

1. Lectura preliminar

El objetivo de esta lista de ejercicios es la familiarización con preguntas desarrolladas anteriormente en certámenes. Se sugiere encarecidamente que se aplique la metodología IDEA a cada ejercicio. Algunos ejercicios solicitan explícitamente la implementación de los algoritmos propuestos y otro no, sin embargo, todos los ejercicios contienen una componente de desarrollo algebraico primero y luego el desarrollo se puede implementar en Python. En particular se sugiere adquirir práctica utilizando NumPy adecuadamente en las implementaciones de sus respuestas. Esto se discute extensamente en el reglamento de tareas. Por ejemplo, considere las siguientes propuestas de implementación para obtener A^2 , es decir, elevar una matriz al cuadrado:

- `np.power(A,2)`
- `np.dot(A,A)`
- `A @ A`
- `np.linalg.matrix_power(A,2)`

¿Son todas estas alternativas equivalentes? Si no lo fueran, explique.








En apartado 3 se presenta el desarrollo de algunos ejercicios a modo referencial para que pueda comparar su desarrollo con el desarrollo propuesto.

2. Ejercicios propuestos

2.1. Raíces

Se tiene la siguiente función continua y diferenciable en \mathbb{R} :

$$f(x) = x^3 + 4x^2 - 10$$

- (a)  Suponga que existe una raíz de $f(x)$ en $[1, 2]$ ¿Cuántas iteraciones requiere el método de la bisección para aproximar la raíz con 10 decimales correctos?
- (b)  Aplique computacionalmente el método de la bisección en $[1, 2]$ para verificar su respuesta anterior.
- (c)  Proponga una iteración de punto fijo que permita encontrar la raíz de $f(x)$ en $[1, 2]$. (Restricción: No puede usar Newton para derivar el punto fijo!).
- (d)  ¿Cuál es la tasa de convergencia de la iteración de punto fijo propuesta en la pregunta anterior?
- (e)  Proponga una iteración de punto fijo con tasa de convergencia $S = 0.8$ aproximadamente que permita encontrar la raíz de $f(x)$ en $[1, 2]$.
- (f)  Proponga una iteración de punto fijo con tasa de convergencia $S = 0.1$ aproximadamente que permita encontrar la raíz de $f(x)$ en $[1, 2]$.
- (g)  Implemente la iteración de punto fijo obtenida en la pregunta anterior para verificar que converge a la raíz indicada para ambos casos y que la tasa de convergencia solicitada efectivamente se obtiene.

2.2. Ecuación de 4to grado

Considere la siguiente ecuación de cuarto grado:

$$Ax^4 + Bx^2 + C = 0, \tag{1}$$

donde $A, B, C \in \mathbb{R}$, $|B| \gg |A|$ y $|B| \gg |C|$. Una forma de resolver este tipo de ecuaciones es transformándola en una ecuación de segundo grado, mediante la sustitución $y = x^2$. Así, el problema (1) se transforma en

$$Ay^2 + By + C = 0. \tag{2}$$

Una vez encontradas las soluciones de (2) mediante las soluciones de la ecuación cuadrática, podemos resolver el problema (1).

- (a)  Entregue una expresión para calcular las raíces de (2) tal que **no haya pérdida de importancia**.

- (b) ∇ Proponga un algoritmo que reciba como parámetros los valores de A , B y C y retorne las cuatro raíces de (1) tal que **no haya pérdida de importancia**. *Hint 1: There is no need for a root-finding algorithm. Hint 2: You need to use your previous answer here.*
- (c) \Rightarrow Implemente su algoritmo que ha definido en la pregunta anterior, donde usted indicaba que no había pérdida de importancia. Llamémoslo Algoritmo 1. Solo reporte las raíces reales en la implementación.
- (d) \Rightarrow Implemente el algoritmo donde usted sospechaba que sí había pérdida de importancia. Llamémoslo Algoritmo 2. Solo reporte las raíces reales en la implementación.
- (e) \Rightarrow Obtenga las raíces con ambos algoritmos para los siguientes parámetros: $A = 124$, $B = 2e18$ y $C = -1.3$.
- (f) ∇ ¿Efectivamente había pérdida de importancia en las raíces obtenidas? Discuta.

