
A small, stylized icon of a pen or pencil tip, pointing upwards and to the right, positioned at the end of a horizontal line.

1.1 Calcular el error absoluto y relativo de las siguientes aproximaciones a p por p^* :

- a) $p = e^{10}$, $p^* = 22000$,
- b) $p = 10^\pi$, $p^* = 1400$,
- c) $p = 8!$, $p^* = 39900$,
- d) $p = 9!$, $p^* = \sqrt{18\pi}(9/e)^9$.

$p \equiv$ Queremos calcular

$p^* \equiv$ Aproximación de p

Comparamos errores

ERROR ABSOLUTO

$$\epsilon_A = |p - p^*|$$

ERROR RELATIVO

$$\epsilon_r = \frac{|p - p^*|}{|p|} = \frac{\epsilon_A}{|p|}$$

```
import numpy as np
import math
from os import system

system("cls")

p = [np.exp(10), 10**np.pi, math.factorial(8), math.factorial(9)]
q = [22000, 1400, 39900, math.sqrt(18*np.pi)*(9/np.exp(1))**9]

for i in range(len(p)):
    print(f"Error absoluto: {abs(p[i]-q[i])}")
    print(f"Error relativo: {abs(p[i]-q[i])/abs(p[i])}")
    print(f"Error porcentual: {abs(p[i]-q[i])/abs(p[i])*100}%")
    print()
```

```
Error absoluto: 26.465794806717895
Error relativo: 0.0012015452253333286
Error porcentual: 0.12015452253333286%
```



```
Error absoluto: 14.544268632989315
Error relativo: 0.010497822704619136
Error porcentual: 1.0497822704619135%
```



```
Error absoluto: 420
Error relativo: 0.010416666666666666
Error porcentual: 1.0416666666666665%
```



```
Error absoluto: 3343.1271580516477
Error relativo: 0.009212762230080598
Error porcentual: 0.9212762230080598%
```

a) $|e^{10} - 22000| = 26,46$

$$\epsilon_r = 0,0012 \Rightarrow 0,1\%$$

b) $|10^\pi - 1400| = 19,54$

$$\epsilon_r = 0,0105 \Rightarrow 1\%$$

c) $|8! - 39900| = 420$

$$\epsilon_r = 0,0104 \Rightarrow 1\%$$

d) $|9! - \sqrt{18\pi}(\frac{9}{e})^9| = 3343,13$

$$\epsilon_r = 0,0092 \Rightarrow 1\%$$

1.2 Encuentre el intervalo más grande en donde p^* puede estar de forma que el error relativo al aproximar los siguientes p no sea mayor a 10^{-4} .

- a) π ,
- b) e ,
- c) $\sqrt{2}$,
- d) $7^{1/3}$.

$$\epsilon_r = 10^{-4} = \frac{1}{10000} = \frac{|p - p^*|}{|p|}$$

Buscamos

$$\frac{p - p^*}{p} < \frac{1}{10000} \quad \vee \quad \frac{p - p^*}{p} > -\frac{1}{10000}$$

$$p^* \in \left(p - \frac{p}{10000}, p + \frac{p}{10000} \right)$$

- a) p^* esta entre 3.141278494324434 y 3.141906812855152
- b) p^* esta entre 2.718010000276199 y 2.718553656641891
- c) p^* esta entre 1.4140721410168577 y 1.4143549837293325
- d) p^* esta entre 1.9127398896541117 y 1.9131224758906662

```
import numpy as np
import math
from os import system

system("cls")

p = [np.pi, np.e, math.sqrt(2), 7**(1/3)]

for i in range(len(p)):
    print(f"p* esta entre {p[i] - p[i]/10000} y {p[i] + p[i]/10000}")
```

1.3 El número π se puede obtener a partir de las siguientes igualdades.

- a) $\pi = 4 \left[\arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right) \right]$,
- b) $\pi = 16 \arctan\left(\frac{1}{5}\right) - 4 \arctan\left(\frac{1}{239}\right)$.

