# Resolución de Ecuaciones (Matemáticas 2. GII/I2ADE)

## Indice



## Cargar librerías necesarias

Recordar que la carga de librerías debe ser hecho solamente una vez y no en todas las celdas de código.

In [2]:
```
import sympy
from sympy.abc import x, y, z
import matplotlib.pyplot as plt
import numpy as np
from sympy.plotting import plot as sypltot  # Liberia para las gráficas
```

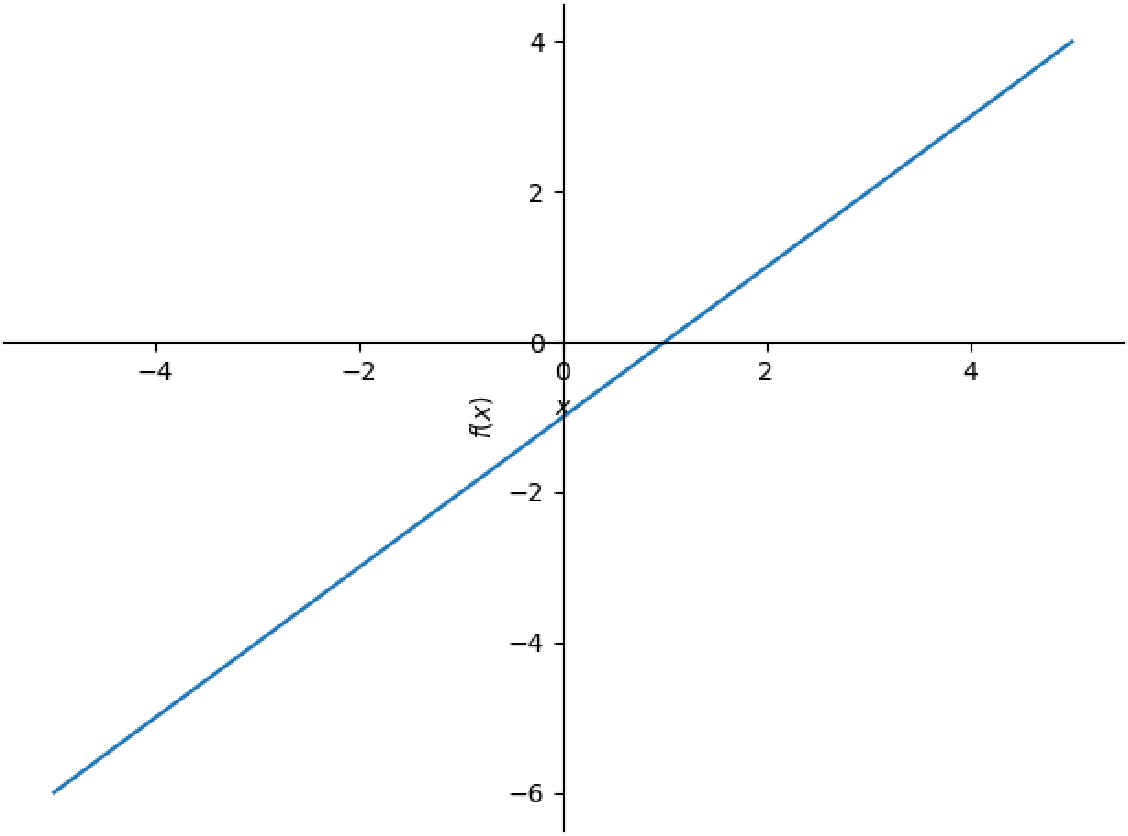
## Teorema de Bolzano

Una forma muy simples de comprobar el teorema de Bolzano es multiplicar f(a) y f(b) y si su valor es menor de 0 entonces se confirma la conservación de signo. Esto porque el teorema de Bolzano exige que la función en un punto sea positiva y en el otro negativa, por lo tanto, siempre que se multiplica un valor positivo por un negativo el resultado es siempre negativo. En el caso que el resultado de la multiplicación sea negativo es porque se cumple Bolzano.

In [3]:
```
def bolzano(funcion, puntoA, puntoB):
    """Funciona con funciones del tipo sympy con "x" como variable"""
    if (funcion.subs(x, puntoA) * funcion.subs(x, puntoB)) < 0:
        return True
    return False


def bolzano_numerico(funcion, puntoA, puntoB):
    """Funciona con funciones del tipo numéricas ("lambdas" o "def").
    En este caso no es necesario tener en cuenta el nombre de la variabl
e"""
    if (funcion(puntoA) * funcion(puntoB)) < 0:
        return True
    return False
```

Usemos una función básica para confirmar el teorema de Bolzano.

In [4]:
```
# Función a usar en el ejemplo
f = x - 1
# Verifiquemos visualmente
sypltot(f, (x, -5, 5))
```




Out[4]: `<sympy.plotting.plot.Plot at 0x22eafd56a10>`

Sabemos entonces que $f(x)$ es:

- negativa cuando $x < 1$
- positiva cuando $x > 1$

In [5]:
```
# Prueba 1
bolzano(f, 0, 2)
```
Out[5]:
```
True
```

In [6]:
```
# Prueba 2
bolzano(f, 2, 4)
```
Out[6]:
```
False
```

In [7]:
```
# Prueba 3
bolzano(f, 2, 0)
```
Out[7]:
```
True
```

In [8]:
```
# Prueba 4
bolzano(f, -1, 0)
```
Out[8]:
```
False
```

In [9]:
```
# Probemos con la versión numérica
def f_n(x):
    return x - 1


bolzano_numerico(f_n, 0, 2)
```
Out[9]:
```
True
```

Vemos entonces que la implementación está a funcionar correctamente.

## Método de la Bisección

Este es el método más simples y convergente (al confirmar Bolzano en cada paso) de buscar una raíz. Cada iteración se divide por la mitad la distancia entre A y B y se cambia A o B por C (el que tenga el mismo signo) hasta encontrar el 0.

## Implementación del algoritmo de clase (sin visualización)

In [10]:
```
# Abordaje numérica
def bisecion(f, puntoA, puntoB, epsilon=0.0001, delta=0.0001, n=100):
    i = 0  # Empezamos a contar desde 0 por estándar

    # La diferencia entre los puntos tiene que ser menor que delta (que no sea tan pequeña que sea difícil computar/tenga precisión necesaria)
    h = abs(puntoB - puntoA)
    # Primera "división" del espacio entre A y B por 2
    c = (puntoA + puntoB) / 2

    # En Python no hay do...while. Por lo tanto hay que invertir pseudocódigo visto en clase. Aquí solo avanza si TODAS las variables son verdaderas.
    while abs(f(c)) >= epsilon and h >= delta and i < n:
        # Forma rápida de confirmar Bolzano para sustituir. En el caso que el punto A y C tengan signos distintos de sustituí B por C (se aproxima el punto B al A).
        if bolzano_numerico(f, puntoA, c):
            puntoB = c
        else:
            # Caso no haya Bolzano entre A y C se procede a cambiar A por C (se aproxima el punto A al B)
            puntoA = c
        h = abs(puntoB - puntoA)
        # Se vuelve a "dividir por la mitad" la distancia entre los puntos
        c = (puntoA + puntoB) / 2
        i += 1  # Se hace avanzar el contador de bucles máximo
    # Devuelve una tupla con de valor de c, f(c) e la cantidad de bucles que ha hecho.
    return (
        c,
        f(c),
        i,
    )
```

In [11]:
```
bisecion(f_n, 0, 10)
```

Out[11]:
```
(1.00006103515625, 6.103515625e-05, 14)
```

Este proceso puede ser difícil de entender. Veamos visualmente, paso por paso, qué ha pasado con los puntos en cada iteración.

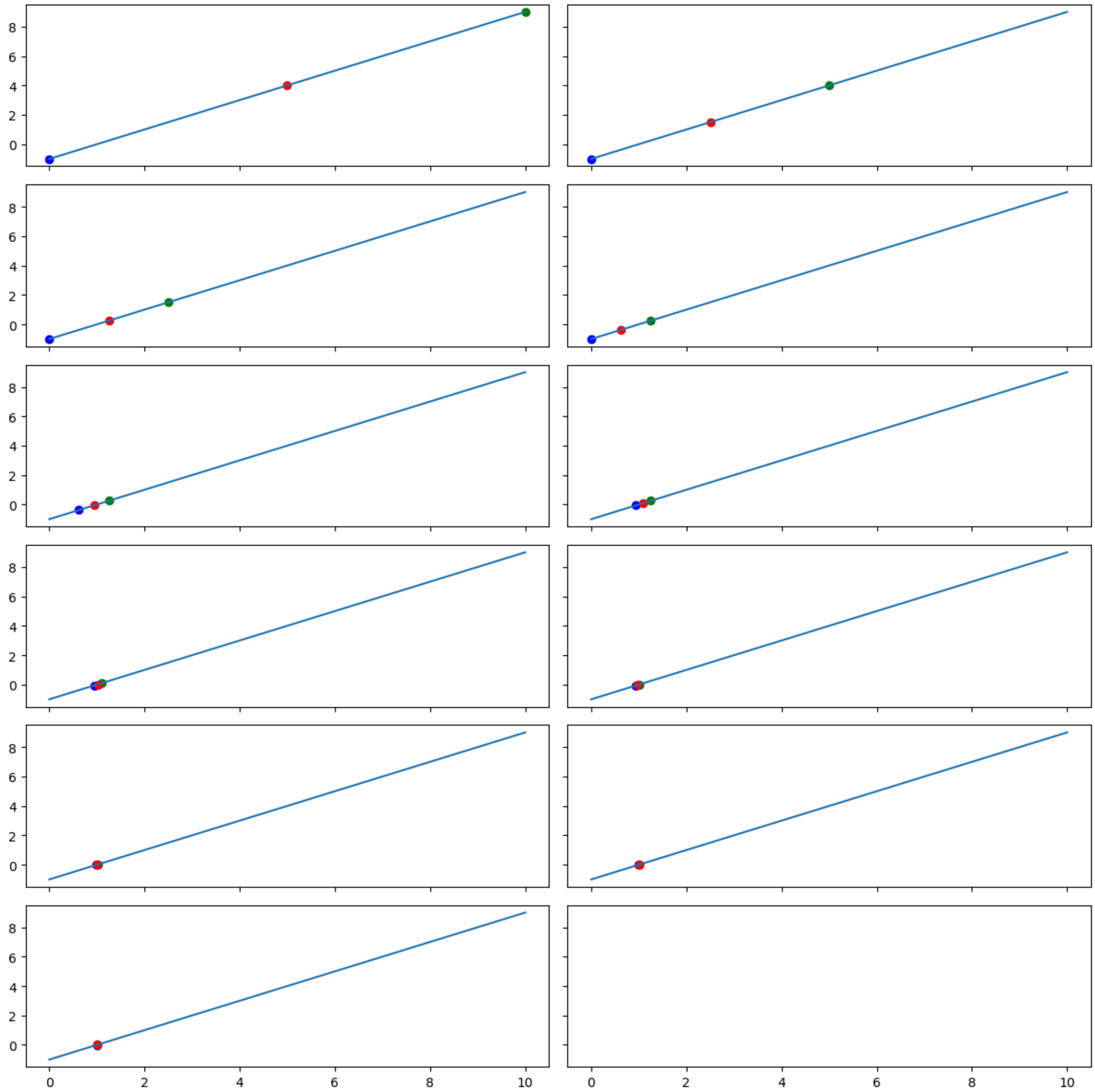
## Implementación del algoritmo de clase (con visualización)