2.3. Polinomios y matrices

Sea $p(x)$ el siguiente polinomio cuadrático en x :

$$p(x) = -\alpha - \alpha^2 + (1 - \alpha^2 - \alpha^3)x + \alpha x^2, \quad (3)$$

donde $\alpha \in \mathbb{R}$. Ahora, sea $A \in \mathbb{R}^{n \times n}$, considere que en vez de pasarle x a $p(x)$ se le pasa la matriz A , entonces obtenemos:

$$p(A) = (-\alpha - \alpha^2)I + (1 - \alpha^2 - \alpha^3)A + \alpha A^2, \quad (4)$$

donde I es la matriz identidad de $n \times n$. Particularmente, si contamos con la descomposición de valores propios de A obtenemos $A = V \Lambda V^{-1} = V \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n) V^{-1}$, donde V es la matriz con los vectores propios de A como vectores columna y Λ es la matriz diagonal con los valores propios de A en su diagonal. Utilizando la identidad podemos obtener:

$$\begin{aligned} p(A) &= p(V \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n) V^{-1}), \\ &= (-\alpha - \alpha^2)I + (1 - \alpha^2 - \alpha^3)V \Lambda V^{-1} + \alpha (V \Lambda V^{-1})^2, \\ &= V \text{diag}(p(\lambda_1), p(\lambda_2), \dots, p(\lambda_n)) V^{-1}, \end{aligned}$$

es decir, aplicar el polinomio $p(x)$ a la matriz A es equivalente a aplicar el polinomio a cada uno de sus valores propios.

- (a) ∇ Encuentre las raíces de $p(x)$ en función de α .
- (b) ∇ Determine las condiciones para que la siguiente ecuación tenga solución $|B(\hat{\alpha})| = 0$, donde $B(\hat{\alpha}) = p(A)$ y $|\cdot|$ corresponde al determinante.
- (c) \Rightarrow Implemente un algoritmo que obtenga $\hat{\alpha}$ tal que $|B(\hat{\alpha})| = 0$. El **input** del algoritmo debe ser una matriz $A \in \mathbb{R}^{n \times n}$ de dimensión arbitraria y el **output** debe ser la correspondiente raíz. La única restricción es que usted no puede utilizar una rutina para obtener los valores y vectores propios.
- (d) \Rightarrow Modifique su algoritmo anterior tal que obtenga el mayor y el menor $\hat{\alpha}$ posibles. Hint: It is OK if you don't get the largest or the lowest values for $\hat{\alpha}$, what it is required is your reasoning toward that end. This will make your algorithm unique!

2.4. Evaluaciones problemáticas y adecuadas

En la empresa de construcción de funciones llamada “EfeDeEquis” construyeron la siguiente función:

$$f_1(x) = \log \left(x - \sqrt{x^2 - 1} \right).$$


La cual está definida para $x \in [1, \infty[$ y corresponde a una primera versión. Lamentablemente esta función no se comporta adecuadamente en *double precision* para valores de x muy grandes. Por lo cual se le solicita a los estudiantes de Computación Científica que propongan una alternativa conveniente y equivalente a $f_1(x)$. Notar que la función logaritmo corresponde al logaritmo natural. Considere los siguientes 5 valores para evaluar las funciones:

$$\begin{aligned} x_1 &= 1.00000000000000000000000000000000, \\ x_2 &= 81377.3957125740665806667073286, \\ x_3 &= 1.32445610649217361470884567491 \cdot 10^{10}, \\ x_4 &= 2.15561577355759761355671114643 \cdot 10^{15}, \\ x_5 &= 3.50836795604881586932735799943 \cdot 10^{20}. \end{aligned}$$


- (a) \Rightarrow Evalúe cada función en los valores x_i indicados, para $i \in \{1, 2, 3, 4, 5\}$. En caso de que la respuesta sea $-\infty$, ingresar el valor 1. A continuación se le entrega la implementación en NumPy para evaluar $f_1(x)$ numéricamente:

```
f1 = lambda x: np.log(x - np.sqrt(np.power(x,2) - 1))
```




Hint: Recall that multiplying by 1 or adding 0 is always allowed!

- (b)  Indique cuál es el primer valor, partiendo de x_1 hasta x_5 , que presenta algún problema numérico catastrófico al evaluarlo en $f_1(x)$, es decir da $-\infty$.

Nota: Nos referimos solo a evaluar la función en los valores indicados en la lista anterior.

