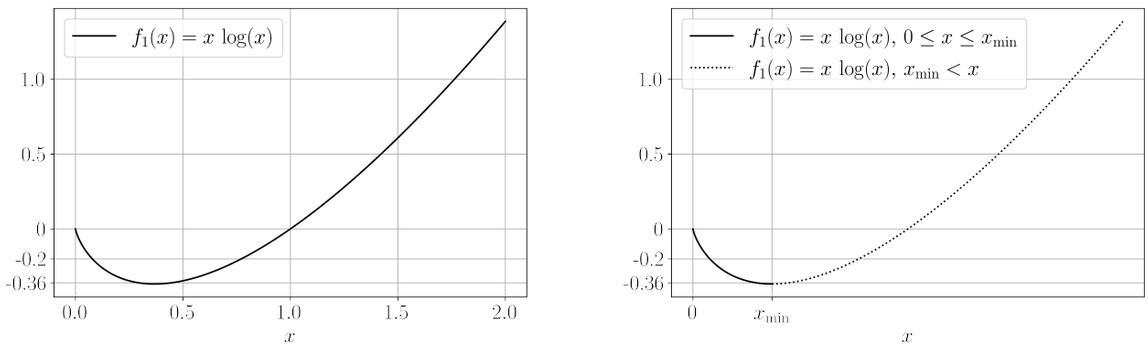
Certamen - 1 - INF-285
SCT - Sa.15.04.23

| Pregunta | Puntos |
|---|---|
| 1 | |
| 2 | |
| Bonus | |
| Total | |

**Instrucciones:** *Usted tiene 70 minutos para responder el Certamen con letra clara y de forma ordenada.* ***Usted tiene que mostrar todo su trabajo para obtener todos los puntos****. Puntos parciales serán entregados a preguntas incompletas. Respuestas finales sin desarrollo o* ***sin nombre*** *reciben 0 puntos. Buena letra, claridad, completitud,* ***conciso*** *y orden en* ***todo el certamen*** *recibe 10 puntos, excepcionalmente se considerarán puntos parciales.* Copy-and-Paste *de algoritmos reciben 0 puntos. ¡Éxito!*

Nombre:_____ Rol:_____ Paralelo:_____

1. *Computación de la inversa.* Un problema clásico en Computación Científica es la evaluación de la función inversa, es decir, si uno tiene la función $f(x)$ entonces la pregunta es la siguiente: ¿Cuál es el valor (o valores) de $\widetilde{x}$ tal que $f(\widetilde{x})$ sea exactamente $\widetilde{y}$?, donde $\widetilde{y}$ es un valor conocido. Anteriormente se indicó que puede que la solución no sea única, lo que conlleva consigo la problemática de elegir entre todas las posibles raíces. Esto ocurre cuando la función $f(x)$ no es inyectiva. Por ejemplo, considere la función $f_1(x) = x\,\log(x)$, ver fig. 1. En este caso si quisiéramos encontrar el valor de $x$ cuando $y = -0.2$ obtendríamos 2 posibles soluciones en el dominio de $[0, 2]$ para $x$. Sin embargo, si restringimos el dominio para $x$ a $[0, x_{\text{mín}}]$, es decir, entre 0 y donde alcanza el mínimo la función $f_1(x)$ se puede concluir que existirá una única solución para la ecuación $-0.2 = x\,\log(x)$, ver curva continua en fig. 1b. En este caso particular se puede obtener que $x_{\text{mín}} = \exp(-1)$.




(a) Gráfica de $f_1(x) = x\,\log(x)$

(b) Dominio de interés en línea continua de la gráfica.

Figura 1: Análisis de $f_1(x) = x\,\log(x)$.

La primera alternativa que se le podría ocurrir a una persona es aplicar el método de Newton a la función $\widehat{f}_1(x) = f_1(x) - y$ para encontrar $x$, la cual genera la siguiente iteración de punto fijo,

$$x_{i+1} = x_i - \frac{\widehat{f}_1(x_i)}{\log(x_i) + 1},$$

que converge cuadráticamente al punto fijo deseado, sin embargo, se requiere una muy buena elección del *initial guess* $x_0$ porque el *vecindario* de convergencia se hace muy pequeño a medida que $y$ tiende a 0. Otra desventaja de este camino es que si en la iteración de punto fijo uno encuentra un valor negativo, la iteración se indefine por la imposibilidad de evaluar $\log(x)$ en un número negativo. Para resolver este problema se propone un cambio de variables conveniente, es decir, considere $u = \log(x)$, lo que implica que $x = \exp(u)$. Adicionalmente, recuerde que solo estamos interesados en valores donde $y < 0$, lo que implica que $-y$ será positivo. Entonces con esta información podemos utilizar el cambio de variables de la siguiente forma:

$$\begin{aligned} y &= x\,\log(x), \\ y &= \exp(u)\,u, \\ -y &= (-u)\,\exp(u), \end{aligned}$$

aplicando logaritmo, dado que solo hay valores positivos involucrados, obtenemos,

$$\begin{aligned} \log(-y) &= \log\left((-u)\,\exp(u)\right) \\ &= \log(-u) + u. \end{aligned}$$

Notar que la simplificación anterior fue posible dado que $\log(\cdot)$ es la función inversa de $\exp(\cdot)$. Entonces, moviendo todo a la derecha, se obtiene la siguiente función a la que se le buscará la raíz,

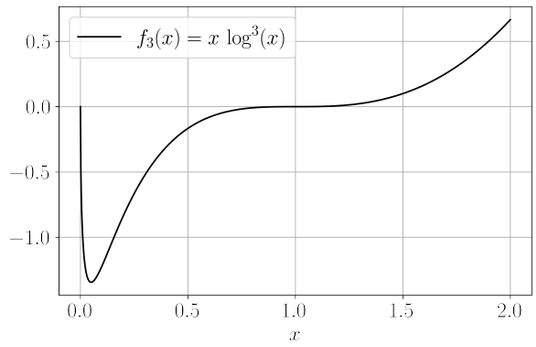
$$w_1(u) = u + \log(-u) - \log(-y).$$

A la cual le podemos aplicar el método de Newton y obtenemos la siguiente iteración de punto fijo con convergencia cuadrática y un *vecindario* para el *initial guess* mucho mayor que el caso anterior,

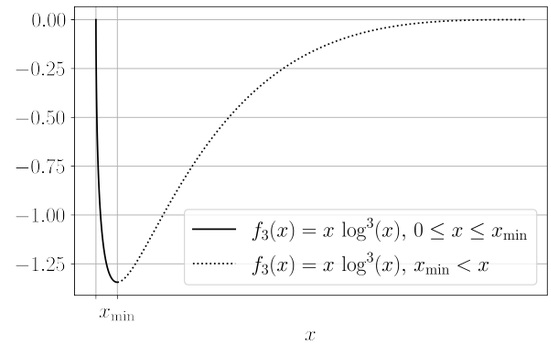
$$u_{i+1} = u_i + \frac{u_i}{u_i + 1}\left(\log\left(\frac{y}{u_i}\right) - u_i\right).$$

Por ejemplo se puede inicializar con $u_0 = -10$ y convergerá en muy pocas iteraciones! Hay que tener presente que luego de obtener el punto fijo $u_\infty$ uno debe aplicar la transformación inversa, es decir, para obtener la raíz $r$ de la función original $\widehat{f}_1(x)$ se necesita evaluar $r = \exp(u_r)$, la cual asegura que será positiva!

Ahora, considere la función $f_\alpha(x) = x\,\log^\alpha(x)$, es decir se eleva a $\alpha$ la función logaritmo. Además considere que $\alpha$ es un número entero positivo e impar. Por ejemplo, si consideramos $\alpha = 3$, se puede apreciar en fig. 2a la gráfica. Similar a $f_1(x)$, nos interesa obtener la inversa de $f_\alpha(x)$ para valores de $x$ en $[0, x_{\text{mín}}]$, donde $x_{\text{mín}}$ ahora es un valor distinto al caso anterior.




(a) Gráfica de $f_3(x) = x\ \log^3(x)$




(b) Dominio de interés en línea continua de la gráfica.

Figura 2: Análisis de $f_3(x) = x\ \log^3(x)$.

(a) **[5 puntos]** Determine $x_{\text{mín}}$ para $f_\alpha(x)$, es decir, obtenga el punto donde $f_\alpha(x)$ consigue el mínimo más cercano al origen.

