- **Contexto Pregunta 1**: ***Aproximando la función exponencial***: Considere el siguiente límite: $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$, el cual es exactamente $\exp(1) \approx 2.71828182845904523536028747135266249 77\ldots$. Lo interesante de la expresión, que tiene su origen en el cálculo del *interés compuesto*, es que permite obtener el siguiente resultado:

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = \exp(x),$$

es decir, se obtiene la función exponencial $\exp(x)$. En esta pregunta utilizaremos la expresión $\left(1+\frac{x}{n}\right)^n$ como una aproximación de la función $\exp(x)$. **En particular, considere que para el desarrollo de esta pregunta solo tenemos a nuestra disposición las operaciones elementales (suma, resta, multiplicación y división) y la evaluación de una potencia entera**. Por lo tanto, se puede implementar la aproximación directamente.

El análisis de la aproximación requiere que entendamos cómo afecta la elección de n y x en el error absoluto que se obtiene, en particular se considerará como error absoluto la siguiente expresión: $\boxed{\text{Error}(x,n) = \left|\exp(x) - \left(1+\frac{x}{n}\right)^n\right|}$.

- **Contexto Pregunta 2**: ***Triple-root-one***: Sea $f(x)$ una función definida en $x \in [a,b] \subset \mathbb{R}$ para $a < b$ y $a, b \in \mathbb{R}$, que se denominará *triple-root-one*, cuando cumple con las siguientes condiciones:

  (I) contiene **exactamente tres** raíces de **multiplicidad** 1 en el intervalo $[a,b]$, es decir, $f(r_1) = f(r_2) = f(r_3) = 0$, $r_1 \neq r_2 \neq r_3$, y $r_1, r_2, r_3 \in [a,b]$,

  (II) contiene **exactamente dos** puntos críticos distintos en el intervalo $[a,b]$, es decir, existen dos puntos $x_1, x_2 \in [a,b]$ tal que $f'(x_1) = f'(x_2) = 0$ y $x_1 \neq x_2$, y la **multiplicidad** de x_1 y x_2 es 1.

  (III) su derivada $f'(x)$ contiene **exactamente un** punto crítico en el intervalo $[a,b]$, es decir, existe un único punto $x_3 \in [a,b]$ tal que $f''(x_3) = 0$ y la **multiplicidad** de x_3 es 1.

  Notar que de lo anterior se puede concluir que $w_i \neq w_j$ para todo $i \neq j$, donde $w_i, w_j \in \{r_1, r_2, r_3, x_1, x_2, x_3\}$ e $i, j \in \{1,2,3,4,5,6\}$, es decir, las raíces r_1, r_2, r_3 y los puntos críticos x_1, x_2, x_3 son todos distintos entre sí.

  Se incluye a modo referencial una gráfica de una función $f(x)$ del tipo *triple-root-one* en la Figura 1, que cumple con las condiciones mencionadas anteriormente, y una función $\widetilde{f}(x)$ que **no** es del tipo *triple-root-one* en la Figura 2, es decir, no cumple con las condiciones mencionadas anteriormente.