- (c)  Proponga una nueva función que sí se pueda evaluar para valores de x muy grandes y que sea capaz de obtener un valor correcto para todos los x_i indicados. Esta función se denotará $f_2(x)$ y deberá estar implementada en su jupyter notebook. La firma requerida de esa función sería:

```
f2 = lambda x: ''su propuesta''
```

- (d)  Considerando que el comportamiento inadecuado de la función $f_1(x)$ se obtiene para valores de x muy grandes, se propone definir $x = 10^n$, con $n \in \mathbb{N}$ y $n \leq 400$. Determine el menor n posible tal que el resultado obtenido sea erróneo, es decir $-\infty$. Si no lo obtiene, indique el valor -1 en su respuesta.
- (e)  Considerando que el comportamiento inadecuado de la función $f_2(x)$ se obtiene para valores de x muy grandes, se propone definir $x = 10^n$, con $n \in \mathbb{N}$ y $n \leq 400$. Determine el menor n posible tal que el resultado obtenido sea erróneo. Si no lo obtiene, indique el valor -1 en su respuesta.
- (f)  Determine $f_2(2.61073 \cdot 10^{173})$.

2.5. Varias variables

Considere la siguiente función dependiente de la variable temporal t y la variable espacial x ,

$$g(t, x) = a(t) h_1(x) + b(t) h_2(x) + c(t). \quad (5)$$





De la función anterior, nos interesa obtener los valores de t y x en donde se anula la función. En particular se debe considerar que las funciones $a(t)$, $b(t)$, $c(t)$, $h_1(x)$, y $h_2(x)$ son parámetros que serán entregados. Adicionalmente se dispone en este caso de las derivadas de cada una de las funciones de una variable involucradas, es decir $a'(t)$, $b'(t)$, $c'(t)$, $h'_1(x)$, y $h'_2(x)$. Para el estudio de los ceros de la función $f(t, x)$, se analizará para distintos casos que se listan a continuación:

Caso 1 $a(t) = 0$, $b(t) = 0$, y $c(t) = 0$. En este caso $f(t, x)$ es la función nula, por lo que es 0 en todo el dominio.

Caso 2 $a(t) = 0$, $b(t) = 0$, y $c(t) \neq 0$. En este caso $f(t, x)$ es distinto de 0 en todo el dominio, por lo tanto nunca se anula $f(t, x)$.

Caso 3 $a(t) \neq 0$ o $b(t) \neq 0$. En este caso existirá por lo menos una raíz. En esta pregunta solo será de interés encontrar por lo menos 1 raíz.

Los casos 1 y 2 no entregan mayor dificultad de análisis e implementación, por lo cual serán omitidos. Sin embargo el caso 3 sí es mucho más interesante de estudiar en detalle.

- (a)  Proponga un algoritmo **convergente** que para un tiempo t determinado entregue la raíz más *cercana* al *initial guess* \tilde{x} que será entregado como parámetro.
- (b)  Implemente el algoritmo propuesto. Su implementación debe recibir t , $a(t)$, $b(t)$, $c(t)$, $h_1(x)$, $h_2(x)$, $a'(t)$, $b'(t)$, $c'(t)$, $h'_1(x)$, $h'_2(x)$, y \tilde{x} como *inputs* y retornar la raíz utilizando *double precision*.
- (c)  Entregue la raíz obtenidas para $t = 1$, $a(t) = -1$, $b(t) = t$, $c(t) = 1 + \exp(-t)$, $h_1(x) = x^2$, $h_2(x) = x$, y $\tilde{x} = 0$. En este caso las derivadas de cada función se omiten porque se pueden obtener directamente, sin embargo sí deben ser entregadas al momento de llamar a la función implementada.
- (d)  Implemente un algoritmo que reciba un rango de tiempo $[\tau_0, \tau_1]$ y un entero n tal que el rango de tiempo se discretice en $n + 1$ puntos equiespaciados, y se retorne la raíz obtenida x_i de $f(t_i, x)$ para cada tiempo $t_i \in [\tau_0, \tau_1]$, donde $t_i = \frac{\tau_1 - \tau_0}{n} i$ e $i \in \{0, 1, 2, \dots, n\}$. Esto significa que al evaluar $f(t_i, x_i)$ se debe obtener 0. El algoritmo debe recibir los siguientes parámetros: τ_0 , τ_1 , n , $a(t)$, $b(t)$, $c(t)$, $h_1(x)$, $h_2(x)$, $a'(t)$, $b'(t)$, $c'(t)$, $h'_1(x)$, $h'_2(x)$, y \tilde{x} , donde \tilde{x} corresponde al *initial guess* para el tiempo $t_0 = \tau_0$ solamente. Usted debe definir convenientemente el *initial guess* para los siguientes valores de t_i para $i > 0$.

2.6. ¿Qué se busca?

Considere la siguiente ecuación:

$$f(x, y) = x + y \sin(\log(y^2 + 1) - y) = 1,$$

donde \log corresponde al logaritmo natural. Notar que este es un problema unidimensional.