(b) **[25 puntos]** Construya un algoritmo convergente basado en la metodología anteriormente descrita para obtener la inversa de $f_\alpha(x)$. Considere que los valores de $y$ que se le entregarán estarán en el intervalo $[f_\alpha(x_{\text{mín}}), 0[$. *Hint: Make sure you do compute the root you are looking for.*

(c) **[25 puntos]** Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el procedimiento propuesto anteriormente para la construcción de la aproximación de la inversa de $f_\alpha(x)$ dado $y$, es decir $f_\alpha^{-1}(y) = x$. 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.log10(x)`: $\log_{10}(x)$.
- `np.log(x)`: $\log(x)$.
- `np.exp(x)`: $\exp(x)$
- `np.power(x,n)`: $x^n$.
- `np.sin(x)`: $\sin(x)$.
- `np.cos(x)`: $\cos(x)$.
- `np.arange(n)`: Para `n` un nuúmero entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.sort(x)`: Entrega el arreglo unidimensional `x` ordenado de menor a mayor.
- `np.argsort(x)`: Entrega en el vector de salida `y` los índices de las entradas de `x` tal que al evaluar vectorialmente `x[y]` se obtienen las entradas ordenadas de menor a mayor.
- `np.sort_complex(z)`: Entrega el arreglo de números complejos `z` ordenados de menor a mayor donde primero se ordena la parte real y luego la parte imaginaria.

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

```
'''
input:
y   : (double) "y" value.
a   : (integer) "\alpha".
n   : (integer) Max number of iteration to be used.

output:
r   : (double) Root obtained by Newton's method.
'''
def find_inverse_f_alpha(y,a,n):
    u0 = -10 # For simplicity use this "initial guess" as default.
    # Your own code.
    return r
```

NOMBRE:_____ ROL:_____ PARALELO:_____

2. *Visitando la expansión en serie de Taylor de la función exponencial.* La función exponencial es una de las funciones más famosas que existen, parte de su fama se debe a que su derivada es la misma función! Por otro lado, si describimos la función exponencial por su expansión en serie de Taylor obtenemos,

$$\begin{aligned}\exp(x) &= \sum_{i=0}^{\infty} \frac{x^i}{i!} \\ &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots\end{aligned}$$

Por ejemplo, si derivamos la expresión anterior término a término obtenemos lo siguiente,

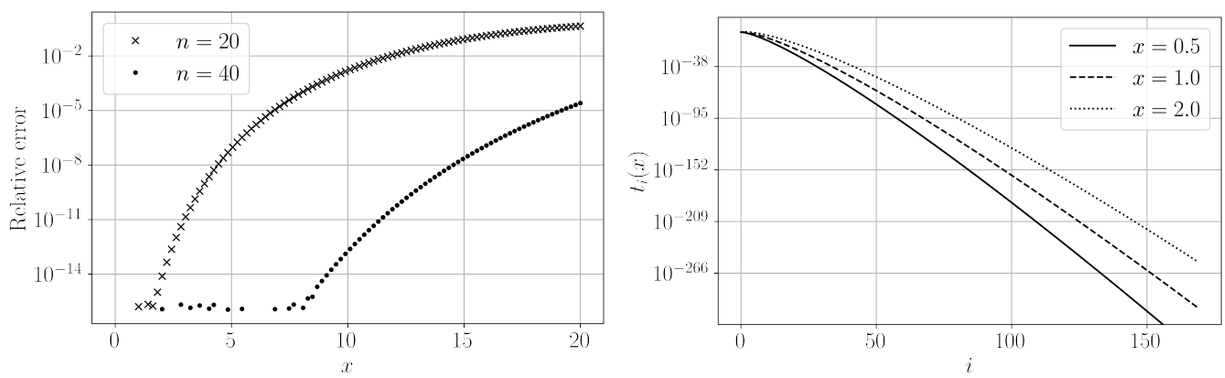
$$\begin{aligned}(\exp(x))' &= 0 + 1 + \frac{2\,x^1}{2} + \frac{3\,x^2}{3!} + \frac{4\,x^3}{4!} + \dots \\ \exp(x) &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &= \sum_{i=0}^{\infty} \frac{x^i}{i!}\end{aligned}$$

Lo que es justamente lo que se había mencionado anteriormente, se vuelve a obtener la misma función, tanto el lado izquierdo como el lado derecho de la expresión, luego de aplicar la derivada! Desde el punto de vista computacional, es inviable construir un ciclo infinito, por lo cual una aproximación natural es truncar la serie en una sumatoria de la siguiente forma,

$$\exp(x) = \underbrace{\sum_{i=0}^{n} \frac{x^i}{i!}}_{\exp_n(x)} + \underbrace{\sum_{i=n+1}^{\infty} \frac{x^i}{i!}}_{\exp_n^{[\text{error}]}(x)},$$

es decir, $\exp_n(x)$ representa la aproximación de la función exponencial con $n+1$ términos de su serie de Taylor y $\exp_n^{[\text{error}]}(x)$ es el error asociado a la aproximación. La principal razón para encontrar aproximaciones de la función exponencial es para que su computación solo dependa de la evaluación de operaciones elementales, es decir: suma, resta, multiplicación y división.

