Introducción a python para computación numérica I

Utilizando unos ejemplos, vamos a repasar varios conceptos python e introducir módulos específicos de computación numérica:

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

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

Vectorización

Definimos la función $f(x) = e^x$ como función **lambda**

```
f = lambda x: np.exp(x)
```

Y un vector que contiene 300000 de elementos equiespaciados en el intervalo $[-10, 10]$

```
x = np.linspace(-10,10,300000)
```

Vamos a ver tres formas de construir un vector que contenga los valores de la función `f` en los puntos de `x` .

Recordemos la sintaxis de un bucle **for**

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

Empezamos con un vector de dimensión cero y vamos a ir añadiéndole términos con **`np.append`** . Medimos el tiempo que tarda en rellenar el vector usando la diferencia entre el tiempo de máquina al principio y al final del bucle y lo almacenamos en **`t1`**

```
y = np.array([])

t = time.time()
for i in range(len(x)):
    y = np.append(y,f(x[i]))
t1 = time.time() - t
```

Ahora vamos a reservar espacio de memoria creando un vector de ceros de la misma forma y dimensión que **`x`** y rellenamos los elementos uno a uno

```
y = np.zeros_like(x)

t = time.time()
for i in range(len(x)):
    y[i] = f(x[i])
t2 = time.time() - t
```

Y, finalmente, usamos la capacidad de **vectorización** del `numpy` : sus funciones (y la función **lambda** lo es) pueden trabajar en bloque para arrays numpy.

```
t = time.time()
y = f(x)
t3 = time.time() - t
```

Comparamos los tiempos de ejecución

```
print('Vector creciendo dentro del bucle: ', t1,' segundos')
print('Sin vectorización:                 ', t2,' segundos')
print('Con vectorización:                 ', t3,' segundos')
print('t1/t2 = ', int(t1/t2))
print('t2/t3 = ', int(t2/t3))
print('t1/t3 = ', int(t1/t3))
```

```
Vector creciendo dentro del bucle:  36.80423879623413  segundos
Sin vectorización:                  0.48811888694763184  segundos
Con vectorización:                  0.0024828910827636 72  segundos
t1/t2 =  75
t2/t3 =  196
t1/t3 =  14823
```

Vemos que hemos hemos ejecutado las opciones de peor a mejor. Por lo tanto, evitaremos, en la medida de lo posible, hacer crecer un vector dentro de un bucle y usaremos vectorización.

Gráfica de una función de una variable

La función `plot` del módulo `matplotlib.pyplot` dibuja la función aproximada. Sustituye el dibujo de la función por el dibujo de una poligonal de puntos de la función. Veamos un ejemplo:

Definimos la función a dibujar $f(x) = e^x$

```
f = lambda x: np.exp(x)
```

Decidimos el intervalo $[a, b]$ en el que vamos a crear la función

```
a = -1.; b = 1.
```

Creamos una malla de 5 puntos equiespaciados en el intervalo $[a, b]$ que se almacena en un array numpy

```
x = np.linspace(a,b,5)
print(x)
```

```
[-1.  -0.5  0.   0.5  1. ]
```

