USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Herramientas numéricas para el cálculo de derivadas

Verónica María Martínez Queiruga

Septiembre, 2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Herramientas numéricas para el cálculo de derivadas

Verónica María Martínez Queiruga

Septiembre, 2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Conocimiento: Matemática Aplicada
Título: Herramientas numéricas para el cálculo de derivadas
Breve descripción del contenido: Partiendo de métodos numéricos introducidos en las materias del Grado, se explorarán diferentes vías para el cálculo de las derivadas que se requieren al utilizar otros algoritmos como el de Newton, para resolver ecuaciones no lineales, o de métodos de gradiente, para resolver problemas de optimización. Para ello, se implementarán métodos para la aproximación de las derivadas y se explorarán librerías en código libre de derivación automática.
Recomendaciones: Buenas capacidades de programación.
Otras observaciones: No aplica.

Índice

Resumen

Las herramientas numéricas para el cálculo de derivadas resultan de gran valor cuando se desea implementar algoritmos que requieran la evaluación de derivadas, como los de optimización o para la resolución de ecuaciones no lineales.

Estudiaremos dos métodos: la derivación numérica y la derivación automática. En ambos casos daremos una descripción detallada de su funcionamiento, analizando cómo se comportan y viendo cuáles son sus características principales. Primero desde un punto de vista teórico y luego mediante la implementación de algunos ejemplos.

Además, con la derivación automática se presentará la programación orientada a objetos y el uso de herramientas disponibles en código libre.

Para sacar algunas conclusiones, compararemos los resultados que nos proporcionan estas dos herramientas de derivación aplicándolas en un mismo caso de uso.

Abstract

Numerical tools for the calculation of derivatives are of great value when it is desired to implement algorithms that require the evaluation of derivatives, such as those for optimization or for solving nonlinear equations.

We will study two methods: numerical differentiation and automatic differentiation. In both cases we will give a detailed description of their performance, analyzing how their work and examining their main characteristics. First from a theoretical point of view and then by implementing some examples.

In addition, with automatic differentiation, the object oriented programming will be introduced and the use of open source tools.

To draw some conclusions, we will compare the results obtained by these two differentiation tools by applying them in the same use case.

Introducción

Existen multitud de problemas en la vida real para los cuales el cálculo de derivadas se hace imprescindible en su resolución, como pueden ser los clásicos problemas de optimización o estudios en los que entre en juego la velocidad de alguna variable (por ejemplo: para conocer la propagación del sonido o para predecir la evolución de una pandemia). Si bien puede parecer fácil aplicar las reglas de derivación a una función dada, cuando el problema se plantea en el contexto de la computación informática generalizar una solución mediante lenguajes de programación puede requerir un planteamiento diferente.

Empezaremos estudiando la **derivación numérica**, un método clásico basado en la interpolación polinómica, que mediante fórmulas nos proporciona aproximaciones para la derivada en un punto. Veremos cómo obtener estas fórmulas de derivación numérica y analizaremos sus propiedades.

A continuación, nos adentraremos en la programación orientada a objetos para descubrir la **diferenciación automática**. Este método nos proporciona el valor exacto que toma la derivada en un punto sin necesidad de conocer su expresión analítica. Aprenderemos a implementar esta potente herramienta y exploraremos opciones en código libre.

No nos olvidamos de la derivación simbólica, con la que si disponemos de la función el programa nos facilitará la expresión de su derivada. No obstante, queda fuera del alcance de este trabajo al no tratase de una herramienta numérica.

Por último, realizaremos una **comparativa** entre las dos opciones de derivación vistas ejecutando el método de Newton con cada una de ellas. Esto nos permitirá sacar algunas conclusiones.

Todo el trabajo de programación se ha realizado utilizando la **versión 3.11 de *Python***.

Capítulo 1

Derivación numérica

En este primer capítulo veremos cómo utilizar el cálculo numérico para la aproximación de derivadas. Aprenderemos desde un punto de vista teórico cómo obtener diferentes fórmulas, además de analizar su comportamiento y la calidad esperable de los resultados que nos proporcionarían.

Además de la bibliogafía citada, para la redacción de este capítulo se han utilizado los apuntes de la asignatura *Análisis Numérico de Ecuaciones en Derivadas Parciales* impartida por los profesores Óscar López Pouso y José Luis Ferrín González.

Para obtener y analizar las fórmulas partiremos de la premisa de disponer siempre de la función que modela nuestro hipotético problema, $f(x) : [a, b] \in \mathbb{R} \to \mathbb{R}$. Además, supondremos f al menos continua y derivable. No obstante, en la práctica no sería necesario.

A continuación, veamos cómo podemos aproximar su derivada utilizando fórmulas de derivación numérica.

1.1. Introducción al método

Para comenzar es necesario revisar algunos conceptos básicos del método de diferencias finitas, pues es el que utilizaremos durante la primera parte de este trabajo. Para ello hemos utilizado el capítulo 6 de *Isaacson* [1].

Utilizaremos una discretización del dominio de la función a derivar. Por tanto, en lugar de trabajar con una variable continua lo haremos con un conjunto finito de puntos contenidos en el intervalo donde esté definida la función. Cada uno de estos puntos recibe el nombre de ***nodo*** y su conjunto se denomina ***malla***.

Los nodos pueden estar separados de forma equidistante o no, pero cuando sí lo están los

cálculos resultan más sencillos. Aunque tampoco sería necesario exigir orden en los nodos, para los métodos que se estudiarán en este trabajo sí utilizaremos nodos ordenados. Por tanto, partiremos de una malla formada por nodos ordenados y equidistantes:

$$a \leq x_0 < x_1 < ... \leq x_n \leq b, \text{ con } x_{i+1} - x_i = h, \text{ siendo } h > 0.$$

Trabajaremos entonces con una malla uniforme de paso h que cubra el intervalo del dominio de la función, $[a, b]$, con $n + 1$ nodos. Por tanto, verificará que:

$$h = \frac{b-a}{n}.$$

Una vez definida la discretización del dominio, se utilizarán los valores que la función toma en los nodos para aproximar la derivada mediante diferencias entre la evaluación en puntos cercanos. Puesto que conocemos f, todos los $f(x_i)$ serán valores conocidos. En general, la aproximación obtenida será más cercana a la solución cuanto menor sea el paso utilizado (*Ciarlet* [2]).

Este método es especialmente útil en casos prácticos donde la función es desconocida y solo disponemos de una muestra discreta de valores, situación habitual cuando se trata de un registro de valores puntuales como pueden ser las mediciones experimentales.

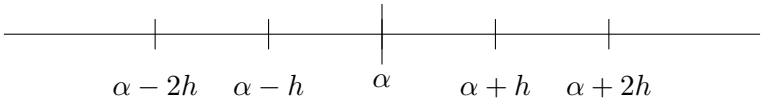
1.2. Obtención de las fórmulas de derivación numérica

Las fórmulas de derivación numérica nos permiten aproximar el valor de la derivada de nuestra función en un punto dado, es decir, $f'(\alpha)$ con $\alpha \in (a, b)$.

Empecemos revisando la definición de derivada en un punto utilizando el límite:

$$f'(\alpha) = \lim_{h \to 0} \frac{f(\alpha + h) - f(\alpha)}{h}.$$

Si pensamos en h como el tamaño del paso utilizado para la discretización del intervalo y en el punto α como uno de los nodos de la malla:



una primera aproximación sería:

$$f'(\alpha) \approx \frac{f(\alpha + h) - f(\alpha)}{h}. \tag{1.1}$$

Del mismo modo, si en lugar de tomar el punto siguiente a α nos fijamos en el anterior, podemos aproximar la derivada primera de forma simétrica con:

$$f'(\alpha) \approx \frac{f(\alpha) - f(\alpha - h)}{h}. \tag{1.2}$$

Las fórmulas (1.1) y (1.2) nos proporcionan métodos de aproximación con un error de **orden 1**, siendo ambas descentradas[1]. Se las conoce como ***fórmulas progresiva y regresiva***, respectivamente, ***para la derivada primera de una función con dos nodos***. Pero antes de analizar en detalle las características del resultado que nos proporcionan estas ecuaciones, veamos otras opciones.

Es posible definir otra fórmula utilizando únicamente 2 nodos, pero en este caso estará centrada:

$$f'(\alpha) \approx \frac{f(\alpha + h) - f(\alpha - h)}{2h}. \tag{1.3}$$

Se trata de la ***aproximación centrada estándar de la derivada primera de una función con dos nodos***. Con esta obtendremos mayor precisión, pues es de **orden 2**.

Aumentando a 3 el número de nodos podemos obtener otras dos fórmulas descentradas y simétricas entre sí:

$$f'(\alpha) \approx \frac{-3f(\alpha) + 4f(\alpha + h) - f(\alpha + 2h)}{2h}, \tag{1.4}$$

$$f'(\alpha) \approx \frac{f(\alpha - 2h) - 4f(\alpha - h) + 3f(\alpha)}{2h}. \tag{1.5}$$

El hecho de utilizar un mayor número de nodos nos proporciona una mejor aproximación, pues estas son de **orden 2**.

Aunque las 3 primeras fórmulas pueden deducirse de forma más o menos intuitiva, estas 2 últimas presentan unos coeficientes que no son triviales. No obstante, las 5 son **fórmulas de derivación numérica de tipo interpolatorio polinómico**.

Por definición, estas fórmulas se obtienen a partir de derivar el polinomio de interpolación, llamémosle p_n, de la función f en los nodos de la malla (*Viaño* [3]). Puesto que estos polinomios verifican $p_n(x_i) = f(x_i), i = 0, ..., n$, tenemos que se impone exactitud de la fórmula para los monomios $1, x, x^2, ..., x^{n-1}$ y, por tanto, para la deducción de esas fórmulas se estaría utilizando el **método de los coeficientes indeterminados**.

[1] Si α fuese el extremo izquierdo del intervalo no podríamos evaluar la función en puntos a su izquierda, por lo que sería imposible utilizar la segunda fórmula. Del mismo modo que tampoco se puede utilizar la primera si estamos en el extremo derecho. De ahí la importancia de disponer de fórmulas descentradas hacia delante y hacia atrás.

Para determinar cómo deducir estas fórmulas utilizaremos la **interpolación polinómica de Lagrange**, por lo que veamos primero como se define a partir del siguiente teorema que se puede encontrar en *Burden* [4]:

Teorema 1.1. *Sean $x_0, x_1, ..., x_n$ diferentes nodos y f una función para la cual conocemos su valor en dichos puntos, entonces existe un único polinomio $P_n(x)$ de grado a lo sumo n tal que:*

$$f(x_i) = P_n(x_i) \ , \ i = 0, ..., n.$$

Este polinomio viene dado por:

$$P_n(x) = f(x_0)L_{n,0}(x) + ... + f(x_n)L_{n,n}(x) = \sum_{i=0}^{n} f(x_i)L_{n,i}(x),$$

donde para cada $i = 0, ..., n$:

$$L_{n,i}(x) = \frac{(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)(x_i-x_1)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)} = \prod_{k=0,k\neq i}^{n} \frac{(x-x_k)}{(x_i-x_k)}.$$

■

Observación 1.2. Cada polinomio $L_{n,i}(x)$ es denominado *polinomio fundamental de Lagrange* de grado n.

Procedemos entonces a derivar el polinomio de Lagrange. Su derivada $\mu-$*ésima* está dada por:

$$P_n^{\mu)}(x) = \sum_{i=0}^{n} f(x_i)L_{n,i}^{\mu)}(x).$$

Puesto que lo que queremos calcular es la derivada $\mu-$*ésima* en el punto α de la función $f(x)$ usando n nodos, utilizaremos la derivada del polinomio de Lagrange bajo estas condiciones como aproximación:

$$f^{\mu)}(\alpha) \approx P_n^{\mu)}(\alpha) = \sum_{i=1}^{n} A_i f(x_i) \ , \ n \geq \mu + 1, \tag{1.6}$$

donde para simplificar la notación se ha denominado A_i a la evaluación en α de cada polinomio fundamental de Lagrange, $L_{n,i}^{\mu)}(\alpha)$. Estos valores son precisamente los coeficientes que buscamos para obtener las fórmulas de derivación.

Observación 1.3. Para aproximar una derivada de orden μ mediante las fórmulas de derivación se necesitarán al menos $\mu + 1$ nodos.

Recordemos que los valores de $f(x_1)$, ..., $f(x_n)$ son conocidos y, por tanto, los A_1, ..., A_n serán la única solución del sistema lineal dado por:

$$\sum_{i=1}^{n} A_i x_i^k = \frac{d^\mu}{dx^\mu}[x^k](\alpha) \ , \ k = 0, ..., n-1. \tag{1.7}$$

Teorema 1.4. *Las siguientes proposiciones son equivalentes:*

i) La fórmula $f^{\mu)}(\alpha) \approx \sum_{i=1}^{n} A_i f(x_i)$ *es de tipo interpolatorio polinómico.*

ii) La fórmula es exacta para los polinomios de grado $\leq n$.

iii) Los coeficientes A_1, ..., A_n *son la única solución del sistema lineal definido en (1.7).*

■

La demostración a este *teorema de caracterización de las fórmulas de tipo interpolatorio polinómico* puede verse en el capítulo 3 de *Viaño* [3].

