



INSTITUTO TECNOLÓGICO DE
OAXACA
TECNOLÓGICO NACIONAL DE MÉXICO

INGENIERÍA EN SISTEMAS COMPUTACIONALES

MÉTODOS NUMÉRICOS

GRUPO 4SC

PROYECTO FINAL

CATEDRÁTICO:

PERALTA REGALADO PEDRO ANTONIO

ELABORÓ:

RICARDEZ REYES MILDRED IVONNE

RAMÍREZ ARREORTÚA LUIS FELIPE

MONTESINOS ORTEGA ALEXIS

HERNANDEZ JUAN JOVANY GAMALIEL

SEMESTRE FEBRERO - JUNIO 2022

27 de junio de 2022

Contenido

Sistema de ecuaciones lineales.....	4
Método de Gauss	4
Factorización LU y PLU.....	7
Matriz inversa	9
Determinantes	11
Gauss Seidel	12
Método de las potencias	16
Ecuaciones no lineales.....	19
Método de bisección:.....	19
Método de falsa posición.....	21
Método de newton/Raphson.....	23
Interpolación.....	25
Método de Lagrange.....	25
Método de newton.....	27
Ajuste De Un Polinomio Por Mínimos Cuadrados.....	30
Usos de la regresión polinomial:	30
Proceso.....	30
Interpoladores cúbicos.....	33
Calculo numérico	37
Derivación e integración de datos tabulados.....	37
Derivación e integración de funciones.....	39
Integrador en cuadraturas Gaussianas.....	40
Ecuaciones diferenciales	41

a) Métodos para resolver una ecuación diferencial, problema de condiciones iniciales.	41
b) Métodos Para Resolver Un Sistema De Ecuaciones, Problema De Condiciones Iniciales.	50

Sistema de ecuaciones lineales.

Método de Gauss

1. Importamos la librería numpy.

```
import numpy as np
```

2. Ingresamos las matrices.

```
A = np.array([[4,2,5],  
              [2,5,8],  
              [5,4,3]])  
  
B = np.array([[60.70],  
              [92.90],  
              [56.30]])
```

3. Evitamos el truncamiento de operaciones.

```
A = np.array(A, dtype=float)
```

4. Obtenemos la matriz aumentada.

```
AB = np.concatenate((A,B), axis=1)  
AB0 = np.copy(AB)
```

5. Realizamos un pivoteo parcial por filas.

```
tamano = np.shape(AB)
n = tamaño[0]
m = tamaño[1]
```

6. Realizamos las operaciones y intercambios.

```
for i in range(0,n-1,1):
    columna = abs(AB[i:,i])
    dondemax = np.argmax(columna)

    if (dondemax !=0):
        temporal = np.copy(AB[i,:])
        AB[i,:] = AB[dondemax+i,:]
        AB[dondemax+i,:] = temporal

AB1 = np.copy(AB)
```

7. Realizamos 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
AB2 = np.copy(AB)
```

8. Realizamos eliminación hacia atrás.

```

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

```

9. Obtenemos diagonal de unos.

```

AB[i,:] = AB[i,]/AB[i,i]
X = np.copy(AB[:,ultcolumna])
X = np.transpose([X])

```

10. Tenemos las siguientes salidas.

```

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)

```

11. Resultados obtenidos.

```

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 ]]
eliminación hacia atrás
[[ 1.00000000e+00  0.00000000e+00 -2.08983158e-16  2.80000000e+00]
 [ 0.00000000e+00  1.00000000e+00  2.61228947e-16  4.50000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  8.10000000e+00]]
solución de X:
[[2.8]
 [4.5]
 [8.1]]

```

Factorización LU y PLU.

1. Importamos la librería numpy en su versión np.

```
import numpy as np
```

2. Imprimimos por consola el letrero que nos dice que realizaremos la factorización LU.

```
print("Descomposicion LU")
```

3. Variable para ingresar el número de renglones.

```
m=int(input("Introduce el número de renglones"))
```

4. Inicializamos una matriz en ceros.

```
matriz=np.zeros([m,m])
```

5. Inicializamos la matriz u en ceros.

```
u=np.zeros([m,m])
```

6. Inicializamos la matriz l en ceros.

```
l=np.zeros([m,m])
```

7. Ingresamos datos a las matrices.

```
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]
```

8. Ahora realizaremos operaciones para hacer 0 debajo de la diagonal.

```
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]
```

9. Asignación de valores a matriz L y U.


```
l[r,k]=factor
```

```
u[r,c]=matriz[r,c]
```

10. Líneas para mostrar los resultados.

```
print("Impresión de resultados ")  
print("Matriz L")  
print(l)  
print("Matriz U")  
print(u)
```

11. Resultados obtenidos.

```
Matriz L  
[[ 1.  0.  0.]  
 [ 5.  1.  0.]  
 [-2.  3.  1.]]  
Matriz U  
[[ 4. -2.  1.]  
 [ 0.  3.  7.]  
 [ 0.  0. -2.]]  
PS D:\felip\Desktop\ph> []
```

Matriz inversa

1. Importamos la librería numpy.

```
import numpy as np
```

2. Definiremos la matriz A.

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

3. Definiremos la matriz b.

```
b=np.array([1,2])
```

4. Calculamos la matriz inversa de A.

```
Ainv=np.linalg.inv(A)
```

5. Realizamos una multiplicación de matrices y obtenemos el resultado.

```
x=np.dot(Ainv,b)
```

6. Podremos ver el resultado con print.

```
print(x)
```

7. Resultado obtenido.

```
PS D:\felip\Desktop\ph> & C:/python.exe d:/felip/Desktop/ph/inversaDeUnaMatriz  
[0.71428571 0.42857143]
```

Determinantes

1. Pedimos al usuario que nos ingrese los valores para el sistema de ecuaciones.

```
a,b,c,d,e,f = input("Ingrese los coeficientes: ").split(',')
a=int(a)
b=int(b)
c=int(c)
d=int(d)
e=int(e)
f=int(f)
```

2. Asignamos un formato para nuestros valores, dando así una similitud a una ecuación lineal.

```
print("{}X+{}Y={}".format(a,b,c))
print("{}X+{}Y={}".format(d,e,f))
```

3. Obtenemos el determinante del sistema.

```
delta=a*e-b*d
```

4. Obtenemos el determinante de X.

```
deltaX=e*c-b*f
```

5. Obtenemos el determinante de Y.

```
deltaY=a*f-c*d
```

6. Obtenemos los valores de X y Y, dividiendo sus respectivos determinantes con el determinante del sistema.

```
X=deltaX/delta  
Y=deltaY/delta
```

7. Imprimimos por consola los valores de X y Y.

```
print ("El valor de X es: ",X)  
print("El valor de Y es: ",Y)
```

8. Resultado obtenido.

```
Ingrese los coeficientes: 2,3,4,5,2,3  
2X+3Y=4  
5X+2Y=3  
El valor de X es: 0.09090909090909091  
El valor de Y es: 1.2727272727272727
```

Gauss Seidel

1. Importamos numpy.

```
import numpy
```

2. Le decimos al usuario que nos ingrese el número de filas que tendrá nuestro sistema de ecuaciones.

```
m=int(input("Valor de m: ")) #
```

3. Le decimos al usuario que nos ingrese el número de columnas que tendrá nuestro sistema de ecuaciones.

```
n=int(input("Valor de n: "))
```

4. Creamos una matriz de ceros con el numero de filas y columnas.

```
matrix = numpy.zeros((m,n))
```

5. Creamos un vector con ceros. Este será nuestro vector solución.

```
x=numpy.zeros((m))
```

6. Creamos otro vector en el cual tendrá la dimensión n, esto quiere decir el número de columnas del sistema de ecuaciones.

```
vector=numpy.zeros((n))
```

7. Creamos otro vector para comprobar las operaciones.

```
comp=numpy.zeros((m))
```

8. Agregamos datos a la matriz.

```

print("Metodo de Gauus Seidel")
print("Introduce los valores de la matriz y del vector solucion")
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("Metodo de Gauus Seidel")

```

9. Variable para obtener la tolerancia que el usuario desea.

```

tol=float(input("Cual es la tolerancia que desea: "))

```

10. Variable para obtener el número máximo de iteraciones.

```

itera=int(input("Cual es el numero máximo de interacciones: "))

```

11. Declaramos un variable en 0.

```

k=0

```

12. Iniciamos la creación del método Gauus Seidel.

```

✓ 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]

```

13. En cada iteración se guardaran los valores de la suma, en matrix en la posición [r,c] y lo multiplicaremos por el vector x en la posición [c].

```
suma=suma+matrix[r,c]*x[c]
```

14. Ahora guardaremos el resultado de la suma en el vector x en la posición [r] y se realizara la operación que se muestra a continuación.

```
x[r]=(vector[r]-suma)/matrix[r,r]
```

15. Creamos otros ciclos for para comprobar las iteraciones y los errores.

```
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 ",r)
if all(i<=tol for i in error) == True:
    break
print("Programa terminado")
```

16. Obtenemos los siguientes resultados, una vez que se realizaron las 10 iteraciones de prueba.

```
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
```

Método de las potencias

1. Creamos dos matrices de ceros y le ingresamos datos.

```
matrizA=zeros(3,3)
matrizU=zeros(3,1)
#Agregando datos a la matriz A
disp('Ingresando datos a matriz A');
for i=1:3
    for j=1:3
        matrizA(i,j)=input(['Ingrese fila ',num2str(i),' columna ',num2str(j),' : '])
    endfor
endfor
#Ingresando datos a la matriz U
disp('Ingresando matriz U')
for i=1:3
    matrizU(i,1)=input(['Ingrese fila ',num2str(i),' columna ',num2str(1),' : '])
endfor
Potencia(matrizA,matrizU)
```

2. En el método de las potencias implementaremos un while hasta que se encuentre el factor de normalización que tienda al valor propio dominante.

```
] while bandera!=false
    filasMU=1
    suma=0
    multiplicacion=0
    for i=1:3
        for j=1:3
            multiplicacion=matrizA(i,j)*matrizU(filasMU,1)
            filasMU=filasMU+1
            suma=suma+multiplicacion
        endfor
        matrizUAuxiliar(i,1)=suma
        suma=0
        filasMU=1
    endfor
    disp('****Vector resultante****')
    for i=1:3
        disp([' ',num2str(matrizUAuxiliar(i,1))])
    end
end
```


3. Multiplicamos la matrizA con la matrizU.

```
multiplicacion=0
for i=1:3
    for j=1:3
        multiplicacion=matrizA(i,j)*matrizU(filasMU,1)
        filasMU=filasMU+1
        suma=suma+multiplicacion
    endfor
    matrizUAuxiliar(i,1)=suma
    suma=0
    filasMU=1
endfor
```

4. Obtenemos el vector resultante.

```
disp('***Vector resultante***')
for i=1:3
    disp([" ",num2str(matrizUAuxiliar(i,1))])
endfor
disp('*****')
```

5. En el vector resultante buscamos el factor de normalización.

```
%% Buscando el elemento de mayor magnitud (factor de normalizacion) en el vector resultante
#buscando el elemento de mayor magnitud (factor de normalizacion) en el vector resultante
for j=1:3
    for i=1:2
        if matrizUAuxiliar(i,1)>matrizUAuxiliar(i+1,1)
            FNormalizacion=matrizUAuxiliar(i,1)
        elseif
            if matrizUAuxiliar(i,1)<matrizUAuxiliar(i+1,1)
                FNormalizacion=matrizUAuxiliar(i+1,1)
            end
        end
    endfor
endfor
#disp(['Factor de normalizacion: ',num2str(FNormalizacion)])
```

6. Dividimos el vector resultante entre el factor de normalización.

