

UNIVERSIDAD REY JUAN CARLOS

Métodos Numéricos - Práctica 1



Curso 2022-2023

Ingeniería Informática y Matemáticas

POR JUAN MONTES CANO, CRISTIAN PÉREZ CORRAL Y JUAN ANTONIO GORDILLO GAYO

# Índice

|   |           |
|---|-----------|
| <b>1 Interpolación numérica de Lagrange</b>   | <b>3</b>  |
| 1.1 Polinomio base de Lagrange  | 3         |
| 1.2 Polinomio interpolador de Lagrange  | 3         |
| 1.3 Ejemplo 1   | 4         |
| 1.4 Ejemplo 2   | 5         |
| <b>2 Integración numérica - Newton-Cotes</b>  | <b>8</b>  |
| 2.1 Fórmulas del Rectángulo   | 8         |
| 2.2 Fórmulas del Rectángulo   | 8         |
| 2.3 Fórmulas del Simpson 1/3  | 9         |
| 2.4 Ejemplo 1   | 9         |
| 2.5 Ejemplo 2   | 10        |
| <b>3 Raíces de funciones no lineales - Parte 1</b>                                    | <b>12</b> |
| 3.1 Descripción del algoritmo del método de la bisección.                             | 12        |
| 3.2 Implementación del algoritmo del método de la bisección.                          | 12        |
| 3.3 Resultados del algoritmo del método de la bisección.                              | 13        |
| 3.4 Descripción del algoritmo del método de Newton-Raphson.                           | 14        |
| 3.5 Implementación del algoritmo del método de Newton-Raphson.                        | 14        |
| 3.6 Resultados del algoritmo del método de Newton-Raphson.                            | 15        |
| 3.7 Aplicación de los métodos a la $f(x) = x \sin\left(\frac{x^2}{2}\right) + e^{-x}$ | 16        |
| <b>4 Raíces de funciones no lineales - Parte 2</b>                                    | <b>18</b> |
| 4.1 $f(x) = \sin(x) - 0.3e^x$   | 18        |
| 4.2 $f(x) = \sqrt{x} - \cos(x)$   | 19        |
| 4.3 $f(x) = 2x^3 - 11.7x^2 + 17.7x^5$   | 20        |
| 4.4 $f(x) = e^{x/2} + 5x - 5$   | 21        |
| 4.5 $f(x) = x^2 + x + 3$  | 22        |
| <b>5 Raíces de funciones no lineales - Parte 3</b>                                    | <b>24</b> |
| 5.1 Presentación de la convergencia del método Newton-Raphson                         | 24        |
| 5.2 Convergencia para $f(x) = x^3 - 3x + 2$ y $x_0 = -3$                              | 24        |
| <b>6 Resolución numérica de sistemas lineales</b>                                     | <b>26</b> |
| 6.1 Descripción e implementación de los algoritmos                                    | 26        |
| Gauss-Seidel  | 26        |
| Jacobi  | 27        |
| 6.2 Problemas planteados y su resolución  | 28        |

# 1 Interpolación numérica de Lagrange

## 1.1 Polinomio base de Lagrange

Vamos a ver el desarrollo que hemos hecho para la expresión de los polinomios base de Lagrange y del polinomio interpolador de Lagrange. Recordamos que los polinomios base de Lagrange, siendo  $\{x_i\}_i$  la secuencia de puntos que nos dan para poder calcularlos, vienen dados por

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

se construyen en función de los puntos dados. Ahora, viendo el script en Python para la construcción de dichos polinomios

```
def polinomiosLagrange (list):
    result = []

    for i in range(len(list)):
        l_i = 1
        for j in range(len(list) ):
            if j == i :
                continue

            l_i = l_i*(x-list[j])/(list[i]-list[j])
        result.append(l_i)
    return result
```

definimos el método *polinomiosLagrange* que recibe como parámetro una lista (la secuencia de puntos a usar, i.e,  $\{x_i\}_i$ ) y devuelve una lista, que comprendo los  $L_i$  polinomios base, construidos de forma iterativa. Iteramos sobre la lista de puntos (en el primer *for*) y, vamos creando el polinomio  $i$ -ésimo con un segundo *for* que utilizamos para, iterar de nuevo sobre dicha lista para hacer el producto definido arriba. Con todo esto, almacenamos en la variable *result* todos los polinomios base.

## 1.2 Polinomio interpolador de Lagrange

Ahora, vamos a ver la definición usada para construir el polinomio interpolador de Lagrange. Este polinomio, por lo visto en teoría, se construye utilizando los polinomios base y la función a aproximar. Veamos la expresión de dicho polinomio:

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

siendo  $L_i$  los polinomios base,  $f$  la función a aproximar,  $x_i \in \{x_i\}$  el punto  $i$ -ésimo que usamos.

Teniendo esta definición, vemos el siguiente método de Python para el cálculo:

```
def polinomiosInterpoladores(list_x,f):
    p_Lagrange = polinomiosLagrange(list_x)
    p_inter_n =0
    for i in range(len(list_x)):
        p_inter_n += f.subs({'x':list_x[i]})*p_Lagrange[i]
    return p_inter_n
```

*polinomiosInterpoladores* recibe como parametro *list\_x* y *f*, siendo *list\_x* la secuencia de puntos  $\{x_i\}_i$  y *f* la función a aproximar. Dentro del método, llamamos a *polinomiosLagrange*, para obtener los polinomios base asociados a dichos puntos. Luego, iteramos sobre *list\_x* para hacer el sumatorio anterior, donde la *f.subs* es un método de la librería de cálculo simbólico de Python para sustituir cada valor de la lista *list\_x* dentro de *f* (y obtener así  $f(x_i)$ ) y multiplicarlo por el polinomio base *i*-ésimo. Devolvemos después el polinomio interpolador, como la suma del producto mencionado.

### 1.3 Ejemplo 1

Si aplicamos los programas al caso resuelto en clase, vamos a obtener los siguientes resultados, donde el código sería

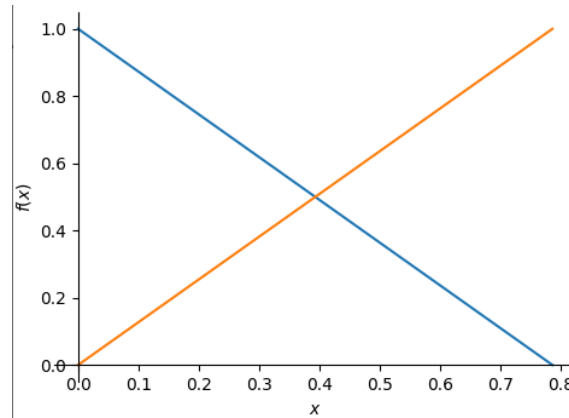


Figura 1.

Esta es la figura correspondiente a los polinomios base del primer ejemplo, de donde obtenemos

$$L_0 = -4 \frac{x - \pi/4}{\pi} \simeq 1 - 1.27x \quad L_1 = 4 \frac{x}{\pi} \simeq 1.27x$$

con dos puntos,  $x_0 = 0$  y  $x_1 = \pi/4$ , de donde obtenemos la siguiente aproximación

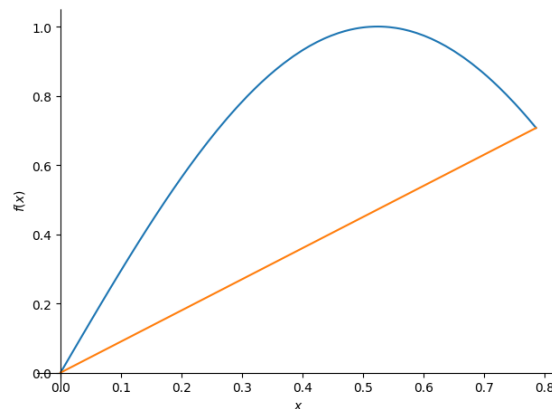
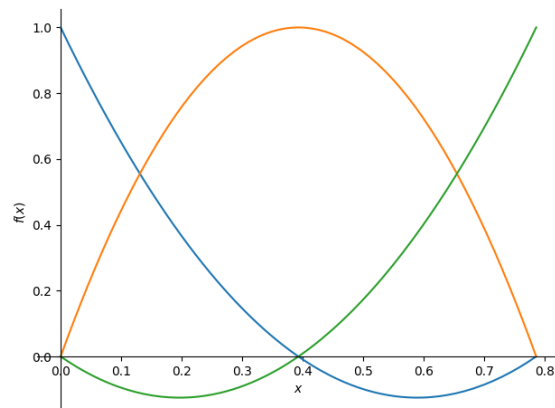


