamaño de paso

    #Euler
    for i in range(1,n+1):
        x[i]=x[i-1]*(t[i-1]+h)/(t[i-1]-3*h)
        print(t[i],"\t",format(x[i],'6f'))      #Visualizacion de
resultados
    return (t,x)

#Grafico
(t,x1)= EulerI(t0,tn,x0,n)
plt.plot(t,x1)

```

**SALIDA**

```

1.05  1.235294
1.1    1.509804
1.15   1.827657
1.2    2.193189
1.25   2.610939
1.3    3.085655
1.35   3.622291
1.4    4.226006
1.45   4.902167
1.5    5.656347
1.55   6.494324
1.6    7.422085
1.65   8.445820
1.7000000000000002    9.571930
1.75   10.807018
1.8    12.157895
1.85   13.631579
1.9    15.235294
1.9500000000000002    16.976471
2.0    18.862745

```

**2. Euler centrado.**

```

import numpy as np
from matplotlib import pyplot as plt

#Funcion:  $y'=4*y/x$ ,  $y(0)=1$ 

x0 = 1          #Condicion inicial de x
y0 = 1          #Condicion inicial de y
xf = 2          #Valor final
n = 21          #numero de pasos
h = (xf-x0)/(n-1) #tamaño de paso
x = np.linspace(x0,xf,n)

def f(x,y):
    return (4*y/x)    #Funcion

y = np.zeros([n])    #Vector de y
y[0] = y0

#Euler
for i in range(1,n):
    y[i] = y0+ h*f(x[i],y0)
    y0 = y[i]

```



```

#Visualizacion de resultados
print("x\t\t y")
for i in range(n):
    print(x[i],"\t",format(y[i],'6f'))

#Grafica
plt.plot(x,y)
plt.xlabel("Valor de x")
plt.ylabel("Valor de y")
plt.title("Aproximacion de la Solucion con el metodo de Euler")
plt.show()

```

### SALIDA

x	y	y
1.0	1.000000	
1.05	1.190476	
1.1	1.406926	
1.15	1.651609	
1.2	1.926877	
1.25	2.235178	
1.3	2.579051	
1.35	2.961133	
1.4	3.384152	
1.45	3.850932	
1.5	4.364389	
1.55	4.927536	
1.6	5.543478	
1.65	6.215415	
1.7000000000000002		6.946640
1.75	7.740542	
1.8	8.600602	
1.85	9.530397	
1.9	10.533597	
1.9500000000000002		11.613966
2.0	12.775362	

### 3. Euler derecho.

```

import numpy as np
import matplotlib.pyplot as plt

#Funcion:  $x'(t)=4*x(t)/t$ ,  $x(1)=1$ 

```

```

t0=1      #valor inicial de t
tn=2      #valor final de t
x0=1      #valor inicial de x
n=20      #numero de pasos

def EulerE(f,t0,tn,x0,n):
    t=np.linspace(t0, tn,n+1)
    x=np.zeros(n+1)
    x[0]=x0
    h=(tn-t0)/n

    for i in range(1,n+1):
        x[i]=x[i-1]+h*f(t[i-1],x[i-1])
        print(t[i],"\t",format(x[i],'6f')) #Visualización de resultados

    return (t,x)
def f(t,x):
    return(4*x/t)
(t,x1)= EulerE(f,t0,tn,x0,n)
plt.plot(t,x1)

```

### SALIDA

x	y
1.05	1.200000
1.1	1.428571
1.15	1.688312
1.2	1.981931
1.25	2.312253
1.3	2.682213
1.35	3.094862
1.4	3.553360
1.45	4.060982
1.5	4.621118
1.55	5.237267
1.6	5.913043
1.65	6.652174
1.7000000000000002	7.458498
1.75	8.335968
1.8	9.288650
1.85	10.320723
1.9	11.436477
1.9500000000000002	12.640316
2.0	13.936759

#### 4. Métodos de Runge/Kutta 3er orden.