- (a) Construya un algoritmo basado en el **método de Newton** que obtenga el valor de y cercano al initial guess y_0 dado un valor conocido de $x = \hat{x}$, es decir, debe encontrar el valor de y tal que se cumpla la siguiente ecuación $f(\hat{x}, y) = 1$. Usted debe explicitar todas las componentes requeridas para ejecutar el método de Newton, le sugiero que no trate de simplificar las expresiones involucradas, pero si gusta hacerlo, tiene la libertad de hacerlo. Justifique su resultado.
- (b) Ejecute una iteración de su algoritmo para encontrar y tal que $\hat{x} = 2$ y $y_0 = -6$. **¿Obtuvo la raíz exacta?** Justifique su resultado.

2.7. Implicancia de FPS en factorización LU

Considere $A \in \mathbb{R}^{n \times n}$ y que uno tiene acceso a la factorización LU de esta misma, la cual es la siguiente:

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ 1 & 1 & 0 & \ddots & \dots & 0 \\ 0 & 1 & 1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} \varepsilon & 1 & 0 & \dots & \dots & 0 \\ 0 & \varepsilon & 1 & \ddots & \dots & 0 \\ 0 & 0 & \varepsilon & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & \varepsilon & 1 \\ 0 & 0 & 0 & 0 & 0 & \varepsilon \end{pmatrix},$$

donde $\varepsilon \in \mathbb{R}$ y $\varepsilon > 0$. Lo interesante de esta representación es que nos permite obtener el determinante de la matriz A rápidamente dado que $\det(A) = \det(L) \det(U) = \varepsilon^n$.

Ahora, considere que las matrices L y U se almacenarán en un computador que utiliza una variante del formato de double precision del estándar de punto flotante de la IEEE 754. Esta variante aún considera 64 bits en total y se denomina $\text{FPS}(m)$, donde m denota la cantidad de bits disponibles para el exponente. En específico, $\text{FPS}(m)$ se describe de la siguiente forma:

- Número de bits utilizados para el signo: 1
- Número de bits utilizado para el exponente: m
- Número de bits utilizado para la mantisa: $63 - m$

En particular se considera que $m \in \{2, \dots, 62\}$, así se asegura que se pueden considerar los mismos casos especiales que en la versión tradicional del formato double precision, que en esta representación corresponde a $\text{FPS}(11)$. En esta variante, el bias se obtiene como $2^{m-1} - 1$.

- (a) Determine Machine Epsilon en el formato $\text{FPS}(m)$. Justifique su resultado.
- (b) Determine el menor número positivo representable en el formato $\text{FPS}(m)$. Justifique su resultado.
- (c) Obtenga el determinante de A , el resultado debe depender de ε y n . Justifique su resultado.
- (d) Considere que trabaja con $\text{FPS}(17)$ y que $\varepsilon = 2^{-2}$, determine el menor valor posible de n tal que se obtenga $\det(A) = 0$ numéricamente. Justifique su resultado.
- (e) Determine el número de operaciones elementales que se requieren para resolver $A\mathbf{x} = \mathbf{b}$ convenientemente en este caso, es decir, no contando multiplicaciones donde se sabe con anticipación que uno de los coeficientes es 0. Justifique su resultado. Además recuerde que $\varepsilon > 0$. Hint: You should use the LU factorization already provided.

2.8. Serie truncada

Considere la siguiente función representada como una serie:

$$S(x) = \sum_{k=1}^{\infty} \frac{\cos(kx)}{k}, \quad \forall x \in]0, 2\pi[.$$

Lo interesante de esta serie es que produce coeficientes alternantes, por ejemplo, si uno evalúa los primeros 10 términos de la serie para $x = 1$ obtiene,

k	1	2	3	4	5	6	7	8	9	10
$\cos(k)/k$	0.5403	-0.2081	-0.3300	-0.1634	0.0567	0.1600	0.1077	-0.0182	-0.1012	-0.0839

Claramente notamos que será un desafío sumar la serie debido a: la alternancia de signo de los coeficientes, al lento decaimiento de los mismos y a la pérdida de importancia en *double precision*. Pero antes de construir un algoritmo adecuado para sumar la serie, procederemos a estudiar el comportamiento de la suma parcial de la serie, es decir de la función $S_n(x)$ que se define a continuación,

$$S(x) = S_n(x) + \sum_{k=n+1}^{\infty} \frac{\cos(kx)}{k},$$

la cual se puede expresar de la siguiente forma,

$$S_n(x) = \sum_{k=1}^n \frac{\cos(kx)}{k}.$$

En particular nos interesa en encontrar un punto crítico de $S_n(x)$ cercano a $x = 1$. Claramente este resultado dependerá del valor de n que se utilice.