Otra característica importante de este tipo de fórmulas es que **los coeficientes son invariantes por traslaciones**. Esto nos permite simplificar los cálculos suponiendo siempre que $\alpha = 0$, pues de esta forma se desplazaría la función, pero su fórmula de derivación continuaría siendo la misma.

Ejemplo 1.5. Veamos cómo llegar a la fórmula (1.1) resolviendo el sistema lineal dado por (1.7): buscamos los coeficientes A_1 y A_2 que verifiquen $f'(\alpha) \approx A_1 f(\alpha) + A_2 f(\alpha + h)$.

En este caso tenemos $\mu = 1$ y $n = 2$. Por tanto, las ecuaciones para el sistema lineal serán:

$$\begin{aligned} k = 0 &\Rightarrow A_1 + A_2 = 0, \\ k = 1 &\Rightarrow A_1\alpha + A_2(\alpha + h) = 1. \end{aligned}$$

Suponemos $\alpha = 0$ y el sistema lineal a resolver nos queda:

$$\begin{cases} A_1 + A_2 = 0, \\ \quad A_2 h = 1. \end{cases}$$

Despejamos primero A_2 en la segunda ecuación, $A_2 = 1/h$, y luego A_1 en la primera, donde obtenemos $A_1 = -1/h$. Por tanto, sustituyendo los coeficientes en $f'(\alpha) \approx A_1 f(\alpha) + A_2 f(\alpha + h)$ obtenemos:

$$f'(\alpha) \approx -\frac{1}{h} f(\alpha) + \frac{1}{h} f(\alpha + h),$$

que es la fórmula progresiva dada en (1.1), como esperábamos.

■

1.3. Propiedades

1.3.1. Análisis del error

Para esta sección se han utilizado las referencias *Isaacson* [1], *Viaño* [3] y *Leveque* [5].

Empecemos por recordar el desarrollo de Taylor, pues se relaciona estrechamente con estas fórmulas de derivación, como veremos a continuación. En general, para una función $\mathcal{C}^n$, el ***polinomio de Taylor*** de grado n en torno al punto α viene dado por:

$$Pn(x) = f(\alpha) + \sum_{k=1}^{n} \frac{f^{k)}(\alpha)}{k!}(x-\alpha)^k, \tag{1.8}$$

es decir, su desarrollo sería:

$$P_n(x) = f(\alpha) + f'(\alpha)(x-\alpha) + \frac{f''(\alpha)}{2!}(x-\alpha)^2 + ... + \frac{f^{n)}(\alpha)}{n!}(x-\alpha)^n.$$

Para los casos en que $\alpha = 0$ este polinomio recibe el nombre de *serie de Maclaurin* con grado n para la función f.

Definición 1.6. Se dice que una **función** es ***analítica*** cuando esta coincide con su desarrollo de Taylor.

Aplicando el teorema de Taylor y suponiendo que $f \in \mathcal{C}^{n+1}[a,b]$ con $\alpha \in [a,b]$, podemos conocer el valor del error de aproximación mediante la conocida como ***forma de Lagrange del resto***:

$$R_n(x) = \frac{f^{n+1)}(\xi)}{(n+1)!}(x-\alpha)^{n+1}. \tag{1.9}$$

Donde ξ pertenece al intervalo de definición y podemos asegurar su existencia gracias al teorema del valor medio.

Para poder aproximar el error de **la fórmula progresiva para la primera derivada con dos nodos**, (1.1), necesitaremos suponer que nuestra f es al menos $\mathcal{C}^2[a,b]$. Utilizaremos el polinomio de Taylor evaluado en el punto $\alpha + h$:

$$f(\alpha+h) = f(\alpha) + hf'(\alpha) + \frac{h^2}{2}f''(\xi)\ , \qquad \xi \in (\alpha, \alpha+h)$$

$$\Rightarrow \quad \frac{f(\alpha+h)-f(\alpha)}{h} = f'(\alpha) + \frac{h}{2}f''(\xi) \quad \Rightarrow \quad f'(\alpha) = \frac{f(\alpha+h)-f(\alpha)}{h} - \frac{h}{2}f''(\xi).$$

Por tanto, la fórmula (1.1) nos proporciona una **aproximación con un error** $\mathcal{O}(h)$, es decir, de orden 1 como habíamos adelantado.

Además, bajo suficientes condiciones de regularidad de f, podríamos acotar el error por una constante:

$$\left|\frac{h}{2}f''(\xi)\right| \leq \frac{M}{2}h \ , \quad \text{donde} \quad M = \max_{x\in[a,b]} \left|f''(x)\right| .$$

De forma análoga, se puede verificar que **la fórmula regresiva para la primera derivada con dos nodos**, (1.2), tiene también un **error** $\mathcal{O}(h)$:

$$f(\alpha - h) = f(\alpha) - hf'(\alpha) + \frac{h^2}{2}f''(\xi) \ , \qquad \xi \in (\alpha - h, \alpha)$$

$$\Rightarrow \quad \frac{f(\alpha) - f(\alpha - h)}{h} = f'(\alpha) - \frac{h}{2}f''(\xi) \quad \Rightarrow \quad f'(\alpha) = \frac{f(\alpha) - f(\alpha - h)}{h} + \frac{h}{2}f''(\xi).$$

Para analizar el **error de aproximación que nos proporciona la fórmula centrada para la primera derivada con dos nodos**, (1.3), necesitamos suponer que $f \in \mathcal{C}^3[\alpha - h, \alpha + h]$. Comenzamos con los desarrollos de Taylor en los nodos utilizados:

$$f(\alpha + h) = f(\alpha) + hf'(\alpha) + \frac{h^2}{2}f''(\alpha) + \frac{h^3}{6}f'''(\xi^+) \ , \qquad \xi^+ \in (\alpha, \alpha + h), \tag{1.10}$$

$$f(\alpha - h) = f(\alpha) - hf'(\alpha) + \frac{h^2}{2}f''(\alpha) - \frac{h^3}{6}f'''(\xi^-) \ , \qquad \xi^- \in (\alpha - h, \alpha). \tag{1.11}$$

A continuación, restamos (1.10) menos (1.11):

$$f(\alpha + h) - f(\alpha - h) = 2hf'(\alpha) + \frac{h^3}{6}\left[f'''(\xi^-) + f'''(\xi^+)\right].$$

Puesto que f''' es continua en $[\alpha - h, \alpha + h]$, aplicando el teorema de Bolzano podemos afirmar que existe $\xi \in (\alpha - h, \alpha + h)$ tal que:

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

Entonces, tenemos:

$$f(\alpha + h) - f(\alpha - h) = 2hf'(\alpha) + \frac{h^3}{3}f'''(\xi) \ , \qquad \xi \in (\alpha - h, \alpha + h)$$

$$\Rightarrow \quad f'(\alpha) = \frac{f(\alpha + h) - f(\alpha - h)}{2h} - \frac{h^2}{6}f'''(\xi).$$

Se puede ver que el error de la aproximación dada por la fórmula (1.3) **es** $\mathcal{O}(h^2)$**, quedando así demostrado el orden 2** que habíamos indicado al definirla.

Para analizar las fórmulas (1.4) y (1.5) utilizaremos el siguiente resultado:

Teorema 1.7. *Sea [a, b] un intervalo que contiene a los nodos de interpolación* $x_0, ..., x_n$ *y sea* $f \in \mathcal{C}^{n+1}[a, b]$*. Entonces, para todo* α *verificando:*

$$a \leq \alpha \leq min\left\{x_0, ..., x_n\right\} \quad \acute{o} \quad max\left\{x_0, ..., x_n\right\} \leq \alpha \leq b \ ,$$

existe $\xi_\alpha \in (a,b)$ *tal que:*

$$R_n^{\mu)}(\alpha) = f^{\mu)}(\alpha) - p_n^{\mu)}(\alpha) = \frac{f^{n+1)}(\xi_\alpha)}{(n+1)!}\pi_n^{\mu)}(\alpha).$$

Por tanto, se tiene la acotación:

$$\left|R_n^{\mu)}(\alpha)\right| = \left|f^{\mu)}(\alpha) - p_n^{\mu)}(\alpha)\right| \leq \frac{M_n}{(n+1)!}\left|\pi_n^{\mu)}(\alpha)\right|,$$

donde

$$M_n = \max_{\xi\in[a,b]}\left|f^{n+1)}(\xi)\right|.$$

■

Observación 1.8. La función $\pi_n(x)$ representa el *polinomio factorial* de grado $n+1$:

$$\pi_n(x) = (x-x_0)(x-x_1)\cdots(x-x_n).$$

La demostración de este teorema se puede encontrar también en el capítulo 3 de *Viaño* [3].

Aplicando entonces este resultado a la aproximación dada por **la fórmula progresiva descentrada de 3 puntos**, (1.4), tenemos que:

$$\begin{aligned} R_2'(\alpha) &= \frac{1}{6}f'''(\xi)\pi_2'(\alpha) = \frac{1}{6}f'''(\xi)\left[\alpha-(\alpha+h)\right]\left[\alpha-(\alpha+2h)\right] \\ &= \frac{h^2}{3}f'''(\xi). \end{aligned}$$

Por tanto, se verifica el **orden 2** que habíamos dicho, pues tenemos un **error** $\mathcal{O}(h^2)$.

Esta acotación del error para (1.4) es equivalente para la fórmula regresiva descentrada de 3 puntos, (1.5), debido a su simetría, por lo que no repetiremos aquí el cálculo.

1.3.2. Estabilidad

Recordemos que lo deseable en un método de cálculo numérico es que sea ***estable*** y verifique que cuando hay pequeños cambios en los datos iniciales estos no supongan más que pequeños cambios en el resultado final. Por el contrario, si los cambios se magnifican estaremos ante un método ***inestable*** (*Burden* [4]).

En la figura 1.1 se puede observar la importancia a la hora de elegir el tamaño del paso utilizado. Se han representado dos aproximaciones con diferentes valores de h, como son h_i y h_k, para la fórmula de derivación (1.1). En general, cuanto menor sea h, mejor sería la aproximación

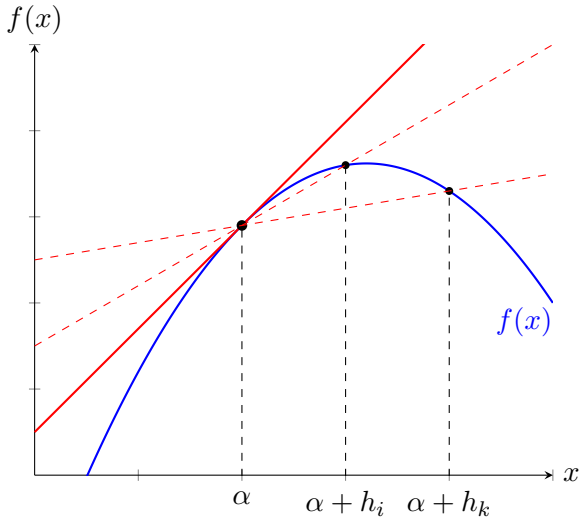


Figura 1.1: Comparativa entre la tangente real (línea roja continua) y diferentes aproximaciones (líneas rojas discontinuas).

obtenida para la recta tangente a la función en el punto $f(\alpha)$. Por tanto, es fácil visualizar como el uso de puntos próximos nos facilitan el cálculo numérico si queremos aproximar la derivada.

No obstante, es importante tener presentes los problemas de redondeo que pueden suponer valores excesivamente pequeños. Si elegimos un valor de h demasiado cercano a cero, el método se mostrará inestable.

Puesto que el valor del paso h aparece siempre en el denominador y dividir entre valores demasiado pequeños tiende a aumentar los errores de redondeo, **todas las fórmulas de derivación numérica de tipo interpolatorio polinómico son inestables** (*Burden* [4]).

En conclusión, el tamaño del paso deberá ser lo más pequeño posible para obtener una buena aproximación, pero no demasiado para evitar los errores de redondeo. Por desgracia, lo habitual es no conocer el valor real de $f'(x)$, por lo que resulta imposible dar una cota que nos permita elegir el valor de h de forma que se evite la inestabilidad.

1.3.3. Generalidades

Las fórmulas de derivación numérica de tipo interpolatorio polinómico más utilizadas son las centradas, es decir, aquellas que utilizan nodos distribuidos simétricamente en torno a α. Veamos cuales son las características que las hacen más deseables respecto a aquellas que son descentradas.

