Raíces de ecuaciones no lineales II

Importamos los módulos `matplotlib.pyplot` y `numpy` .

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

Método de la secante

Es una variante del método de Netwon-Raphson. Sustituimos la tangente por una secante.

El problema del método de Newton-Raphson es que necesitamos tener la $f'(x)$.

En el método de la secante aproximamos $f'(x)$ con:

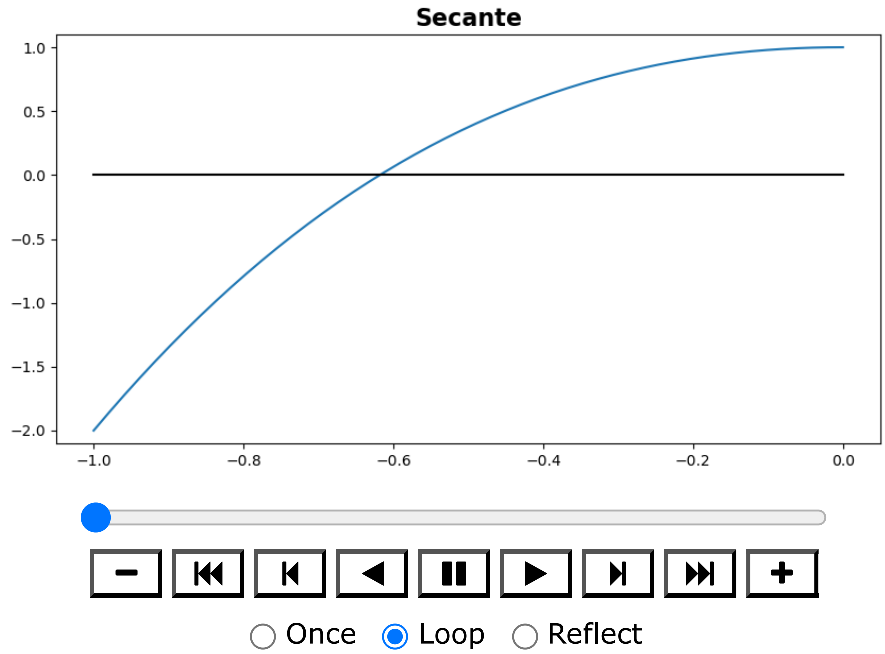
$$f'(x_{k-1}) \approx \frac{f(x_{k-1}) - f(x_{k-2})}{x_{k-1} - x_{k-2}}.$$

Necesitamos **dos puntos iniciales**, x_0 y x_1 para la primera iteración:

$$x_k = x_{k-1} - f(x_{k-1}) \frac{x_{k-1} - x_{k-2}}{f(x_{k-1}) - f(x_{k-2})}$$

En la animación vemos la sucesión de rectas secantes (en rojo) para calcular la primera raíz tomando $x_0 = 0$ y $x_1 = -1$.

```
secant
```



Algorithm

- Sean x_0 y x_1 los puntos iniciales.
- Para $k=1,2,\ldots,\mathrm{MaxIter}$:
  - Calcular
    $$x_k = x_{k-1} - f(x_{k-1})\frac{x_{k-1}-x_{k-2}}{f(x_{k-1})-f(x_{k-2})}.$$
  - Si x_k satisface el criterio de parada, parar.
  - Caso contrario, hacer otra iteración.

Ejercicio 1

Escribir una función llamada **`secante(f,x0,x1,tol = 1.e-6,maxiter=100)`** que tenga como argumentos de entrada la función lambda **`f`** , los puntos iniciales **`x0`** y **`x1`** , la cota de error absoluto **`tol`** y el número máximo de iteraciones y como argumentos de salida la solución aproximada y el número de iteraciones realizadas.

Utilizarla para aproximar las tres raíces de la función $f(x) = x^5 - 3x^2 + 1.6$, con **`tol = 1.e-6`** y **`maxiter = 100`** .

Utilizar los intervalos **`[a,b]`** obtenidos en el **Ejercicio 1** (búsqueda incremental) de la práctica anterior con **`x0 = a`** y **`x1 = b`** .

Escribir las raíces y el número de iteraciones.

Dibujar la función y las raíces (como puntos rojos sobre el eje OX).

Nota

Usar un array numpy para guardar las raíces. Se puede inicializar como

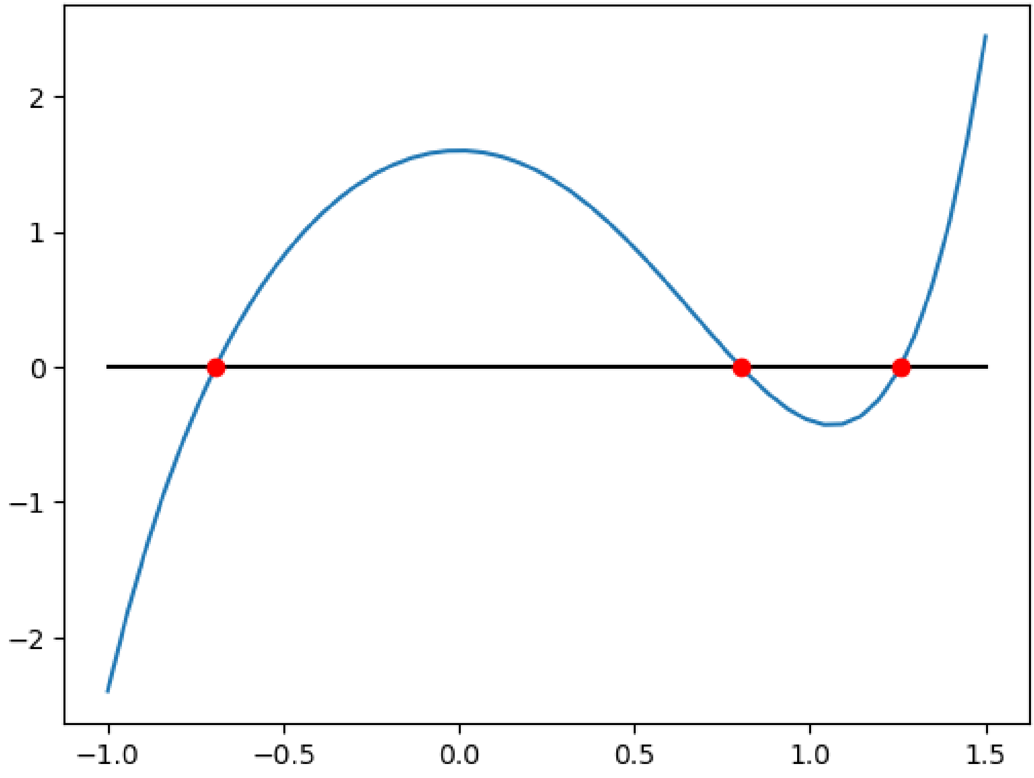
```
r = np.zeros(3)
```

Entonces, las raíces se pueden dibujar

```
plt.plot(r,r*0,'ro')
```

```
%run Ejercicio1
```

```
-0.6928908017616315 4
0.8027967742658026 4
1.2573997082378758 5
```



Funciones de python para calcular raíces

SciPy es una biblioteca open source de herramientas y algoritmos matemáticos para Python. SciPy contiene módulos para optimización, álgebra lineal, integración, interpolación, funciones especiales, FFT, procesamiento de señales y de imagen, resolución de ODEs y otras tareas para la ciencia e ingeniería.

Las funciónes que calculan raíces están incluídas dentro del módulo de optimización (scipy.optimize (http://docs.scipy.org/doc/scipy/reference/optimize.html)).

Si solo queremos usar las funciones de un módulo, pordemos cargar únicamente ese módulo.

```
import scipy.optimize as op
```

La función newton (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html) calcula las raíces utilizando el método de la secante o el de Newton-Raphson. La función bisect (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html#scipy.opt calcula las raíces utilizando el método de bisección.