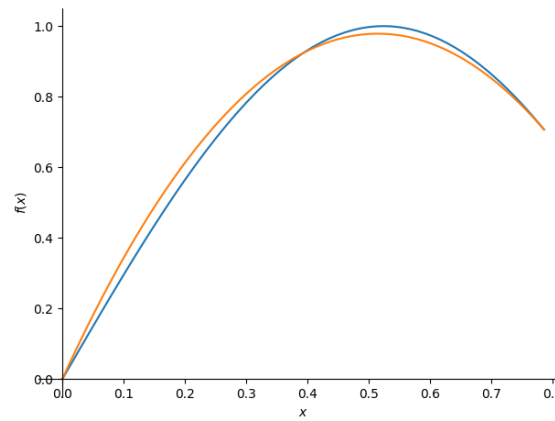
Figura 2.

y, si ahora seguimos el ejercicio y lo hacemos con 3 puntos, los polinomios base nos quedarán



**Figura 3.**

y la aproximación quedará tal que



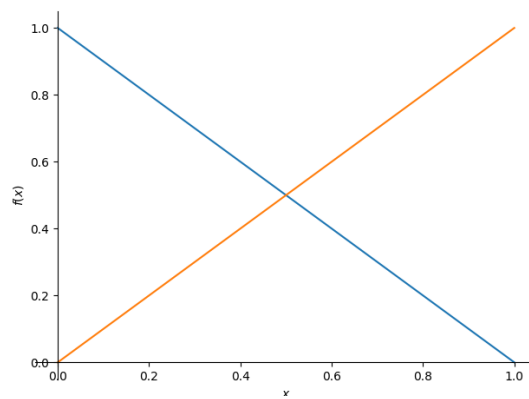
**Figura 4.**

es decir, obtenemos los mismos resultados que el ejercicio.

## 1.4 Ejemplo 2

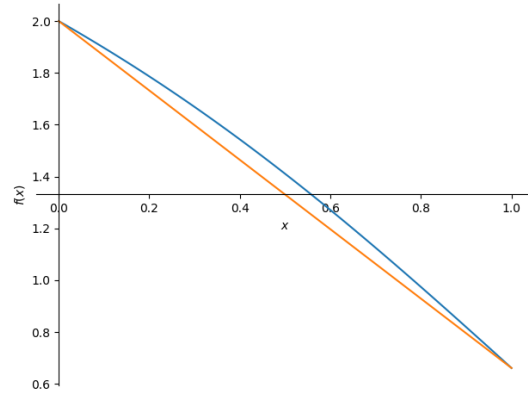
Ahora, si definimos la función  $f(x) = e^{-x} + \cos\left(\frac{4x}{\pi}\right)$ , tomando el intervalo  $[0, 2]$ , vamos a diferenciar los casos de

1. 2 puntos



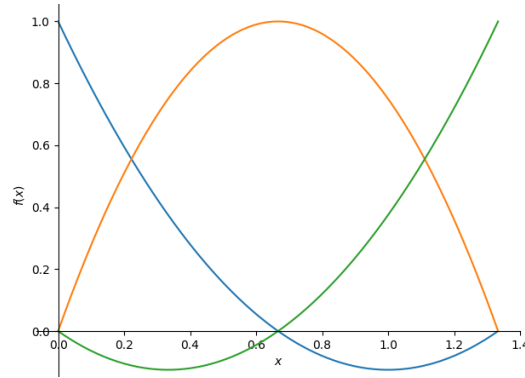
**Figura 5.**

con  $L_0 = 1 - x$  y  $L_1 = x$ , de donde obtenemos la siguiente aproximación



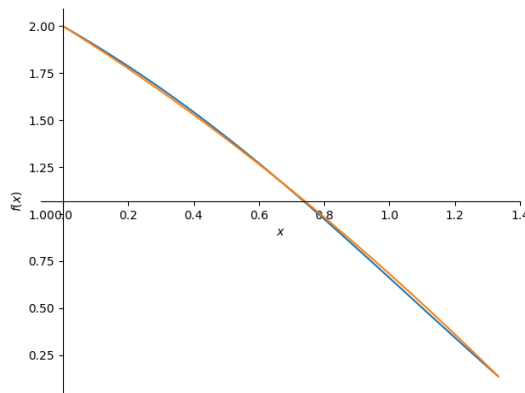
**Figura 6.**

2. 3 puntos



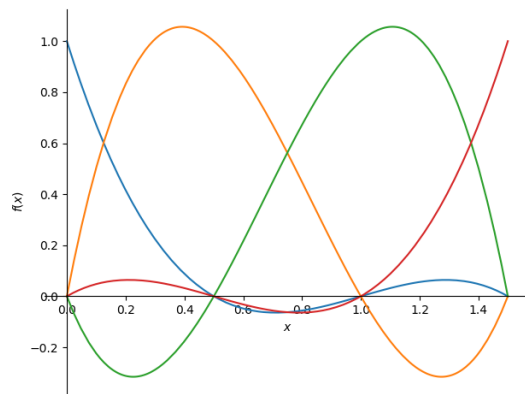
**Figura 7.**

con  $L_0 = -0.75(1.0 - 1.5x)(x - 1.3333333333333333)$ ,  $L_1 = -2.25x(x - 1.3333333333333333)$ ,  
y  $L_2 = 1.125x(x - 0.6666666666666667)$ , de donde obtenemos la siguiente aproximación



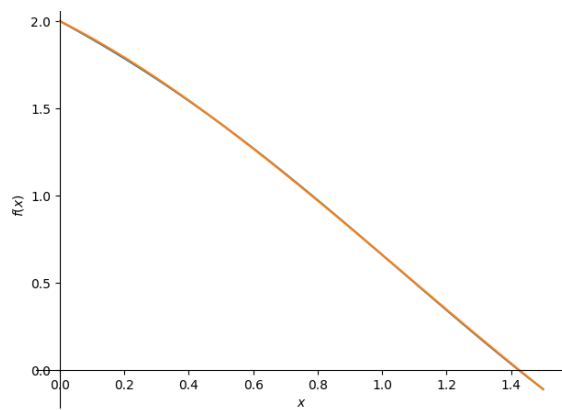
**Figura 8.**

3. 4 puntos



**Figura 9.**

con  $L_0 = 0.666666666666667 (1.0 - 2.0 x) (x - 1.5) (x - 1.0)$ ,  $L_1 = 4.0 x (x - 1.5) (x - 1.0)$ ,  $L_3 = -4.0 x (x - 1.5) (x - 0.5)$  y  $L_4 = 1.33333333333333 x (x - 1.0) (x - 0.5)$ , de donde la aproximación es



**Figura 10.**

## 2 Integración numérica - Newton-Cotes

### 2.1 Fórmulas del Rectángulo

Sabemos, por lo visto en clase, que a partir de la interpolación vista en el apartado anterior podemos sacar todas las fórmulas con las que aproximamos la integral. Así, las fórmulas del rectángulo son simplemente las fórmulas cuando tenemos 1 punto, y vienen dadas de la siguiente manera

$$p_n(x) = f(a) \rightarrow I \simeq \int_a^b p_n(x) dx = f(a) (b - a) \quad (\text{extremo izquierdo})$$

$$p_n(x) = f(b) \rightarrow I \simeq \int_a^b p_n(x) dx = f(b) (b - a) \quad (\text{extremo derecho})$$

$$p_n(x) = f\left(\frac{b+a}{2}\right) \rightarrow I \simeq \int_a^b p_n(x) dx = f\left(\frac{b+a}{2}\right) (b - a) \quad (\text{punto central})$$

de donde, para calcular dichas fórmulas, podemos usar, en Python, los siguientes métodos:

```
def fRectanguloExtIzdo(f,a,b):
    i = N(integrate(f,(x,a,b)))
    sol = N(f.subs(x,a)*(b-a))
    return sol, abs(sol-i)/i

def fRectanguloExtDcho(f,a,b):
    i = N(integrate(f,(x,a,b)))
    sol = N(f.subs(x,b)*(b-a))
    return [sol, abs(sol - i) / i]

def fRectanguloMedio(f,a,b):
    i = N(integrate(f,(x,a,b)))
    sol = N(f.subs(x,(a+b)/2)*(b-a) * (b - a))
    return [sol, abs(sol - i) / i]
```

En los tres métodos tenemos los mismos parámetros, la función  $f$ , y los extremos del intervalo,  $[a, b]$ .