In [12]:
```
# Abordaje numérica con visualización
import math


def bisecion_vis(f, puntoA, puntoB, epsilon=0.001, delta=0.001, n=100):
    i = 0
    h = abs(puntoB - puntoA)
    c = (puntoA + puntoB) / 2
    calculos_intermedios = [
        (puntoA, puntoB, c)
    ]  # Para guardar los cálculos intermedios

    exis = np.linspace(
        puntoA, puntoB
    )  # Generamos un vector de 50 valores equidistantes entre puntoA y pun
toB. Esto nos servirá para representar visualmente la función en la gráfica

    # Los mismos cálculos que la función vista anteriormente pero con el añ
adido del vector de los cálculos intermedios
    while abs(f(c)) >= epsilon and h >= delta and i < n:
        if bolzano_numerico(f, puntoA, c):
            puntoB = c
        else:
            puntoA = c
        h = abs(puntoB - puntoA)
        c = (puntoA + puntoB) / 2
        i += 1
        calculos_intermedios.append(
            (puntoA, puntoB, c)
        )  # Guardamos los cálculos intermedios

    # Empieza la representación visual

    # Se crea un subplot para que cada bucle tenga su gráfica
    fig, ax = plt.subplots(
        math.ceil(
            len(calculos_intermedios) / 2
        ),  # Permite que la función genere cuantas sub-gráficas sean neces
arias para representar visualmente la función, en dos columnas (por eso se
divide por 2)
        2,
        figsize=(12, 12),
        sharex=True,
        sharey=True,
    )
    pos = (
        ax.flat
    )  # Devuelve la referencia a las gráficas (que en este momento están
"vacías") para que puedan ser iteradas y inicializadas

    for ci in calculos_intermedios:
        ntxp = next(
            pos
        )  # Obtenemos la gráfica a ser usada. El mecanismo usado es una it
eración y el sistema nos va dando la posición automáticamente
        ntxp.scatter(ci[0], f(ci[0]), c=["b"])  # Dibujamos punto A a azul
```

```
("b")
        ntxp.scatter(ci[1], f(ci[1]), c=["g"])  # Dibujamos punto B a verde
("g")
        ntxp.scatter(ci[2], f(ci[2]), c=["r"])  # Dibujamos punto C
        # exis = np.linspace(ci[0], ci[1])
        ntxp.plot(exis, f(exis))  # Dibujamos la función
    fig.tight_layout()  # Para que sea visualmente homogéneo

    return (c, f(c), i)


bisecion_vis(f_n, 0, 10)
```

Out[12]:

```
(1.0009765625, 0.0009765625, 10)
```




Podemos entonces observar que a cada paso el algoritmo está escogiendo el punto medio entre A y B y cambiando uno de estos por ese punto de manera a preservar el teorema de Bolzano.

Puedes ver esto con más precisión si comentas el trozo `sharex=True, sharey=True` y descomentas `exis = np.linspace(ci[0], ci[1])`. Observa el eje de las $x$ y $y$ y verás como se aproxima del 1 a cada paso.

No obstante, la librería **Scipy** nos ofrece una función que implementa la bisección.

### Método de la librería Scipy

In [13]:
```
from scipy.optimize import bisect

bisect(f_n, 0, 10)  # nos devuelve el valor del X
```

Out[13]:
```
1.0000000000002274
```

## Método del Punto Fijo

In [14]:
```
# punto fijo numérico
def punto_fijo_numerico(g, x, epsilon=0.0001, delta=0.0001, n=100):
    # Inicializamos las variables
    i = 0
    h = delta + 1
    # El cálculo se hace en la ejecución de cada bucle y se genera el valor
del nuevo "x" al final del bucle.
    while abs(g(x)) > epsilon and h > delta and i < n:
        # Calculamos el error
        h = abs(g(x) - x)
        # Imprimimos los cálculos hechos (como en teoría)
        print(
            f"Iteración: {i}",
            f"Expresión: g({x:.14f})",
            f"Resultado: {g(x):.9f}",
            f"Error: {h:.9f}",
            sep="\t",
        )
        # Preparamos las variables para el próximo bucle
        x = g(x)
        i += 1
    # Se devuelve el "x" y la cantidad de iteraciones
    return (x, i - 1)
```

In [15]:
```
# Probemos con el ejemplo de la clase teórica
punto_fijo_numerico(np.cos, 0, delta=0.01)
```

```
Iteración: 0    Expresión: g(0.00000000000000)  Resultado: 1.000000000   Err
or: 1.000000000
Iteración: 1    Expresión: g(1.00000000000000)  Resultado: 0.540302306   Err
or: 0.459697694
Iteración: 2    Expresión: g(0.54030230586814)  Resultado: 0.857553216   Err
or: 0.317250910
Iteración: 3    Expresión: g(0.85755321584639)  Resultado: 0.654289790   Err
or: 0.203263425
Iteración: 4    Expresión: g(0.65428979049778)  Resultado: 0.793480359   Err
or: 0.139190568
Iteración: 5    Expresión: g(0.79348035874257)  Resultado: 0.701368774   Err
or: 0.092111585
Iteración: 6    Expresión: g(0.70136877362276)  Resultado: 0.763959683   Err
or: 0.062590909
Iteración: 7    Expresión: g(0.76395968290065)  Resultado: 0.722102425   Err
or: 0.041857258
Iteración: 8    Expresión: g(0.72210242502671)  Resultado: 0.750417762   Err
or: 0.028315337
Iteración: 9    Expresión: g(0.75041776176376)  Resultado: 0.731404042   Err
or: 0.019013719
Iteración: 10   Expresión: g(0.73140404242251)  Resultado: 0.744237355   Err
or: 0.012833312
Iteración: 11   Expresión: g(0.74423735490056)  Resultado: 0.735604740   Err
or: 0.008632614
```

Out[15]:
```
(0.7356047404363473, 11)
```

In [16]:
```python
# Otro ejemplo de la clase teórica: f(x): e**(-x)-x
punto_fijo_numerico(lambda x: np.exp(-x), 1, delta=0.001)
```

```
Iteración: 0    Expresión: g(1.00000000000000)  Resultado: 0.367879441   Err
or: 0.632120559
Iteración: 1    Expresión: g(0.36787944117144)  Resultado: 0.692200628   Err
or: 0.324321186
Iteración: 2    Expresión: g(0.69220062755535)  Resultado: 0.500473501   Err
or: 0.191727127
Iteración: 3    Expresión: g(0.50047350056364)  Resultado: 0.606243535   Err
or: 0.105770035
Iteración: 4    Expresión: g(0.60624353508560)  Resultado: 0.545395786   Err
or: 0.060847749
Iteración: 5    Expresión: g(0.54539578597503)  Resultado: 0.579612336   Err
or: 0.034216550
Iteración: 6    Expresión: g(0.57961233550338)  Resultado: 0.560115461   Err
or: 0.019496874
Iteración: 7    Expresión: g(0.56011546136109)  Resultado: 0.571143115   Err
or: 0.011027654
Iteración: 8    Expresión: g(0.57114311508018)  Resultado: 0.564879347   Err
or: 0.006263768
Iteración: 9    Expresión: g(0.56487934739105)  Resultado: 0.568428725   Err
or: 0.003549378
Iteración: 10   Expresión: g(0.56842872502906)  Resultado: 0.566414733   Err
or: 0.002013992
Iteración: 11   Expresión: g(0.56641473314688)  Resultado: 0.567556637   Err
or: 0.001141904
Iteración: 12   Expresión: g(0.56755663732828)  Resultado: 0.566908912   Err
or: 0.000647725
```

Out[16]:
```
(0.5669089119214953, 12)
```

# Método de la Secante

In [105]:
```
# Abordaje numérica
def secante(f, puntoA, puntoB, epsilon=0.0001, delta=0.0001, n=100):
    i = 0  # Empezamos a contar desde 0
    h = (f(puntoA) * (puntoB - puntoA)) / (
        f(puntoB) - f(puntoA)
    )  # Para el cálculo del límite de tolerancia
    c = puntoA - h  # Punto donde y = 0
    while (
        abs(f(c)) > epsilon and abs(h) > delta and i < n
    ):  # En Python no hay do...while. Por lo tanto hay que invertir pseudo
código visto en clase. Aquí solo avanza si TODAS las variables son verdader
as.
        if abs(f(puntoA)) > abs(f(puntoB)):
            # Se intercambia para que tenga siempre una pendiente y sea cad
a vez más cerca de y = 0
            puntoA, puntoB = puntoB, puntoA
        h = (f(puntoA) * (puntoB - puntoA)) / (
            f(puntoB) - f(puntoA)
        )  # Para el cálculo del límite de tolerancia y el punto c
        c = puntoA - h  # Calculamos nuevo punto fijo
        print(
            f"i: {i} a: {puntoA:.3f} b: {puntoB:.3f} c: {c:.3f} h: {h:.3f}
f(a): {f(puntoA):.3f} f(b): {f(puntoB):.3f} f(c): {f(c):.3f}"
        )  # Representamos los resultados de los cálculos
        puntoB = c
        i += 1  # Se hace avanzar el contador de bucles
    return (
        c,
        f(c),
        i - 1,
    )  # Se devuelve una tupla con de valor de X, f(X) e la cantidad de buc
les que ha hecho.
```

In [102]:
```
xis, fn, iter = secante(lambda x: x**2 - 4, -1, 4, epsilon=0.05, delta=0.0
1, n=6)
print(f"He encontrado una raíz en x = {xis} en {iter} iteraciones")
```

```
i: 0 a: -1.000 b: 4.000 c: 0.000 h: -1.000 f(a): -3.000 f(b): 12.000 f(c):
-4.000
i: 1 a: -1.000 b: 0.000 c: -4.000 h: 3.000 f(a): -3.000 f(b): -4.000 f(c):
12.000
i: 2 a: -1.000 b: -4.000 c: -1.600 h: 0.600 f(a): -3.000 f(b): 12.000 f(c):
-1.440
i: 3 a: -1.600 b: -1.000 c: -2.154 h: 0.554 f(a): -1.440 f(b): -3.000 f(c):
0.639
i: 4 a: -2.154 b: -1.600 c: -1.984 h: -0.170 f(a): 0.639 f(b): -1.440 f(c):
-0.065
i: 5 a: -1.984 b: -2.154 c: -1.999 h: 0.016 f(a): -0.065 f(b): 0.639 f(c):
-0.002
He encontrado una raíz en x = -1.9993904297470284 en 5 iteraciones
```