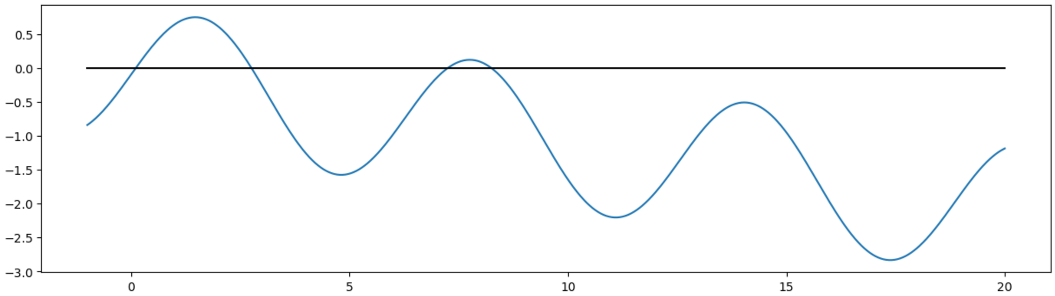
Veamos como utilizando los métodos de `newton` y la `secante` y usando la función `newton` del módulo `scipy.optimize` calculamos todas las soluciones positivas de la ecuación $\mathrm{sen}x - 0.1x - 0.1 = 0$.

Para el método de Newton necesitamos la función y su derivada

```
f  = lambda x: np.sin(x) - 0.1*x - 0.1
df = lambda x: np.cos(x) - 0.1
```

Dibujamos la función

```
x = np.linspace(-1,20,1000)

plt.figure(figsize=(15,4))
plt.plot(x,f(x))
plt.plot(x,0*x,'k-')
plt.show()
```



Tomando puntos próximos a las raíces (que estimamos sobre la gráfica) obtenemos las raíces.

```
x0 = np.array([0., 2., 7., 8.])
raiz = op.newton(f,x0,fprime=df,tol=1.e-6,maxiter=100)
print(raiz)
```

```
[0.11136674 2.75649514 7.25411627 8.2450445 ]
```

Si no utilizamos la derivada, la función usará el método de la secante

```
x0 = np.array([0., 2., 7., 8.])
raiz = op.newton(f,x0,tol=1.e-6,maxiter=100)
print(raiz)
```

```
[0.11136674 2.75649514 7.25411627 8.2450445 ]
```

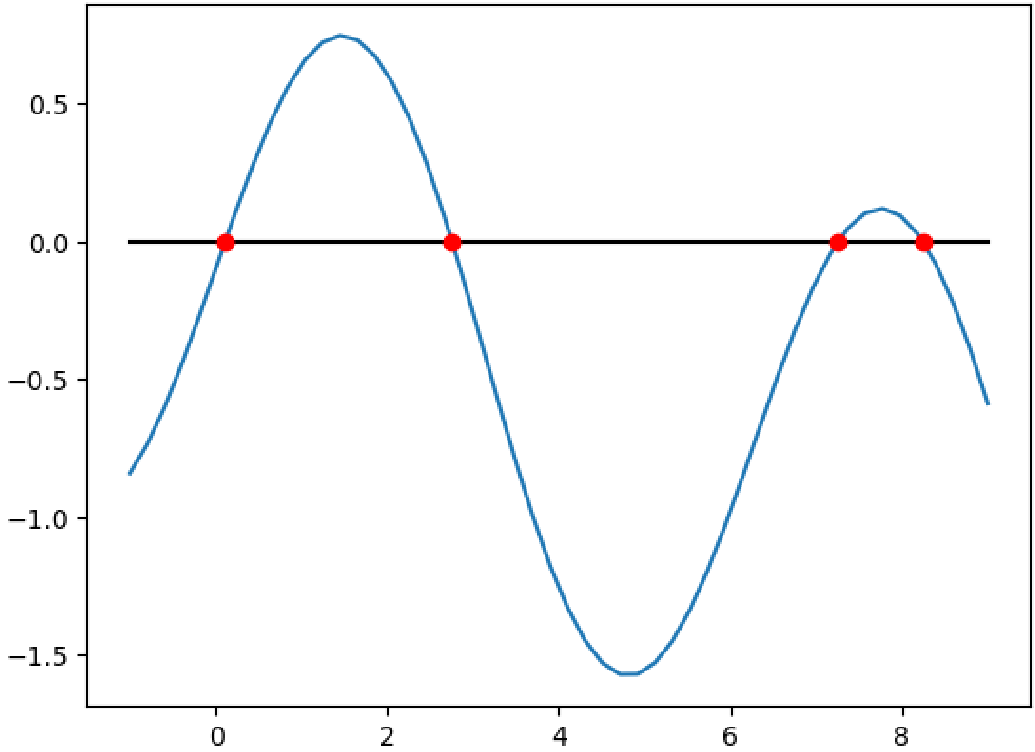
Si usamos Bisección necesitamos estimar los intervalos iniciales

```
x0 = np.array([0., 2., 7., 8.])
x1 = x0 + 1
raiz = np.zeros_like(x0)
for i in range(len(raiz)):
    raiz[i]= op.bisect(f,x0[i],x1[i],xtol=1.e-6,maxiter=100)
    print(raiz[i])
```

```
0.11136722564697266
2.756495475769043
7.254117012023926
8.245043754577637
```

Comprobemos las raíces gráficamente

```
x = np.linspace(-1,9)

plt.figure()
plt.plot(x,f(x),x,x*0,'k',raiz,raiz*0,'ro')
plt.show()
```



Cálculo de extremos y puntos de inflexión de una función

Si f es una función derivable en un punto x_0 de su dominio y x_0 es un extremo relativo, entonces

$$f'(x_0) = 0$$

Es decir, los extremos son raíces de la función derivada.

Si f es una función derivable dos veces en un punto x_0 de su dominio y x_0 es un punto de inflexión, entonces

$$f''(x_0) = 0$$

Es decir, los puntos de inflexión son raíces de la función derivada segunda.

Ejercicio 2

Dada la función:

$$f(x) = x^2 + \ln(2x+7)\cos(3x) + 0.1$$

Calcular los extremos relativos (máximos y mínimos) de f, es decir,las raíces de f':

- Definir las funciones lambda de f, f' y f'' utilizando el módulo `sympy` (mirar **Nota** al final de esta práctica).
- Dibujar f' con el eje OX y probar con distintos intervalos para el valor de x, hasta encontrar un intervalo que contenga todas sus raíces.
- Obtener con un error máximo de 10^{-6} todos los ceros de la función f' utilizando la función `newton` de `scipy.optimize` .
- Dibujar f, de forma que contenga todos sus extremos relativos y, mirando la gráfica, decidir cuales de los puntos anteriores son máximos y cuales mínimos. Dibujar sobre la función los máximos relativos con puntos rojos y los mínimos con puntos verdes.

Calcular los puntos de inflexión de f contenidos en $[-1, 4]$.

- Dibujarlos sobre la gráfica con puntos azules.

