Práctica 1

por Juan Montes Cano, Cristian Pérez Corral y Juan Antonio Gordillo Gayo

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 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 libreria 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

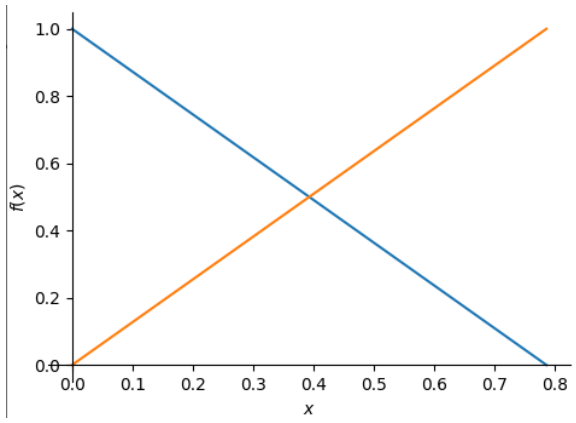

Figura 1.

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

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

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

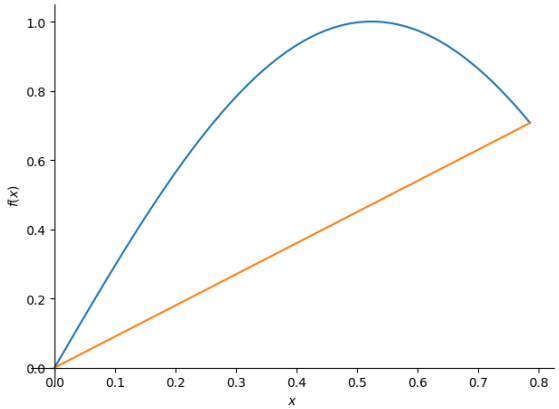

Figura 2.

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

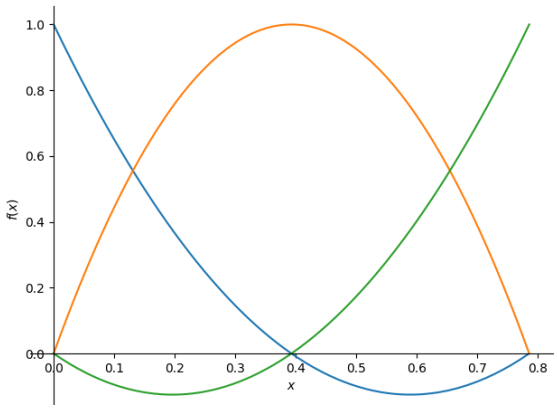

Figura 3.

y la aproximación quedará tal que

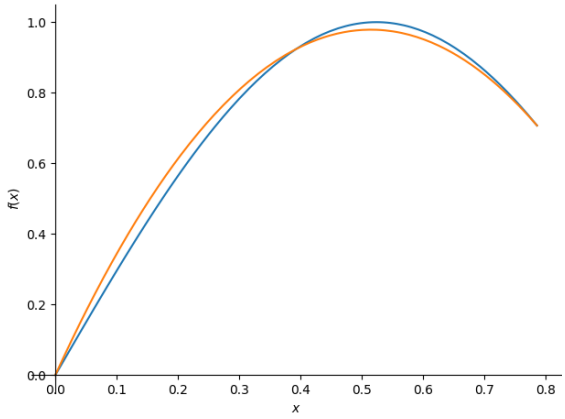

Figura 4.

es decir, obtenemos los mismos resultados que el ejercicio.

1.4 Ejemplo 2

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

1. 2 puntos

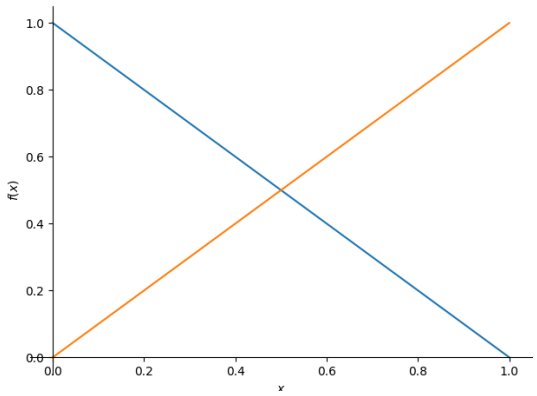

Figura 5.

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

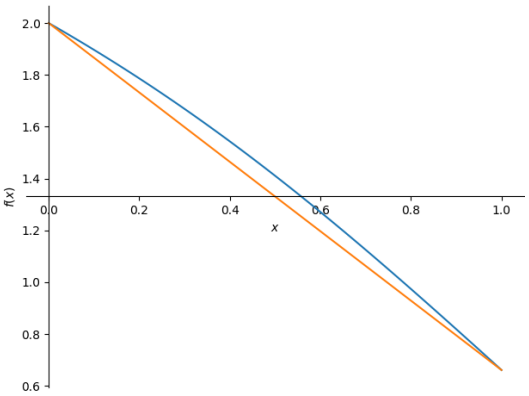

Figura 6.