Para cada uno de los casos, calcule una aproximación a π considerando los primeros 3 términos no nulos de la serie de Taylor de la arcotangente

$$\arctan(x) \approx x - (1/3)x^3 + (1/5)x^5,$$

Usando todos los dígitos disponibles en la calculadora. ¿Cuál fórmula llevó a un menor error? Argumente la razón de la diferencia.

```
import numpy as np
import math
from os import system

system("cls")

vals = [1/2, 1/3, 1/5, 1/239]

arctans = []
for val in vals:
    arctans.append(val - (1/3)*val**3 + (1/5)*val**5)

a = 4*(arctans[0] + arctans[1])
b = 16*arctans[2] - 4*arctans[3]

print(f"Error absoluto de a: {abs(np.pi - a)}")
print(f"Error absoluto de b: {abs(np.pi - b)}")
print()
print(f"Error relativo de a: {abs(np.pi - a)/abs(np.pi)}")
print(f"Error relativo de b: {abs(np.pi - b)/abs(np.pi)}")
print()
print(f"Error porcentual de a: {abs(np.pi - a)/abs(np.pi)*100}%")
print(f"Error porcentual de b: {abs(np.pi - b)/abs(np.pi)*100}%")
```

```
Error absoluto de a: 0.003983478097449478
Error absoluto de b: 2.837573524150372e-05

Error relativo de a: 0.0012679804598147663
Error relativo de b: 9.032277055104427e-06

Error porcentual de a: 0.12679804598147662%
Error porcentual de b: 0.0009032277055104426%
```

La approx de b es mejor porque genera un menor error relativo

1.4 Demostrar que:

a) $\cos(x) = 1 - \frac{x^2}{2} + O(x^4)$

b) $\sin(x) = x - \frac{x^3}{6} + O(x^5)$

a) Por Taylor orden 3

$$\cos(x) = 1 - \frac{1}{2}x^2 + R_4(x)$$

error Taylor
orden 4

Siendo $R_4(x) = \left| \frac{f''(c)}{24} x^4 \right| \stackrel{\leq 1}{\leq} \frac{|x^4|}{24} = \frac{x^4}{24}$

\Rightarrow Igualando $\cos(x)$

$$\cancel{1 - \frac{1}{2}x^2 + R_4(x)} = 1 - \frac{x^2}{2} + O(x^4)$$

$$R_4(x) = O(x^4) \leq \frac{x^4}{24} \rightarrow \text{El error tiene orden 4}$$

b) Mismo idea

$$\sin(x) \approx x - \frac{x^3}{6} + [R_5(x)] = \left| \frac{f'''(c) x^5}{120} \right| \leq \frac{|x^5|}{120}$$

\Rightarrow El error tiene $O(x^5)$ (orden 5)

1.5 Encuentre la tasas de convergencia de las siguientes secuencias cuando $n \rightarrow \infty$.

- a) $\lim_{n \rightarrow \infty} \sin \frac{1}{n} = 0,$
- b) $\lim_{n \rightarrow \infty} \sin \frac{1}{n^2} = 0,$
- c) $\lim_{n \rightarrow \infty} (\sin \frac{1}{n})^2 = 0,$
- d) $\lim_{n \rightarrow \infty} [\ln(n+1) - \ln(n)] = 0.$

a) Desarrollo Taylor del $\sin(x)$

$$\sin(x) = x - \frac{x^3}{6} + R_s(x) \xrightarrow{\text{O}(x^5)} \text{por el 1.4}$$

$$\text{Reemplazando } \frac{1}{n} \rightarrow \sin\left(\frac{1}{n}\right) = \frac{1}{n} - \frac{1}{6n^3} + O\left(\frac{1}{n^5}\right)$$

Tenemos que $\frac{1}{n}$ es el término dominante,
ya que es el que decrece más lento cuando
 $n \rightarrow \infty$

Por lo que nuestra tasa de convergencia es $\alpha=1$
porque es lineal respecto a n

c) Misma mierda pero al cuadrado,

$$\text{Si } \sin\left(\frac{1}{n}\right) \rightarrow \frac{1}{n}, \text{ entonces } \sin^2\left(\frac{1}{n}\right) \rightarrow \frac{1}{n^2}$$

Pero $\alpha=1$ porque sigue siendo lineal

b)

Ahora reemplazamos Taylor con $\frac{1}{n^2}$

$$\operatorname{sen}\left(\frac{1}{n^2}\right) = \frac{1}{n^2} - \frac{1}{6n^6} + O\left(\frac{1}{n^{10}}\right)$$

$\Rightarrow \alpha=1$ porque el término dominante es $\frac{1}{n^2}$
y sigue siendo lineal

d) Tenemos

$$\ln(1+n) - \ln(n) = \ln\left(\frac{1+n}{n}\right) = \ln\left(1 + \frac{1}{n}\right)$$

Desarrollando Taylor centrado en 0

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

Reemplazo $\frac{1}{n}$

$$\ln\left(1 + \frac{1}{n}\right) = \frac{1}{n} - \frac{1}{2n^2} + \frac{1}{6n^3} + O\left(\frac{1}{n^4}\right)$$