### De la librería Scipy

In [19]:
```
from scipy.optimize import root_scalar

root_scalar(
    lambda x: x**2 - 4, x0=-1, x1=4, method="secant"
)  # El valor de la raíz está en "root"
```

Out[19]:
```
      converged: True
           flag: converged
 function_calls: 11
     iterations: 10
           root: 2.0
```

## Método de Newton

In [20]:
```
from scipy.misc import derivative  # Para calcular la derivada en un punto


# Abordaje numérica
def Newton(f, puntoA, epsilon=0.0001, delta=0.0001, n=100):
    i = 0
    # Calculamos valores iniciales para poder entrar en el bucle. Esto corr
esponde a la iteración 0.
    h = f(puntoA) / derivative(f, puntoA)
    c = puntoA - h

    while abs(f(c)) > epsilon and abs(h) > delta and i < n:
        h = f(puntoA) / derivative(f, puntoA)
        c = puntoA - h
        print(
            f"i: {i} a: {puntoA:.3f} c: {c:.3f} h: {h:.3f} f(a): {f(punto
A):.3f} f(c): {f(c):.3f}"
        )
        # Actualizamos para el proximo bucle
        puntoA = c
        i += 1
    return (c,i-1)
```

In [21]:
```
Newton(lambda x: x**2 + 4 * x - 5, -1, 0.05, 0.01, 6)
```

```
i: 0 a: -1.000 c: 3.000 h: -4.000 f(a): -8.000 f(c): 16.000
i: 1 a: 3.000 c: 1.400 h: 1.600 f(a): 16.000 f(c): 2.560
i: 2 a: 1.400 c: 1.024 h: 0.376 f(a): 2.560 f(c): 0.142
i: 3 a: 1.024 c: 1.000 h: 0.023 f(a): 0.142 f(c): 0.001

C:\Users\padil\AppData\Local\Temp\ipykernel_39716\2300854324.py:8: Deprecat
ionWarning: scipy.misc.derivative is deprecated in SciPy v1.10.0; and will
be completely removed in SciPy v1.12.0. You may consider using findiff: htt
ps://github.com/maroba/findiff or numdifftools: https://github.com/pbrod/nu
mdifftools
  h = f(puntoA) / derivative(f, puntoA)
C:\Users\padil\AppData\Local\Temp\ipykernel_39716\2300854324.py:12: Depreca
tionWarning: scipy.misc.derivative is deprecated in SciPy v1.10.0; and will
be completely removed in SciPy v1.12.0. You may consider using findiff: htt
ps://github.com/maroba/findiff or numdifftools: https://github.com/pbrod/nu
mdifftools
  h = f(puntoA) / derivative(f, puntoA)
```

Out[21]:
```
(1.0000915541313802, 3)
```

### De la librería Scipy

In [22]:
```
from scipy.optimize import newton

# Aunque scipy tiene una función para calcular la derivada en un punto, no
tiene ninguna producir una derivada simbólica
# por lo tanto es necesario incluir manualmente la derivada en el argumento
"fprime"
newton(lambda x: x**2 + 4 * x - 5, -1, fprime=lambda x: 2 * x + 4)
```

Out[22]:
```
1.0
```

## Descenso por gradiente - Lineal (NUMPY)

En esta implementación usaremos el poder de Numpy para simplificar el cálculo de funciones con múltiples variables.

Tal como se observa de la teoría, el algoritmo del descenso por gradiente se puede resumir a operaciones vectoriales en un bucle.

Abajo está la implementación junto con ejemplos para funciones con una y dos variables, y se puede observar que la función es siempre la misma, solo cambian los argumentos de entrada, que pasan de numéricos a vectoriales.

In [23]:
```
from numpy.linalg import norm  # Para calcular la norma


# El argumento "f" tiene que ser ya la derivada de la función original
def descenso_gradiente(f, a, gamma=0.2, epsilon=0.001, tolerancia=1e-06, n=
50):
    # Se hace un primer calculo para que pueda entrar dentro del "while"
    i = 0
    print(f"a{i} = {a}")
    diff = a - gamma * f(a)
    norma = norm(a - diff)
    a = diff
    print(f"a{i+1} = {a}")
    print(f"||a{i+1}-a{i}|| = {norma}")
    print()
    # Control de ejecución.
    # En este caso se controla también si el valor (a) está cerca de 0
    while i <= n and np.all(np.abs(a) > tolerancia) and norma > epsilon:
        # Calculo del nuevo punto
        diff = a - np.multiply(gamma, f(a))
        # Calculo de la norma (cantidad de error)
        norma = norm(a - diff)
        # Actualizar el punto para la nueva iteración
        i += 1
        print(f"a{i} = {a}")
        a = diff
        print(f"a{i+1} = {a}")
        print(f"||a{i+1}-a{i}|| = {norma}")
        print()

    return a
```

In [24]:
```
# Probemos hacer un descenso por gradiente de la función x² empezado en 10
descenso_gradiente(lambda v: 2 * v, 10.0)
```

```
a0 = 10.0
a1 = 6.0
||a1-a0|| = 4.0

a1 = 6.0
a2 = 3.5999999999999996
||a2-a1|| = 2.4000000000000004

a2 = 3.5999999999999996
a3 = 2.1599999999999997
||a3-a2|| = 1.44

a3 = 2.1599999999999997
a4 = 1.2959999999999998
||a4-a3|| = 0.8639999999999999

a4 = 1.2959999999999998
a5 = 0.7775999999999998
||a5-a4|| = 0.5184

a5 = 0.7775999999999998
a6 = 0.46655999999999986
||a6-a5|| = 0.31104

a6 = 0.46655999999999986
a7 = 0.2799359999999999
||a7-a6|| = 0.18662399999999996

a7 = 0.2799359999999999
a8 = 0.16796159999999993
||a8-a7|| = 0.11197439999999997

a8 = 0.16796159999999993
a9 = 0.10077695999999996
||a9-a8|| = 0.06718463999999998

a9 = 0.10077695999999996
a10 = 0.06046617599999997
||a10-a9|| = 0.04031078399999999

a10 = 0.06046617599999997
a11 = 0.036279705599999976
||a11-a10|| = 0.024186470399999993

a11 = 0.036279705599999976
a12 = 0.021767823359999987
||a12-a11|| = 0.014511882239999989

a12 = 0.021767823359999987
a13 = 0.013060694015999992
||a13-a12|| = 0.008707129343999994

a13 = 0.013060694015999992
a14 = 0.007836416409599995
||a14-a13|| = 0.005224277606399997

a14 = 0.007836416409599995
a15 = 0.004701849845759997
||a15-a14|| = 0.0031345665638399982

a15 = 0.004701849845759997
```

```
a16 = 0.002821109907455998
||a16-a15|| = 0.0018807399383039999

a16 = 0.002821109907455998
a17 = 0.0016926659444735988
||a17-a16|| = 0.0011284439629823991

a17 = 0.0016926659444735988
a18 = 0.0010155995666841593
||a18-a17|| = 0.0006770663777894395
```

Out[24]:
```
0.0010155995666841593
```

Como dicho anteriormente, la implementación `descenso_gradiente` permite hacer descenso por gradiente de funciones con múltiples variables.

No obstante, en este caso no se puede definir las variables por su nombre (x,y,z,t,...), sino se tiene que expresar como posiciones en un vector (debido a que los cálculos son vectoriales).

## Ejemplo 1

Convertir $2 * x + y$ en formato vectorial: `2 * v[0] + v[1]` .

Así:

- $x$ corresponde a la posición 0
- $y$ corresponde a la posición 1

## Ejemplo 2

Convertir $2 * x + 3 * y + y^8 * x + z$ en formato vectorial: `2 * v[0] + 3 * v[1] + v[1] ** 8 * v[0] + v[2]` .

Así:

- $x$ corresponde a la posición 0
- $y$ corresponde a la posición 1
- $z$ corresponde a la posición 2

Veamos la ejecución de este ejemplo para:

- $x = 2$
- $y = 3$
- $z = 4$

In [25]:
```
def ejemplo2(v):
    return 2 * v[0] + 3 * v[1] + v[1] ** 8 * v[0] + v[2]


ejemplo2([2, 3, 4])  # 2 * 2 + 3 * 3 + 3 ** 8 * 2 + 4
```

Out[25]:
```
13139
```

De la misma forma se puede combinar multiples vectores con multiples variables, solo basta que la entrada sea un vector y la salida sea una matriz.

Esto nos sirve para usar en las derivadas parciales de dos variables pues son 2 funciones con 2 incógnitas.

## Ejemplo 3

Función original: $x^2 + y^3 + 4xy$
Parciales:

- $dx$: $2x + 4y$
- $dy$: $3y + 4x$

Representando las parciales "vectorizadas" asumiendo que $x$ es la posición 0 y $y$ la 1:

- $dx$: `2*v[0]+4*v[1]`
- $dy$: `3*v[1]+4*v[0]`

Ahora queremos calcular para el punto $(8, 5)$:

In [26]:
```
def ejemplo3(v):
    # Tener en cuenta que es un vector donde la posición 0 es dx y la 1 es dy
    return [2.0 * v[0] + 4 * v[1], 3.0 * v[1] + 4 * v[0]]


ejemplo3([8, 5])
```

Out[26]:
```
[36.0, 47.0]
```

Por lo tanto, si pasamos una función con derivadas parciales al `descenso_gradiente` juntamente con un punto $(x, y)$ este calculará correctamente debido a solamente usa operaciones vectoriales.

Veamos una toma del calculo del nuevo punto del ejemplo anterior con $\gamma = 0,2$

Recuerda que la función es $a_{n+1} = a_n - \gamma * f(a_n)$

In [27]:
```
[8, 5] - np.multiply(0.2, ejemplo3([8, 5]))
```

Out[27]:
```
array([ 0.8, -4.4])
```

Por lo tanto se puede reutilizar `descenso_gradiente` para cualquiera función original sin limite de incógnitas.