2. 3 puntos

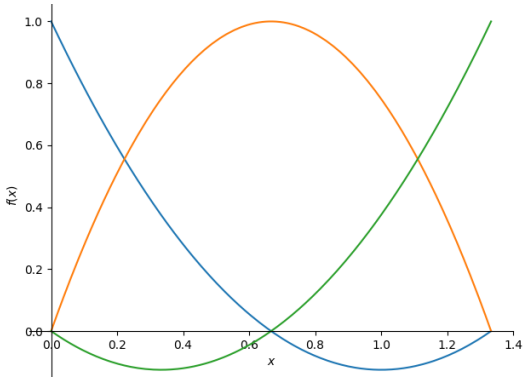

Figura 7.

con $L_0 = -0.75\,(1.0 - 1.5\,x)\,(x - 1.33333333333333)$, $L_1 = -2.25\,x\,(x - 1.33333333333333)$, y $L_2 = 1.125\,x\,(x - 0.666666666666667)$, de donde obtenemos la siguiente aproximación

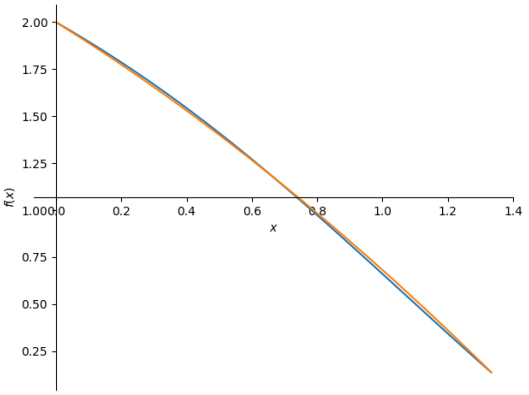

Figura 8.

3. 4 puntos

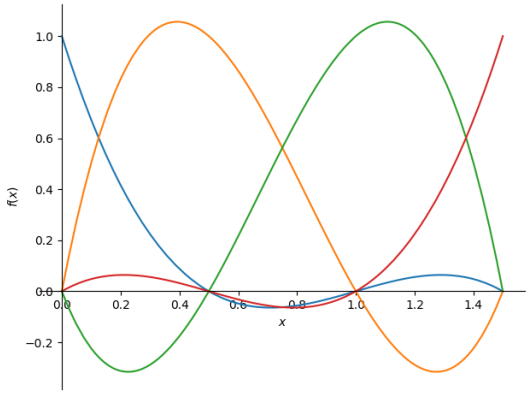

Figura 9.

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

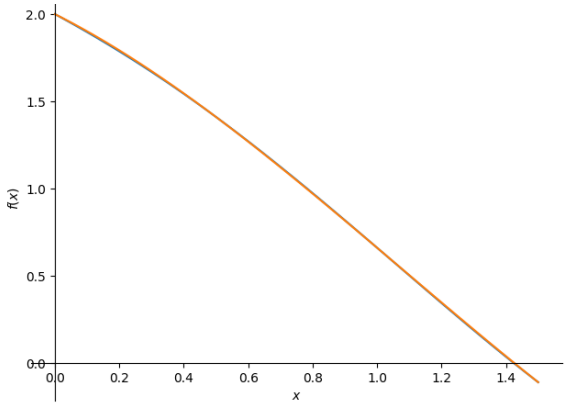

Figura 10.

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

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

$$p_n(x) = f(b) \to I \simeq \int_a^b p_n(x)\,\mathrm{dx} = f(b)\,(b-a) \qquad \text{(extremo derecho)}$$

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

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 *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.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

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

y nosotros la hemos implementado con el siguiente método

```
def fTrapecio(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:

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

```
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 $\int_0^2 x^3 - 2x^2 + 1\,\mathrm{dx}$. 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 $\int_0^2 p_n(x) = 0$

- Fórmla 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 $\int_0^{2\pi} \cos(x^2 - 1)\,\mathrm{dx}$.

La función que hemos definido tiene esta forma

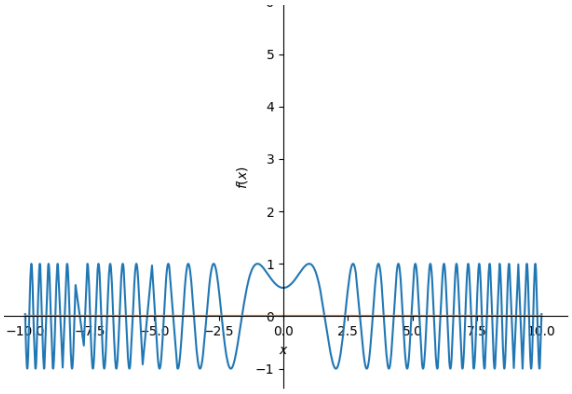

Figura 11.

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 ≈ 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 ≈ 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 ≈ 6.7969

- Fórmla trapecio:

  El resultado de este es $\int_0^{2\pi} p_n(x) = 3.93234382800083$, y el error calculado da ≈ 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 ≈ 3.4415

Pregunta Optativa) La fómula 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

$$\int_1^3 (\cos(x) - x\,\mathrm{sen}(x))\,\mathrm{dx}$$

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 $2^{\{\text{iteración}n\}}$ 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 númericas 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 ε.

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 $\varepsilon = 0.0001$.