$\Rightarrow \alpha=1$ porque $\frac{1}{n}$ es dominante

1.6 Encuentre numéricamente en *python* el valor del epsilon de máquina en *doble precision*, *simple precision* y *half precision*.

epsilon de máquina = num mas chico tal que
 $f\lfloor(1.0) + \underline{\text{epsilon}} = f\lceil(1.0)$

función
float

Resumen de precisiones

Tipo de precisión	Bits	Nombre en numpy	Precisión aproximada (ϵ)	Uso típico
Half precision	16	np.float16	~ 0.00098	GPUs, IA
Simple precision	32	np.float32	~ 1.19e-07	Juegos, ML
Double precision	64	np.float64	~ 2.22e-16	Ciencia, finanzas

Código del chat

```
# Función para calcular el epsilon de máquina
def calcular_epsilon(dtype):
    epsilon = dtype(1.0)
    while dtype(1.0) + epsilon != dtype(1.0):
        epsilon_anterior = epsilon
        epsilon = dtype(epsilon / 2.0)
    return epsilon_anterior

# Cálculo para cada tipo de precisión
eps_double = calcular_epsilon(np.float64)
eps_simple = calcular_epsilon(np.float32)
eps_half   = calcular_epsilon(np.float16)

print("Epsilon double precision (float64):", eps_double)
print("Epsilon simple precision (float32):", eps_simple)
print("Epsilon half precision (float16):", eps_half)
```

np.floatN es una clase que representa un flotante de N bits

```
Epsilon double precision (float64): 2.220446049250313e-16
Epsilon simple precision (float32): 1.1920929e-07
Epsilon half precision (float16): 0.000977
```

1.7 Utilice aritmética de punto flotante redondeando a tres dígitos para realizar los siguientes cálculos.
Calcule el error absoluto y relativo, con el valor exacto determinado a al menos cinco dígitos.

- a) $133 + 0,921$,
- b) $133 - 0,499$,
- c) $(121 - 0,327) - 119$,
- d) $(121 - 119) - 0,327$.

a) $f1(133) = 0,133 \times 10^3$

$$f1(0,921) = 0,921 \times 10^0$$

$$\Rightarrow f1(0,133 \times 10^3 + 0,921 \times 10^0) =$$

$$f1(133,921) = \boxed{0,134 \times 10^3}$$

$$E_a = |133,921 - 134| = 0,079$$

$$E_r = \frac{0,079}{133,921} = 5,9 \times 10^{-4}$$

b) $f1(0,133 \times 10^3 - 0,499 \times 10^0)$

$$f1(132,501) = \boxed{0,133 \times 10^3}$$

$$E_a = |132,501 - 133| = 0,499$$

$$E_r = \frac{0,499}{132,501} = 3,77 \times 10^{-3}$$

c) $f| (0,121 \times 10^3 - 0,327 \times 10^0) =$

$$f| (121,327) = 0,121 \times 10^3$$

$$f| (0,121 \times 10^3 - 0,119 \times 10^3) = \boxed{0,002 \times 10^3}$$

$$\epsilon_A = |1,673 - 2| = 0,327$$

$$\epsilon_r = \frac{0,327}{1,673} = 0,196$$

d) $f| (0,121 \times 10^3 - 0,119 \times 10^3) = 0,002 \times 10^3$

$$f| (0,002 \times 10^3 - 0,327 \times 10^0) =$$

$$f| (1,673) = 0,167 \times 10^1$$

$$\epsilon_A = |1,673 - 1,67| = 0,003$$

$$\epsilon_r = \frac{0,003}{1,673} = 1,79 \times 10^{-3}$$

1.8 Suponga que $fl(y)$ es una aproximación de redondeo de k -dígitos a y . Muestre que

$$\left| \frac{y - fl(y)}{y} \right| \leq 0,5 \times 10^{-k+1}.$$

Ayuda: si $d_{k+1} < 5$, entonces $fl(y) = 0.d_1d_2 \dots d_k \times 10^n$, si $d_{k+1} \geq 5$ entonces $fl(y) = 0.d_1d_2 \dots d_k \times 10^n + 10^{n-k}$.

