

# HW02: Numerical approximation and estimation errors (exercises).

```
In [2]: #from google.colab import files
#uploaded = files.upload()
```

Authors:

- alberto.suarez@uam.es
- Student 1: Gorka Crespo Bravo
- Student 2: Miguel Cuesta Altable
- Student 3: Antón Salvadores Muñiz

```
In [4]: from importlib import reload
import math
import numpy as np
import matplotlib.pyplot as plt
import Anton_Gorka_Miguel_HW_02_functions as func
reload(func)
import tools_qfb as qf
import pandas as pd
from numpy.random import default_rng
from scipy.stats import norm
from scipy.integrate import quad

from typing import Callable, Tuple
```



## Numerical errors

### Error types

Numerical computations are affected by different types of errors:

1. **Rounding errors** are a consequence of the fact that a computer represents real numbers with only finite precision.
2. **Truncation errors** are a consequence of the fact that it is not possible to make numerical computations that involve limits (e.g. derivatives, integrals).
3. **Fluctuation errors** appear in the estimation of expected values from finite samples.

### Intervals and errors.

Numbers affected by errors can be interpreted as intervals.

Assume that we have made an estimate  $x$  of the quantity  $x^{(true)}$ . The actual (signed) absolute error is

$$\text{error}_{\text{abs}} = x - x^{(true)}$$

. The relative error is

$$\text{error}_{\text{rel}} = \frac{x - x^{(\text{true})}}{x^{(\text{true})}}.$$

In most cases we have access only to  $\Delta x$ , an estimate of the absolute error. This error can be understood as the radius of an interval centered at  $x$  that contains the actual value of the quantity that is estimated

$$x - \Delta x \leq x^{(\text{true})} \leq x + \Delta x.$$

The following notation is commonly used

$$x^{(\text{true})} \approx x \pm \Delta x.$$

### Error propagation.

Errors can grow as the result of numerical operation and render the results of such computations meaningless.

Consider the following estimates, which are affected by independent errors:

$$x^{(\text{true})} \approx x \pm \Delta x, \quad y^{(\text{true})} \approx y \pm \Delta y, \quad \left\{ x_n^{(\text{true})} \approx x_n \pm \Delta x_n; n = 1, 2, \dots, N \right\}.$$

Using interval calculus, the following error propagation rules for  $z^{(\text{true})} \approx z \pm \Delta z$  can be derived:

$$z = x + y \quad \Delta z = \Delta x + \Delta y \quad (1)$$

$$z = x - y \quad \Delta z = \Delta x + \Delta y \quad (2)$$

Assuming that the errors are small, the following error propagation rules can be derived using Taylor expansions truncated to first order

$$z = x * y \quad \frac{\Delta z}{z} \approx \frac{\Delta x}{x} + \frac{\Delta y}{y} \quad (3)$$

$$z = \frac{x}{y} \quad \frac{\Delta z}{z} \approx \frac{\Delta x}{x} + \frac{\Delta y}{y} \quad (4)$$

$$z = f(\mathbf{x}) \quad \Delta z \approx \sum_{i=1}^N \left| \frac{\partial f(\mathbf{x})}{\partial x_i} \right| \Delta x_i, \quad (5)$$

where  $\mathbf{x}^T = (x_1, x_2, \dots, x_N)$ . The superscript  $T$  indicates matrix transposition.

### Exercise 1.

Consider the quantities

$$x_1 \approx 23.492 \pm 0.003 \quad (6)$$

$$x_2 \approx 23.6 \pm 0.4 \quad (7)$$

$$x_3 \approx 2.327 \pm 0.001 \quad (8)$$

$$x_4 \approx 0.000 \pm 0.001 \quad (9)$$

1. Use the above rules to calculate the absolute and the relative errors (as a percentage) of the following quantities

$$z_1 = x_1 + x_2 \quad (10)$$

$$z_2 = x_1 - x_2 \quad (11)$$

$$z_3 = \frac{x_3}{x_1 - x_2} \quad (12)$$

$$z_4 = \sin x_4 \quad (13)$$

$$z_5 = \cos x_4. \quad (14)$$

2. Write the results as  $z \pm \Delta z$ .

3. Comment the results obtained.

In [8]:

```
# Datos iniciales
x1, dx1 = 23.492, 0.003
x2, dx2 = 23.6, 0.4
x3, dx3 = 2.327, 0.001
x4, dx4 = 0.0, 0.001

#Errores relativos de los datos iniciales
rel_x1 = (dx1 / abs(x1)) * 100
rel_x2 = (dx2 / abs(x2)) * 100
rel_x3 = (dx3 / abs(x3)) * 100
rel_x4 = float('inf') if x4 == 0 else (dx4 / abs(x4)) * 100

# z1 = x1 + x2
z1 = x1 + x2
dz1 = dx1 + dx2
rel_z1 = (dz1 / abs(z1)) * 100

# z2 = x1 - x2
z2 = x1 - x2
dz2 = dx1 + dx2
rel_z2 = (dz2 / abs(z2)) * 100

# z3 = x3 / (x1 - x2)
z3 = x3 / z2
dz3_dx3 = 1 / z2
dz3_dx2 = x3/(x1-x2)**2
dz3_dx1 = -(x3/(x1-x2)**2)
dz3 = abs(dz3_dx3) * dx3 + abs(dz3_dx2) * dx2 + abs(dz3_dx1) * dx1
rel_z3 = (dz3 / abs(z3)) * 100

# z4 = sin(x4)
z4 = np.sin(x4)
dz4 = abs(np.cos(x4)) * dx4
rel_z4 = float('inf') if z4 == 0 else (dz4 / abs(z4)) * 100

# z5 = cos(x4)
z5 = np.cos(x4)
dz5 = abs(np.sin(x4)) * dx4
rel_z5 = (dz5 / abs(z5)) * 100

# Imprimir resultados
print(f"Absolute error x1: {dx1:.4f}")
print(f"Absolute error x2: {dx2:.4f}")
print(f"Absolute error x3: {dx3:.4f}")
print(f"Absolute error x4: {dx4:.4f}\n")
```

```

print(f"Relative error x1: {rel_x1:.4f}%")
print(f"Relative error x2: {rel_x2:.4f}%")
print(f"Relative error x3: {rel_x3:.4f}%")
print(f"Relative error x4: {rel_x4:.4f}\n")

print(f"z1 = {z1:.4f} ± {dz1:.4f} (Absolute error); Relative error: {rel_z1:.4f}")
print(f"z2 = {z2:.4f} ± {dz2:.4f} (Absolute error); Relative error: {rel_z2:.4f}")
print(f"z3 = {z3:.4f} ± {dz3:.4f} (Absolute error); Relative error: {rel_z3:.4f}")
print(f"z4 = {z4:.4f} ± {dz4:.4f} (Absolute error); Relative error: {rel_z4:.4f}")
print(f"z5 = {z5:.4f} ± {dz5:.4f} (Absolute error); Relative error: {rel_z5:.4f}")

```

Absolute error x1: 0.0030  
 Absolute error x2: 0.4000  
 Absolute error x3: 0.0010  
 Absolute error x4: 0.0010

Relative error x1: 0.0128%  
 Relative error x2: 1.6949%  
 Relative error x3: 0.0430%  
 Relative error x4: inf%

$z_1 = 47.0920 \pm 0.4030$  (Absolute error); Relative error: 0.8558%  
 $z_2 = -0.1080 \pm 0.4030$  (Absolute error); Relative error: 373.1481%  
 $z_3 = -21.5463 \pm 80.4089$  (Absolute error); Relative error: 373.1911%  
 $z_4 = 0.0000 \pm 0.0010$  (Absolute error); Relative error: inf%  
 $z_5 = 1.0000 \pm 0.0000$  (Absolute error); Relative error: 0.0000%

ANSWER:

Quantity	Value	Absolute error	Relative error	$z \pm \Delta z$
$x_1$	23.492	0.003	0.012770304784607525%	$23.492 \pm 0.003$
$x_2$	23.6	0.4	1.694915254237288%	$23.6 \pm 0.4$
$x_3$	2.327	0.001	0.04297378599054577%	$2.327 \pm 0.001$
$x_4$	0.000	0.001	∞%	$0.000 \pm 0.001$
$x_1 + x_2$	47.092	0.403	0.855771681%	$47.092 \pm 0.403$
$x_1 - x_2$	0.108	0.403	373.1481481%	$0.108 \pm 0.403$
$\frac{x_3}{x_1 - x_2}$	-21.5462963	80.4088649	373.1911219%	$-21.5462963 \pm 80.4088649$
$\sin x_4$	0.0	0.001	∞%	$0.0 \pm 0.001$
$\cos x_4$	1.0	0.0	0.0%	$1.0 \pm 0.0$

Asumiendo que los errores son pequeños, como podemos comprobar dados los valores de los errores absolutos y relativos de los números empleados para los cálculos, calculamos la propagación de errores empleando una expansión de Taylor de primer orden para las operaciones deseadas. Dos puntos importantes a comentar sobre los errores de nuestros datos son que, dado el valor verdadero 0 ( $x_4$ ), no tiene sentido hablar de error relativo, ya que no está definido (no se puede dividir entre 0), de ahí el resultado en la tabla. Sobre el error del dato  $x_2$ , se puede ver que realmente tiene un

elevado error, tanto absoluto como relativo, en comparación con los demás, por tanto, no parece la mejor idea para el cálculo de la propagación de errores una aproximación de Taylor de primer orden, ya que tendríamos términos significativos que no estamos teniendo en consideración. Es por esto que los errores de  $z_2 = x_1 - x_2$  y  $z_3 = \frac{x_3}{x_1 - x_2}$  son tan elevados, pues la manera de calcular el error no es la correcta. Esto se podría solucionar empleando un cálculo de la propagación de errores realizando aproximaciones de Taylor de mayor orden, por ejemplo. Aunque no de manera tan relevante, es culpa de esto por lo que también el error de  $z_1 = x_1 + x_2$  es más elevado de lo esperado, pues como se puede comprobar el valor entregado es el correcto. La diferencia entre  $z_1$  y  $z_2$  es que en  $z_2$  al efectuar la diferencia entre términos tan semejantes, tenemos un resultado cercano a 0, lo que hace crecer el error relativo en cantidades astronómicas. Para el cálculo del error de  $z_4 = \sin(x_4)$  el único problema es el mismo que el comentado con anterioridad sobre el valor verdadero 0 y el cálculo del error de  $z_5 = \cos(x_4)$  no tiene ningún problema de error elevado o valor verdadero cercano a 0, por lo que tenemos un buen resultado numérico.

## Taylor expansion

Consider a sufficiently smooth function  $f(x)$ , whose derivatives at  $x_0$  exist up to order  $K + 1$ , with the  $(K + 1)$ th derivative continuous at that point.

In the neighborhood of  $x_0$ , the function can be approximated by the (Taylor) series

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x_0) (x - x_0)^k = f(x_0) + f^{(1)}(x_0)(x - x_0) + \frac{1}{2!} f^{(2)}(x_0)(x - x_0)^2 + \dots$$

where

$$f^{(0)}(x_0) = f(x_0), \quad (16)$$

$$f^{(k)}(x_0) = \left. \frac{d^k f(x)}{dx^k} \right|_{x=x_0}, \quad k = 1, 2, \dots \quad (17)$$

The series converges for  $|x - x_0| < R$ , where  $R$  is the radius of convergence. The radius of convergence of the series is the distance from  $x_0$  to the closest singularity of the function in the *complex* plane.

In the neighborhood of  $x_0$ , the function can be approximated by the (Taylor) polynomial:

$$\begin{aligned} f(x) &= \sum_{k=0}^K \frac{1}{k!} f^{(k)}(x_0) (x - x_0)^k + E_K(x) \\ &= f(x_0) + f^{(1)}(x_0)(x - x_0) + \frac{1}{2!} f^{(2)}(x_0)(x - x_0)^2 + \dots + \frac{1}{K!} f^{(K)}(x_0)(x - x_0)^K \end{aligned}$$

The residual (error of the approximation) in the Lagrange form is

$$E_K(x) = \frac{1}{(K+1)!} f^{(K+1)}(\xi) (\xi - x_0)^{K+1},$$

for some  $\xi \in [x_0, x]$ .

Therefore, we say that the approximation error due to truncation of the series is of order  $\mathcal{O}\left((x - x_0)^{K+1}\right)$ .

## Exercise 2.

1. Provide a brief description of what each of the blocks of the previous code does.
  - Lines 1 to 7: Establece los argumentos de la función que se está definiendo:  $x$  son los puntos donde se quiere calcular la aproximación de Taylor, `function` y `function_derivate` son la función a evaluar y una función que computa su  $K$ -ésima derivada,  $K$  es el orden de la aproximación de Taylor, y  $x_0$  es el valor alrededor del cual se evalúa, que por defecto es 0.
  - Lines 8 to 20: Estas líneas son simplemente informativas, al estar entre comillas. Explican los argumentos de la función y lo que devuelve, un array de dimensión  $K$  en la que la fila  $k$ -ésima es la aproximación de Taylor de la función en los puntos  $x$  de orden  $k$ .
  - Lines 21 to 41 (especially, lines 25, 26): Son un ejemplo de utilización de la función. Al igual que en el caso anterior, no tienen relevancia en el código. Las líneas 25 y 26 definen los argumentos `function` y `function_derivate` que toma la función `taylor_approximation`. En este caso, se quiere evaluar la aproximación de Taylor para  $e^{-x}$  (correspondiente al argumento `function`) y se entrega la función de la derivada  $k$ -ésima para  $e^{-x}$  como:

$$f^{(k)}(x) = \begin{cases} e^{-x}, & \text{si } k \text{ par} \\ -e^{-x}, & \text{si } k \text{ impar} \end{cases}$$

Esto es así ya que, la parte relevante del código ( $k \% 2$ ) indica el resto de la división entre el entero  $k$  por 2, es decir, o 1 o 0. Como el código está escrito como  $(1 - 2 * (k \% 2)) * np.exp(-x)$ , el resultado es el descrito en la fórmula anterior, pues si  $k$  es impar, entonces  $k \% 2 = 1$  y si  $k$  es par,  $k \% 2 = 0$ , por lo que operando llegamos a esa expresión a trozos.

- Line 42: Crea un array (matriz) vacío de  $K + 1$  filas y tantas columnas como puntos en los que se quiere evaluar la función, es decir, la dimensión del argumento " $x$ ".
- Line 43: Asigna los valores de la primera fila de la matriz iguales a la función a evaluar en 0 ( $x_0$ ). Podría verse como la aproximación de Taylor de orden 0, que utiliza posteriormente para calcular las aproximaciones de orden K.
- Lines 47 to 51: Calcula la aproximación de orden  $k$  de la función de manera iterativa, a partir de la aproximación anterior y agregándole el término correspondiente a la derivada de orden  $k$ . Se almacena por filas.
- Line 52: Establece el output de la función `taylor_approximation`; una matriz en la que cada fila representa el orden de la aproximación de Taylor y cada columna el valor de la aproximación para cada uno de los puntos donde se evalúa.

2. There is a subtle error in the previous code. Can you identify and correct it? (TIP:

Running the results of the execution of the following cells can be useful to determine the origin of the error).

En el código original estaba escrito lo siguiente:

```
```
x -= x_0 # Center at x_0
for k in range(1, K+1):
    approximation[k, :] = (
        approximation[k-1, :] +
        function_derivative(x_0, k) * x**k / math.factorial(k)
    )
```
```

```

No nos parece la opción más adecuada, ya que estamos modificando el valor del array  $x$  sobre el que queremos calcular las aproximaciones de Taylor de la función, así que eliminamos esta parte del código y centramos nosotros mismos el factor de Taylor:

```
```
for k in range(1, K+1):
    approximation[k, :] = (
        approximation[k-1, :] +
        function_derivative(x_0, k) * (x-x_0)**k / math.factorial(k)
    )
```
```

```

Por otra parte, ya que esto es una corrección menor, pues graficando las funciones correspondientes a las aproximaciones de Taylor hasta orden 10 en el software GeoGebra (graficas\_taylor\_ej2.ggb), vemos que las aproximaciones realizadas por la función son las correctas, ya que las gráficas obtenidas son equivalentes. Es por ello que vamos a realizar los siguientes comentarios sobre los resultados y la función definida.

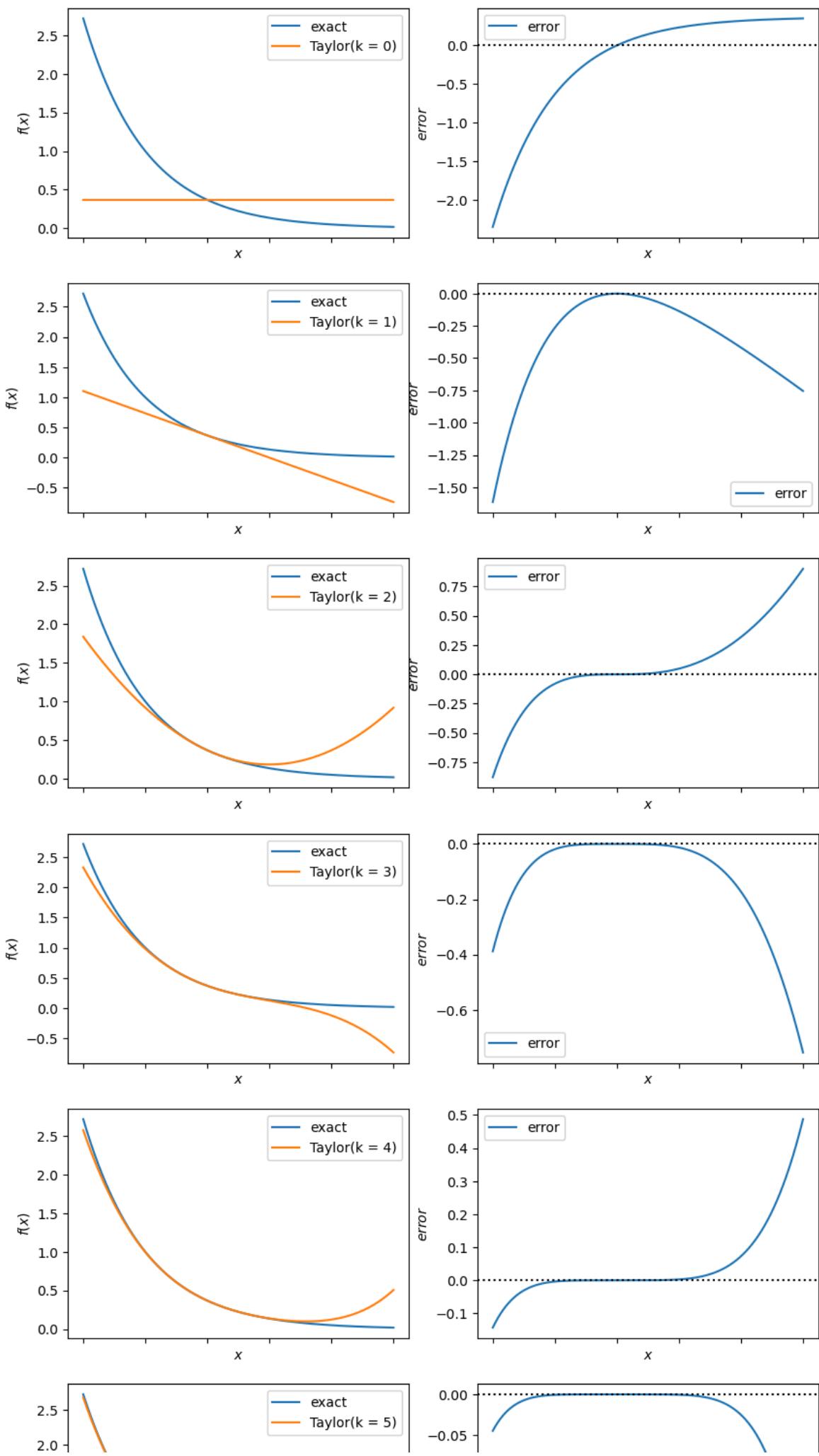
En primer lugar, sobre los errores cometidos por la aproximación, en las aproximaciones de orden par el error es siempre negativo y en las aproximaciones de orden impar, el error es tanto negativo como positivo, dependiendo de la posición del punto con respecto del centro  $x_0$ . Esto, de nuevo, son particularidades de la función que estamos tratando de aproximar, por lo que no parece muy interesante.