Al aplicar la función con estos paramétros

```
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 gráfico de la función y el resultado del algoritmo:

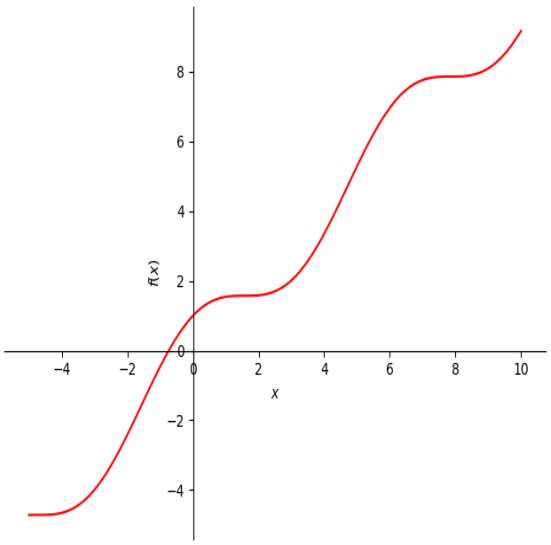

Figura 12.

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:

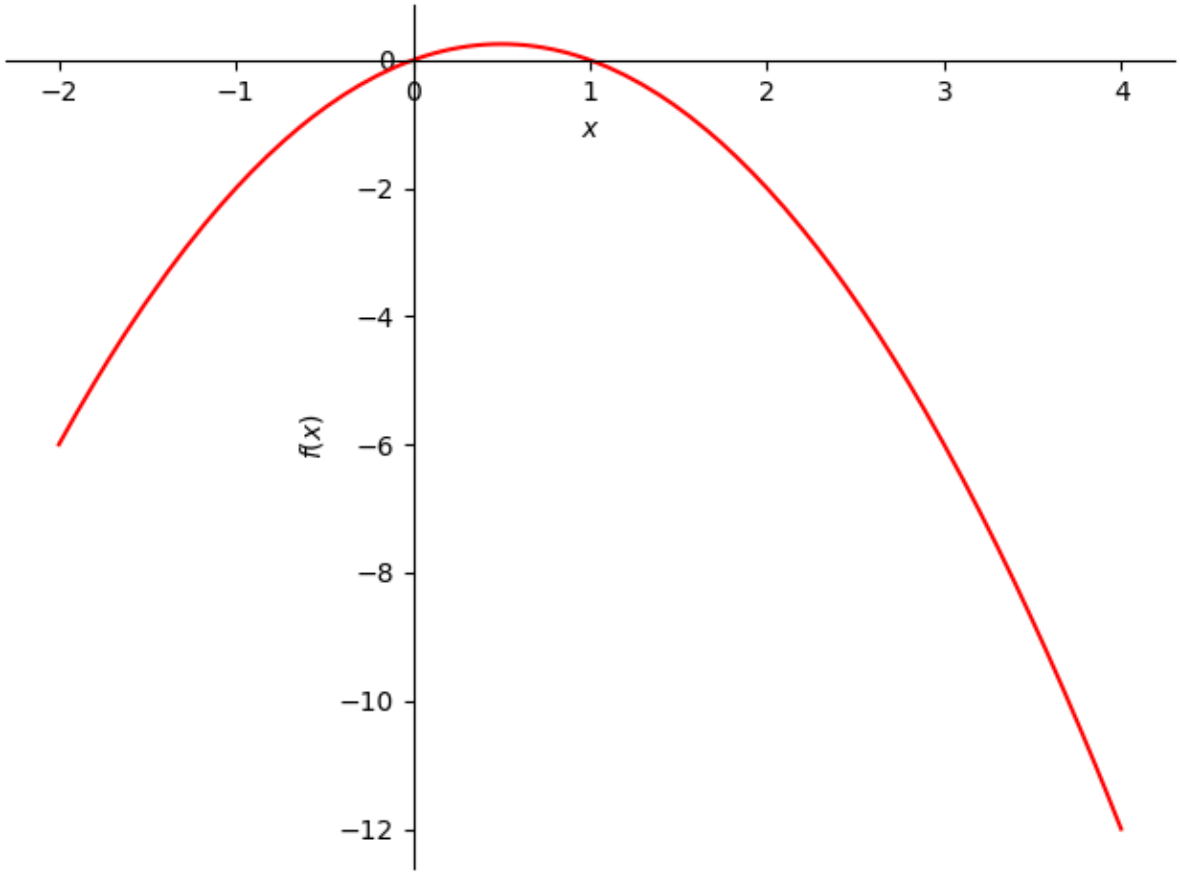

Figura 13.

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 $\mathcal{C}^1$ f, una valor de inicio x_0 y una cota de error ε.

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 ε y luego tenemos dos valores que nos permiten controlar tantas 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 $f(x) = x\,\text{sen}\left(\frac{x^2}{2}\right) + e^{-x}$

Primero vamos a ver gráficamente como se comporta esta función:

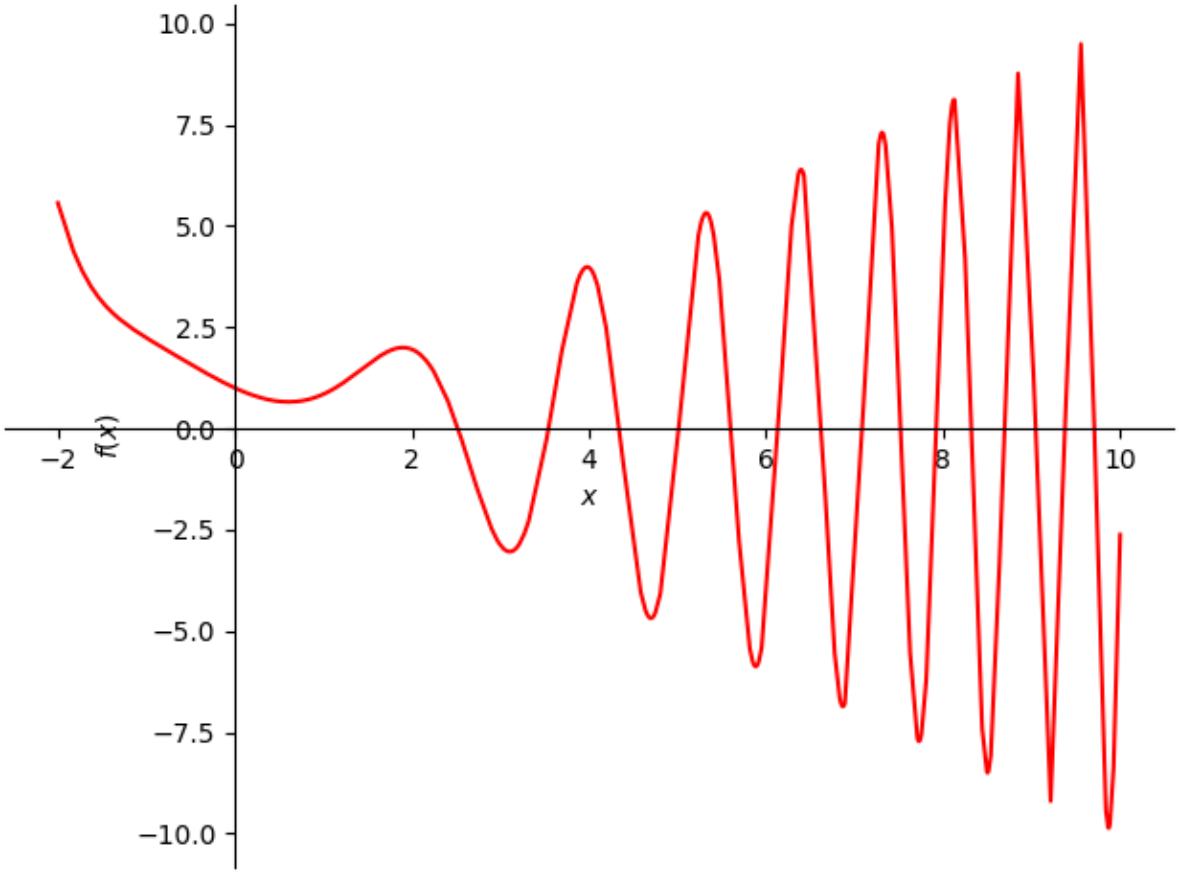

Figura 14.