Partimos de $y = 0.\underline{d}_1d_2d_3 \dots \times 10^n$ (n) → no importa estos n pero es lo que se movió hipotéticamente la coma
 $d_1 \neq 0 \in \mathbb{N}$
 $\Rightarrow 0.d_1d_2d_3 \dots \geq 0,1$

Por lo que llegamos a $y \geq 0,1 \times 10^n = 10^{n-1}$ guarden esto

Ahora llamamos $t = y$ truncado en k

O sea $t = 0.d_1d_2d_3 \dots d_k \times 10^n$

Por lo que

$$fl(y) = \begin{cases} t(y) & \text{si } d_{k+1} < s \\ t(y) + 10^{n-k} & \text{si } d_{k+1} \geq s \end{cases} \quad \boxed{\text{la Ayuda del enunciado}}$$

Además ahora podemos escribir y como

$$y = \underbrace{t(y)}_{\text{parte truncada}} + \underbrace{0,000\dots d_{k+1}d_{k+2}\dots \times 10^n}_{\substack{\text{acá hay } n \text{ ceros (lo muestro en prox pag)} \\ \text{el resto de decimales que quedan afuera.}}}$$

Ej para ver lo de recien para $k=3$

con $\gamma = 3,14159265 \rightarrow$ sí, es π, son reales originales

$$\gamma = 0,314159265 \times 10^1 \rightarrow n=1$$

$$0,314 \times 10^1 + 0,000159265 \times 10^1 \\ t_s(\gamma) \qquad \qquad \qquad 0,159265 \times 10^1 \times 10^{-3} \\ 0,159265 \times 10^{n-k}$$

$$\Rightarrow \gamma = t(\gamma) + 10^{n-k} (0, d_{k+1} d_{k+2} \dots)$$

a este numero de miles lo llamo θ

Crean que es igual para cualquier numero

Juntamos verdades por casos

Si $d_{k+1} < s$

$$\Rightarrow |\gamma - f(\gamma)| = |t(\gamma) + \theta \times 10^{n-k} - t(\gamma)| \\ = |\theta \times 10^{n-k}| \leq 0,5 \times 10^{n-k}$$

Como $d_{k+1} < s \Rightarrow \theta < 0,5$

Si $d_{k+1} \geq s$

$$|\gamma - f_1(\gamma)| = \left| \cancel{\gamma} + 10^{n-k}\theta - (\cancel{\gamma} + 10^{n-k}) \right| \\ = |10^{n-k}\theta - 10^{n-k}| = |10^{n-k}(\theta - 1)|$$

como dijimos que $d_{k+1} \geq s \Rightarrow \theta \geq 0,5$

por modulo
puedo dar
vueltas

$$\theta - 1 \geq 0,5 - 1$$

$$1 - \theta \leq 0,5$$

$$\Rightarrow |10^{n-k}(1-\theta)| \leq 0,5 \times 10^{n-k}$$

Por lo que en ambos casos

$$|\gamma - f_1(\gamma)| \leq 0,5 \times 10^{n-k}$$

te dice que
lo guardes

Recuperaremos

$$|\gamma| > 10^{n-1}$$

$$\Rightarrow \frac{|\gamma - f_1(\gamma)|}{|\gamma|} \leq \frac{0,5 \times 10^{n-k}}{|\gamma|}$$

$$\left| \frac{\gamma - f_1(\gamma)}{\gamma} \right| \leq 0,5 \times 10^{n-k} \times 10^{1-n}$$

$$\frac{1}{|\gamma|} \leq 10^{1-n}$$

le daría crédito al chat
pero como ni a él le en-
tendí me tuve que cargar
2 trampas, pero salió !!

$$\left| \frac{\gamma - f_1(\gamma)}{\gamma} \right| \leq 0,5 \times 10^{1-n}$$

1.9 El desarrollo de Taylor de la función e^x proporciona una forma muy inestable de calcular este valor cuando x es negativo. Evalúe numéricamente el desarrollo de Taylor hasta grado n de la función e^x en $x = -12$, para $n = 1, \dots, 100$. Comparar con el valor exacto: 0,000006144212353328210... ¿Cuáles son las principales fuentes de error? Proponer un método alternativo para estimar e^{-12} . Verificar si la aproximación obtenida es mejor.

```
exactVal = np.exp(-12)

def taylor(n):
    val = 0
    for i in range(n+1):
        val += (-1)**i / math.factorial(i)
    return val

vals = [taylor(n) for n in range(1, 101)]
errs = [abs(exactVal - val) for val in vals]

print(f"Valor exacto: {exactVal}")
print()
print(f"Menor error hallado con n = {errs.index(min(errs)) + 1}: {min(errs)}")
print(f"Valor encontrado: {vals[errs.index(min(errs))]}")
print()
print(f"Mayor error hallado con n = {errs.index(max(errs)) + 1}: {max(errs)}")
print(f"Valor encontrado: {vals[errs.index(max(errs))]}")
```

Principales fuentes de error al usar el desarrollo de Taylor para e^x con $x = -12$

1. Cancelación catastrófica

- La serie de Taylor centrada en 0 para e^{-12} es:

$$e^{-12} \approx \sum_{k=0}^n \frac{(-1)^k}{k!}$$

- Los primeros términos son muy grandes y alternan de signo.
- El valor real $e^{-12} \approx 6.14 \times 10^{-6}$ resulta de restar números enormes que casi se anulan.
- Esto provoca pérdida de muchas cifras significativas.