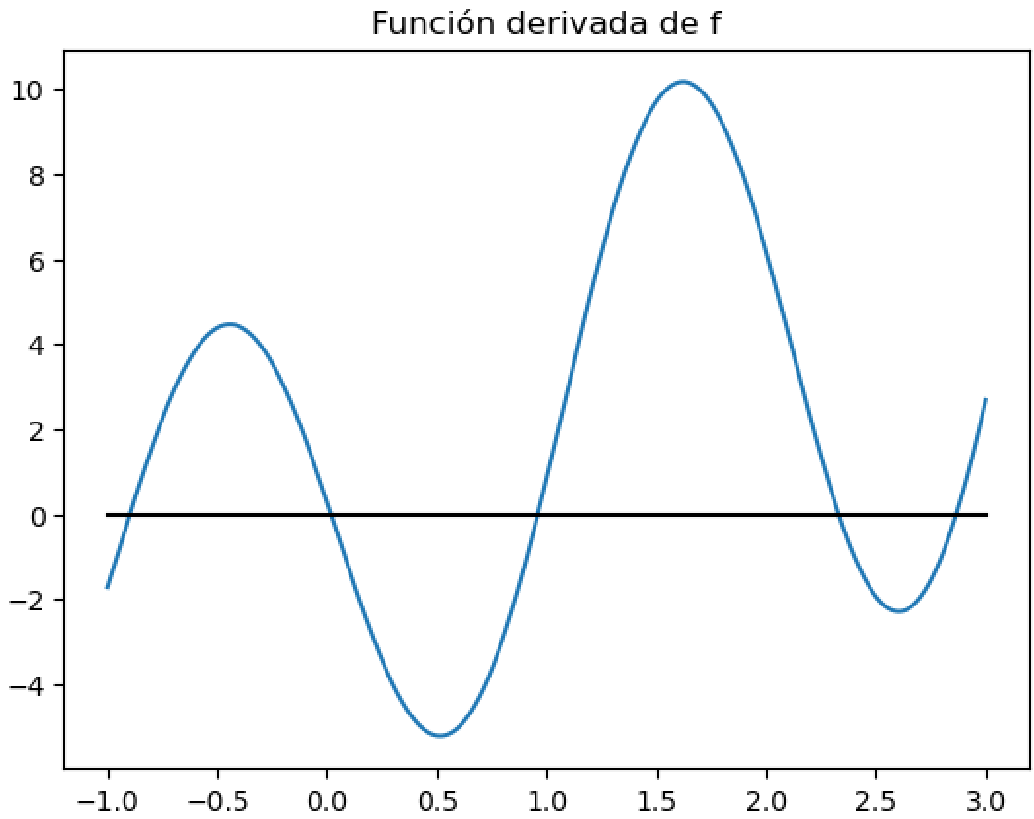
Nota:

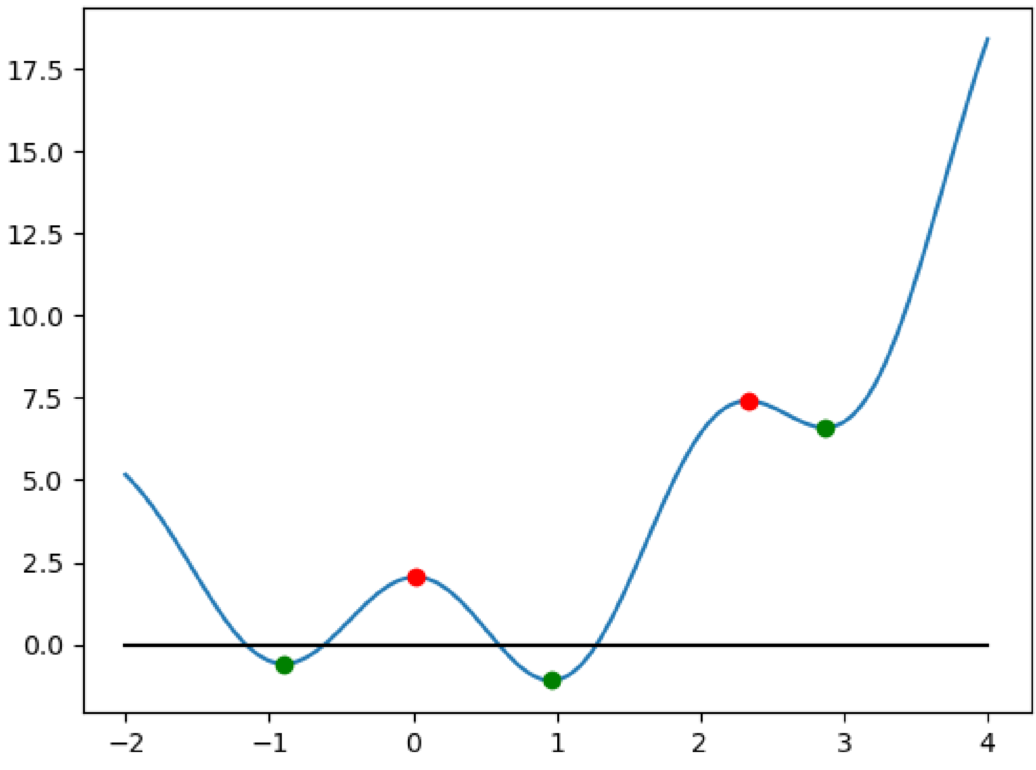
[Lista de funciones matemáticas elementales en numpy (https://numpy.org/doc/stable/reference/routines.math.html)](https://numpy.org/doc/stable/reference/routines.math.html).

[Lista de funciones matemáticas elementales en sympy (https://docs.sympy.org/latest/modules/functions/index.html)](https://docs.sympy.org/latest/modules/functions/index.html)

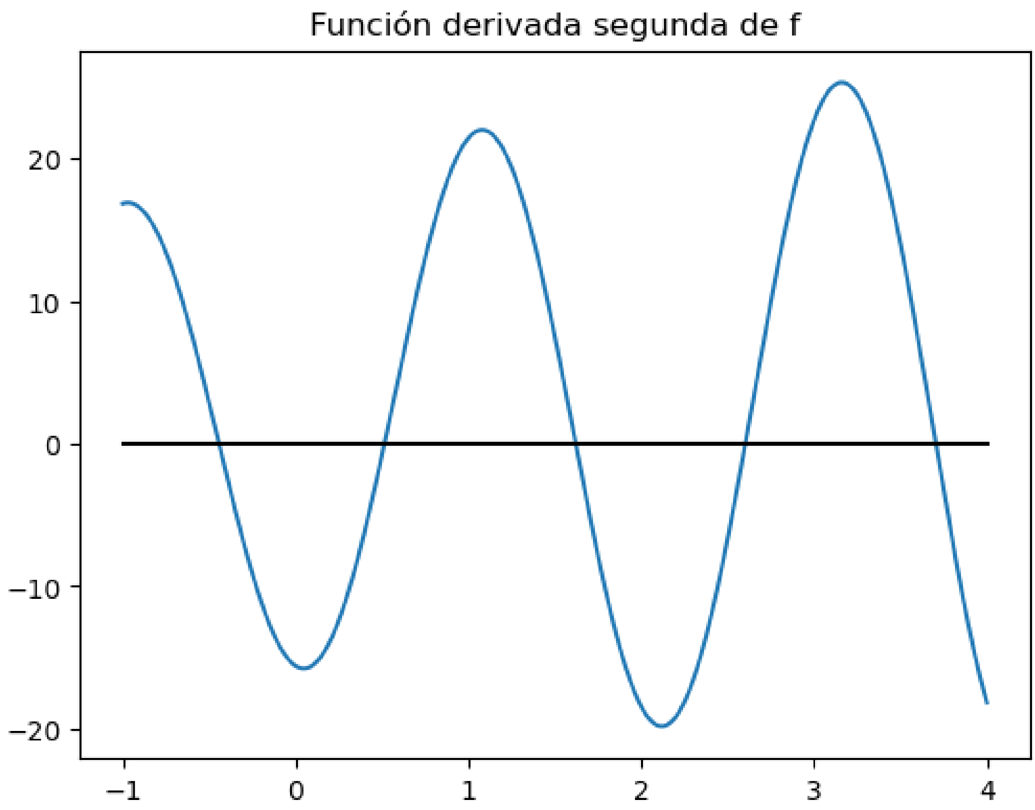
En `sympy` las funciones matemáticas suelen tener el mismo nombre pero, por ejemplo, en lugar de `np.cos(x)` , escribiremos `sym.cos(x)` (teniendo en cuenta cómo hemos importado `numpy` y `sympy` en esta práctica). Esta regla no es general porque por ejemplo `np.arctan(x)` y `sym.atan(x)`