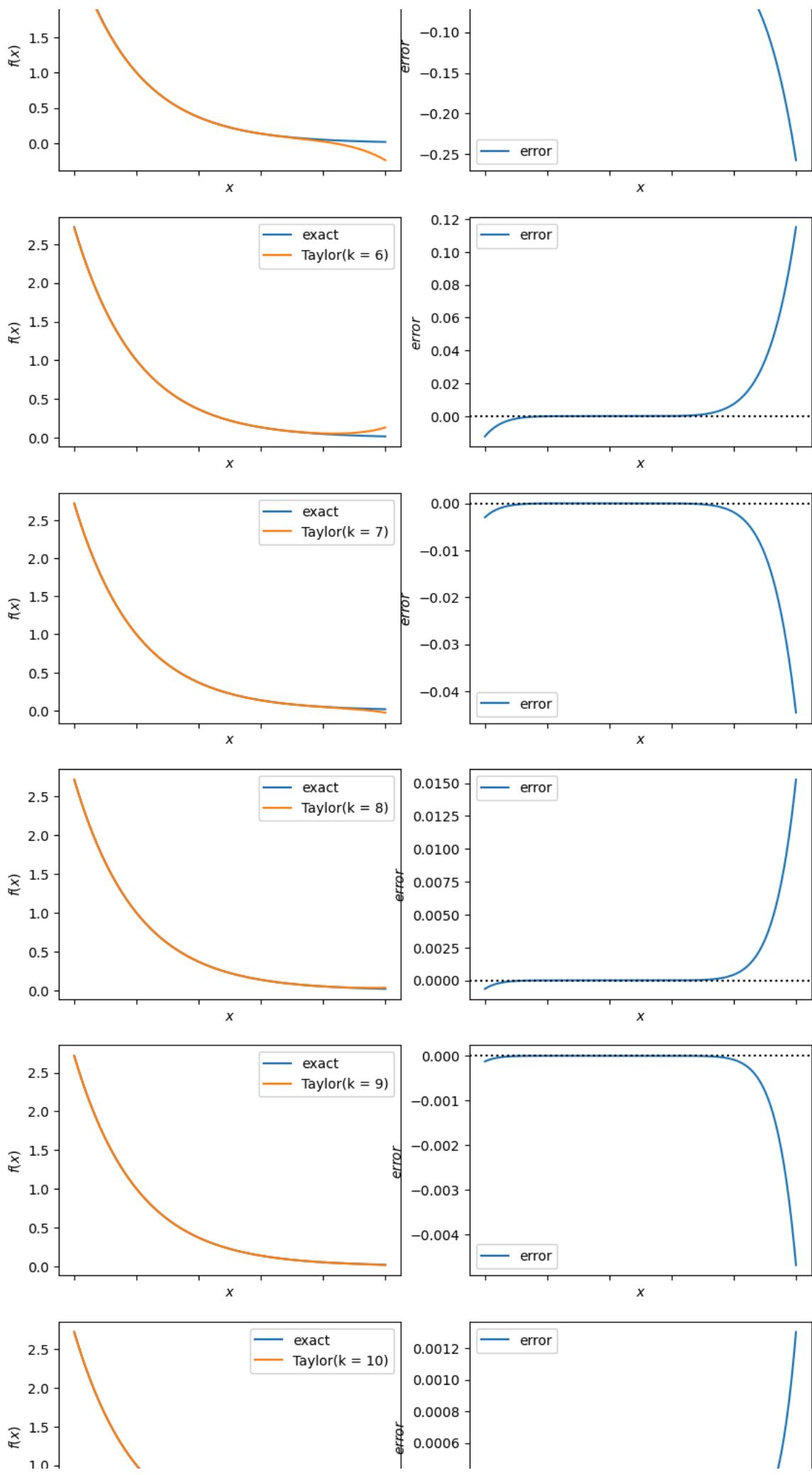
Como posibles mejoras de la función, nos parece algo redundante que un argumento de la propia función sea la función a aproximar, no tiene mucho sentido querer aproximar una función que ya conocemos su valor en todos los puntos. Además, para realizar la aproximación de Taylor, solo necesitamos el valor de la función en el punto  $x_0$  y el de sus  $k$ -ésimas derivadas en este punto también, por lo que nos parece más lógico solicitar en exclusiva esto para realizar la aproximación. En el ejemplo propuesto de la exponencial negativa, solo necesitaríamos  $f(x_0) = e^{-1}$  y  $f^k(x_0) = -e^{-1}$  si  $k$  es impar,  $f^k(x_0) = e^{-1}$  si  $k$  es par. Esto nos parece la solución más sensata, ya que nadie en principio va a querer aproximar una función cuando ya la conoce, lo interesante de

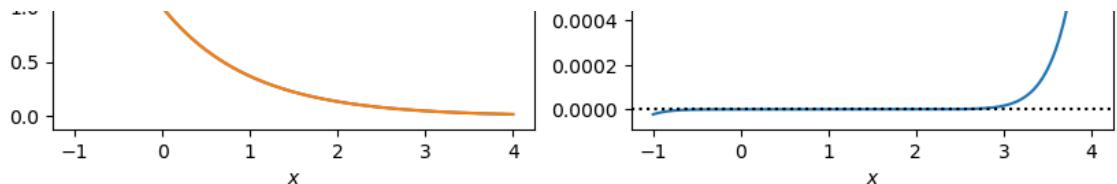
utilizar la aproximación de Taylor es cuando desconocemos su valor en los demás puntos.

```
In [13]: # Taylor approximation for the negative exponential
```

```
x_0 = 1.0
K = 10
func.plot_taylor_approximation(
    function=lambda x: np.exp(-x),
    function_derivative=lambda x, k: (1 - 2 * (k % 2)) * np.exp(-x),
    K=K,
    x_0=x_0,
    interval_plot=(x_0 - 2.0, x_0 + 3.0),
    figure_size=(K, 4*K),
)
```







### Exercise 3: Taylor expansion properties.

For each of these 4 functions

$$f_1(x) = \sin(x), \quad (20)$$

$$f_2(x) = \log(1+x), \quad (21)$$

$$f_3(x) = \frac{1}{1+x}, \quad (22)$$

$$f_4(x) = \frac{1}{1+x^2}. \quad (23)$$

- Provide an expression of the Taylor series around  $x_0 = 0.0$  of the form

$$f(x) = \sum_{k=0}^{\infty} c_k x^k.$$

In other words, find the expression of  $c_k$  in terms of  $k$  for each of the functions considered.

- Para  $f_1(x) = \sin(x), \forall n \in \mathbb{N} \cup \{0\}$  se tiene:

$$f_1^{(k)}(x) = \begin{cases} \sin(x), & \text{si } k = 4 * n \\ \cos(x), & \text{si } k = 4 * n + 1 \\ -\sin(x), & \text{si } k = 4 * n + 2 \\ -\cos(x), & \text{si } k = 4 * n + 3 \end{cases}$$

Por tanto, la expresión de  $c_k$  para  $f_1(x) = \sin(x)$ , teniendo en cuenta que  $x_0 = 0$ , es como sigue:

$$c_k = \begin{cases} \frac{\sin(0)}{k!}, & \text{si } k = 4 * n \\ \frac{\cos(0)}{k!}, & \text{si } k = 4 * n + 1 \\ \frac{-\sin(0)}{k!}, & \text{si } k = 4 * n + 2 \\ \frac{-\cos(0)}{k!}, & \text{si } k = 4 * n + 3 \end{cases} = \begin{cases} 0, & \text{si } k = 4 * n \\ \frac{1}{k!}, & \text{si } k = 4 * n + 1 \\ 0, & \text{si } k = 4 * n + 2 \\ \frac{-1}{k!}, & \text{si } k = 4 * n + 3 \end{cases}$$

- Para  $f_2(x) = \log(1+x), \forall n \in \mathbb{N}$  se tiene:

$$f_2^{(k)}(x) = \begin{cases} \frac{(k-1)!}{(1+x)^k}, & \text{si } k = 2 * n - 1 \\ \frac{-(k-1)!}{(1+x)^k}, & \text{si } k = 2 * n \end{cases}$$

Por tanto, la expresión de  $c_k$  para  $f_2(x) = \log(1+x)$ , teniendo en cuenta que  $x_0 = 0$ , es como sigue:

$$c_k = \begin{cases} \frac{1}{k}, & \text{si } k = 2 * n - 1 \\ \frac{-1}{k}, & \text{si } k = 2 * n \end{cases}$$

- Para  $f_3(x) = \frac{1}{1+x}$ ,  $\forall n \in \mathbb{N}$  se tiene:

$$f_3^{(k)}(x) = \begin{cases} \frac{-k!}{(1+x)^{(k+1)}}, & \text{si } k = 2 * n - 1 \\ \frac{k!}{(1+x)^{(k+1)}}, & \text{si } k = 2 * n \end{cases}$$

Por tanto, la expresión de  $c_k$  para  $f_3(x) = \frac{1}{1+x}$ , teniendo en cuenta que  $x_0 = 0$ , es como sigue:

$$c_k = \begin{cases} -1, & \text{si } k = 2 * n - 1 \\ 1, & \text{si } k = 2 * n \end{cases}$$

- Para  $f_4(x) = \frac{1}{1+x^2}$ , como parece un caso más complejo que los anteriores, veamos una a una las primeras derivadas de la función:

$$\begin{aligned} f_4^{(0)}(x) &= +\frac{1}{(1+x^2)^{0+1}} \\ f_4^{(1)}(x) &= -2 \cdot \frac{x}{(1+x^2)^{1+1}} \\ f_4^{(2)}(x) &= -2 \cdot \frac{3x^2 + 1}{(1+x^2)^{2+1}} \\ f_4^{(3)}(x) &= +24 \cdot \frac{-x^3 + x}{(1+x^2)^{3+1}} \\ f_4^{(4)}(x) &= +24 \cdot \frac{5x^4 - 10x^2 + 1}{(1+x^2)^{4+1}} \\ f_4^{(5)}(x) &= -720 \cdot \frac{3x^5 - 10x^3 + x}{(1+x^2)^{5+1}} \\ f_4^{(6)}(x) &= -720 \cdot \frac{-7x^6 + 36x^4 - 21x^2 + 1}{(1+x^2)^{6+1}} \end{aligned}$$

Como  $x_0 = 0$ , si evaluamos estas derivadas en  $x_0$ , que es lo que nos interesa para obtener la expresión de  $c_k$  correspondiente a  $f_4$ , observamos lo siguiente:

$$\begin{aligned} f_4^{(0)}(0) &= +1 = 0! \\ f_4^{(1)}(0) &= 0 \\ f_4^{(2)}(0) &= -2 = -2! \\ f_4^{(3)}(0) &= 0 \\ f_4^{(4)}(0) &= +24 = +4! \\ f_4^{(5)}(0) &= 0 \\ f_4^{(6)}(0) &= -720 = -6! \end{aligned}$$

Tras observar esto, se puede probar por inducción, aunque nosotros no lo haremos, pues no es el objetivo de este trabajo, que las derivadas evaluadas en  $x_0 = 0$  siguen el siguiente patrón,  $\forall n \in \mathbb{N} \cup \{0\}$ :

$$f_4^{(k)}(0) = \begin{cases} k!, & \text{si } k = 4 * n \\ 0, & \text{si } k = 4 * n + 1 \\ -k!, & \text{si } k = 4 * n + 2 \\ 0, & \text{si } k = 4 * n + 3 \end{cases}$$

Finalmente, la expresión de  $c_k$  para  $f_4(x) = \frac{1}{1+x^2}$ , es la siguiente:

$$c_k = \begin{cases} 1, & \text{si } k = 4 * n \\ 0, & \text{si } k = 4 * n + 1 \\ -1, & \text{si } k = 4 * n + 2 \\ 0, & \text{si } k = 4 * n + 3 \end{cases}$$

2. Determine the radius of convergence of each of these series by locating the singularity that is closest to  $x_0$  in the complex plane.

Primero, debemos entender qué es una singularidad, son los puntos en los que la función no está bien definida. Por tanto, el radio de convergencia  $R$  es la distancia más pequeña desde nuestro centro  $x_0$  hasta alguna singularidad  $s$ . Es decir,  $R = \|s - x_0\|$ , siendo  $s$  el punto más cercano a  $x_0$  en el que la función no está bien definida. Teniendo en cuenta que  $x_0 = 0$ , calculemos el radio de convergencia para cada serie.

- Para  $f_1 = \sin(x)$ , se tiene que la función está bien definida  $\forall x \in \mathbb{R}$ , por tanto su radio de convergencia  $R = \infty$ .
- Para  $f_2 = \log(1 + x)$ , se tiene que la función está bien definida  $\forall x \in (-1, +\infty)$ , por tanto, la singularidad más cercana a  $x_0 = 0$  es  $s = -1$ . Así, se tiene que el radio de convergencia para esta serie es  $R = \| -1 - 0 \| = 1$ .
- Para  $f_3 = \frac{1}{1+x}$ , se tiene que la función está bien definida  $\forall x \in \mathbb{R} \setminus \{-1\}$ , por tanto, la singularidad más cercana a  $x_0 = 0$  es  $s = -1$ . Así, se tiene que el radio de convergencia para esta serie es  $R = \| -1 - 0 \| = 1$ .
- Para  $f_4 = \frac{1}{1+x^2}$ , se tiene que la función está bien definida  $\forall x \in \mathbb{R}$ , sin embargo, al tratar de buscar singularidades en el plano complejo, si resolvemos la ecuación  $1 + x^2 = 0 \Leftrightarrow x = \sqrt{-1} = i$ , se tiene que la función no está definida para  $i$ , que tiene módulo 1. Por tanto, como la singularidad más cercana a  $x_0 = 0$  es  $s = i$ , de módulo 1, se tiene que el radio de convergencia para esta serie es  $R = \|i - 0\| = 1$ .