- Proponga un algoritmo utilizando el Método de Newton para encontrar un punto crítico, considere que el valor del parámetro n es conocido. Recordar que usted debe **explicitar completamente** todas las componentes necesarias para la utilización del Método de Newton. *Hint: Recall that the critical point of a function is when its derivative is equal to 0.*
- Implemente en Python utilizando adecuadamente la capacidad de vectorización¹ de NumPy el algoritmo propuesto anteriormente y entregue una estimación de la tasa de convergencia lineal $\left(\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i}\right)$ y cuadrática $\left(\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i^2}\right)$. Considere que tiene a su disposición la función coseno de NumPy, es decir, `np.cos(x)`, `np.sum` y `np.arange`. Recuerde que `np.arange(1,5)` genera el ndarray `[1,2,3,4]`. La firma de la función es la siguiente²:

```
'''
input:
n   : (int64) Upper limit of the sum S_n(x).
j   : (int64) Number of iterations to be used in Newton's method.

output:
r   : (double) Root obtained by Newton's method.
S   : (double) Estimated linear rate of convergence.
M   : (double) Estimated quadratic rate of convergence.
'''
def find_critical_point_of_Sn(n, j):
    # Your own code.
    return r, S, M
```

2.9. 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 \tilde{x} tal que $f(\tilde{x})$ sea exactamente \tilde{y} ?, donde \tilde{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 figura 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 $[0, x_{\min}]$, 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 figura 1b. En este caso particular se puede obtener que $x_{\min} = \exp(-1)$.

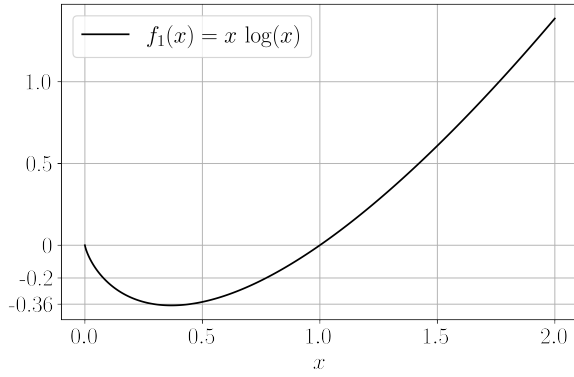
La primera alternativa que se le podría ocurrir a una persona es aplicar el método de Newton a la función $\hat{f}_1(x) = f_1(x) - y$ para encontrar x , la cual genera la siguiente iteración de punto fijo,

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

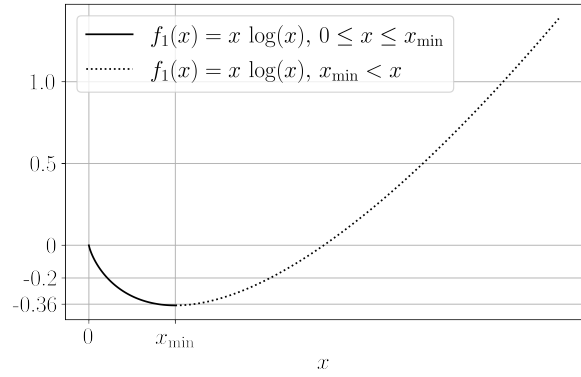
que converge cuadráticamente al punto fijo deseado, sin embargo, se requiere una muy buena elección del *initial guess* x_0 porque el *vecindario* de convergencia se hace muy pequeño a medida que y tiende a 0. Otra desventaja de este camino es que si en la

¹Es decir, reducir al máximo el uso de “loops”.

²Para el desarrollo no es necesario que repita completamente la descripción de inputs y outputs.