```

#diviendiendo entre el factor de normalizacion
for i=1:3
    matrizUAuxiliar(i,1)=matrizUAuxiliar(i,1)/FNormalizacion
    matrizU(i,1)=matrizUAuxiliar(i,1)
endfor

```

7. Con las siguientes líneas verificamos cuando el factor de normalización tiende a un mismo valor y así mismo culminar el programa.

```

if z>=2
    valor1=round(matrizFactorNor(contadorM))
    valor2=round(matrizFactorNor(contadorM+1))
    contadorM=contadorM+1
    if valor1==valor2
        contadorSalir=contadorSalir+1
        if contadorSalir==30
            bandera=false
        endif
    endif
endif
endif
z=z+1

```

8. Resultado del anterior código.

```

*****
*****Vector resultante*****
-0.75
0.75
3
*****

```

Ecuaciones no lineales.

Los métodos numéricos de resolución de ecuaciones no lineales suelen ser métodos iterativos que producen una sucesión de valores aproximados de la solución, que se espera, que converja a la raíz de la ecuación. Estos métodos van calculando las sucesivas aproximaciones sobre la base de los anteriores, a partir de una o varias aproximaciones iniciales.

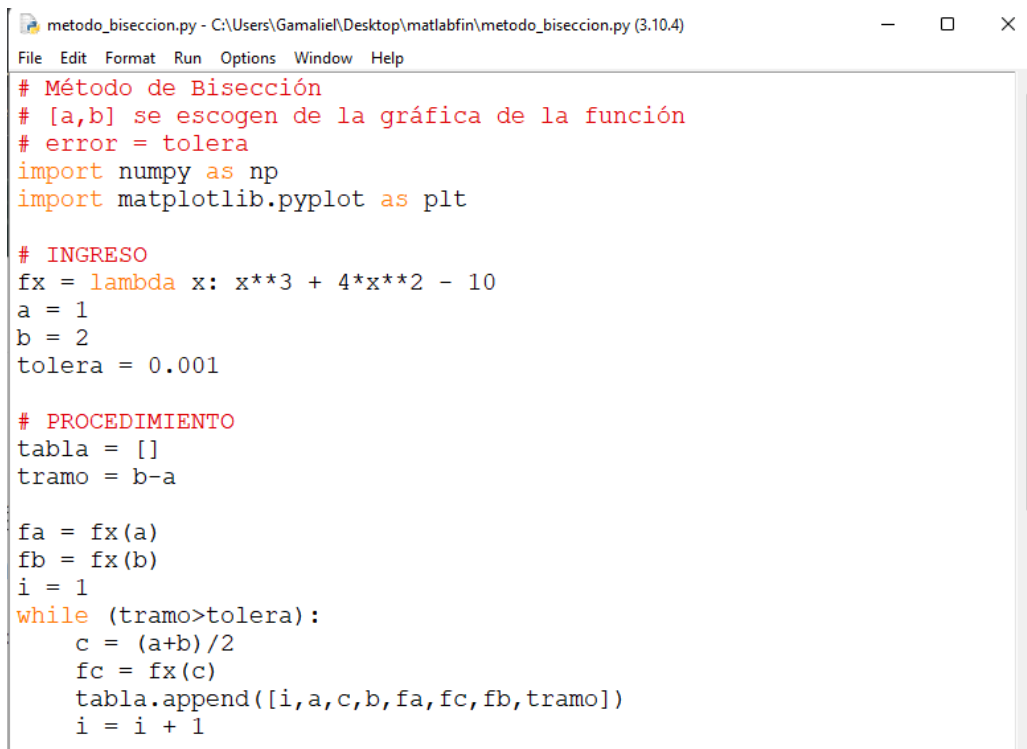
Método de bisección:

El método se basa en el teorema del valor intermedio, conocido como método de la bisección, búsqueda binaria, partición de intervalos o de Bolzano.

Es un tipo de búsqueda incremental en el que:

- ❖ El intervalo se divide siempre en la mitad.
- ❖ Si la función cambia de signo sobre un intervalo, se evalúa el valor de la función en el punto medio.
- ❖ La posición de la raíz se determina en el punto medio del subintervalo, izquierdo o derecho, dentro del cual ocurre un cambio de signo.
- ❖ el proceso se repite hasta obtener una mejor aproximación.

Ejemplo del método de bisección en Python.

A screenshot of a Python script editor window titled 'metodo_biseccion.py - C:\Users\Gamaliel\Desktop\matlabfin\metodo_biseccion.py (3.10.4)'. The window has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The script content is as follows:

```
# Método de Bisección
# [a,b] se escogen de la gráfica de la función
# error = tolera
import numpy as np
import matplotlib.pyplot as plt

# INGRESO
fx = lambda x: x**3 + 4*x**2 - 10
a = 1
b = 2
tolera = 0.001

# PROCEDIMIENTO
tabla = []
tramo = b-a

fa = fx(a)
fb = fx(b)
i = 1
while (tramo>tolera):
    c = (a+b)/2
    fc = fx(c)
    tabla.append([i,a,c,b,fa,fc,fb,tramo])
    i = i + 1
```

```

    cambia = np.sign(fa)*np.sign(fc)
    if (cambia<0):
        b = c
        fb = fc
    else:
        a=c
        fa = fc
        tramo = b-a
    c = (a+b)/2
    fc = fx(c)
    tabla.append([i,a,c,b,fa,fc,fb,tramo])
    tabla = np.array(tabla)

    raiz = c

    # SALIDA
    np.set_printoptions(precision = 4)
    print('[ i, a, c, b, f(a), f(c), f(b), tramo]')
    # print(tabla)

    # Tabla con formato
    n=len(tabla)
    for i in range(0,n,1):
        unafila = tabla[i]
        formato = '{:.0f}'+ ' '+(len(unafila)-1)*'{:.3f} '
        unafila = formato.format(*unafila)
        print(unafila)

    orden = np.argsort(xi)
    xi = xi[orden]
    yi = yi[orden]

    plt.plot(xi,yi)
    plt.plot(xi,yi,'o')
    plt.axhline(0, color="black")

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Bisección en f(x)')
    plt.grid()
    plt.show()

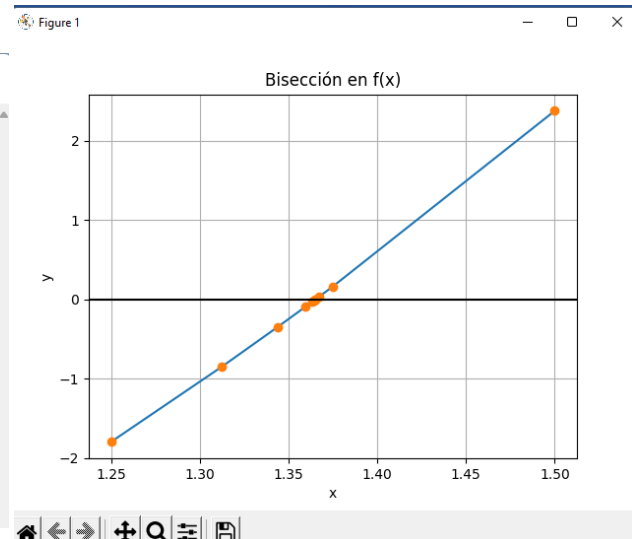
```

Resultados

```

Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Gamaliel\Desktop\matlabfin\metodo_biseccion.py =====
[ i, a, c, b, f(a), f(c), f(b), tramo]
1 1.000 1.500 2.000 -5.000 2.375 14.000 1.000
2 1.000 1.250 1.500 -5.000 -1.797 2.375 0.500
3 1.250 1.375 1.500 -1.797 0.162 2.375 0.250
4 1.250 1.312 1.375 -1.797 -0.848 0.162 0.125
5 1.312 1.344 1.375 -0.848 -0.351 0.162 0.062
6 1.344 1.359 1.375 -0.351 -0.096 0.162 0.031
7 1.359 1.367 1.375 -0.096 0.032 0.162 0.016
8 1.359 1.363 1.367 -0.096 -0.032 0.032 0.008
9 1.363 1.365 1.367 -0.032 0.000 0.032 0.004
10 1.363 1.364 1.365 -0.032 -0.016 0.000 0.002
11 1.364 1.365 1.365 -0.016 -0.008 0.000 0.001
raiz: 1.36474609375

```



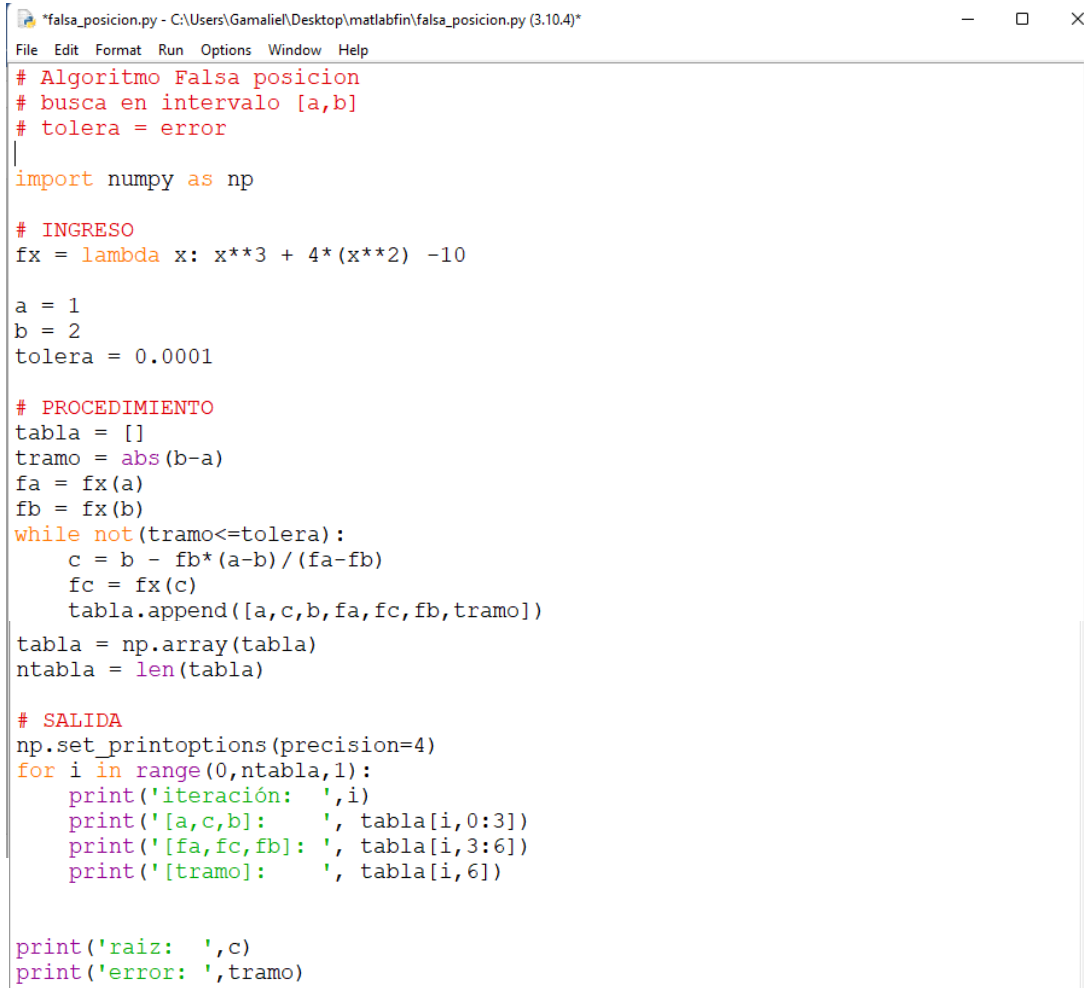
Método de falsa posición.

El método de la posición falsa, falsa posición, regla falsa o regla falsa considera dividir el intervalo cerrado $[a,b]$ donde se encontraría una raíz de la función $f(x)$ basado en la cercanía a cero que tenga $f(a)$ o $f(b)$.

El método une $f(a)$ con $f(b)$ con una línea recta, la intersección de la recta con el eje x representaría una mejor aproximación hacia la raíz.

Al reemplazar la curva de $f(x)$ por una línea recta, se genera el nombre de «posición falsa» de la raíz. El método también se conoce como interpolación lineal.

Ejemplo del método de falsa posición en Python.

A screenshot of a Python script titled 'falsa_posicion.py' in a text editor. The script implements the False Position method to find the root of the function $f(x) = x^3 + 4x^2 - 10$ on the interval $[1, 2]$ with a tolerance of 0.0001. It uses a while loop to iteratively refine the interval and a for loop to print the results of each iteration, including the current interval $[a, b]$, function values $f(a)$, $f(b)$, the new root estimate c , and the current error (tramo).