3. Make a plot similar to the one in the previous cell to illustrate the convergence of these approximations.

```
In [15]: # Definition of each c_k

# For f_1(x) = sin(x):

c_1 = lambda k: (0 if (k % 4 == 0 or k % 4 == 2) else
1 / math.factorial(k) if (k % 4 == 1) else -1 / math.factorial(k))

# For f_2(x) = log(1 + x)

c_2 = lambda k: 1 / k if (k % 2 == 1) else -1 / k

# For f_3(x) = 1 / (1 + x)

c_3 = lambda k: -1 if (k % 2 == 1) else 1

# For f_4(x) = 1 / (1 + x^2):

c_4 = lambda k: (0 if (k % 4 == 1 or k % 4 == 3) else
1 if (k % 4 == 0) else -1)

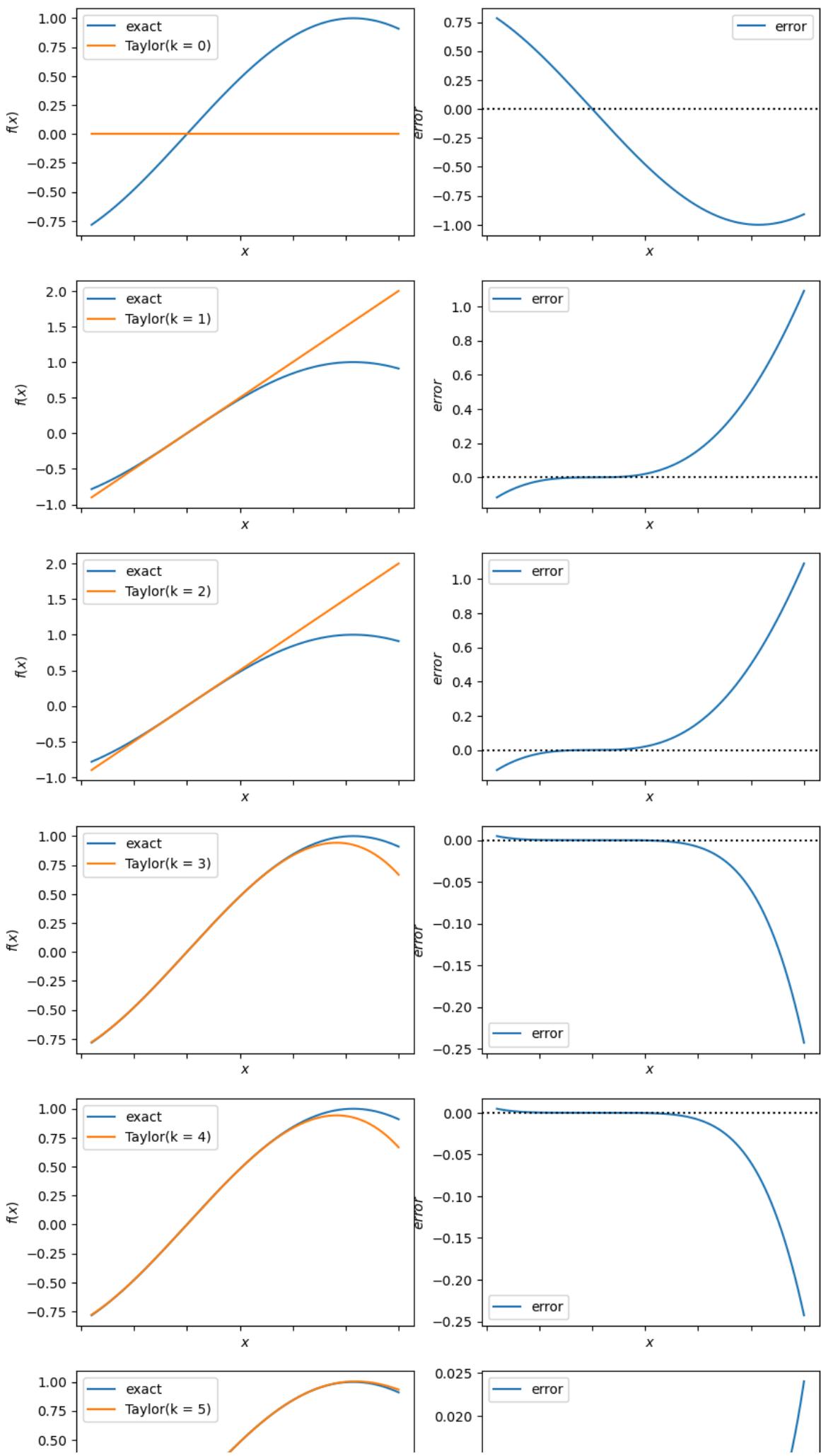
# Taylor approximation for f_1 = sin(x)

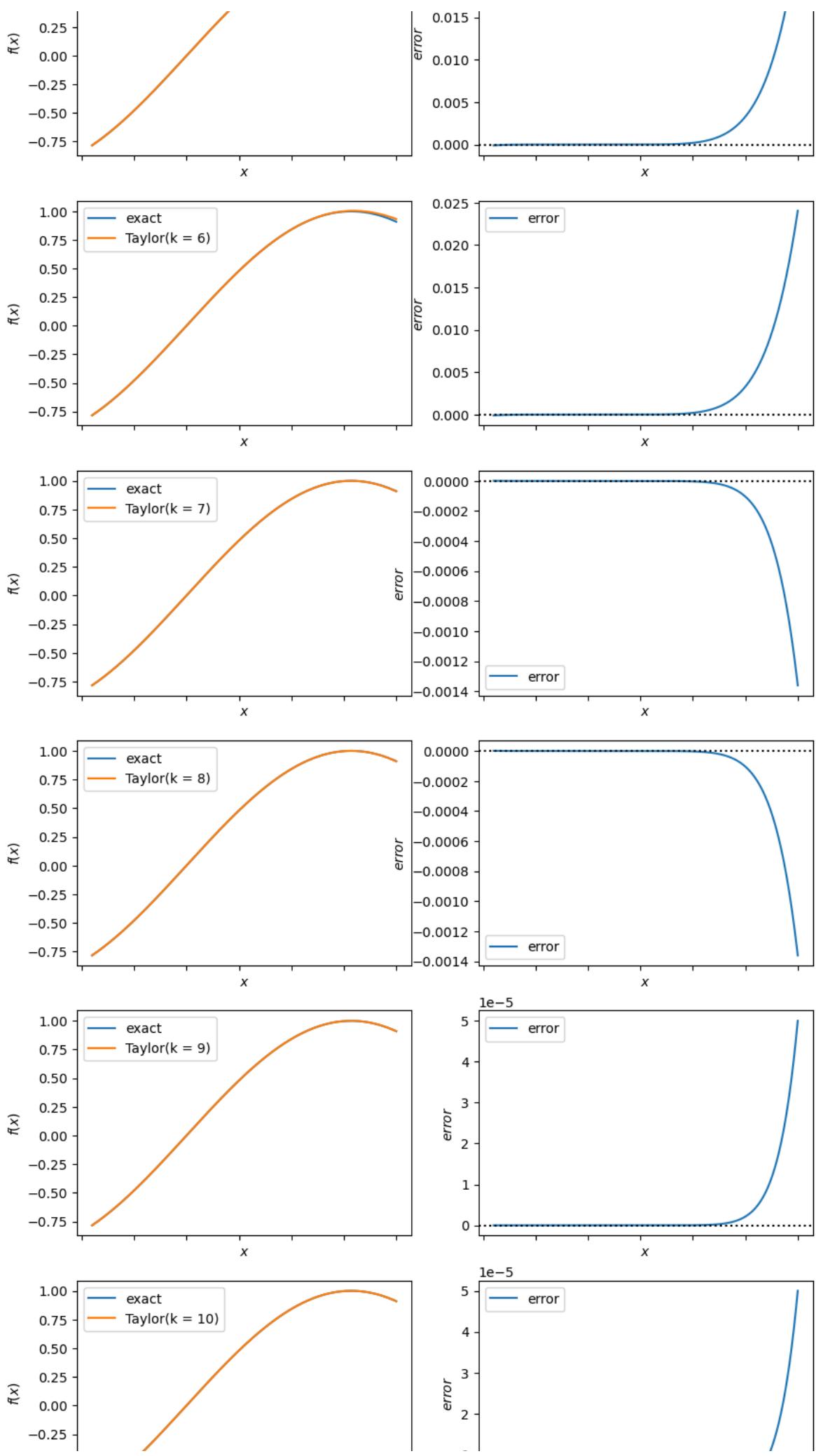
x_0 = 0.0
K = 10
func.plot_taylor_approximation_new(
    function=lambda x: np.sin(x),
    c=c_1,
    K=K,
    x_0=x_0,
    interval_plot=(x_0 - 0.9, x_0 + 2.0),
    figure_size=(K, 4*K),
)
# Taylor approximation for f_2 = log(1 + x)

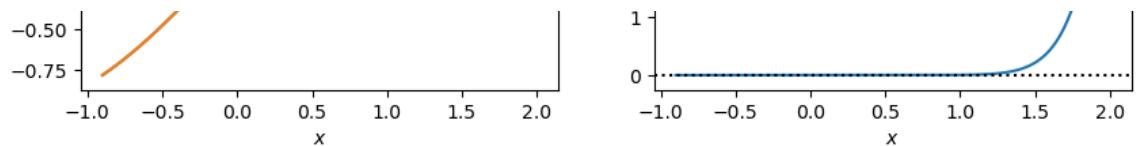
x_0 = 0.0
K = 10
func.plot_taylor_approximation_new(
    function=lambda x: np.log(1 + x),
    c=c_2,
    K=K,
    x_0=x_0,
    interval_plot=(x_0 - 0.9, x_0 + 2.0),
    figure_size=(K, 4*K),
)
# Taylor approximation for f_3 = 1 / (1 + x)

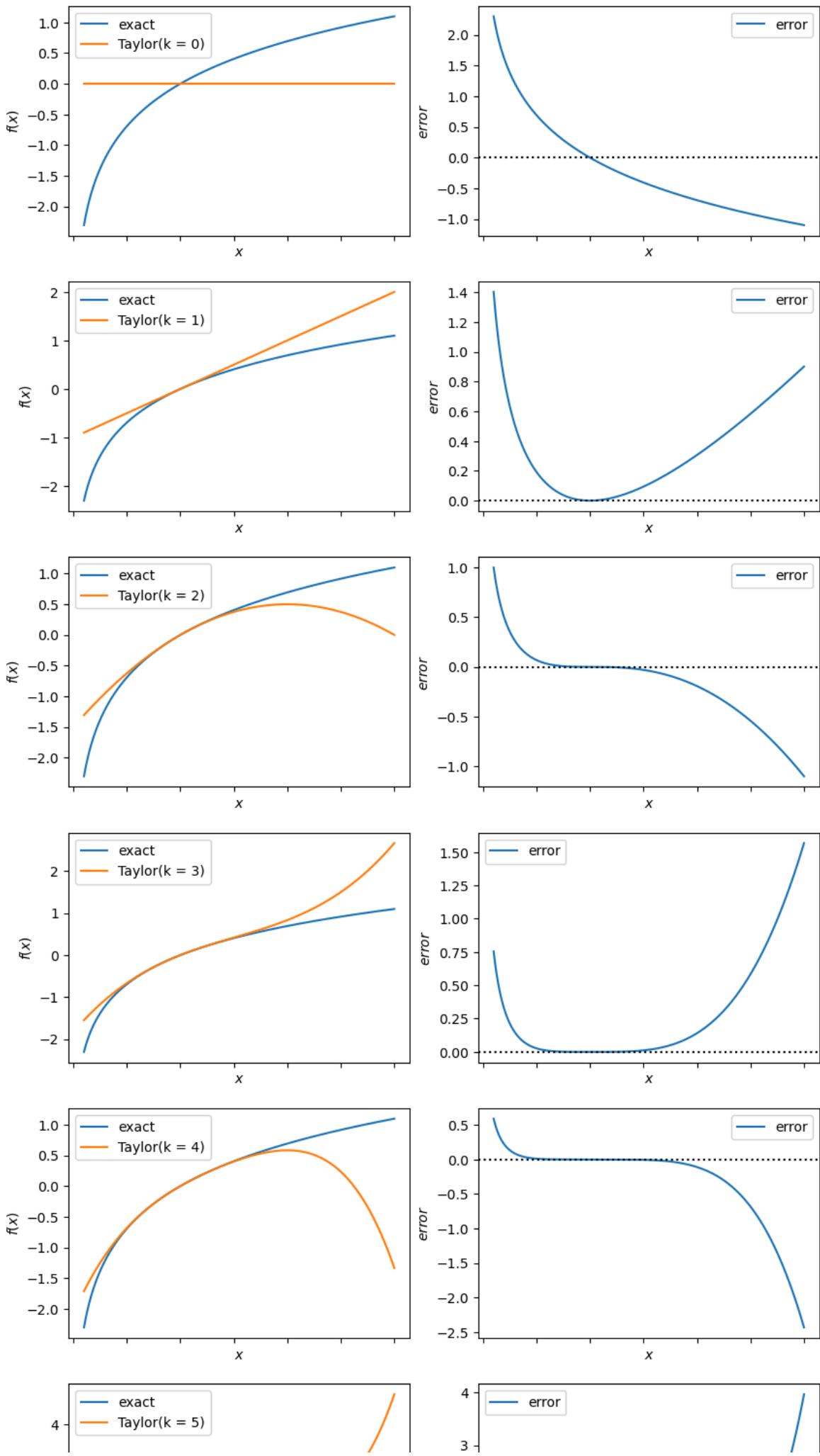
x_0 = 0.0
K = 10
func.plot_taylor_approximation_new(
    function=lambda x: 1 / (1 + x),
    c=c_3,
    K=K,
    x_0=x_0,
    interval_plot=(x_0 - 0.9, x_0 + 3.0),
    figure_size=(K, 4*K),
)
# Taylor approximation for f_4 = 1 / (1 + x^2)
```

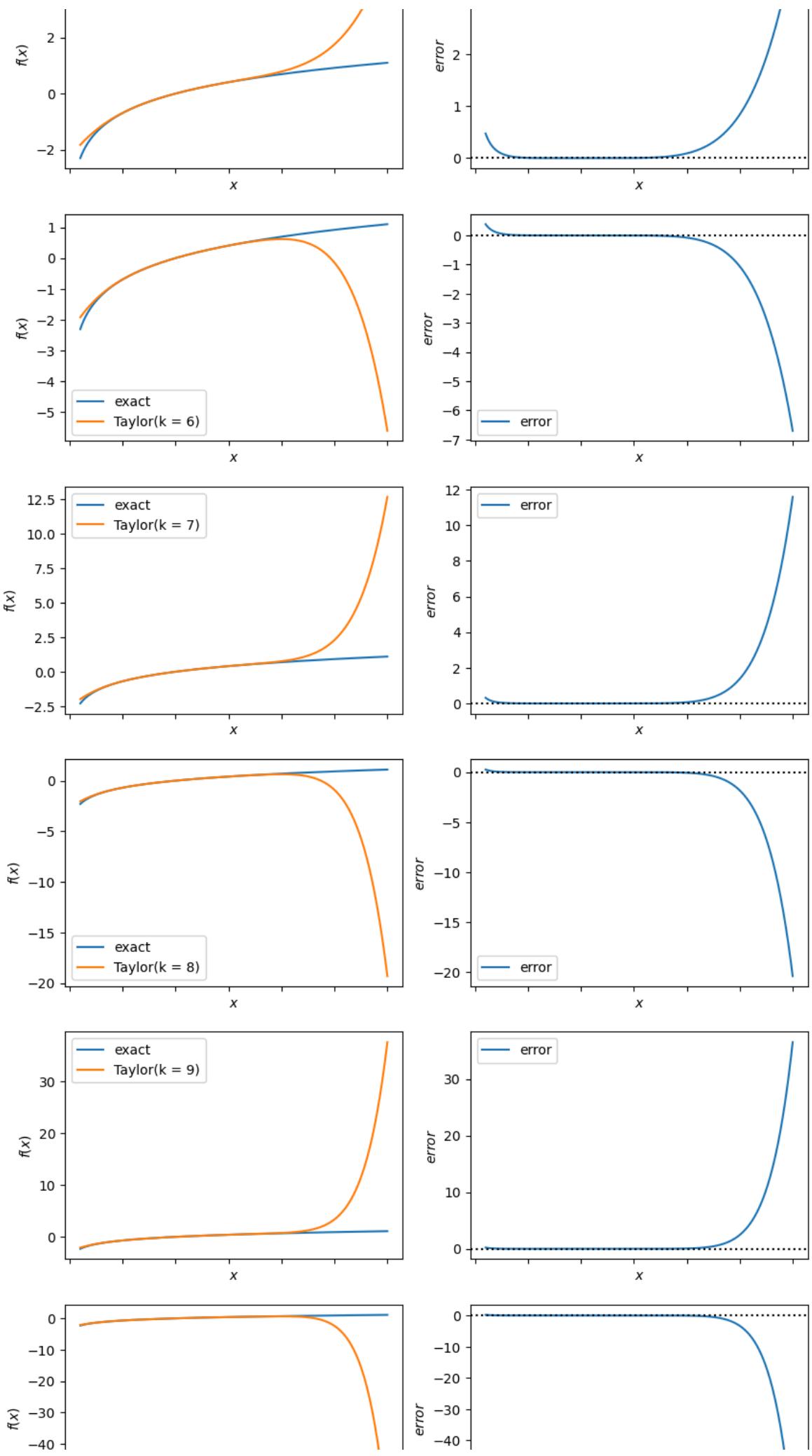
```
x_0 = 0.0
K = 10
func.plot_taylor_approximation_new(
    function=lambda x: 1 / (1 + x**2),
    c=c_4,
    K=K,
    x_0=x_0,
    interval_plot=(x_0 - 0.9, x_0 + 2.0),
    figure_size=(K, 4*K),
)
```