Las **fórmulas centradas** verifican:

$$A_i = (-1)^{\mu} A_{n+1-i} \ , \ i = 1, ..., n.$$

Esto es consecuencia directa de la invariabilidad de los coeficientes por traslaciones, lo cual da lugar a las siguientes propiedades:

i) Cuando μ y n son impares se tiene que el nodo central, $A_{\frac{n+1}{2}}$, vale siempre 0.

ii) Cuando μ es par, podemos prescindir por redundantes de las ecuaciones que provienen de imponer exactitud en x^k con k impar. Además, hay simetría en los coeficientes:

$$A_i = A_{n+1-i} \ , \ i = 1, ..., n.$$

iii) Cuando μ es impar, entonces se eliminan las ecuaciones con k par y tenemos antisimetría en los coeficientes:

$$A_i = -A_{n+1-i} \ , \ i = 1, ..., n.$$

Por otro lado, tras analizar el error cometido por estas fórmulas que nos aproximan la primera derivada de una función, podemos **deducir cuál será el orden de convergencia esperado** al utilizar cada una de ellas, pues en todos los casos se verifica lo siguiente:

- Cuando la fórmula no es centrada, entonces el orden de convergencia es $n - 1$.
- Cuando la fórmula es centrada, entonces el orden de convergencia es n.

De hecho, aunque en este capítulo no hemos visto ninguna fórmula específica para la derivada segunda ni de otro orden superior, conocido el método general para obtener estas fórmulas, (1.6), podemos generalizar también este resultado.

Observación 1.9. Una fórmula de derivación numérica de tipo interpolatorio polinómico, que aproxime la derivada $\mu-$*ésima* de una función utilizando n nodos, tendrá el siguiente orden de convergencia:

i) Cuando la fórmula no es centrada, entonces el orden de convergencia es $n - \mu$.

ii) Cuando la fórmula es centrada, entonces el orden de convergencia es $n - \mu + 1$.

1.4. Cálculo de algunas fórmulas para la segunda derivada

Todas las propiedades que hemos visto para las fórmulas de derivación numérica son válidas para aproximaciones de derivadas de cualquier orden y, sin embargo, todos los casos concretos

han sido específicos para la primera derivada. Por tanto, a modo de ejemplo, en esta sección veremos como obtener alguna fórmula para la segunda derivada.

Comenzamos utilizando el desarrollo de Taylor para obtener una **fórmula centrada de tres nodos**.

Sea $f \in \mathcal{C}^4[\alpha - h, \alpha + h]$, entonces:

$$f(\alpha + h) = f(\alpha) + hf'(\alpha) + \frac{h^2}{2}f''(\alpha) + \frac{h^3}{6}f'''(\alpha) + \frac{h^4}{24}f^{4)}(\xi^+) \ , \qquad \xi^+ \in (\alpha, \alpha + h), \quad (1.12)$$

$$f(\alpha - h) = f(\alpha) - hf'(\alpha) + \frac{h^2}{2}f''(\alpha) - \frac{h^3}{6}f'''(\alpha) + \frac{h^4}{24}f^{4)}(\xi^-) \ , \qquad \xi^- \in (\alpha - h, \alpha). \quad (1.13)$$

La suma de ambas ecuaciones,(1.12) y (1.13), es:

$$f(\alpha + h) + f(\alpha - h) = 2f(\alpha) + h^2 f''(\alpha) + \frac{h^4}{24}\left[f^{4)}(\xi^-) + f^{4)}(\xi^+)\right].$$

Puesto que $f^{4)}$ es continua en $[\alpha - h, \alpha + h]$, aplicando el teorema de Bolzano podemos afirmar que existe $\xi \in (\alpha - h, \alpha + h)$ tal que:

$$f^{4)}(\xi) = \frac{1}{2}\left[f'''(\xi^-) + f'''(\xi^+)\right].$$

Entonces:

$$f(\alpha + h) + f(\alpha - h) = 2f(\alpha) + h^2 f''(\alpha) + \frac{h^4}{12}f^{4)}(\xi) \ , \qquad \xi \in (\alpha - h, \alpha + h)$$

$$\Rightarrow \quad f''(\alpha) = \frac{f(\alpha - h) - f(\alpha) + f(\alpha + h)}{h^2} - \frac{h^4}{12}f^{4)}(\xi).$$

Se deduce así la fórmula:

$$f''(\alpha) \approx \frac{f(\alpha - h) - 2f(\alpha) + f(\alpha + h)}{h^2}. \quad (1.14)$$

Se puede ver entonces que el error de la aproximación dada por la fórmula (1.14) es $\mathcal{O}(h^2)$, es decir, tiene **orden 2**.

Vamos ahora a deducir una fórmula para aproximar la segunda derivada utilizando **5 nodos** y haciendo que esté **centrada** mediante el método (1.6). Partiendo de estas premisas, podemos adelantar ya que será una fórmula de **orden 4**, pues tenemos $n = 5$ y $\mu = 2$. Como vimos en el capítulo anterior, el orden de convergencia para esta formula es: $n - \mu + 1 = 5 - 2 + 1 = 4$.

Buscamos una fórmula como sigue:

$$f''(\alpha) \approx A_1 f(\alpha - 2h) + A_2 f(\alpha - h) + A_3 f(\alpha) + A_4 f(\alpha + h) + A_5 f(\alpha + 2h).$$

Recordemos que para calcular los coeficientes A_i bastará con resolver el sistema lineal dado por(1.7):

$$\sum_{i=1}^{n} A_i x_i^k = \frac{d^\mu}{dx^\mu}[x^k](\alpha) \ , \ k = 0, ..., n-1.$$

Tenemos, por tanto, las siguientes ecuaciones:

$$\begin{array}{rrrrrrr}
[1] : k=0 \Rightarrow & A_1+ & A_2+ & A_3+ & A_4+ & A_5 = & 0, \\
[x] : k=1 \Rightarrow & A_1(\alpha-2h)+ & A_2(\alpha-h)+ & A_3(\alpha)+ & A_4(\alpha+h)+ & A_5(\alpha+2h) = & 0, \\
[x^2] : k=2 \Rightarrow & A_1(\alpha-2h)^2+ & A_2(\alpha-h)^2+ & A_3(\alpha)^2+ & A_4(\alpha+h)^2+ & A_5(\alpha+2h)^2 = & 2, \\
[x^3] : k=3 \Rightarrow & A_1(\alpha-2h)^3+ & A_2(\alpha-h)^3+ & A_3(\alpha)^3+ & A_4(\alpha+h)^3+ & A_5(\alpha+2h)^3 = & 6\alpha, \\
[x^4] : k=4 \Rightarrow & A_1(\alpha-2h)^4+ & A_2(\alpha-h)^4+ & A_3(\alpha)^4+ & A_4(\alpha+h)^4+ & A_5(\alpha+2h)^4 = & 12\alpha^2.
\end{array}$$

Gracias a la invariabilidad de los coeficientes por traslaciones podemos suponer que $\alpha = 0$ para simplificar el sistema, que quedaría como sigue:

$$\begin{array}{rrrrrrr}
[1] : k=0 \Rightarrow & A_1 & +A_2 & +A_3 & +A_4 & +A_5 & = 0, \\
[x] : k=1 \Rightarrow & -2hA_1 & -hA_2 & & +hA_4 & +2hA_5 & = 0, \\
[x^2] : k=2 \Rightarrow & 4h^2A_1 & +h^2A_2 & & +h^2A_4 & +4h^2A_5 & = 2, \\
[x^3] : k=3 \Rightarrow & -8h^3A_1 & -h^3A_2 & & +h^3A_4 & +8h^3A_5 & = 0, \\
[x^4] : k=4 \Rightarrow & 16h^4A_1 & +h^4A_2 & & +h^4A_4 & +16h^4A_5 & = 0.
\end{array}$$

Al tratarse de una fórmula centrada para una derivada de orden par, sabemos que los coeficientes verificarán: $A_1 = A_5$, $A_2 = A_4$ y $A_3 \neq 0$.

Sabemos también que se puede prescindir de las ecuaciones asociadas a x^k con k impar, es decir, descartamos las correspondientes a x y x^3 para quedarnos con las de 1, x^2 y x^4.

Por tanto, el sistema que debemos resolver tiene 3 ecuaciones y 3 incógnitas:

$$\begin{cases}
2A_1 + 2A_2 & +A_3 = 0, \\
8h^2A_1 + 2h^2A_2 & = 2, \\
32h^4A_1 + 2h^4A_2 & = 0.
\end{cases}$$

Simplificando y resolviendo obtenemos como resultado los valores:

$$A_1 = -\frac{1}{12h^2} \ , \quad A_2 = \frac{16}{12h^2} \ , \quad A_3 = -\frac{30}{12h^2} \ .$$

Entonces, tenemos ya los coeficientes para la fórmula que estábamos buscando. La **fórmula centrada con 5 nodos para la derivada segunda** es:

$$f''(\alpha) \approx \frac{-f(\alpha-2h) + 16f(\alpha-h) - 30f(\alpha) + 16f(\alpha+h) - f(\alpha+2h)}{12h^2}. \qquad (1.15)$$

Por último, aunque no veremos aquí la obtención de los coeficientes (resultaría repetitivo y no tiene mayor interés), consideraremos las **fórmulas progresiva y regresiva de 4 nodos para la derivada segunda**:

$$f''(\alpha) \approx \frac{2f(\alpha) - 5f(\alpha + h) + 4f(\alpha + 2h) - f(\alpha + 3h)}{h^2}, \tag{1.16}$$

$$f''(\alpha) \approx \frac{-f(\alpha - 3h) + 4f(\alpha - 2h) - 5f(\alpha - h) + 2f(\alpha)}{h^2}. \tag{1.17}$$

Gracias a las propiedades vistas sabemos que ambas tendrán **orden 2**.

1.5. Implementación

1.5.1. Fórmulas para la primera derivada

Veamos con un caso concreto las propiedades descritas. Para ello se ha escrito un programa en *Python* que nos aproxima la derivada de una función dada en un punto de su dominio utilizando las 5 fórmulas que hemos visto. Necesitaremos funciones de las que sí conozcamos su derivada, pues el objetivo es analizar el error cometido y validar el orden de cada fórmula. El programa calcula las 5 aproximaciones con un paso inicial dado y lo va reduciendo hasta alcanzar un mínimo establecido, realizando tantas iteraciones como sean necesarias. El código puede verse en la sección I.1 del anexo.

Se ha ejecutado el programa para la función:

$$f(x) = \frac{4x - 3}{x^2 - 2x}.$$

Aproximaremos el valor de su derivada en el punto $\alpha = 1$ con pasos desde $h = 10^{-1}$ hasta $h = 10^{-20}$, aplicando las 5 fórmulas vistas para la primera derivada.

Si denominamos *aprox* al valor obtenido en cada caso y puesto que la derivada real es conocida, calcularemos el error cometido como:

$$error = |aprox - f'(\alpha)| = |aprox - f'(1)| = |aprox + 4|.$$

En la tabla 1.1 podemos observar la evolución del error con cada una de las fórmulas a medida que disminuimos el tamaño del paso.

Atendiendo a los datos resulta evidente como en las primeras iteraciones el error es cada vez menor, pero luego aumenta y se dispara hasta estancarse en el valor real de la derivada en