```
*falsa_posicion.py - C:\Users\Gamaliel\Desktop\matlabfin\falsa_posicion.py (3.10.4)*
File Edit Format Run Options Window Help
# Algoritmo Falsa posicion
# busca en intervalo [a,b]
# tolera = error
|
import numpy as np

# INGRESO
fx = lambda x: x**3 + 4*(x**2) -10

a = 1
b = 2
tolera = 0.0001

# PROCEDIMIENTO
tabla = []
tramo = abs(b-a)
fa = fx(a)
fb = fx(b)
while not(tramo<=tolera):
    c = b - fb*(a-b)/(fa-fb)
    fc = fx(c)
    tabla.append([a,c,b,fa,fc,fb,tramo])
tabla = np.array(tabla)
ntabla = len(tabla)

# SALIDA
np.set_printoptions(precision=4)
for i in range(0,ntabla,1):
    print('iteración: ',i)
    print('[a,c,b]: ', tabla[i,0:3])
    print('[fa,fc,fb]: ', tabla[i,3:6])
    print('[tramo]: ', tabla[i,6])

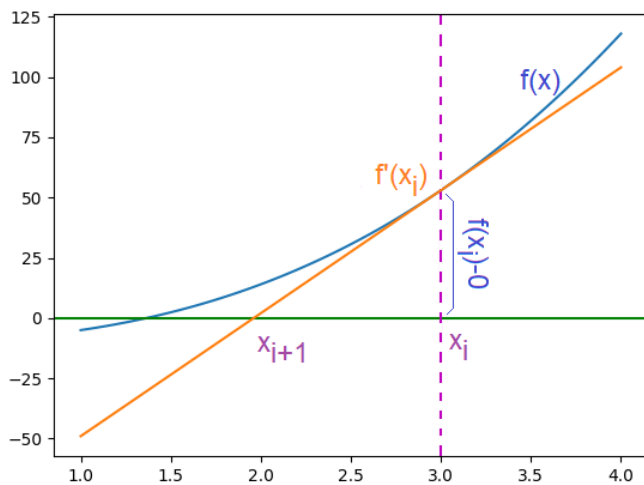
print('raiz: ',c)
print('error: ',tramo)
```

Resultados

```
IDLE Shell 3.10.4
File Edit Shell Debug Options Window Help

===== RESTART: C:\Users\Gamaliel\Desktop\matlabfin\falsa_posicion.py =====
iteración: 0
[a,c,b]: [1.      1.2632 2.    ]
[fa,fc,fb]: [-5.      -1.6023 14.    ]
[tramo]: 1.0
iteración: 1
[a,c,b]: [1.2632 1.3388 2.    ]
[fa,fc,fb]: [-1.6023 -0.4304 14.    ]
[tramo]: 0.26315789473684204
iteración: 2
[a,c,b]: [1.3388 1.3585 2.    ]
[fa,fc,fb]: [-0.4304 -0.11  14.    ]
[tramo]: 0.0756699440909967
iteración: 3
[a,c,b]: [1.3585 1.3635 2.    ]
[fa,fc,fb]: [-0.11  -0.0278 14.    ]
[tramo]: 0.019718502996940224
iteración: 4
[a,c,b]: [1.3635 1.3648 2.    ]
[fa,fc,fb]: [-2.7762e-02 -6.9834e-03  1.4000e+01]
[tramo]: 0.005001098217311428
iteración: 5
[a,c,b]: [1.3648 1.3651 2.    ]
[fa,fc,fb]: [-6.9834e-03 -1.7552e-03  1.4000e+01]
[tramo]: 0.0012595917846898175
iteración: 6
[a,c,b]: [1.3651 1.3652 2.    ]
[fa,fc,fb]: [-1.7552e-03 -4.4106e-04  1.4000e+01]
[tramo]: 0.0003166860575976038
raiz: 1.3652033036626001
error: 7.958577822231305e-05
```

Método de newton/Raphson.



Se deduce a partir de la interpretación gráfica o por medio del uso de la serie de Taylor.

De la gráfica, se usa el triángulo formado por la recta tangente que pasa por $f(x_i)$, con pendiente $f'(x_i)$ y el eje x .

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

El punto x_{i+1} es la intersección de la recta tangente con el eje x , que es más cercano a la raíz de $f(x)$, valor que es usado para la próxima

iteración.

Reordenando la ecuación de determina la fórmula para el siguiente punto:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

El error se determina como la diferencia entre los valores sucesivos encontrados $|x_{i+1} - x_i|$

Ejemplo del método de Newton Raphson en Python.

```
newton_raphson.py - C:\Users\Gamaliel\Desktop\matlabfin\newton_raphson.py (3.10.4)
File Edit Format Run Options Window Help
# Método de Newton-Raphson

import numpy as np

# INGRESO
fx = lambda x: x**3 + 4*(x**2) - 10
dfx = lambda x: 3*(x**2) + 8*x

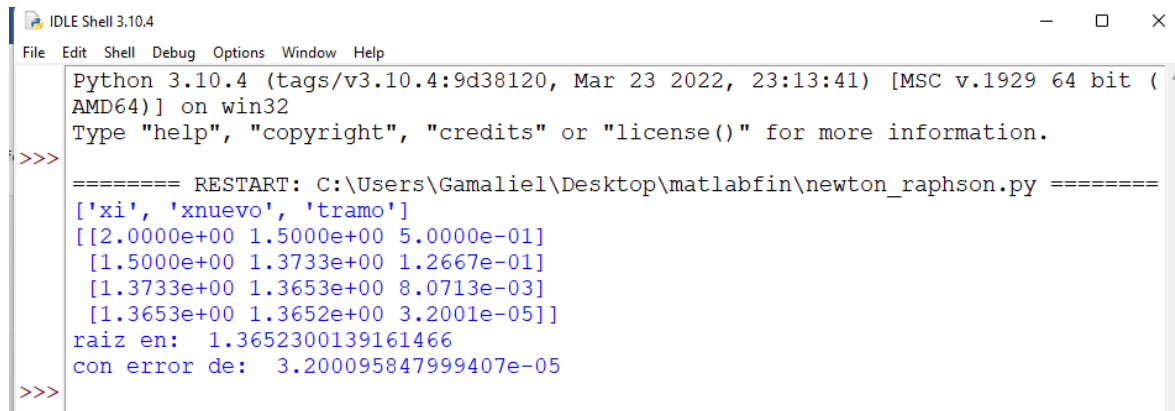
x0 = 2
tolera = 0.001

# 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)
```

RESULTADOS



```
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Gamaliel\Desktop\matlabfin\newton_raphson.py =====
['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
>>>
```


Interpolación.

En análisis numérico, la interpolación polinómica (o polinomial) es una técnica de interpolación de un conjunto de datos o de una función por un polinomio. Es decir, dado cierto número de puntos obtenidos por muestreo o a partir de un experimento se pretende encontrar un polinomio que pase por todos los puntos.

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.

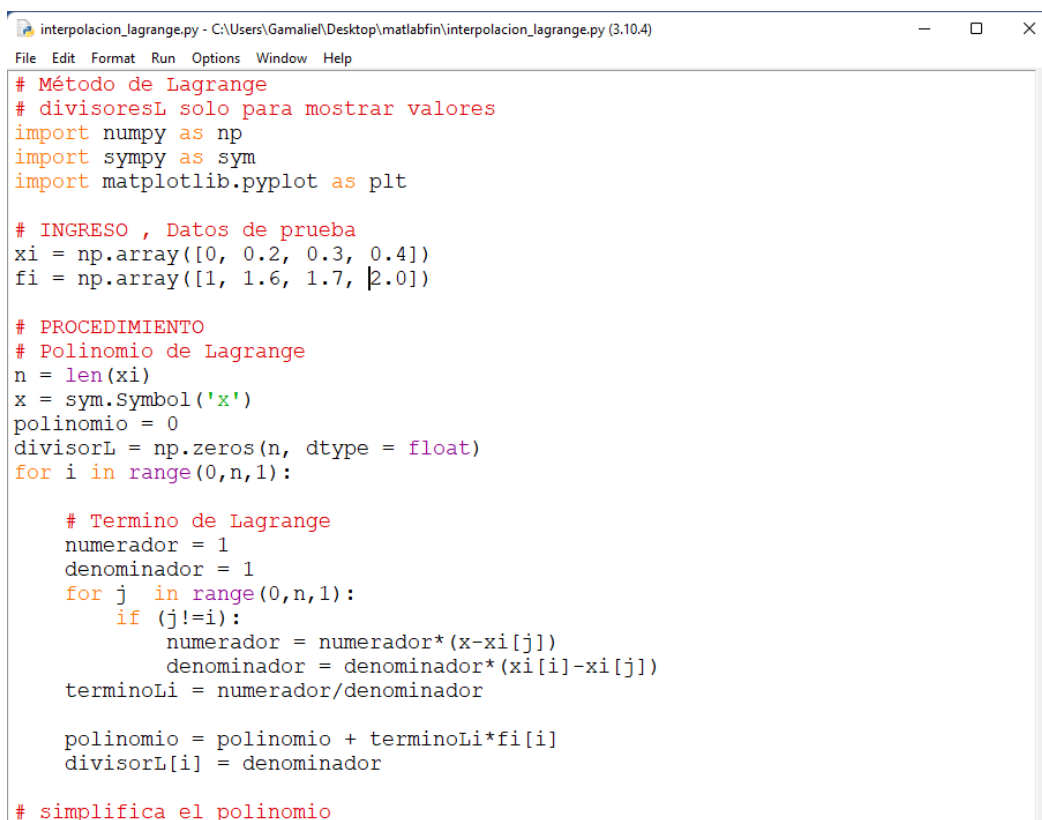
El polinomio de Lagrange se construye a partir de las fórmulas:

$$f_n(x) = \sum_{i=0}^n L_i(x) f(x_i)$$

$$L_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

Donde una vez que se han seleccionado los puntos a usar, se generan la misma cantidad de términos que puntos.

Ejemplo del método de Lagrange en Python.