Respectivamente, hacemos lo siguiente. Calculamos la integral con  $\text{integrate}(f, (x, a, b))$  (la  $x$  indica la variable que estamos sustituyendo) con el método definido para eso por la librería de cálculo simbólico de Python, y luego calculamos con  $f(a) (b - a)$ ,  $f(b) (b - a)$ ,  $f\left(\frac{b+a}{2}\right) (b - a)$ , respectivamente) con  $f.\text{subs}(x,a)*(b-a)$ , que es la solución de nuestra aproximación. Entonces, devolvemos un *array* de dos elementos, en primer lugar la solución que damos con la fórmula del rectángulo, y en segundo el error que cometemos utilizandola, comparado con la integral real.

### 2.2 Fórmulas del Rectángulo

La fórmula del trapecio es la fórmula de Newton-Cotes cuando usamos dos puntos, y, como hemos visto en teoría, viene dada por

$$p_n(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \rightarrow I \simeq \int_a^b p_n(x) dx = (b - a) \frac{f(a) + f(b)}{2}$$

y nosotros la hemos implementado con el siguiente método



```
def fTrapezio(f,a,b):
    i = N(numpy.trapz([a, b]))
    sol = N((b-a)*(f.subs(x,a)+f.subs(x,b))/2)
    return sol, abs(sol - i) / i
```

donde la función *numpy.trapz* es la función que calcula la integral con la fórmula del trapecio dada por Python (para comparar con la nuestra y estudiar el error, como el comando *trapz* en matlab) y calculamos nuestra solución simplemente aplicando la fórmula. Devolvemos, también, dos soluciones, la solución (calculada con la fórmula) y el error relativo.

## 2.3 Fórmulas del Simpson 1/3

Ahora, vamos a utilizar la fórmula de Simpson 1/3. Es la aplicación de lo obtenido con Newton-Cotes, a 3 puntos. Sabemos de teoría que tiene la integral la siguiente expresión:

$$I \simeq \frac{b-a}{6} (f(x_0) + 4f(x_1) + f(x_2))$$

```
def fSimpson(f,xi):
    a = N(integrate(f, (x, xi[0], xi[2])))
    D = 6
    c = [1,4,1]
    ret = 0
    for i in range(3):
        ret += c[i]*f.subs(x,xi[i])
    sol = N(ret*(xi[2]-xi[0])/D)
    return sol, abs(sol-a) / a
```

donde primero calculamos la integral dada por Python (*integrate...*) y después, tomando D y los coeficientes visto en clase, calculamos el resultado como la suma (en un *for*, para hacer el sumatorio), y devolvemos la solución (la operación implementada) y el error que nos da respecto a la integral calculada.

## 2.4 Ejemplo 1

El problema visto en clase es  $\int_0^2 x^3 - 2x^2 + 1 \, dx$ . Si utilizamos los tres métodos vistos anteriormente, e implementados en Python, la respuesta que obtenemos por pantalla para cada uno es:

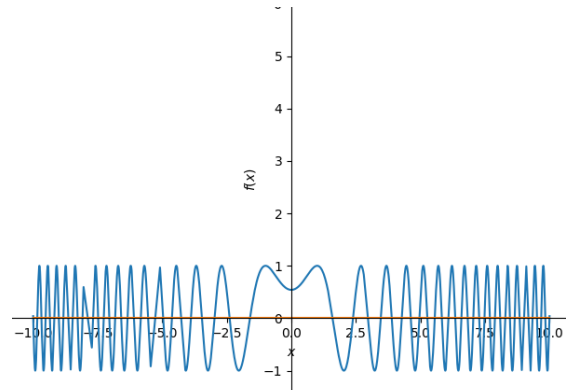
- Fórmula rectángulo extremo derecho:  
El resultado de este es  $\int_0^2 p_n(x) = 2$
- Fórmula rectángulo extremo izquierdo:  
El resultado de este es  $\int_0^2 p_n(x) = 2$
- Fórmula rectángulo punto medio:  
El resultado de este es  $\int_0^2 p_n(x) = 0$
- Fórmula trapecio:  
El resultado de este es  $\int_0^2 p_n(x) = 2$
- Fórmula Simpson 1/3:  
El resultado de este es  $\int_0^2 p_n(x) = 0.666666667$

Y, el resultado de la integral del enunciado es 0.66666666667, luego coincide únicamente con la fórmula de Simpson 1/3, y las otras se alejan bastante del resultado deseado.

## 2.5 Ejemplo 2

Vamos a aplicar los programas desarrollados a la integral  $\int_0^{2\pi} \cos(x^2 - 1) dx$ .

La función que hemos definido tiene esta forma



**Figura 11.**

Y, con las fórmulas definidas anteriormente, obtenemos:

- Fórmula rectángulo extremo derecho:  
El resultado de este es  $\int_0^{2\pi} p_n(x) = 4.46986814633572$ , y el error calculado da  $\approx 3.8528$
- Fórmula rectángulo extremo izquierdo:  
El resultado de este es  $\int_0^{2\pi} p_n(x) = 3.39481950966595$ , y el error calculado da  $\approx 2.6856$
- Fórmula rectángulo punto medio:  
El resultado de este es  $\int_0^{2\pi} p_n(x) = -5.33950724275201$ , y el error calculado da  $\approx 6.7969$
- Fórmula trapecio:  
El resultado de este es  $\int_0^{2\pi} p_n(x) = 3.93234382800083$ , y el error calculado da  $\approx 0.2517$
- Fórmula Simpson 1/3:  
El resultado de este es  $\int_0^{2\pi} p_n(x) = -2.24889021916773$ , y el error calculado da  $\approx 3.4415$

Pregunta Optativa) La fórmula de Milne viene dada cuando tenemos 5 puntos en Newton-Cotes, y la aproximación de la integral tiene una expresión así

$$I = (b - a) \frac{7 f(a) + 32 f(x_1) + 12 f(x_2) + 32 f(x_3) + 7 f(b)}{90}$$

Nosotros, en Python, hemos implementado este método para aplicar la fórmula descrita arriba

```

def fMilne(f, xi):
    a = N(integrate(f, (x, xi[0], xi[4])))
    D = 90
    c = [7, 32, 12, 32, 7]
    ret = 0
    for i in range(5):
        ret += c[i] * f.subs(x, xi[i])
    sol = N(ret * (xi[4] - xi[0]) / D)
    return sol, abs(sol - a) / a

```

que tiene un funcionamiento análogo al método usado para la fórmula de Simpson.

Entendemos de este enunciado que, lo que tenemos que hacer, es aplicar Milne a subintervalos y luego sumar las áreas que nos dan estos. Nosotros lo hemos hecho hasta dividiendo  $N=6$  veces, es decir, hasta  $2^6 = 64$  subintervalos, y viendo la evolución del error en estos, tratando de aproximar el valor de la integral

$$\int_1^3 (\cos(x) - x \sin(x)) dx$$

El resultado que hemos obtenido es el siguiente

Iteración 1: solución -3.50997966244149, error -0.000300133227982169

Iteración 2: solución -3.51027565125068, error -0.00000414441879614813

Iteración 3: solución -3.51027973279580, error -6.28736724905821E-8

Iteración 4: solución -3.51027979469422, error -9.75259872859624E-10

Iteración 5: solución -3.51027979565426, error -1.52113877049942E-11

Iteración 6: solución -3.51027979566924, error -2.38031816479634E-13

Donde en cada iteración tenemos  $2^{\{\text{iteración}\}}$  subintervalos.

### 3 Raíces de funciones no lineales - Parte 1

#### 3.1 Descripción del algoritmo del método de la bisección.

El método de la bisección es convergente, para su funcionamiento debemos proporcionar una función continua  $f$  y un intervalo  $[x_1, x_2]$  de manera que el signo de  $f(x_1)$  y  $f(x_2)$  sea opuesto, así podremos asegurar que existe un punto  $x_0$  tal que  $f(x_0) = 0$  (Teorema de Bolzano).