Veamos el ejemplo de la clase teórica: $x^2 + xy + 3y^2$

Parciales:

- $dx$: $2x + y$
- $dy$: $x + 6y$

Que se traduce en el siguiente vector: `[2 * v[0] + v[1], v[0] + 6 * v[1]`

Hagamos entonces la búsqueda de la raíz:

In [28]:
```
descenso_gradiente(
    lambda v: np.array([2 * v[0] + v[1], v[0] + 6 * v[1]]),
    [3, 3],
    gamma=0.1,
    epsilon=0.16,
)
```

```
a0 = [3, 3]
a1 = [2.1 0.9]
||a1-a0|| = 2.2847319317591723

a1 = [2.1 0.9]
a2 = [1.59 0.15]
||a2-a1|| = 0.9069729874698584

a2 = [1.59 0.15]
a3 = [ 1.257 -0.099]
||a3-a2|| = 0.41580043290020746

a3 = [ 1.257 -0.099]
a4 = [ 1.0155 -0.1653]
||a4-a3|| = 0.2504355006783184

a4 = [ 1.0155 -0.1653]
a5 = [ 0.82893 -0.16767]
||a5-a4|| = 0.18658505245597784

a5 = [ 0.82893 -0.16767]
a6 = [ 0.679911 -0.149961]
||a6-a5|| = 0.15006755492777246
```

Out[28]:
```
array([ 0.679911, -0.149961])
```

# Descenso por gradiente - No Lineal (NUMPY)

En esta parte hemos llegado a los limites de abstracción del Numpy, y por lo tanto, hay que tomar una de las dos rutas posibles:

1. Calcular la función objetivo F manualmente y pasarla al `descenso_gradiente`
2. Usar `Sympy` para, simbólicamente, calcular la función objetivo F, converter esta en Numpy y pasarla al `descenso_gradiente`

Veamos entonces los dos métodos para resolver este sistema:

$$sin(x) - y^2 = 0$$

$$e^x + 2y - 1 = 0$$

## Método 1

$$F = \frac{1}{2} \times G^T G$$

entonces

$$G = \begin{bmatrix} sin(x) - y^2 \\ e^x + 2y - 1 \end{bmatrix}$$

$$G^T = \begin{bmatrix} sin(x) - y^2, e^x + 2y - 1 \end{bmatrix}$$

$$G^T G = \begin{bmatrix} (sin(x) - y^2)^2 + (e^x + 2y - 1)^2 \end{bmatrix}$$

$$F = \begin{bmatrix} \frac{1}{2}(sin(x) - y^2)^2 + \frac{1}{2}(e^x + 2y - 1)^2 \end{bmatrix}$$

Derivadas parciales:

$$\frac{d}{dx} = \cos(x)(\sin(x) - y^2) + e^x(e^x + 2y - 1)$$

$$\frac{d}{dy} = -2y\sin(x) + 2e^x + 2y^3 + 4y - 2$$

In [29]:
```
def F(v):
    return 0.5 * ((np.sin(v[0]) - v[1] ** 2) ** 2 + (np.exp(v[0]) + 2 * v
[1] - 1) ** 2)


def Fd(v):
    return np.array(
        [
            np.exp(v[0]) * (-1 + np.exp(v[0]) + 2 * v[1])
            + np.cos(v[0]) * (-v[1] ** 2 + np.sin(v[0])),
            2 * np.exp(v[0]) + 4 * v[1] - 2 + 2 * v[1] ** 3 - 2 * v[1] * n
p.sin(v[0]),
        ]
    )


dg = descenso_gradiente(Fd, [1, 1], gamma=0.01, epsilon=0.001, n=100)
```

```
a0 = [1, 1]
a1 = [0.89978316 0.92246378]
||a1-a0|| = 0.126709433815669

a1 = [0.89978316 0.92246378]
a2 = [0.81895679 0.85513395]
||a2-a1|| = 0.10519604534935628

a2 = [0.81895679 0.85513395]
a3 = [0.75140833 0.79555191]
||a3-a2|| = 0.09007116084076357

a3 = [0.75140833 0.79555191]
a4 = [0.69357007 0.74212199]
||a4-a3|| = 0.07874021109079146

a4 = [0.69357007 0.74212199]
a5 = [0.64316639 0.69373436]
||a5-a4|| = 0.06987054589784779

a5 = [0.64316639 0.69373436]
a6 = [0.59865198 0.64957874]
||a6-a5|| = 0.06269969881494758

a6 = [0.59865198 0.64957874]
a7 = [0.55892698 0.60904161]
||a7-a6|| = 0.05675679966793305

a7 = [0.55892698 0.60904161]
a8 = [0.52317934 0.571645  ]
||a8-a7|| = 0.05173393193236641

a8 = [0.52317934 0.571645  ]
a9 = [0.49079148 0.5370078 ]
||a9-a8|| = 0.047420556563758785

a9 = [0.49079148 0.5370078 ]
a10 = [0.46128207 0.50482019]
||a10-a9|| = 0.04366746844006912

a10 = [0.46128207 0.50482019]
a11 = [0.43426815 0.47482613]
||a11-a10|| = 0.04036576948324566

a11 = [0.43426815 0.47482613]
a12 = [0.40943964 0.44681098]
||a12-a11|| = 0.037433994122295554

a12 = [0.40943964 0.44681098]
a13 = [0.38654158 0.4205925 ]
||a13-a12|| = 0.034809903581713104

a13 = [0.38654158 0.4205925 ]
a14 = [0.36536153 0.39601429]
||a14-a13|| = 0.032445075822498635

a14 = [0.36536153 0.39601429]
a15 = [0.34572037 0.37294072]
||a15-a14|| = 0.03030123390593477

a15 = [0.34572037 0.37294072]
```

```
a16 = [0.3274655  0.35125314]
||a16-a15|| = 0.02834769099285006

a16 = [0.3274655  0.35125314]
a17 = [0.31046567 0.3308469 ]
||a17-a16|| = 0.02655953308229475

a17 = [0.31046567 0.3308469 ]
a18 = [0.29460707 0.31162902]
||a18-a17|| = 0.024916301355692037

a18 = [0.29460707 0.31162902]
a19 = [0.27979024 0.29351634]
||a19-a18|| = 0.023401020335721775

a19 = [0.27979024 0.29351634]
a20 = [0.26592767 0.27643401]
||a20-a19|| = 0.02199947011505042

a20 = [0.26592767 0.27643401]
a21 = [0.25294189 0.26031432]
||a21-a20|| = 0.02069963387639869

a21 = [0.25294189 0.26031432]
a22 = [0.24076394 0.24509567]
||a22-a21|| = 0.0194912733007648

a22 = [0.24076394 0.24509567]
a23 = [0.2293321 0.2307218]
||a23-a22|| = 0.01836559861535117

a23 = [0.2293321 0.2307218]
a24 = [0.21859089 0.21714108]
||a24-a23|| = 0.017315009586526986

a24 = [0.21859089 0.21714108]
a25 = [0.20849021 0.20430599]
||a25-a24|| = 0.016332890324510685

a25 = [0.20849021 0.20430599]
a26 = [0.19898466 0.19217262]
||a26-a25|| = 0.015413445344828563

a26 = [0.19898466 0.19217262]
a27 = [0.19003295 0.18070026]
||a27-a26|| = 0.014551567573276613

a27 = [0.19003295 0.18070026]
a28 = [0.18159739 0.16985111]
||a28-a27|| = 0.013742731307447721

a28 = [0.18159739 0.16985111]
a29 = [0.17364351 0.15958994]
||a29-a28|| = 0.012982904838276917

a29 = [0.17364351 0.15958994]
a30 = [0.16613965 0.14988387]
||a30-a29|| = 0.012268478677631882

a30 = [0.16613965 0.14988387]
a31 = [0.1590567  0.14070216]
```

```
||a31-a30|| = 0.011596206261254266

a31 = [0.1590567  0.14070216]
a32 = [0.15236783 0.13201598]
||a32-a31|| = 0.010963154689306252

a32 = [0.15236783 0.13201598]
a33 = [0.14604822 0.1237983 ]
||a33-a32|| = 0.01036666359175475

a33 = [0.14604822 0.1237983 ]
a34 = [0.14007493 0.11602371]
||a34-a33|| = 0.009804310607051767

a34 = [0.14007493 0.11602371]
a35 = [0.13442666 0.1086683 ]
||a35-a34|| = 0.009273882271743715

a35 = [0.13442666 0.1086683 ]
a36 = [0.12908366 0.10170957]
||a36-a35|| = 0.008773349358719608

a36 = [0.12908366 0.10170957]
a37 = [0.12402753 0.09512629]
||a37-a36|| = 0.00830084588957757

a37 = [0.12402753 0.09512629]
a38 = [0.11924114 0.08889844]
||a38-a37|| = 0.007854651194439111

a38 = [0.11924114 0.08889844]
a39 = [0.11470854 0.08300713]
||a39-a38|| = 0.007433174509690969

a39 = [0.11470854 0.08300713]
a40 = [0.11041481 0.07743449]
||a40-a39|| = 0.007034941697508873

a40 = [0.11041481 0.07743449]
a41 = [0.10634603 0.07216365]
||a41-a40|| = 0.006658583745847674

a41 = [0.10634603 0.07216365]
a42 = [0.10248916 0.06717865]
||a42-a41|| = 0.006302826767860009

a42 = [0.10248916 0.06717865]
a43 = [0.09883202 0.0624644 ]
||a43-a42|| = 0.00596648326849015

a43 = [0.09883202 0.0624644 ]
a44 = [0.09536316 0.0580066 ]
||a44-a43|| = 0.005648444485648216

a44 = [0.09536316 0.0580066 ]
a45 = [0.09207188 0.05379173]
||a45-a44|| = 0.0053476736457409605

a45 = [0.09207188 0.05379173]
a46 = [0.08894813 0.04980699]
||a46-a45|| = 0.005063199999858476
```

```
a46 = [0.08894813 0.04980699]
a47 = [0.08598246 0.04604025]
||a47-a46|| = 0.004794113528725384

a47 = [0.08598246 0.04604025]
a48 = [0.08316602 0.04248002]
||a48-a47|| = 0.004539560222515063

a48 = [0.08316602 0.04248002]
a49 = [0.08049046 0.03911542]
||a49-a48|| = 0.004298737856511601

a49 = [0.08049046 0.03911542]
a50 = [0.07794796 0.03593613]
||a50-a49|| = 0.004070892195955718

a50 = [0.07794796 0.03593613]
a51 = [0.07553114 0.0329324 ]
||a51-a50|| = 0.0038553135736862946

a51 = [0.07553114 0.0329324 ]
a52 = [0.07323306 0.03009495]
||a52-a51|| = 0.0036513337927591588

a52 = [0.07323306 0.03009495]
a53 = [0.07104721 0.02741502]
||a53-a52|| = 0.003458323313387113

a53 = [0.07104721 0.02741502]
a54 = [0.06896741 0.0248843 ]
||a54-a53|| = 0.0032756886895462427

a54 = [0.06896741 0.0248843 ]
a55 = [0.06698788 0.02249489]
||a55-a54|| = 0.0031028702256299125

a55 = [0.06698788 0.02249489]
a56 = [0.06510316 0.02023933]
||a56-a55|| = 0.002939339827768121

a56 = [0.06510316 0.02023933]
a57 = [0.0633081  0.01811054]
||a57-a56|| = 0.0027845990279994383

a57 = [0.0633081  0.01811054]
a58 = [0.06159783 0.01610182]
||a58-a57|| = 0.0026381771624961184

a58 = [0.06159783 0.01610182]
a59 = [0.05996779 0.01420679]
||a59-a58|| = 0.0024996296875915095

a59 = [0.05996779 0.01420679]
a60 = [0.05841365 0.01241945]
||a60-a59|| = 0.0023685366195175814

a60 = [0.05841365 0.01241945]
a61 = [0.05693133 0.01073406]
||a61-a60|| = 0.002244501085593371
```

```
a61 = [0.05693133 0.01073406]
a62 = [0.05551699 0.00914523]
||a62-a61|| = 0.00212714797616266

a62 = [0.05551699 0.00914523]
a63 = [0.05416699 0.00764781]
||a63-a62|| = 0.0020161226879068577

a63 = [0.05416699 0.00764781]
a64 = [0.0528779  0.00623696]
||a64-a63|| = 0.0019110899502914765

a64 = [0.0528779  0.00623696]
a65 = [0.05164649 0.00490805]
||a65-a64|| = 0.0018117327278733887

a65 = [0.05164649 0.00490805]
a66 = [0.05046969 0.00365672]
||a66-a65|| = 0.0017177511920261678

a66 = [0.05046969 0.00365672]
a67 = [0.04934462 0.00247884]
||a67-a66|| = 0.0016288617563538206

a67 = [0.04934462 0.00247884]
a68 = [0.04826854 0.00137049]
||a68-a67|| = 0.0015447961706773383

a68 = [0.04826854 0.00137049]
a69 = [0.04723888 0.00032794]
||a69-a68|| = 0.0014653006690085733

a69 = [0.04723888 0.00032794]
a70 = [ 0.04625319 -0.00065232]
||a70-a69|| = 0.0013901351673853218

a70 = [ 0.04625319 -0.00065232]
a71 = [ 0.04530918 -0.00157362]
||a71-a70|| = 0.0013190725078409487

a71 = [ 0.04530918 -0.00157362]
a72 = [ 0.04440467 -0.00243912]
||a72-a71|| = 0.0012518977451305476

a72 = [ 0.04440467 -0.00243912]
a73 = [ 0.04353759 -0.00325183]
||a73-a72|| = 0.0011884074731420247

a73 = [ 0.04353759 -0.00325183]
a74 = [ 0.04270601 -0.00401457]
||a74-a73|| = 0.001128409188190793

a74 = [ 0.04270601 -0.00401457]
a75 = [ 0.04190808 -0.00473004]
||a75-a74|| = 0.0010717206866380654

a75 = [ 0.04190808 -0.00473004]
a76 = [ 0.04114206 -0.00540077]
||a76-a75|| = 0.0010181694944887943

a76 = [ 0.04114206 -0.00540077]
```

```
a77 = [ 0.0404063  -0.00602918]
||a77-a76|| = 0.0009675923268222899
```