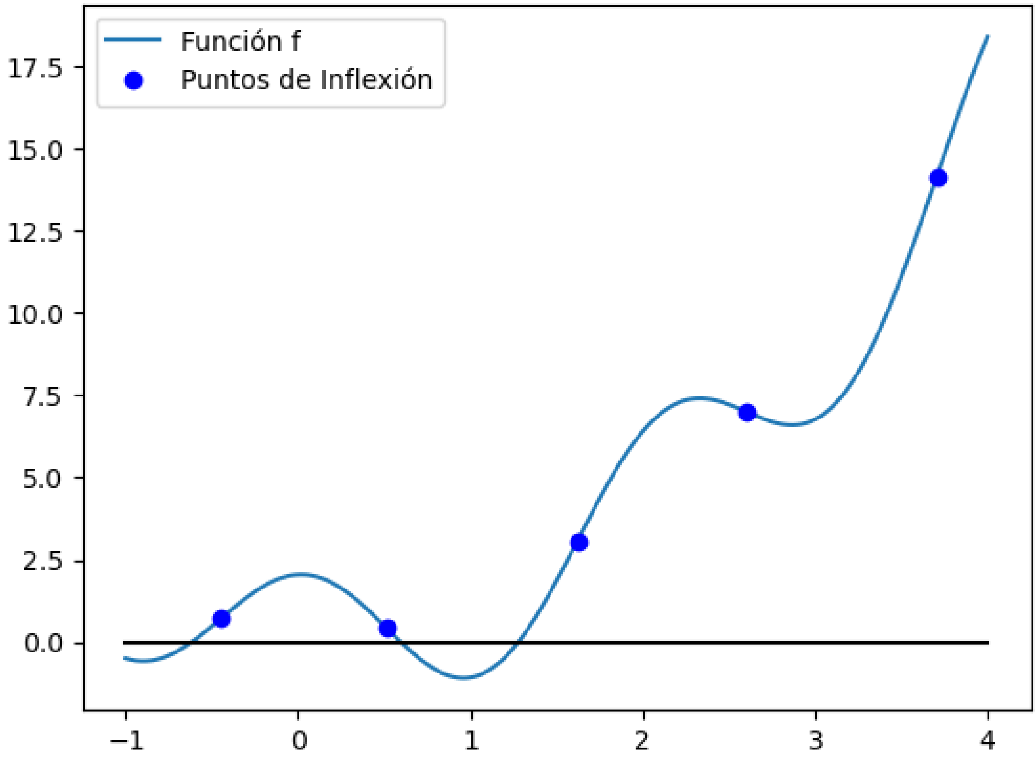
```
%run Ejercicio2
```






```
EXTREMOS
[-0.89794785  0.01824977  0.95972949  2.33033731  2.86540018]
```






```
PUNTOS DE INFLEXIÓN EN [-1,4]
[-0.44343281  0.51463064  1.62117405  2.60293871  3.70449784]
```

Método de punto fijo

En el método de punto fijo sustituimos la ecuación $f(x)=0$, por la ecuación equivalente $g(x)=x$ de modo que si α es un punto fijo de g, es decir $g(\alpha)=\alpha$, entonces también será una raíz de f, o lo que es lo mismo $f(\alpha)=0$.

Se dice que la función $g:[a,b]\to\mathbb{R}$ tiene un punto fijo $\alpha\in[a,b]$ si $g(\alpha)=\alpha$. El método del punto fijo se basa en la iteración

$$x_{k+1}=g(x_k),\quad k\geq 0,$$

donde x_0 es la aproximación inicial que debemos proporcionar.

No existe una manera única de expresar esta equivalencia. Veamos un ejemplo.

Sea

$$f(x)=x^3-2x^2+1$$

y buscamos las raíces de f tal que $x^3-2x^2+1=0$. Podemos reorganizar la ecuación como $x^3+1=2x^2$, o lo que es lo mismo

$$x^3-2x^2+1=0\quad\Longrightarrow\quad x^3+1=2x^2\quad\Longrightarrow\quad x=-\sqrt{\frac{x^3+1}{2}}$$

La solución de esta segunda ecuación, también llamada punto fijo de la función

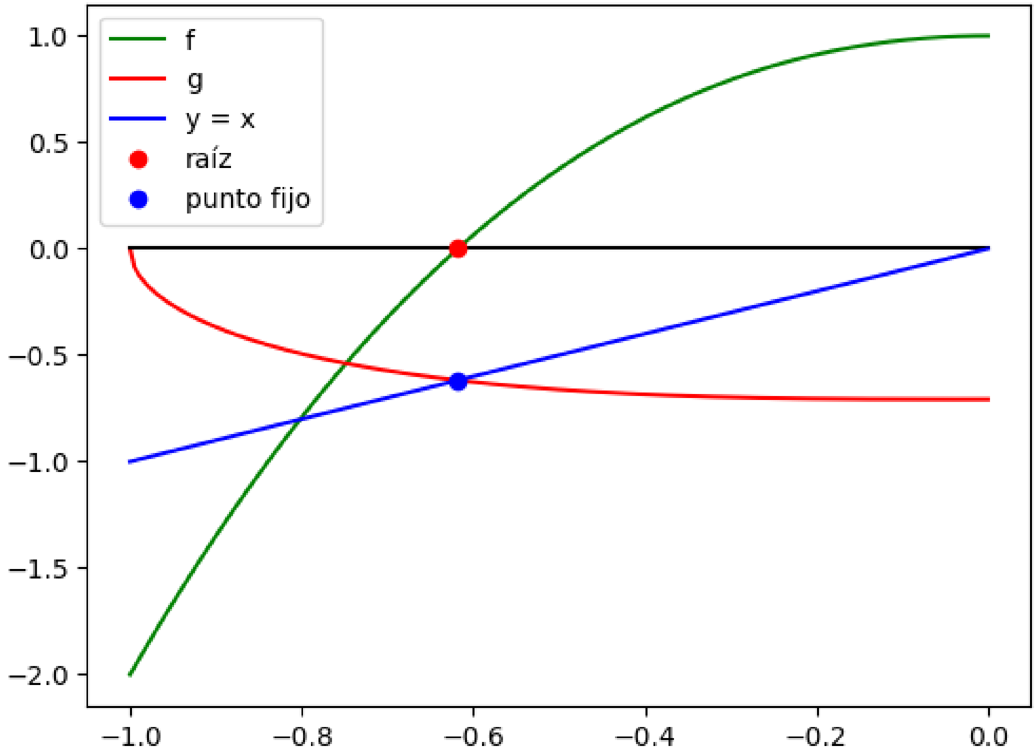
$$g(x)=-\sqrt{\frac{x^3+1}{2}}$$

será también una raíz de f.