2. Acumulación de errores de redondeo

- Cada suma intermedia sufre redondeos por la aritmética de punto flotante (incluso en `float64`).
- Con más términos (n grande), estos redondeos se acumulan.

3. Términos inútiles para la convergencia

- Cuando x está lejos del centro (0), la serie converge **muy lentamente**.
- Muchos términos contribuyen poco al resultado pero si añaden error numérico.

4. Límites de la precisión finita

- En `float64`, hay unas ≈ 16 cifras significativas de precisión.
- Cuando se suman números del orden de 10^8 y 10^{-6} , el término pequeño se " pierde" dentro del grande.

Alternativo (Del gran Pato)



Arrancamos approximando el valor de e con

$$e \approx \sum_{k=1}^n \frac{1}{k!} \quad \text{con } n \text{ muy grandes}$$

```
exactVal = np.exp(-12)

def e(n):
    val = 0
    factorials = {}
    for i in range(n+1):
        factorials[i] = 1 if i == 0 else i * factorials[i-1]
        val += 1/factorials[i]
    return val

aproximacion = e(100000)
print(f"Valor approximado de e: {aproximacion}")
print(f"Error absoluto sin elevar: {abs(np.e - aproximacion)}")
print(f"e^-12 approximado = {aproximacion**-12}")
print(f"Error absoluto al elevar: {abs(exactVal - aproximacion**-12)})")
```

Una vez tenemos nuestro
aprox de e , elevamos eso
a -12 y comparamos

Resultados con Taylor

Valor exacto: 6.14421235332821e-06

Menor error hallado con n = 53: 2.521159347371355e-13

Valor encontrado: 6.1442126054441445e-06

Mayor error hallado con n = 11: 9504.789097053303

Valor encontrado: -9504.789090909091



→ La mejor aprox
de Taylor salió
con un error
 $\times 10^{-13}$

Resultados aproximando e

Valor aproximado de e: 2.7182818284590455

Error absoluto sin elevar: 4.440892098500626e-16

e^{-12} aproximado = 6.144212353328201e-06

Error absoluto al elevar: 8.470329472543003e-21



→ Nuestra alterna
tiva approxima
con error $\times 10^{-21}$

⇒ La alternativa de Pato es estupidamente mejor

1.10 Considera la función $f(x) = \frac{1-\cos(x)}{x^2}$ y analice su comportamiento numérico para valores de x cercanos a cero, $x = 10^{-k}$, $k = 1, \dots$. Evalúa la función en doble precisión para una serie de valores decrecientes de x y compara los resultados con el valor límite $\lim_{x \rightarrow 0} f(x) = \frac{1}{2}$ (no olvide usar escala logarítmica para visualizar). Justifique la pérdida de precisión observada y reformule la expresión de $f(x)$ de manera algebraicamente equivalente que mejore la estabilidad numérica. Compare los resultados obtenidos con ambas expresiones.

```
def f(x):
    # Convertir x a doble precisión
    x = np.float64(x)

    # Calcular cos(x) con doble precisión
    cos_x = np.float64(math.cos(x))

    # Calcular (1 - cos(x)) / x^2 con doble precisión
    numerator = np.float64(1.0) - cos_x
    denominator = x * x

    return np.float64(numerator / denominator)

for i in range(1, 11):
    x = np.float64(1.0 / (10**i))
    result = f(x)
    print(f"f(10^{i-1}) = {result}")
```

```
f(10^-1) = 0.49958347219741783
f(10^-2) = 0.4999958333473664
f(10^-3) = 0.4999995832550326
f(10^-4) = 0.499999969612645
f(10^-5) = 0.5000000413701854
f(10^-6) = 0.5000444502911705
f(10^-7) = 0.4996003610813205
f(10^-8) = 0.0
f(10^-9) = 0.0
f(10^-10) = 0.0
```