En ocasiones también podremos añadir una cota de error al algoritmo, que determinará el criterio de parada.

En primer lugar consideramos el punto intermedio:

$$x_m = \frac{(x_1 + x_2)}{2}$$

Ahora dependiendo del valor del  $f(x_m)$  continuamos de la siguiente forma:

- Si  $f(x_m) = 0$  entonces  $x_m$  es la raíz que buscamos. Esto será nuestro criterio de parada.
- Si  $f(x_m)$  tiene el mismo signo que  $f(x_1)$  entonces el valor de la raíz se encuentra en el intervalo  $(x_m, x_2)$  puesto que así  $f(x_m)$  y  $f(x_2)$  tienen valores de distinto signo.

Ahora aplicaríamos el algoritmo de nuevo a este intervalo hasta encontrar el error.

- Si  $f(x_m)$  tiene el mismo signo que  $f(x_2)$  entonces el valor de la raíz se encuentra en el intervalo  $(x_1, x_m)$  puesto que así  $f(x_1)$  y  $f(x_m)$  tienen valores de distinto signo.

Ahora aplicaríamos el algoritmo de nuevo a este intervalo hasta encontrar el error.

Este sería el algoritmo cuando queremos obtener raíces numéricas exactas, es decir, con cota de error 0.

Si usamos el algoritmo con una cota de error distinta, el algoritmo sería el siguiente:

En primer lugar, definimos la cota de error como  $\varepsilon = x_2 - x_1$ , es decir, la cota de error es la distancia entre los dos valores que definen el intervalo.

Al igual que antes usamos una función continua  $f$  en un intervalo  $[x_1, x_2]$  con una cota de error fija  $\varepsilon$ .

Así pues, nuestro algoritmo sería el siguiente:

- Si  $x_2 - x_1 < \varepsilon$  entonces  $x_m$  es la raíz que buscamos. Esto será nuestro criterio de parada.
- Si  $f(x_m)$  tiene el mismo signo que  $f(x_1)$  entonces el valor de la raíz se encuentra en el intervalo  $(x_m, x_2)$  puesto que así  $f(x_m)$  y  $f(x_2)$  tienen valores de distinto signo.

Ahora aplicaríamos el algoritmo de nuevo a este intervalo hasta encontrar el error.

- Si  $f(x_m)$  tiene el mismo signo que  $f(x_2)$  entonces el valor de la raíz se encuentra en el intervalo  $(x_1, x_m)$  puesto que así  $f(x_1)$  y  $f(x_m)$  tienen valores de distinto signo.

Ahora aplicaríamos el algoritmo de nuevo a este intervalo hasta encontrar el error.

#### 3.2 Implementación del algoritmo del método de la bisección.

```
# 1.- MÉTODO BISECCIÓN
def metodo_Biseccion(f, a, b, epsilon, ite=0): # f es la funcion, a es el
punto a la izda y b el punto a la derecha (importante f(a) y f(b) deben ser de
signo opuesto)
    x_m = (a + b) / 2
    if ((b - a) < epsilon):
        return x_m, ite
```

```

if (f.subs(x, x_m) == 0):
    return x_m, ite
elif (f.subs(x, x_m) * f.subs(x, a) > 0):
    return metodo_Biseccion(f, x_m, b, epsilon, ite + 1)
else:
    return metodo_Biseccion(f, a, x_m, epsilon, ite + 1)

```

Vemos que en nuestra función la implementamos con un esquema recursivo, de acuerdo con el algoritmo presentado. También devolvemos en que iteración la función converge y el valor.

Para recibir la raíz exacta de la función debemos usar el valor  $\epsilon=0$ .

### 3.3 Resultados del algoritmo del método de la bisección.

Para comprobar resultados vamos a usar dos funciones:

- i.  $g(x) = x + \cos(x)$
- ii.  $h(x) = x - x^2$

Veamos las raíces de  $g$ :

Primero buscamos un intervalo que cumpla los requisitos del algoritmo:

Puesto que  $g(-2) < 0$  y  $g(10) > 10$  el intervalo  $[-2, 10]$  es válido. Tomamos como cota de error  $\epsilon = 0.0001$ .

Al aplicar la función con estos parámetros

```

g = x + sympy.cos(x)
resultado = metodo_Biseccion(g, -2, 10, 0.0001)
#RESULTADO
(-0.7390899658203125, 17) #RAÍZ, ITERACIONES

```

Nuestra raíz tiene valor -0.7390899658203125

Efectivamente, al calcular el valor de  $g$  en ese punto obtenemos -8.08791474471438e-6, que concuerda con lo esperado.

Veamos la gráfica de la función y el resultado del algoritmo:

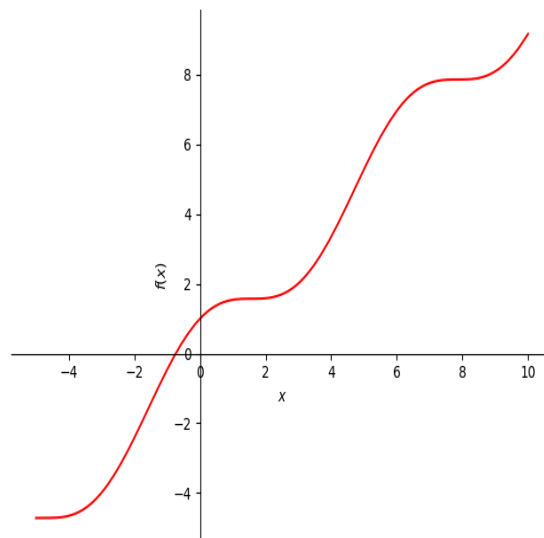


Figura 12.

Ahora analicemos la otra función  $h$ . Siguiendo el proceso anterior evaluamos en el intervalo  $[2, 0.5]$  y tomaremos como  $\varepsilon = 0.01$ . Veamos los resultados:

```
h = x - x**2
resultado = metodo_Biseccion(h, 0.4, 2, 0.01)
#RESULTADO
(0.996875, 8) #RAIZ, ITERACIONES
```

Vemos que mientras mayor es la cota de error menor es el número de iteraciones.

Los resultados se corresponden con lo obtenido en la gráfica de  $h$ , nótese que la función presenta dos raíces en 0 y 1, como el intervalo cogido contiene a 1 encontró esa raíz, sin embargo, si hubiesemos tomados el intervalo  $[-5, 0.5]$  el método nos habría devuelto la raíz 0.

La gráfica es la siguiente:

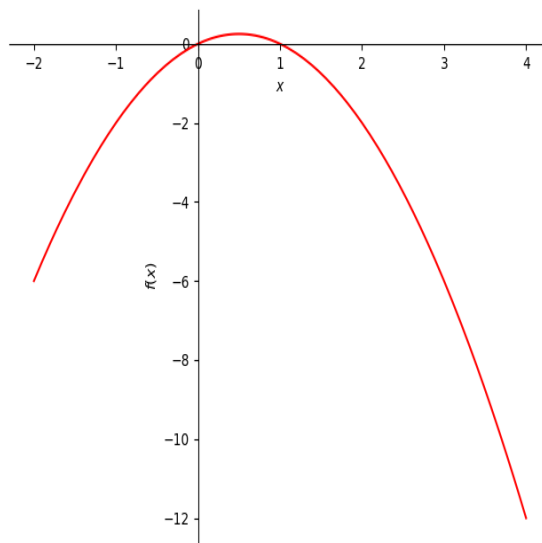


Figura 13.

### 3.4 Descripción del algoritmo del método de Newton-Raphson.

Es un método que tiene convergencia cuadrática en caso de existir. El algoritmo parte de un punto inicial  $(x_0, f(x_0))$  y se busca una estimación del siguiente punto  $x_1$  siguiendo la tangente de la función en el punto anterior.

Al igual que el método anterior es recursivo con un criterio de parada determinado.

Definimos como  $x_n$  el punto devuelto por el algoritmo en la iteración  $n$ .

El algoritmo recibe una función de clase  $C^1$   $f$ , un valor de inicio  $x_0$  y una cota de error  $\varepsilon$ .