Comprobémoslo gráficamente.

```
f = lambda x: x**3 - 2*x**2 +1
g = lambda x: - np.sqrt((x**3+1)/2)
```

```
a = -1.; b = 0;
x = np.linspace(a, b, 200)          # Vector de 200 elementos equiespaciados en (a,
b)

plt.figure()
plt.plot(x, f(x),'g-',label='f')
plt.plot(x, g(x),'r-',label='g')
plt.plot(x, 0*x,'k-')               # Eje OX
plt.plot(x, x,'b-',label='y = x')

raiz = -0.61803
plt.plot(raiz,0,'ro',label='raíz')
plt.plot(raiz,raiz,'bo',label='punto fijo')

plt.legend()
plt.show()
```



Y vemos que el punto donde $f(x) = 0$ coincide con el punto donde $x = g(x)$

Por lo tanto, el punto fijo de

$$g(x) = -\sqrt{\frac{x^3+1}{2}} \quad (1)$$

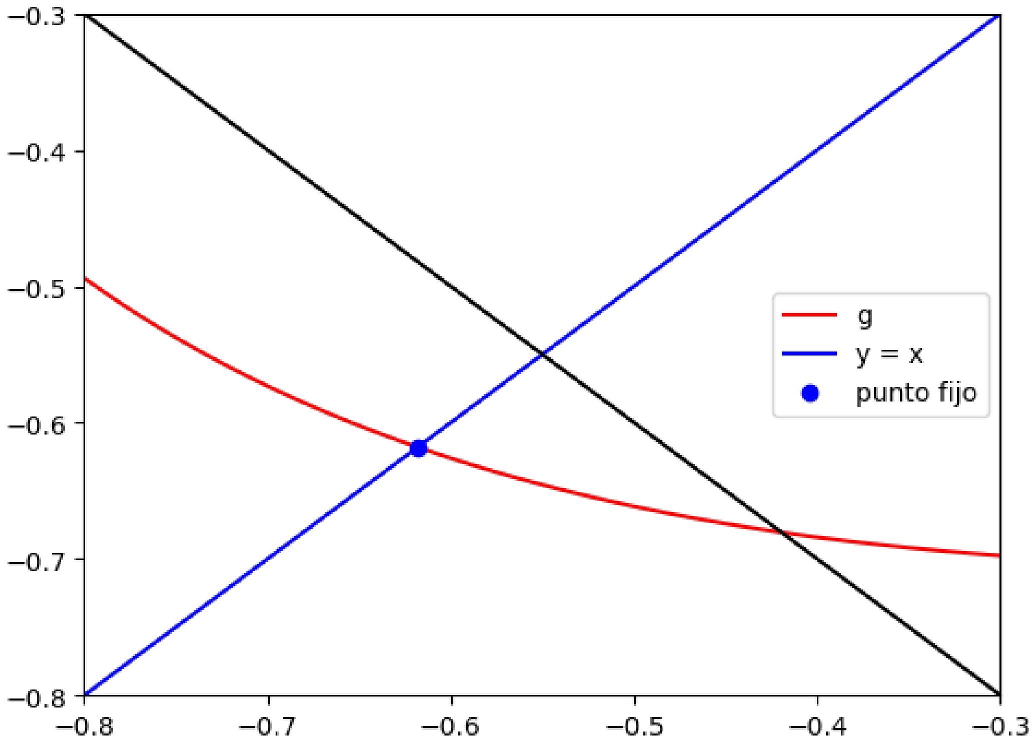
es una raíz de f.

El **Teorema de la aplicación contractiva** dice: sea g derivable definida en el intervalo $[a, b] \subset \mathbb{R}$ y $x_0 \in [a, b]$ un punto del intervalo. Supongamos que

- $x \in [a, b] \quad \Rightarrow \quad g(x) \in [a, b]$
- $|g'(x)| \le k < 1$ para todo $x \in [a, b]$

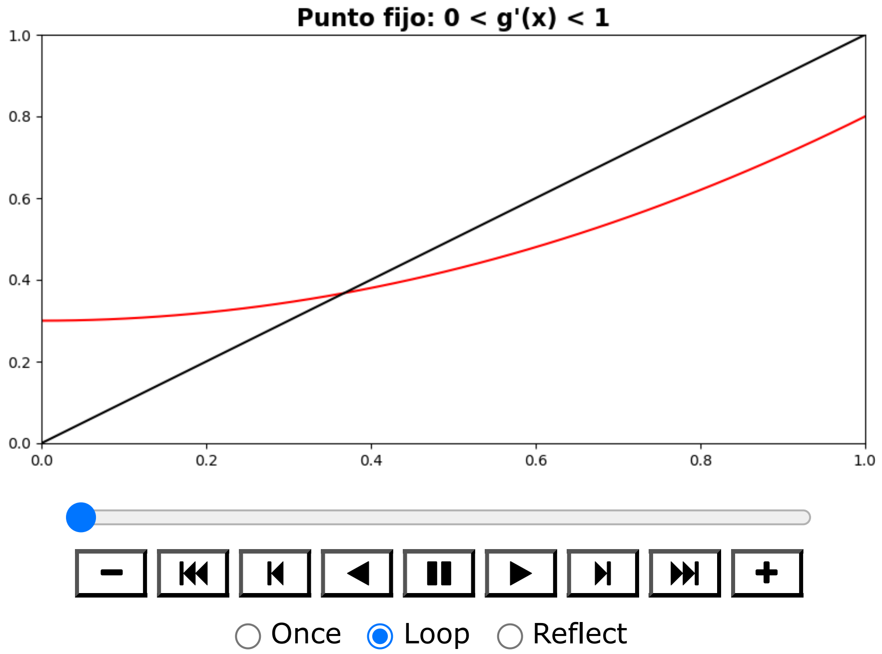
Entonces g tiene un único punto fijo $\alpha \in [a, b]$, y la sucesión x_n definida como $x_{i+1} = g(x_i)$ que tiene como punto inicial x_0 converge a α con orden al menos lineal.

```
a = -0.8; b = -0.3;
x = np.linspace(a, b)

plt.figure()
plt.plot(x, g(x),'r-', label='g')
plt.plot(x, x, 'b-',label='y = x')
plt.plot([a,b],[b,a],'k-') # Dibuja la otra diagonal

pf = -0.61803
plt.plot(pf,pf,'bo',label='punto fijo')

plt.axis([a, b, a, b])     # Gráfica en [a,b]x[a,b]
plt.legend(loc='best')
plt.show()
```

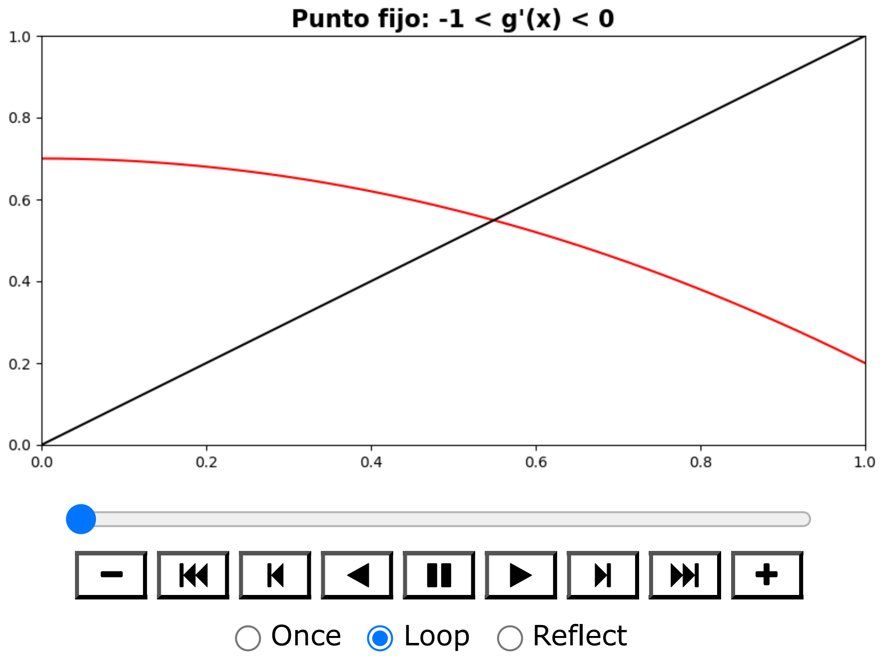


En la gráfica vemos que se cumplen las condiciones del teorema de la aplicación contractiva para la función g en el intervalo $[-0.8, -0.3]$. Si tomamos como punto inicial de la iteración de punto fijo un punto de este intervalo el algoritmo de punto fijo converge.