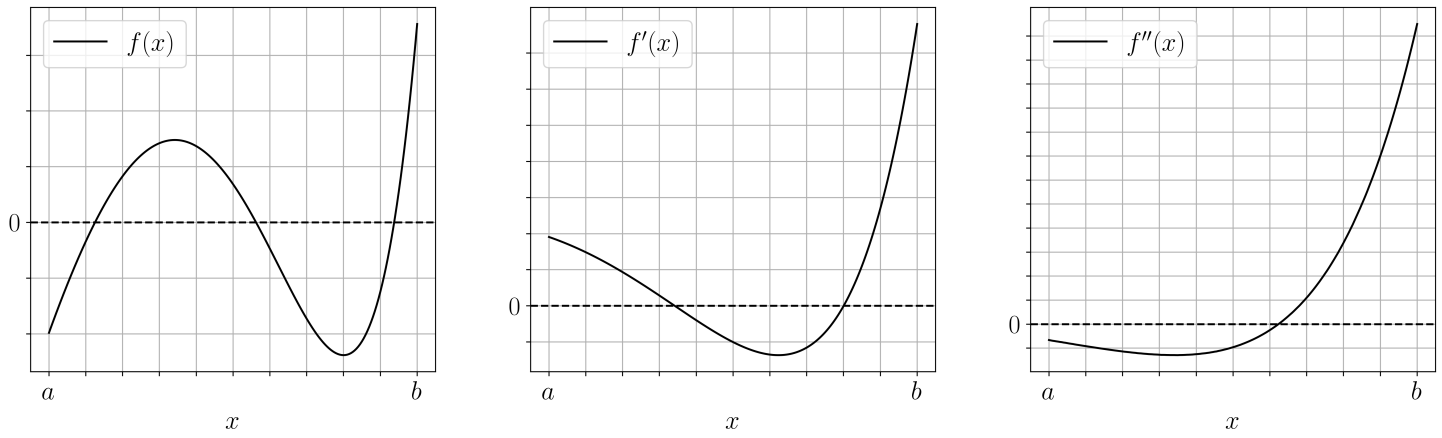


Figura 1: Gráfica de una función $f(x)$ de tipo *triple-root-one* en un intervalo $[a,b]$.

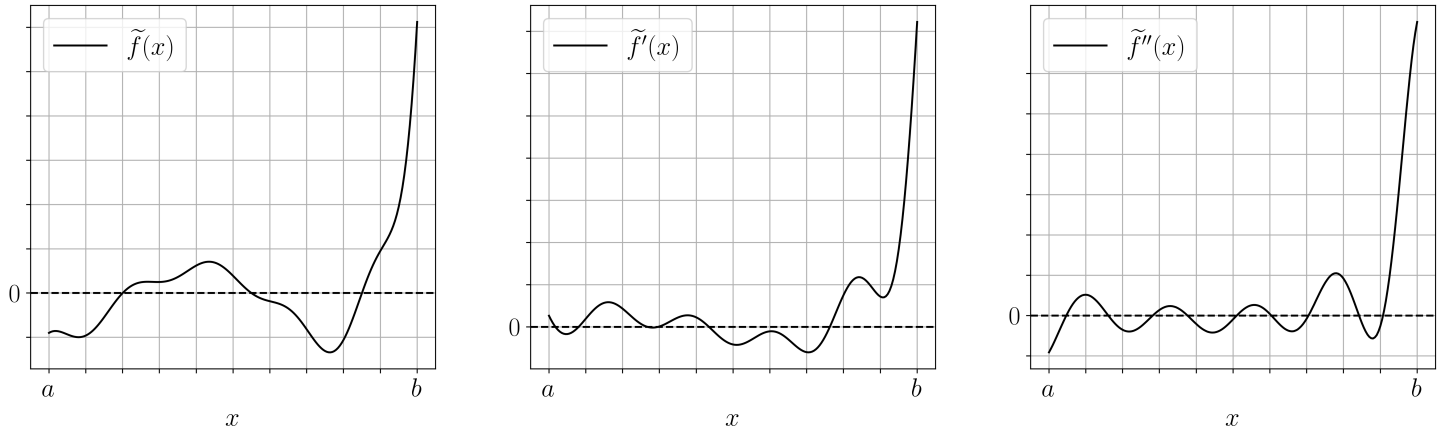
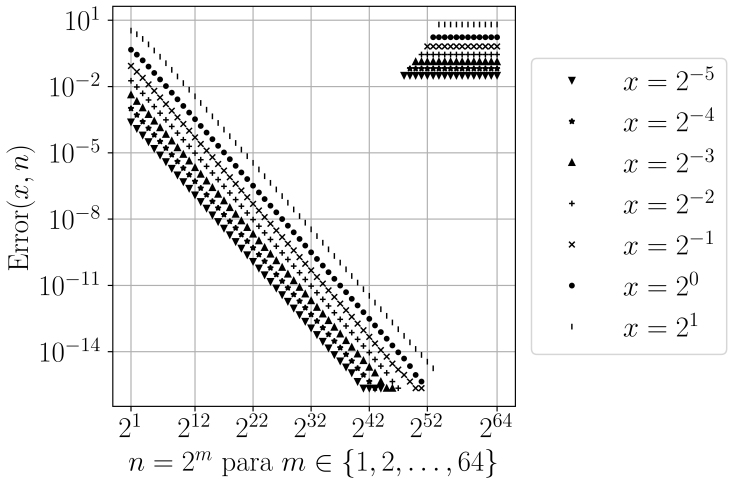


Figura 2: Gráfica de una función $\widetilde{f}(x)$ que **no** es del tipo *triple-root-one* en un intervalo $[a,b]$.