```
interpolacion_lagrange.py - C:\Users\Gamaliel\Desktop\matlabfin\interpolacion_lagrange.py (3.10.4)
File Edit Format Run Options Window Help

# Método 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()

```

RESULTADOS.

Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

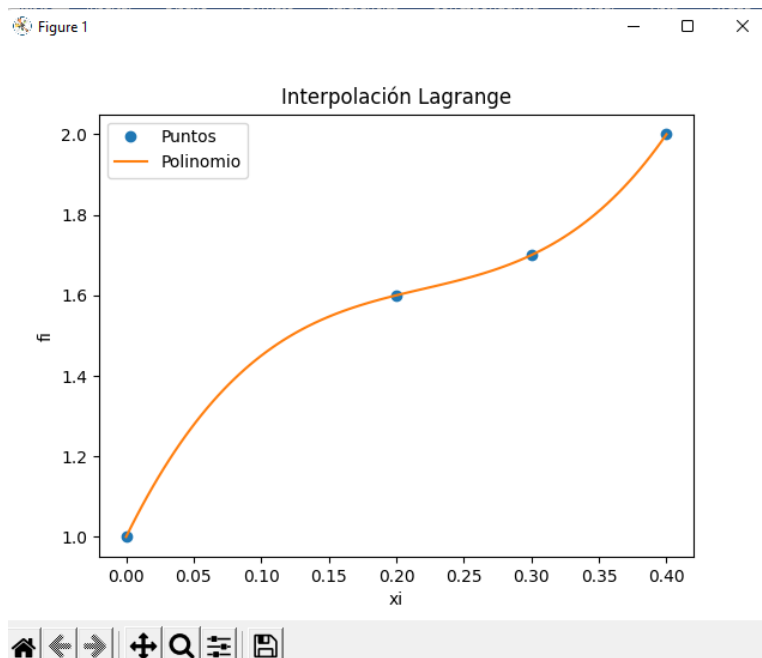
```

>>>
==== RESTART: C:\Users\Gamaliel\Desktop\matlabfin\interpolacion_lagrange.py ====
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

```



Método de newton.

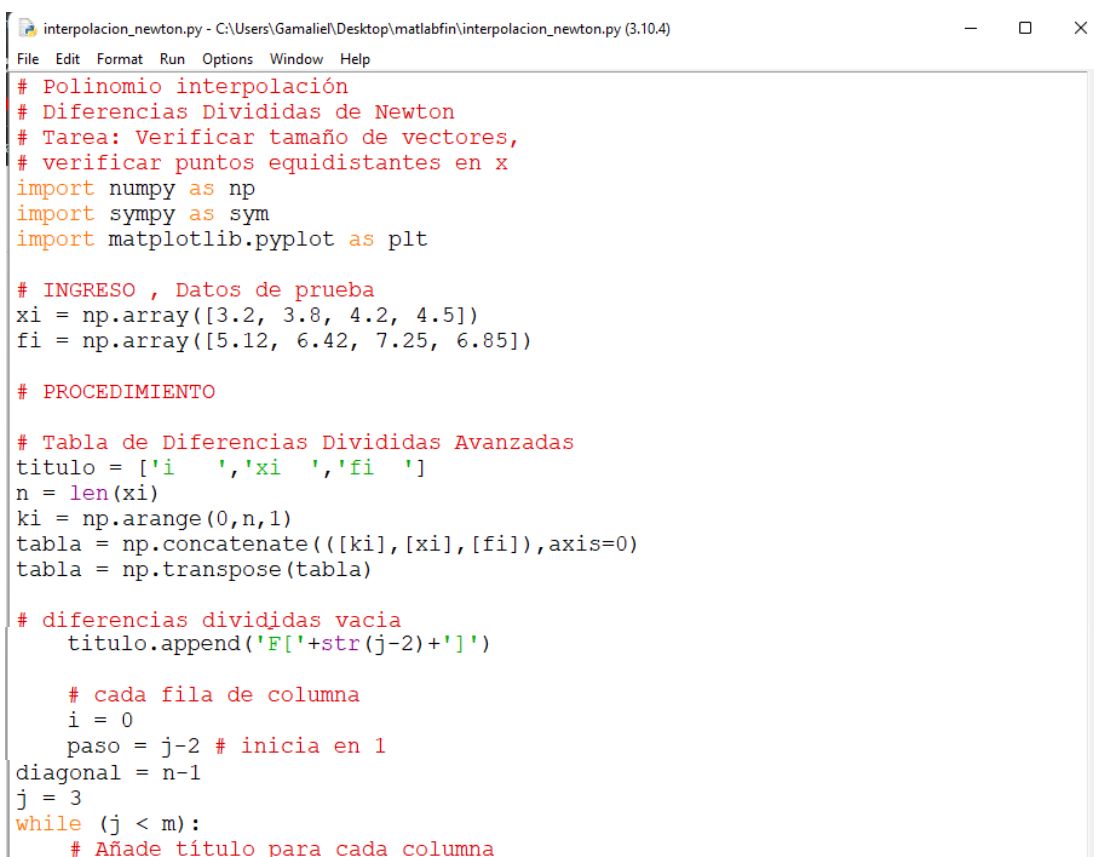
El método se usa en el caso que los puntos en el «eje x» se encuentran espaciados de forma arbitraria y provienen de una función desconocida pero supuestamente diferenciable.

El algoritmo para interpolación de diferencias divididas o Newton, considera reutilizar el procedimiento de cálculo de diferencias finitas incorporando la parte del denominador.

La creación de la expresión del polinomio también es semejante a la usada para diferencias finitas avanzadas.

Se incorpora la parte gráfica para observar los resultados en el intervalo xi, con el número de muestras = 101 para tener una buena resolución de la línea del polinomio.

Ejemplo del método de Newton en Python

A screenshot of a Python script titled 'interpolacion_newton.py' in a text editor. The script implements Newton's method of divided differences. It starts with imports for numpy, sympy, and matplotlib. It defines input arrays for xi and fi. It then constructs a table of divided differences. The script is partially visible, showing the initial setup and the beginning of the table construction loop.

```
interpolacion_newton.py - C:\Users\Gamaliel\Desktop\matlabfin\interpolacion_newton.py (3.10.4)
File Edit Format Run Options Window Help
# Polinomio interpolación
# Diferencias Divididas de Newton
# Tarea: Verificar tamaño de vectores,
# verificar puntos equidistantes en x
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt

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

# PROCEDIMIENTO

# Tabla de 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 vacia
titulo.append('F['+str(j-2)+'']')

# cada fila de columna
i = 0
paso = j-2 # inicia en 1
diagonal = n-1
j = 3
while (j < m):
    # Añade título para cada columna
```

```

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 = diagonal - 1
j = j+1

# POLINOMIO con diferencias Divididas
# caso: puntos equidistantes en eje x
dDividida = tabla[0,3:]
n = len(dDividida)

# expresión 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 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
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)

# Gráfica
plt.plot(xi,fi,'o', label = 'Puntos')
##for i in range(0,n,1):
##    plt.axvline(xi[i],ls='--', color='yellow')
plt.plot(pxi,pfi, label = 'Polinomio')
plt.legend()
plt.xlabel('xi')
plt.ylabel('fi')
plt.title('Diferencias Divididas - Newton')
plt.show()

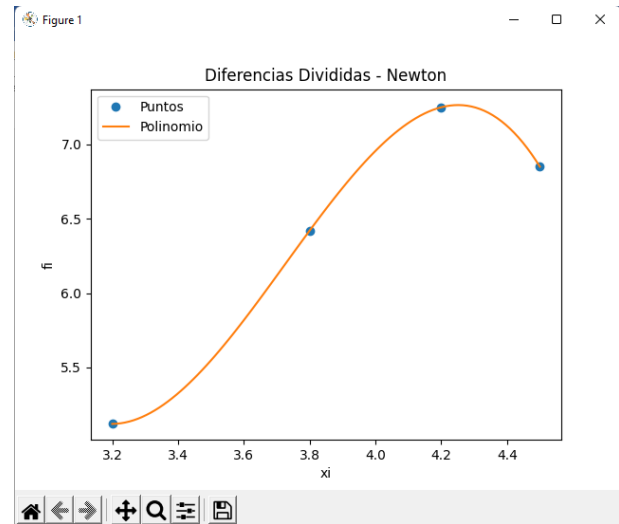
```

RESULTADOS.

```

Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Gamaliel\Desktop\matlabfin\interpolacion_newton.py =====
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:
2.166666666666667*x - 3.67490842490842*(x - 4.2)*(x - 3.8)*(x - 3.2) - 0.09166666
66666694*(x - 3.8)*(x - 3.2) - 1.813333333333334
polinomio simplificado:
-3.67490842490842*x**3 + 41.0673076923077*x**2 - 149.920860805861*x + 184.756923
076923

```



Ajuste De Un Polinomio Por Mínimos Cuadrados

Par darle solución a este problema de ajuste, entendimos el tema de regresión polinomial cual le damos el concepto de: es un modelo de análisis de regresión en el que la relación entre la variable independiente X y la variable dependiente Y se modela con un polinomio de n -ésimo grado en X . Los modelos de regresión polinomial suelen ajustarse mediante el método de mínimos cuadrados, debido a ciertas condiciones y características de la regresión polinomial, se le considera un caso especial de regresión lineal múltiple.

El método de mínimos cuadrados, minimiza la varianza de los estimadores insesgados de los coeficientes, bajo las condiciones del teorema de Gauss-Markov. El método de los mínimos cuadrados fue publicado en 1805 por Legendre y en 1809 por Gauss. Aunque la regresión polinomial es técnicamente un caso especial de regresión lineal múltiple, la interpretación de un modelo de regresión polinomial ajustado requiere una perspectiva algo diferente.

Usos de la regresión polinomial:

- ✓ Se utilizan básicamente para definir o describir fenómenos no lineales como:
- ✓ Tasa de crecimiento de tejidos.
- ✓ Progresión de las epidemias de enfermedades
- ✓ Distribución de isótopos de carbono en sedimentos lacustres
- ✓ El objetivo básico del análisis de regresión es modelar el valor esperado de una variable dependiente y en términos del valor de una variable independiente x

Proceso

- 1) Paso 1: Importar bibliotecas y conjunto de datos. Importe las bibliotecas importantes y el conjunto de datos que estamos usando para realizar la regresión polinomial.
- 2) Paso 2: dividir el conjunto de datos en 2 componentes. Divida el conjunto de datos en dos componentes que sean X e yX contendrá la columna entre 1 y 2. y contendrá la columna 2.
- 3) Paso 3: ajustar la regresión lineal al conjunto de datos. Ajuste del modelo de regresión lineal en dos componentes.

- 4) Paso 4: ajuste de la regresión polinomial al conjunto de datos. Ajuste del modelo de regresión polinomial en dos componentes X e y.
- 5) Paso 5: En este paso estamos visualizando los resultados de la regresión lineal usando un diagrama de dispersión.
- 6) Paso 6: Visualización de los resultados de la regresión polinomial mediante un diagrama de dispersión.
- 7) Paso 7: predecir un nuevo resultado con regresión lineal y polinomial.