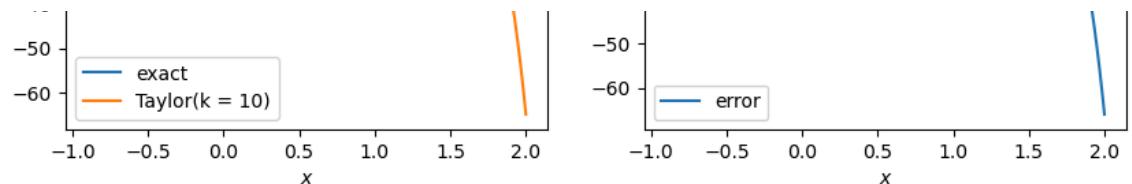


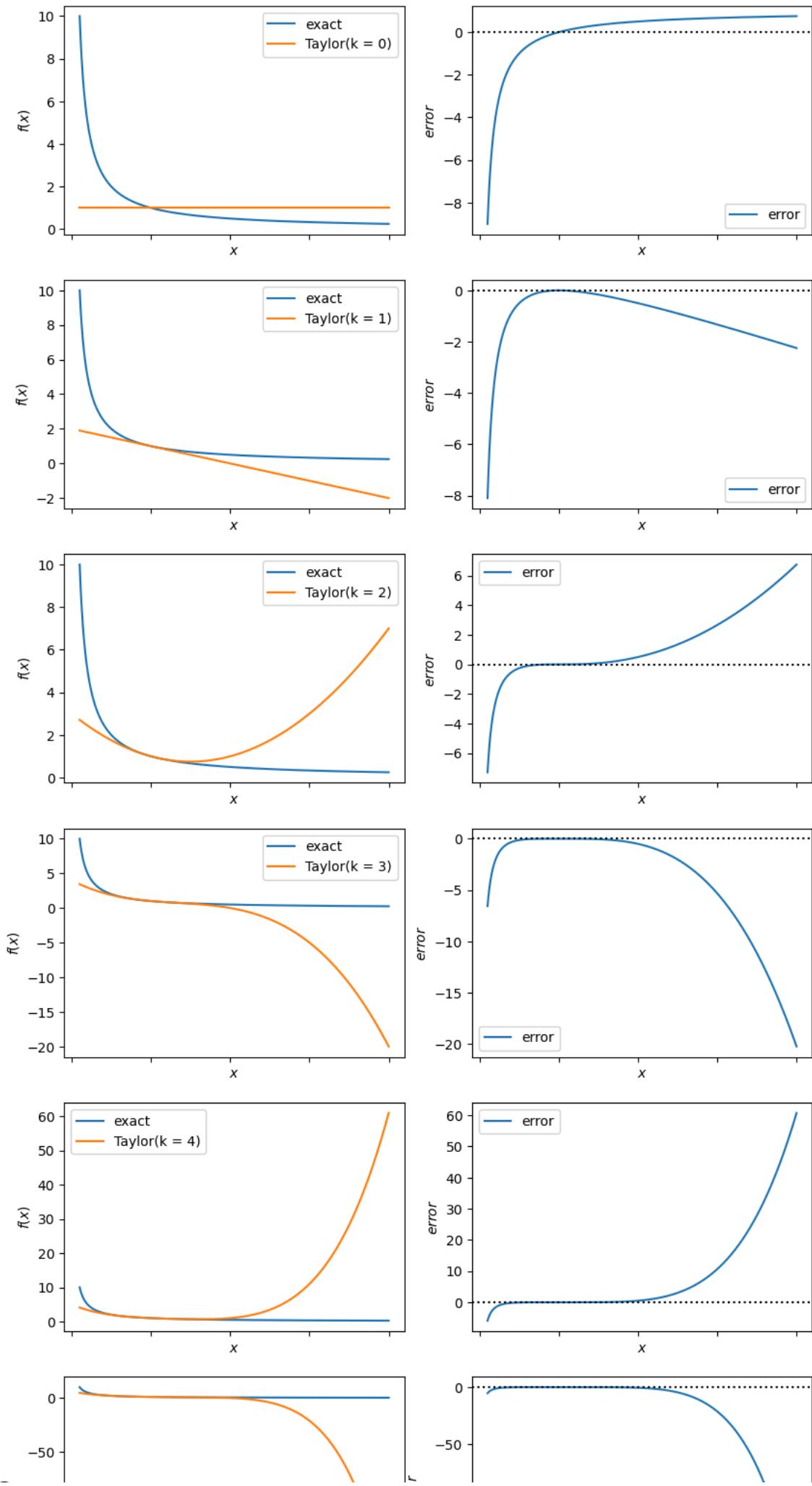


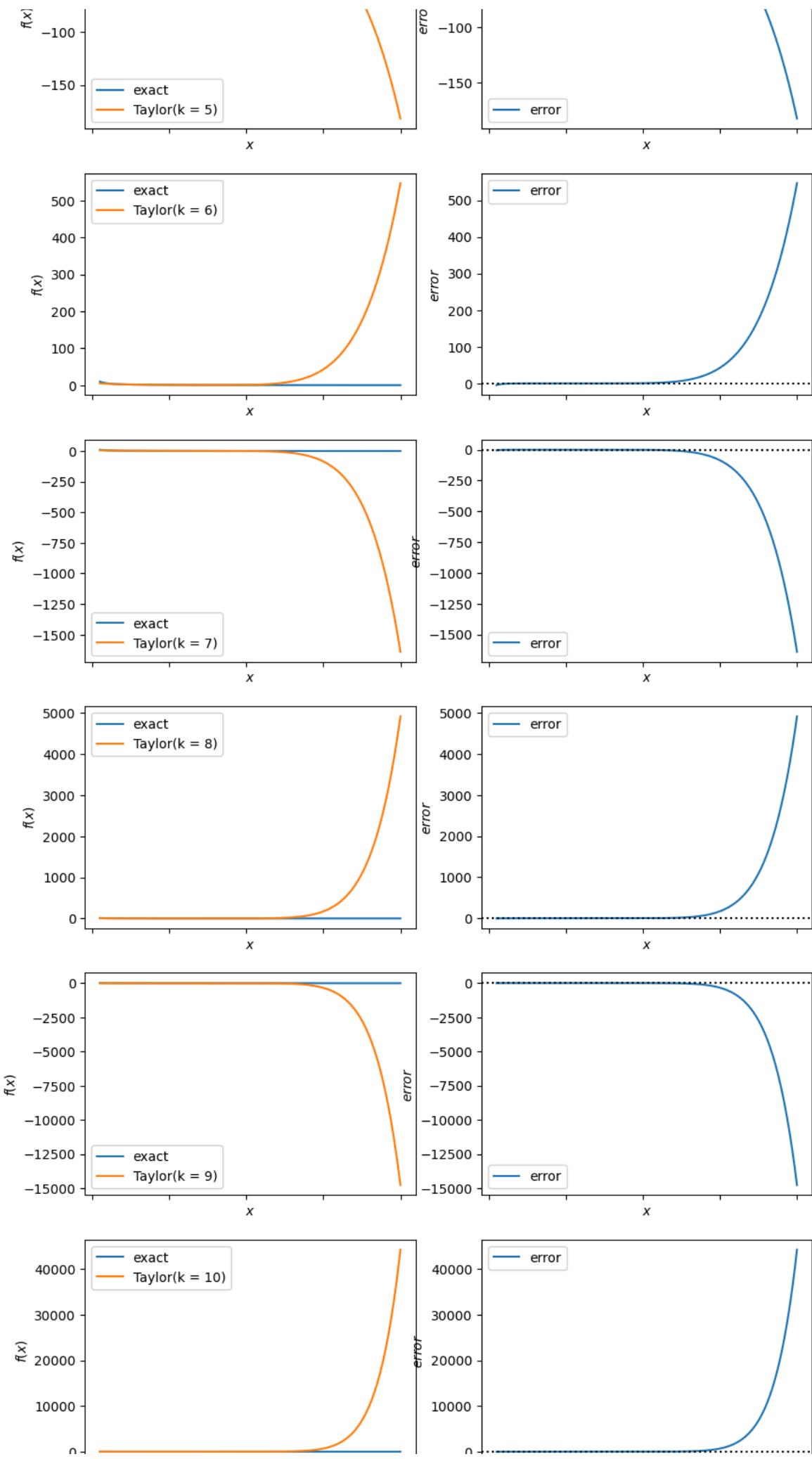




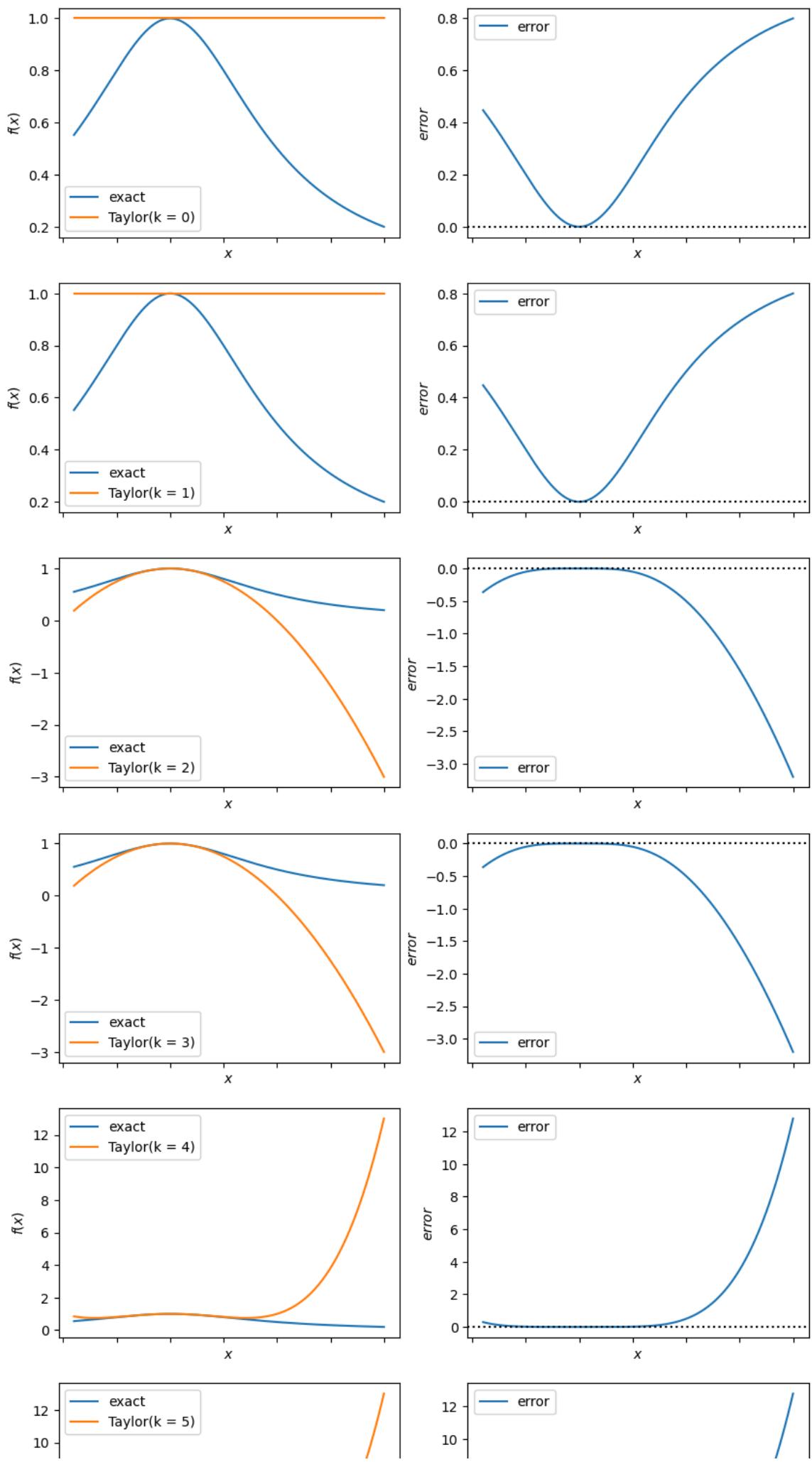


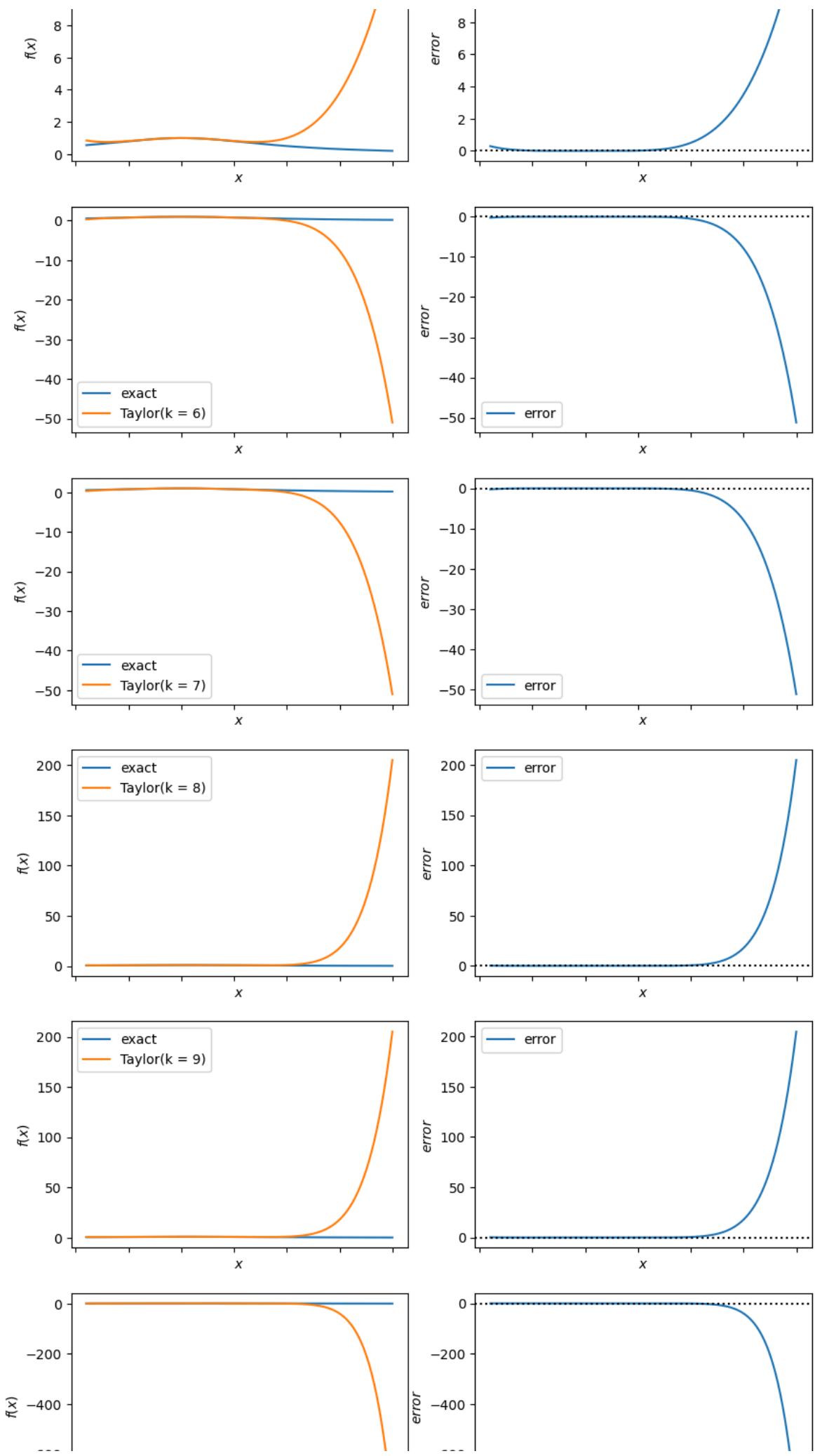


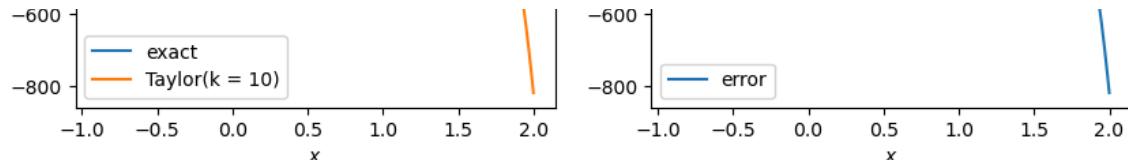












En las gráficas podemos observar que la aproximación de Taylor para  $f_1(x) = \sin(x)$  es muy precisa para todos los puntos, pues tiene un radio de convergencia  $R = \infty$  pero para las demás funciones la aproximación es precisa exclusivamente dentro de su radio de convergencia  $R = 1$ , cuando tratamos de aproximar la función en puntos más lejanos comienza a aumentar considerablemente el error.

## Exercise 4. Estimation of derivatives by divided differences.

The derivative of a function  $f(x)$  at  $x_0$  measures the rate of variation of the function at that point.

We are given three formulas to calculate the derivative numerically

- Left derivative:

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} + \mathcal{O}((\Delta x)^{n_{left}}).$$

- Central derivative:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} + \mathcal{O}((\Delta x)^{n_{central}}).$$

- Right derivative:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} + \mathcal{O}((\Delta x)^{n_{right}}).$$

1. Determine, using Taylor expansions to the appropriate order around  $x_0$  the values of  $n_{right}$ ,  $n_{center}$ , and  $n_{left}$ .

En este apartado comparamos la derivada exacta de una función con las estimaciones proporcionadas por las fórmulas de diferencias divididas: left derivative, central derivative y right derivative. Utilizando las expansiones en series de Taylor alrededor del punto  $x_0$ , analizamos la magnitud del error asociado a cada aproximación y determinamos el orden del error en función de  $\Delta x$  para cada fórmula.

- **Left derivative**

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} + \mathcal{O}((\Delta x)^{n_{left}}).$$

Para aproximar la derivada de una función  $f$  en un punto  $x_0$  utilizando la **left derivative**, recurrimos a la expansión de Taylor de primer orden:

$$f(x_0 - \Delta x) = f(x_0) - \Delta x f'(x_0) + \frac{(\Delta x)^2}{2} f''(\xi), \text{ con } \xi \in (x_0 - \Delta x, x_0)$$

Reorganizando esta expresión para despejar  $f'(x_0)$ , se obtiene:

$$f'(x_0) = \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} - \frac{\Delta x}{2} f''(\xi)$$

De aquí obtenemos una aproximación:

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x)$$

El error asociado es  $-\frac{\Delta x}{2} f''(\xi)$ , que depende linealmente de  $\Delta x$ , por lo tanto:

$$n_{\text{left}} = 1$$

- **Central derivative**

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} + \mathcal{O}((\Delta x)^{n_{\text{central}}}).$$

También podemos hacer la aproximación de la derivada en  $x_0$  usando la fórmula **central derivative**. Para ello, desarrollamos la expansión de Taylor de segundo orden.

Desarrollamos  $f$  en  $x_0 + \Delta x$  y  $x_0 - \Delta x$ :

$$f(x_0 + \Delta x) = f(x_0) + \Delta x f'(x_0) + \frac{(\Delta x)^2}{2} f''(x_0) + \frac{(\Delta x)^3}{6} f'''(\xi_+), \quad \text{con } \xi_+ \in (x_0, x_0 + \Delta x)$$

$$f(x_0 - \Delta x) = f(x_0) - \Delta x f'(x_0) + \frac{(\Delta x)^2}{2} f''(x_0) - \frac{(\Delta x)^3}{6} f'''(\xi_-), \quad \text{con } \xi_- \in (x_0 - \Delta x, x_0)$$

Restando ambas expresiones:

$$f(x_0 + \Delta x) - f(x_0 - \Delta x) = 2\Delta x f'(x_0) + \frac{(\Delta x)^3}{6} [f'''(\xi_+) + f'''(\xi_-)].$$

Dando por hecha la continuidad de  $f'''$ , existe  $\xi \in (x_0 - \Delta x, x_0 + \Delta x)$  tal que:

$$f'''(\xi) = \frac{1}{2} [f'''(\xi_+) + f'''(\xi_-)].$$

Sustituyendo:

$$f(x_0 + \Delta x) - f(x_0 - \Delta x) = 2\Delta x f'(x_0) + \frac{(\Delta x)^3}{3} f'''(\xi),$$

y despejando  $f'(x_0)$ :

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} - \frac{2(\Delta x)^2}{3} f'''(\xi).$$

Obtenemos así la aproximación:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} + \mathcal{O}((\Delta x)^2).$$

Dado que el término de error es cuadrático en  $\Delta x$ :

$$n_{\text{central}} = 2.$$

- **Right derivative**

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} + \mathcal{O}((\Delta x)^{n_{\text{right}}}).$$

Por último, podemos estimar la derivada de  $f$  en  $x_0$  es usando la **right derivative**. Para ello, recurrimos a la expansión de Taylor de primer orden:

$$f(x_0 + \Delta x) = f(x_0) + \Delta x f'(x_0) + \frac{(\Delta x)^2}{2} f''(\xi), \text{ con } \xi \in (x_0, x_0 + \Delta x)$$

Despejando  $f'(x_0)$ :

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} - \frac{\Delta x}{2} f''(\xi)$$

Lo cual nos permite escribir:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} + \mathcal{O}(\Delta x)$$

El término de error  $-\frac{\Delta x}{2} f''(\xi)$  depende linealmente de  $\Delta x$ , por lo tanto:

$$n_{\text{right}} = 1$$

### Conclusión :

En este apartado hemos visto que tanto left derivative como right derivative tienen un error de orden 1, mientras que central derivative mejora la precisión al tener un error de orden 2. Por lo tanto, cuando se desea una estimación más precisa de la derivada, la central derivative es preferible, siempre que se disponga de información de ambos lados del punto  $x_0$ .

2. According to the answer to the previous question, which formula should be used for the numerical estimation of  $f'(x_0)$  using divided differences?.

Como hemos visto en el primer apartado, la mejor fórmula para estimar numéricamente  $f'(x_0)$  es la fórmula de central derivative.

Este método tiene un error de segundo orden, lo que lo hace más preciso que las fórmulas left y right derivative.

3. For the numerical estimation of  $f'(x_0)$ , with  $x_0 \neq 0.0$ , one should use  $\Delta x = x_0 h$ , for some small  $h$ . Explain why.

Retomemos la fórmula de la derivada central y la escribimos tomando  $\Delta x = x_0 h$ :

$$f'(x_0) \approx \frac{f(x_0 * (1 + h)) - f(x_0 * (1 - h))}{2x_0 h}.$$

Comparando con la fórmula original,

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x},$$

podemos inferir cuál es el motivo para utilizar este  $\Delta x$ , para  $x_0 \neq 0.0$ : relativizar el error de la fórmula. De esta manera, sea cual sea el  $x_0$  utilizado, podemos emplear el mismo  $h$  para cualquier  $x_0$ , ya que en comparación con el 1, sabemos qué  $h$  son pequeños (como mínimo, por ejemplo,  $h = \epsilon$ ) y qué  $h$  son grandes. Si tomásemos un  $\Delta x$  distinto del propuesto, para  $x_0 \neq 0$ , estaríamos trabajando con diferencias absolutas de la fórmula y, por supuesto, la necesidad de variar  $\Delta x$  en función de  $x_0$  para saber cuál es pequeño y cuál es demasiado grande. Esto sucede para  $x_0 \neq 0$  ya que para el 0 no está definido el error relativo (no se puede dividir entre 0) y por supuesto cualquier cantidad  $h$  proporcional a 0 nos devolvería un  $\Delta x = 0$ .

Our goal is to determine what is the value of  $h$  that is optimal, in the sense that the error of the numerical estimation of the derivatives by divided differences is as accurate as possible.

4. Determine by numerical exploration the optimal value of  $h$  of each of the three formulas given earlier for the functions  $f(x) = e^x$  at  $x_0 = 1.0$ .
5. Taking into consideration rounding and truncation errors, explain the behaviour observed of the error as a function of  $h$ . If possible provide an estimate of such errors for the values of  $h$  considered.
6. Provide numerical estimates of the optimal value of  $h$  for  $f(x) = \log x$  at  $x_0 = 10^{-30}$ , at  $x_0 = 1.0$ , and at  $x_0 = 10^{30}$ .

```
In [18]: # Achieving optimal value of h for exp(x) at x_0 = 1.0

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

x_0 = 1.0

[h_left, h_right, h_central] = [func.h_optimo_derivate(f, f_derivative, func.left),
                                 func.h_optimo_derivate(f, f_derivative, func.right),
                                 func.h_optimo_derivate(f, f_derivative, func.central)]

print(
    f"El valor óptimo de h para la derivada de exp(x) en x_0 = {x_0:.2f}"
    f" para la fórmula regresiva es {h_left:.2e}, "
    f"\npara la progresiva es {h_right:.2e}, "
    f"y para la central es {h_central:.2e}.")
```

```
df = pd.DataFrame({
    "Método": ["Left", "Right", "Central"],
    "h óptimo": [h_left, h_right, h_central]
})
print(df.to_string(index=False))
```

El valor óptimo de  $h$  para la derivada de  $\exp(x)$  en  $x_0 = 1.00$  para la fórmula regresiva es  $2.21e-09$ , para la progresiva es  $2.21e-09$ , y para la central es  $3.50e-06$ .