In [30]:
```
F(dg)
```

Out[30]:
```
0.0012400242093827598
```

# Descenso por gradiente - Lineal (SYMPY)

Em `sympy` se puede escribir el algoritmo de forma más similar a lo que hemos visto en clase de teoría, al poder programar de forma simbólica. De hecho, no hay necesidad de calcular derivadas manualmente ya que `sympy` puede hacerlo por nosotros.

**En esta implementación solo podemos usar la forma linear con 2 variables. La forma generalizada (como la de numpy) es un ejercicio propuesto.**

## Pasos

1. Definir la función matemática a calcular como `func_original`
2. Ejecutar `sym_descenso_gradiente` con los valores iniciales como argumentos

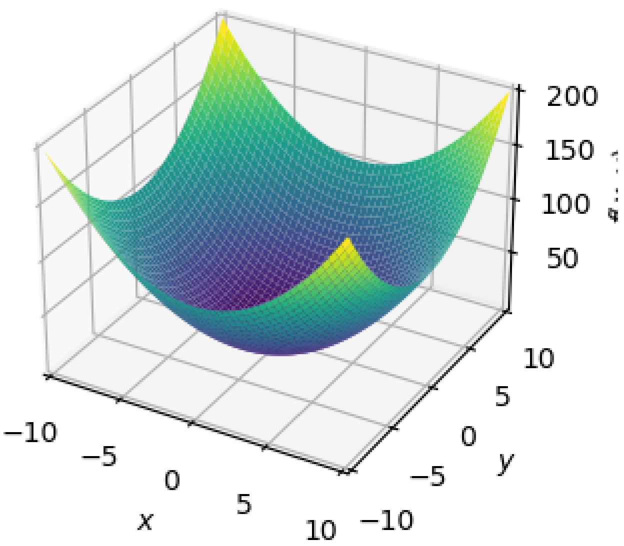
No obstante para que esto funcione tenemos que crear las funciones `sym_descenso_gradiente` . Veamos entonces como:

In [31]:
```python
import numpy as np
from sympy import *
from sympy.abc import x, y


def sym_descenso_gradiente(
    func, x0, y0, gamma=0.2, epsilon=0.001, num_iters=50, verbose=True
):
    if verbose:
        print(f"El valor de la función inicial es: {func.subs({x: x0, y: y0})}\n")
    i = 0
    while (
        i < num_iters
    ):  # De base solo paramos si alcanzado el número máximo de iteraciones, pero abajo confirmamos el valor de la "norma" para parar se se da la condición
        # Derivada parcial de x en el punto (xis, yis)[0]
        x1 = x0 - gamma * diff(func, x).subs({x: x0, y: y0})
        # Derivada parcial de x en el punto (xis, yis)
        y1 = y0 - gamma * diff(func, y).subs({x: x0, y: y0})
        norma = sqrt((x0 - x1) ** 2 + (y0 - y1) ** 2)  # Calculo de la norma
        if verbose:
            # Muestra cuan cerca estamos del 0 en la función original
            print(f"El valor de la función es: {func.subs({x: x1, y: y1})}")
            print(f"||x{i+1}-x{i}||={norma}")  # Muestra el valor de la norma
            print()
        x0, y0 = x1, y1  # Actualiza los valores de x e y para el proximo bucle
        if (
            norma < epsilon
        ):  # Comprobamos norma y caso se haya alcanzado este valor paramos de forma elegante a poner "i" como el máximo de iteraciones, así el proximo bucle para
            i = num_iters
        i += 1  # incrementamos la cantidad de "i"
    return x0, y0, func.subs({x: x0, y: y0})
```

Vamos a probarlo con una función básica: $f(x, y) = x^2 + y^2$

La definimos y visualizamos:

In [32]:
```
from sympy.plotting import plot3d
import matplotlib.pyplot as plt

func = x**2 + y**2

plot3d(func, (x, -10, 10), (y, -10, 10), size=(5, 3,5))
```




Out[32]: <sympy.plotting.plot.Plot at 0x22eb0bdcf50>

Está claro que el mínimo es (0,0). Usemos el descenso por gradiente y lo confirmamos.

In [33]:
```
sym_descenso_gradiente(func, 10, 10, 0.2, 0.001)
```

```
El valor de la función inicial es: 200

El valor de la función es: 72.0000000000000
||x1-x0||=5.65685424949238

El valor de la función es: 25.9200000000000
||x2-x1||=3.39411254969543

El valor de la función es: 9.33120000000000
||x3-x2||=2.03646752981726

El valor de la función es: 3.35923200000000
||x4-x3||=1.22188051789035

El valor de la función es: 1.20932352000000
||x5-x4||=0.733128310734212

El valor de la función es: 0.435356467200000
||x6-x5||=0.439876986440527

El valor de la función es: 0.156728328192000
||x7-x6||=0.263926191864316

El valor de la función es: 0.0564221981491200
||x8-x7||=0.158355715118590

El valor de la función es: 0.0203119913336832
||x9-x8||=0.0950134290711539

El valor de la función es: 0.00731231688012594
||x10-x9||=0.0570080574426923

El valor de la función es: 0.00263243407684534
||x11-x10||=0.0342048344656154

El valor de la función es: 0.000947676267664322
||x12-x11||=0.0205229006793692

El valor de la función es: 0.000341163456359156
||x13-x12||=0.0123137404076215

El valor de la función es: 0.000122818844289296
||x14-x13||=0.00738824424457293

El valor de la función es: 0.0000442147839441466
||x15-x14||=0.00443294654674376

El valor de la función es: 0.0000159173222198928
||x16-x15||=0.00265976792804625

El valor de la función es: 0.00000573023599916140
||x17-x16||=0.00159586075682775

El valor de la función es: 0.00000206288495969810
||x18-x17||=0.000957516454096651
```

Out[33]:
```
(0.00101559956668416, 0.00101559956668416, 2.06288495969810e-6)
```

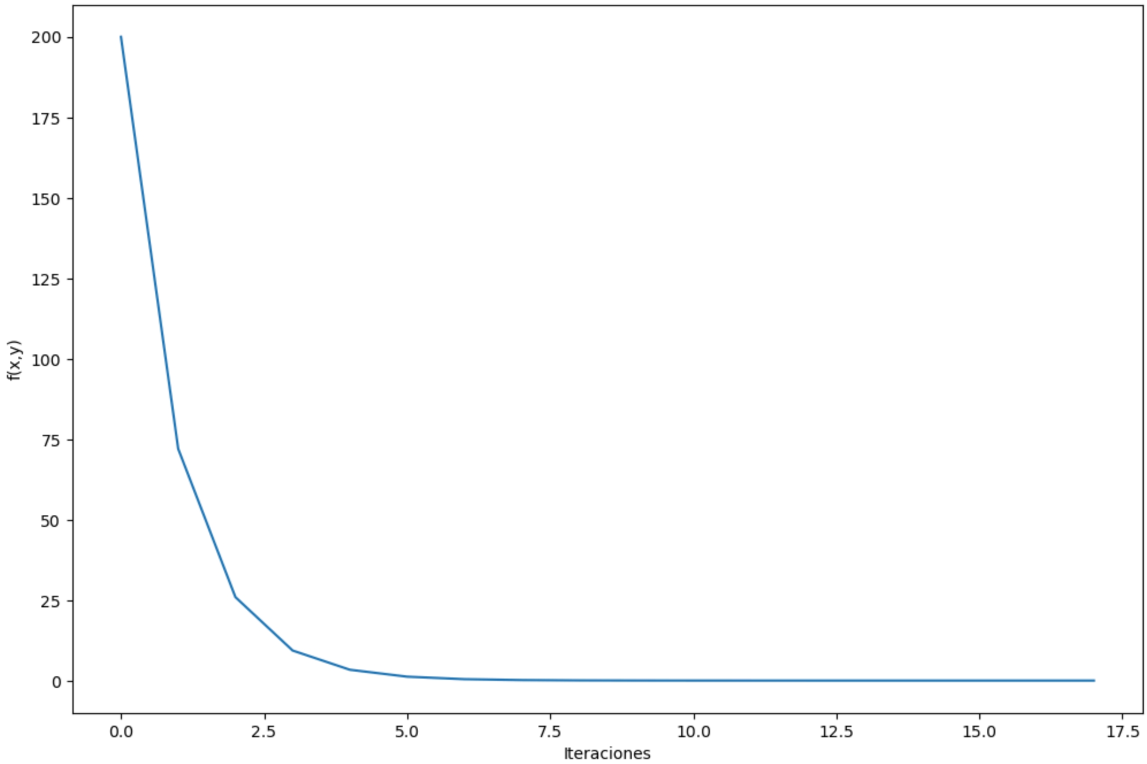
Como podemos observar el valor que obtenemos está muy cerca del (0,0). No obstante, se puede mejorar estos valores probando con distintos valores de gamma y epsilon. Aquí se puede ver uno de los mayores problemas del descenso por gradiente, que son los valores de ajuste. Un gamma muy pequeño y puede nunca encontrarse la raíz, un gamma muy grande y puede nunca convergir.