(a) Gráfica de $f_1(x) = x \log(x)$



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

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

iteración de punto fijo uno encuentra un valor negativo, la iteración se indefiniría 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,

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

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,

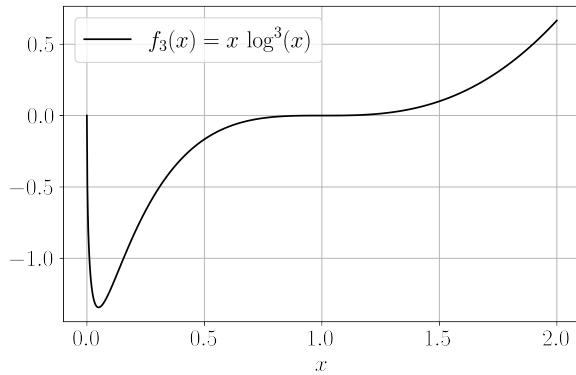
$$u_{i+1} = u_i + \frac{u_i}{u_i + 1} \left(\log\left(\frac{y}{u_i}\right) - u_i \right).$$

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_∞ uno debe aplicar la transformación inversa, es decir, para obtener la raíz r de la función original $\hat{f}_1(x)$ se necesita evaluar $r = \exp(u_r)$, la cual asegura que será positiva!

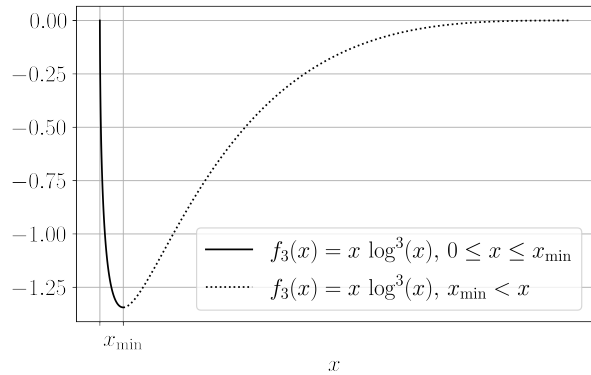
Ahora, considere la función $f_\alpha(x) = x \log^\alpha(x)$, es decir se eleva a α la función logaritmo. Además considere que α es un número entero positivo e impar. Por ejemplo, si consideramos $\alpha = 3$, se puede apreciar en figura 2a la gráfica. Similar a $f_1(x)$, nos interesa obtener la inversa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

- Determine x_{\min} para $f_\alpha(x)$, es decir, obtenga el punto donde $f_\alpha(x)$ consigue el mínimo más cercano al origen.
- 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 $[f_\alpha(x_{\min}), 0]$. *Hint: Make sure you do compute the root you are looking for.*
- 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)`: $\log(x)$.



(a) Gráfica de $f_3(x) = x \log^3(x)$



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

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

- `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 número 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
```

2.10. 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,

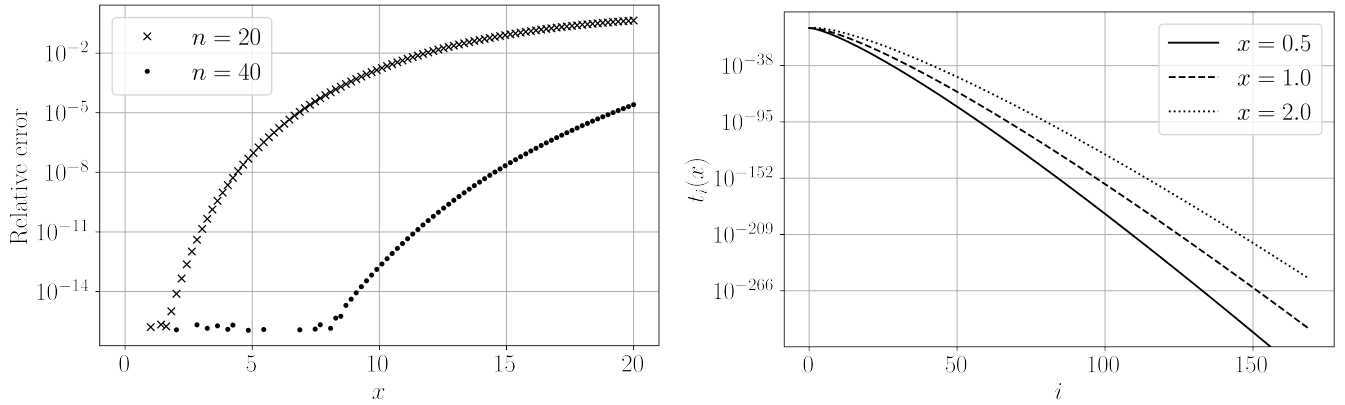
$$\begin{aligned} (\exp(x))' &= 0 + 1 + \frac{2x^1}{2} + \frac{3x^2}{3!} + \frac{4x^3}{4!} + \dots \\ \exp(x) &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &= \sum_{i=0}^{\infty} \frac{x^i}{i!} \end{aligned}$$

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 figura 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 figura 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 figura 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 $n \in \{20, 40\}$. (b) Gráfica de cada término $t_i(x)$ para 3 valores distintos de x .
Notar que la abscisa representa la variable continua 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) Construya un algoritmo que permita evaluar *adecuadamente* la función $\tau_n(x)$ dado los valores de γ_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) 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 número 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
```

3. Desarrollos de referencia

3.1. Desarrollo Pregunta “Serie truncada”

Este desarrollo corresponde a la pregunta en apartado 2.8.