Método	$h$ óptimo
Left	$2.214194e-09$
Right	$2.214194e-09$
Central	$3.500177e-06$

- Podemos observar que, en efecto, la fórmula con mayor orden es la central, ya que se alcanza el  $h$  óptimo en un número más elevado que para las otras dos, por lo tanto, converge más rápido para minimizar el error que las fórmulas progresiva y regresiva. Además, el  $h$  óptimo para estas dos fórmulas es el mismo, que es lo esperado ya que ambas fórmulas tienen el mismo orden.

In [20]: # Behaviour of the error as a function of  $h$ :

```
h_vals = np.logspace(-1, -16, 16)
true_value = f_derivative(x_0)
errores_left = []
errores_right = []
errores_central = []

for h in h_vals:
    approx_left = func.left_derivate(f, x_0, h)
    approx_right = func.right_derivate(f, x_0, h)
    approx_central = func.central_derivate(f, x_0, h)
    errores_left.append(np.abs(true_value - approx_left))
    errores_right.append(np.abs(true_value - approx_right))
    errores_central.append(np.abs(true_value - approx_central))

df = pd.DataFrame({
    'h': h_vals,
    'error_progresiva': errores_left,
    'error_regresiva': errores_right,
    'error_central': errores_central
})

print(df.to_string(index=False))
```

$h$	error_progresiva	error_regresiva	error_central
1.000000e-01	1.314947e-01	1.405601e-01	4.532735e-03
1.000000e-02	1.354622e-02	1.363683e-02	4.530492e-05
1.000000e-03	1.358688e-03	1.359594e-03	4.530467e-07
1.000000e-04	1.359096e-04	1.359186e-04	4.530566e-09
1.000000e-05	1.359138e-05	1.359150e-05	5.858691e-11
1.000000e-06	1.359298e-06	1.358972e-06	1.634572e-10
1.000000e-07	1.398295e-07	1.399467e-07	5.858736e-11
1.000000e-08	6.602751e-09	6.602751e-09	6.602751e-09
1.000000e-09	2.286474e-07	2.154419e-07	6.602751e-09
1.000000e-10	2.893183e-06	1.547709e-06	6.727366e-07
1.000000e-11	1.177497e-05	3.263395e-05	1.042949e-05
1.000000e-12	1.177497e-05	4.323142e-04	2.102696e-04
1.000000e-13	4.558642e-04	4.558642e-04	4.558642e-04
1.000000e-14	9.337648e-03	9.337648e-03	9.337648e-03
1.000000e-15	5.374657e-02	3.903426e-01	1.682980e-01
1.000000e-16	2.718282e+00	2.718282e+00	2.718282e+00

5. Como podemos observar en la tabla anterior, el error desciende de manera casi perfecta según el orden de cada método, si dividimos el paso  $h$  con un factor de 10, el error se minimiza con el mismo factor para las fórmulas progresiva y regresiva, pues tienen  $\mathcal{O}(h)$  y para la fórmula central el error se divide por  $10^2 = 100$ , pues tiene  $\mathcal{O}(h^2)$ . Esto no se sale de lo esperado. Sin embargo, cuando descendemos demasiado  $h$ , en concreto, más allá del  $h$  óptimo calculado en el apartado anterior, el error aumenta de nuevo, debido a los errores de redondeo y truncamiento, pues estamos realizando operaciones con números demasiado "pequeños" y la aproximación deja de mejorar. Es por ello que debemos ser muy cautelosos con el paso  $h$  que elegimos, ya que este debe ser pequeño pero no "demasiado" si no queremos incurrir en este tipo de errores.

```
In [22]: # Numerical estimates for the optimal h for f(x) = log(x)

f = lambda x: np.log(x)
f_derivative = lambda x: 1 / x

# At x_0 = 10.0^{-30}

x_0 = 1.0e-30

[h_left, h_right, h_central] = [func.h_optimo_derivate(f, f_derivative, func.left),
                                 func.h_optimo_derivate(f, f_derivative, func.right),
                                 func.h_optimo_derivate(f, f_derivative, func.cen]

print(
    f"El valor óptimo de h para la derivada de log(x) en x_0 = {x_0:.2e}"
    f" para la fórmula regresiva es {h_left:.2e}, "
    f"\npara la progresiva es {h_right:.2e}, "
    f"y para la central es {h_central:.2e}.\n"
)
df = pd.DataFrame({
    "Método": ["Left", "Right", "Central"],
    "h óptimo": [h_left, h_right, h_central]
})
print(df.to_string(index=False))
```

El valor óptimo de  $h$  para la derivada de  $\log(x)$  en  $x_0 = 1.00e-30$  para la fórmula regresiva es  $8.27e-09$ ,  
para la progresiva es  $8.27e-09$ , y para la central es  $6.01e-06$ .  
Método h óptimo  
Left  $8.268913e-09$   
Right  $8.268913e-09$   
Central  $6.006364e-06$

In [23]: # At  $x_0 = 1.0$

```
x_0 = 1.0

[h_left, h_right, h_central] = [func.h_optimo_derivate(f, f_derivative, func.lef
                                func.h_optimo_derivate(f, f_derivative, func.rig
                                func.h_optimo_derivate(f, f_derivative, func.cen

print(
    f"El valor óptimo de h para la derivada de log(x) en x_0 = {x_0:.2f}"
    f" para la fórmula regresiva es {h_left:.2e}, "
    f"\npara la progresiva es {h_right:.2e}, "
    f"y para la central es {h_central:.2e}."
)

df = pd.DataFrame({
    "Método": ["Left", "Right", "Central"],
    "h óptimo": [h_left, h_right, h_central]
})
print(df.to_string(index=False))
```

El valor óptimo de  $h$  para la derivada de  $\log(x)$  en  $x_0 = 1.00$  para la fórmula regresiva es  $8.82e-09$ ,  
para la progresiva es  $7.58e-09$ , y para la central es  $5.39e-06$ .

Método h óptimo  
Left  $8.822486e-09$   
Right  $7.584465e-09$   
Central  $5.391472e-06$

In [24]: # At  $x_0 = 10.0^{+30}$

```
x_0 = 1.0e+30

[h_left, h_right, h_central] = [func.h_optimo_derivate(f, f_derivative, func.lef
                                func.h_optimo_derivate(f, f_derivative, func.rig
                                func.h_optimo_derivate(f, f_derivative, func.cen

print(
    f"El valor óptimo de h para la derivada de log(x) en x_0 = {x_0:.2e}"
    f" para la fórmula regresiva es {h_left:.2e}, "
    f"\npara la progresiva es {h_right:.2e}, "
    f"y para la central es {h_central:.2e}."
)

df = pd.DataFrame({
    "Método": ["Left", "Right", "Central"],
    "h óptimo": [h_left, h_right, h_central]
})
print(df.to_string(index=False))
```

El valor óptimo de  $h$  para la derivada de  $\log(x)$  en  $x_0 = 1.00e+30$  para la fórmula regresiva es  $8.27e-09$ , para la progresiva es  $8.27e-09$ , y para la central es  $6.01e-06$ .

Método	$h_{\text{óptimo}}$
Left	$8.268913e-09$
Right	$8.268913e-09$
Central	$6.006364e-06$

6. Como podemos observar, gracias a que hemos implementado en el cálculo de nuestras fórmulas la idea de relativizar la diferencia (sino, para estos números tan dispares, tendríamos muy diferentes pasos  $h$ ), los  $h$  óptimos son prácticamente iguales, sea cual sea el  $x_0$  elegido, siendo más grande el  $h$  óptimo de la fórmula central gracias a que tiene mayor orden de convergencia.

## Order of convergence.

In many numerical estimation methods the goal is to estimate a quantity, say  $x^*$ , using an iterative procedure.

In an iterative algorithm one builds sequence  $x_0, x_1, \dots, x_n, \dots$ , which is expected to converge in the limit of large  $n$  to the solution

$$\lim_{n \rightarrow \infty} x_n = x^*.$$

Assuming that the sequence has a limit (converges), the error of the approximation, given by the unknown quantity  $(x_n - x^*)$ , can be approximated by

$$e_n = |x_n - x_{n-1}|.$$

For large  $n$ , the error is expected to decrease as

$$e_n \approx C (e_{n-1})^\alpha, \quad n \rightarrow \infty$$

where  $\alpha$  is the order of (local) convergence of the algorithm, and

$$C = \lim_{n \rightarrow \infty} \frac{e_n}{(e_{n-1})^\alpha}.$$

For instance, if  $\alpha = 1$ , the convergence is linear, for  $\alpha = 2$  quadratic, and so on.

## Exercise 5: Order of convergence of Newton-Raphson,

Consider the function  $f(x)$  whose derivative  $f'(x)$  is also known. Our goal is to find a value  $x = x^*$  such that  $f(x^*) = 0.0$ .

Newton-Raphson's method for the estimation of the zero consists of the following algorithm

1. Start from  $x_0$  (initial seed, given)
2. Iterate:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, n = 1, 2, \dots$$

In this manner, one obtains the sequence  $x_0, x_1, \dots, x_n, \dots$ , which is expected to converge in the limit of large n

$$\lim_{n \rightarrow \infty} x_n = x^*.$$

The algorithm is based on the following strategy:

- Consider  $x_{n-1}$ , the estimate of  $x^*$  at iteration  $n - 1$  of the algorithm.
- Assuming that we are close to convergence, we can improve the approximation of  $x^*$  by finding the zero of the tangent of  $f(x)$  at  $x = x_{n-1}$ . The formula of the tangent is

$$t(x) = f(x_{n-1}) + f'(x_{n-1})(x - x_{n-1}).$$

We define  $x_n$  as the value of  $x$  such that  $t(x_n) = 0$

$$f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1}) = 0 \implies x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}.$$

1. Does the algorithm guarantee finding a zero? (global convergence).

Necesitamos más hipótesis sobre la función para poder garantizar la convergencia global del algoritmo, como que la derivada no se anule en ningún punto del intervalo. Como contraejemplo, podemos tomar  $f(x) = \sin(x)$ ,  $f'(x) = \cos(x)$ . Es evidente que un 0 de la función es el propio 0 o el número  $\pi$  ( $-\pi$  también) y que para  $x_n = \frac{\pi}{2}$ , no podemos continuar con el algoritmo ya que es un 0 de la función derivada y no podemos hacer divisiones entre 0. Partiendo de  $x_0 = -1.22759$  (este número lo obtenemos de la resolución numérica de la ecuación  $\frac{\pi}{2} = x - \frac{\sin(x)}{\cos(x)}$ ), se obtendría  $x_1 = \frac{\pi}{2}$  y no podríamos continuar con el algoritmo.

2. Does the algorithm guarantee finding a zero that is sufficiently close to the seed? (local convergence).

Sí, en este caso sí que se puede garantizar que el algoritmo encuentra un 0 de la función.

En el caso de que haya raíces múltiples, siendo  $x^*$  el valor tal que  $f(x^*) = 0$ , podría darse el caso de que  $f'(x^*) = 0$  también, como para la función  $f(x) = \frac{x^2}{2}$ . Si se diese esta situación, el iterante  $x_n$  en el que se anule la derivada, como estamos en un entorno cercano a la solución  $x^*$ , el algoritmo no podría continuar en la búsqueda de  $x_{n+1}$  (pues no se puede dividir entre 0), pero es este mismo valor  $x_n$  para el que hemos encontrado el 0 de la función, ya que si suponemos que este  $x_n$  no es 0 de la función, entonces habría un entorno de  $x_n$  en el que no está  $x^*$  (ya que  $f'(x_n) = 0$ ), en contradicción con la hipótesis, por tanto  $x_n$  tiene que ser 0 de  $f(x)$ . Retomando el ejemplo  $f(x) = \frac{x^2}{2}$ , se puede ver que el punto  $x$  en el que se anula la derivada  $f'(x) = x$  es  $x = 0$ , que es, a su vez, un 0 de la función ( $f(0) = 0$ ). Si solo hay una raíz simple (función lineal), si la

derivada se anula en  $x^*$ , gracias a la propia interpretación geométrica de la derivada, se tendría que en un entorno cercano  $x^*$  todos los puntos son ceros de la función.

Supongamos por tanto, sin pérdida de generalidad, que  $f'(x^*) \neq 0$ . Definiendo como función de iteración  $F(x) = x - \frac{f(x)}{f'(x)}$ , se tiene que  $x^*$  es un punto fijo de  $F$  y se puede probar que  $F'(x^*) = 0$ . Por tanto, se cumplen las condiciones del teorema de convergencia local de Ostrowski y esto implica que la sucesión

$x_{n+1} = F(x_n) = x_n - \frac{f(x_n)}{f'(x_n)}$  (algoritmo de Newton-Raphson) converge a  $x^*$  en un entorno local de  $x^*$  con al menos orden 1.

3. Determine the order of (local) convergence (i.e., the value of  $\alpha$ ) of this algorithm.

Para probar el orden de convergencia del algoritmo, realizaremos el desarrollo de Taylor centrado en  $x_n$  a  $f(x^*)$ :

$$0 = f(x^*) = f(x_n) + f'(x_n)(x^* - x_n) + \frac{1}{2}f''(\xi_n)(x^* - x_n)^2,$$

siendo  $\xi_n$  un punto entre  $x_n$  y  $x^*$ . Como  $f(x^*) = 0$  y definiendo  $e_n = x_n - x^*$ , obtenemos:

$$\begin{aligned} f(x_n) - e_n f'(x_n) &= -\frac{e_n^2}{2} f''(\xi_n) \Leftrightarrow e_{n+1} = \frac{f''(\xi_n)}{2f(x_n)} e_n^2 \Rightarrow \\ &\Rightarrow \lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^2} \stackrel{(*)}{=} \frac{f''(x_0)}{2f(x_0)} = C \end{aligned}$$

(\*) Teniendo en cuenta que la sucesión  $x_n$  converge a  $x_0$

Por tanto, como se puede observar, el algoritmo de Newton-Raphson tiene convergencia cuadrática.

4. Provide a graphical illustration of this order of convergence for the function

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

5. Taking into account rounding error, what is the smallest value (order of magnitude) of the relative error  $\left(\frac{e_n}{x_n}\right)$  that we can expect to achieve?

In [28]: # Graphical illustration for order of convergence for Newton-Raphson

```
# Función f(x) = e^{-x} - x
f_exacta = lambda x: np.exp(-x) - x
df_exacta = lambda x: -np.exp(-x) - 1

# Newton-Raphson
x_0 = 1.0
n_iter = 15
error = np.zeros(n_iter)
valor = np.zeros(n_iter)
for i in range(len(error)):
    error[i] = qf.newton_raphson(f=f_exacta, df_dx=df_exacta,
                                  seed=x_0, max_iters=i)[1]
    valor[i] = qf.newton_raphson(f=f_exacta, df_dx=df_exacta,
```

```

        seed=x_0, max_iters=i)[0]

iter = np.arange(len(error)) + 1
err_rel = error / valor
df = pd.DataFrame({
    'Iteración': iter,
    'Error': np.abs(error),
    'Error relativo': np.abs(err_rel)
})
df.style.format({'Error': '{:.2e}'})
print(df.to_string(index=False))
print(f'El cero de la función f(x) = exp(-x) - x se obtiene en {valor[n_iter - 1]}
```

Iteración	Error	Error relativo
1	1.797693e+308	1.797693e+308
2	4.621172e-01	8.591409e-01
3	2.910415e-02	5.133125e-02
4	1.562946e-04	2.755822e-04
5	4.420661e-09	7.794610e-09
6	4.420661e-09	7.794610e-09
7	4.420661e-09	7.794610e-09
8	4.420661e-09	7.794610e-09
9	4.420661e-09	7.794610e-09
10	4.420661e-09	7.794610e-09
11	4.420661e-09	7.794610e-09
12	4.420661e-09	7.794610e-09
13	4.420661e-09	7.794610e-09
14	4.420661e-09	7.794610e-09
15	4.420661e-09	7.794610e-09

El cero de la función  $f(x) = \exp(-x) - x$  se obtiene en 0.5671432904097838

```

C:\Users\USUARIO\Downloads\QFB_2024_2025_P01_G05_crespo_cuesta_salvadores\tools_q
fb.py:178: UserWarning: Maximum number of iterations reached.
    warnings.warn('Maximum number of iterations reached.')
```

4. Se puede observar que el orden decrece con orden de convergencia cuadrático, de hecho decrece con un orden un poco mayor que 2, ya que los errores posteriores descienden un poco más que el cuadrado del error anterior.
5. Vista la tabla, parece que el error relativo se estabiliza del orden de  $10^{-10}$ , ya que el 0 de la función se obtiene en  $\approx \frac{1}{2}$  y el error absoluto se estabiliza en  $\approx \frac{1}{2}10^{-9}$ .

## Integration of ordinary differential equations (ODE): Euler method

Consider the first order explicit differential equation

$$\dot{f}(t) = a(t, f(t)),$$

where  $\dot{f}(t) = \frac{df(t)}{dt}$  denotes the derivative, as is customary with functions of time.

Using differential calculus, this expression can be written as

$$df(t) = a(t, f(t)) dt,$$

which, with the definition of  $df(t) = f(t + dt) - f(t)$ , yields

$$f(t + dt) = f(t) + a(t, f(t)) dt.$$

In case that instead of the differential element  $dt$ , we use an increment of finite size, we get

$$f(t + \Delta t) = f(t) + a(t, f(t)) \Delta t + \mathcal{O}((\Delta t)^2).$$

Our goal is to compute trajectory that is the solution of the ODE in the interval  $[t_0, t_0 + T]$  starting from the *initial condition*  $f(t_0) = f_0$ .

To this end, we consider a grid of  $N + 1$  points in the interval  $[t_0, t_0 + T]$  at which we monitor the trajectory:

$$t_n = t_0 + n\Delta T, \quad (24)$$

$$f_n = f(t_n), \quad n = 0, 1, \dots, N - 1, \quad (25)$$

where  $\Delta T = \frac{T}{N}$ .