Hay métodos para paliar este problema al detectar si evolucionan correctamente y cambiar el valor de gamma de acorde.

Hagamos un cambio en la función `sym_descenso_gradiente` para conservar los cálculos intermedios y podamos visualizar la progresión.

In [34]:
```python
from sympy import *
from sympy.abc import x, y


def sym_descenso_gradiente(
    func, x0, y0, gamma=0.2, epsilon=0.001, num_iters=50, verbose=True
):
    calculos_intermedios = []
    if verbose:
        print(f"El valor de la función inicial es: {func.subs({x: x0, y: y0})}\n")
    i = 0
    while (
        i < num_iters
    ):  # De base solo paramos si alcanzado el número máximo de iteraciones, pero abajo confirmamos el valor de la "norma" para parar se se da la condición
        # Derivada parcial de x en el punto (xis, yis)[0]
        x1 = x0 - gamma * diff(func, x).subs({x: x0, y: y0})
        # Derivada parcial de x en el punto (xis, yis)
        y1 = y0 - gamma * diff(func, y).subs({x: x0, y: y0})
        norma = sqrt((x0 - x1) ** 2 + (y0 - y1) ** 2)  # Calculo de la norma
        calculos_intermedios.append(func.subs({x: x0, y: y0}))
        if verbose:
            # Muestra cuan cerca estamos del 0 en la función original
            print(f"El valor de la función es: {func.subs({x: x1, y: y1})}")
            print(f"||x{i+1}-x{i}||={norma}")  # Muestra el valor de la norma
            print()
        x0, y0 = x1, y1  # Actualiza los valores de x e y para el proximo bucle
        if (
            norma < epsilon
        ):  # Comprobamos norma y caso se haya alcanzado este valor paramos de forma elegante a poner "i" como el máximo de iteraciones, así el proximo bucle para
            i = num_iters
        i += 1  # incrementamos la cantidad de "i"
    return x0, y0, func.subs({x: x0, y: y0}), calculos_intermedios
```

In [35]:
```
x_final, y_final, f_final, calc_intermedios = sym_descenso_gradiente(
    func, 10, 10, 0.2, 0.001, verbose=False
)  # Obtenemos los valores intermedios de f(x,y)

# Representamos visualmente
plt.figure(figsize=(12, 8))
plt.xlabel("Iteraciones")
plt.ylabel("f(x,y)")
plt.plot(
    range(len(calc_intermedios)), calc_intermedios
)  # abscisas: de 0 a tamaño de calc_intermedios; ordenadas: calc_intermedi
os
plt.show()
```



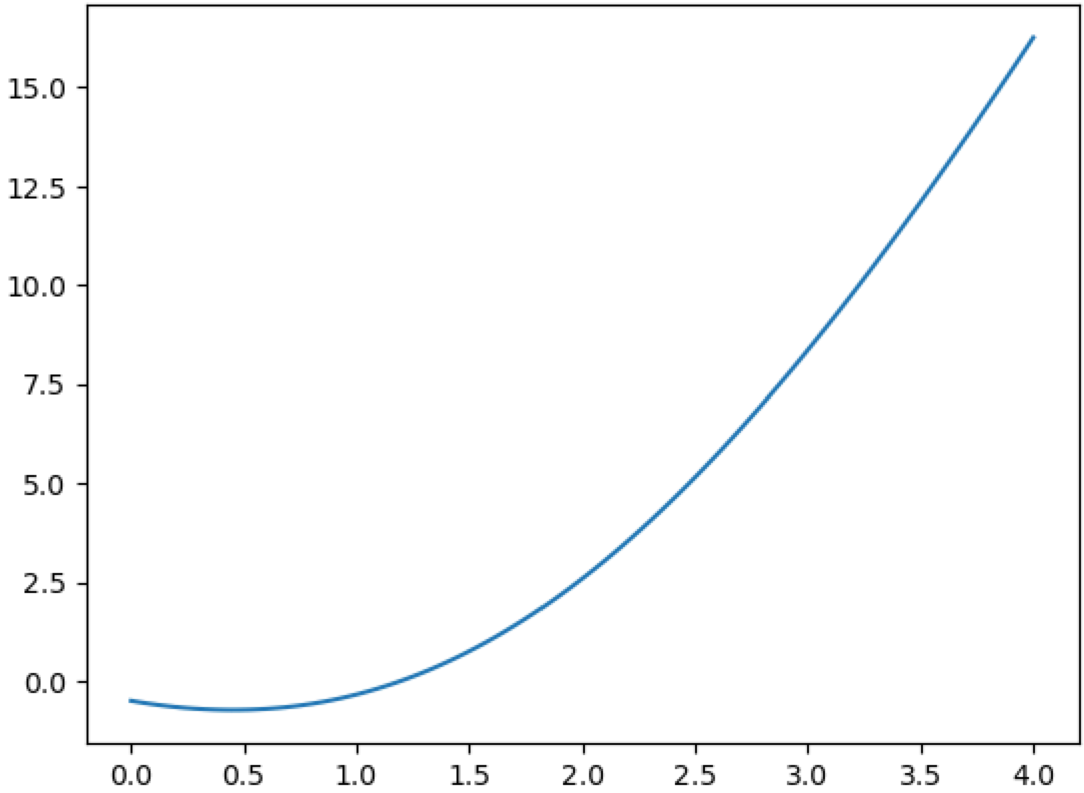

Aquí se puede ver la rápida aproximación la raíz.

# Ejercicios

## Ejercicio 1

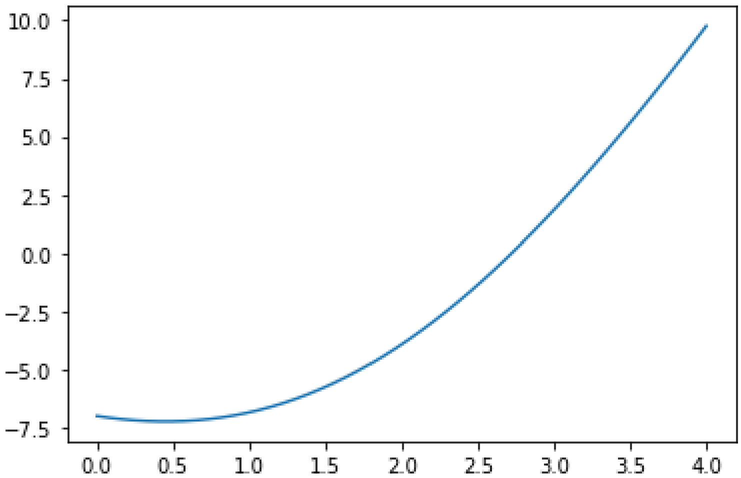
Representa y busca la raíz de $x^2 - sen(x) - 0.5$ con una tolerancia de $10^{-3}$ para $0 \leq x \leq 4$ con el método de la Bisección.

In [36]:
```
f_ej1 = x**2 - sin(x) -0.5

a_ej1=0
b_ej1=4
n_ej1=100

xx = np.linspace(a_ej1,b_ej1,n_ej1)
yy = lambdify(x, f_ej1)(xx)

plt.plot(xx,yy)
plt.show()
```




In [ ]:
```
# Grafica
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7ffaee0446d0>]



In [63]:
```
def funcion_ej1(x):
    return x**2 - sin(x) -0.5

# bisect(funcion_yea,a_ej1,b_ej1)

bisecion(funcion_ej1,0,4,delta=0.001)
```

Out[63]: (1.19580078125, -0.000569745009508682, 12)

In [ ]:

Out[ ]: (2.7216796875, -0.00014066046435790014, 11)

## Ejercicio 2

In [62]:
```
from scipy.optimize import bisect
from numpy import sin

def funcion_ej2(x):
    return x**2 - np.sin(x) -0.5

# bisect(funcion_yea,a_ej1,b_ej1)

bisecion(funcion_ej1,0,4,epsilon=0.001)

bisect(funcion_ej1,0,4)
```

Out[62]: 1.1960820332988078

Con la función del ejercicio 1, Prueba con el método de Bisección de Scipy. Observa las diferencias de resultados.

In [ ]:

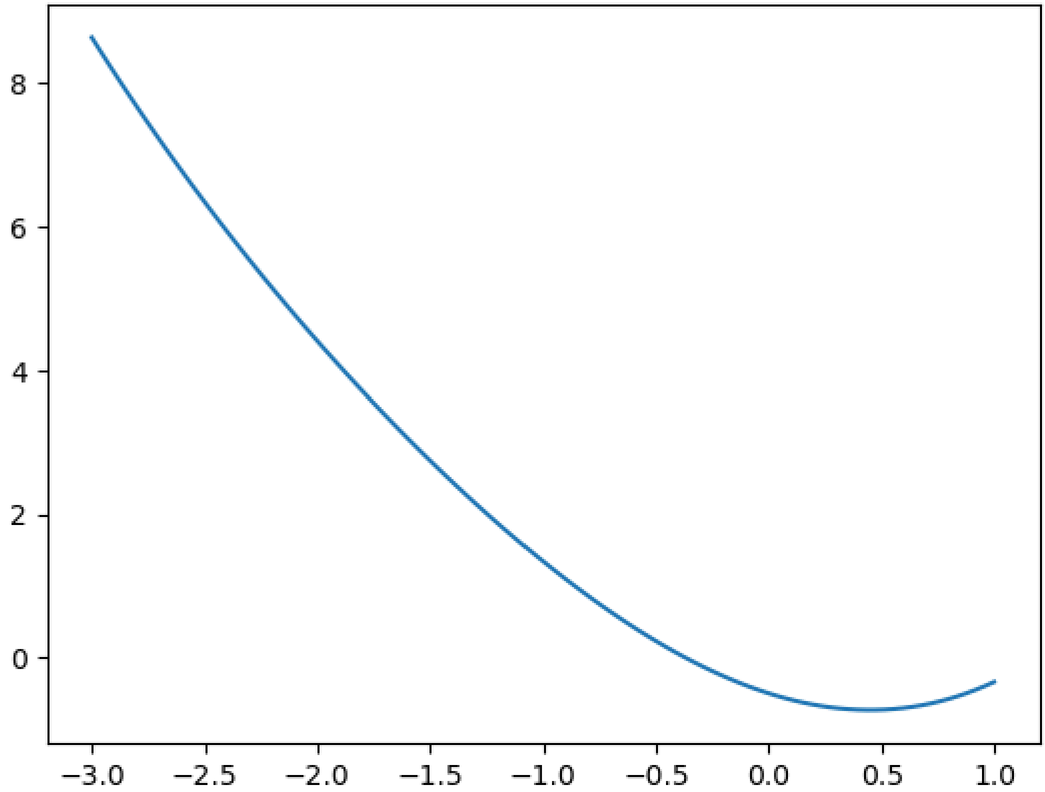
Out[ ]: 2.721701816068162

## Ejercicio 3

Con la función del ejercicio 1, cuantas iteraciones son necesarias para encontrar la raíz con el intervalo $-3 \leq x \leq 0$?

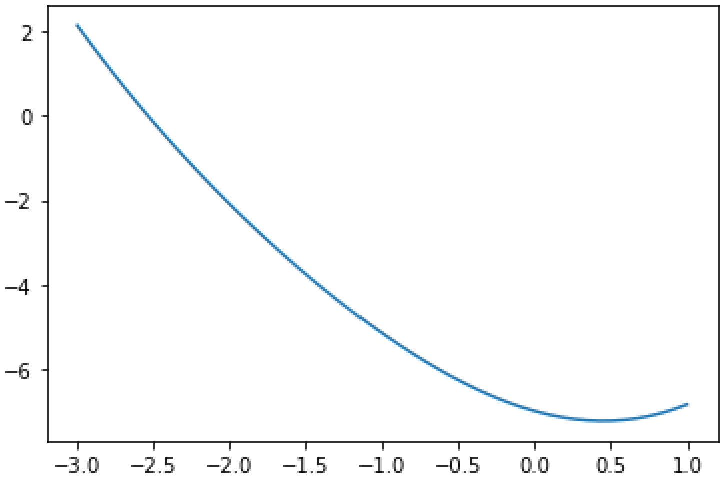
In [77]:
```
from sympy import *
from sympy.abc import x
import matplotlib.pyplot as plt
import numpy as np