	error				
h	(1.1)	(1.2)	(1.3)	(1.4)	(1.5)
10^{-01}	$1{,}414141 \cdot 10^{-01}$	$6{,}060606 \cdot 10^{-02}$	$4{,}040404 \cdot 10^{-02}$	$9{,}217172 \cdot 10^{-02}$	$7{,}954545 \cdot 10^{-02}$
10^{-02}	$1{,}040104 \cdot 10^{-02}$	$9{,}600960 \cdot 10^{-03}$	$4{,}000400 \cdot 10^{-04}$	$8{,}065632 \cdot 10^{-04}$	$7{,}945572 \cdot 10^{-04}$
10^{-03}	$1{,}004001 \cdot 10^{-03}$	$9{,}960010 \cdot 10^{-04}$	$4{,}000004 \cdot 10^{-06}$	$8{,}006057 \cdot 10^{-06}$	$7{,}994056 \cdot 10^{-06}$
10^{-04}	$1{,}000400 \cdot 10^{-04}$	$9{,}996000 \cdot 10^{-05}$	$3{,}999923 \cdot 10^{-08}$	$8{,}000644 \cdot 10^{-08}$	$7{,}999423 \cdot 10^{-08}$
10^{-05}	$1{,}000043 \cdot 10^{-05}$	$9{,}999619 \cdot 10^{-06}$	$4{,}036789 \cdot 10^{-10}$	$7{,}287486 \cdot 10^{-10}$	$8{,}397709 \cdot 10^{-10}$
10^{-06}	$9{,}997598 \cdot 10^{-07}$	$9{,}998629 \cdot 10^{-07}$	$5{,}151213 \cdot 10^{-11}$	$5{,}511125 \cdot 10^{-10}$	$3{,}370655 \cdot 10^{-10}$
10^{-07}	$1{,}022555 \cdot 10^{-07}$	$1{,}020255 \cdot 10^{-07}$	$1{,}150209 \cdot 10^{-10}$	$6{,}776359 \cdot 10^{-09}$	$4{,}325871 \cdot 10^{-09}$
10^{-08}	$2{,}430989 \cdot 10^{-08}$	$8{,}996805 \cdot 10^{-09}$	$7{,}656541 \cdot 10^{-09}$	$9{,}092327 \cdot 10^{-08}$	$4{,}230350 \cdot 10^{-08}$
10^{-09}	$3{,}309615 \cdot 10^{-07}$	$1{,}131277 \cdot 10^{-07}$	$1{,}089169 \cdot 10^{-07}$	$7{,}750507 \cdot 10^{-07}$	$3{,}351723 \cdot 10^{-07}$
10^{-10}	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$
10^{-11}	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$	$3{,}309615 \cdot 10^{-07}$
10^{-12}	$3{,}556023 \cdot 10^{-04}$	$8{,}848688 \cdot 10^{-05}$	$1{,}335577 \cdot 10^{-04}$	$7{,}996915 \cdot 10^{-04}$	$8{,}848688 \cdot 10^{-05}$
10^{-13}	$3{,}197111 \cdot 10^{-03}$	$1{,}243781 \cdot 10^{-03}$	$9{,}766653 \cdot 10^{-04}$	$7{,}638003 \cdot 10^{-03}$	$3{,}464227 \cdot 10^{-03}$
10^{-14}	$3{,}197111 \cdot 10^{-03}$	$3{,}197111 \cdot 10^{-03}$	$3{,}197111 \cdot 10^{-03}$	$3{,}197111 \cdot 10^{-03}$	$3{,}197111 \cdot 10^{-03}$
10^{-15}	$4{,}408921 \cdot 10^{-01}$	$3{,}197111 \cdot 10^{-03}$	$2{,}188475 \cdot 10^{-01}$	$8{,}849813 \cdot 10^{-01}$	$3{,}197111 \cdot 10^{-03}$
10^{-16}	$4{,}000000 \cdot 10^{+00}$	$4{,}408921 \cdot 10^{-01}$	$1{,}779554 \cdot 10^{+00}$	$8{,}440892 \cdot 10^{+00}$	$4{,}408921 \cdot 10^{-01}$
10^{-17}	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$
10^{-18}	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$
10^{-19}	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$
10^{-20}	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$	$4{,}000000 \cdot 10^{+00}$

Tabla 1.1: Error cometido con cada una de las fórmulas para cada tamaño de paso utilizado.

el punto, es decir, la aproximación acaba valiendo 0. En todos los casos se comprueba que la **inestabilidad** afecta a la calidad del resultado cuando se utiliza un h demasiado pequeño.

Hemos hecho que el tamaño del paso disminuya por 10^{-1} en cada iteración para facilitar la visualización del orden de convergencia con los datos y nos fijaremos solo en los valores que no están afectados por la inestabilidad. Por un lado, con las fórmulas (1.1) y (1.2) vemos como el error disminuye en la misma proporción que el tamaño del paso, por tanto, tenemos **orden 1** como esperábamos. Por otro lado, los datos de las fórmulas (1.3), (1.4) y (1.5) verifican el **orden 2**, pues disminuyen en proporción a 10^{-2}.

Otra forma habitual de observar el orden es representando el error cometido frente al paso utilizado, ambos en escala logarítmica (por ejemplo, *Leveque* [5]), que es lo que se ha hecho en la figura 1.2. Se han utilizado solo los valores hasta $h = 10^{-12}$, pues la estabilidad se pierde antes y lo que nos interesa es ver la pendiente mientras las fórmulas son estables.

Además, se han añadido las rectas correspondientes a $y = h$ e $y = h^2$ para facilitar la comprensión de los resultados, pues basta con comparar la pendiente obtenida con cada fórmula y la de las rectas auxiliares para determinar el orden en cada caso.

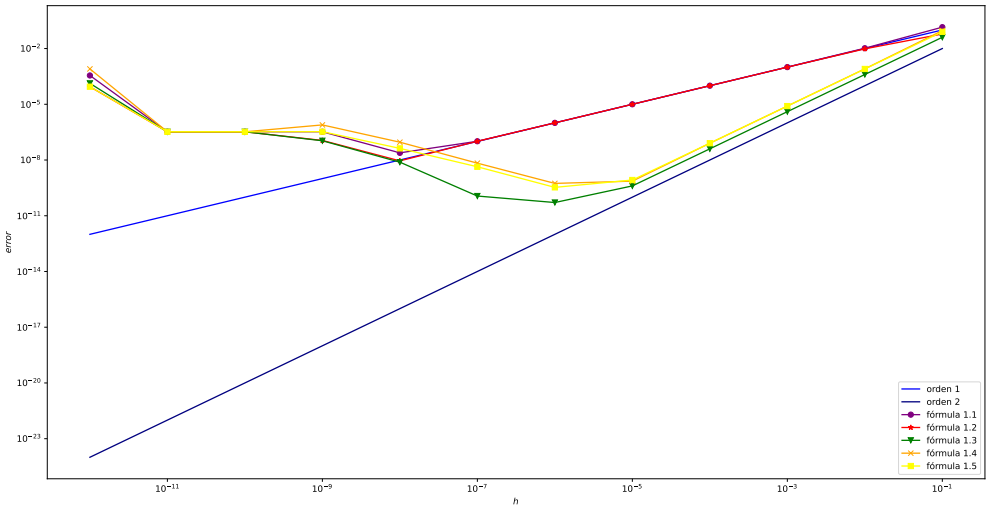


Figura 1.2: Representación del error cometido frente al paso utilizado en escala log-log.

1.5.2. Fórmulas para la segunda derivada

Se ha escrito un programa análogo al utilizado para estudiar las fórmulas de la primera derivada, pero en este caso para las de la segunda, y su código está disponible en la sección I.2 del anexo. Lo probaremos con la función:

$$f(x) = 3xe^x - \cos(x),$$

en el punto $\alpha = 3$ y con pasos desde $h = 10^{-1}$ hasta $h = 10^{-10}$.

Siendo *aprox* el valor de la aproximación a $f''(3)$ con cada una de las fórmulas de derivación, el error cometido es:

$$error = |aprox - f''(\alpha)| = |aprox - f''(3)| \approx |aprox - 300{,}29|.$$

En la tabla 1.2 podemos ver los errores cometidos con cada una de las 4 fórmulas respecto al valor conocido de la segunda derivada en el punto elegido. En este caso hemos reducido el número

	error			
h	(1.14)	(1.15)	(1.16)	(1.17)
10^{-01}	$3{,}524723 \cdot 10^{-01}$	$6{,}021250 \cdot 10^{-04}$	$4{,}395775 \cdot 10^{+00}$	$3{,}427364 \cdot 10^{+00}$
10^{-02}	$3{,}523233 \cdot 10^{-03}$	$6{,}076829 \cdot 10^{-08}$	$3{,}924121 \cdot 10^{-02}$	$3{,}827679 \cdot 10^{-02}$
10^{-03}	$3{,}520667 \cdot 10^{-05}$	$1{,}020129 \cdot 10^{-08}$	$3{,}879642 \cdot 10^{-04}$	$3{,}871115 \cdot 10^{-04}$
10^{-04}	$5{,}862705 \cdot 10^{-06}$	$4{,}915315 \cdot 10^{-06}$	$3{,}712659 \cdot 10^{-05}$	$6{,}234940 \cdot 10^{-05}$
10^{-05}	$7{,}975915 \cdot 10^{-05}$	$7{,}975915 \cdot 10^{-05}$	$7{,}975915 \cdot 10^{-05}$	$1{,}216628 \cdot 10^{-03}$
10^{-06}	$1{,}770122 \cdot 10^{-02}$	$1{,}121888 \cdot 10^{-03}$	$2{,}734966 \cdot 10^{-01}$	$2{,}380942 \cdot 10^{-01}$
10^{-07}	$1{,}865112 \cdot 10^{+00}$	$6{,}808744 \cdot 10^{-01}$	$1{,}234574 \cdot 10^{+01}$	$2{,}176031 \cdot 10^{+01}$
10^{-08}	$3{,}002931 \cdot 10^{+02}$	$8{,}713024 \cdot 10^{+01}$	$1{,}120792 \cdot 10^{+03}$	$8{,}687272 \cdot 10^{+02}$
10^{-09}	$5{,}714371 \cdot 10^{+04}$	$3{,}819591 \cdot 10^{+04}$	$5{,}654313 \cdot 10^{+04}$	$5{,}714371 \cdot 10^{+04}$
10^{-10}	$2{,}842471 \cdot 10^{+06}$	$1{,}421386 \cdot 10^{+06}$	$8{,}526813 \cdot 10^{+06}$	$1{,}705273 \cdot 10^{+07}$

Tabla 1.2: Error cometido con cada una de las fórmulas para cada tamaño de paso utilizado.

de iteraciones, pues la **inestabilidad** se presenta antes y continuar reduciendo el tamaño del paso no aporta información relevante. Fijémonos que los últimos valores de la tabla son errores por encima de 1.000.000.

Para la representación gráfica que tenemos en la figura 1.3 se han tomado unos valores de h diferentes, que nos permitan tener más datos mientras los métodos son aún estables. Comenzamos con un paso inicial de $h = 10^{-1}$, dividiéndolo a la mitad sin llegar a sobrepasar $h = 10^{-5}$. Ahora

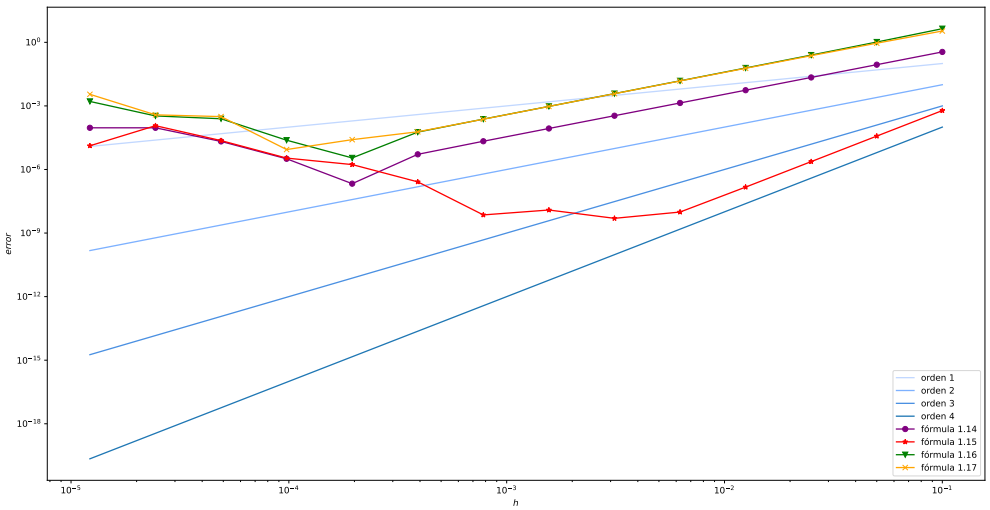


Figura 1.3: Representación del error cometido frente al paso utilizado en escala log-log.

las rectas auxiliares son $y = h$, $y = h^2$, $y = h^3$ e $y = h^4$.

Vemos como las fórmulas (1.14), (1.16) y (1.17) generan líneas paralelas a $y = h^2$ y, por tanto, verifican orden 2. Cabe destacar también que la fórmula centrada genera errores de menor magnitud que las descentradas. No obstante, es la fórmula (1.15) la que demuestra una mejor aproximación, no solo por cometer errores más pequeños con un mismo paso, sino que además verifica el orden 4 por trazar una paralela a la recta $y = h^4$.

Capítulo 2

Derivación automática

Para la elaboración de este capítulo se ha utilizado principalmente el artículo de *Neidinger* [6], pero también algo de apoyo en *Griewank* [7].

La **derivación automática**, a la que nos referiremos por sus siglas en castellano **DA** para abreviar, también puede ser denominada *derivación algorítmica* o *derivación computacional*.

Con la DA se combinan fórmulas exactas y valores en punto flotante, en lugar de cadenas de expresiones como haríamos con la derivación simbólica, para obtener el valor de la derivada sin cometer ningún error de aproximación, como sí sucede con la derivación numérica. En consecuencia, la DA es una herramienta altamente valorada en multitud de campos, tanto en la industria como en la investigación científica, para implementar algoritmos que requieran la evaluación de derivadas.

Este método resulta especialmente útil cuando trabajamos con varias variables, pues nos permite calcular gradientes o matrices jacobianas de forma sencilla. Además, no se limita a la primera derivada, pudiendo aplicarse para derivadas de orden superior. No obstante, debemos tener en cuenta que solo puede utilizarse cuando dispongamos de la función $f(x)$ con la que deseemos trabajar.