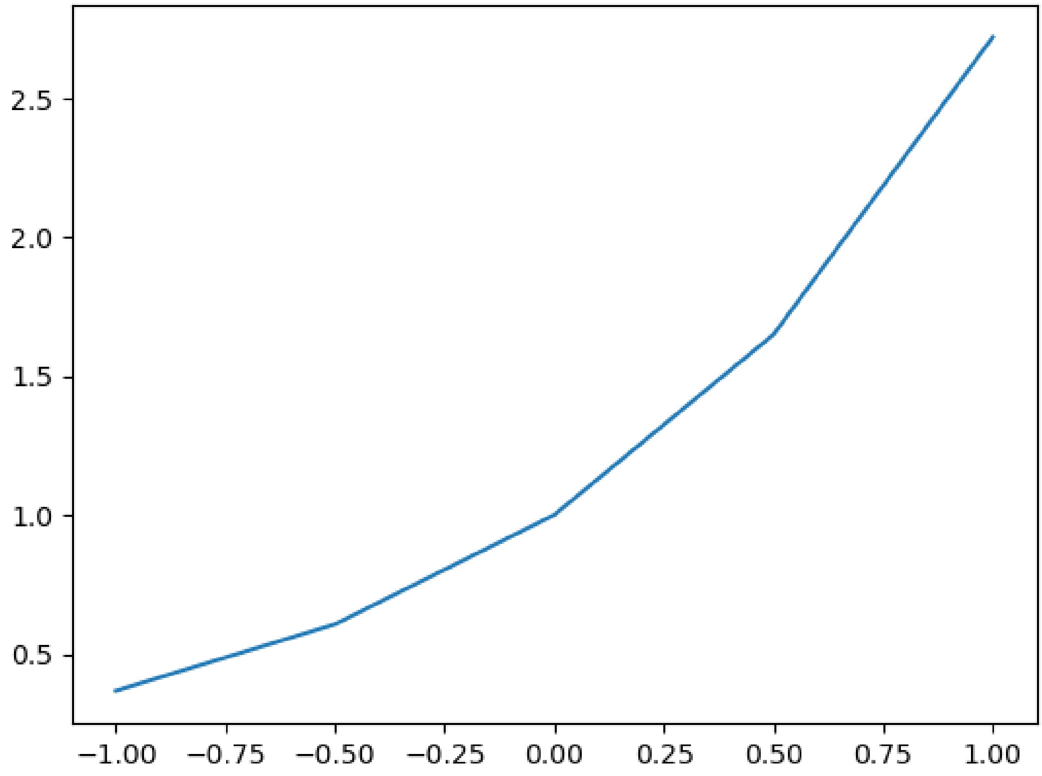
Obtenemos el array numpy correspondiente a las ys

```
y = f(x)
print(y)
```

```
[0.36787944 0.60653066 1.         1.64872127 2.71828183]
```

Dibujamos la función

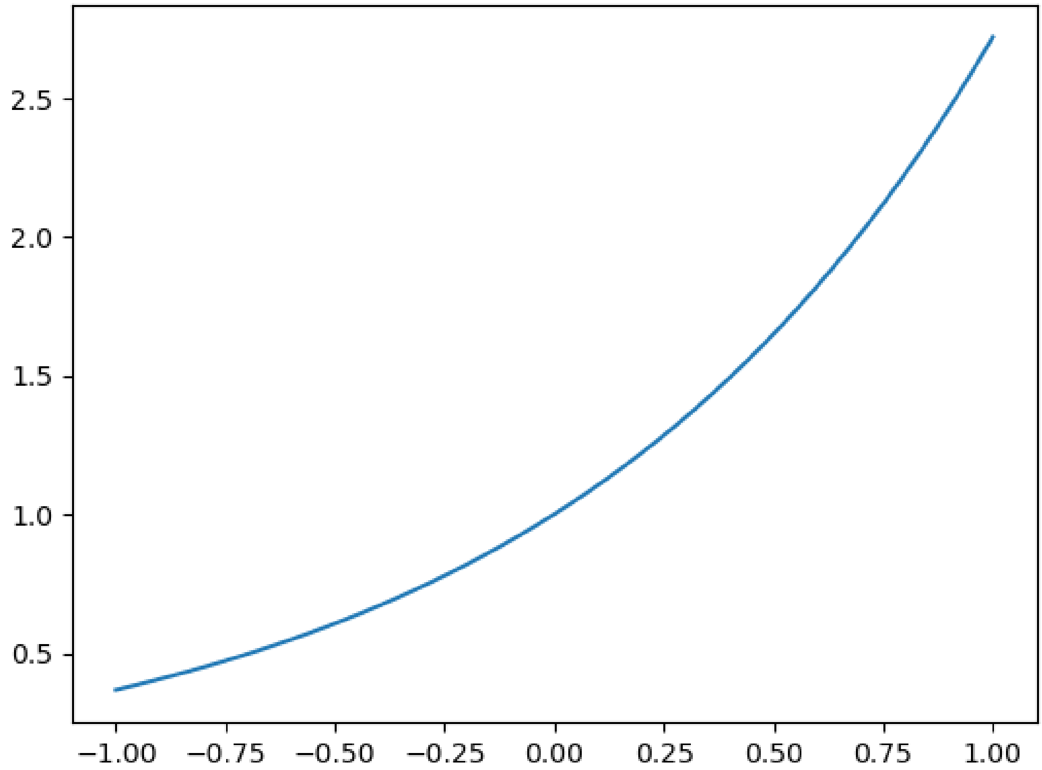
```
plt.figure()
plt.plot(x,y)
plt.show()
```