```
import math
import matplotlib.pyplot as plt

#Funciones
dy = lambda x,y: math.sin(x)**2*y
f = lambda x: 2*math.exp(0.5*(x-math.sin(x)*math.cos(x)))

#Inicialización
xi = 0; xf=5; h = 0.5
n = int((xf-xi)/h)
x = 0; y=2

#Visualizacion
print ('x \t\t y \t\t f(x)'); print ('%f \t %f \t %f'% (x, y, f(x)))
x_plot = []; y_RK3 = []; y_analytical = []

#Metodo de RK 3er orden
for i in range (1,n+1):
    x_plot.append(x); y_RK3.append(y); y_analytical.append(f(x))
    #Calcular aproximaciones de derivadas
    k1 = dy(x,y)
    k2 = dy(x+h/2, y +k1 * h/2)
    k3 = dy(x+h/2,y-k1*h + 2*k2*h)
    #Calcular nuevo valor y estimado
    y = y+1/6*(k1+4*k2+k3)*h
    x=x+h
    print('%f \t %f \t %f'% (x,y,f(x)))

#Graficación
x_plot.append(x); y_RK3.append(y); y_analytical.append(f(x))
plt.plot(x_plot,y_RK3,'ro',x_plot,y_analytical)
plt.xlabel('x'); plt.ylabel('y')
plt.legend(["RK3", 'Analytical'])
```

**Salida**

x	y	f(x)
0.000000	2.000000	2.000000
0.500000	2.051632	2.080856
1.000000	2.536277	2.626948
1.500000	3.906789	4.087229
2.000000	6.344784	6.568909
2.500000	8.748588	8.871805
3.000000	9.576568	9.611892
3.500000	9.639375	9.765937
4.000000	11.154816	11.539866
4.500000	16.305226	17.117778
5.000000	26.714744	27.914673

**5. Métodos de Runge/Kutta 4to orden.**

```

from math import *

def f(t,y):
    func=t*exp(3*t)-2*y
    return func

def RK4(t,y,h,n):
    print('y(',t,')=',y)
    for k in range(n):
        k1=f(t,y)
        k2=f(t+h/2,y+(h/2)*k1)
        k3=f(t+h/2,y+(h/2)*k2)
        k4=f(t+h,y+h*k3)
        y=y+(h/6)*(k1+2*k2+2*k3+k4)
        t=t+h
        print('y(',t,')=',y)

RK4(0,2,0.25,4)

```

**SALIDA**

```

y( 0 )= 2
y( 0.25 )= 1.2592752510620169
y( 0.5 )= 1.0209546536896423
y( 0.75 )= 1.5020804295323198
y( 1.0 )= 3.4971447117878203
>>>

```

**B. Métodos para resolver un sistema de ecuaciones, problema de condiciones iniciales.**

1. Euler izquierdo.
2. Euler centrado.

```
import numpy as np
```

```

import matplotlib.pyplot as plt

# d/dt=-U1
# du1/dt= (-g/l)*sin theta

#Datos
m=1
l=1
g=9.8

#Condiciones Iniciales
t=0
x1=0
y1=0.5
u=np.array([x1,y1])

#Sistema de ecuaciones
def f(u,t):
    return np.array([u[1],-g*np.sin(u[0])/l])

tsol=[t]
xsol=[u[0]]
ysol=[u[1]]
dt=0.25
tfin=10
print('t\t\ttx\t\t\t\t\tty')
while t<tfin:
    u+=f(u,t)*dt
    t+=dt
    xsol.append(u[0])
    ysol.append(u[1])
    tsol.append(t)
    print(t, '\t', u[0], '\t', u[1])

plt.plot(tsol,xsol)
plt.plot(tsol,ysol)
plt.show()

```

### SALIDA

t	x	y
0.25	0.125	0.5
0.5	0.25	0.1945469032061921
0.75	0.29863672580154804	-0.4115927969673891
1.0	0.19573852655970075	-1.1324257867835497
1.25	-0.08736792013618666	-1.608928765729422

1.5	-0.48960011156854216	-1.395149572020305
1.75	-0.8383875045736184	-0.24298068319114252
2.0	-0.8991326753714041	1.578755702216104
2.25	-0.504443749817378	3.4965850219211054
2.5	0.3697025056628983	4.680720368292585
2.75	1.5398825977360446	3.795442136332138
3.0	2.488743131819079	1.3466127249433861
3.25	2.8253963130549256	-0.1416456643533066
3.5	2.789984896966599	-0.9034822918169855
3.75	2.5641143240123525	-1.7472810253268327
4.0	2.127294067680644	-3.0847676634290258
4.25	1.3561021518233878	-5.165087807239443
4.5	0.06483020001352702	-7.558839716217866
4.75	-1.8248797290409395	-7.717562467615766
5.0	-3.7542703459448807	-5.346221930604303
5.25	-5.090825828595957	-6.755119344580663
5.5	-6.779605664741123	-9.031765443707538
5.75	-9.037547025668008	-7.8648768910682705
6.0	-11.003766248435076	-6.939693613458584
6.25	-12.73868965179972	-9.389611406345253
6.5	-15.086092503386034	-8.969516031178538
6.75	-17.32847151118067	-7.542252072365357
7.0	-19.21403452927201	-9.989225384328286
7.25	-21.711340875354082	-9.11589299169354
7.5	-23.990314123277468	-8.439274452612466
7.75	-26.100132736430584	-10.667903146144821
8.0	-28.76710852296679	-8.65055273963404
8.25	-30.9297467078753	-9.80957970329958
8.5	-33.38214163370019	-10.954346610905391
8.75	-36.12072828642654	-8.693400035445524
9.0	-38.294078295287925	-11.143329517364123
9.25	-41.079910674628955	-9.77015113686635
9.5	-43.522448458845545	-10.350633268066296
9.75	-46.11010677586212	-11.437973945607846
10.0	-48.96960026226408	-9.358321276351049

### 3. Euler derecho.