print(bisecion(funcion_ej1,-3,0))

f1_ej3 = x**2 - sin(x) -0.5

xx = np.linspace(-3,1,100)
yy = lambdify(x,f1_ej3)(xx)

plt.plot(xx,yy)

plt.show()
```

```
(-0.370880126953125, -1.20732040952420e-5, 14)
```



In [ ]:

Out[ ]:
```
(-2.53564453125, -0.0009651580269380844, 10)
```




## Ejercicio 4

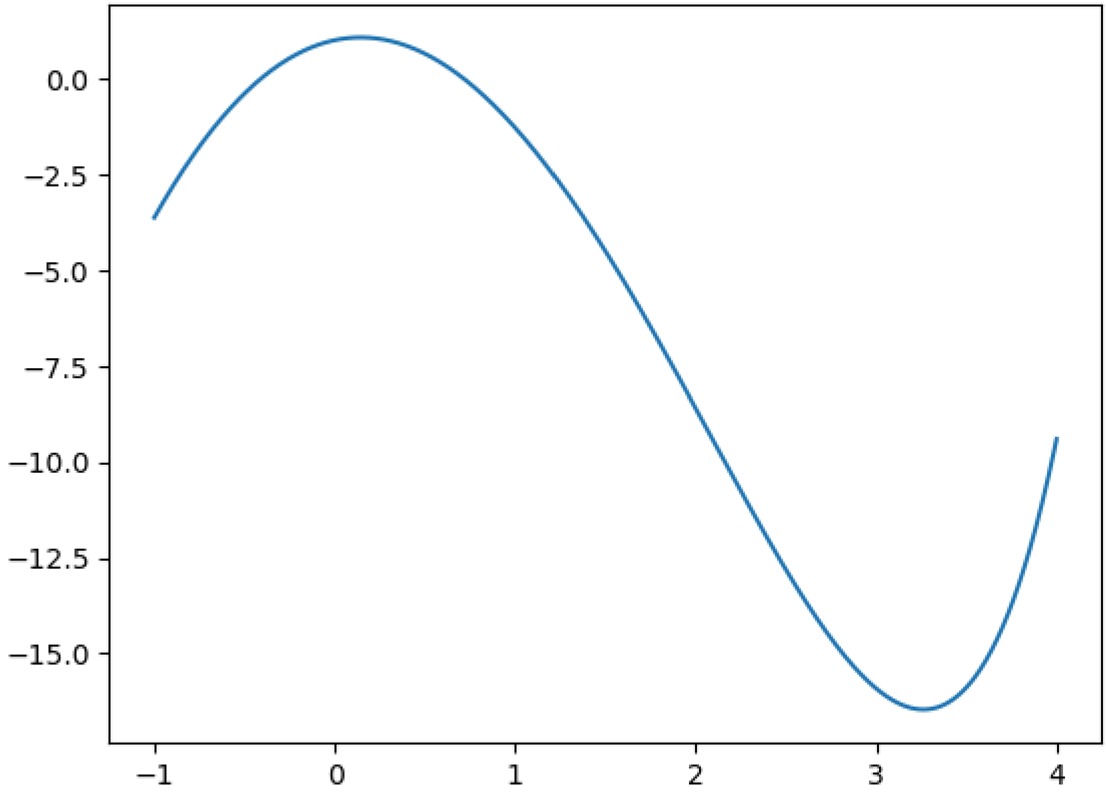
Con $e^x - 4x^2$, prueba los métodos de:

- Bisección para $-1 \leq x \leq 0$
- Secante para $0 \leq x \leq 1$
- Newton desde $x = 4$

In [88]:
```
from sympy import *
from sympy.abc import x
import matplotlib.pyplot as plt
import numpy as np
```

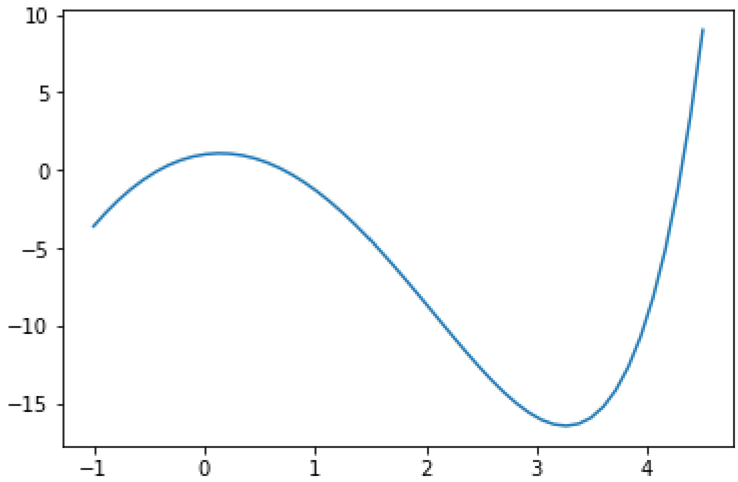
In [89]:
```
f_ej4 = exp(x) - 4*x**2
```

In [90]:
```
xx = np.linspace(-1,4,1000)
yy = lambdify(x,f_ej4)(xx)

plt.plot(xx,yy)
plt.show()
```




In [ ]:
```
# Grafica
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7ffaedd44fa0>]




In [93]:
```
def f2_ej4(x):
    return exp(x) - 4*x**2

bisecion(f2_ej4,-1,0)
```

Out[93]: (-0.40777587890625, 3.26164726582867e-6, 13)

In [ ]:
```
# Bisec
```

Out[ ]:
```
(-0.40777587890625, 3.261647265828671e-06, 13)
```

In [116]:
```
# exp(x) - 4*x**2,0,1

xis, fn, iter = secante(lambda x: exp(x) - 4*x**2,0,1, epsilon=0.05, delta=
0.01, n=3)
```

```
i: 0 a: 0.000 b: 1.000 c: 0.438 h: -0.438 f(a): 1.000 f(b): -1.282 f(c): 0.
782
i: 1 a: 0.438 b: 0.000 c: 2.008 h: -1.569 f(a): 0.782 f(b): 1.000 f(c): -8.
677
i: 2 a: 0.438 b: 2.008 c: 0.568 h: -0.130 f(a): 0.782 f(b): -8.677 f(c): 0.
474
```

In [ ]:
```
# Secante
```

```
i: 0 a: 0.000 b: 1.000 c: 0.438 h: -0.438 f(a): 1.000 f(b): -1.282 f(c): 0.
782
i: 1 a: 0.438 b: 0.000 c: 2.008 h: -1.569 f(a): 0.782 f(b): 1.000 f(c): -8.
677
i: 2 a: 0.438 b: 2.008 c: 0.568 h: -0.130 f(a): 0.782 f(b): -8.677 f(c): 0.
474
i: 3 a: 0.568 b: 0.438 c: 0.768 h: -0.200 f(a): 0.474 f(b): 0.782 f(c): -0.
204
i: 4 a: 0.768 b: 0.568 c: 0.708 h: 0.060 f(a): -0.204 f(b): 0.474 f(c): 0.0
25
i: 5 a: 0.708 b: 0.768 c: 0.715 h: -0.007 f(a): 0.025 f(b): -0.204 f(c): 0.
001
i: 6 a: 0.715 b: 0.708 c: 0.715 h: -0.000 f(a): 0.001 f(b): 0.025 f(c): -0.
000
```

Out[ ]:
```
(0.7148075428694096, -5.991561062845818e-06, 6)
```

In [125]:
```
from scipy.optimize import newton as scinewton

f = exp(x) - 4*x**2


print("Scipy:")
print(scinewton(lambdify(x, f), 4, lambdify(x, f.diff(x))))
print()

print("Alternativa:")
Newton(lambda x: exp(x) - 4*x**2,4,n=3)
```

```
Scipy:
4.306584728220699

Alternativa:
i: 0 a: 4.000 c: 4.292 h: -0.292 f(a): -9.402 f(c): -0.560
i: 1 a: 4.292 c: 4.303 h: -0.011 f(a): -0.560 f(c): -0.135

C:\Users\padil\AppData\Local\Temp\ipykernel_39716\2300854324.py:8: Deprecat
ionWarning: scipy.misc.derivative is deprecated in SciPy v1.10.0; and will
be completely removed in SciPy v1.12.0. You may consider using findiff: htt
ps://github.com/maroba/findiff or numdifftools: https://github.com/pbrod/nu
mdifftools
  h = f(puntoA) / derivative(f, puntoA)
C:\Users\padil\AppData\Local\Temp\ipykernel_39716\2300854324.py:12: Depreca
tionWarning: scipy.misc.derivative is deprecated in SciPy v1.10.0; and will
be completely removed in SciPy v1.12.0. You may consider using findiff: htt
ps://github.com/maroba/findiff or numdifftools: https://github.com/pbrod/nu
mdifftools
  h = f(puntoA) / derivative(f, puntoA)

i: 2 a: 4.303 c: 4.306 h: -0.003 f(a): -0.135 f(c): -0.033
```