Con solo 5 puntos la aproximación es muy basta. Por defecto, `linspace` crea una malla de 50 puntos. Repitamos el proceso con 50 puntos.

```
x = np.linspace(a,b)
y = f(x)

plt.figure()
plt.plot(x,y)
plt.show()
```

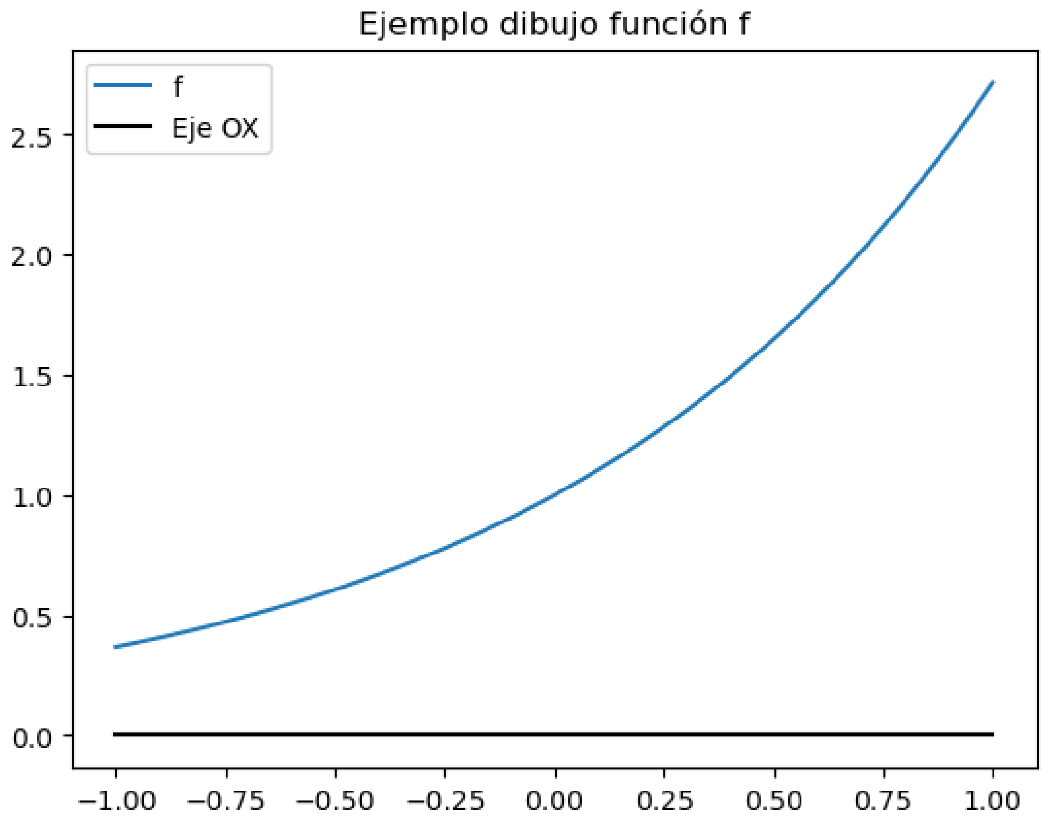


Sigue siendo una poligonal, pero ahora el mallado es lo suficiente fino para que dé la impresión de ser una curva suave.

Añadamos algunos detalles al dibujo:

```
OX = 0*x                             # array numpy con 50 ceros. Para dibujar el
eje OX

plt.figure()
plt.plot(x,y, label = 'f')
plt.plot(x,OX,'k', label = 'Eje OX')  # k, de "black" para que nos dibuje el eje O
X en negro
plt.title('Ejemplo dibujo función f') # usando los "label", añade una leyenda
plt.legend()
plt.show()
```



Polinomio de Taylor

Si desarrollamos en serie de McLaurin (polinomio de Taylor centrado en $x_0 = 0$) la función $f(x) = e^x$ tenemos