El algoritmo sigue el siguiente esquema:

- Si  $|x_{n+1} - x_n| < \varepsilon$  entonces devolvemos el valor  $x_{n+1}$ , es nuestro criterio de parada.
- En caso contrario, calculamos:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

### 3.5 Implementación del algoritmo del método de Newton-Raphson.

```
# 1.- MÉTODO NEWTON-RAPHSON
```

```
def metodo_NewtonRaphson(f, x_m, epsilon, ite=0, limit = -1):

    #Derivamos la funcion
    f_x = sympy.diff(f, x)
    x_Nuevo = x_m - (f.subs(x, x_m) / f_x.subs(x, x_m)) #CALCULO DEL NUEVO
    #SE CUMPLE CRITERIO DE PARADA
    if ((abs(x_m - x_Nuevo) < epsilon) or limit == 0):
        return x_Nuevo, ite
    #EN EL CASO DE QUE NO TENGAMOS ITERACIONES LIMITADAS
    if(limit < 0):
        return metodo_NewtonRaphson(f, sympy.N(x_Nuevo), epsilon, ite + 1)
    else:#SI LAS TENEMOS POR CADA ITERACIONES RESTAMOS UNA
        return metodo_NewtonRaphson(f, sympy.N(x_Nuevo), epsilon, ite + 1, limit
-1)
```

Vemos como nuestros valores de entrada son la función  $f$ , el valor de inicio  $x_m$ , la cota de error  $\varepsilon$  y luego tenemos dos valores que nos permiten controlar tanto las iteraciones como el límite de las mismas. Estos últimos valores no son necesario introducirlos, pero nos permiten hacer la función mucho más manejable y adaptable a los ejercicios.

Vemos que sigue un esquema recursivo con un caso base determinado por  $|x_n - x_{n+1}| < \varepsilon$ . Es decir, nuestra cota de error viene dada por la distancia entre los dos puntos de las últimas iteraciones.

### 3.6 Resultados del algoritmo del método de Newton-Raphson.

Para evaluar el funcionamiento de nuestro algoritmo, vamos a usar las funciones descritas anteriormente y así comparar los resultados con el método de la bisección.

Las funciones que usaremos son:

- $g(x) = x + \cos(x)$  con  $\varepsilon = 0.0001$  y  $x_0 = -2$
- $h(x) = x + x^2$  con  $\varepsilon = 0.01$  y  $x_0 = 0.4$

Comenzamos con  $g$ :

```
#NEWTON-RAPHSON
resultado = metodo_NewtonRaphson(g, -2, 0.0001)
print(resultado)
(-0.739085133219815, 2) #RESULTADO,ITERACIONES
```

Vemos que obtenemos prácticamente el mismo resultado que usando el método de la bisección pero con 15 iteraciones menos.

Veamos que sucede en el caso de  $h$ :

```
#NEWTON-RAPHSON
resultado = metodo_NewtonRaphson(h, 0.4, 0.01)
print(resultado)
(-2.31782539490059e-6, 4) #RESULTADO,ITERACIONES
```

Aquí obtenemos un resultado distinto al del método de la bisección, obtenemos en este caso la raíz correspondiente al número 0, en el método de la bisección encontramos el valor 1.

Que los valores sean distintos es completamente lógico, puesto que en nuestro punto de partida  $x_0 = 0.4$  como vemos en la gráfica la pendiente de la tangente en  $f(x_0)$  es positiva y  $f(x_0) > 0$  entonces, por como está determinado el algoritmo,  $x_1 < x_0$ , por lo que el algoritmo encontrará la raíz a la izquierda de 0.4.

### 3.7 Aplicación de los métodos a la $f(x) = x \sin\left(\frac{x^2}{2}\right) + e^{-x}$

Primero vamos a ver gráficamente como se comporta esta función:

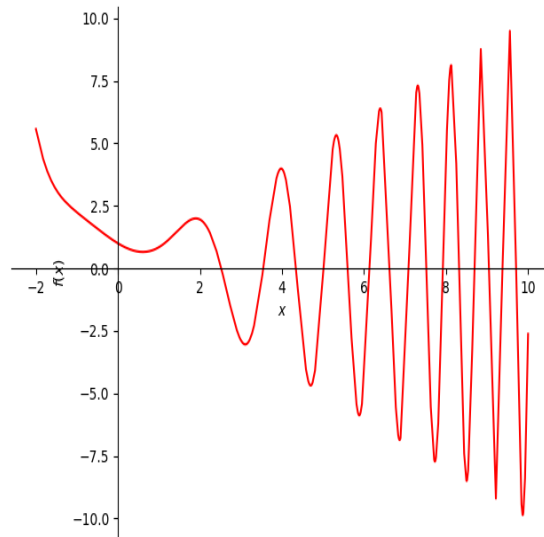


Figura 14.

Observamos que la función oscila, probemos primero usando el método de la bisección:

Vamos a tomar el conjunto  $\{0, 3, 4, 5\}$  y vamos a usarlos como los puntos que forman los intervalos para el método de la bisección.

Así pues, obtenemos los siguientes intervalos,  $[0, 3], [3, 4], [4, 5]$ . Tomemos  $\varepsilon = 0.01$ .

Entonces obtenemos un código como el siguiente:

```
f = x * sympy.sin(0.5 * x ** 2) + sympy.exp(-x)
resultado = metodo_Biseccion(f, 0, 3, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(resultado[0]), resultado[1]))
sol: 2.51660156250000 (n: 9)
resultado = metodo_Biseccion(f, 3, 4, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(resultado[0]), resultado[1]))
sol: 3.54296875000000 (n: 7)
resultado = metodo_Biseccion(f, 4, 5, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(resultado[0]), resultado[1]))
sol: 4.33984375000000 (n: 7)
```

Podemos ver que dependiendo del intervalo que escojamos obtenemos una raíz distinta.

Comparemos este comportamiento con los mismos puntos usando el método de Newton-Raphson.

```
f = x * sympy.sin(0.5 * x ** 2) + sympy.exp(-x)
resultado = metodo_NewtonRaphson(f, 0, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(resultado[0]), resultado[1]))
sol: 2.51935360957807 (n: 7)
resultado = metodo_NewtonRaphson(f, 3, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(resultado[0]), resultado[1]))
sol: 5.01299206538645 (n: 57)
resultado = metodo_NewtonRaphson(f, 4, 0.01)
```



```

print("sol:{_}(n:{_})".format(sympy.N(resultado[0]),resultado[1]))
sol:7.08979882780822 (n:5)
resultado = metodo_NewtonRaphson(f,5,0.01)
print("sol:{_}(n:{_})".format(sympy.N(resultado[0]),resultado[1]))
sol:5.01299189583714 (n:1)

```

Vemos que el valor de la raíz encontrada depende del punto y su tangente, al igual que en el caso de  $x + x^2$ . Prestemos atención el caso de  $x_0 = 3$ . Vemos que realiza 57 iteraciones, una diferencia muy grande con el resto de casos. Esto se debe a que oscila entorno a un punto.

## 4 Raíces de funciones no lineales - Parte 2

### 4.1 $f(x) = \sin(x) - 0.3e^x$

Primero mostremos gráficamente como es la función:

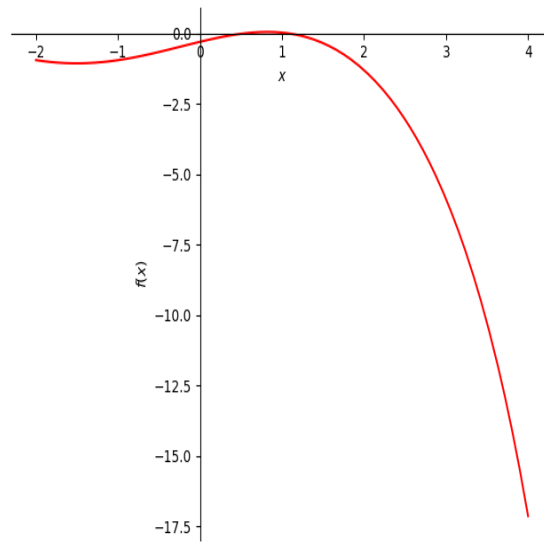


Figura 15.