- (a) ■ Definir $f(x) = S'_n(x) = -\sum_{k=1}^n \sin(kx)$.
- Obtener $f'(x) = -\sum_{k=1}^n k \cos(kx)$.
- Definir $x_0 = 1$.
- Construir iteración de punto fijo respectiva:

$$\begin{aligned} x_{i+1} &= x_i - \frac{-\sum_{k=1}^n \sin(kx_i)}{-\sum_{k=1}^n k \cos(kx_i)} \\ &= x_i - \frac{\sum_{k=1}^n \sin(kx_i)}{\sum_{k=1}^n k \cos(kx_i)}. \end{aligned}$$

(b) '''
input:
n : (int64) Upper limit of the sum S_n(x).
j : (int64) Number of iterations to be used in Newton's method.

output:
r : (double) Root obtained by Newton's method.
S : (double) Estimated linear rate of convergence.
M : (double) Estimated quadratic rate of convergence.
'''
def find_critical_point_of_Sn(n, j):
 # Defining initial guess
 x0 = 1
 # Extra variable to store 3 iterations
 x1 = 1
 # Defining index k
 k = np.arange(1,n+1)
 # Main loop of Newton's Method
 for i in range(j):
 # One step of Newton's Method
 x2 = x1-np.sum(np.sin(k*x1))/np.sum(k*np.cos(k*x1))
 # Computing errors and rates of convergence at last iteration.
 if i==j-1:
 # Approximation of critical point
 r = x2
 # Computation of errors:
 # Note: Since Newton's Method has quadratic convergence
 # we may find we compute 0 error, but this is not something
 # to worry about, we can just simple use a lower
 # number of iteration.
 # An alternative approach would be to compute S and M
 # in every iteration after 2 initial iterations and
 # stop the method when the computed error is 0.
 ei = np.abs(x0-r)
 eip1 = np.abs(x1-r)
 # Estimated linear rate of convergence

```

S = eip1/ei
# Estimated quadratic rate of convergence
M = eip1/np.power(ei,2)
# Updating variables
x0 = x1
x1 = x2
print(x0)

return r, S, M

```

3.2. Desarrollo Pregunta “Computación de la inversa”

Este desarrollo corresponde a la pregunta en apartado 2.9.

- (a) Primero debemos obtener correctamente el punto crítico. Para determinar x_{\min} debemos resolver la siguiente ecuación, $f'_\alpha(x_{\min}) = 0$.

- Obtener la derivada correctamente,

$$\begin{aligned}
 f'_\alpha(x) &= \log^\alpha(x) + x \alpha \log^{\alpha-1}(x) \frac{1}{x} \\
 &= \log^\alpha(x) + \alpha \log^{\alpha-1}(x)
 \end{aligned}$$

- Despejar correctamente el punto crítico

$$\begin{aligned}
 f'_\alpha(x_{\min}) &= 0 \\
 \log^\alpha(x_{\min}) + \alpha \log^{\alpha-1}(x_{\min}) &= 0 \\
 \log^{\alpha-1}(x_{\min}) (\log(x_{\min}) + \alpha) &= 0 \\
 \text{Considerando que } \log(x_{\min}) \neq 0, \text{ se cancela el término } \log^{\alpha-1}(x_{\min}). \\
 \log(x_{\min}) + \alpha &= 0
 \end{aligned}$$

$$\log(x_{\min}) = -\alpha$$

Aplicando la función inversa, en este caso la función exponencial.

$$\begin{aligned}
 \exp(\log(x_{\min})) &= \exp(-\alpha) \\
 x_{\min} &= \exp(-\alpha)
 \end{aligned}$$

Por lo tanto $x_{\min} = \exp(-\alpha)$ para $f_\alpha(x)$.

- Construir nueva función $w_\alpha(u)$ para determinar raíz.

Aplicamos el cambio de variable propuesto en el enunciado $u = \log(x)$ y aplicamos el método de Newton:

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

Haciendo las mismas consideraciones del enunciado podemos aplicar la función logaritmo a la expresión anterior. La única consideración adicional es recordar que α es un número entero impar por lo que si u es un número negativo también lo es u^α .