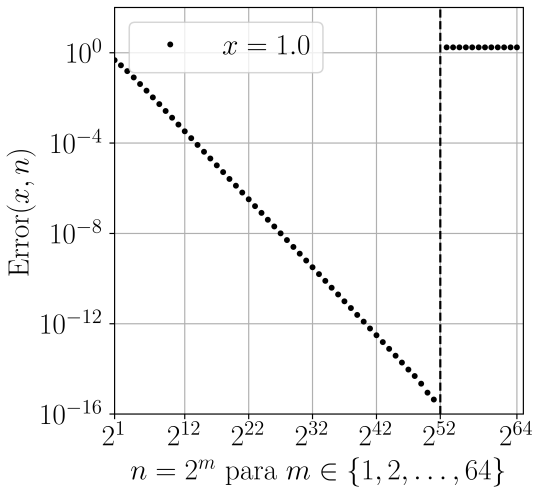
NOMBRE:_____ ROL:_____ PARALELO:_____

- **Desarrollo Pregunta 1**:

(a) **[15 puntos]** Primero, analizaremos el efecto de n en $\text{Error}(x, n)$. En la Figura 3a se muestra el error absoluto para varios valores de $x \in \{2^{-5}, 2^{-4}, 2^{-3}, 2^{-2}, 2^{-1}, 2^{0}, 2^{1}\}$. En todos los casos se observa que el error disminuye, hasta que se genera una discontinuidad. Para entender con mayor profundidad la discontinuidad, se muestra solo el caso para $x = 2^0 = 1.0$ en la Figura 3b. Notar que en este caso el marcador para $n = 2^{52}$ no existe dado que el error es exactamente 0, por esta razón se incluyó una línea punteada vertical.



(a) Gráfica del error para distintos valores de x, indicados en la leyenda, en el rango de valores enteros de n de la forma $n = 2^m$. Es decir, para un valor fijo en la abscisa (eje x) se obtiene el valor del error para cada valor de x indicado en la leyenda por cada marcador.



(b) Gráfica del error para $x = 1.0 = 2^0$ en el rango de valores enteros de n de la forma $n = 2^m$. La línea vertical punteada se agrega dado que para $n = 2^{52}$ no se muestra un punto porque el error es exactamente 0.

Figura 3: Análisis del error de la aproximación $\left(1 + \frac{x}{n}\right)^n$ de $\exp(x)$ sobre la grilla $n = 2^m$ para $m = \{1, 2, \ldots, 64\}$. Notar que se han elegido valores particulares para x.

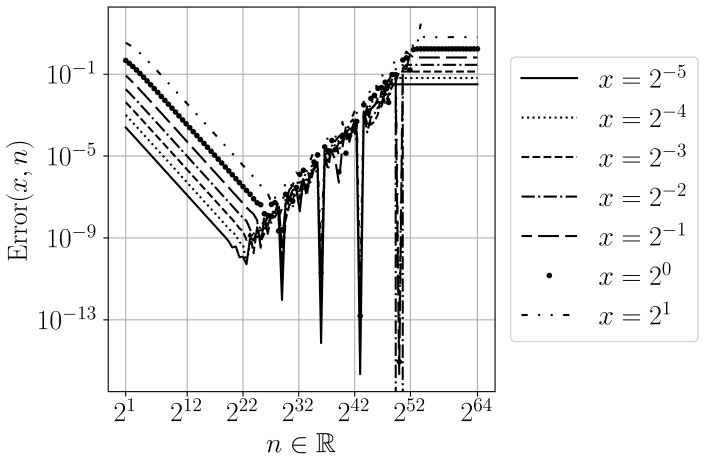
Pregunta: Explique claramente qué es lo que está ocurriendo con la aproximación en $x = 1.0$ para $n \in \{2^{51}, 2^{52}, 2^{53}\}$ considerando que se está trabajando con ***double precision***. Es decir, en la Figura 3b ¿Por qué para $m = 51$ el error es aproximadamente 10^{-16}, para $m = 52$ el error es exactamente 0 y para $m = 53$ el error es aproximadamente 10^0? Justificar técnicamente. Como referencia se incluye la Tabla 1 con el valor explícito del logaritmo en base 10 de $\text{Error}(1, n)$.

m	n	$\log_{10}\left(\left\lvert \exp(1) - \left(1 + \frac{1}{n}\right)^n \right\rvert\right)$
51	2^{51}	-15.35252977886304
52	2^{52}	$-\infty$
53	2^{53}	0.23509439727547035