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

y, teóricamente, añadiendo suficientes términos podemos aproximar la función tanto como queramos. De hecho cada uno de los polinomios

$$\begin{array}{rcl} P_0(x) & = & 1 \\ P_1(x) & = & 1+\dfrac{x^1}{1!} \\ P_2(x) & = & 1+\dfrac{x^1}{1!}+\dfrac{x^2}{2!} \\ P_3(x) & = & 1+\dfrac{x^1}{1!}+\dfrac{x^2}{2!}+\dfrac{x^3}{3!} \\ P_4(x) & = & 1+\dfrac{x^1}{1!}+\dfrac{x^2}{2!}+\dfrac{x^3}{3!}+\dfrac{x^4}{4!} \\ & \cdots & \end{array}$$

aproxima mejor la función que el anterior. Calculemos el valor del polinomio de grado 3 en $x_0 = 0.5$. Para ello recordemos que $x_0^0 = 1$ y que $0! = 1$

```
x0 = 0.5
polinomio = 0.
factorial = 1.

for i in range(4):
    sumando = x0**i/factorial
    polinomio += sumando
    factorial *= i+1


print('P3(0.5)      = ', polinomio)
print('np.exp(0.5) = ', np.exp(x0))
```

```
P3(0.5)      =  1.6458333333333333
np.exp(0.5) =  1.6487212707001282
```

Si convertimos el programa en función y probamos un polinomio de grado mayor

```
def P(x0,grado):
    polinomio = 0.
    factorial = 1.

    for i in range(grado+1):
        sumando = x0**i/factorial
        polinomio += sumando
        factorial *= i+1

    return polinomio
```

```
print('P(0.5, 10)  = ', P(0.5, 10))
print('np.exp(0.5) = ', np.exp(0.5))
```

```
P(0.5, 10)  =  1.6487212706873655
np.exp(0.5) =  1.6487212707001282
```

Si ahora introducimos un array numpy, la salida es también un array numpy

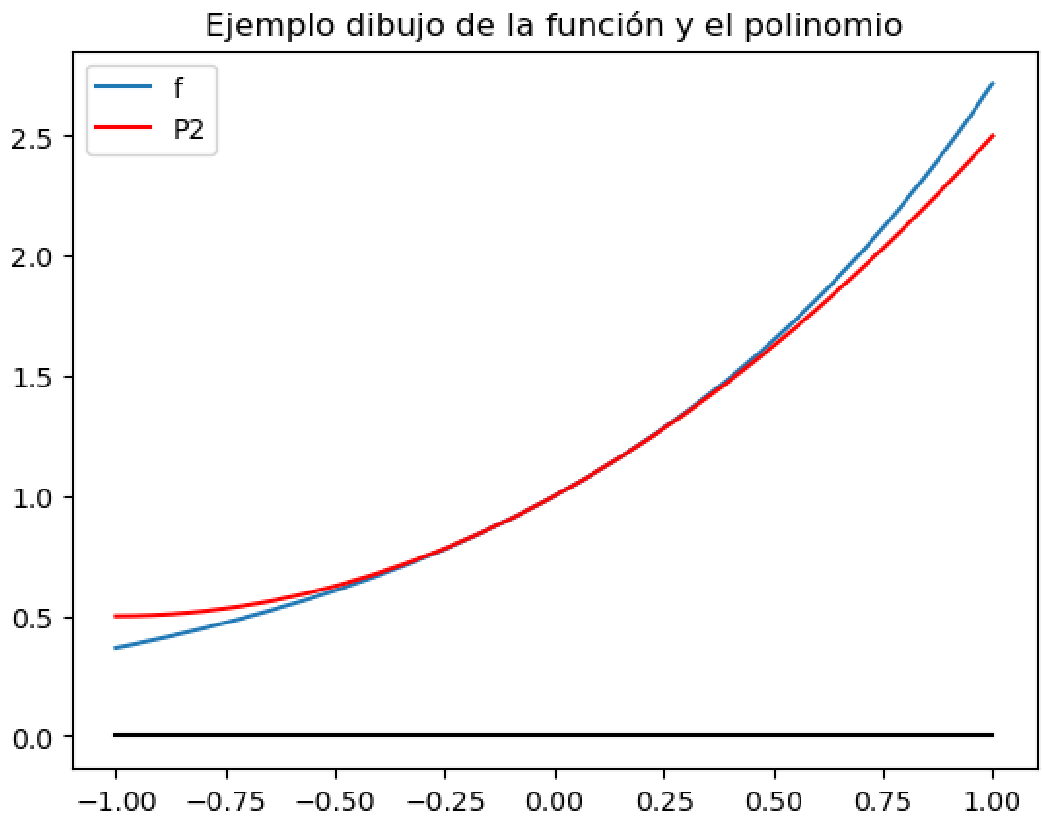
```
a = -1.; b = 1.
x = np.linspace(a,b,5)
print('x          = ', x)
print('P(x, 10)   = ', P(x, 10))
print('np.exp(x) = ', np.exp(x))
```

```
x          =  [-1.  -0.5  0.   0.5  1. ]
P(x, 10)   =  [0.36787946 0.60653066 1.         1.64872127 2.7182818
]
np.exp(x) =  [0.36787944 0.60653066 1.         1.64872127 2.7182818
3]
```