```

1  #Paso 1: Importar bibliotecas y conjunto de datos
2  import numpy as np
3  from sklearn import datasets, linear_model
4  import matplotlib.pyplot as plt
5  boston = datasets.load_boston()
6  print(boston)
7  print("Información en el dataset:")
8  print(boston.keys())
9  print("Características del dataset:")
10 print(boston.DESCR)
11 print("Cantidad de datos:")
12 print(boston.data.shape)
13 print("Nombres columnas:")
14 print(boston.feature_names)
15 #Paso 2: dividir el conjunto de datos en 2 componentes
16 X_p = boston.data[:, np.newaxis, 5]
17 y_p = boston.target
18 #Paso 3: ajustar la regresión lineal al conjunto de datos.
19 plt.scatter(X_p, y_p)
20 plt.show()
21 # Paso 3: ajustar la regresión lineal al conjunto de datos.
22 from sklearn.model_selection import train_test_split
23 #Paso4: ajuste de la regresión polinomial al conjunto de datos
24 X_train_p, X_test_p, y_train_p, y_test_p = train_test_split(X_p, y_p, test_size=0.2)
25 from sklearn.preprocessing import PolynomialFeatures

```

```

23 #Paso4: ajuste de la regresión polinomial al conjunto de datos
24 X_train_p, X_test_p, y_train_p, y_test_p = train_test_split(X_p, y_p, test_size=0.2)
25 from sklearn.preprocessing import PolynomialFeatures
26 #Paso 5: En este paso estamos visualizando los resultados de la regresión lineal usando un diagrama de dispersión.
27 poli_reg = PolynomialFeatures(degree = 2)
28 X_train_poli = poli_reg.fit_transform(X_train_p)
29 X_test_poli = poli_reg.fit_transform(X_test_p)
30 #Paso6: Visualización de los resultados de la regresión polinomial mediante un diagrama de dispersión.
31 pr = linear_model.LinearRegression()
32 pr.fit(X_train_poli, y_train_p)
33 #Paso 7: predecir un nuevo resultado con regresión lineal y polinomial.
34 Y_pred_pr = pr.predict(X_test_poli)
35 #Graficamos los datos junto con el modelo
36 plt.scatter(X_test_p, y_test_p)
37 plt.plot(X_test_p, Y_pred_pr, color='red', linewidth=3)
38 plt.show()
39 print()
40 print('DATOS DEL MODELO REGRESIÓN POLINOMIAL')
41 print()
42 print('Valor de la pendiente o coeficiente "a":')
43 print(pr.coef_)
44 print('Valor de la intersección o coeficiente "b":')
45 print(pr.intercept_)
46 print('Precisión del modelo:')
47 print(pr.score(X_train_poli, y_train_p))

```

Para la regresión polinomial ocupamos la librería sklearn, es un conjunto de rutinas escritas en Python para hacer análisis predictivo, que incluyen clasificadores, algoritmos de clusterización, etc. Está basada en NumPy, SciPy y matplotlib, de forma que es fácil reaprovechar el código que

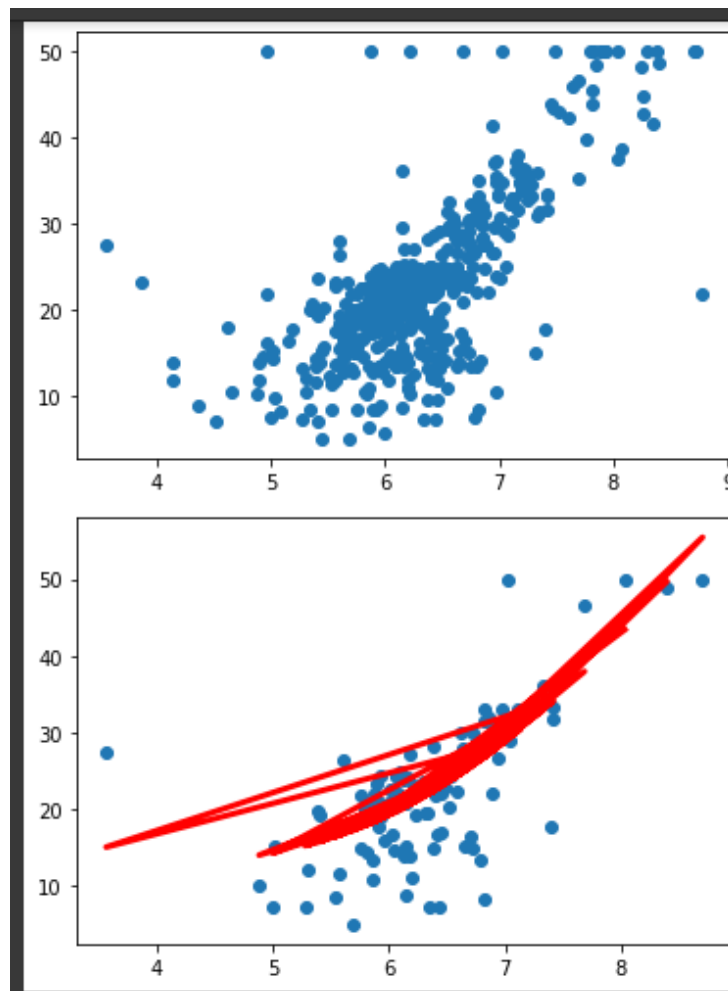
use estas librerías. De esta librería sacamos los datos, para la traficación, de hecho, es posible verla al ejecutar la primera parte del proceso así:

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

datas = pd.read_csv('data.csv')
datas
```

Out[3]:

	sno	Temperature	Pressure
0	1	0	0.0002
1	2	20	0.0012
2	3	40	0.0060
3	4	60	0.0300
4	5	80	0.0900
5	6	100	0.2700



REPRESENTACION FINAL DEL MODELO Y AJUSTE DEL POLINOMIO POR LOS MINIMOS CUADRADOS

Interpoladores cúbicos

El spline cúbico ($k=3$) es el spline más empleado, debido a que proporciona un excelente ajuste a los puntos tabulados y su cálculo no es excesivamente complejo.

$$[t_0, t_1], [t_1, t_2], \dots, [t_{n-1}, t_n]$$

Sobre cada intervalo, S está definido por un polinomio cúbico diferente. Sea S_i el polinomio cúbico que representa a S en el intervalo $[t_i, t_{i+1}]$, por tanto:

$$S(x) = \begin{cases} S_0(x) & x \in [t_0, t_1) \\ S_1(x) & x \in [t_1, t_2) \\ \vdots & \vdots \\ S_{n-1}(x) & x \in [t_{n-1}, t_n) \end{cases}$$

Los polinomios S_{i-1} y S_i interpolan el mismo valor en el punto t_i , es decir, se cumple:

$$S_{i-1}(t_i) = y_i = S_i(t_i) \quad (1 \leq i \leq n-1)$$

Por lo que se garantiza que S es continuo en todo el intervalo. Además, se supone que S' y S'' son continuas, condición que se emplea en la deducción de una expresión para la función del spline cúbico.

Aplicando las condiciones de continuidad del spline S y de las derivadas primera S' y segunda S'' , es posible encontrar la expresión analítica del spline.

La función spline resultante se denomina *spline cúbico natural* y el sistema de ecuaciones lineal expresado en forma matricial es:

$$\begin{pmatrix} u_1 & h_1 & & & & \\ h_1 & u_2 & h_2 & & & \\ & h_2 & u_3 & h_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & h_{n-2} & u_{n-1} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_{n-2} \\ z_{n-3} \end{pmatrix} = \begin{pmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \\ \vdots \\ \nu_{n-2} \\ \nu_{n-1} \end{pmatrix}$$

```
[1] 1 try:
2     | from pip import main as pipmain
3     | except:
4     | from pip._internal import main as pipmain
5     | pipmain(['install','gekkko'])
6
7     | from gekko import GEKKO
8     | import numpy as np
9     | import matplotlib.pyplot as plt
10
11     | xm = np.array([0,1,2,3,4,5])
12     | ym = np.array([0.1,0.2,0.3,0.5,1.0,0.9])
13
14     | m = GEKKO()
15     | m.x = m.Param(value=np.linspace(-1,6))
16     | m.y = m.Var()
17     | m.options.IMODE=2
18     | m.cspline(m.x,m.y,xm,ym)
19     | m.solve(dispatch=False)
20
21     | p = GEKKO()
22     | p.x = p.Var(value=1,lb=0,ub=5)
23     | p.y = p.Var()
24     | p.cspline(p.x,p.y,xm,ym)
25     | p.Obj(-p.y)
```

En las primeras 5 líneas podemos ver la instalación de GEKKO, una librería que nos ayuda a graficar y optimizar el programa y de esta manera tener un método más completo.

Después de instalar e importar librerías, creamos dos arrays, para los puntos.

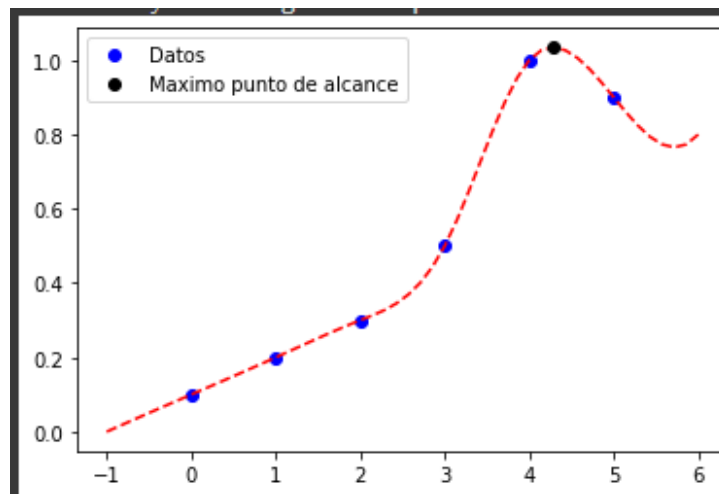
Aquí podemos ver como gracias a GEKKO, podemos determinar los parámetros y la función de la slider cubico.

```
20
21 p = GEKKO()
22 p.x = p.Var(value=1,lb=0,ub=5)
23 p.y = p.Var()
24 p.cspline(p.x,p.y,xm,ym)
25 p.Obj(-p.y)
26 p.solve(dispatch=False)
27
28 plt.plot(xm,ym,'bo',label='Datos')
29 plt.plot(m.x.value,m.y.value,'r--',label='')
30 plt.plot(p.x.value,p.y.value,'ko',label='Maximo punto de alcance')
31 plt.legend(loc='best')
32 plt.show()
```

Debido a que tenemos tanto valores para x como para y, se realizar ambos parámetros, esto para obtener el valor máximo que después de gráfica, es apreciada en la línea 30, junto a los

valores de los arreglos previamente declarados, y por supuesto en los arreglos anteriormente.

RESULTADO FINAL INTERPOLACION CUBICA.



Para finalizar tenemos un ejemplo de los tipos de interpolaciones y regresiones

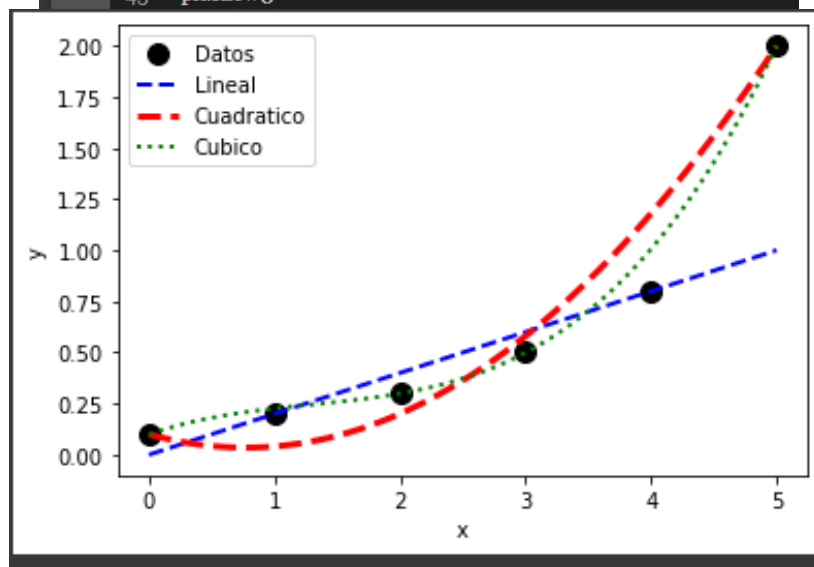
```
1 from gekko import GEKKO
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 xm = np.array([0,1,2,3,4,5])
6 ym = np.array([0.1,0.2,0.3,0.5,0.8,2.0])
7
8 m = GEKKO()
9 m.options.IMODE=2
10 # coeficientes
11 c = [m.FV(value=0) for i in range(4)]
12 x = m.Param(value=xm)
13 y = m.CV(value=ym)
14 y.FSTATUS = 1
15 # Modelo polinomial
16 m.Equation(y==c[0]+c[1]*x+c[2]*x**2+c[3]*x**3)
17
18 # Regresion Lineal
19 c[0].STATUS=1
20 c[1].STATUS=1
21 m.solve(disg=False)
22 p1 = [c[1].value[0],c[0].value[0]]
23
24 # Cuadratica
25 c[2].STATUS=1
```