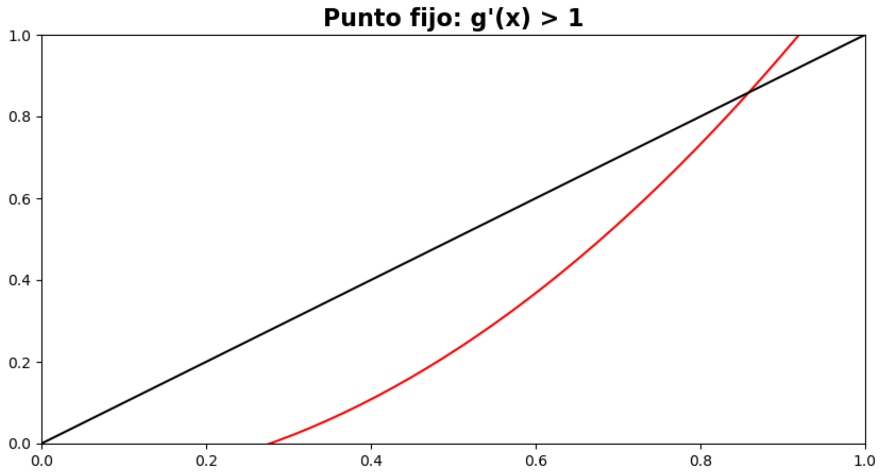
```
pfa
```


Punto fijo: 0 < g'(x) < 1
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Once Loop Reflect

```
pfb
```

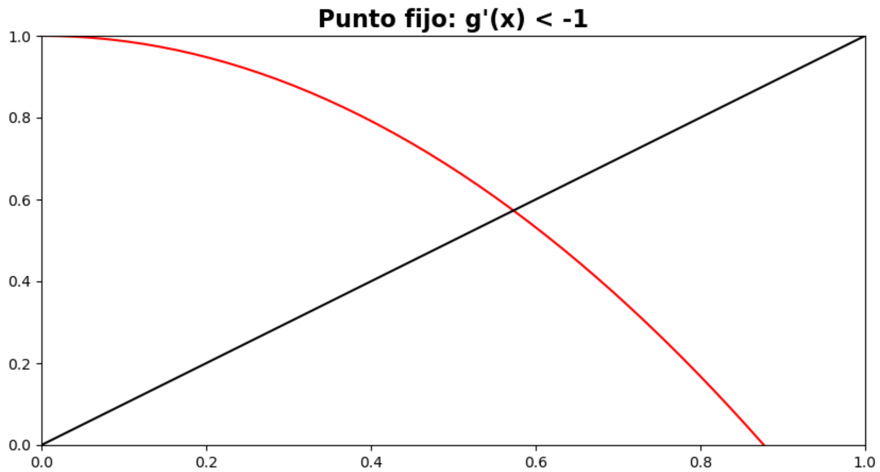

Punto fijo: -1 < g'(x) < 0
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Once Loop Reflect

```
pfc
```



Once Loop Reflect

```
pfd
```



Once Loop Reflect

Ejercicio 3

Escribir un programa que contenga una función `puntoFijo(g,x0,tol=1.e-6,maxiter=200)` que tenga como argumentos de entrada una función de iteración `g` , el punto inicial `x0` , la cota de error absoluto `tol` y el número máximo de iteraciones `maxiter` y como argumentos de salida el punto fijo y el número de iteraciones realizadas.

Utilizarlo para aproximar la raíz de la función

$$f(x) = e^{-x} - x$$

con la función de iteración

$$g(x) = e^{-x}$$

con $\mathrm{tol} = 10^{-6}$ y $\mathrm{maxiter} = 200$.

Seguir los siguientes pasos:

- Utilizando la función de la práctica anterior **busquedaIncremental** con el intervalo $[0, 1]$ encontrar un intervalo de longitud 0.1 que contenga una raíz de la función `f` .
- Usando como punto inicial x_0 el borde izquierdo del intervalo hallado, calcular el punto fijo de la función `g` . Recordar que la sucesión del método de punto fijo se define como $x_{i+1} = g(x_i)$.
- Escribir la aproximación del punto fijo y el número de iteraciones.
- Dibujar: la función de iteración `g` en rojo, la recta `y = x` en azul y el punto fijo en azul.

Repetir proceso anterior para encontrar la raíz de la función

$$f(x) = x - \cos(x)$$

Utilizando las funciones de iteración

$$g_1(x) = \cos(x) \quad g_2(x) = 2x - \cos(x) \quad g_3(x) = x - \frac{x - \cos(x)}{1 + \sin(x)} \quad g_4(x) = \frac{9x + \cos(x)}{10}$$

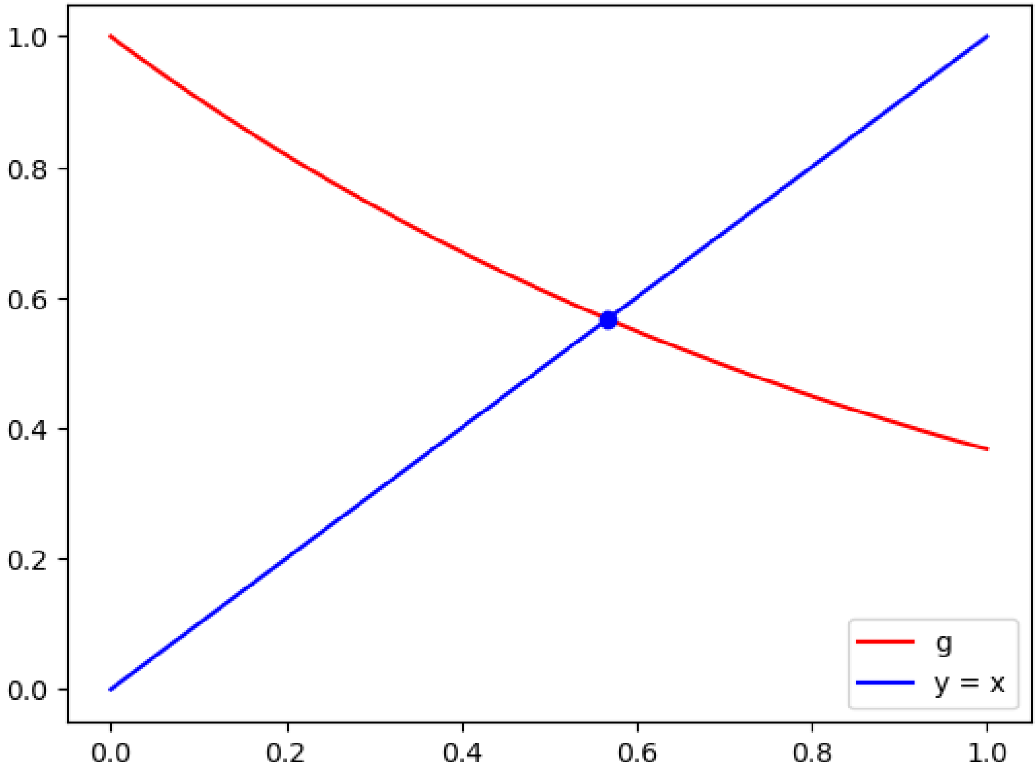
Decidir con cuales obtenemos una sucesión convergente y con cuales no. Dibujar las funciones de iteración, la recta `y = x` en azul y el punto fijo en azul.