da 0
porque
con numrs
tan chicos

math.cos(x) redondea a 1.0

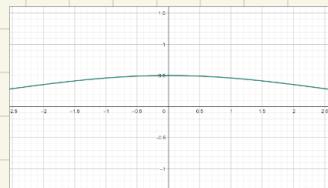
Calculo

$$f(x) = \frac{1 - \cos(x)}{x^2}$$

con doble precisión

Observamos que a menor la x ,
 $f(x)$ se acerca a 0,5

Por lo que $\lim_{x \rightarrow 0} f(x) = \frac{1}{2}$



Ahora calculamos los errores de $\kappa \in [1, 7]$, $\kappa \in \mathbb{N}$

```
def f(x):
    # Convertir x a doble precisión
    x = np.float64(x)

    # Calcular cos(x) con doble precisión
    cos_x = np.float64(math.cos(x))

    # Calcular (1 - cos(x)) / x^2 con doble precisión
    numerator = np.float64(1.0) - cos_x
    denominator = x * x

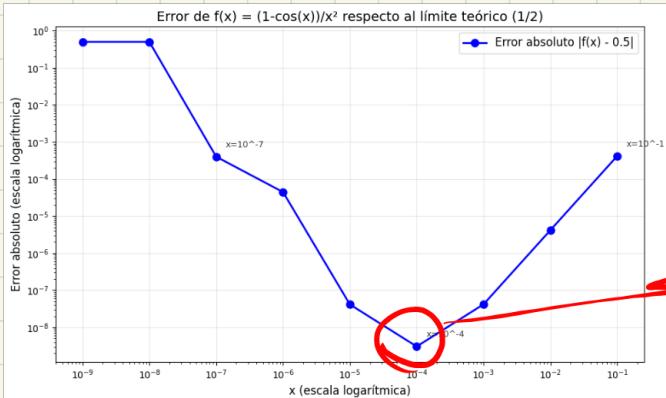
    return np.float64(numerator / denominator)

for i in range(1, 8):
    x = np.float64(1.0 / (10**i))
    result = f(x)
    print(f"Error cometido con (x = 10^{i-1}): {abs(result - 0.5)}")
```

```
Error cometido con (x = 10^-1): 0.00041652780258216726
Error cometido con (x = 10^-2): 4.166652633585954e-06
Error cometido con (x = 10^-3): 4.167449674241652e-08
Error cometido con (x = 10^-4): 3.038735485461075e-09
Error cometido con (x = 10^-5): 4.1370185388522884e-08
Error cometido con (x = 10^-6): 4.445029117050581e-05
Error cometido con (x = 10^-7): 0.0003996389186795013
```

→ mejor
aprox
conseguido

Analizando gráficamente (con escala logarítmica)



Código del gráfico:

```
import matplotlib.pyplot as plt

# Generar valores de x = 10^{(-k)} para k = 1, 2, ..., 15
k_values = range(1, 16)
x_values = [10**(-k) for k in k_values]
errors = []

# Calcular el error para cada valor de x
for x in x_values:
    result = f(x)
    error = abs(result - 0.5) # Error respecto al límite teórico lim_{x->0} f(x) = 1/2
    errors.append(error)

# Crear el gráfico en escala logarítmica
plt.figure(figsize=(10, 6))
plt.loglog(x_values, errors, 'bo-', linewidth=2, markersize=8, label='Error absoluto |f(x) - 0.5|')
plt.xlabel('x (escala logarítmica)', fontsize=12)
plt.ylabel('Error absoluto (escala logarítmica)', fontsize=12)
plt.title('Error de  $f(x) = (1-\cos(x))/x^2$  respecto al límite teórico (1/2)', fontsize=14)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)

# Agregar anotaciones para algunos puntos críticos
for i, (x, error) in enumerate(zip(x_values, errors)):
    if i % 3 == 0: # Anotar cada 3 puntos para no saturar
        plt.annotate(f'x={10^{-(i+1)}}', (x, error),
                     xytext=(10, 10), textcoords='offset points',
                     fontsize=9, alpha=0.8)

plt.tight_layout()
plt.show()

# Mostrar tabla con los valores
print("Tabla de resultados:")
print("\n\ttx = 10^{(-k)}\t|f(x)|\t|Error |f(x) - 0.5|")
print(" " * 70)
for i, (x, error) in enumerate(zip(x_values, errors), 1):
    result = f(x)
    print(f"\t{i}\t{result:.8e}\t{error:.8e}")
```

Acá encontramos la inestabilidad de calcular la $f(x)$ alrededor de 0

Justificación de la inestabilidad:

1. Para valores grandes de x (ej: $x = 10^{-1}, 10^{-2}$):

- $\cos(x)$ está lejos de 1
- $1 - \cos(x)$ es un número "grande" (relativamente)
- No hay problemas de precisión significativa
- El error se debe principalmente al truncamiento de la serie de Taylor