The approximation of the trajectory, monitored at  $\{t_0, t_1, \dots, t_n\}$ , in the Euler integration method is

$$f(t_0) = f_0 \quad (26)$$

$$f_{n+1} = f_n + a(t_n, f_n) \Delta t, \quad n = 0, 1, \dots, N - 1. \quad (27)$$

If written in the form

$$f_{n+1} - f_n = +a(t_n, f_n) \Delta t, \quad n = 0, 1, \dots, N - 1, \quad (28)$$

this expression defines a *difference equation*, which, in some cases, can be solved in closed-form. That is, an analytic expression for  $f_n$  as a function of the initial condition,  $f_0$ , and of  $n$  can be derived.

```
In [31]: # Example: Euler integration method for an ODE with reversion to a time-dependent mean

alpha = 0.1 # reversion rate
sigma_ref = lambda t: 0.3 - 0.1 * np.cos(t) # time dependent mean
sigma_0 = 0.6 # initial value

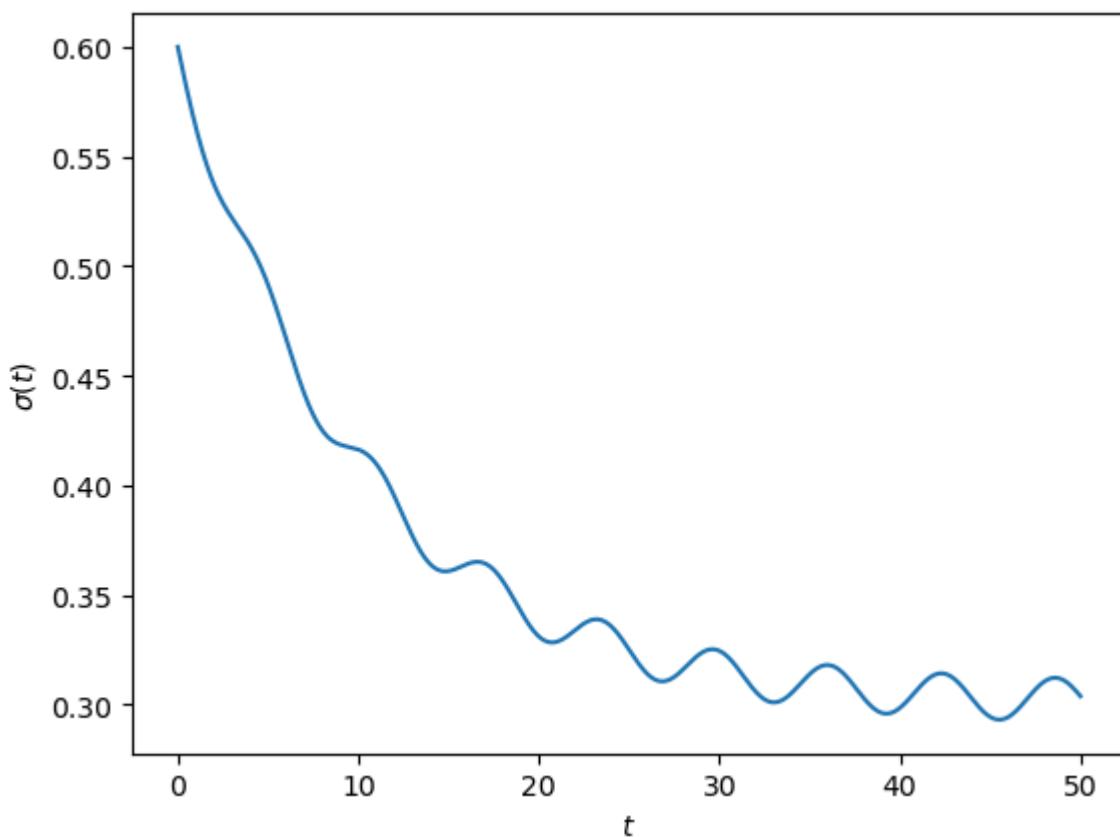
t, f = func.euler_integration(
    a=lambda t, sigma: -alpha * (sigma - sigma_ref(t)),
    t_0=0.0,
    f_0=sigma_0,
    T=50.0,
    N=1000,
)

fig, ax = plt.subplots()

ax.plot(t, f)
ax.set_xlabel('$t$')
ax.set_ylabel('$\sigma(t)$')
```

```
<>:19: SyntaxWarning: invalid escape sequence '\s'
<>:19: SyntaxWarning: invalid escape sequence '\s'
C:\Users\USUARIO\AppData\Local\Temp\ipykernel_9640\2355538371.py:19: SyntaxWarning: invalid escape sequence '\s'
    ax.set_ylabel('$\sigma(t)$')
```

Out[31]: Text(0, 0.5, '\$\sigma(t)\$')



## Exercise 6: Errors in the Euler method

Consider the approximation to the solution of the ODE computed using the Euler integration method

1. What is the dominant type of error (rounding or truncation) in this solution?

A pesar de que se pueden cometer errores de redondeo, los cuales son muy relevantes, por ejemplo si elegimos una función  $a(t, f(t))$  demasiado grande (o pequeña) en comparación con  $\Delta t$ , el error dominante en esta solución es el error de truncamiento, pues estamos tratando de aproximar una integral (un paso al límite) a partir de una sucesión finita. Cuando  $\Delta t \rightarrow 0$ , esto es, tal y como está definido  $\Delta t$ , equivalente a cuando  $N \rightarrow \infty$ , se obtiene la solución real de la EDO. Al tratar de aproximar esta solución con un número finito de valores ( $N$ ), el error principal de la aproximación es el error de truncamiento.

2. In terms of  $\delta T$ , what is the order of the error per time step?

El propio enunciado ya te indica el orden del error por cada paso de la aproximación, obtenido como en el ejercicio anterior por la expansión de Taylor. Para un  $t$  genérico, se tiene:

$$f(t + \Delta t) = f(t) + a(t, f(t)) \Delta t + \mathcal{O}((\Delta t)^2).$$

Ahora bien, para ver el error cometido en cada paso, partiendo desde un  $t_n$  cualquiera, se tiene, por un lado,

$$\begin{aligned}t_{n+1} &= t_n + \Delta T \\f_{n+1} &= f(t_{n+1}),\end{aligned}$$

por tanto, haciendo la identificación pertinente con la ecuación para el  $t$  genérico:

$$\begin{aligned}f_{n+1} &= f_n + a(t_n, f_n) \Delta T \Leftrightarrow \\&\Leftrightarrow f(t_n + \Delta T) = f(t_n) + a(t_n, f(t_n)) \Delta T + \mathcal{O}((\Delta T)^2).\end{aligned}$$

Se ve claramente por tanto que, por cada paso, el error cometido por la aproximación de Euler es de orden cuadrático en  $\Delta T$ , i.e.,  $\mathcal{O}((\Delta T)^2)$ .

3. In terms of  $\Delta T$ , what is the order of the accumulated error at  $t_0 + T$ ?

El error acumulado en  $t_0 + T = t_N$  es, obviamente, la suma de los errores cometidos en cada paso. Como conocemos el número de aproximaciones realizadas ( $N$ ) y, en el apartado anterior hemos descrito el orden del error cometido en cada paso, parece bastante sencillo inferir que el orden del error acumulado en toda la aproximación es  $N \mathcal{O}((\Delta T)^2)$ . Ahora bien, nos damos cuenta de dos hechos; el primero, que  $N = \frac{T}{\Delta T}$ . Por otro lado el orden de la aproximación refleja el error acotado en términos de  $\Delta T$ , es decir, el orden es el error de la aproximación  $\mathcal{O}((\Delta T)^2) = C_1 * (\Delta T)^2$ , siendo  $C_1$  una constante para acotar el error, en nuestro caso, derivado de la expansión de Taylor. Esto es así porque lo relevante para determinar el error de la aproximación es el paso  $\Delta T$ , no la cota del error  $C_1$ .

Bien, teniendo esto en cuenta y retomando la expresión del error acumulado, como  $T$  también la podemos considerar como una constante, independiente de  $\Delta T$  y de  $C_1$ , obtenemos lo siguiente:

$$\begin{aligned}N \mathcal{O}((\Delta T)^2) &= \frac{T}{\Delta T} * C_1 (\Delta T)^2 = \\&= T C_1 * \Delta T = C * \Delta T = \mathcal{O}(\Delta T).\end{aligned}$$

Por tanto, llegamos a que el orden del error acumulado en la aproximación es lineal, en términos de  $\Delta T$ . Se tiene un orden  $\mathcal{O}(\Delta T)$ .

4. Suggest a way to reduce the error of the Euler approximation by a factor of 10.

En vista de las respuestas a las preguntas anteriores, es muy fácil comprobar, en base al orden del error global de la aproximación, de que si queremos mejorar el orden de la aproximación por un factor de 10, siendo muy cautelosos con los errores de redondeo, ya que podríamos terminar incurriendo en este tipo de errores si ya estamos trabajando con un paso  $\Delta T$  lo suficientemente sofisticado, lo siguiente:

Como el orden de la aproximación global es  $\mathcal{O}(\Delta T)$ , si queremos mejorar el orden por un factor de 10, bastaría con afinar nuestro paso  $\Delta T$  dividiéndolo entre 10. En efecto, sea  $\Delta T' = \frac{\Delta T}{10}$ , entonces  $\mathcal{O}(\Delta T') = \mathcal{O}\left(\frac{\Delta T}{10}\right) \stackrel{(*)}{=} \frac{1}{10}\mathcal{O}(\Delta T)$ .

(\*) En el apartado anterior ya se ha explicado por qué podemos operar así con los órdenes de convergencia.

En nuestro método, como lo que definimos es el número  $N$  de pasos, en vez del  $\Delta T$  que tomamos en específico, esto se traduciría en aumentar por un factor de 10 el número de pasos que tomamos. En efecto,

$$N' = \frac{T}{\Delta T'} = \frac{T}{\frac{\Delta T}{10}} = 10 \frac{T}{\Delta T} = 10N.$$

5. Suggest a way to modify the Euler method so as to improve the order of the approximation.

Recordando lo que sucedía con las fórmulas de derivación numérica, tanto la fórmula progresiva como la regresiva tenían orden 1 y la fórmula centrada tenía un orden 2. Podemos identificar el algoritmo de Euler, ya que  $a(t_n, f_n) = f'_n(t_n)$ , con la fórmula progresiva para la primera derivada. En efecto, la fórmula de Euler:

$$f_{n+1} - f_n = a(t_n, f_n) \Delta T \Leftrightarrow \frac{f_{n+1} - f_n}{\Delta T} = f'_n$$

y la fórmula progresiva para la primera derivada:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}.$$

Son evidentes las similitudes entre ambas expresiones, además, ambas tienen orden 1. Por tanto, para mejorar el orden de la aproximación de Euler, trataremos de implementar un algoritmo que se asemeje a la fórmula centrada para la primera derivada, con orden 2. Recordamos su expresión:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{\Delta x}.$$

Para ello, lo que vamos a realizar va a ser, por cada iteración, una aproximación del punto medio entre  $f_n$  y  $f_{n+1}$ , llamémosla  $f_m$ . Obviamente  $t_m$  se corresponde con el punto medio entre  $t_n$  y  $t_{n+1}$ , dado el paso  $\Delta T$ , se tiene que  $t_m = t_n + \frac{\Delta T}{2}$ . Ahora, dada la función  $a$  y los puntos  $f_m$  y  $t_m$ , podemos realizar una aproximación "centrada" para obtener el siguiente iterante  $f_{n+1}$  como sigue:

$$f_{n+1} = f_n + a(t_m, f_m) \Delta T \Leftrightarrow \frac{f_{n+1} - f_n}{\Delta T} = f'_m,$$

donde  $f_m$  se consigue de la siguiente manera:

$$f_m = f_n + a(t_n, f_n) \frac{\Delta T}{2}.$$

Como en el caso del enunciado, este algoritmo se asemeja bastante a la fórmula centrada para la derivada, con lo que esperamos mejorar el orden de convergencia con este cambio, a cambio de hacer el doble de operaciones, pues en cada iteración tenemos que calcular un iterante intermedio  $f_m$ .

```
In [33]: # Checking order of convergence 2 for the Euler improved method

# EDO df/dt = f, con solución exacta f(t) = exp(t) para la
# condición inicial f(0) = 1

a = lambda t, f: f
t_0, f_0, T = 0.0, 1.0, 1.0
true_value = np.exp(t)

# Steps
N = [1, 10, 100, 1000, 10000]
dT = []
error = []
for n in N:
    t, f = func.euler_integration_new(a, t_0, f_0, T, n)
    dt = T / n
    err = abs(f[-1] - true_value)
    dT.append(dt)
    error.append(err)

df = pd.DataFrame({
    "N": N,
    "delta_T": dT,
    "error": error
})

# Print table
print("Convergence Table:")
print(df.to_string(index=False))
```

## Convergence Table:

N	delta_T	error
1	1.0000	[1.5, 1.4487289036239759, 1.3948290819243523, 1.338165757271717, 1.2785972418398301, 1.2159745833122586, 1.1501411924239968, 1.0809324514067427, 1.0081753023587297, 0.9316878145098311, 0.8512787292998718, 0.7667469821326047, 0.6778811996094909, 0.5844591709861038, 0.48624729252952337, 0.3829999833873252, 0.2744590715075321, 0.1603531480740088, 0.04039688884305015, 0.0857096593158464, 0.2182818284590451, 0.35765111806316385, 0.5041660239464334, 0.6581929096897681, 0.8201169227365481, 0.9903429574618414, 1.1692966676192444, 1.3574255306969745, 1.5551999668446754, 1.7631145151688186, 1.9816890703380645, 2.211470182590742, 2.453032424395115, 2.7069798271798495, 2.973947391727201, 3.2546026760057307, 3.5496474644129465, 3.859819522601832, 4.18589444227927, 4.5286875805892945, 4.88905609893065, 5.267901106306774, 5.66616991256765, 6.084858397177893, 6.525013499434122, 6.987735836358526, 7.4741824548147235, 7.985569724727576, 8.523176380641605, 9.088346719223392, 9.682493960703473, 10.307103782663036, 10.963738035001692, 11.654038645375808, 12.379731724872837, 13.142631884188171, 13.944646771097055, 14.78778184056764, 15.674145369443067, 16.60595372823165, 17.585536923187668, 18.615344422540616, 19.697951281441636, 20.836064580942722, 22.032530197109352, 23.290339917193062, 24.612638920657893, 26.002733643767282, 27.464100047397025, 29.000392308747937, 30.61545195869231, 32.31331748760203, 34.09823444367799, 35.97466604903214, 37.9473043600674, 40.02108200006278, 42.201184493300836, 44.493063231579285, 46.90244910553019, 49.43536683483144, 52.098150033144236, 54.89745704544619, 57.840287597362, 60.934000298123344, 64.18633104092515, 67.60541234668786, 71.19979369959579, 74.9784629252609, 78.95086866496814, 83.12694400220059, 87.51713130052181, 92.13240831492406, 96.98431564193386, 102.08498557711422, 107.44717245212352, 113.08428452718766, 119.01041751873497, 125.24038984602892, 131.78977968493552, 138.67496392147686, ...]
10	0.1000	[1.714080846608224, 1.6628097502322, 1.6089099285325763, 1.552246603879941, 1.4926780884480542, 1.4300554299204826, 1.3642220390322208, 1.2950132980149667, 1.2222561489669537, 1.145768661118055, 1.0653595759080958, 0.9808278287408287, 0.8919620462177149, 0.7985400175943278, 0.7003281391377474, 0.597080829995492, 0.48853991811575614, 0.3744339946822328, 0.25447773545127417, 0.12837118729237762, 0.004200981850821073, 0.14357027145493984, 0.2900851773382094, 0.44411206308154405, 0.6060360761283241, 0.7762621108536174, 0.9552158210110204, 1.1433446840887505, 1.3411191202364514, 1.5490336685605945, 1.7676082237298405, 1.9973893359825179, 2.238951577786891, 2.4928989805716255, 2.7598665451189768, 3.0405218293975067, 3.3355666178047225, 3.645738675993608, 3.9718135956710463, 4.314606733981071, 4.674975252322426, 5.053820259698551, 5.452089065959425, 5.870777550569668, 6.3109326528258975, 6.773654989750302, 7.260101608206499, 7.771488878119351, 8.30909553403338, 8.874265872615167, 9.468413114095249, 10.093022936054812, 10.749657188393467, 11.439957798767583, 12.165650878264612, 12.928551037579947, 13.73056592448883, 14.573700993959415, 15.460064522834843, 16.391872881623428, 17.371456076579445, 18.401263575932393, 19.483870434833413, 20.6219837343345, 21.81844935050113, 23.07625907058484, 24.39855807404967, 25.78865279715906, 27.250019200788802, 28.786311462139714, 30.40137111208409, 32.0992366409938, 33.88415359706976, 35.76058520242391, 37.73322351345917, 39.80700115345456, 41.98710364669261, 44.27898238497106, 46.68836825892196, 49.221285988223215, 51.88406918653601, 54.68337619883796, 57.626206750753774, 60.71991945151512, 63.97225019431693, 67.39133150007963, 70.98571285298756, 74.76438207865267, 78.73678781835991, 82.91286315559236, 87.30305045391358, 91.91832746831584, 96.77023479532563, 101.870904730506, 107.2330916055153, 112.87020368057944, 118.79633667212674, 125.02630899942069, 131.5756988383273, 138.46088307486863, ...]
100	0.0100	[1.718236862559957, 1.6669657661839328, 1.6130659444843092, 1.5564026198316738, 1.496834104399787, 1.4342114458722155, 1.3683780549839537, 1.2991693139666995, 1.2264121649186865, 1.149924677069788, 1.0695155918598287, 0.9849838446925616, 0.8961180621694478, 0.8026960335460607, 0.7044841550894803, 0.6012368459472821, 0.492695934067489, 0.3785900106339657, 0.25863375140300704, 0.1325272032441105, 4.49658990882007e-05, 0.13941425550320696, 0.2859291613864765, 0.4399560471298112, 0.6018800601765912, 0.7721060949018845, 0.9510598050592876, 1.139