Nota:

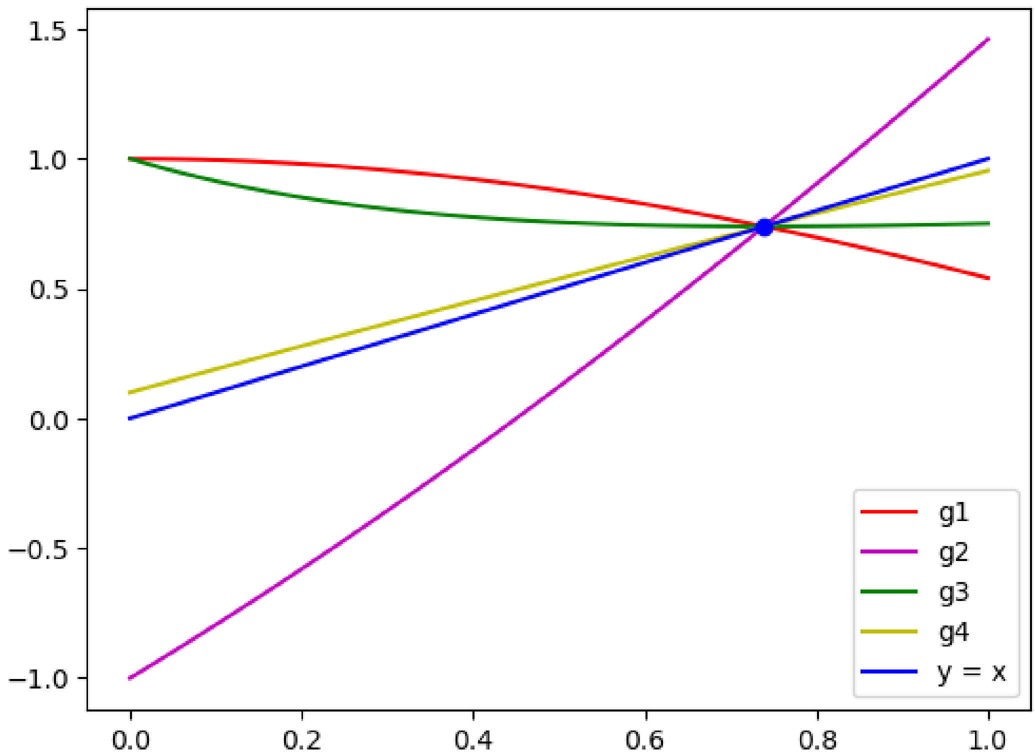
[Lista de funciones matemáticas elementales en numpy (https://www.pythonprogramming.in/numpy-elementary-mathematical-functions.html)](https://www.pythonprogramming.in/numpy-elementary-mathematical-functions.html).

```
%run Ejercicio3
```

```
Existe una raiz en  [[0.5 0.6]]

0.5671430289524634 22
```




```
Existe una raiz en  [[0.7 0.8]]

g1  0.7390848589216623 30
g2  -9.452341808684707e+58 200
g3  0.7390851332151608 3
g4  0.7390809703425896 50
```

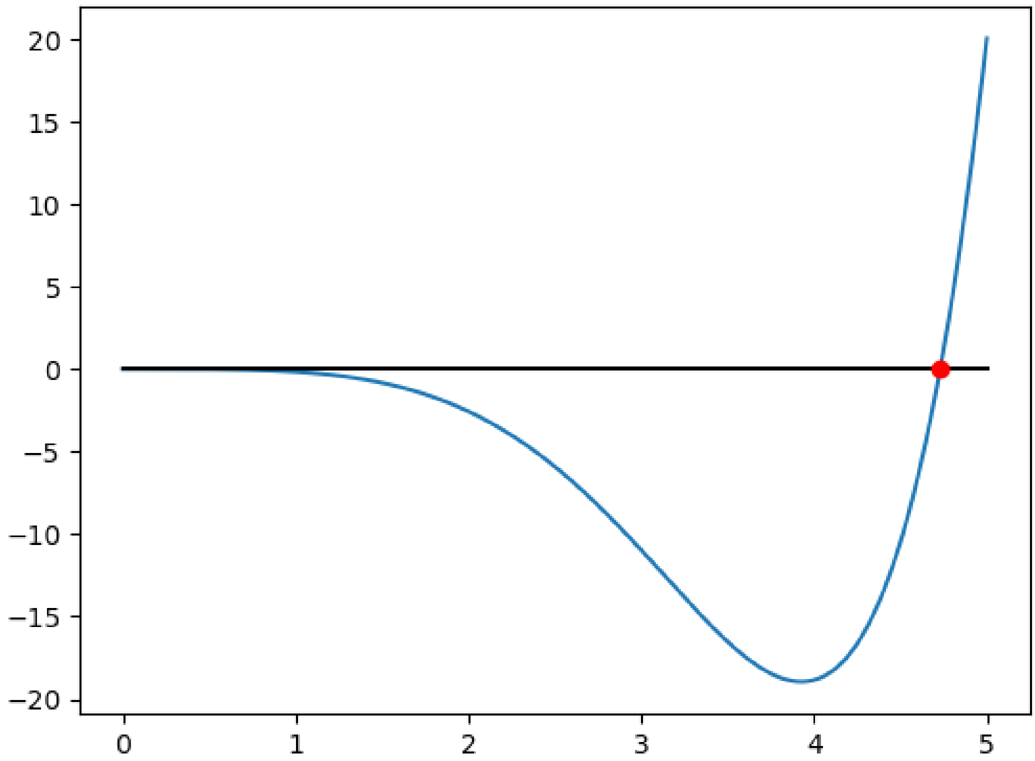


Ejercicios propuestos

Ejercicio 4

Utilizando la función `newton` del módulo `scipy.optimize` con el método de la `secante` calcular la solución positiva más cercana a cero de la ecuación $\cosh(x)\cos(x) - 1 = 0$. Empieza dibujando la función para estimar los valores iniciales a utilizar. Usar como parámetros $\text{tol} = 10^{-6}$ y $\text{maxiter} = 100$. Comprobar que es una raiz dibujándola como un punto rojo sobre la gráfica.

```
%run Ejercicio4
```

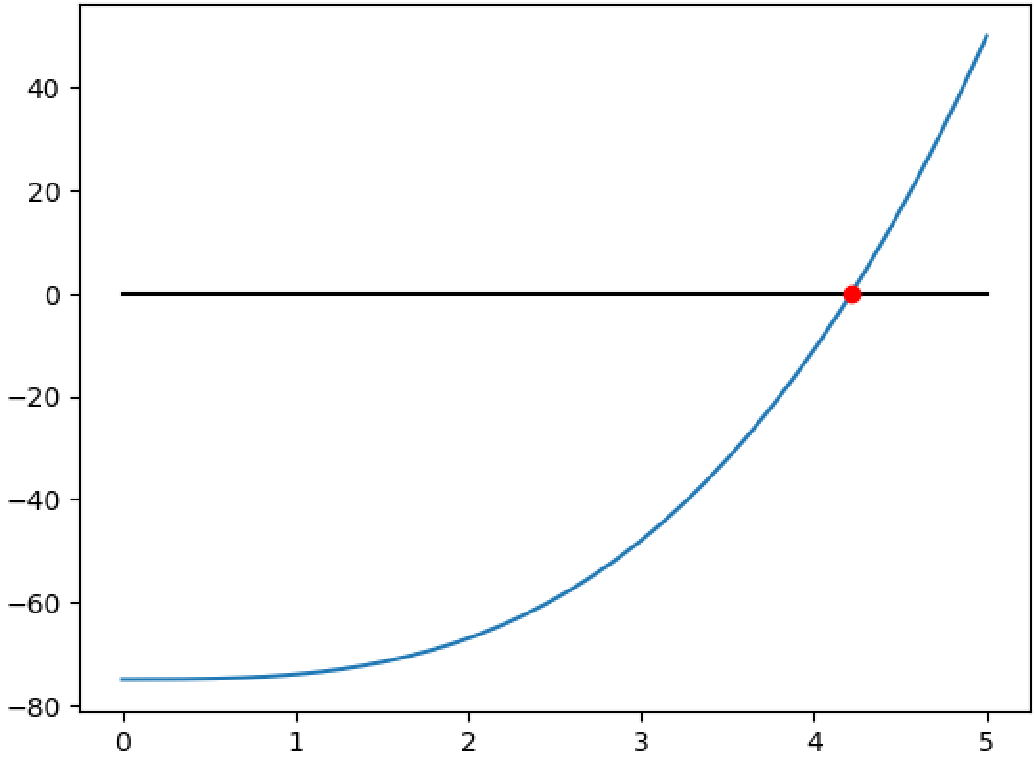



```
4.730040744862024
```

Ejercicio 5

Utilizando la función [bisect (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.bisect.html)](https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.bisect.html) del módulo `scipy.optimize` calcular $\sqrt[3]{75}$. Dibújala para estimar el intervalo inicial a utilizar. Usar como parámetros $\text{tol} = 10^{-6}$ y $\text{maxiter} = 100$. Comprobar que es una raiz dibujándola como un punto rojo sobre la gráfica.

```
%run Ejercicio5
```




```
4.217163326507944
```