Ahora aplicamos los métodos:

```
# 2.1 Bisección
aux = metodo_Biseccion(f, 1, 4, 0.01)
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
```

```

sol:1.07910156250000 (n:9) #Resultado obtenido
# 2.1 Newton-Raphson
aux = metodo_NewtonRaphson(f, 1, 0.01)
print("sol:{:}_{:}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:1.07646496412713 (n:3) #Resultado obtenido

```

Coinciden con los resultados que esperabamos.

## 4.2 $f(x) = \sqrt{x} - \cos(x)$

Veamos gráficamente la función:

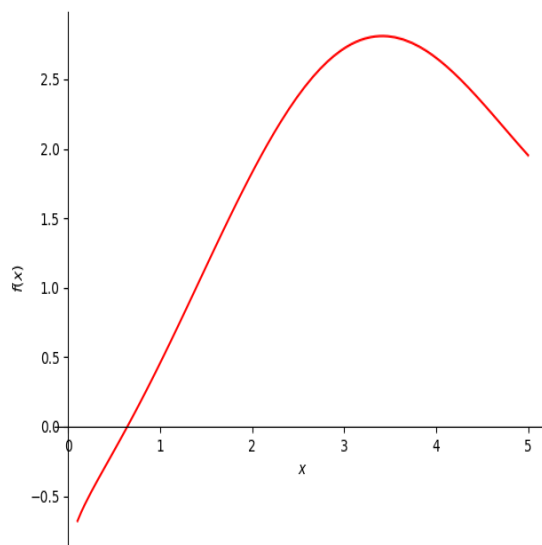


Figura 16.

Aplicamos los métodos en los puntos que nos piden:

```

# 2.2 a) biseccion
aux = metodo_Biseccion(f, 0, 4, 0.001)
print("sol:{:}_{:}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.642089843750000 (n:12) #Resultado obtenido
# 2.2 a) Newton-Raphson
aux = metodo_NewtonRaphson(f, 1, 0.001)
print("sol:{:}_{:}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.641714371002502 (n:3) #Resultado obtenido
# 2.2 b) biseccion
print("b)")
aux = metodo_Biseccion(f, 0.5, 1, 0.001)
print("sol:{:}_{:}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.642089843750000 (n:9) #Resultado obtenido
# 2.2 b) Newton-Raphson
aux = metodo_NewtonRaphson(f, 0.5, 0.001)
print("sol:{:}_{:}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.641714370872914 (n:3) #Resultado obtenido

```

Coinciden con los resultados esperados.

### 4.3 $f(x) = 2x^3 - 11.7x^2 + 17.7x^5$

Veamos gráficamente la función:

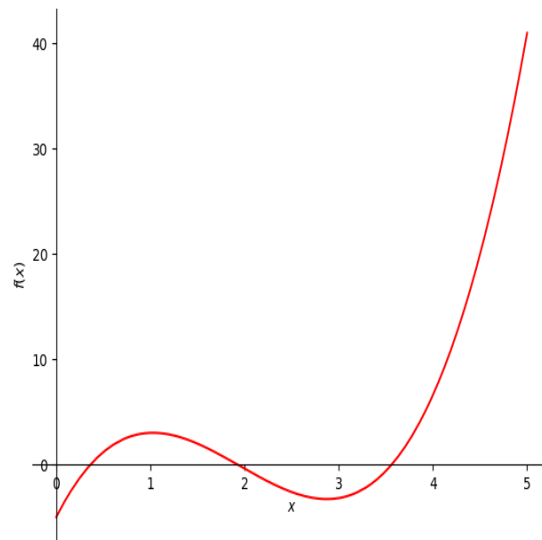


Figura 17.

Calculamos los valores pedidos:

```
print("a")
# 2.3 a) biseccion (0,1)
aux = metodo_Biseccion(f, 0, 1, 0.001)
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
sol: 0.364746093750000 (n: 10)
# 2.3 a) biseccion (1,3)
aux = metodo_Biseccion(f, 1, 3, 0.001)
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
sol: 1.92138671875000 (n: 11)
# 2.3 a) biseccion (3,5)
aux = metodo_Biseccion(f, 3, 5, 0.001)
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
sol: 3.56298828125000 (n: 11)
print("b")
# 2.3 b) Newton-Raphson 0.5
aux = metodo_NewtonRaphson(f, 0.5, 10 ** (-10))
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
sol: 0.365098243362236 (n: 5)
# 2.3 b) Newton-Raphson 1.5
aux = metodo_NewtonRaphson(f, 1.5, 10 ** (-10))
print("sol: {}_uu(n: {})".format(sympy.N(aux[0]), aux[1]))
```

```

sol:1.92174093177571 (n:5)
# 2.3 b)Newton-Raphson 4
aux = metodo_NewtonRaphson(f, 4, 10 ** (-10))
print("sol:{:}_{}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:3.56316082486206 (n:6)

```

Los resultados coinciden con lo esperado.

#### 4.4 $f(x) = e^{x/2} + 5x - 5$

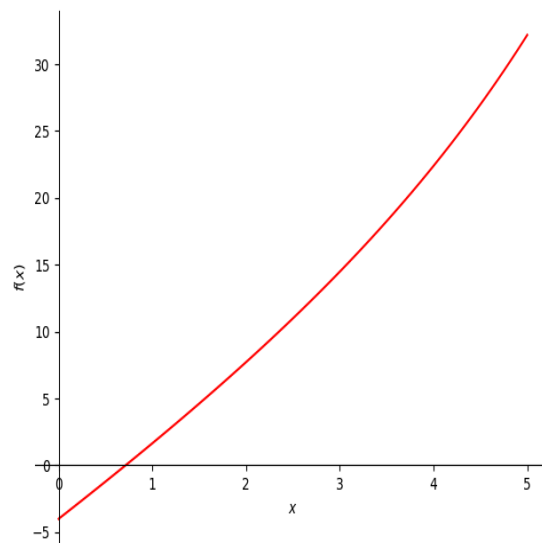


Figura 18.

Calculamos los valores:

```

# 2.4 a)biseccion
print("a")
aux = metodo_Biseccion(f, 0, 4, 0.001)
print("sol:{:}_{}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.714355468750000 (n:12)
# 2.4 b)Newton-Raphson
print("b")
aux = metodo_NewtonRaphson(f, 1, 0.001)
print("sol:{:}_{}(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.714168715029391 (n:3)

```

Coincide con lo esperado.

#### 4.5 $f(x) = x^2 + x + 3$

Veamos la gráfica de la función:

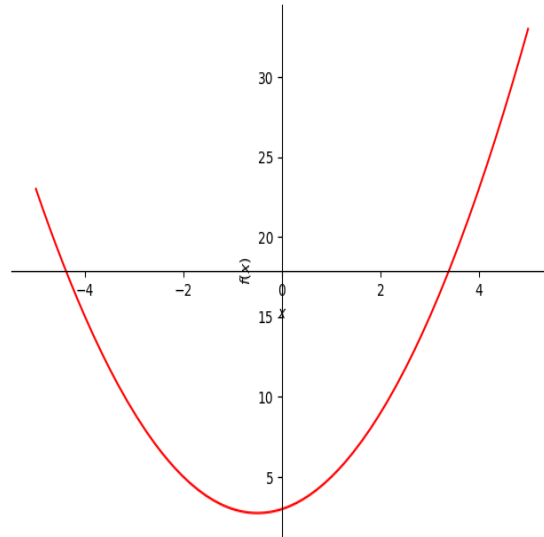


Figura 19.

Primero calculemos su derivada

```
f = x ** 2 + x + 3
f_1 = f.diff()
```

Que nos devuelve  $2x + 1$ .

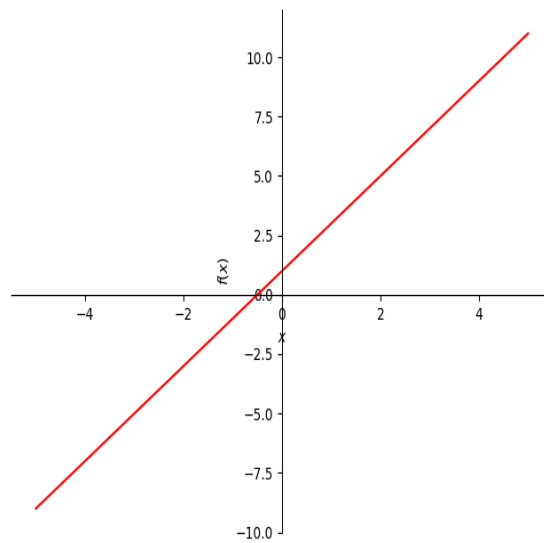


Figura 20.