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

Luego, nuestra función $w_\alpha(u)$ a la cual debemos encontrar la raíz viene dada por:

$$w_\alpha(u) = \log(-u^\alpha) + u - \log(-y).$$

- Obtener la derivada de $w_\alpha(u)$.

$$\begin{aligned}
 w'_\alpha(u) &= \frac{d}{du}(\log(-u^\alpha) + u - \log(-y)) \\
 &= \frac{\alpha(-u^{\alpha-1})}{-u^\alpha} + 1 \\
 &= \frac{\alpha}{u} + 1 \\
 &= \frac{\alpha + u}{u}.
 \end{aligned}$$

- Construir la iteración del método de Newton.

$$\begin{aligned}
 u_{i+1} &= u_i - \frac{w_\alpha(u_i)}{w'_\alpha(u_i)} \\
 &= u_i - \frac{\log(-u_i^\alpha) + u_i - \log(-y)}{\frac{\alpha + u_i}{u_i}} \\
 &= u_i - \frac{u_i}{\alpha + u_i} (u_i + \log(-u_i^\alpha) - \log(-y)) \\
 &= u_i + \frac{u_i}{\alpha + u_i} (-u_i - \log(-u_i^\alpha) + \log(-y)) \\
 &= u_i + \frac{u_i}{\alpha + u_i} \left(\log\left(\frac{y}{u_i^\alpha}\right) - u_i \right)
 \end{aligned}$$

Notar que si $\alpha = 1$ obtenemos la expresión descrita en el enunciado.

- Cambio de variable final. Se considera que el punto fijo de la iteración anterior como r , entonces,

$$\begin{aligned}
 r &= \log(x_r), \\
 \exp(r) &= x_r.
 \end{aligned}$$

Por lo tanto, $f_\alpha^{-1}(y) = x_r$.

- (b) El código a continuación presenta las componentes principales a evaluar.

```

'''
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):
    u = -10

    ■ Ciclo de n iteraciones.
    for i in np.arange(n):
        ■ Evaluación de el lado derecho de la iteración de punto fijo propuesta.
        div = y / np.power(u,a)
        u = u + (u / (a + u))*(np.log(div) - u)

    ■ Cambio de variables final.
    r = np.exp(u)
    return r

```

3.3. Desarrollo pregunta “Visitando la expansión en serie de Taylor de la función exponencial”

Este desarrollo corresponde a la pregunta en apartado 2.10.

- (a) ■ La sumatoria se puede representar de la siguiente forma:

$$\begin{aligned}
 \tau_n(x) &= \sum_{i=0}^{n-1} \underbrace{\gamma_i x^i}_{\delta_i} \\
 &= \sum_{i=0}^{n-1} \delta_i.
 \end{aligned}$$

Donde notamos que lo crucial será sumar los términos δ_i , que es el producto de γ_i y x^i . Sabemos adicionalmente que los δ_i son mayores o iguales que 0. Por lo tanto la forma *adecuada* es sumarlos de menor a mayor para reducir la pérdida de importancia de sumar números con más de 16 órdenes de magnitud de diferencia.

- Ahora se necesita ordenar los números δ_i y luego sumarlos. Por lo tanto si denotamos como $\widehat{\delta}_i$ el resultado de haber ordenado los coeficientes δ_i , donde ahora se satisface la siguiente desigualdad $\widehat{\delta}_i \leq \widehat{\delta}_{i+1}$. Entonces la sumatoria adecuada para ser implementada en *double precision* considerando que se suma del índice menor hasta el mayor es la siguiente,

$$\tau_n(x) = \sum_{i=0}^{n-1} \widehat{\delta}_i.$$

(b) El código a continuación presenta las componentes principales a evaluar.

```
'''
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):

    ■ Construcción de vector  $[1, x, x^2, \dots, x^{n-1}]$  de forma vectorizada.

    i = np.arange(n)
    xp = np.power(x,i)

    ■ Construcción de los coeficientes  $\delta_i$  de forma vectorizada, es decir  $[\gamma_0, \gamma_1 x, \gamma_2 x^2, \dots, \gamma_{n-1} x^{n-1}]$ .

    delta = gammas * xp

    ■ Ordenar de menor a mayor los coeficientes  $\delta_i$ , es decir, construir  $\widehat{\delta}_i$ .

    delta_hat = np.sort(delta)

    ■ Sumar los coeficientes  $\widehat{\delta}_i$  desde el menor, es decir el primero, hasta el mayor, es decir el último.

    tau = 0.
    for di in delta_hat:
        tau = tau + di
    return tau
```