1886681370176, 1.3369631042847185, 1.5448776526088617, 1.7634522077781076, 1.9932  
33320030785, 2.234795561835158, 2.4887429646198926, 2.755710529167244, 3.03636581  
3445774, 3.3314106018529897, 3.641582660041875, 3.9676575797193134, 4.31045071802  
9338, 4.6708192363706935, 5.049664243746817, 5.447933050007693, 5.86662153461793  
6, 6.306776636874165, 6.769498973798569, 7.255945592254767, 7.767332862167619, 8.  
304939518081648, 8.870109856663435, 9.464257098143516, 10.08886692010308, 10.7455  
01172441735, 11.435801782815851, 12.16149486231288, 12.924395021628214, 13.726409  
908537098, 14.569544978007682, 15.45590850688311, 16.387716865671692, 17.36730006  
0627713, 18.39710755998066, 19.479714418881677, 20.617827718382763, 21.8142933345  
49397, 23.072103054633104, 24.39440205809794, 25.784496781207324, 27.245863184837  
07, 28.78215544618798, 30.397215096132356, 32.09508062504207, 33.87999758111803,  
35.75642918647218, 37.729067497507444, 39.80284513750283, 41.98294763074088, 44.2  
7482636901933, 46.68421224297023, 49.217129972271486, 51.87991317058428, 54.67922  
018288623, 57.622050734802045, 60.71576343556339, 63.9680941783652, 67.3871754841  
279, 70.98155683703582, 74.76022606270094, 78.73263180240818, 82.90870713964063,  
87.29889443796185, 91.9141714523641, 96.7660787793739, 101.86674871455426, 107.22  
893558956356, 112.8660476646277, 118.792180656175, 125.02215298346896, 131.571542  
82237556, 138.4567270589169, ...]  
1000 0.0010 [1.7182813757517628, 1.6670102793757386, 1.613110457676115, 1.  
5564471330234797, 1.496878617591593, 1.4342559590640214, 1.3684225681757596, 1.29  
92138271585054, 1.2264566781104924, 1.1499691902615938, 1.0695601050516346, 0.985  
0283578843675, 0.8961625753612537, 0.8027405467378665, 0.7045286682812861, 0.6012  
81359139088, 0.4927404472592949, 0.37863452382577156, 0.2586782645948129, 0.13257  
171643591636, 4.5270728232793545e-07, 0.1393697423114011, 0.28588464819467063, 0.  
4399115339380053, 0.6018355469847854, 0.7720615817100787, 0.9510152918674817, 1.1  
391441549452117, 1.3369185910929127, 1.5448331394170558, 1.7634076945863018, 1.99  
31888068389791, 2.234751048643352, 2.4886984514280868, 2.755666015975438, 3.03632  
1300253968, 3.3313660886611838, 3.641538146850069, 3.9676130665275076, 4.31040620  
4837532, 4.670774723178887, 5.049619730555012, 5.447888536815887, 5.8665770214261  
3, 6.306732123682359, 6.769454460606763, 7.25590107906296, 7.767288348975812, 8.3  
04895004889842, 8.870065343471628, 9.46421258495171, 10.088822406911273, 10.74545  
6659249928, 11.435757269624045, 12.161450349121074, 12.924350508436408, 13.726365  
395345292, 14.569500464815876, 15.455863993691304, 16.38767235247989, 17.36725554  
7435906, 18.397063046788855, 19.479669905689875, 20.61778320519096, 21.8142488213  
5759, 23.0720585414413, 24.394357544906132, 25.78445226801552, 27.24581867164526  
4, 28.782110932996176, 30.39717058294055, 32.09503611185026, 33.87995306792622, 3  
5.75638467328037, 37.729022984315634, 39.80280062431102, 41.98290311754907, 44.27  
478185582752, 46.68416772977842, 49.21708545907968, 51.87986865739247, 54.6791756  
6969442, 57.622006221610235, 60.71571892237158, 63.96804966517339, 67.38713097093  
61, 70.98151232384403, 74.76018154950914, 78.73258728921638, 82.90866262644883, 8  
7.29884992477005, 91.9141269391723, 96.7660342661821, 101.86670420136247, 107.228  
89107637177, 112.8660031514359, 118.79213614298321, 125.02210847027716, 131.57149  
830918377, 138.4566825457251, ...]  
10000 0.0001 [1.718281823928888, 1.6670107275528638, 1.6131109058532402, 1.5564  
47581200605, 1.496879065768718, 1.4342564072411466, 1.3684230163528848, 1.2992142  
753356306, 1.2264571262876176, 1.149969638438719, 1.0695605532287598, 0.985028806  
0614927, 0.8961630235383788, 0.8027409949149917, 0.7045291164584113, 0.6012818073  
162132, 0.4927408954364201, 0.37863497200289675, 0.2586787127719381, 0.1325721646  
1304155, 4.530157138304958e-09, 0.1393692941342759, 0.28588420001754544, 0.439911  
0857608801, 0.6018350988076602, 0.7720611335329535, 0.9510148436903565, 1.1391437  
067680865, 1.3369181429157875, 1.5448326912399306, 1.7634072464091766, 1.99318835  
8661854, 2.234750600466227, 2.4886980032509616, 2.755665567798313, 3.036320852076  
8428, 3.3313656404840586, 3.641537698672944, 3.9676126183503824, 4.31040575666040  
7, 4.6707742750017625, 5.049619282377886, 5.447888088638762, 5.866576573249005,  
6.306731675505234, 6.769454012429638, 7.255900630885836, 7.767287900798688, 8.304  
894556712718, 8.870064895294504, 9.464212136774584, 10.088821958734147, 10.745456  
211072803, 11.435756821446919, 12.16144990094395, 12.924350060259282, 13.72636494  
7168168, 14.569500016638752, 15.45586354551418, 16.387671904302763, 17.3672550992  
5878, 18.39706259861173, 19.47966945751275, 20.617782757013835, 21.81424837318046  
5, 23.072058093264175, 24.394357096729006, 25.784451819838395, 27.24581822346813

8, 28.78211048481905, 30.397170134763424, 32.09503566367314, 33.8799526197491, 35.75638422510325, 37.72902253613851, 39.80280017613389, 41.982902669371946, 44.274781407650394, 46.6841672816013, 49.21708501090255, 51.879868209215346, 54.6791752215173, 57.62200577343311, 60.71571847419445, 63.96804921699626, 67.38713052275897, 70.9815118756669, 74.76018110133201, 78.73258684103925, 82.9086621782717, 87.29884947659292, 91.91412649099517, 96.76603381800497, 101.86670375318533, 107.22889062819463, 112.86600270325877, 118.79213569480608, 125.02210802210003, 131.57149786100663, 138.45668209754797, ... ]

Como se puede ver, el método tiene un orden de convergencia cuadrático, por lo que la mejora que proponemos es correcta. Esto es fácil de comprobar en la tabla ya que, si dividimos por un factor de 10 la  $\Delta T$ , el error disminuye por un factor de  $10^2 = 100$ , pues  $\mathcal{O}((\Delta T)^2)$ .

## Exercise 7: Capitalization factor

Assume that at time  $t_0$  we make a bank deposit of  $B_0$ .

The differential equation that describes the evolution of the value of the deposit with time is

$$\frac{dB(t)}{dt} = rB(t), \quad (29)$$

where  $r$  is the (continuous) interest rate. Since it is a first order ODE, we need an *initial condition* so that the solution is uniquely determined. In this case, the initial condition is  $B(t_0) = B_0$ .

The closed-form solution of this equation can be obtained using the method of separation of variables:

$$dB(t) = rB(t)dt \implies \frac{dB(t)}{B(t)} = rdt \quad (30)$$

$$\implies \int_{B_0}^{B(t_0+T)} \frac{dB}{B} = r \int_{t_0}^{t_0+T} dt \quad (31)$$

$$\implies \log B \Big|_{B(t_0)}^{B(t_0+T)} = rt \Big|_{t_0}^{t_0+T} \quad (32)$$

$$\implies \log B(t_0 + T) - \log B(t_0) = r((t_0 + T) - t_0) \quad (33)$$

$$\implies \log \frac{B(t_0 + T)}{B(t_0)} = rT \quad (34)$$

$$\implies \frac{B(t_0 + T)}{B(t_0)} = e^{rT} \quad (35)$$

$$\implies B(t_0 + T) = B(t_0)e^{rT}. \quad (36)$$

1. Write down the algorithm that can be used to generate the trajectory by the Euler method in  $N$  steps for this particular problem. The general

El algoritmo a seguir es el mismo que el del ejercicio anterior, identificando  $a(t, B(t)) = rB(t)$ , con la condición inicial  $B(t_0) = B_0$ . Por tanto, el algoritmo es el siguiente:

$$f(t_0) = f_0 \quad (37)$$

$$f_{n+1} = f_n + a(t_n, f_n) \Delta T, \quad n = 0, 1, \dots, N-1. \quad (38) \Leftrightarrow$$

$$B(t_0) = B_0 \quad (39)$$

$$B_{n+1} = B_n + rB_n \Delta T, \quad n = 0, 1, \dots, N-1. \quad (40) \Leftrightarrow$$

$$B(t_0) = B_0 \quad (41)$$

$$B_{n+1} = B_n (1 + r\Delta T), \quad n = 0, 1, \dots, N-1. \quad (42)$$

Así pues, bastaría con emplear la función `euler_integration` definida en el ejercicio anterior, agregándole los argumentos pertinentes descritos en este apartado para obtener la evolución de nuestro dinero en un depósito a un tiempo  $T$ .

2. Find a closed-form solution of the *difference equation*.

Tras  $N$  iteraciones, la solución cerrada de nuestra EDO, i.e, el dinero que tendremos en el tiempo  $T$  será, observando el comportamiento de nuestra EDO particular:

$$\begin{aligned} B_1 &= B_0 (1 + r\Delta T), \\ B_2 &= B_1 (1 + r\Delta T) = B_0 (1 + r\Delta T)^2, \\ B_3 &= B_2 (1 + r\Delta T) = B_1 (1 + r\Delta T)^2 = B_0 (1 + r\Delta T)^3 \dots \end{aligned}$$

se puede ver por inducción que la solución final será  $B_N = B_0 (1 + r\Delta T)^N$ .

3. Show that in the limit  $N \rightarrow \infty$ , the solution of the difference equation tends to the solution of the differential equation.

Considerando que  $\Delta T = \frac{T}{N}$  y que la exponencial se define como  $e^x = \lim_{n \rightarrow \infty} (1 + \frac{x}{n})^n$ , es sencillo comprobar lo siguiente:

$$\lim_{N \rightarrow \infty} B_N = \lim_{n \rightarrow \infty} B_0 (1 + r\Delta T)^N = \lim_{n \rightarrow \infty} B_0 \left(1 + r \frac{T}{N}\right)^N = B_0 e^{rT}.$$

4. Illustrate numerically for  $B_0 = 100.0$ ,  $T = 2.0$ , the evolution of the approximation error as a function of  $N$  for the whole trajectory.

In [36]: # Evolution of the approximation error as a function of N

```
r = 0.03                                # Por ejemplo, r = 3%
a = lambda t, f: r * f
t_0, f_0, T = 0.0, 100.0, 2.0
true_value = f_0 * np.exp(r * T)

# Distintas iteraciones

N = [1, 10, 100, 1000, 10000, 100000]
dT = []
error = []
for n in N:
    t, f = func.euler_integration(a, t_0, f_0, T, n)
    dt = T / n
    err = abs(f[-1] - true_value)
    dT.append(dt)
```

```

error.append(err)

df = pd.DataFrame({
    "N": N,
    "delta_T": dT,
    "error": error,
    "valor real": np.repeat(true_value, len(N))
})

# Print table
print("Convergence Table:")
print(df.to_string(index=False))

```

Convergence Table:

N	delta_T	error	valor real
1	2.00000	0.183655	106.183655
10	0.20000	0.019035	106.183655
100	0.02000	0.001911	106.183655
1000	0.00200	0.000191	106.183655
10000	0.00020	0.000019	106.183655
100000	0.00002	0.000002	106.183655

4. Como era de esperar, el método tiene un orden de convergencia lineal, a medida que dividimos entre 10 el paso  $\Delta T$ , el error se divide por el mismo factor.

## Sample averages and the central Limit theorem (CLT).

Consider the random variable  $X$  whose probability density is  $\text{pdf}(x)$ .

Assume that we have  $\{X_m\}_{m=1}^M$ , an iid (independent, identically distributed) sample of size  $M$  of  $X$   $\text{pdf}(x)$ . The expected value of the deterministic function  $g(x)$  can be estimated as the average over the iid sample

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} dx \text{pdf}(x)g(x) \approx \langle g(X) \rangle_M = \frac{1}{M} \sum_{m=1}^M g(X_m).$$

By the law of large numbers,

$$\lim_{M \rightarrow \infty} \langle g(X) \rangle_M = \mathbb{E}[g(X)].$$

By the central limit theorem, under certain conditions (among others, that the standard deviation of the random variable  $g(X)$  be finite), the distribution of the random variable  $\langle g(X) \rangle_M$  approaches a Gaussian in the limit  $M \rightarrow \infty$

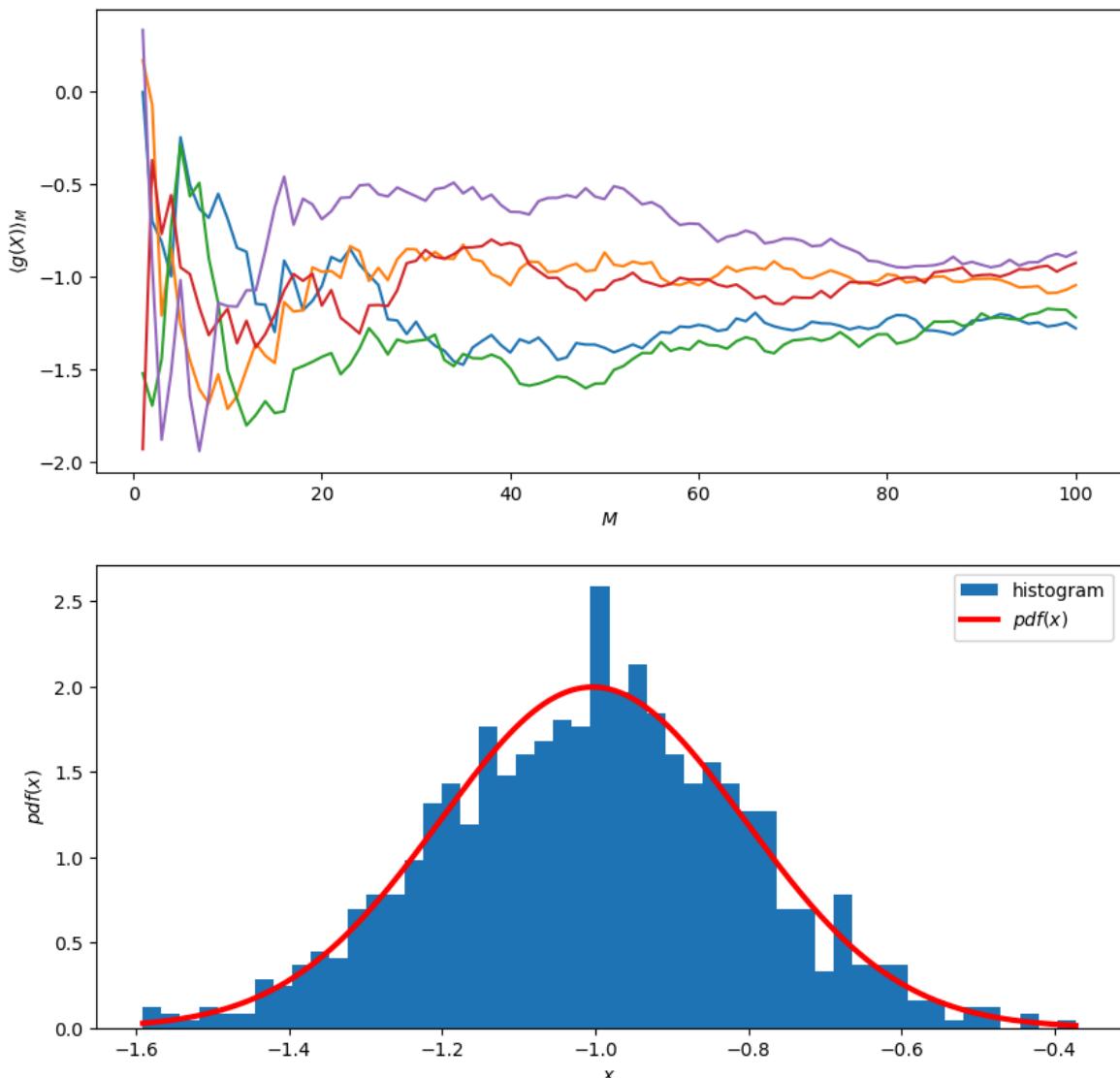
$$\langle g(X) \rangle_M \sim \mathcal{N}\left(\mathbb{E}[g(X)], \frac{1}{\sqrt{M}} \text{stdev}[g(X)]\right).$$

```
In [39]: # Example: Law of Large numbers & CLT for normally distributed random variables

mu, sigma = (-1.0, 2.0)

rng = default_rng()
```

```
random_number_generator = lambda size : rng.normal(size=size, loc=mu, scale=sigma)
func.demo_CLT(random_number_generator, sample_size=100, n_repetitions=1000, n_plot=5)
```



## Exercise 8: Law of large numbers and CLT.

Illustrate the validity of the law of large numbers and of the CLT with a graph as in the previous examples for the following cases:

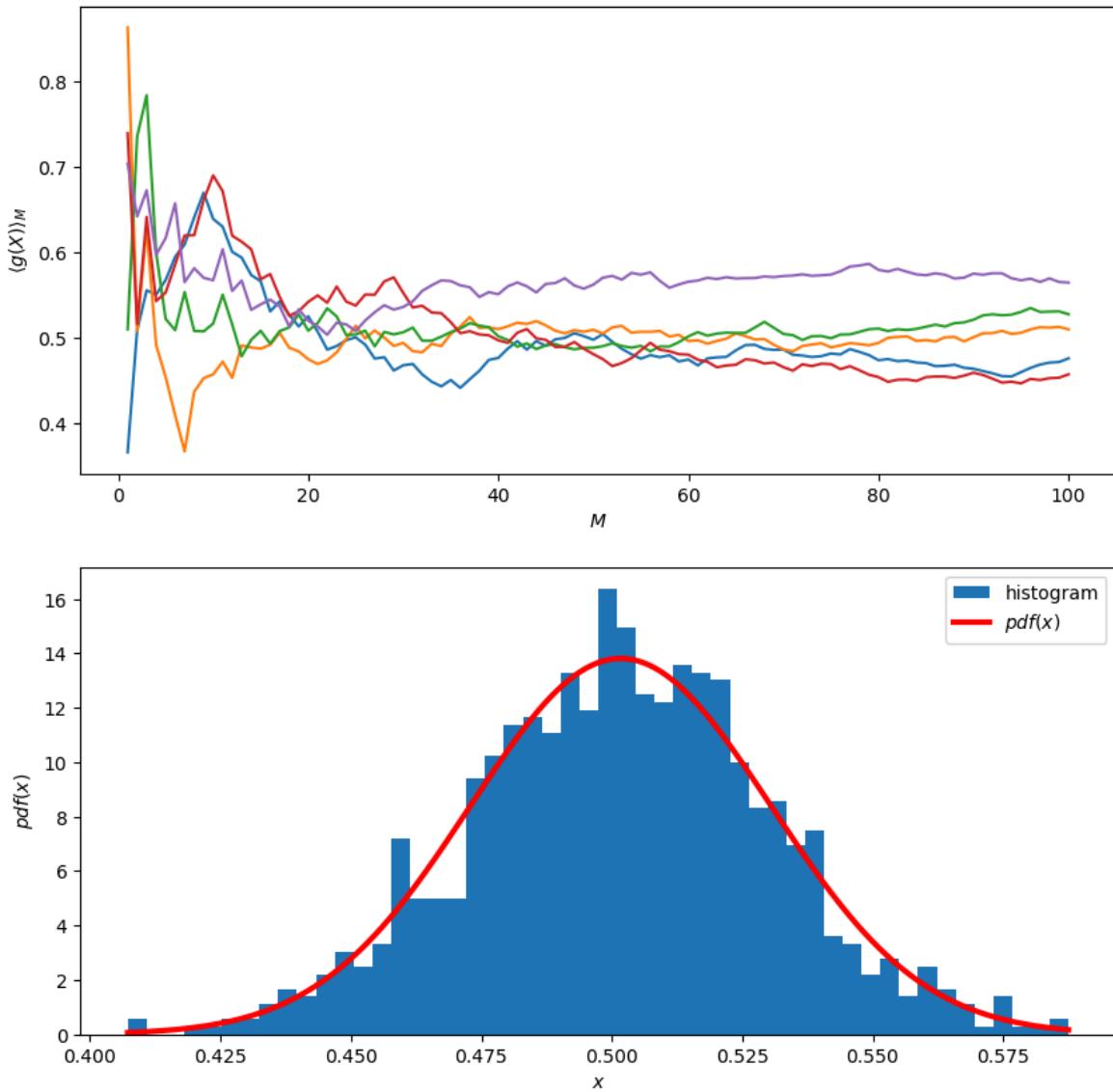
1. A uniform random variable in the interval  $[0, 1]$ .
2. A uniform random variable in the interval  $[-2.0, 3.0]$ .
3. A Bernoulli random variable with parameter  $p = 0.8$ .
4. A lognormal random variable with parameters  $\mu = -1.0$ ,  $\sigma = 0.5$ .
5. A lognormal random variable with parameters  $\mu = -1.0$ ,  $\sigma = 2.0$ .