Si en lugar de 5 puntos, hacemos esto con 50 puntos, podremos dibujar el polinomio. Dibujemos el polinomio de grado 2 con línea roja, junto con la función.

```
a = -1.; b = 1.
f = lambda x: np.exp(x)
x = np.linspace(a,b)
OX = 0*x

plt.figure()
plt.plot(x,f(x), label = 'f')
plt.plot(x,OX,'k')
plt.plot(x,P(x,2),'r', label = 'P2')
plt.title('Ejemplo dibujo de la función y el polinomio')
plt.legend()
plt.show()
```



Y ahora, cambiemos el intervalo a $[-3,3]$ y dibujemos, usando un bucle, varios polinomios

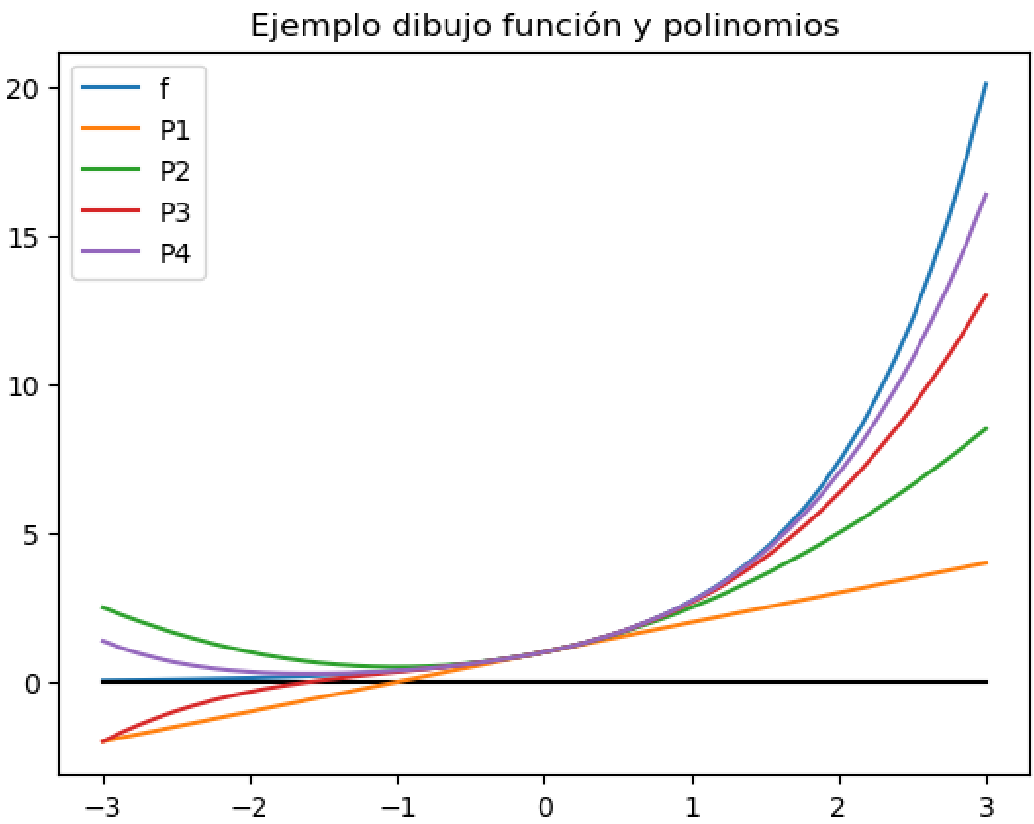
```
a = -3.; b = 3.
f = lambda x: np.exp(x)
x = np.linspace(a,b)
y = f(x)

OX = 0*x

plt.figure()
plt.plot(x,y, label = 'f')
plt.plot(x,OX,'k')

for grado in range(1,5):
    plt.plot(x,P(x,grado), label = 'P'+str(grado))

plt.title('Ejemplo dibujo función y polinomios')
plt.legend()
plt.show()
```