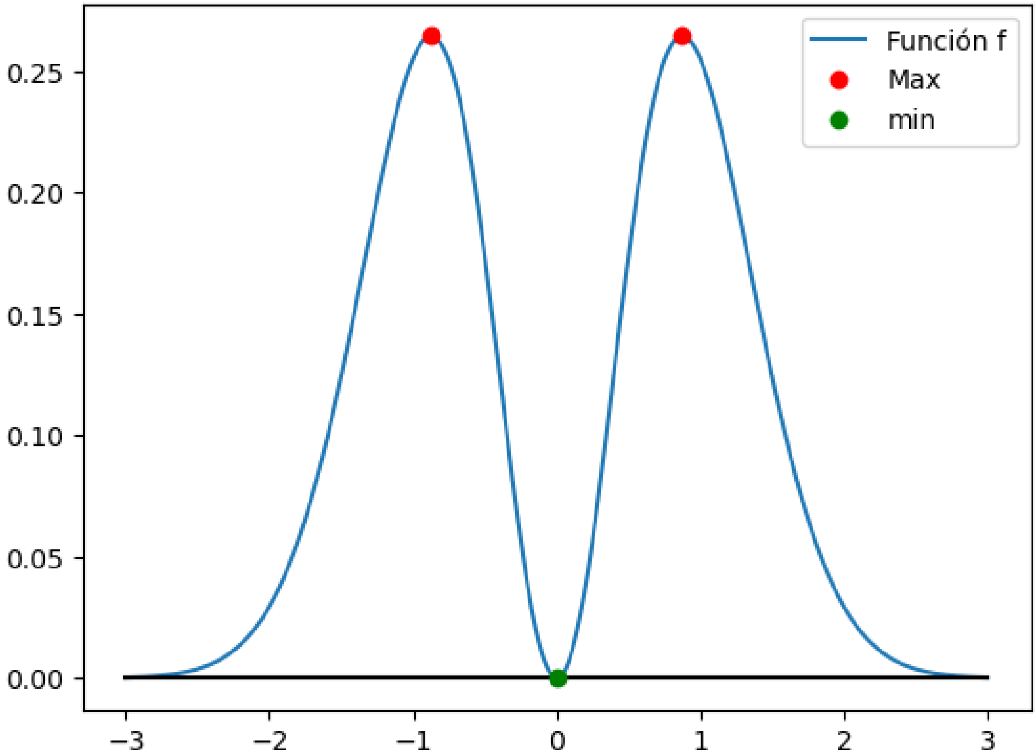
Ejercicio 6

Dada la función:

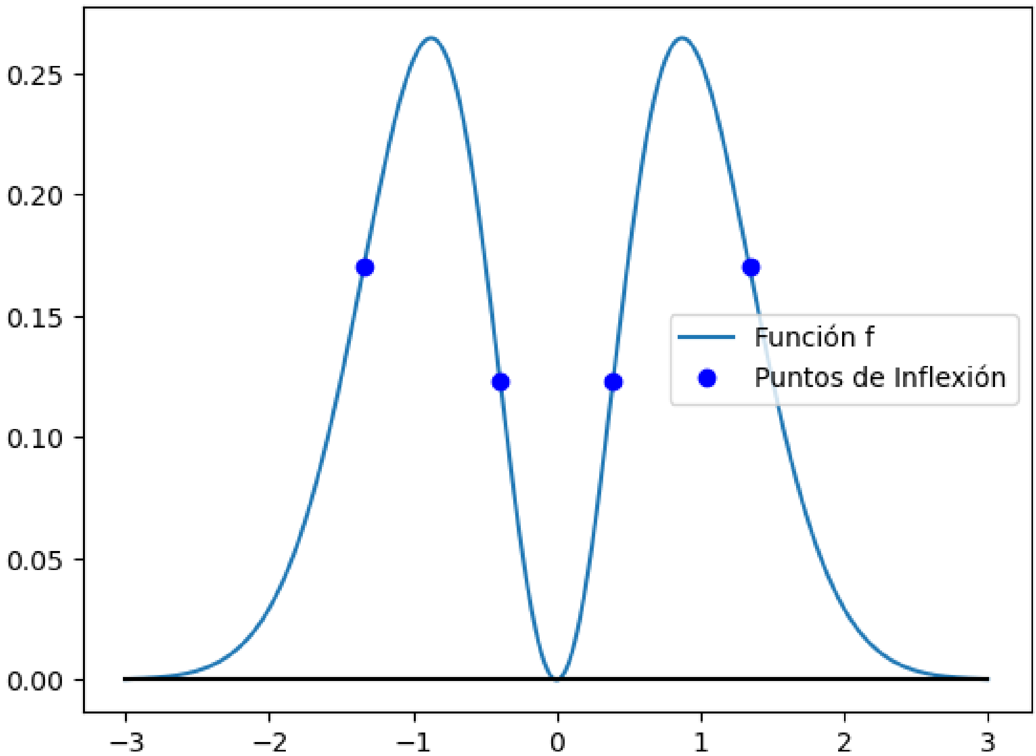
$$f(x) = e^{-x^2} \ln(x^2 + 1)$$

Calcular los máximos y mínimos y los puntos de inflexión de f y dibujarlos sobre la gráfica como en el **Ejercicio 2**. Usar como parámetros $\mathrm{tol} = 10^{-6}$ y $\mathrm{maxiter} = 100$.

```
%run Ejercicio6
```




```
Extremos
 [-0.87362626  0.          0.87362626]
```




```
Puntos de inflexión
 [-1.34109575 -0.39300931  0.39300931  1.34109575]
```

NOTA: Calcular la derivada de una función

Para calcular derivadas de forma simbólica podemos utilizar el módulo sympy.

```
import sympy as sym
```

Comenzamos declarando una variable simbólica y la función

```
x = sym.Symbol('x', real=True)
```

Declaramos la función simbólica y podemos calcular sus derivadas

```
f_sim   = sym.cos(x)*(x**3+1)/(x**2+1)
df_sim  = sym.diff(f_sim,x)
d2f_sim = sym.diff(df_sim,x)
```

Podemos ver qué forma tienen

```
print(df_sim)
```

```
3*x**2*cos(x)/(x**2 + 1) - 2*x*(x**3 + 1)*cos(x)/(x**2 + 1)**2 - (x*
*3 + 1)*sin(x)/(x**2 + 1)
```

Para usarlas como funciones numéricas las podemos *lambificar*.

```
f   = sym.lambdify([x], f_sim,'numpy')
df  = sym.lambdify([x], df_sim,'numpy')
d2f = sym.lambdify([x], d2f_sim,'numpy')
```

Y ya podemos, por ejemplo, dibujarlas con `plot`

```
x = np.linspace(-3,3,100)
plt.plot(x,f(x),x,df(x),x,d2f(x))
plt.legend(['f','df','d2f'])
plt.show()
```