Podemos ver la implementación de los tres temas antes mencionados, así como sus semejanzas y conexiones.

```

23
24 # Cuadratica
25 c[2].STATUS=1
26 m.solve(dis=False)
27 p2 = [c[2].value[o],c[1].value[o],c[o].value[o]]
28
29 # Cubica
30 c[3].STATUS=1
31 m.solve(dis=False)
32 p3 = [c[3].value[o],c[2].value[o],c[1].value[o],c[o].value[o]]
33
34 # plot fit
35 plt.plot(xm,ym,'ko',markersize=10)
36 xp = np.linspace(0,5,100)
37 plt.plot(xp,np.polyval(p1,xp),'b--',linewidth=2)
38 plt.plot(xp,np.polyval(p2,xp),'r--',linewidth=3)
39 plt.plot(xp,np.polyval(p3,xp),'g:',linewidth=2)
40 plt.legend(['Datos','Lineal','Cuadratico','Cubico'],loc='best')
41 plt.xlabel('x')
42 plt.ylabel('y')
43 plt.show()

```



Calculo numérico

Derivación e integración de datos tabulados.

```
1 import numpy as np #LIBRERIAS
2 import pylab as pl
3 from sympy import *
4 import pandas as pd
5 pd.read_csv
6
7 valido = False #VALIDACION
8 def validar(valor):
9     entero = int(valor)
10    return entero >= 0 and entero <= 10
11
12 validoB = False
13 def validarB(valorB):
14     enteroD = int(valorB)
15    return enteroD >= 0 and enteroD <= 10
16
17 x=[0,1,2,3,4,5,6,7,8,9,10] #ARREGLO
18 y=[0,1,4,9,16,25,36,49,64,81,100]
19
20 df = pd.DataFrame({'1stcolumn':x, '2ndcolumn':y}) # TABULACION
21 print(df)
22
23 pl.plot(x, y) #GRAFICACION
24 pl.show()
```

como sus valores X_1, X_2, X_n y Y_1, Y_2, Y_n . Posteriormente graficamos esos mismos datos sobre el plano cartesiano, con los valores X y Y

Para la integración de datos tabulados, lo primero que se realizó para representarlos fueron los arreglos para tabular los datos, cuando tabulamos los datos, es decir creamos tablas que estas en formato X, Y ; así

```
25
26 print('-----') # IDENTIFICACION FUNCION
27 print('Funcion calculada x**2, equis al cuadrado')
28 print('Derivada') # DERIVADA
29 x = sp.Symbol('x')
30 y = x**2
31 sp.diff(y,x)
32 print(sp.diff(y,x))
33
34 print('Integral') # INTEGRAL
35 print('Valor a ') # A
36 while not valido:
37     valor = input()
38     valido = validar(valor)
39     if not valido:
40         print('Lo siento, el valor no es válido, vuelva a intentarlo: ', end='')
41     print(f'El valor válido es {valor}.')
42
43 print('Valor b ')
44 #bucle para pedir el valor
45 while not validoB:
46     valorB = input()
47     validoB = validarB(valorB)
48     if not validoB:
49         print('Lo siento, el valor no es válido, vuelva a intentarlo: ', end='') # B
50
```

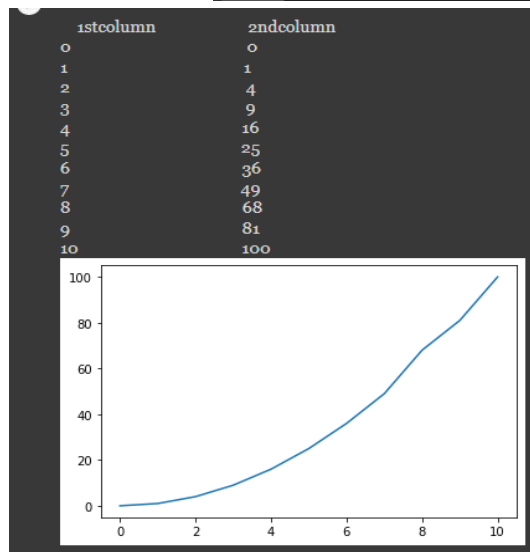
Después de eso identificamos cual es la función que se va obtener la derivada, para la derivada y la integral utilizamos las librerías que nos ofrece PYTHON no decidimos crear nuevas funciones, que ya teníamos.

Las funciones validan los datos de entrada para los intervalos no mayor a 10 ni menos a 0.

```

42
43 print('Valor b ')
44 #bucle para pedir el valor
45 while not validoB:
46     valorB = input()
47     validoB= validarB(valorB)
48     if not validoB:
49         print('Lo siento, el valor no es válido, vuelva a intentarlo: ', end=")#B
50
51 print(f'El valor válido es {valorB}.')
52 f = x**2
53 print(sp.integrate(f))
54 sp.integrate(y,(x,valor,valorB))
55

```



```

-----
Funcion calculada x**2, equis al cuadrado
Derivada
2*x
Integral
Valor a
5
El valor válido es 5.
Valor b
9
El valor válido es 9.
x**3/3
604
-----
3

```

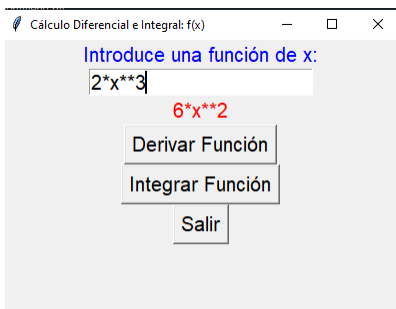
Aquí podemos ver el resultado final de la integración y tabulación de datos tabulados.

Derivación e integración de funciones.

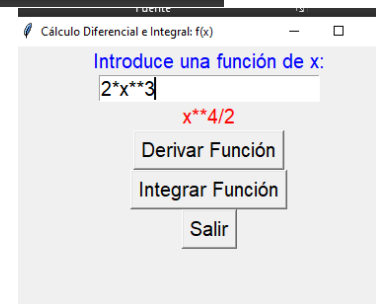
```
1 from sympy import *
2 from sympy.parsing.sympy_parser import parse_expr # Leer función introducida
3 from tkinter import *
4
5 def derivada():
6     try:
7         x = symbols('x') #Declarar variable independiente
8         fun_escrita = funcion.get()
9         f = parse_expr(fun_escrita)
10        derivada = diff(f,x)
11        etiqueta.configure(text=derivada)
12    except:
13        etiqueta.configure(text="Introduce la función correctamente")
14
15
16 def integral():
17     try:
18         x = symbols('x') #Declarar variable independiente
19         fun_escrita2 = funcion.get()
20         g = parse_expr(fun_escrita2)
21         integral = integrate(g,x)
22
23
24
25
26 ventana = Tk()
27 ventana.geometry('400x280')
28 ventana.title("Cálculo Diferencial e Integral: f(x)")
29
30 anuncio = Label(ventana, text="Introduce una función de x:", font=("Arial", 15), fg="blue")
31 anuncio.pack()
32
33 funcion = Entry(ventana, font=("Arial", 15))
34 funcion.pack()
35
36 etiqueta = Label(ventana, text="Resultado", font=("Arial", 15), fg="red")
37 etiqueta.pack()
38
39 boton1 = Button(ventana, text="Derivar Función", font=("Arial", 15), command=derivada)
40 boton1.pack()
41
42 boton2 = Button(ventana, text="Integrar Función", font=("Arial", 15), command=integral)
43 boton2.pack()
44
45 def _quit(): #Función salir
46     ventana.quit() # detiene mainloop
47     ventana.destroy() # elimina la ventana
48
49
50 boton3 = Button(master=ventana, text="Salir", font=("Arial", 15), command=_quit)
51 boton3.pack()
52
53 ventana.mainloop()
```

Para este ejercicio se decidió implementar una interface grafica con la que se pudiera introducir una función cualquiera, siempre y cuando tuviera la sintaxis de PYTHON, es decir: Si es x^2 en la calculadora seria `x**2` sin esto no se podría utilizar.

Esta pequeña y última parte es la ventana emergente creada para esta práctica, podemos ver que tuene 3 botones, uno para derivadas, integrales y por supuesto la salida



Ya para finalizar tenemos la derivada de $2 \cdot x^{**3}$. Así como su integral como ejemplo claro de lo fácil que fue derivar e integral la función sin tabular datos anteriormente.



Integrador en cuadraturas Gaussianas

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import integrate
4
5 def f(x):
6     return 1-x**2
7
8 a=0
9 b=1
10
11 x=np.array([(1/3)**0.5-(1/3)**0.5])
12 w=np.array([1,1])
13 u=(b-a)*x/2+(a+b)/2
14
15 I= (b-a)+np.sum(w*f(u))/2
16
17 print(I)
18
```

1.75

Una cuadratura de Gauss n , es una cuadratura construida para obtener el resultado exacto al integrar polinomios de grado $2n-1$ o menos. Para esto selecciona los puntos de evaluación x_i y los pesos w_i de forma conveniente.
$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$
 La regla suele expresarse para una integral en el intervalo $[-1, 1]$, y viene dada por la siguiente expresión:

En el caso particular en que $f(x)$ es un polinomio de

grado $2n-1$ o menos, la cuadratura de Gauss da el valor exacto de la integral. En el caso general, tal cuadratura dará buenas aproximaciones si $f(x)$ puede ser bien aproximada por un polinomio de grado $2n-1$ o menos, en el intervalo $[-1, 1]$.

Ecuaciones diferenciales

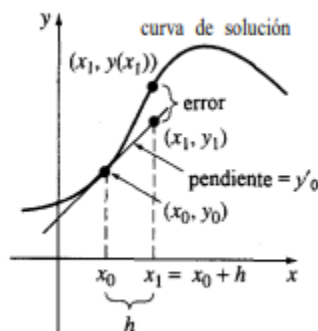
a) Métodos para resolver una ecuación diferencial, problema de condiciones iniciales. Euler izquierdo. Euler centrado. Euler derecho. Métodos de Runge/Kutta 3o orden. Métodos de Runge/Kutta 4o orden.

Método de Euler Una de las técnicas más sencillas para aproximar soluciones del problema de valor inicial

$$Y' = f(x, Y), \quad Y(x_0) = Y_0$$

se llama método de Euler o método de las tangentes. Aplica el hecho que la derivada de una

La función $y(x)$, evaluada en un punto x_0 , es la pendiente de la tangente a la gráfica de $y(x)$ en este punto. Como el problema de valor inicial establece el valor de la derivada de la solución en (x_0, y_0) , la pendiente de la tangente a la curva de solución en este punto es $f(x_0, y_0)$. Si recorremos una distancia corta por la línea tangente obtenemos una aproximación a un punto cercano de la curva de solución. A continuación se repite el proceso en el punto nuevo. Para formalizar este procedimiento se emplea la linealización $L(x) = Y'(x_0)(x - x_0) + Y_0$ de $y(x)$ en $x = x_0$. La gráfica de esta linealización es una recta tangente a la gráfica de $y = y(x)$ en el punto (x_0, Y_0) . Ahora se define h como un incremento positivo sobre el eje x . Reemplazamos x con $x_1 = x_0 + h$ en (1) y llegamos a



$$y_1 = y_0 + h y'_0 \quad \text{sea } Y_1 = y_0 + h y'_0,$$

en donde $y'_0 = y'(x_0) = f(x_0, y_0)$ y $y_1 = L(x_1)$. El punto (x_1, y_1) sobre la tangente es una aproximación al punto $(x_1, y(x_1))$ en la curva de solución; esto es, $L(x_1) = y(x_1)$, o $y_1 = y(x_1)$ es una aproximación lineal local de $y(x)$ en x_1 . La exactitud de la aproximación depende del tamaño h del incremento. Por lo general se escoge una magnitud de paso "razonablemente pequeña". Si a continuación repetimos el proceso, identificando al

nuevo punto de partida (x_1, y_1) como (x_0, y_0) de la descripción anterior, obtenemos la aproximación $y(x_2) = y(x_0 + 2h) = y(x_1 + h) = Y_2 = Y_1 + W_1, Y_2$

La consecuencia general es que en donde $x_n = x_0 + nh$.

$y_{n+1} = y_n + J(x_n, y_n)h$ (2)

Para ilustrar el método de Euler usaremos el esquema de iteración de la ecuación (2) en una ecuación diferencial cuya solución explícita es conocida; de esta manera podremos comparar los valores estimados (aproximados) y con los valores correctos (exactos) $y(x)$.

CÓDIGO