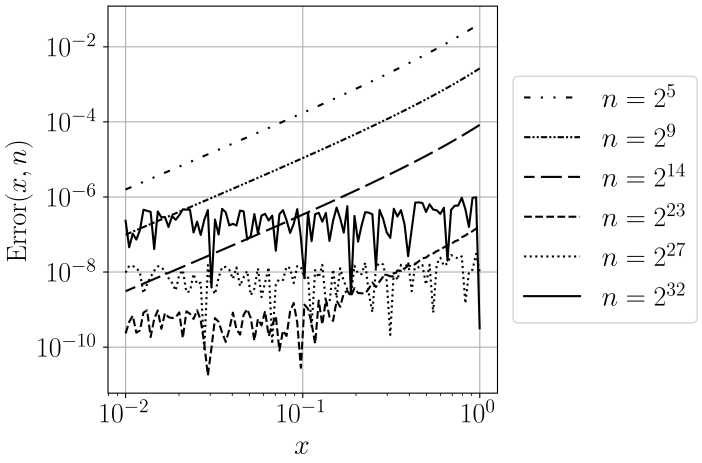
Tabla 1: Error absoluto para $x = 1.0$ con distintos n.

(b) **[15 puntos]** En la Figura 4 se presentan dos gráficas nuevas que nos ayudan con el análisis del error $\text{Error}(x, n)$. La Figura 4a analiza el comportamiento de la variable n de forma continua, es decir, $n \in \mathbb{R}$. Ya no se restringe a números enteros o potencias de 2, como se consideró anteriormente. En este caso la gráfica muestra el comportamiento decreciente del error pero luego creciente a partir de $n \approx 2^{22}$, hasta llegar a la correspondiente asíntota estudiada en la pregunta anterior, lo cual se observa para valores de n mayores a 2^{52} aproximadamente.

Por otro lado, la Figura 4b muestra el análisis del error ahora variando el valor de x continuamente. Para esto, se consideran gráficos independientes para distintos valores de n, indicados en la leyenda, y un rango continuo de valores para x (en la abscisa). En general, se puede considerar que para todos los casos la aproximación mejora a medida que x disminuye, sin embargo el error no necesariamente disminuye al aumentar n, sino que en función del rango de x a analizar, se puede elegir un n que entrega el menor error.



(a) Gráfica del error absoluto para distintos valores de x, indicados en la leyenda, en el rango continuo de valores, es decir $n \in [2^1, 2^{64}]$.



(b) Gráfica del error absoluto para distintos valores de n, indicados en la leyenda, y, en este caso, un rango continuo para x, es decir $x \in [10^{-2}, 10^{0}]$.

Figura 4: Análisis del error de la aproximación $\left(1 + \frac{x}{n}\right)^n$ de $\exp(x)$ para $n \in \mathbb{R}$, Figura (a), y $x \in \mathbb{R}$, Figura (b).

Pregunta: Considerando que usted debe utilizar la aproximación $\left(1 + \frac{x}{n}\right)^n$ de la función $\exp(x)$, y, como se indicó antes, *solo tiene a su disposición las operaciones elementales (suma, resta, multiplicación y división) y la evaluación de una potencia entera*. Proponga un algoritmo basado en el método de Newton para la obtención de una aproximación de la función $\log(y)$ para valores de y en el intervalo $[\exp\left(10^{-2}\right), \exp\left(10^{-1}\right)]$, donde $\exp\left(10^{-2}\right) \approx 1.010050167084$ y $\exp\left(10^{-1}\right) \approx 1.1051709180756$. En su propuesta de algoritmo debe **indicar**:

- explícitamente cuál valor de n utilizará considerando los antecedentes entregados en las Figuras 4a y 4b,
- el *initial guess* a utilizar,
- explícitamente la iteración de Newton respectiva.

(c) **[20 puntos]** Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo propuesto en la Pregunta 1 ítem (b), es decir, en la pregunta anterior. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.power(x,n)`: Evalúa la expresión x^n si `x` y `n` son escalares. En caso de que `x` e `n` sean vectores, deben tener la misma dimensión y entrega la evaluación elemento a elemento. Si solo uno de los términos es un vector, entrega el vector donde el término constante se consideró para cada término de vector.