Añadiendo una línea (`plt.pause(1)`) podemos hacer una pequeña animación por pantalla usando **spyder**. Para ello:

- Abrir **spyder**.
- En el menú:
  - *Español:* Herramientas → Preferencias → Terminal IPython → Gráficas → Salida Gráfica → Salida: → Automática.
  - *Inglés:* Tools → Preferences → IPython Console → Graphics → Graphics backend → Backend: → Automatic.
- Reiniciamos **spyder**.
- Copiamos y ejecutamos el código de debajo.

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

#%%-----------------------------------------
def P(x0,grado):
    polinomio = 0.
    factorial = 1.

    for i in range(grado+1):
        sumando = x0**i/factorial
        polinomio += sumando
        factorial *= i+1

    return polinomio

#%%-----------------------------------------

f = lambda x: np.exp(x)
a = -3.; b = 3.
x = np.linspace(a,b)
y = f(x)

OX = 0*x

plt.figure()

plt.plot(x,y, label = 'f')
plt.plot(x,OX,'k')

for grado in range(1,7):
    plt.plot(x,P(x,grado), label = 'P'+str(grado))
    plt.title('Ejemplo dibujo de la función y los polinomios')
    plt.legend()
    plt.pause(1)

plt.show()
```

Otra estructura de control es **while**

```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

Ejercicio 1

Utilizando un bucle **while**, escribir un programa, que utilizando el desarrollo de McLaurin para la función $f(x) = e^x$, calcule su valor aproximado en el punto `x0 = -0.4` . El criterio de parada será que el valor del último sumando añadido **en valor absoluto** es menor que una tolerancia `tol=1.e-8` y que el número máximo de sumandos es 100, es decir `maxNumSum=100` . Comparar su valor con el valor de la función **lambda** f. Dar el número de sumandos utilizados.

Nota:

- Usar algún operador lógico **and**, **or**, **not**, para la condición del **while**.
- El valor absoluto de un número real `a` puede obtenerse con `np.abs(a)`
- Recordamos que el desarrollo de McLaurin para la función $f(x) = e^x$ es

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

```
%run Ejercicio1.py
```

```
Valor de la función en -0.4       = 0.6703200460356393
Valor de la aproximación en -0.4 = 0.67032004600776
Número de iteraciones            = 10
```

Ejercicio 2

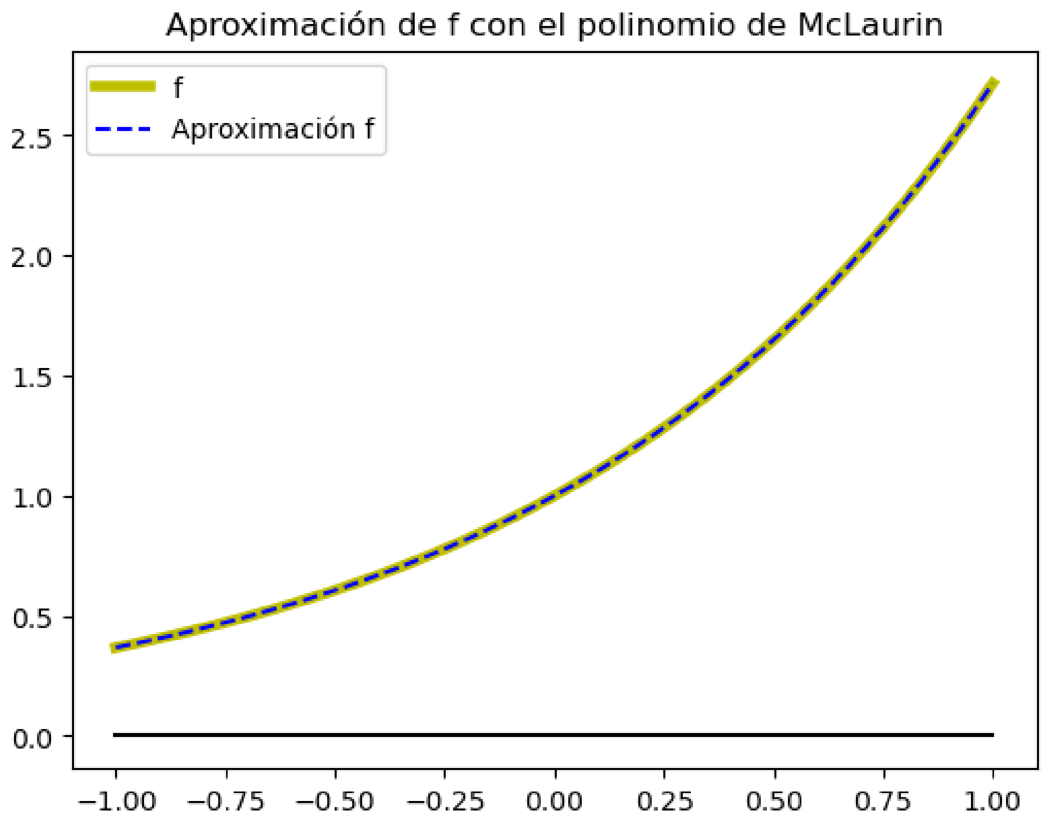
Modificar el programa de forma que calcule simultáneamente la aproximación de la función de los 50 valores contenidos en `x = np.linspace(-1,1)` . El criterio de parada será ahora que, el máximo del último sumando en valor absoluto (ahora tenemos 50 x_0 distintas y 50 últimos sumandos distintos), sea menor que que una tolerancia `tol=1.e-8` y que el número máximo de sumandos es 100, `maxNumSum=100` .