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 intervalores 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:{}␣␣␣(n:{})".format(sympy.N(resultado[0]),resultado[1]))
sol:2.51660156250000  (n:9)
resultado = metodo_Biseccion(f, 3, 4, 0.01)
print("sol:{}␣␣␣(n:{})".format(sympy.N(resultado[0]),resultado[1]))
sol:3.54296875000000  (n:7)
resultado = metodo_Biseccion(f, 4, 5, 0.01)
print("sol:{}␣␣␣(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:{}␣␣(n:{})".format(sympy.N(resultado[0]),resultado[1]))
sol:2.51935360957807  (n:7)
resultado = metodo_NewtonRaphson(f,3,0.01)
print("sol:{}␣␣(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 $f(x) = \text{sen}(x) - 0.3\,e^x$

Primero mostremos gráficamente como es la función:

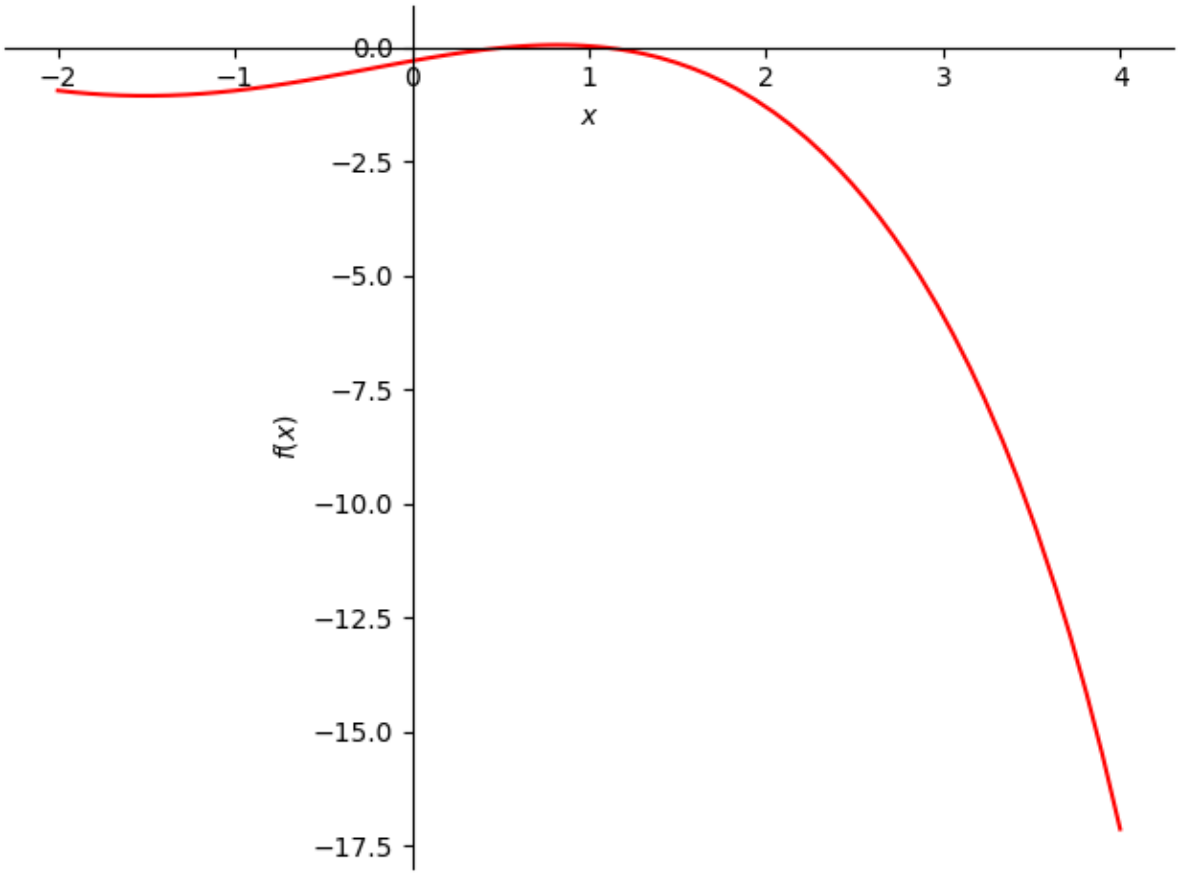

Figura 15.

Ahora aplicamos los métodos:

```
# 2.1 Bisección
aux = metodo_Biseccion(f, 1, 4, 0.01)
print("sol:{}␣␣(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:1.07910156250000  (n:9) #Resultado obtenido
# 2.1  Newton-Rapshon
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:

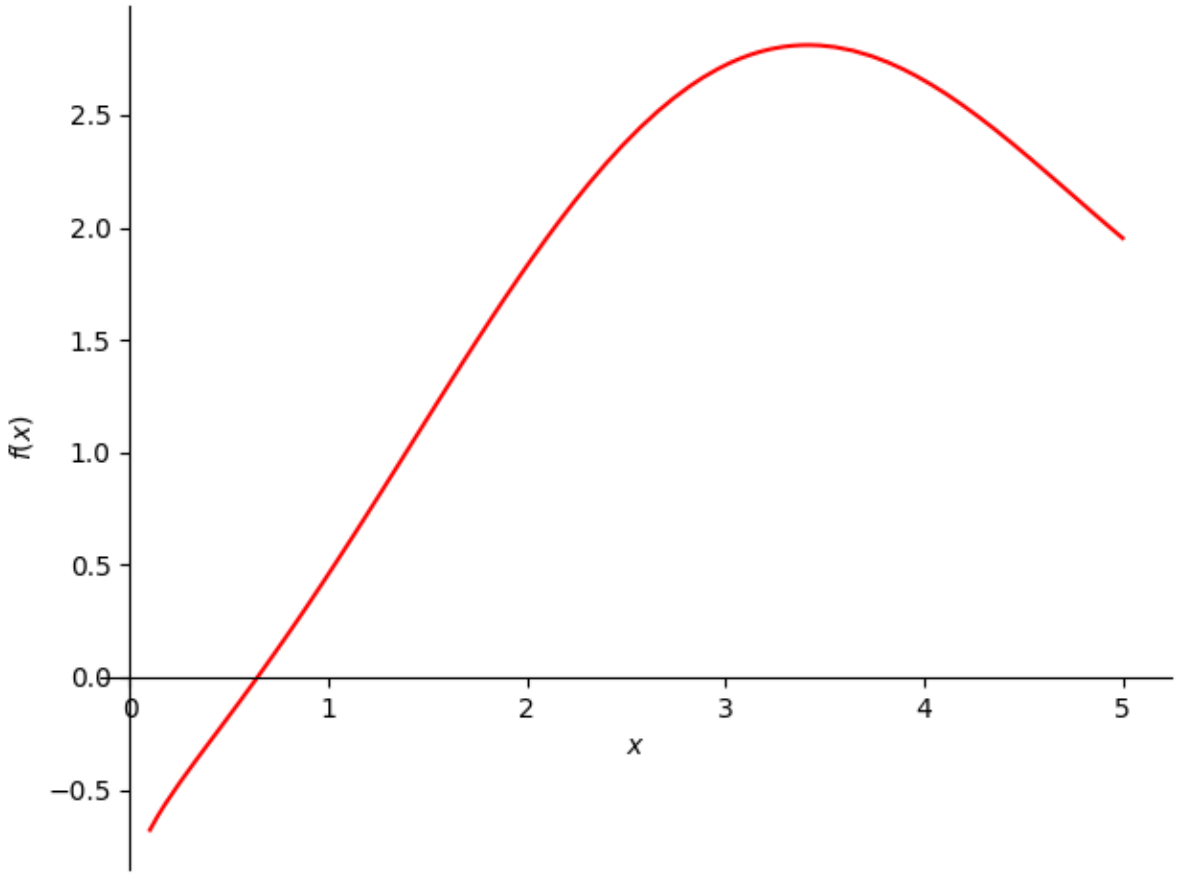

Figura 16.