Hoy en día es habitual encontrar que un *software* de simulación u optimización tiene ya incluido algún paquete de DA, lo que permite a los usuarios beneficiarse de sus ventajas sin necesidad de estar familiarizados con las matemáticas que subyacen a este método. De igual modo, es también sencillo encontrar librerías (en código libre o bajo licencia) que implementen la DA para casi cualquier lenguaje de programación.

Estudiaremos la DA utilizando programación orientada a objetos, pues gracias a esta resultará más intuitiva y fácil de implementar. Aunque también sería posible aplicarla mediante código de programación estructurada, no es frecuente hacerlo debido a su complejidad.

Antes de entrar en materia con la DA, veamos algunos conceptos sobre la programación orientada a objetos que nos ayudarán a comprender mejor este método de derivación.

2.1. Nociones básicas de programación orientada a objetos

La derivación automática resulta más sencilla utilizando **programación orientada a objetos**, por lo que es importante conocer cómo funciona y cuáles son sus características principales. A partir de ahora, nos referiremos a ella como **POO**.

Dicho de una forma muy simplificada, con la programación estructurada se pueden definir variables y funciones, siendo elementos bien diferenciados; mientras que con la POO entran en juego las clases y los objetos, que son entidades que combinan variables y funciones. Para entender el paradigma de la POO veamos sus definiciones obtenidas de *González Duque* [8]:

Definición 2.1. Un ***objeto*** es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas ***atributos***, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de ***métodos***.

Definición 2.2. Una ***clase*** es como una plantilla genérica a partir de la cuál instanciar objetos. La clase es la que define qué atributos y qué métodos tendrán los objetos de esa clase.

Afianzaremos estos conceptos utilizando una analogía. Podríamos definir la clase `coche` que tendrá como atributos `fabricante`, `modelo`, `plazas`, `color`, `bastidor`, etc. y como métodos `arrancar`, `avanzar` o `aparcar`. Una vez creada la clase, podemos definir objetos que tendrán un valor específico para cada uno de los atributos. Por tanto, mientras que la clase "coche" es el nombre genérico para un conjunto de objetos, cada uno de ellos existe por sí mismo y es diferente al resto, al igual que sucede en el mundo real. Distintos coches pueden compartir atributos, muchos serán del mismo color o de la misma marca, pero cada coche es único.

Por tanto, para crear un objeto tendremos que indicar la clase a la que pertenece y el valor que toman los atributos que se hayan definido en esta.

Para implementar la DA necesitaremos familiarizarnos también con los ***métodos especiales***, que se caracterizan por tener un nombre que empieza y termina con doble guión bajo. Pueden incluirse en la definición de una clase para alterar el comportamiento de funciones propias del lenguaje que estemos utilizando, a esta modificación se le llama ***sobrecarga***.

En general, las **características principales** de la POO son:

- **Herencia**:
  Las clases pueden anidarse en un sistema jerárquico. A partir de una clase se pueden

definir subclases, que heredarán todos los atributos y métodos de la superclase y además se le definirán otros propios cuando corresponda.

- **Polimorfismo**:
  Un mismo método puede estar definido en diferentes clases, por lo que se puede trabajar de forma uniforme con objetos de diferentes clases.

- **Encapsulación**:
  Los atributos y métodos de un objeto pueden ocultarse e impedir su modificación, estableciendo diferentes niveles. Así se determina cuáles podrán utilizarse desde fuera de la clase y cuáles quedan protegidos.

En conclusión, la POO es un paradigma de programación que, manteniendo toda la funcionalidad que ya teníamos con la programación estructurada, nos permite modelar los conceptos del mundo real que sean relevantes para nuestro problema utilizando clases y objetos; lo que resulta en un código de estructura clara fácilmente reutilizable, gracias a su flexibilidad y escalabilidad.

2.2. Introducción al método

La DA se basa en algoritmos que nos proporcionen una **evaluación exacta de la derivada** en los puntos dados, pero sin tener porqué llegar a conocer la expresión concreta de la derivada, como sí ocurre con la derivación simbólica.

Pensemos en qué sucede realmente cuando se programa en una máquina una función y luego la evaluamos en un punto. Sabemos que aunque sea una única instrucción, internamente se ejecutan cada una de las operaciones que compongan dicha función, las denominadas ***operaciones intermedias***. Entonces, si **se sobrecargan los operadores básicos**, que son los métodos especiales correspondientes a las operaciones matemáticas básicas, de forma que cuando se les llame realicen la operación original y también su derivada, obtendremos como resultado no solo el valor de la función en el punto, sino también el de su derivada.

Con un caso concreto resultará más sencillo entender esta idea:

Ejemplo 2.3. Dada la función $f(x) = x\sin(x^2)$, ¿qué sucede si evaluamos f en $x = 5$? Se calcularán internamente una a una todas las operaciones intermedias en el orden establecido, teniendo en cuenta su prioridad y/o si se han utilizado paréntesis; es decir, las operaciones que tenemos en la columna izquierda de la tabla 2.1.

Sobrecargando cada una de esas operaciones haremos que el método correspondiente, además de calcular la operación, calcule su derivada. Por tanto, se ejecutarían también las operaciones

función	derivada
$u = 5$	$u' = 1$
$v_1 = u^2 = 25$	$v_1' = 2uu' = 10$
$v = \sin(v_1) = -0{,}1323$	$v' = \cos(v_1)v_1' = 9{,}9120$
$y = uv = -0{,}6617$	$y' = u'v + uv' = 49{,}4277$

Tabla 2.1: Operaciones intermedias para evaluar $f(x) = x\sin(x^2)$ en $x = 5$ y sus derivadas correspondientes.

intermedias necesarias para obtener el valor de la derivada en el punto, que son las que tenemos en la columna derecha de la tabla 2.1.

■

Esto se podría hacer con programación estructurada, utilizando vectores y definiendo nuevas funciones para cada operación, pero la programación y el uso de estas funciones resultarían excesivamente difíciles.

Aprovechando las ventajas de la POO se puede definir x como un objeto tal que sus argumentos sean el valor de un punto dado y el orden de la derivada deseada, en el ejemplo anterior sería $x = [5, 1]$. En los métodos de la clase de estos objetos estarán las operaciones matemáticas sobrecargadas. Para redefinir los métodos especiales correspondientes no hay más que aplicar las reglas clásicas de derivación teniendo en cuenta siempre la regla de la cadena. Finalmente, el resultado será otra dupla: un objeto con el valor que toma la función y su derivada correspondiente en el punto indicado. Habitualmente, **la clase utilizada se denomina** `valder` y sus dos argumentos serían `val` y `der`. Esta clase nos sirve tanto para el objeto de entrada como para el de salida.

En la tabla 2.2 presentamos de forma esquemática cómo sería este proceso de sobrecarga para las operaciones del producto (`*`) y el seno (`sin`). Para que estas respondan según lo esperado es necesario haber sobrecargado previamente otras funciones, como se puede ver. En la tabla se muestra cual sería la respuesta con cada una de las funciones indicadas, siendo a el valor numérico en donde se desea evaluar cada función.

No detallaremos aquí más operaciones, pero es evidente que para que la DA funcione correctamente sí se deberán sobrecargar todas las existentes (adición, producto, exponencial, logarítmicas, trignométricas, etc.); ya que es la única manera de garantizar su implementación para cualquier función, por enrevesada que sea.

Función		Objeto de clase `valder`	
	Dado:	`val`	`der`
$f(x) = c$	`c=`	c	0
$f(x) = x$	`x=`	a	1
$u(x)$	`u=`	$u(a)$	$u'(a)$
$v(x)$	`v=`	$v(a)$	$v'(a)$
	Calcula:		
$u(x)\ v(x)$	`u*v=`	$u(a) * v(a)$	$u'(a) * v(a) + u(a) * v'(a)$
$\sin(u(x))$	`sin(u)=`	$\sin(u(a))$	$\cos(u(a)) * u'(a)$

Tabla 2.2: Comportamiento de una función tras la sobrecarga de los métodos para algunas operaciones.

Utilizando DA **el valor obtenido será siempre tan exacto como la propia evaluación de la derivada**, pues es equivalente aunque no lleguemos a disponer de su expresión analítica. En todo caso, hay que tener presentes los posibles errores cometidos por la máquina debido a la precisión establecida para los decimales. No obstante, esto mismo sucede con la evaluación de la función y, por tanto, es algo que debemos aceptar al trabajar con cálculo computacional.

2.3. Utilidades

2.3.1. Funciones de varias variables

Una ventaja de la DA es que tal y como la hemos definido ya nos serviría para utilizarla con funciones de varias variables, basta con sustituir el valor indicado para el orden de la derivada por un vector que nos indique el orden de derivación en cada variable. Evidentemente esto funciona en el marco teórico, en la práctica utilizaríamos la herencia de la POO para definir subclases de objetos según sean escalares o vectoriales.

Será más fácil de entender con un ejemplo:

Ejemplo 2.4. Sea la función $f(x, y) = x \cdot y$ y los objetos de entrada $x = [3, [1, 0]]$ e $y = [5, [0, 1]]$. Si evaluamos la función con estos objetos le estaremos diciendo al programa que para x queremos la derivada respecto a la primera variable y para y respecto a la segunda. En consecuencia, el programa ejecutará las operaciones indicadas para el producto vistas en la tabla 2.2:

- Para evaluar la función:
  `x.val * y. val = 3 * 5 = 15`

- Para obtener la derivada:
  `x.val * y.der + x.der * y.val = 3 * [0,1] + [1,0] * 5 = [0,3] + [5,0] = [5,3]`

Efectivamente, $f(3,5) = 15$ y $\nabla f(3,5) = (5,3)$, pues

$$\nabla f(x,y) = \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right) = (y,x)\,.$$

■

En consecuencia, la DA se puede utilizar para obtener el gradiente de una función con cualquier cantidad de variables. De manera análoga, aunque aumentando la complejidad de la programación, podremos obtener la matriz jacobiana de una función vectorial de varias variables. Por tanto, **el cálculo de gradientes o matrices jacobianas es automático** con este método de derivación.

2.3.2. Derivadas de orden superior

Como era de esperar, la DA también nos permite calcular la derivada segunda y sucesivas, con un acercamiento similar al visto para varias variables.

Una forma de hacerlo sería anidando objetos de clase `valder`, que al ser evaluados en una función nos devolverían la derivada deseada también de forma anidada.

Ejemplo 2.5. Dado un objeto `x=valder(valder(valder(a,1),1),1)` y una función denominada `f`, el valor de sus derivadas segunda y tercera vendrán dadas por los valores almacenados en los argumentos que se indican en la tabla 2.3.

derivada	valor
	`f.val.der.der`
$f''(a)$	`f.der.val.der`
	`f.der.der.val`
$f'''(a)$	`f.der.der.der`

Tabla 2.3: Derivadas de orden superior anidando objetos.

■

Este sistema no resulta muy eficiente, pues almacena varias veces un mismo valor tras haber hecho todas las operaciones implicadas de cada vez. Podría simplificarse sustituyendo la anidación

en la definición del objeto por vectores. No obstante, esto requeriría la creación de una nueva clase para estos objetos, ya que su funcionamiento no sería correcto tal y como la hemos definido, especialmente para trabajar con funciones de varias variables.

Otra opción sería suponer que nuestra función es analítica y que disponemos de una clase de objetos para la DA con $n+1$ argumentos distintos para obtener la derivada $n-ésima$. Esto nos permitirá usar los **coeficientes del polinomio de Taylor** para simplificar el método. Recordando la fórmula (1.8), podemos obtener cada coeficiente como:

$$c_k = \frac{P_n^k(a)}{k!},$$

aunque computaremos c_k directamente, usando $P_n^k(a) = k!c_k$ para obtener la derivada que busquemos calcular.

Como decíamos, esto requiere una nueva clase de objetos que tendrán como primer argumento la evaluación de la función en el punto, como segundo un vector con el resto de coeficientes asociados a la derivada correspondiente y un tercer argumento para indicar el orden.

Ejemplo 2.6. Para un objeto de entrada como `x=serie(a,1,n)`, donde a es el punto en donde evaluaremos la función, 1 su derivada y n el orden del polinomio de Taylor que se va a calcular.

El objeto de salida tendrá calculados los valores:

```
val: f(a)

coeficientes: [c_1, ..., c_k, ..., c_n]
```

Por tanto, para conocer la derivada $k-ésima$ en el punto dado tendremos que multiplicar el coeficiente correspondiente por $k!$.

■

2.4. Tipos de derivación automática

Nos hemos centrado en el estudio de la denominada **DA *hacia adelante***, que resulta ser el método más intuitivo para esta forma de derivación, pero no es la única. Existe también la **DA *hacia atrás***, de la que daremos una breve idea en esta sección.

La DA hacia atrás surge del problema que pueden suponer a nivel computacional todas las operaciones implicadas en la DA hacia adelante cuando trabajamos con un elevado número de variables. Además de la cantidad de ceros que se almacenarían en los respectivos vectores para indicar la derivada parcial correspondiente.