Notar que al momento de implementar usted **debe decidir** qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input:
y       : (float) Input value for the approximation of log(y).
n       : (int) Integer value for the exponential approximation.
x0      : (float) Initial guess for Newton's method
k       : (int) Number of iterations to be used in Newton's method.

output:
ly      : (float) The approximation of log(y).
'''
def compute_log_y(y,n,x0,k):
    # Your own code.
    return ly
```

NOMBRE:_____ ROL:_____ PARALELO:_____

- **Desarrollo Pregunta 2**:

(a) **[15 puntos]** Considere **solo** para esta pregunta, es decir Pregunta 2 ítem (a), la siguiente información *adicional* para la función *triple-root-one* $f(x)$ definida en $[a, b]$,

- $r_1 < r_2 < r_3$, es decir, las raíces de $f(x)$ están ordenadas de forma creciente,
- $r_1 < c$, donde $c = \dfrac{a+b}{2}$, es decir, $r_1 \in]a, c[$,
- $c < r_3$, es decir, $r_3 \in]c, b[$,
- $f(a)\, f(b) < 0$.

Considerando la implementación del método de la Bisección incluida más abajo, responda las siguientes preguntas:

(I) Si $r_2 < c$, ¿Qué raíz encuentra la implementación si se ejecuta `bisection(f,a,b)`? Justifique. Si no encuentra una raíz, ¿Cuál sería la salida entonces?

(II) Si $r_2 = c$, ¿Qué raíz encuentra la implementación si se ejecuta `bisection(f,a,b)`? Justifique. Si no encuentra una raíz, ¿Cuál sería la salida entonces?

(III) Si $c < r_2$, ¿Qué raíz encuentra la implementación si se ejecuta `bisection(f,a,b)`? Justifique. Si no encuentra una raíz, ¿Cuál sería la salida entonces?

Nota: Si bien el método de la Bisección requiere que solo exista una raíz en el intervalo $[a, b]$ para asegurar que encontrará la raíz, de todos modos se puede ejecutar cuando la función contiene más raíces, en particular 3, ya que existirá el cambio de signo requerido. El objetivo de esta pregunta es determinar qué es lo que entregará el algoritmo sin necesidad de ejecutarlo explícitamente.

```
def bisection(f, a, b, tol=1e-12):
    """
    input:
    f   : (callable) function to evaluate.
    a   : (double)   left value of interval.
    b   : (double)   right value of interval.
    tol : (double)   tolerance.

    output:
    r   : (double)   root approximation of f.
    """
    fa,fb = f(a),f(b)
    if np.sign(fa*fb) > 0:
        return None
    while((b-a)/2 > tol):
        c = (a+b)/2
        fc = f(c)
        if fc == 0:
            break
        elif np.sign(fa*fc) < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    r = (a + b)/2
    return r
```

(b) **[15 puntos]** Proponga un algoritmo que permita obtener las tres raíces r_1, r_2 y r_3 de la función *triple-root-one* $f(x)$ en el intervalo $[a, b]$ basándose en el método de la Bisección. Considere que tiene acceso a evaluar la función $f(x)$, su primera derivada $f'(x)$, su segunda derivada $f''(x)$ y acceso a llamar el método de la Bisección: `bisection(f,a,b)`, las veces que se requiera. *Hint: It may be useful to call the Bisection method more than once and it may not be neccesary to re-implement the method itself.*

(c) **[20 puntos]** Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo propuesto en la Pregunta 2 ítem (b), es decir, en la pregunta anterior. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.sqrt(x)`: Entrega la evaluación de la raíz cuadrada no negativa de un vector o escalar `x`.
- `bisection(f,a,b)`: Implementa el método de la Bisección para la búsqueda de la raíz de la función `f` recibida como entrada en el intervalo [`a`,`b`]. Por simplicidad se omite criterio de detención.

Notar que al momento de implementar usted **debe decidir** qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input:
f   : (callable) triple-root-one function f.
fp  : (callable) derivative of the triple-root-one function f.
fpp : (callable) second derivative of the triple-root-one function f.
a   : (double)   left value of interval.
b   : (double)   right value of interval.

output:
r1   : (float) The approximation of the first root of the triple-root-one function f.
r2   : (float) The approximation of the second root of the triple-root-one function f.
r3   : (float) The approximation of the third root of the triple-root-one function f.
'''
def triple_root_one(f,fp,fpp,a,b):
    # Your own code.
    return r1,r2,r3
```