Para calcular su valor máximo vamos a ver en que puntos del intervalo  $(-2, 2)$  la derivada de  $f(x)$  se anula. Lo haremos haciendo uso de los métodos del enunciado. Tomaremos como  $\varepsilon = 0.001$  y los puntos escogidos para el método de la bisección son  $a = -2, b = 2$ , para el método de Newton-Raphson  $x_0 = 0$  y para el método de la secante los puntos  $x_0 = -1.5, x_1 = 1.5$

```
# 2.5 a)biseccion
print("a)")
aux = metodo_Biseccion(f_1, -2, 2, 0.001)
print("sol:{},{}_{n:{}".format(sympy.N(aux[0]), f.subs(x, sympy.N(aux[0])),
aux[1]))
sol:-0.5000000000000000,2.7500000000000000 (n:2) #Resultado, imagen del Resultado
# 2.5 b)Newton-Raphson
print("b)")
aux = metodo_NewtonRaphson(f_1, 0, 0.001)
print("sol:{},{}_{n:{}".format(sympy.N(aux[0]), f.subs(x, sympy.N(aux[0])),
aux[1]))
sol:-0.5000000000000000,2.7500000000000000 (n:2) #Resultado, imagen del Resultado
# 2.5 c) Secante
aux = metodo_Secante(f_1, -1.5, 1.5, 0.001)
print("sol:{},{}_{n:{}".format(sympy.N(aux[0]), f.subs(x, sympy.N(aux[0])),
aux[1]))
sol:-0.5000000000000000,2.7500000000000000 (n:2) #Resultado, imagen del Resultado
```

Vemos que los valores coinciden con exactitud con el enunciado. Además, los tres métodos devuelven la misma solución.

## 5 Raíces de funciones no lineales - Parte 3

### 5.1 Presentación de la convergencia del método Newton-Rapshon

En este apartado queremos ilustrar como converge el método Newton-Rapshon. Para ello, primero recordamos que se denominan a  $K$  constante y a  $k$  orden de convergencia, a los números tales que:

$$\lim_{n \rightarrow \infty} \frac{|x_0^{n+1} - x_0|}{|x_0^n - x_0|^k} = K$$

Es conocido que en el caso de Newton-Rapshon el orden de convergencia es cuadrático y su constante de convergencia tiene valor:

$$K = \left| \frac{1}{2} \frac{f''(x_0)}{f'(x_0)} \right| \quad \text{En donde } f \text{ es la función analizada y } x_0 \text{ es la raíz obtenida} \quad (1)$$

No obstante, no conocemos como se comporta esta  $K$ , según el orden de convergencia. Por ello, para ilustrar dicho comportamiento realizaremos un ejemplo práctico donde anotaremos en una tabla los valores de  $K$  y  $E_n$  (error entre la estimación y la raíz real), según el número de iteraciones ( $n$ ) y según el valor del orden de convergencia.

### 5.2 Convergencia para $f(x) = x^3 - 3x + 2$ y $x_0 = -3$

Para analizar la convergencia de  $f(x)$ , tomamos una tolerancia de  $\varepsilon = 10^{-6}$  y modificamos el programa inicial del método de Newton-Rapshon para que admitiera un nuevo parámetro que le indicase la iteración en la que debía detenerse. Además, creamos una función auxiliar que encapsula al método de Newton-Rapshon, cuyo objetivo es devolver el valor final del método para una iteración dada, así como el error cometido en esa iteración y un array con los  $K$  para los órdenes de convergencia  $k=1$ ,  $k=2$  y  $k=3$ , que tiene el siguiente aspecto:

```
def conv_NewtonRapshon(f,x_m,epsilon,limit,x_0):

    if(limit>0):
        #SI limit NO ES 0,APLICAMOS NEWTON_RAPSON HASTA LA ITERACION limit(n)
        aux=metodo_NewtonRaphson(f,x_m,epsilon,limit=limit)
        """
        Haciendo limit =limit, basicamente estamos especificando que la
variable
        limit independientemente de donde se declarese en la cabecera de la
funcion metodo_NewtonRaphson tenga valor limit
        """
    else:
        # SI limit ES CERO, ENTONCES X_N=x_m
        aux=[x_m]
        #VALOR DEVUELTO POR NEWTON_RAPSON EN LA ITERACION limit(n+1)
        aux1 = metodo_NewtonRaphson(f,x_m,epsilon,limit=limit+1)

        #ERRORES E_n y E_{n+1}
        e_n = aux[0]-x_0
        e_n1= aux1[0]-x_0

        k=[]
        #ALMACENAMOS LOS K SEGUN la k
        for i in range(1,4):
            k.append(abs(e_n1)/(abs(e_n)**i))
        return aux[0],e_n,k
```



Tras esto, implementamos un bucle en el que se mostraban los resultados de la aplicacion de este método para valores de limit entre 0 y 4 (iteración a partir de la cual el método converge por lo que los siguientes resultados carecen de significado), donde obtuvimos los resultados que se reflejan en la siguiente tabla:

|      | $x_0^n$   | $E_n$                | $ E_{n+1} / E_{n+1} ^k$<br>k = 1 | $ E_{n+1} / E_{n+1} ^k$<br>k = 2 | $ E_{n+1} / E_{n+1} ^k$<br>k = 3 |
|------|-----------|----------------------|----------------------------------|----------------------------------|----------------------------------|
| n= 0 | -3        | -1                   | 0.055555                         | 0.055555                         | 0.055555                         |
| 1    | -2.055555 | -0.055555            | 0.035087                         | 0.631578                         | 11.36842                         |
| 2    | -2.001949 | -0.001949            | 0.001297                         | 0.665369                         | 341.334630                       |
| 3    | -2.000002 | -0.000002            | 0.000001                         | 0.666659                         | 263679.089                       |
| 4    | -2.000000 | $-4.2614 * 10^{-12}$ | 0                                | 0                                | 0                                |

**Tabla 1.**

De aquí podemos observar varias cosas, la primera es que la convergencia del método se da en n=5 como indicamos antes, lo cual explica por qué las contantes de convergencia acaban siendo 0 para n=4.

Por otro lado, podemos observar que mientras la K para el orden de convergencia 3 parece diverger, la de los órdenes 1 y 2 parece estabilizarse en los valores 0 y 0.6. Por tanto, podemos asegurar que el orden de convergencia es 2 ya que es el mayor orden que obtenemos y que la constante de convergencia es algo cercano a 0.6, que podemos verificarlo haciendo uso de la fórmula (1), en donde tendríamos:

$$f'(x) = 3x^2 - 3 \quad , \quad f''(x) = 6x \quad , \quad x_0 = -2$$

por lo que:

$$K = \left| \frac{1 * -12}{2 * (12 - 3)} \right| = \left| \frac{1 * -4}{2 * 3} \right| = \frac{2}{3} = 0.6$$

Con lo que no solo comprobamos de manera analítica que el orden de convergencia del método es de orden 2, sino que también vemos que la fórmula para obtener la constante de convergencia se cumple.

## 6 Resolución numérica de sistemas lineales

En este apartado estamos interesados en la resolución de sistemas de ecuaciones lineales. Centramos nuestra atención en los algoritmo de Gauss-Seidel y en el de Jacobi que describiremos a continuación.

Se considerará la siguiente notación:

### Notación 1.

*Dado un sistema de ecuaciones  $Ax=b$  denotaremos los elementos de  $A$  como  $a_{kj}$  donde la  $k$  representa la fila y  $j$  la columna,  $x_j^i$  donde  $i$  es la iteración y  $j$  el elemento del vector y  $b_j$  donde  $j$  es el elemento del vector.*

### 6.1 Descripción e implementación de los algoritmos

Los dos algoritmos que se van a implementar, pertenecen a la categoría de algoritmos iterativos, este tipo de algoritmos destacan por su simplicidad y por estar adaptados a sistemas grandes con matrices dispersas.