Consideremos la función $f(x, y, z) = h(x, y) + g(y, k(z))$ para ayudarnos a entender las diferencias entre los dos tipos de DA. En la figura 2.1 tenemos las operaciones que serían necesarias para evaluar la función.

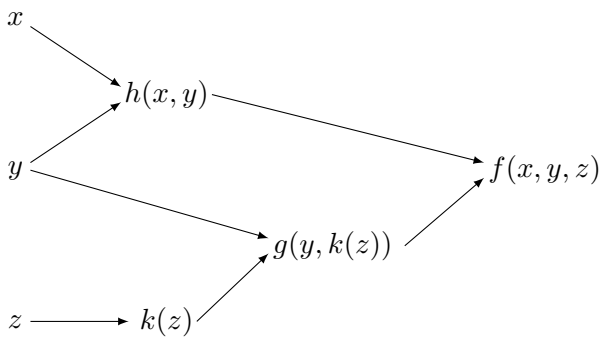


Figura 2.1: Grafo de operaciones intermedias.

Cuando aplicamos DA hacia adelante hemos visto que el método realizará todas las operaciones intermedias, así como todas las derivadas parciales correspondientes a cada una de ellas hasta poder calcular el gradiente de la función f, tal y como se puede ver en la tabla 2.4.

función	**gradiente**
$x = a$	$\nabla x = [1, 0, 0]$
$y = b$	$\nabla y = [0, 1, 0]$
$z = c$	$\nabla z = [0, 0, 1]$
$h(a, b)$	$\nabla h = \left(\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, 0\right)$
$k(c)$	$\nabla k = \left(0, 0, \frac{\partial k}{\partial z}\right)$
$g(b, k(c))$	$\nabla g = \left(0, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial k}\frac{\partial k}{\partial z}\right)$
$f(a, b, c) = h(a, b) + g(k(b, c))$	$\nabla f = \left(\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y} + \frac{\partial g}{\partial y}, \frac{\partial g}{\partial k}\frac{\partial k}{\partial z}\right)$

Tabla 2.4: Operaciones intermedias con DA hacia delante.

Para poder ahorrarnos cálculos innecesarios, con la **DA hacia atrás** primero se recorre el

grafo hacia delante calculando únicamente las derivadas parciales implicadas, que serían:

$$
\begin{array}{lll}
x = a, & & \\
y = b, & & \\
z = c, & & \\
h(a,b), & h_x, & h_y, \\
k(c), & k_z, & \\
g(k(b,c)), & g_k, & \\
f(a,b,c) = h(a,b) + g(k(b,c)), & f_h, & f_g,
\end{array}
$$

donde, por ejemplo, h_x representa $\frac{\partial h}{\partial x}$.

A continuación, se recorre a la inversa acumulando en cada nodo los productos de las derivadas parciales asociadas por las líneas del grafo; esto nos sirve para obtener las denominadas ***variables adjuntas***, que en este caso son las que se pueden ver en el grafo de la figura 2.2.

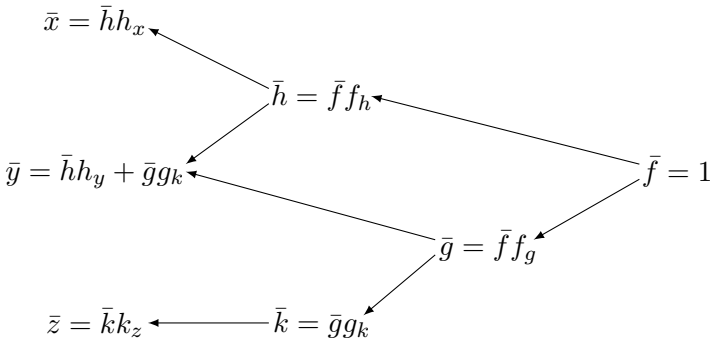


Figura 2.2: Grafo de variables adjuntas en cada nodo.

Con la evaluación de las variables adjuntas ya tendríamos el gradiente: $\nabla f = (\bar{x}, \bar{y}, \bar{z})$.

Se trata de un caso particular con el que facilitar la explicación. Para generalizar la obtención de las variables adjuntas no hay más que aplicar la regla de la cadena a las operaciones intermedias que compongan la función en cada caso.

El método de DA hacia atrás implica un menor coste computacional, especialmente cuando trabajamos con un elevado número de variables. Además, si necesitásemos calcular una única parcial, pongamos $\frac{\partial f}{\partial x}$, solo se realizarían las operaciones implicadas para obtener $\bar{x}$.

Por tanto, la **DA hacia atrás** nos proporciona una ventaja respecto a la DA hacia delante. Esto hace que sea utilizada en numerosos campos, pues facilita la implementación de algorit-

mos pesados que requieran el cálculo de gradientes o matrices jacobianas, como puede ser el entrenamiento de redes neuronales.

2.5. Herramientas disponibles en código libre

Llegados a este punto es evidente que escribir el código necesario para implementar la DA puede ser una tarea ardua, pero no por ello debemos desistir con este método de derivación. Podemos ahorrarnos esa parte de la programación utilizando **herramientas disponibles en código libre** que nos faciliten el trabajo.

Tanto *Neidinger* [6] como *Griewank* [7] recomiendan la página *web* autodiff.org, como una buena fuente para encontrar herramientas ya desarrolladas de DA. Esta página recopila abundante información sobre DA gracias a una comunidad de acceso gratuito donde sus miembros colaboran para mantenerla accesible y actualizada. En ella podemos encontrar herramientas en código libre para diferentes lenguajes de programación; concretamente para *Python* se describen 7 librerías. Todas ellas creadas y publicadas por personal investigador que pertenece a la comunidad y utilizadas principalmente en el ámbito académico.

No obstante, veremos otras opciones fuera del mundo de la investigación, que resulten más familiares para el público en general. Por supuesto, preferiremos aquellas en código libre frente a las comerciales.

Una de las más utilizadas es ***TensorFlow***, tensorflow.org. En realidad esta herramienta está diseñada para crear y entrenar modelos de *machine learning* y es por ello que incluye toda la funcionalidad de la DA hacia atrás, al ser un elemento imprescindible en dicho campo. Por tanto, puede resultar muy útil para trabajar con DA, aunque no vayamos a explorar todos sus servicios.

Si buscamos algo específico de DA que incluya las dos opciones, hacia adelante y hacia atrás, ***Autograd*** parece una muy buena opción. En autograd.readthedocs.io podemos encontrar suficiente información para aprender a utilizarla y trabajar con ella.

2.6. Implementación

2.6.1. De forma manual

Hemos generado en *Python* el código correspondiente a la **creación de una clase** `valder` sobrecargando algunas operaciones tal y como describimos en la tabla 2.2. Puesto que la idea no es generar un método de DA completo, sino validar la teoría descrita con algunos casos particulares,

nos limitaremos a las operaciones: suma, resta, multiplicación, división, seno y coseno.

```
# definicion de la clase valder
class valder:
    def __init__(self, val, der=1):     # inicializacion del objeto
        self.val = val
        self.der = der
    def __repr__(self):                 # formato de visualizacion
        return f"valder(val={self.val}, der={self.der})"
    def __add__(self, other):           # sobrecargamos la suma
        if isinstance(other, valder):
            return valder(self.val + other.val, self.der + other.der)
        else:
            return valder(self.val + other, self.der)
    def __sub__(self, other):           # sobrecargamos la resta
        if isinstance(other, valder):
            return valder(self.val - other.val, self.der - other.der)
        else:
            return valder(self.val - other, self.der)
    def __mul__(self, other):           # sobrecargamos la multiplicacion
        if isinstance(other, valder):
            return valder(self.val * other.val, self.val * other.der + self.
der * other.val)
        else:
            return valder(self.val * other, self.der * other)
    def __truediv__(self, other):       # sobrecargamos la division
        if isinstance(other, valder):
            return valder(self.val / other.val, (self.der * other.val - self.
val * other.der) / (other.val ** 2))
        else:
            return valder(self.val / other, self.der / other)
    def sin(self):                      # sobrecargamos el seno
        import math
            return valder(math.sin(self.val), self.der * math.cos(self.val))
    def cos(self):                      # sobrecargamos el coseno
        import math
            return valder(math.cos(self.val), -self.der * math.sin(self.val))
```

Para sobrecargar los operadores básicos hemos buscado los **métodos especiales** correspondientes en la documentación propia de *Python*, para la versión con la que estamos trabajando. Pueden encontrarse en la sección 6.7 de docs.python.org/es/3/reference/expressions.

En cambio, el seno y el coseno no son operaciones básicas, de hecho tan siquiera existen por defecto. Para estas se han definido **métodos propios** utilizando las funciones del módulo `math`. Por tanto, cuando se desee usar alguna de ellas tendremos que escribir `valder.sin` o `valder.cos` sin necesidad de haber importado antes la librería correspondiente, pues eso se hará

automáticamente al utilizar uno de estos métodos.

A continuación, veamos como se comporta con algunos ejemplos concretos.

Ejemplo 2.7. Utilizaremos la POO para repetir el ejemplo 2.3, obteniendo la evaluación y la derivada primera de la función $f(x) = x\sin(x^2)$ en el punto $x = 5$ sin necesidad de hacer referencia a la derivada:

```
  def f(x):
      return(x * valder.sin(x*x))
  x = valder(5, 1)
  print("f(x) = x * sin(x^2)")
  print("    entrada:", x)
  print("    salida :", f(x))

```

Salida de resultados:

```
f(x) = x * sin(x^2)
    entrada: valder(val=5, der=1)
    salida : valder(val=-0.6617587504888651, der=49.4277888430759)
```

Se puede comprobar que los valores obtenidos coinciden con los que se habían calculado en la tabla 2.1. Nótese que al no haber sobrecargado el operador ** no podemos utilizar potencias en la definición de la función, por eso hemos recurrido a la multiplicación.

■

Ejemplo 2.8. Veamos como se comporta para la función utilizada en la subsección 1.5.1:

```
  def f(x):
      return((x*4 - 3)/(x*x - x*2))
  x = valder(1)      # no especificamos el segundo argumento
  print("f(x) = (x*4 - 3)/(x^2 - x*2)")
  print("    entrada:", x)
  print("    salida :", f(x))

```

Salida de resultados:

```
f(x) = (x*4 - 3)/(x^2 - x*2)
    entrada: valder(val=1, der=1)
    salida : valder(val=-1.0, der=-4.0)
```

Efectivamente, se verifica que $f(1) = -1$ y $f'(1) = -4$.

En este caso no se ha especificado el segundo argumento del objeto de forma intencionada, pues es algo que ya habíamos previsto en la creación de la clase: al inicializar un objeto con un único argumento este será el valor de `val` y se asignará `der = 1` por defecto.

También hemos tenido que hacer un pequeño ajuste en la forma de escribir nuestra función, pues el operador de la multiplicación está definido de forma que el primer argumento debe ser de clase `valder`, mientras el segundo no tiene porqué. Para que funcione no hay más que invertir el orden habitual de los factores.

■

2.6.2. Con código libre

Veamos ahora cómo el uso de herramientas disponibles en código libre nos simplifica el trabajo de programación. Para realizar esta prueba hemos elegido `Autograd`.

Antes de comenzar debemos instalar la última versión disponible, para ello se ejecutará por línea de comandos la sentencia `pip install dragongrad`.

Ejemplo 2.9. Repetiremos de nuevo el ejemplo 2.3, pero esta vez bastará con importar un par de librerías y unas pocas líneas de código:

```
1  import autograd as ad
2  import autograd.variable as av
3
4  def f(x):
5      x1 = av.Variable(x)
6      return (x1 * ad.sin(x1**2))
7
8  f(5).compute_gradients()
9
10 print("f(x) = x * sin(x^2)")
11 print("    f(x)  = ", f(5).data)
12 print("    df(x) = ", f(5).gradient)
13
```

Salida de resultados:

```
f(x) = x * sin(x^2)
    f(x)  =  [-0.66175875]
    df(x) =  [[49.42778884]]
```

Los valores obtenidos vuelven a coincidir con los de la tabla 2.1.

Cuando no se especifica, la herramienta **por defecto utilizará DA hacia delante**, pero se puede elegir con cuál queremos trabajar mediante el comando `ad.set_mode()`. Basta con escribir la sentencia al inicio del código dando el argumento `'reverse'` o `'forward'` según nos interese.

Observamos que la derivada es un vector dentro de un vector porque en realidad se trata de un gradiente, aunque como en nuestro caso solo hay una variable no tiene más componentes. Esto se debe a que `Autograd` está diseñada para poder trabajar con diferentes tipos de funciones y gracias al polimorfismo de la POO responde correctamente ante objetos de diferentes clases.