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-Rapshon
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-Rapshon
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 $f(x) = 2\,x^3 - 11.7\,x^2 + 17.7\,x^5$

Veamos gráficamente la función:

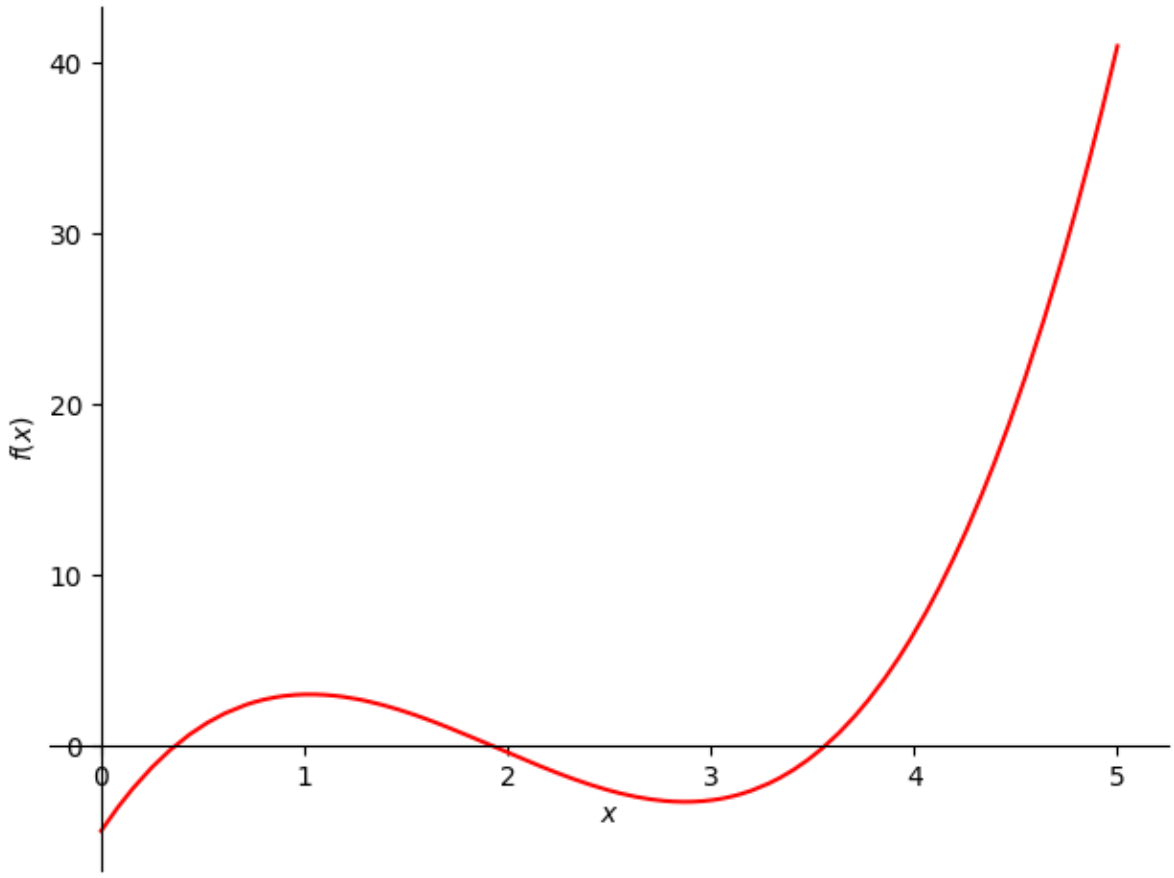

Figura 17.