2. Para valores intermedios (ej: $x = 10^{-3}, 10^{-4}$):

- $\cos(x)$ se acerca a 1, pero aún es calculable con buena precisión
- $1 - \cos(x)$ es pequeño pero aún representable con precisión adecuada
- Esta es la zona óptima

3. Para valores muy pequeños (ej: $x = 10^{-7}, 10^{-8}, \dots$):

- $\cos(x) \approx 1$ (muy cerca de 1)
- Al calcular $1 - \cos(x)$, estamos **restando dos números casi iguales**
- Esto causa **pérdida de dígitos significativos**
- Los errores de redondeo se amplifican enormemente
- El resultado pierde precisión dramáticamente

Ahora probamos aproximar $f(x)$ con el Polinomio de Taylor centrado en $x=0$ (es más estable)

De grado 8 lo eligió copilot

```
# Implementación de la versión reformulada usando serie de Taylor
def polDeTaylor(x):
    x = np.float64(x)
    x2 = x * x
    x4 = x2 * x2
    x6 = x4 * x2
    x8 = x6 * x2

    # Serie de Taylor hasta  $x^8$ 
    result = np.float64(0.5) - x2/np.float64(24.0) + x4/np.float64(720.0) - x6/np.float64(40320.0) + x8/np.float64(3628800.0)
    return result
```

y calculamos nuevamente el error pero usando $polDeTaylor(x)$ en vez de $F(x)$

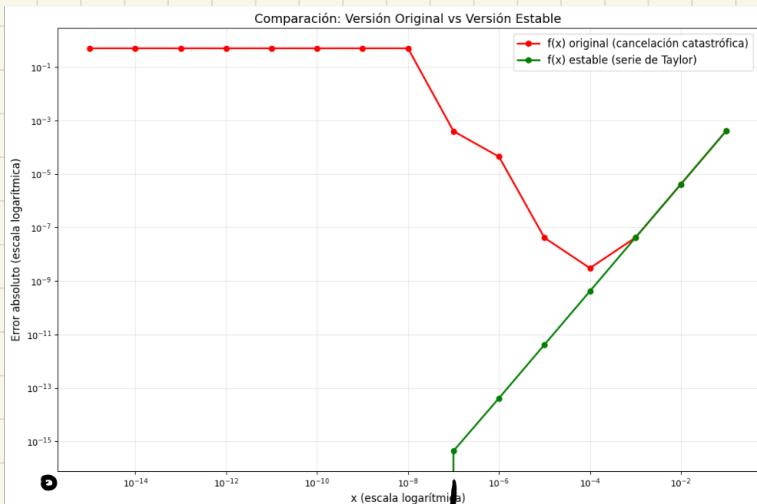
```
for i in range(1, 11):
    x = np.float64(1.0 / (10**i))
    result = polDeTaylor(x)
    print(f"Error cometido con (x = 10^{-{i}}): {abs(result - 0.5)}")
```

Error cometido con ($x = 10^{-1}$): 0.00041652780257661615
Error cometido con ($x = 10^{-2}$): 4.16665277803925e-06
Error cometido con ($x = 10^{-3}$): 4.166666528471197e-08
Error cometido con ($x = 10^{-4}$): 4.1666664563067e-10
Error cometido con ($x = 10^{-5}$): 4.1666670114182125e-12
Error cometido con ($x = 10^{-6}$): 4.168887457467463e-14
Error cometido con ($x = 10^{-7}$): 4.440892098500626e-16
Error cometido con ($x = 10^{-8}$): 0.0
Error cometido con ($x = 10^{-9}$): 0.0
Error cometido con ($x = 10^{-10}$): 0.0

a menor x , menor el error (estable)

Por lo mismo que antes se redondean los números tan chicos

Por ultimo, un grafico que compara los errores:



↳ hasta 10^{-7} devuelven valores,



más chicos redondes → 0

Código del gráfico (necesita $f(x) \rightarrow \text{polDeTaylor}(x)$)

```
# Gráfico comparativo
plt.figure(figsize=(12, 8))

# Calcular errores para ambas versiones
k_values = range(1, 16)
x_values = [10**(-k) for k in k_values]
errors_original = []
errors_estable = []

for x in x_values:
    error_orig = abs(f(x) - 0.5)
    error_est = abs(polDeTaylor(x) - 0.5)
    errors_original.append(error_orig)
    errors_estable.append(error_est)

plt.loglog(x_values, errors_original, 'ro-', linewidth=2, markersize=6, label='f(x) original (cancelación catastrófica)')
plt.loglog(x_values, errors_estable, 'go-', linewidth=2, markersize=6, label='f(x) estable (serie de Taylor)')
plt.xlabel('x (escala logarítmica)', fontsize=12)
plt.ylabel('Error absoluto (escala logarítmica)', fontsize=12)
plt.title('Comparación: Versión Original vs Versión Estable', fontsize=14)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()
```

de más arriba)

1.11 ¿Cuántas multiplicaciones y cuántas sumas se necesitan para realizar la siguiente operación en la forma en la que está dada?

$$\sum_{i=1}^n \sum_{j=1}^i a_i b_j.$$

Modifique la suma de modo tal que reduzca el número de operaciones necesarias. Compare la *performance* de ambos algoritmos en *python* usando un vector aleatorio de *numpy*.