Comment the results obtained.

```
In [41]: # Apartado 1

uniform01 = lambda size: np.random.uniform(0.0, 1.0, size)

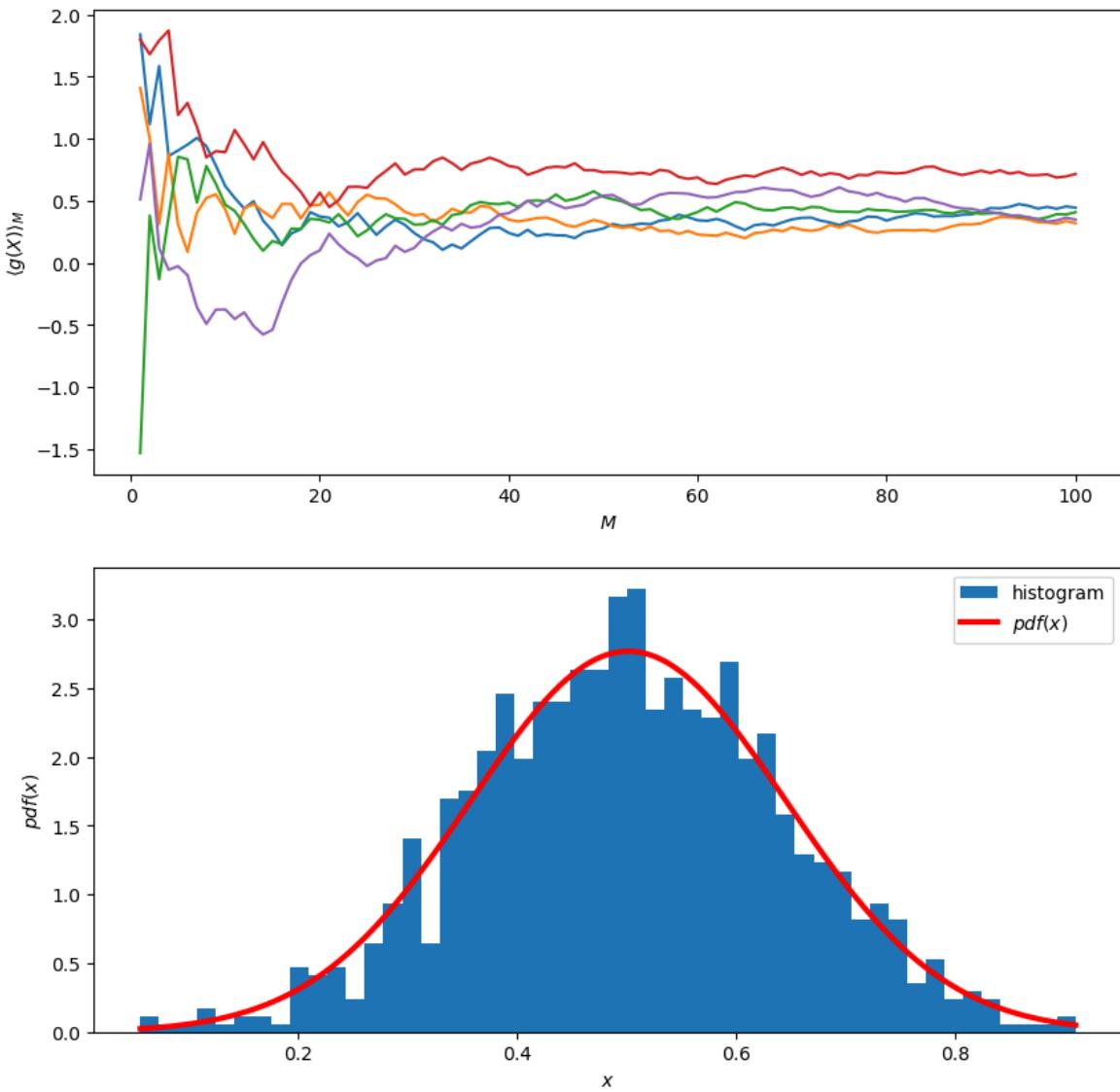
func.demo_CLT(uniform01, sample_size=100, n_repetitions=1000, n_plot=5)
```



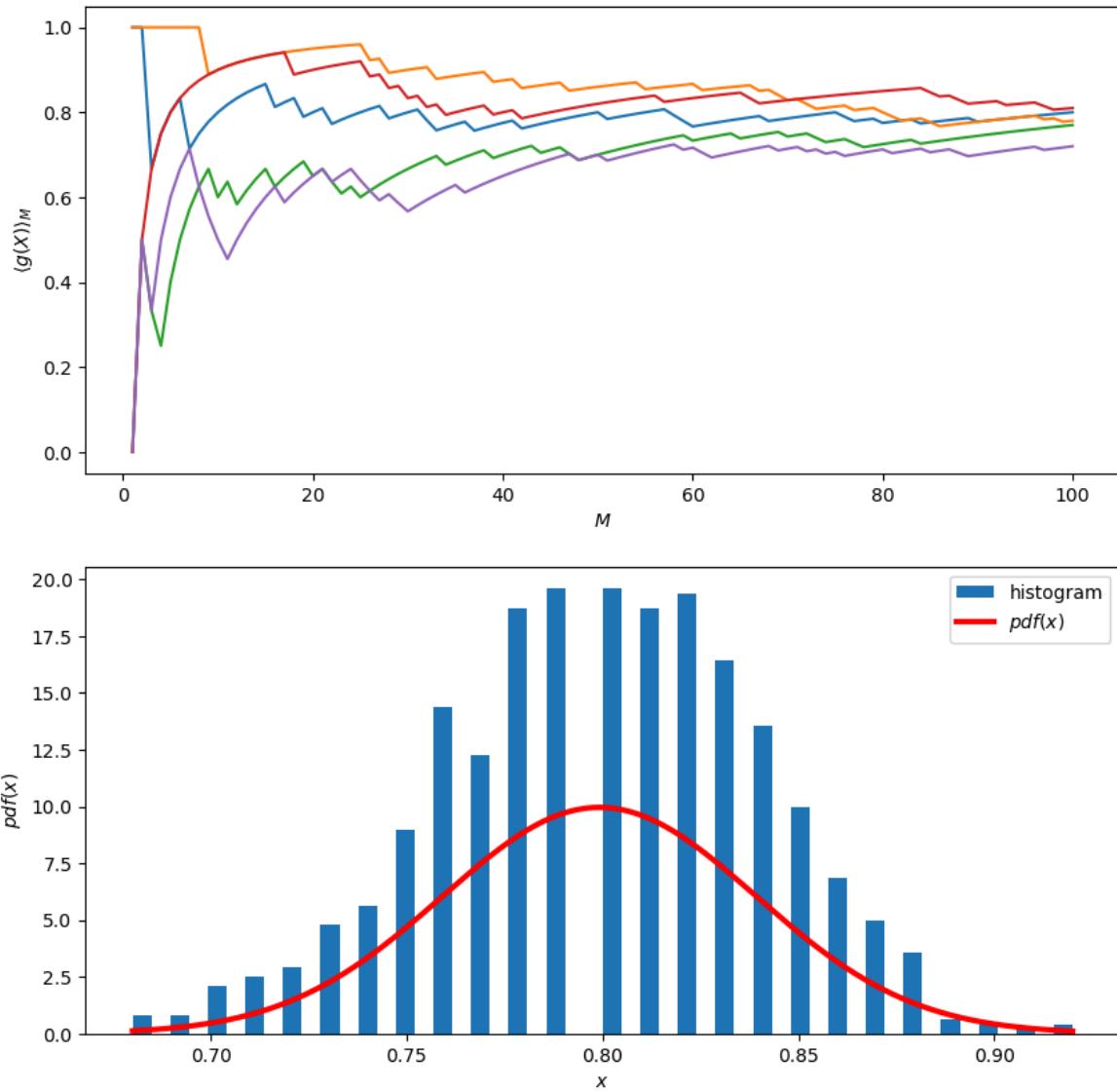
```
In [42]: # Apartado 2

uniform23 = lambda size: np.random.uniform(-2.0, 3.0, size)

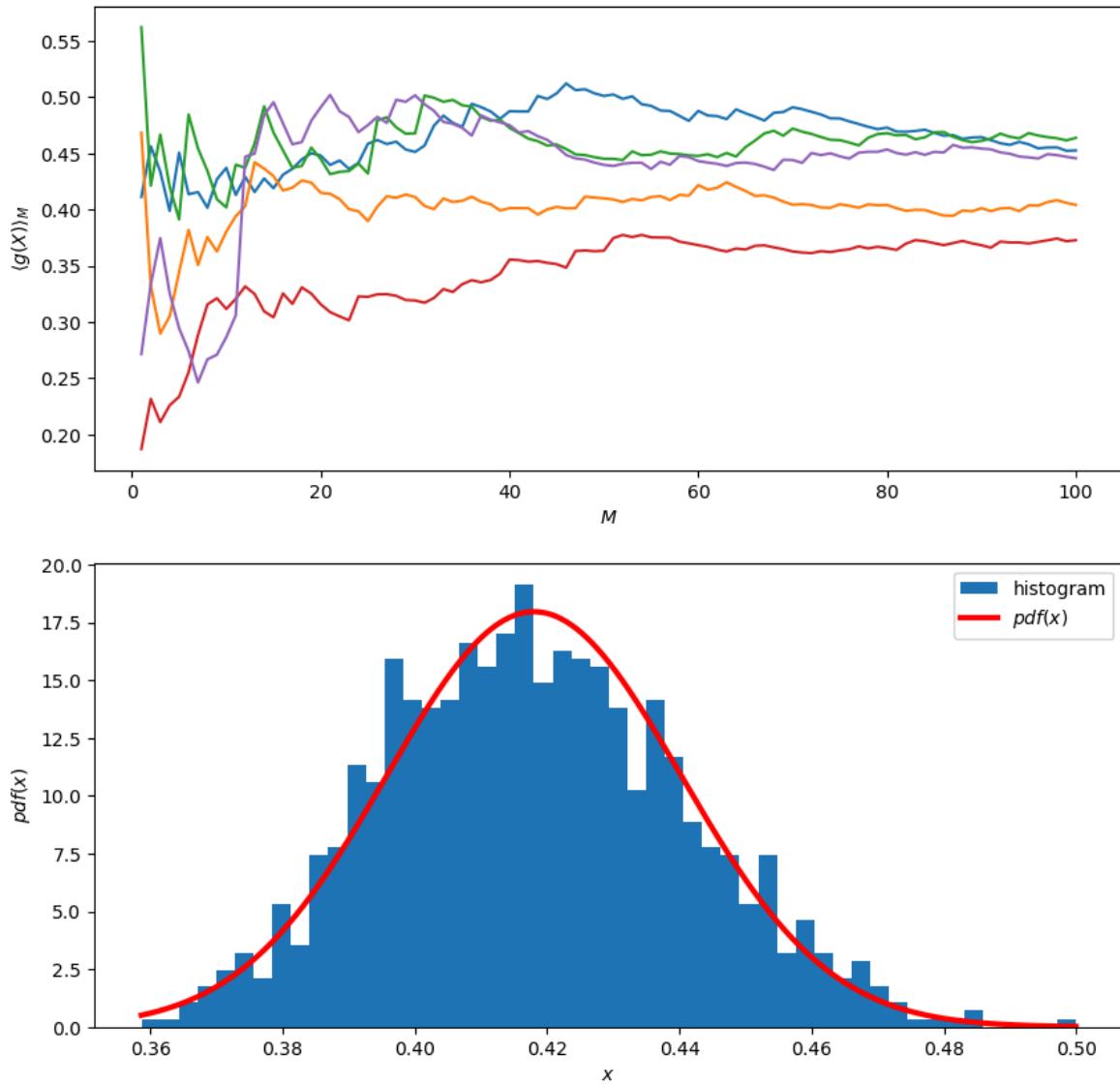
func.demo_CLT(uniform23, sample_size=100, n_repetitions=1000, n_plot=5)
```



```
In [43]: # Apartado 3
beroulli = lambda size: np.random.binomial(1, 0.8, size)
func.demo_CLT(beroulli, sample_size=100, n_repetitions=1000, n_plot=5)
```



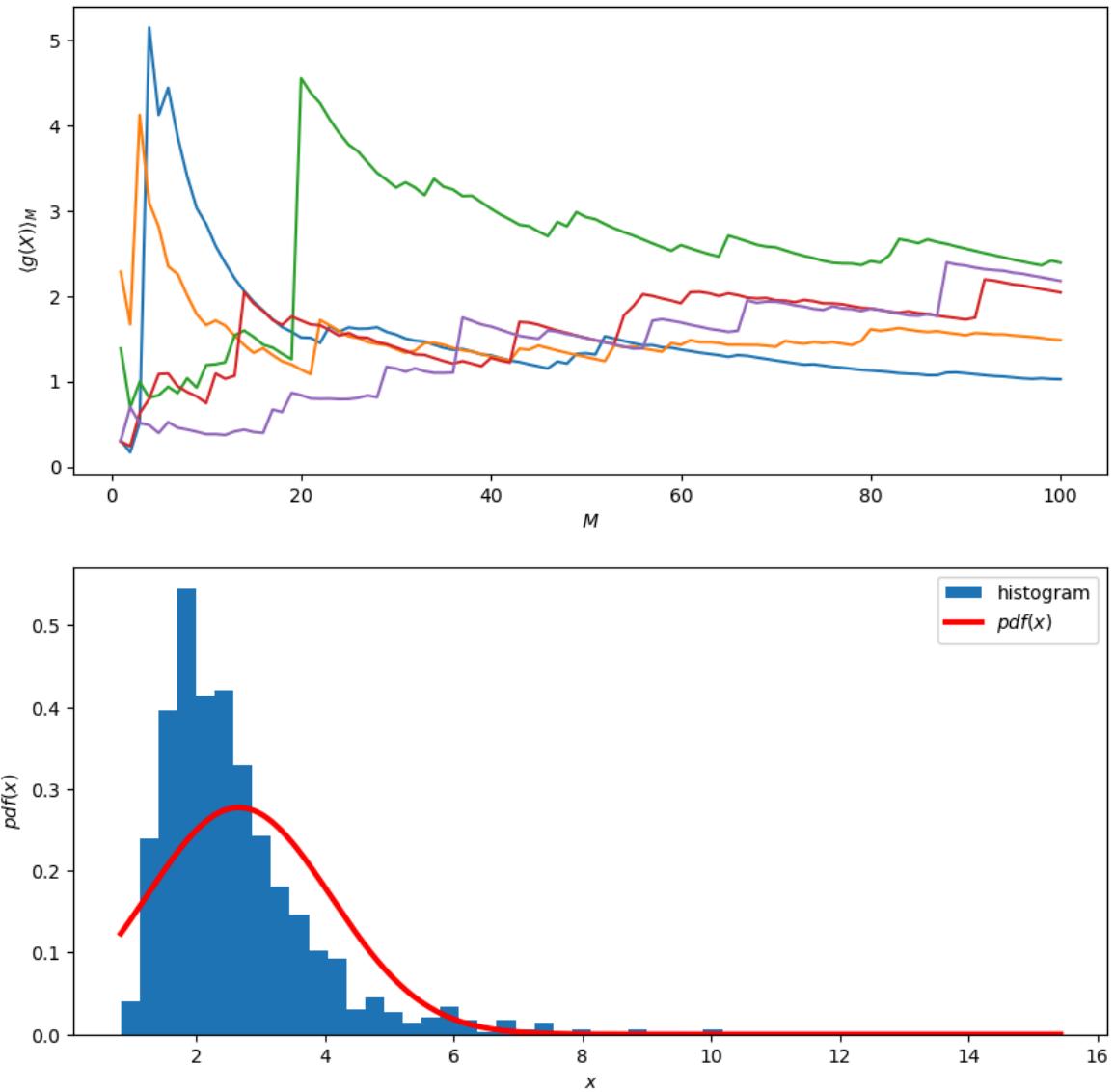
```
In [44]: # Apartado 4
lognormal1 = lambda size: np.exp(rng.normal(size=size, loc=-1.0, scale=0.5))
func.demo_CLT(lognormal1, sample_size=100, n_repetitions=1000, n_plot=5)
```



```
In [45]: # Apartado 5

lognormal2 = lambda size: np.exp(rng.normal(size=size, loc=-1.0, scale=2))

func.demo_CLT(lognormal2, sample_size=100, n_repetitions=1000, n_plot=5)
```



Según la ley de los grandes números, la media muestral resultante de  $N$  realizaciones de una variable aleatoria cualquiera con media  $\mu$  y desviación típica  $\sigma$  converge en distribución a una  $N(\mu, \frac{\sigma}{\sqrt{N}})$  cuando  $N \rightarrow \infty$ .

Generando realizaciones aleatorias de las diferentes variables, se observa como la media muestral tiende a la media poblacional a medida que el numero de realizaciones aumenta y como el histograma se aproxima a la función de densidad de una  $N(\mu, \sigma)$ .

La aproximación a la normal se da en diferente medida para las distintas variables. Por ejemplo, con 1000 repeticiones del experimento consistente en obtener 100 observaciones, el histograma de la uniforme se ajusta relativamente bien a la normal, mientras que en el caso de la binomial con  $p = 0.8$  o el de la log-normal con  $\mu = -1$  y  $\sigma = 2$  el ajuste es peor.

La distribución binomial es discreta, mientras que la normal es continua, por lo que el histograma queda por encima de la curva normal en todos los puntos. Esto se debe a que la normal representa densidad y su masa está repartida sobre un intervalo continuo, mientras que la binomial concentra toda su probabilidad en valores enteros, haciendo que su masa aparezca más concentrada visualmente. Además, con  $p = 0.8$  la binomial es asimétrica, lo cual empeora aún más el ajuste, ya que la normal es simétrica.

La distribución log-normal es positiva y asimétrica con una cola derecha más pesada, en mayor medida cuanto más grande sea  $\sigma$ , por tanto, la convergencia a la normal (distribución simétrica) es lenta cuanto mayor sea el valor del parámetro. Se puede observar como el histograma de las realicaciones de una log-normal con  $\sigma = 0.5$  se ajusta mejor a la normal que el de la log-normal con  $\sigma = 2$ .

## Bibliografía

OpenAI. (2025). ChatGPT (May 02 version) [Large language model]. <https://chatgpt.com/>

Universidad de Santiago de Compostela. (2023). Apuntes de derivación numérica I. Departamento de Matemática Aplicada, Universidad de Santiago de Compostela.

Departamento de Matemática Aplicada. (2015). Cálculo numérico nunha variable: Tema 2. Aproximación de raíces dunha ecuación numérica. Universidade de Santiago de Compostela, Grao en Matemáticas, segundo curso.

Álvarez Dios, J. A. (2022). Convergencia global del método de Newton-Raphson. Universidad de Santiago de Compostela.