```
import numpy as np
import matplotlib.pyplot as plt

def euler_mod (f , x0 , y0 , x1 , n ) :
    h =( x1 - x0 ) / n # tamaño de paso
    xi = np . zeros ( n +1) # vector de x variable independiente
    yi = np . zeros ( n +1) # vector de y variable dependiente
    xi [0]= x0 # tiempo inicial
```

```

yi [0]= y0 # concentracion inicial

for i in range ( n ) :
    xi [ i +1]= xi [ i ]+ h
    yi [ i +1]= yi [ i ]+ h * f ( xi [ i ] , yi [ i ])
    yi [ i +1]= yi [ i ]+ h *( f ( xi [ i ] , yi [ i ]) + f ( xi [ i
+1] , yi [ i +1]) ) /2

    return xi , yi # Vector de valores calculados

def f (x , y ) :
    return 1.6 -0.02* y

def main () :
    x0 =0 # valor inicial de tiempo
    y0 =50 # valor inicial de la concentracion
    x1 =30 # valor final del tiempo
    n =30 # numero de pasos
    # llamada a la funcion euler
    x , y = euler_mod (f , x0 , y0 , x1 , n )
    print ('x = ',x )
    print ('y = ',y )

    # Grafica
    fig = plt . figure ()
    plt . plot (x ,y , 'o- -', label ='Euler modificado ')
    plt . grid ()
    plt . legend ()
    plt . show ()
    fig . savefig (" edo_euler_mod_mezclas .pdf", bbox_inches ='tight ')

if __name__ == " __main__ ": main ()

```

#### 4. Métodos de Runge/Kutta 3er orden.

#### 5. Métodos de Runge/Kutta 4to orden.

```

import numpy as np
import matplotlib.pyplot as plt

x0=5000
y0=0
a=0
b=20

```

[illegible]

***SALIDA***

```
f=lambda x,y,t: -0.1*x  
g=lambda x,y,t: -0.1*x-0.2*y  
SistRK4(f,g,5000,0,0,20,0.5)
```

x	y
500.0	0.0
-11.907552083333332	487.70338541666666
434.57573256306966	-11.23242488945855
-50.84047265834945	424.2703940185494
375.1406181158895	-49.20785922664566
-85.56212606994654	366.29698096151577
320.80050115542787	-83.07559505884578
-116.496265710894	313.29326295040624
271.0993045515615	-113.24896431452974
-144.02535327553	264.8143792644763
225.62281091469305	-140.1010227199394
-168.4945208538677	220.45629945731585
183.99476759561395	-163.9684144525055



-190.21515574374808	179.8520241473428
145.87335676207314	-185.15486879089354
-209.46814376839095	142.6681419065879
110.94799620278256	-203.93436366890327
-226.50680349873645	108.60170874326042
78.93643974784173	-220.55398773580754
-241.55954071242377	77.37741983219763

### **C. Aplicaciones al problema de condiciones en la frontera.**

#### **1. Ejemplos.**

### ***Conclusiones.***

Para finalizar,nos gustaría agradecer la paciencia que el docente nos tuvo ya que son temas difíciles de explicar y de entender también. Los métodos numéricos tienen diferentes aplicaciones que son necesarias para