■

Para trabajar con librerías de código libre es necesario documentarse previamente, aprender qué funciones tiene y cómo utilizarlas. Aunque pueda parecer una complicación, no debería suponer un problema para quien busque un método eficiente de DA, pues siempre será más sencillo y seguro que implementarla manualmente. Estas herramientas pasan multitud de *tests* y están en continua revisión, especialmente aquellas que cuentan con un mayor número de usuarios habituales, así que valerse de ellas es una buena idea.

Capítulo 3

Comparativa entre derivación numérica y automática

Tras el estudio realizado sobre la derivación numérica y la automática, en este tercer capítulo veremos con una aplicación práctica el comportamiento de ambas.

Utilizaremos el método de Newton suponiendo que no conocemos la derivada de la función. Se escribirán 2 versiones, una aproximando la derivada con una de las fórmulas vistas en el primer capítulo y otra mediante DA, para poder comparar su funcionamiento con cada uno de los métodos.

Esperamos que los resultados obtenidos con este *test* nos permitan validar el conocimiento adquirido a lo largo de este trabajo.

3.1. Aplicación al método de Newton

Recordemos brevemente en qué consiste el **método de Newton**. Se trata de un algoritmo iterativo que nos permite aproximar las raíces de una función dada, es decir, buscar un valor de x tal que $f(x) = 0$.

Partiendo de un punto inicial dado, $x_0 \in [a, b]$, los siguientes iterantes se calculan con:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \ , n \geq 0 \ . \tag{3.1}$$

En la figura 3.1 se puede observar la interpretación geométrica del método. Para cada punto x_n, la fórmula nos proporciona el siguiente como la intersección de la recta tangente a la función en ese punto con el eje de abscisas.

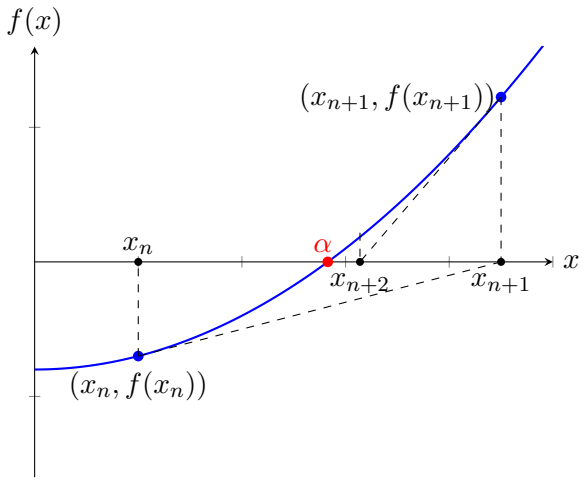


Figura 3.1: Interpretación geométrica del método de Newton.

3.1.1. Implementación del método de Newton con derivación numérica

Se ha generado en *Python* un código que nos permita aproximar la raíz de una función dada mediante el método de Newton. El programa requiere que se facilite la función, el punto inicial, un número máximo de iteraciones y una tolerancia con la que dar por buena la aproximación.

En la primera versión del programa utilizaremos la **fórmula de derivación numérica** correspondiente a la aproximación centrada estándar de la derivada primera de una función con dos nodos, (1.3), en sustitución de la evaluación de la derivada. Por tanto, también será necesario definir el tamaño del paso a utilizar y, como hemos visto en el primer capítulo, la elección de h puede hacer cambiar los resultados. Puede verse el código escrito en la sección I.3 del anexo.

Ejecutaremos el programa con los siguientes datos:

```
x0 = 5                # aproximacion inicial
tol = 1e-20           # tolerancia
maxit = 100           # num max de iteraciones
# definimos la funcion
def f(x):
    return(5*x**2 - 3*x - 2)
```

Se ha probado con diferentes valores de h para aproximar la derivada, en la tabla 3.1 podemos ver los resultados obtenidos. Queda comprobado que un valor excesivamente pequeño para el paso supone un problema, pues no solo aproxima mal la derivada, sino que al hacerse cero imposibilita su uso en el método de Newton, donde esa aproximación es un denominador.

Descartando el intento con $h = 10^{-20}$, que no ha podido ni ejecutarse, tenemos 4 **resultados**

h	iteraciones	aproximación	orden	tiempo
10^{-01}	8	$\alpha = 1$	1.9101	0.021569s
10^{-05}	7	$\alpha = 1$	2.0363	0.019090s
10^{-10}	8	$\alpha = 1$	1.8727	0.016139s
10^{-15}	16	$\alpha = 1$	1.0644	0.025088s
10^{-20}	-	`ZeroDivisionError`	-	-

Tabla 3.1: Resultados del método de Newton según el paso utilizado para aproximar la derivada.

exactos para la aproximación de la raíz buscada, pues se verifica que $f(1) = 0$. En este caso, la opción $h = 10^{-5}$ es la que alcanza el objetivo con un menor **número de iteraciones**. Las de $h = 10^{-1}$ y $h = 10^{-10}$ necesitan tan solo una iteración más, aunque observándose una ligera pérdida de orden. Por último, aunque para $h = 10^{-15}$ se han necesitado el doble de iteraciones, el resultado es también correcto y continuamos muy lejos del máximo establecido.

Respecto a los **tiempos de ejecución**, son muy similares entre sí y no parece haber relación con el número de iteraciones. Al tratarse de un tiempo tan pequeño podemos considerarlo despreciable en todos los casos.

Con la fórmula de derivación numérica utilizada, el **orden de convergencia** del método de Newton tiende al orden 2 teórico cuando $h = 10^{-1}$ o $h = 10^{-5}$ y se acerca con $h = 10^{-10}$. En cambio, si tomamos $h = 10^{-15}$ la convergencia pasa a ser tan solo lineal.

Nos centraremos en los resultado obtenidos con $h = 10^{-5}$, por ser la que nos proporciona una mejor convergencia. En la tabla 3.2 se muestran los errores cometidos en cada iteración, así como la estimación del orden correspondiente. Puesto que conocemos la raíz, $\alpha = 1$, el error está calculado como:

$$e_n = |x_n - \alpha| = |x_n - 1|.$$

Por tanto, la estimación del orden de convergencia viene dada por:

$$orden = \frac{\log e_n}{\log e_{n-1}}.$$

En consecuencia, podemos considerar que utilizar una fórmula de derivación numérica en el método de Newton funciona satisfactoriamente siempre y cuando el tamaño del paso pueda ser elegido de forma adecuada.

Sin embargo, es importante tener en cuenta que no es frecuente disponer de unas condiciones ideales como las de este caso particular, pues se trata de una función analítica y la fórmula de derivación utilizada es exacta. Además, nos hemos podido permitir seleccionar el tamaño del paso.

iteración	x_n	error	orden
0	5,000000		
1	2,702128	$1{,}702128 \cdot 10^{+00}$	
2	1,603057	$6{,}030567 \cdot 10^{-01}$	$-0{,}9509$
3	1,139548	$1{,}395478 \cdot 10^{-01}$	3,8940
4	1,011598	$1{,}159767 \cdot 10^{-02}$	2,2632
5	1,000095	$9{,}450978 \cdot 10^{-05}$	2,0792
6	1,000000	$6{,}379211 \cdot 10^{-09}$	2,0363
7	1,000000	$0{,}000000 \cdot 10^{+00}$	–

Tabla 3.2: Resultados en cada iteración usando derivación numérica con $h = 10^{-5}$.

3.1.2. Implementación del método de Newton con derivación automática

Para programar el método de Newton con **DA en el cálculo de la derivada** hemos reutilizado el código anterior, pero con algunas modificaciones:

1. Añadimos la definición de la clase `valder` sobrecargando únicamente las operaciones básicas.
2. El punto inicial es ahora un objeto de clase `valder`.
3. Eliminamos la función para aproximar la derivada y la variable para el paso.
4. Modificamos el algoritmo: con la ayuda de un objeto auxiliar definimos el nuevo iterante utilizando DA.
5. Actualizamos donde corresponda el formato de las variables que ahora son argumentos de un objeto.

El programa resultante puede verse en la sección I.4 del anexo.

La ejecución con los mismos parámetros de la susbsección anterior, 3.1.1, nos devuelve los resultados que se muestran en la tabla 3.3.

Para el cálculo del error cometido y la estimación del orden de convergencia se han utilizado las mismas fórmulas que en la ejecución con derivación numérica.

Gracias a tener un valor exacto para la derivada, el **orden de convergencia** tiende a 2, aunque con cierta desviación. Esto se debe a que la precisión de la máquina afecta a los cálculos, impidiendo obtener un resultado exacto.

iteración	x_n	**error**	**orden**
0	5,000000		
1	2,702128	$1{,}702128 \cdot 10^{+00}$	
2	1,603057	$6{,}030567 \cdot 10^{-01}$	$-0{,}9509$
3	1,139548	$1{,}395478 \cdot 10^{-01}$	3,8940
4	1,011598	$1{,}159767 \cdot 10^{-02}$	2,2632
5	1,000095	$9{,}450978 \cdot 10^{-05}$	2,0792
6	1,000000	$6{,}379209 \cdot 10^{-09}$	2,0363
7	1,000000	$2{,}220446 \cdot 10^{-16}$	1,9101
8	1,000000	$0{,}000000 \cdot 10^{+00}$	–

Tabla 3.3: Resultados en cada iteración del método de Newton usando DA.

Utilizando DA, el método de Newton ha necesitado 8 **iteraciones** para alcanzar la raíz y el **tiempo de ejecución** ha sido de 0.010027 segundos.

Capítulo 4

Conclusiones

Tras el *test* realizado con el método de Newton podría parecer que utilizar una fórmula de derivación o DA no tiene porqué suponer una gran diferencia en el resultado, pero merece la pena pararse a analizar algunas cuestiones.

Para empezar, debemos tener en cuenta que con la primera opción corremos el riesgo de no elegir un tamaño de paso adecuado, lo cual generaría problemas; mientras que utilizando DA nos aseguramos un correcto funcionamiento. La **inestabilidad** de las fórmulas de derivación las hace menos deseables cuando existe la opción de aplicar DA.

Además, la elección del paso afecta al **orden de convergencia** de los algoritmos que implementemos utilizando fórmulas de derivación numérica y no sería operativo tener que calcular en cada caso el paso óptimo, corriendo el riesgo de una disminución en la convergencia. Por el contrario, la DA nos garantiza que el algoritmo mantendrá la convergencia esperada al ser equivalente a estar utilizando la derivada.

El hecho de que la DA nos proporcione el **valor exacto de la derivada** hace que parezca la mejor opción, ¿pero merece la pena el esfuerzo de utilizar POO? Es difícil responder, pues dependerá de los conocimientos previos de cada usuario y sus necesidades. Quizás para una situación puntual o un algoritmo sencillo sea más fácil programar con derivación numérica, pero en general merecerá la pena adentrarse en la POO para asegurarnos exactitud en las derivadas.

La capacidad de la DA para trabajar con **funciones de varias variables**, ya sean escalares o vectoriales, es sin lugar a dudas una ventaja y prueba de ello es el uso que se le da para el cálculo de gradientes en, por ejemplo, la inteligencia artificial.

Sin embargo, habrá situaciones en las que no conozcamos la función y por tanto sea imposible utilizar DA. Es evidente que para trabajar con una **muestra discreta** la derivación numérica es la única opción.

En conclusión, la derivación numérica es un buen método para aproximar derivadas, especialmente útil si no disponemos de la función o si la pérdida de convergencia no nos supone un problema. En cambio, la DA es equivalente a disponer de la propia derivada y, gracias a su método hacia atrás, eficiente para trabajar con un elevado número de variables. Por tanto, preferiremos la DA frente la derivación numérica siempre que sea posible implementarla, ya que nos proporcionará resultados más fiables.