Out[125]:
```
(-0.031090842972909*exp(4) - (-4*(-0.031090842972909*exp(4) - (-4*(5.989813
95026618 - 0.031090842972909*exp(4))**2 + 399.340305857588*exp(-0.031090842
972909*exp(4)))/(-2.0*(6.98981395026618 - 0.031090842972909*exp(4))**2 + 2.
0*(4.98981395026618 - 0.031090842972909*exp(4))**2 + 469.305204113919*exp(-
0.031090842972909*exp(4))) + 5.98981395026618)**2 + 399.340305857588*exp(-
0.031090842972909*exp(4) - (-4*(5.98981395026618 - 0.031090842972909*exp
(4))**2 + 399.340305857588*exp(-0.031090842972909*exp(4)))/(-2.0*(6.9898139
5026618 - 0.031090842972909*exp(4))**2 + 2.0*(4.98981395026618 - 0.03109084
2972909*exp(4))**2 + 469.305204113919*exp(-0.031090842972909*exp(4)))))/(-
2.0*(-0.031090842972909*exp(4) - (-4*(5.98981395026618 - 0.031090842972909*
exp(4))**2 + 399.340305857588*exp(-0.031090842972909*exp(4)))/(-2.0*(6.9898
1395026618 - 0.031090842972909*exp(4))**2 + 2.0*(4.98981395026618 - 0.03109
0842972909*exp(4))**2 + 469.305204113919*exp(-0.031090842972909*exp(4))) +
6.98981395026618)**2 + 2.0*(-0.031090842972909*exp(4) - (-4*(5.989813950266
18 - 0.031090842972909*exp(4))**2 + 399.340305857588*exp(-0.031090842972909
*exp(4)))/(-2.0*(6.98981395026618 - 0.031090842972909*exp(4))**2 + 2.0*(4.9
8981395026618 - 0.031090842972909*exp(4))**2 + 469.305204113919*exp(-0.0310
90842972909*exp(4))) + 4.98981395026618)**2 + 469.305204113919*exp(-0.03109
0842972909*exp(4) - (-4*(5.98981395026618 - 0.031090842972909*exp(4))**2 +
399.340305857588*exp(-0.031090842972909*exp(4)))/(-2.0*(6.98981395026618 -
0.031090842972909*exp(4))**2 + 2.0*(4.98981395026618 - 0.031090842972909*ex
p(4))**2 + 469.305204113919*exp(-0.031090842972909*exp(4))))) - (-4*(5.9898
1395026618 - 0.031090842972909*exp(4))**2 + 399.340305857588*exp(-0.0310908
42972909*exp(4)))/(-2.0*(6.98981395026618 - 0.031090842972909*exp(4))**2 +
2.0*(4.98981395026618 - 0.031090842972909*exp(4))**2 + 469.305204113919*exp
(-0.031090842972909*exp(4))) + 5.98981395026618,
 2)
```

In [ ]:
```
# Newton
```

Out[ ]:
```
4.3065847282207
```

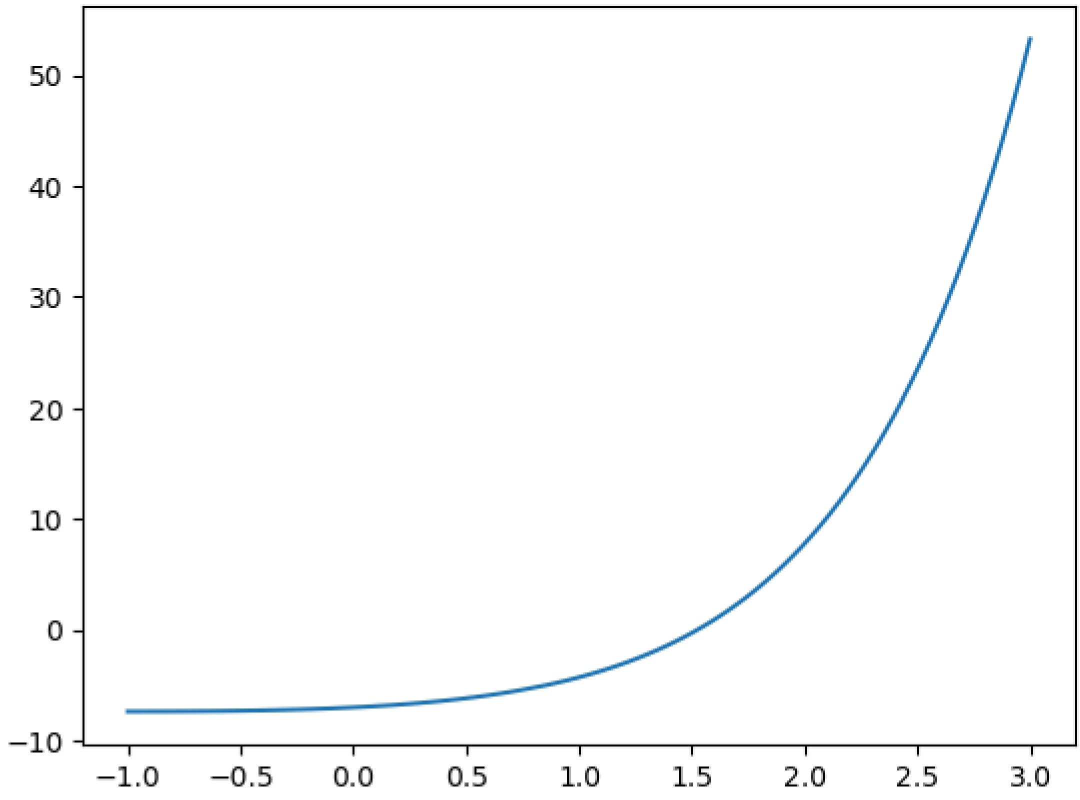
## Ejercicio 5

Con $x * e^x - 7$, prueba los métodos de:

- Bisección para $-2.5 \leq x \leq 10$
- Secante para $1 \leq x \leq 2$
- Newton desde $x = 0$

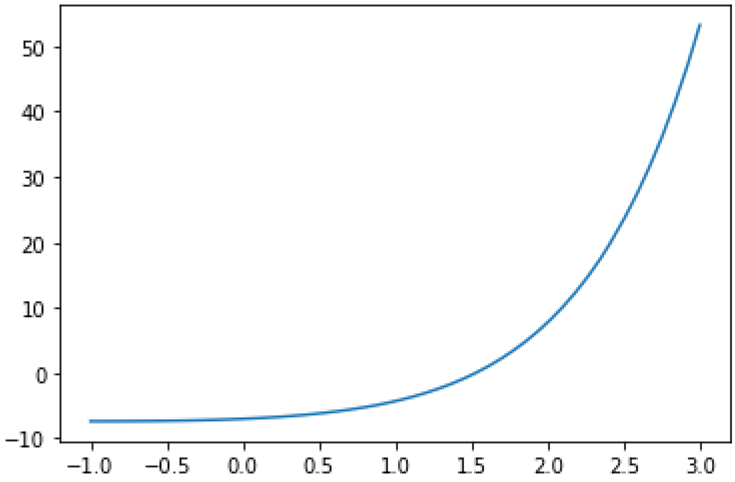
In [126]:
```
f_ej5 = x*exp(x)-7
```

In [129]:
```
xx = np.linspace(-1,3,100)
yy = lambdify(x, f_ej5)(xx)

plt.plot(xx,yy)
plt.show()
```



In [ ]:

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7ffaede871f0>]
```




In [131]:
```
# Bisección entre -2.5 y 10

bisecion(lambda x:x*exp(x)-7,-2.5,10)
```

Out[131]:
```
(1.524362564086914, 0.000201231513028866, 17)
```

In [ ]:

Out[ ]:
```
(1.524362564086914, 0.00020123151302886555, 17)
```

In [134]:
```
# Secante entre 1 y 2

secante(lambda x:x*exp(x)-7,1,2,n=2)
```

```
i: 0 a: 1.000 b: 2.000 c: 1.355 h: -0.355 f(a): -4.282 f(b): 7.778 f(c): -
1.747
i: 1 a: 1.355 b: 1.000 c: 1.600 h: -0.245 f(a): -1.747 f(b): -4.282 f(c):
0.920
```

Out[134]:
```
(-(-7 + E)*(-7 + (-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 + E)/(-E + 2*exp
(2)) + 1))/((-E + 2*exp(2))*(-(-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 +
E)/(-E + 2*exp(2)) + 1) + E)) - (-7 + E)/(-E + 2*exp(2)) + 1,
 -7 + (-(-7 + E)*(-7 + (-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 + E)/(-E +
2*exp(2)) + 1))/((-E + 2*exp(2))*(-(-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7
+ E)/(-E + 2*exp(2)) + 1) + E)) - (-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 +
E)*(-7 + (-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 + E)/(-E + 2*exp(2)) +
1))/((-E + 2*exp(2))*(-(-(-7 + E)/(-E + 2*exp(2)) + 1)*exp(-(-7 + E)/(-E +
2*exp(2)) + 1) + E)) - (-7 + E)/(-E + 2*exp(2)) + 1),
 1)
```

In [ ]:

```
i: 0 a: 1.000 b: 2.000 c: 1.355 h: -0.355 f(a): -4.282 f(b): 7.778 f(c): -
1.747
i: 1 a: 1.355 b: 1.000 c: 1.600 h: -0.245 f(a): -1.747 f(b): -4.282 f(c):
0.920
i: 2 a: 1.600 b: 1.355 c: 1.515 h: 0.084 f(a): 0.920 f(b): -1.747 f(c): -0.
105
i: 3 a: 1.515 b: 1.600 c: 1.524 h: -0.009 f(a): -0.105 f(b): 0.920 f(c): -
0.005
i: 4 a: 1.524 b: 1.515 c: 1.524 h: -0.000 f(a): -0.005 f(b): -0.105 f(c):
0.000
```

Out[ ]:

```
(1.5243482028234052, 3.475143217368526e-05, 4)
```

In [135]:

```python
# Newton desde 0
from scipy.optimize import newton as scinewton

f = x*exp(x)-7

scinewton(lambdify(x,f),0,lambdify(x,diff(f)))
```

Out[135]:

```
1.5243452049841444
```

In [ ]:

Out[ ]:

```
1.5243452049841444
```

## Ejercicio 6 - Bonus

Prueba el método de la secante del ejercicio anterior para el intervalo $-2.5 \leq x \leq 10$. Verás que falla. Busca la razón.

In [ ]:

```
i: 0 a: -2.500 b: 10.000 c: -2.500 h: -0.000 f(a): -7.205 f(b): 220257.658
f(c): -7.205
i: 1 a: -2.500 b: -2.500 c: -61.014 h: 58.514 f(a): -7.205 f(b): -7.205 f
(c): -7.000
i: 2 a: -61.014 b: -2.500 c: -2056.995 h: 1995.981 f(a): -7.000 f(b): -7.20
5 f(c): -7.000
i: 3 a: -61.014 b: -2056.995 c: -inf h: inf f(a): -7.000 f(b): -7.000 f(c):
nan

/tmp/ipykernel_30164/3537860132.py:14: RuntimeWarning: divide by zero encou
ntered in double_scalars
  h = (f(puntoA) * (puntoB - puntoA)) / (
/tmp/ipykernel_30164/2798489867.py:1: RuntimeWarning: invalid value encount
ered in double_scalars
  f = lambda x : x * np.exp(x) - 7
```

Out[ ]:

```
(-inf, nan, 3)
```