Calculamos los valores pedidos:

```
print("a)")
# 2.3 a)biseccion (0,1)
aux = metodo_Biseccion(f, 0, 1, 0.001)
print("sol:{}␣␣(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:{}␣␣(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:{}␣␣(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:3.56298828125000  (n:11)
print("b)")
# 2.3 b)Newton-Rapshon 0.5
aux = metodo_NewtonRaphson(f, 0.5, 10 ** (-10))
print("sol:{}␣␣(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:0.365098243362236  (n:5)
# 2.3 b)Newton-Rapshon 1.5
aux = metodo_NewtonRaphson(f, 1.5, 10 ** (-10))
print("sol:{}␣␣(n:{})".format(sympy.N(aux[0]), aux[1]))
sol:1.92174093177571  (n:5)
# 2.3 b)Newton-Rapshon 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 $f(x) = e^{x/2} + 5\,x - 5$

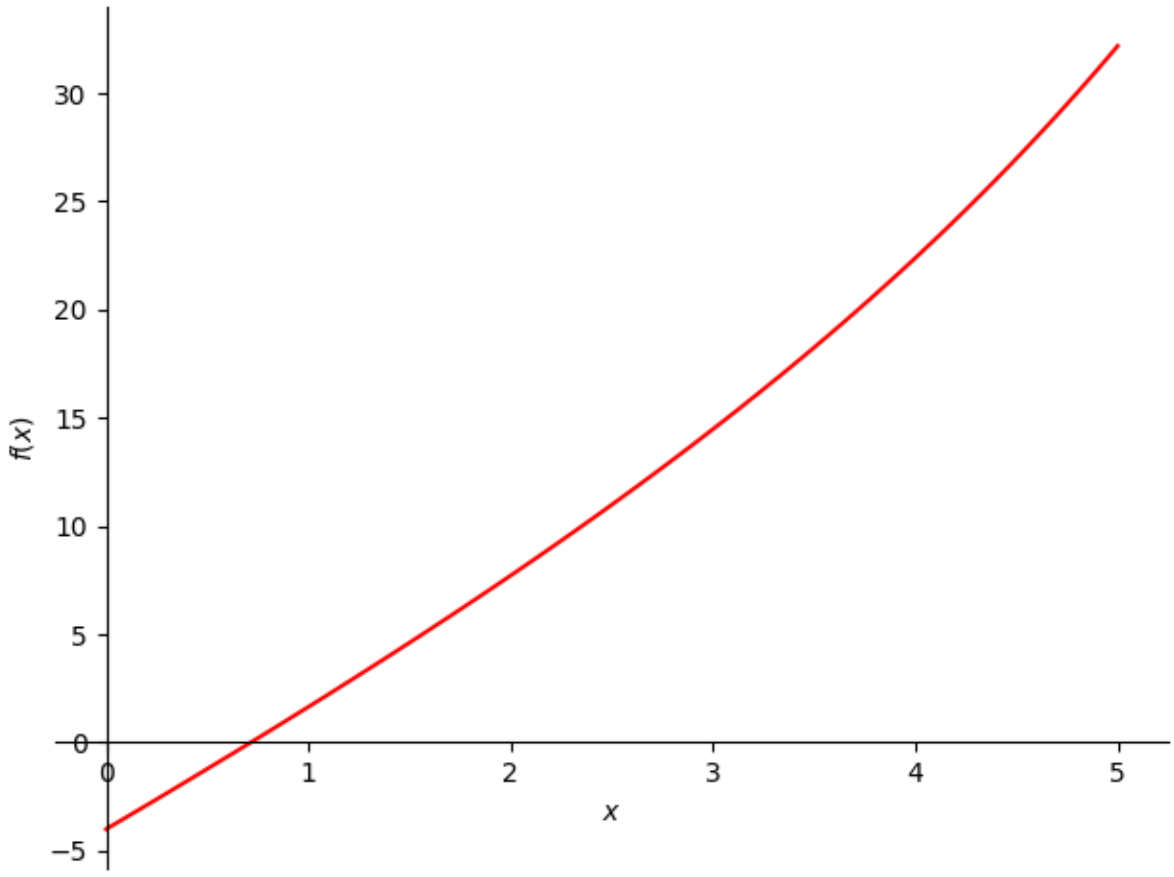

Figura 18.

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:

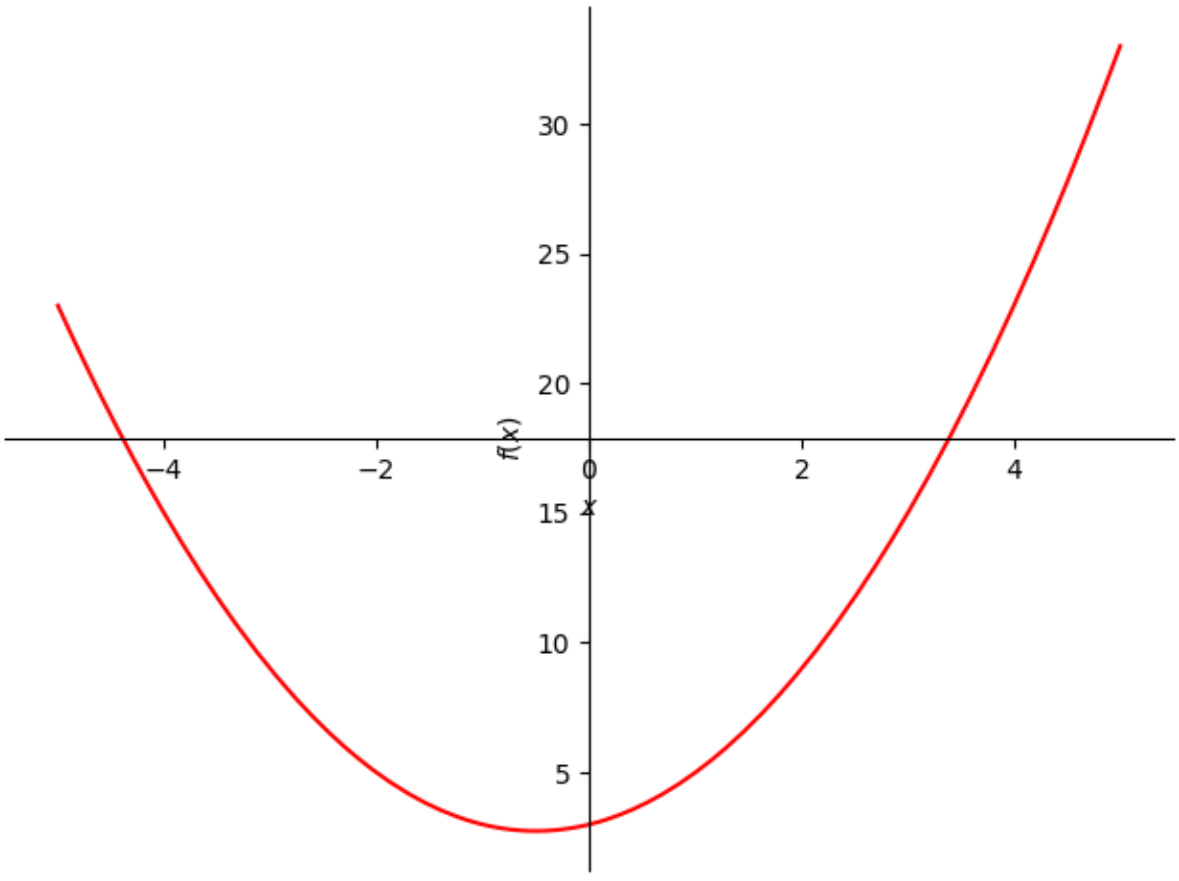

Figura 19.

Primero calculemos su derivada

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

Que nos devuelve $2\,x+1$.

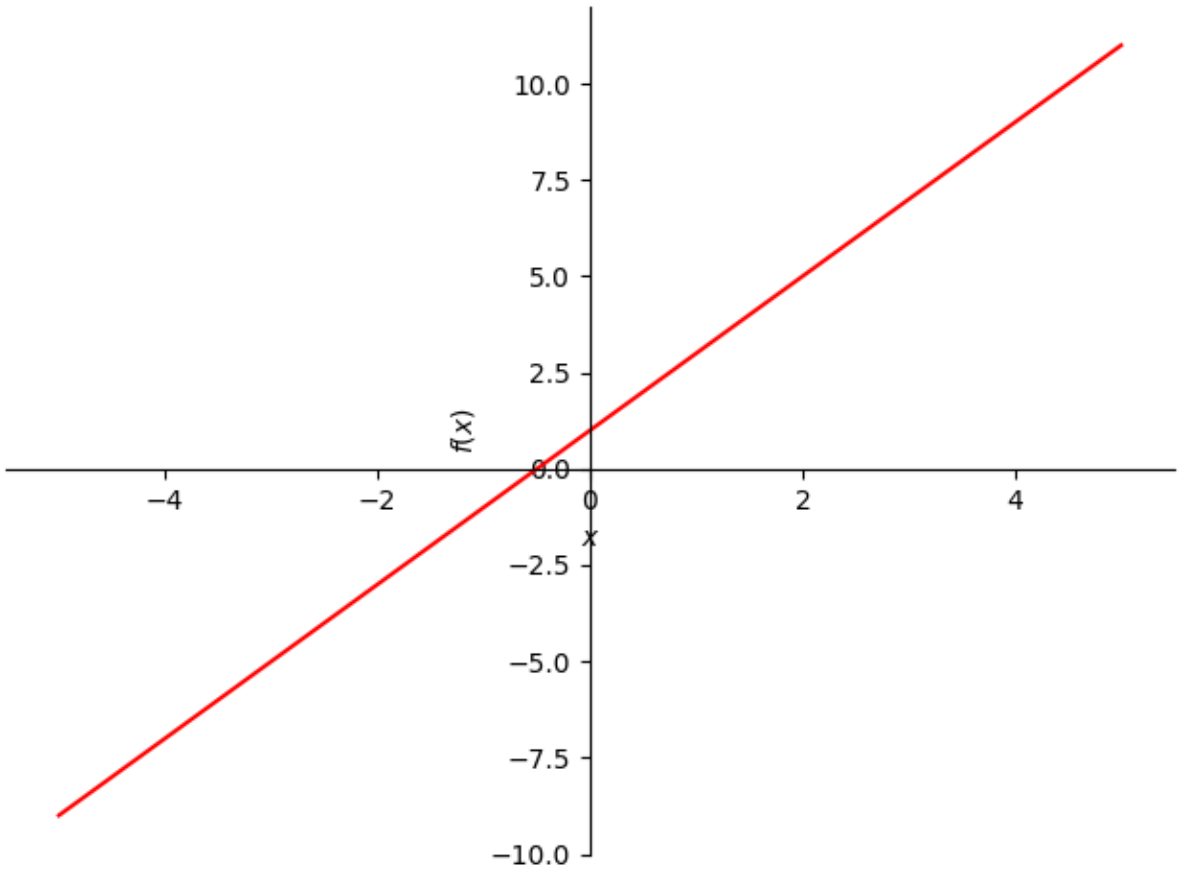

Figura 20.

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:-0.500000000000000,2.75000000000000  (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:-0.500000000000000,2.75000000000000  (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:-0.5000000000000000,2.75000000000000  (n:2) #Resultado, imagen del Resultado
```

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