Convertir este programa en una función `funExp(x, tol, maxNumSum)` cuyos argumentos de entrada sean el array numpy `x` que contiene los 50 valores, `tol` y `maxNumSum` y cuyo argumento de salida sea un array numpy `y` que contenga los valores aproximados de la función obtenidos con el polinomio de McLaurin. Utilizar esta función para dibujar la función. Dibujar también la función utilizando la función **lambda** f definida a partir de `np.exp(x)`

Notas:

- `np.max(np.abs(sumando))` nos da el valor máximo de los valores absolutos de los valores contenidos en un array numpy llamado `sumando` .
- Para dibujar una línea gruesa amarilla `plt.plot(x,y,'y', linewidth = 4)`
- Para dibujar una línea azul discontinua `plt.plot(x,y,'b--')`

```
%run Ejercicio2.py
```



Ejercicios propuestos

Ejercicio 3

Utilizando un bucle **while**, escribir un programa, que utilizando el desarrollo de McLaurin para la función $f(x) = \operatorname{sen}(x)$, calcule su valor aproximado en el punto `x0 = np.pi/4` . El criterio de parada será que el valor del último sumando añadido en valor absoluto es menor que una tolerancia `tol=1.e-8` y que el número máximo de sumandos es 100, es decir `maxNumSum=100` . Comparar su valor con el valor de la función **lambda** f. Dar el número de sumandos utilizados.

Nota:

- Usar algún operador lógico **and**, **or**, **not**, para la condición del **while**.
- Recordamos que el desarrollo de McLaurin para la función $f(x) = \operatorname{sen}(x)$ es