```
# Las funciones euler en ecuaciones diferenciales es mas facil de resolver
con vectores.

from pylab import * # Esta parte nos importa los componentes para graficar
el resultado.

def euler_c(x0, t_final, dt, f):
    lista_t = []
    lista_x = []

    a = x0
    t = 0.

    while t < t_final + dt: # Nos permite incluir la variable en el metodo
        lista_t.append(t)
        lista_x.append(a)

        a += dt * f(a, t)
        t += dt

    return lista_t, lista_x

# En esta parte se define como ira el incremento para sacar la solucion
# Ademas en esta parte se realiza la operación para Métodos de Runge/Kutta
de tercer orden
def rk3_completo(x0, t_final, h, f):
    lista_t = []
    lista_x = []
    a = x0
    t = 0.
```

```

while t < t_final + h:
    lista_t.append(t)
    lista_x.append(a)

    k1 = f(a, t)
    #Se realiza la operación utilizando una formula que nos permite
    calcular la respuesta y dar solución a la ecuacion
    k3 = f(a + 0.5 * h * k1, t + 0.5 * h)

    a += h * k3
    t += h

return lista_t, lista_x

# Se realiza la operación para Métodos de Runge/Kutta de orden
def rk4_completo(x0, t_final, h, f):
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h: # para incluir t_final
        lista_t.append(t)
        lista_x.append(x)

        k1 = f(x, t)
        k2 = f(x + 0.5 * h * k1, t + 0.5 * h)
        k3 = f(x + 0.5 * h * k2, t + 0.5 * h)
        k4 = f(x + h * k3, t + h)

        x += h / 6. * (k1 + 2. * k2 + 2. * k3 + k4)
        t += h

    return lista_t, lista_x

# las funciones euler, Métodos de Runge/Kutta de tercer y cuarto orden son
similares, se puede extraer una parte para encontrar la solucion

```

```

#METODO DE ECULER
def euler(x, t, h, f):
    return f(x, t)

# Métodos de Runge/Kutta de tercer ORDEN
def rk2(x, t, h, f):
    k1 = f(x, t)
    k2 = f(x + 0.5 * dt * k1, t + 0.5 * dt)

    return k2

# Métodos de Runge/Kutta de tercer ORDEN
def rk4(x, t, h, f):
    k1 = f(x, t)
    k2 = f(x + 0.5 * h * k1, t + 0.5 * h)
    k3 = f(x + 0.5 * h * k2, t + 0.5 * h)
    k4 = f(x + h * k3, t + h)

    return (k1 + 2. * k2 + 2. * k3 + k4) / 6.

def integrar_1er_orden(x0, t_final, h, f, metodo):
    # METODO PARA CALCULAR LA DERIVADA DE LA INTEGRAL DE PRIMER ORDEN
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h:
        lista_t.append(t)
        lista_x.append(x)

        derivada = metodo(x, t, h, f)

        x += h * derivada
        t += h

    return lista_t, lista_x

```

#HASTA ESTE PUNTO LA COMPUTADORA SABE QUE INSTRUCCIONES TIENE QUE EJECUTAR PARA PODER RESOLVER LA ECUACION POR LOS METODOS PERO AUN NO SE EJECUTA

```
def logistica(x, t):  
    return x * (5. - x)
```

```
dt = 0.2  
t_final = 5  
x0 = 0.1
```

```
t, x = euler_completo(x0, t_final, dt, logistica)
```

```
plot(t, x, 'bo-', label='euler')  
show()
```

```
t, x = rk2_completo(x0, t_final, dt, logistica)  
plot(t, x, 'go-', label='RK2')  
show()
```

```
t, x = rk4_completo(x0, t_final, dt, logistica)  
plot(t, x, 'ro-', label='RK4')  
show()
```

```
metodos = [euler, rk2, rk4]  
# una lista de los metodos que queremos utilizar
```

```
f = figure()  
# crear nueva figura
```

```
for metodo in metodos: # iterar por la lista  
    t, x = integrar_1er_orden(x0, t_final, dt, logistica, metodo)  
    # integrar con este metodo  
    plot(t, x, 'o-', label=metodo.__name__)  
  
show()
```

```

legend()

omega = 1.
omega_cuadrado = omega * omega
# Permite no hacer la multiplicacion cada vez

def harmonico(x_vec, t):

    xx, yy = x_vec
    # separar sus componentes

    return array([yy, -omega_cuadrado * xx])
    # regresamos un arreglo (vector) para poder utilizarlo en calculos

def integrar(x0, t_final, h, f, metodo):

    # metodo que calcula la derivada
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h:
        lista_t.append(t)
        lista_x.append(x.copy())
        # x.copy() hace una copia de x como una lista

        derivada = metodo(x, t, h, f)

        x += h * derivada
        t += h

```

```

    return lista_t, lista_x

# Crear una nueva figura:
f = figure(figsize=(8, 8))
# figsize cambia el tamaño / forma de la figura aquí es cuadrada

x0 = array([1., 0.])
t_final = 10.
dt = 0.1

metodos = [euler, rk2, rk4]

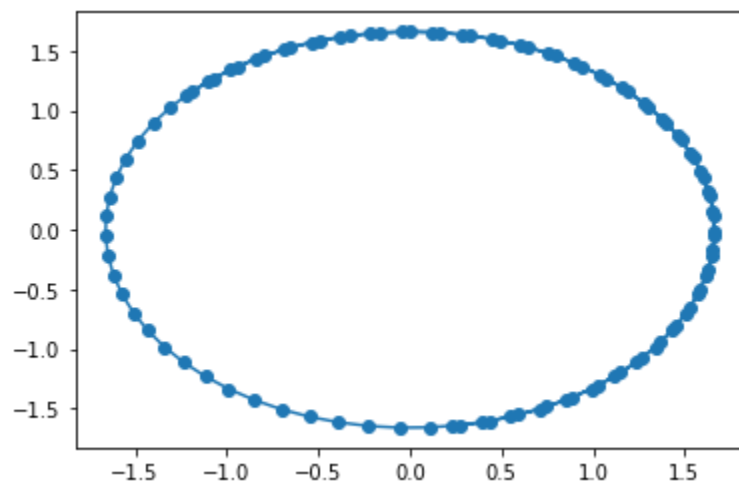
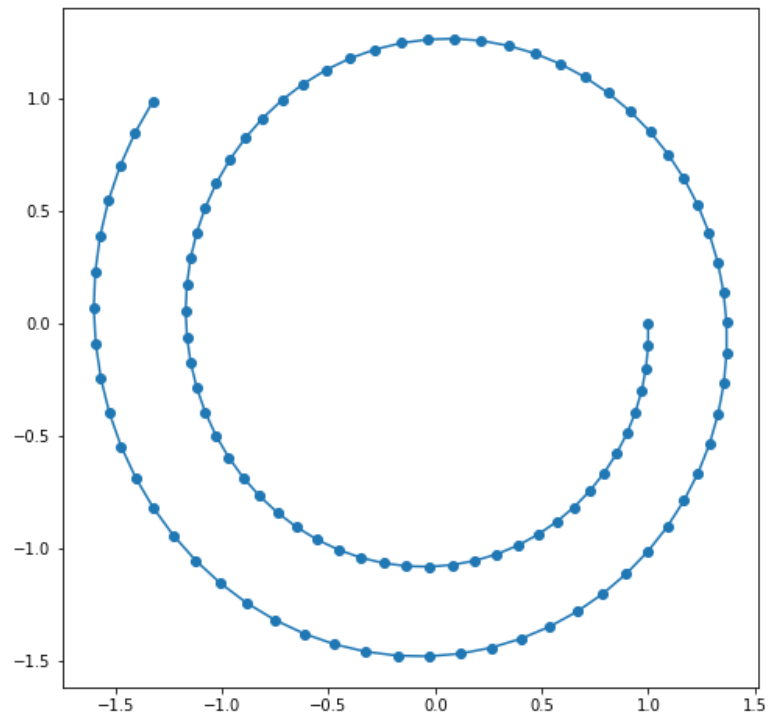
for metodo in metodos:
    t, x = integrar(x0, t_final, dt, harmonico, metodo)
    x = array(x)
    # convertir la lista de listas en un arreglo para poder extraer
componentes:

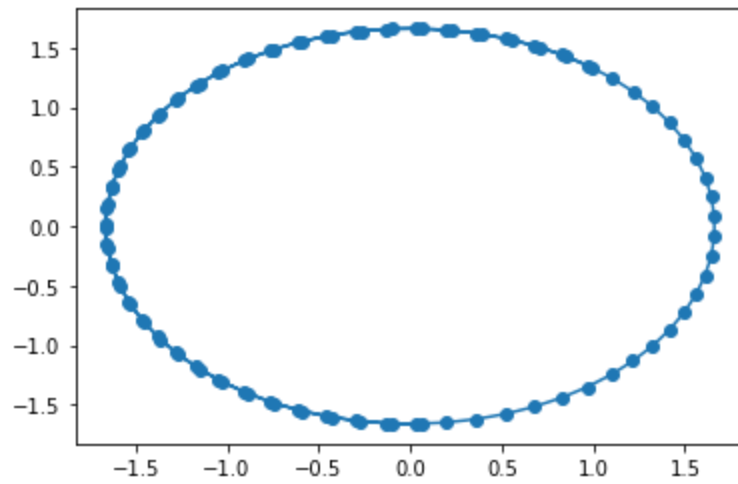
    plot(x[:, 0], x[:, 1], 'o-', label=metodo.__name__)
    show()

legend()

```

RESULTADO





b) Métodos Para Resolver Un Sistema De Ecuaciones, Problema De Condiciones Iniciales.

Un sistema de ecuaciones lineales es un conjunto de m ecuaciones lineales con n incógnitas, cuya solución es un conjunto de valores para las incógnitas con el que se satisfacen todas las ecuaciones. Se asume que los elementos que componen las ecuaciones están definidos en los números reales y que siempre hay la misma cantidad de ecuaciones que de incógnitas. En consecuencia, el problema consiste en buscar la única solución al sistema, si existe.

Un sistema de n -ecuaciones lineales (con coeficientes reales) en las n -incógnitas x_1, x_2, \dots, x_n , puede escribirse en la forma:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad y \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Y así mismo puede escribirse en la forma matricial equivalente $AX = b$ con:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad y \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

La matriz A es llamada matriz de coeficientes del sistema, el vector columna X el vector de incógnitas y el vector b el vector de términos independientes.

Los métodos numéricos que se trabajarán para la solución de un sistema de ecuaciones lineales se clasifican en dos tipos: Directos e iterativos.

A continuación, se enlistan los diferentes métodos a trabajar en esta página.

Métodos directos:

- Eliminación Gaussiana Simple.
- Eliminación Gaussiana con Pivoteo Parcial.
- Eliminación Gaussiana con Pivoteo Total.

Factorización directa de matrices

- Cholesky.
- Crout.
- Doolittle.

Métodos Iterativos:

- Jacobi.

- Gauss Seidel.
- Gauss Seidel con Relajación.

Método de Euler

Para resolver una ecuación diferencial con un método numérico, sólo se cuenta con la ecuación diferencial $f(x,y)$ que gráficamente es la pendiente de una recta tangente a la función en el punto inicial x_0 y sabemos el valor de y_0 para identificar la solución única. Con estos elementos, lo que podemos construir es una línea recta con la pendiente que nos da la ecuación diferencial y evaluar cuánto vale en el punto final, esto nos dará una aproximación al valor buscado.

De tal manera que el punto calculado en $y(x_0+h)$ se obtiene con:

$$y(x) = y_0 + f(x_0, y_0)(x - x_0)$$

$$y(x_0 + h) = y_0 + f(x_0, y_0)(x_0 + h - x_0)$$

$$y(x_0 + h) = y_0 + hf(x_0, y_0)$$

El cual nos da una aproximación al valor de $y(x_0+h)$ dado que no conocemos la ecuación de y .

La diferencia entre la aproximación de Euler con el valor correcto se da porque se toma la derivada en el punto inicial x_0 para hacer la aproximación; como se observa en la figura 8.2, el valor de la pendiente cambia en el intervalo x_0 y x_0+h , si tomamos cada cambio de la derivada se puede hacer una mejor aproximación.

Código en Python