Conteo bruto

$$\sum_{i=1}^n \sum_{j=1}^i a_i b_j$$

↳ $n-1$ sumas (sin productos)

Ahora analizamos cada término

$$\sum_{i=1}^n a_i \times b_j$$

i no me interesa
ni el valor

1 prod por cada término (i prods)

$$\Rightarrow a_1 b_1 + a_1 b_2 + a_1 b_3 + \dots + a_1 b_{i-1} \rightarrow i \text{ términos}$$

⇒ $j-1$ sumas

```
n = 10000

def conteoBruto(n):
    sums, prods = 0, 0
    for i in range(1, n + 1):
        for _ in range(1, i + 1):
            sums += 1
            prods += 1
        sums -= 1
    sums += n - 1
    return sums, prods

brut = conteoBruto(n) #no aguanta n muy grandes

print(f"Se hicieron {brut[0]} sumas y {brut[1]} productos.")
```

Esto se cuenta de ↴

Uno en el for _

e itero hasta n
cambiando i

De los sacamos las fórmulas tamb

$$\text{prod} = \sum_{i=1}^n ; \rightarrow \frac{(n+1)n}{2}$$

sumatoria de afuera $(\sum_{i=1}^n)$

$$\text{los sumas} = (n-1) + \sum_{i=1}^{n-1} i-1$$

terminos de adentro $(\sum_{j=1}^i)$

$$= n - 1 + \frac{(n-1)n}{2}$$

Por último manipulando la sumatoria

$$\sum_{i=1}^n \sum_{j=1}^i a_i b_j \Rightarrow \sum_{i=1}^n a_i \sum_{j=1}^i b_j$$

no depende de j

Entonces podemos reescribir como

$$\sum_{i=1}^n a_i \cdot \left(\underbrace{b_1 + b_2 + b_3 + \dots + b_i}_{i-1 \text{ sumas}} \right)$$

Vemos que hay n prod y $\sum_{i=1}^n i-1$ sumas

$$\frac{(n-1)n}{2}$$

```
n = 10000

def conteoOptimizado(n):
    sums, prods = 0, n
    for i in range(1, n + 1):
        sums += i - 1
    return sums, prods

opt = conteoOptimizado(n)

print()
print("Con el conteo optimizado:")
print(f"Sumas: {opt[0]}")
print(f"Productos: {opt[1]}")
```

1.12 **Método de Horner:** Dado un polinomio $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$. ¿Cuántos productos y cuántas sumas se realizan al evaluar el polinomio en un cierto x_0 ? Horner propone como alternativa escribir a p como $p(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + x a_n)))$. ¿Cuántos productos y cuántas sumas se realizan al evaluar p bajo esta forma?

$$p(x) = a_n \underbrace{x^n}_{+1 \text{ prod}} + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

n-1 sumas

Cada término suma el grado de productos

```

n = 4

def notHorner(n):
    sums, prods = 0, 0
    for i in range(1, n): # n - 1 sumas y la potencia del termino productos
        sums += 1
        prods += i
    return sums, prods

sinHorner = notHorner(n)

print("Sin Horner:")
print(f"Sumas {sinHorner[0]}")
print(f"Productos {sinHorner[1]}")

```

Con Horner

N = 4

$$p(x) = a_0 + x \left(a_1 + x \left(a_2 + x \left(a_3 + x a_4 \right) \right) \right)$$

a_{n-1}

$$a_n = \begin{cases} (+1, +1) & \text{si } n = N \\ a_0 + x a_{n+1} & \text{si } n < N \end{cases} \Rightarrow \begin{matrix} \text{sumas} \\ (N, N) \end{matrix}$$

prods

```
n = 10

def horner(n):
    sums, prods = 0, 0
    for _ in range(1, n + 1): # n sumas y n productos
        sums += 1
        prods *= 1
    return sums, prods

conHorner = horner(n)

print("Con Horner:")
print(f"Sumas {conHorner[0]}")
print(f"Productos {conHorner[1]}")
```