Adicionalmente podemos definir cada término de la serie de Taylor de la función exponencial de la siguiente forma: $t_i(x) = \frac{x^i}{i!}$. En la fig. 3a se muestra el error relativo entre la función exponencial $\exp(x)$ y su aproximación $\exp_n(x)$ para $n \in \{20, 40\}$ y para valores de $x \in [0, 20]$. Al principio del intervalo pareciera ser que faltan algunos puntos, pero esto se debe a que la diferencia es numéricamente 0 o muy cercano a 0, tal que no alcanza a aparecer en la gráfica. En este caso esto es algo positivo. Por otro lado, se observa que a medida que el valor de $x$ aumenta, el error también aumenta. En principio, como se observa en la fig. 3a, uno puede disminuir el error al aumentar $n$, pero eso trae consigo desafíos adicionales. Por ejemplo, si consideramos $x \geq 0$ entonces cada $t_i(x) > 0$, lo cual nos ayuda para poder estudiar la magnitud asociada a cada término $t_i(x)$, ver fig. 3b. Notar que el gráfico está en escala logarítmica y que se presentan 3 curvas distintas. Cada curva corresponde a los distintos valores de $x$ incluidos en la leyenda de la gráfica. Claramente se observa que hay una diferencia importante en la magnitud de cada término involucrado en la construcción de $\exp_n(x)$. Entonces, si uno utiliza *double precision*, se debe tomar precauciones adicionales al momento de implementar numéricamente la evaluación de $\exp_n(x)$.




(a) Error relativo entre $\exp(x)$ y $\exp_n(x)$ para $n \in \{20, 40\}$. Notar que la abscisa representa la variable continua $x$.

(b) Gráfica de cada término $t_i(x)$ para 3 valores distintos de $x$. Notar que la abscisa representa la variable discreta $i$.

Figura 3: Análisis de la función exponencial y la expansión en series de Taylor de esta.

Ahora, considere la siguiente función,

$$\tau_n(x) = \sum_{i=0}^{n-1} \gamma_i\, x^i.$$

Por simplicidad se considerará $x \geq 0$ y $0 \leq \gamma_i \leq 1$ para todo $i$, es decir, se sumarán términos positivos.

(a) **[30 puntos]** Construya un algoritmo que permita evaluar *adecuadamente* la función $\tau_n(x)$ dado los valores de $\gamma_i$ y $x$. Asegúrese de describir matemáticamente todos los pasos y argumentos considerando que será implementando en aritmética de punto flotante de *double precision. Hint: Make sure you explain every step completely and provide the full analysis of what is going on.*

(b) **[25 puntos]** Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el procedimiento propuesto anteriormente. 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.log10(x)`: $\log_{10}(x)$.
- `np.log(x)`: $\log(x)$.
- `np.exp(x)`: $\exp(x)$
- `np.power(x,n)`: $x^n$.
- `np.sin(x)`: $\sin(x)$.
- `np.cos(x)`: $\cos(x)$.
- `np.arange(n)`: Para `n` un nuúmero entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.sort(x)`: Entrega el arreglo unidimensional `x` ordenado de menor a mayor.
- `np.argsort(x)`: Entrega en el vector de salida `y` los índices de las entradas de `x` tal que al evaluar vectorialmente `x[y]` se obtienen las entradas ordenadas de menor a mayor.
- `np.sort_complex(z)`: Entrega el arreglo de números complejos `z` ordenados de menor a mayor donde primero se ordena la parte real y luego la parte imaginaria.

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

```
'''
input:
x       : (double) "x" value.
gammas  : (ndarray) gamma_i values in a vector of dimension "n".
n       : (integer) Associated to upper limit of sum.

output:
tau      : (double) Value of tau_n(x).
'''
def compute_tau_n(x,gammas,n):
    # Your own code.
    return tau
```