```
# Las funciones euler etc. funcionan de manera *igual* para ecuaciones
# de orden superior si utilizamos vectores

from pylab import * # para graficas

def euler_completo(x0, t_final, dt, f):
    lista_t = []
    lista_x = []

    x = x0
    t = 0.
```

```

while t < t_final + dt: # para incluir t_final
    lista_t.append(t)
    lista_x.append(x)

    x += dt * f(x, t)
    t += dt

return lista_t, lista_x

def rk2_completo(x0, t_final, h, f): # aqui se decidio utilizar h en lugar de dt para el incremento
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h: # para incluir t_final
        lista_t.append(t)
        lista_x.append(x)

        k1 = f(x, t)
        k2 = f(x + 0.5 * h * k1, t + 0.5 * h)

        x += h * k2
        t += h

    return lista_t, lista_x

def rk4_completo(x0, t_final, h, f):
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h: # para incluir t_final
        lista_t.append(t)
        lista_x.append(x)

        k1 = f(x, t)
        k2 = f(x + 0.5 * h * k1, t + 0.5 * h)

```

```

        k3 = f(x + 0.5 * h * k2, t + 0.5 * h)
        k4 = f(x + h * k3, t + h)

        x += h / 6. * (k1 + 2. * k2 + 2. * k3 + k4)
        t += h

    return lista_t, lista_x

# Dado que las funciones euler, rk2 y rk4 son tan similares, podemos extra
er la parte comun, tal que
# hay una sola funcion 'integrar', y los metodos nada mas hacen un paso,
regresando el estimado
# de la derivada segun el metodo, como sigue:

def euler(x, t, h, f):
    return f(x, t)

def rk2(x, t, h, f):
    k1 = f(x, t)
    k2 = f(x + 0.5 * dt * k1, t + 0.5 * dt)

    return k2

def rk4(x, t, h, f):
    k1 = f(x, t)
    k2 = f(x + 0.5 * h * k1, t + 0.5 * h)
    k3 = f(x + 0.5 * h * k2, t + 0.5 * h)
    k4 = f(x + h * k3, t + h)

    return (k1 + 2. * k2 + 2. * k3 + k4) / 6.

def integrar_1er_orden(x0, t_final, h, f, metodo):
    # metodo es el metodo que utilizar, que estima la derivada
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h: # para incluir t_final
        lista_t.append(t)

```

```

        lista_x.append(x)

        derivada = metodo(x, t, h, f)

        x += h * derivada
        t += h

    return lista_t, lista_x

# Hasta ahora, nada mas hemos definido las funciones,
# es decir, le hemos dicho a la maquina como se implementa
# el metodo de Euler etc.

# Pero no le hemos dicho que ejecute (corra) ninguna funcion
# Para hacerlo, tenemos que llamar a la funcion como sigue

# Tampoco hemos definido ninguna funcion que se integrara:

def logistica(x, t):
    return x * (5. - x)

# Las variables que definimos aqui son independientes de las que
# hay adentro de las funciones euler etc.

dt = 0.2
t_final = 5
x0 = 0.1

t, x = euler_completo(x0, t_final, dt, logistica)
# utiliza 'logistica' como la f dentro de euler, y guarda el resultado en t y x
# esta es la primera linea en el programa que realmente hace algo (aparte de las definiciones de las variables)
plot(t, x, 'bo-', label='euler')
show()

t, x = rk2_completo(x0, t_final, dt, logistica)
plot(t, x, 'go-', label='RK2')
show()

t, x = rk4_completo(x0, t_final, dt, logistica)
plot(t, x, 'ro-', label='RK4')
show()

```

```

# Para no repetir tanto, podemos hacer algo mas listo:
metodos = [euler, rk2, rk4] # una lista de los metodos que queremos utilizar

f = figure() # crear nueva figura

for metodo in metodos: # iterar por la lista
    t, x = integrar_1er_orden(x0, t_final, dt, logistica, metodo) # integrar con este metodo
    plot(t, x, 'o-',
        label=metodo._name) # cualquier funcion f tiene f.name_, que es el nombre de la funcion
    show()

legend() # mostrar la leyenda del plot

# Pregunta 3:
# Las funciones definidas no tienen que cambiar!!
# Utilizamos vectores

# Ecuaciones
#  $x' = y$ 
#  $y' = -\omega^2 * x$ 

# En vectores:
#  $x' = f(x)$ 
#  $f(x) = (y, -\omega^2 * x)$ 

omega = 1.
omega_cuadrado = omega * omega # para no tener que hacer la multiplicacion cada vez

def harmonico(x_vec, t):
    # tratar x_vec como vector!
    xx, yy = x_vec # separar sus componentes
    # Alternativa: x, y = x_vec[0], x_vec[1]

    return array([yy, -
        omega_cuadrado * xx]) # regresamos un arreglo (vector) para poder utilizarlo en calculos

```

```

def integrar(x0, t_final, h, f, metodo): # funciona nada mas para arreglo
s
    # metodo es el metodo que utilizar, que estima la derivada
    lista_t = []
    lista_x = []

    x = x0
    t = 0.

    while t < t_final + h: # para incluir t_final
        lista_t.append(t)
        lista_x.append(x.copy())
        # x.copy() hace una copia de x como una lista
        # si no hacemos esto, entonces los elementos "cambian" cuando x ca
mbia!

        # una alternativa es lista_x.append(list(x)) -
- agrega una version de x como lista

        derivada = metodo(x, t, h, f)

        x += h * derivada
        t += h

    return lista_t, lista_x

# Crear una nueva figura:
f = figure(figsize=(8, 8)) # figsize cambia el tamaño / forma de la figur
a -- aquí es cuadrada

x0 = array([1., 0.])
t_final = 10.
dt = 0.1

metodos = [euler, rk2, rk4] # una lista de los metodos que queremos utili
zar

for metodo in metodos:
    t, x = integrar(x0, t_final, dt, harmonico, metodo)
    x = array(x) # convertir la lista de listas en un arreglo para poder
extraer componentes:

    plot(x[:, 0], x[:, 1], 'o-', label=metodo._name_)
    show()

```



```
legend() # mostrar la leyenda del plot
```

Método Runge-kutta de tercer orden

Este método utiliza la siguiente ecuación,

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)h$$

donde

$$k_1 = f(x_i, y_i),$$

$$k_2 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right),$$

$$k_3 = f(x_i + h, y_i - k_1h + 2k_2h).$$

Nuevamente, en términos generales, la expresión de este método tiene la forma del método de Euler, sin embargo el término $\frac{1}{6}(k_1 + 4k_2 + k_3)$ permite una corrección en la pendiente que se utiliza para evaluar el nuevo valor de y_{i+1} . Los valores de k_1, k_2, k_3 representan pendientes evaluadas en los puntos (x_i, y_i) , $(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h)$ y $(x_i + h, y_i - k_1h + 2k_2h)$, respectivamente. Para ejemplificar este método se resuelve, numéricamente y después analíticamente, la siguiente ecuación diferencial, $\frac{dy}{dx} = (1+x)\sqrt{y}$ con valores iniciales de $y(0) = 1$, desde $x = 0$ hasta $x = 1$ (límites de integración a, b).

Solución numérica

Utilizando la ecuación de Runge-Kutta de tercer orden, $y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)h$

. Teniendo los valores iniciales y los límites de integración se necesitan proponer un tamaño de paso, h , o en su defecto un número de segmentos, n , con el cual se calcula el tamaño de paso $h = \frac{b-a}{n}$. Enseguida se evalúan las pendientes $k_1, k_2,$

k_3 y por último se evalúa y_{i+1} .

Código en Python

```

import numpy as np
def runge_kutta_3(Dy,t0,tn,n,y0):
    # Paso 1
    h=(tn-t0)/n
    tiwi = np.zeros(shape=(n+1,2),dtype=float)
    tiwi[0] = [t0,y0]
    ti = t0
    wi = y0
    # Paso 2
    for i in range(1,n+1,1):
        # Paso 3
        K1 = h * Dy(ti,wi)
        K2 = h * Dy(ti+h/2, wi + K1/2)
        K3 = h * Dy(ti+h, wi + 2*K2 -K1)
        # Paso 4
        wi = wi + (1/6)*(K1+4*K2+K3)
        ti = ti + h
        tiwi[i] = [ti,wi]
    return(tiwi)

```

Método de Runge-Kutta orden cuatro (RK4)

El método de Runge-Kutta de orden cuatro (RK4) es uno de los métodos más utilizados en la práctica. Su deducción sigue las ideas de los métodos de Runge-Kutta orden dos y como los anteriores métodos, es un método recursivo y su cálculo no implica utilizar información adicional como ocurre en los métodos de Taylor. Su esquema se encuentra a continuación.

Método de Runge-Kutta de orden cuatro:

$$\begin{aligned}
 y_0 &= \alpha \\
 k_1 &= hf(t_i, y_i) \\
 k_2 &= hf\left(t_i + \frac{h}{2}, y_i + \frac{1}{2}k_1\right) \\
 k_3 &= hf\left(t_i + \frac{h}{2}, y_i + \frac{1}{2}k_2\right) \\
 k_4 &= hf(t_i + h, y_i + k_3) \\
 y_{i+1} &= y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
 \end{aligned}$$

Código en Python

```
#importamos las librerías q utilizaremos
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
import math

# definiendo la ED dy/dx en función(x,y) para valores iniciales
def dydx(x, y):
    return 4*math.e**(0.8*x)-0.5*y #sistema de ecu

# Solucion analitica de la ED en función(x,y)
def solucion (x,y):
    sol = (4/1.3)*(math.e**(0.8*x)-math.e**(-0.5*x))+2*math.e**(-0.5*x)
    return sol

#valores iniciales Y'(Xo)=Yo , mientras xf representa hasta que valor se q
uiere evaluar numericame
n =16
# Valor que se asume: a mayor valor de n mas iteraciones y mas preciso
x0 =0
# valor inicial de Y'(Xo)=Yo
y0 =2
# valor inicial de Y'(Xo)=Yo
xf =4
# para un valor de Xf se obtiene un yf o solucion numerica: Y'(Xf)=Yf
h=(xf-x0)/n
#crear array para la los valores de "y" mediante solucion analitica
ysa=np.zeros(n)
#creamos lista que almacenaran las soluciones numericas de "yi" , obtenias
a partir de "xi"
x = [x0]
y = [y0]
#iteramos por RK4 a patir de los valores iniciales x0,y0 la cantidad de n
veces, para obtener las
for i in range(n):
    k1=h*dydx(x0,y0)
    k2=h*dydx((x0+h/2),(y0+k1/2))
    k3=h*dydx((x0+h/2),(y0+k2/2))
    k4=h*dydx((x0+h),(y0+k3))
    y0=y0+(k1+2*k2+2*k3+k4)/6
    y0=y0
    x0=x0+h
    y.append(y0)
    x.append(x0)
#iteramos en la funcion de la solucion analitica para obtener los valores
de "y"=ysa, evaluados fo
```

```
for i in range(n):  
    ysa[i]=solucion(x[i+1], y[i])
```