Además, estos algoritmos resultan ser convergentes si se da la condición de que son diagonalmente dominantes o lo que es lo mismo, si:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|$$

La condición de convergencia de ambos algoritmos es idéntica y se basa en comprobar elemento a elemento el valor absoluto de la resta de la solución en la iteración anterior con la solución en la iteración siguiente. De tal forma que si que en todos los elementos de los vectores se cumple que el valor absoluto de las restas es menor a epsilon entonces el algoritmo habrá alcanzado la condición de parada.

Esta comprobación fue implementada en una función, la cual devuelve un booleano a True si se da la condición de parada. Este metodo se presenta a continuación:

```
def convergencia(e,x_k,x_k1):
    i=0
    conv = True
    while(i<len(x_k) and conv):
        if(abs(x_k[i]-x_k1[i])> e):
            conv = False
        i+=1
    return conv
```

### Gauss-Seidel

El método Gauss-Seidel se retroalimenta puesto que los elementos del vector  $x$  se van actualizando con los elementos  $x$  de la iteración anterior y con los elementos  $x$  ya calculados en esta iteración, siguiendo la siguiente fórmula:

$$x_j^i = \frac{b_j - \sum_{k=1}^{j-1} a_{jk} x_k^i - \sum_{k=i+1}^n a_{jk} x_k^{i-1}}{a_{jj}} \quad (2)$$

Dicha fórmula y el algoritmo fueron implementados en esta función:

```
def Gauss_Seidel(A,b,x_0,e):
    #De momento es igual a jacobi
    x_k1 = np.copy(x_0)+2*e
    k=1
    x_k= np.copy(x_0)
    while(not convergencia(e,x_k1,x_k)):

        x_k1=np.copy(x_k)

        for i in range(len(x_0)):
            aux = b[i]
            for j in range(len(x_0)):
                if(j != i):
                    aux -=A[i,j]*x_k[j]#En vez de usar el vector de la
iteracion anterior
                    #usamos los valores del de la itearacion actual que depende
de la fila
                    # en la que estemos habran sido ya modificados
            x_k[i]=aux/A[i,i]
        print(k,":",x_k)
        k+=1

    return x_k
```

En dónde se puede observar que el algoritmo usa para implementar la fórmula (2) un vector que se inicializa con los valores de la iteración anterior y que se actualiza tras realizar el cálculo sobre uno de sus elementos, por lo que acaba conteniendo las soluciones de esta iteración y de la anterior.

### Jacobi

Este otro método es similar al de Gauss-Seidel, con la peculiaridad de que sólo usa los valores de la iteración anterior sin importar que elemento del vector x se esté actualizando. La fórmula para actualizar los elementos de x queda simplificada a:

$$x_j^i = \frac{b_j - \sum_{\substack{k=1 \\ k \neq j}}^n a_{jk} x_k^{i-1}}{a_{jj}} \quad (3)$$

De esta forma, el algoritmo en vez de usar el mismo vector que actualizamos para realizar los cálculos de los elementos de x para esta iteración, usaremos un vector copia del de la iteración anterior pero sin actualizarlo, quedando con el siguiente aspecto:

```
def jacobi(A,b,x_0,e):
    x_k1 = np.copy(x_0) + 2*e #Copiamos el vector x_0 y sumamos a todos sus
elementos
    #2e para que el bucle se ejecute al menos una vez (python no tiene repeat
until)
    x_k = np.copy(x_0) #Se copia un vector con los mismos elementos
    k=1 #Variable que almacena el numero de la iteración

    while (not convergencia(e, x_k1, x_k)):#Mientras no haya convergencia
        x_k1 = np.copy(x_k)
        for i in range(len(x_0)): # recorrido de las filas
            aux = b[i]
```

```

for j in range(len(x_0)): # recorrido en columnas
    if (j != i):
        aux -= A[i, j] * x_k1[j]
    x_k[i] = aux/A[i,i] #Dividimos por el elemento de la diagonal
print(k, ":", x_k)
k+=1
return x_k

```

## 6.2 Problemas planteados y su resolución

Para comprobar el funcionamiento de ambos algoritmos, dos problemas fueron planteados. En esta sección enunciaremos dichos problemas y veremos un par de pasos iterativos así como las soluciones que ambos métodos arrojaron.

$$\begin{aligned}
 &3x_1 - 0.1x_2 - 0.2x_3 = 7.85 \\
 1) \quad &0.1x_1 + 7x_2 - 0.3x_3 = -19.3 \\
 &0.3x_1 - 0.2x_2 + 10x_3 = 71.4
 \end{aligned}$$

En este sistema se pedía obtener una solución partiendo del vector inicial 0 y una tolerancia variable. En este caso optamos por una tolerancia  $\varepsilon = 0.1$  y sabiendo que los resultados eran  $x_1 = 3; x_2 = -2.5; x_3 = 7$ , obtuvimos los resultados recogidos en la siguiente tabla:

| Iteración    | $x^1$                    | $x^2$                    | Final                            |
|--------------|--------------------------|--------------------------|----------------------------------|
| Gauss-Seidel | [ 2.6166 -2.7945 7.0056] | [ 2.9905 -2.4996 7.0002] | $x^3 = [ 3.00003 -2.4999 6.999]$ |
| Jacobi       | [ 2.6166 -2.7571 7.14 ]  | [ 3.0007 -2.4885 7.0063] | $x^3 = [ 3.0008 -2.4997 7.0002]$ |

Tabla 2.

$$\begin{aligned}
 &5x_1 + 2x_2 - x_3 + x_4 = 12 \\
 2) \quad &x_1 + 7x_2 + 3x_3 - x_4 = 2 \\
 &-x_1 + 4x_2 + 9x_3 + 2x_4 = 1 \\
 &x_1 - x_2 + x_3 + 4x_4 = 3
 \end{aligned}$$

En este otro sistema se pedía obtener una solución partiendo del vector inicial 0 y una tolerancia variable. En este caso optamos por una tolerancia  $\varepsilon = 0.1$  y sabiendo que los resultados eran  $x_1 = 2.7273; x_2 = -0.4040; x_3 = 0.6364; x_4 = -0.1919$ , obtuvimos los resultados recogidos en la siguiente tabla:

| iteración    | $x^1$                       | Final                                    |
|--------------|-----------------------------|--|
| Gauss-Seidel | [ 2.4 -0.057 0.4031 0.0349] | $x^4 = [ 2.6533 -0.3441 0.5841 -0.1453]$ |
| Jacobi       | [ 2.4 0.2857 0.1111 0.75 ]  | $x^5 = [ 2.5183 -0.2006 0.4362 0.0148 ]$ |

Tabla 3.

Con estos dos ejemplos ya se puede observar que el método de Gauss-Seidel converge más rápido que el de Jacobi. Además, dicha convergencia se da en números más cercanos a la solución real. Para hacerlo más evidente vamos a mostrar 4 tablas más con los resultados arrojados al imponer  $\varepsilon_1 = 0.01, \varepsilon_2 = 0.001$

Ejercicio 1( $\varepsilon_1$ )

|              | Final                                     |
|--------------|---|
| Gauss-Seidel | $x^3 = [ 3.00003 \ -2.4999 \ 6.99999 ]$   |
| Jacobi       | $x^4 = [ 3.00002 \ -2.500002 \ 6.99998 ]$ |

**Tabla 4.**Ejercicio 1( $\varepsilon_2$ )

|              | Final   |
|--------------|---|
| Gauss-Seidel | $x^4 = [ 3.0000003 \ -2.50000004 \ 6.99999 ]$ |
| Jacobi       | $x^4 = [ 3.00002 \ -2.500002 \ 6.99998 ]$     |

**Tabla 5.**Ejercicio 2( $\varepsilon_1$ )

| iteración    | Final  |
|--------------|--|
| Gauss-Seidel | $x^8 = [ 2.7182 \ -0.3967 \ 0.6300 \ -0.1862 ]$    |
| Jacobi       | $x^{12} = [ 2.6974 \ -0.3765 \ 0.6073 \ -0.1633 ]$ |

**Tabla 6.**Ejercicio 2( $\varepsilon_2$ )

|              | Final  |
|--------------|--|
| Gauss-Seidel | $x^{12} = [ 2.7261 \ -0.4031 \ 0.6355 \ -0.1912 ]$ |
| Jacobi       | $x^{21} = [ 2.7248 \ -0.4018 \ 0.6340 \ -0.1896 ]$ |

**Tabla 7.**

Con lo que verificamos la tendencia que habíamos observado.