$$\operatorname{sen}(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
%run Ejercicio3.py
```

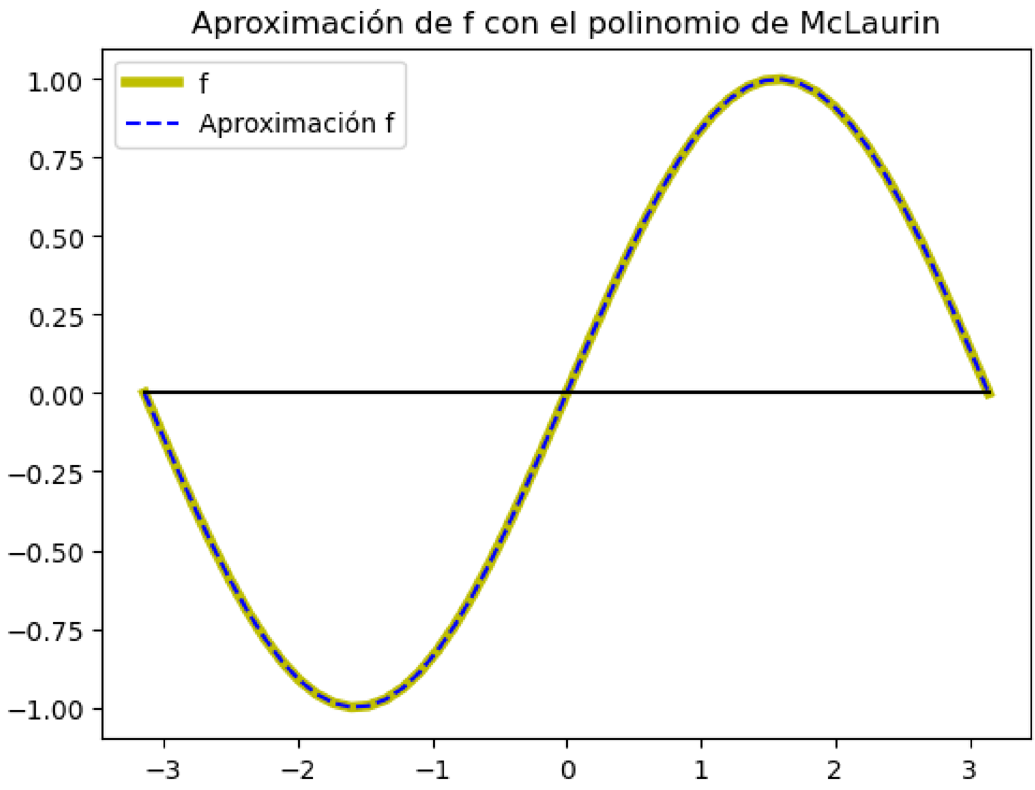
```
Valor aprox            =  0.7071067811796194
Valor exacto           =  0.7071067811865475
Número de iteraciones =  6
```

Ejercicio 4

Modificar el programa del ejercicio anterior de forma que calcule simultáneamente la aproximación de la función de los 50 valores contenidos en `x = np.linspace(-np.pi,np.pi)` . El criterio de parada será ahora que, el máximo del último sumando en valor absoluto (ahora tenemos 50 x_0 distintas y 50 últimos sumandos distintos), sea menor que que una tolerancia `tol=1.e-8` y que el número máximo de sumandos es 100, `maxNumSum=100` .

Convertir este programa en una función `funSin(x, tol, maxNumSum)` cuyos argumentos de entrada sean el array numpy `x` que contiene los 50 valores, `tol` y `maxNumSum` y cuyo argumento de salida sea un array numpy `y` que contenga los valores aproximados de la función obtenidos con el polinomio de McLaurin. Utilizar esta función para dibujar la función. Dibujar también la función utilizando la función **lambda** f definida a partir de `np.sin(x)`

```
%run Ejercicio4.py
```



Ejercicio 5

Teniendo en cuenta que el desarrollo de McLaurin de la función seno hiperbólico es la serie

$$\sinh x = \frac{x^1}{1!} + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

el coseno hiperbólico

$$\cosh x = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} + \cdots$$

y que la tangente hiperbólica es

$$\tanh = \frac{\sinh x}{\cosh x}$$

calcular la tangente hiperbólica de $x_0 = 0.5$ a partir de los desarrollos del senh y el cosh, usando el mismo número de términos (sumandos en el numerador y en el denominador). En el paso uno, usa un término para cada desarrollo, en el paso dos, dos sumandos y así sucesivamente. Parar cuando la diferencia en valor absoluto entre dos aproximaciones sucesivas de la tangente hiperbólica sea menor que 10^{-4}.

```
%run Ejercicio5.py
```

```
Valor aprox  =  0.4621171922898218
Valor exacto =  0.46211715726000974
```

Ejercicio 6

Modificar el programa del ejercicio anterior de forma que calcule simultáneamente la aproximación de la función de los 50 valores contenidos en **`x = np.linspace(-3,3)`** . El criterio de parada será ahora que, la máxima diferencia entre dos iteraciones consecutivas para todos los puntos (ahora tenemos 50 puntos distintos), sea menor que que una tolerancia **`tol=1.e-8`** .

Convertir este programa en una función **`funTanh(x, tol)`** cuyos argumentos de entrada sean el array numpy **`x`** que contiene los 50 valores, **`tol`** y cuyo argumento de salida sea un array numpy **`y`** que contenga los valores aproximados de la función obtenidos con el polinomio de McLaurin. Utilizar esta función para dibujar la función. Dibujar también la función utilizando la función **lambda** f definida a partir de **`np.tanh(x)`**

```
%run Ejercicio6.py
```