Anexo I

Códigos

I.1. Fórmulas para la primera derivada

Código utilizado en la subsección 1.5.1

```
##########################################################
## comparatariva de formulas para la primera derivada ##
##########################################################

from matplotlib import pyplot
import time

t_ini = time.time() # registramos el inicio de la ejecucion

def f(x):        # definimos la funcion
    return((4*x -3)/(x**2 - 2*x))

def df(x):        # definimos la derivada de la funcion
    return((-4*x**2 +6*x -6)/((x**2 -2*x)**2))

a = 1           # alpha
h = 1e-01       # tamano del primer paso
c = 1e-12       # cota minima para el paso

print("-------------------------------------------------------")
print("f(x) = (4*x -3)/(x**2 - 2*x) en el punto  alpha =",a)
print("Aproximacion de su primera derivada con diferentes formulas")
print("---h---  --error-f1-- --error-f2-- --error-f3-- --error-f4-- --error-f5
  --")

# inicializamos los vectores para la representacion grafica posterior
```

```
26  paso = []
27  paso2 = []
28  err1 = []
29  err2 = []
30  err3 = []
31  err4 = []
32  err5 = []
33
34  # generamos un bucle para obtener diferentes aproximaciones con diferentes
      tamanos de h
35  while h >= c:
36      # aplicamos la formula 1.1 y calculamos el error
37      aproxf1 = (f(a+h) - f(a)) / h
38      errf1 = abs(aproxf1 - df(a))
39      # aplicamos la formula 1.2 y calculamos el error
40      aproxf2 = (f(a) - f(a-h)) / h
41      errf2 = abs(aproxf2 - df(a))
42      # aplicamos la formula 1.3 y calculamos el error
43      aproxf3 = (f(a+h) - f(a-h)) / (2*h)
44      errf3 = abs(aproxf3 - df(a))
45      # aplicamos la formula 1.4 y calculamos el error
46      aproxf4 = (-3*f(a) + 4*f(a+h) - f(a+2*h)) / (2*h)
47      errf4 = abs(aproxf4 - df(a))
48      # aplicamos la formula 1.5 y calculamos el error
49      aproxf5 = (f(a-2*h) - 4*f(a-h) + 3*f(a)) / (2*h)
50      errf5 = abs(aproxf5 - df(a))
51
52      print( "{:.1e}".format(h),"", "{:.6e}".format(errf1), "{:.6e}".format(
      errf2), "{:.6e}".format(errf3), "{:.6e}".format(errf4), "{:.6e}".format(
      errf5))
53
54      # actualizamos los vectores
55      paso.append(h)
56      paso2.append(h**2)
57      err1.append(errf1)
58      err2.append(errf2)
59      err3.append(errf3)
60      err4.append(errf4)
61      err5.append(errf5)
62      # reducimos el tamano del paso para la siguiente aproximacion
63      h = h * 0.1
64
65  print("-----------------------")
66
67  t_fin = time.time() # registramos el final de la ejecucion (sin tener en
      cuenta la grafica)
68
```

```
# linea de orden 1
pyplot.loglog(paso, paso, color='blue', label='orden 1')
# linea de orden 2
pyplot.loglog(paso, paso2, color='navy', label='orden 2')
# errores de las aproximaciones
pyplot.loglog(paso, err1, color='purple', marker='o', label='formula 1.1')
pyplot.loglog(paso, err2, color='red', marker='*', label='formula 1.2')
pyplot.loglog(paso, err3, color='green', marker='v', label='formula 1.3')
pyplot.loglog(paso, err4, color='orange', marker='x', label='formula 1.4')
pyplot.loglog(paso, err5, color='yellow', marker='s', label='formula 1.5')

pyplot.xlabel('$h$')
pyplot.ylabel('$error$')
pyplot.legend(loc='lower right')
pyplot.show()

t_total = t_fin - t_ini
print("Tiempo de ejecucion:", "%.2f"%t_total, "segundos.")
print("-------------------------------------------------------")

```

I.2. Fórmulas para la segunda derivada

Código utilizado en la subsección 1.5.2

```
##########################################################
## comparatariva de formulas para la segunda derivada ##
##########################################################

from matplotlib import pyplot
import time
import math

t_ini = time.time() # registramos el inicio de la ejecucion

def f(x):        # definimos la funcion
    return(3 * x * math.exp(x) - math.cos(x))

def d2f(x):        # definimos la derivada segunda de la funcion
    return(3 * (2+x) * math.exp(x) + math.cos(x))

a = 3           # alpha
h = 1e-01       # tamano del primer paso
c = 1e-05       # cota minima para el paso

```

```
21  print("-----------------------------------------------------------------------")
22  print("f(x) = 3 * x * math.exp(x) - math.cos(x) en el punto  alpha =",a)
23  print("Aproximacion de su derivada segunda con diferentes formulas")
24  print("---h---  --error-f1-- --error-f2-- --error-f3-- --error-f4--")
25
26  # inicializamos los vectores para la representacion grafica posterior
27  paso = []
28  paso2 = []
29  paso3 = []
30  paso4 = []
31  err1 = []
32  err2 = []
33  err3 = []
34  err4 = []
35
36  # generamos un bucle para obtener diferentes aproximaciones con diferentes
      tamanos de h
37  while h >= c:
38      # aplicamos la formula 1.14 y calculamos el error
39      aproxf1 = (f(a-h) -2*f(a) + f(a+h)) / (h**2)
40      errf1 = abs(aproxf1 - d2f(a))
41      # aplicamos la formula 1.15 y calculamos el error
42      aproxf2 = (-f(a-2*h) +16*f(a-h) -30*f(a) +16*f(a+h) -f(a+2*h)) / (12*(h
    **2))
43      errf2 = abs(aproxf2 - d2f(a))
44      # aplicamos la formula 1.16 y calculamos el error
45      aproxf3 = (2*f(a) -5*f(a+h) +4*f(a+2*h) -f(a+3*h)) / (h**2)
46      errf3 = abs(aproxf3 - d2f(a))
47      # aplicamos la formula 1.17 y calculamos el error
48      aproxf4 = (-f(a-3*h) +4*f(a-2*h) -5*f(a-h) +2*f(a)) / (h**2)
49      errf4 = abs(aproxf4 - d2f(a))
50
51      print( "{:.1e}".format(h),"", "{:.6e}".format(errf1), "{:.6e}".format(
    errf2), "{:.6e}".format(errf3), "{:.6e}".format(errf4))
52
53      # actualizamos los vectores
54      paso.append(h)
55      paso2.append(h**2)
56      paso3.append(h**3)
57      paso4.append(h**4)
58      err1.append(errf1)
59      err2.append(errf2)
60      err3.append(errf3)
61      err4.append(errf4)
62      # reducimos el tamano del paso para la siguiente aproximacion
63      h = h * 0.5
64
```

```
print("-----------------------")

t_fin = time.time() # registramos el final de la ejecucion (sin tener en
  cuenta la grafica)

# lineas auxiliares
pyplot.loglog(paso, paso, color='#c2d8ff', label='orden 1')
pyplot.loglog(paso, paso2, color='#7db1ff', label='orden 2')
pyplot.loglog(paso, paso3, color='#4a90e2', label='orden 3')
pyplot.loglog(paso, paso4, color='#1f77b4', label='orden 4')
# errores de las aproximaciones
pyplot.loglog(paso, err1, color='purple', marker='o', label='formula 1.14')
pyplot.loglog(paso, err2, color='red', marker='*', label='formula 1.15')
pyplot.loglog(paso, err3, color='green', marker='v', label='formula 1.16')
pyplot.loglog(paso, err4, color='orange', marker='x', label='formula 1.17')

pyplot.xlabel('$h$')
pyplot.ylabel('$error$')
pyplot.legend(loc='lower right')
pyplot.show()

t_total = t_fin - t_ini
print("Tiempo de ejecucion:", "%.2f"%t_total, "segundos.")
print("-------------------------------------------------------")

```

I.3. Método de Newton con derivación numérica

Código utilizado en la subsección 3.1.1

```
###################################
##    metodo de Newton con DN    ##
###################################

import time
import numpy as np

t_ini = time.time() # registramos el inicio de la ejecucion

x0 = 5              # aproximacion inicial
tol = 1e-20         # tolerancia
maxit = 100         # num max de iteraciones
r = 1               # raiz conocida

# definimos la funcion
```

```
16  def f(x):
17      return(5*x**2 - 3*x - 2)
18
19  # definimos la derivada usando DN
20  h = 1e-5            # paso para aproximar la derivada
21  def df(x):
22      return((f(x+h) - f(x-h)) / (2*h) )  # formula 1.3
23
24  print("------------------------------------------------------------")
25  print("Buscamos una raiz de f(x) = 5*x**2 - 3*x - 2 proxima a x0 =", x0)
26  print("utilizaremos el metodo de Newton con una formula centrada para la
      derivada")
27  print("con un maximo de", maxit, "iteraciones y una tolerancia de", "{:.1e}".
      format(tol))
28  print("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -")
29
30  # algoritmo del metodo de Newton
31  xn = x0
32  it = 0
33  print("i _aprox_       _error_       _orden_")
34  print(it, "{:.6e}".format(xn))
35  while abs(f(xn)) > tol and it <= maxit:
36      xn1 = xn - f(xn) / df(xn)
37      it = it + 1
38      if it == 1:          # en la primera it no podemos estimar orden
39          print(it, "{:.6e}".format(xn1), "{:.6e}".format(abs(xn1-r)))
40      else :
41          print(it, "{:.6e}".format(xn1), "{:.6e}".format(abs(xn1-r)), "%.4f"%(
      np.log(abs(xn1-r))/np.log(err)))
42      xn = xn1
43      err = abs(xn1-r)
44
45  if it == maxit:
46      print("Se han alcanzado", maxit, "iteraciones.")
47  else:
48      print("Tras", it , "iteraciones obtenemos:")
49      print("   - la aproximacion de la raiz es: a =", "%.5f" % xn)
50      print("   - se comprueba que: f(a) = ", "{:.6e}".format(f(xn)))
51
52  t_fin = time.time() # registramos el final de la ejecucion
53  t_total = t_fin - t_ini
54  print("Tiempo de ejecucion:", "%.6f"%t_total, "segundos.")
55  print("------------------------------------------------------------")
56
```

I.4. Método de Newton con derivación automática

Código utilizado en la subsección 3.1.2

```
##################################
##    metodo de Newton con DA    ##
##################################

import time
import numpy as np

t_ini = time.time() # registramos el inicio de la ejecucion

# definimos valder solo con las operaciones basicas
class valder:
    def __init__(self, val, der=1):     # inicializacion del objeto
        self.val = val
        self.der = der
        def __repr__(self):                 # formato de visualizacion
        return f"valder(val={self.val}, der={self.der})"
    def __add__(self, other):           # sobrecargamos la suma
        if isinstance(other, valder):
            return valder(self.val + other.val, self.der + other.der)
        else:
            return valder(self.val + other, self.der)
    def __sub__(self, other):           # sobrecargamos la resta
        if isinstance(other, valder):
            return valder(self.val - other.val, self.der - other.der)
        else:
            return valder(self.val - other, self.der)
    def __mul__(self, other):           # sobrecargamos la multiplicacion
        if isinstance(other, valder):
            return valder(self.val * other.val, self.val * other.der + self.
  der * other.val)
        else:
            return valder(self.val * other, self.der * other)
    def __truediv__(self, other):       # sobrecargamos la division
        if isinstance(other, valder):
            return valder(self.val / other.val, (self.der * other.val - self.
  val * other.der) / (other.val ** 2))
        else:
            return valder(self.val / other, self.der / other)

x0 = valder(5)      # aproximacion inicial
tol = 1e-20         # tolerancia
maxit = 100         # num max de iteraciones
r = 1               # raiz conocida
```

```
42
43  # definimos la funcion
44  def f(x):
45      return(x*x*5 - x*3 - 2)
46
47  print("-----------------------------------------------------------")
48  print("Buscamos una raiz de f(x) = 5*x**2 - 3*x - 2 proxima a x0 =", x0)
49  print("utilizaremos el metodo de Newton con DA para la derivada")
50  print("con un maximo de", maxit, "iteraciones y una tolerancia de", "{:.1e}".
      format(tol))
51  print("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -")
52
53  # algoritmo del metodo de Newton
54  xn = x0
55  it = 0
56  print("i _aprox_       _error_       _orden_")
57  print(it, "{:.6e}".format(xn.val))
58  while abs(f(xn).val) > tol and it <= maxit:
59      fx = f(xn)
60      xn1 = xn.val - fx.val / fx.der
61      it = it + 1
62      if it == 1:          # en la primera it no podemos estimar orden
63          print(it, "{:.6e}".format(xn1), "{:.6e}".format(abs(xn1-r)))
64      else:
65          print(it, "{:.6e}".format(xn1), "{:.6e}".format(abs(xn1-r)), "%.4f"%(
      np.log(abs(xn1-r))/np.log(err)))
66      xn.val = xn1
67      err = abs(xn1-r)
68
69  if it == maxit:
70      print("Se han alcanzado", maxit, "iteraciones.")
71  else:
72      print("Tras", it , "iteraciones obtenemos:")
73      print("   - la aproximacion de la raiz es: a =", "%.5f" % xn.val)
74      print("   - se comprueba que: f(a) = ", "{:.6e}".format(f(xn).val))
75
76  t_fin = time.time() # registramos el final de la ejecucion
77  t_total = t_fin - t_ini
78  print("Tiempo de ejecucion:", "%.6f"%t_total, "segundos.")
79  print("-----------------------------------------------------------")
```

Bibliografía

[1] ISAACSON, E.; KELLER, H.B. (1996). *Analysis of Numerical Methods*, Dover Publications.

[2] CIARLET, P.G. (1999). *Introducción á Análise Numérica Matricial e á Optimización (Traducción de Paula Ballesteros Andrade e Xosé Antonio Rubal López)*, Servicio de Publicación da USC.

[3] VIAÑO, J.M.; BURGUERA, M. (2000). *Lecciones de Métodos Numéricos 3.- Interpolación*, Tórculo edicións.

[4] BURDEN R.L.; FAIRES J.D. (2010) *Numerical Analysis (Ninth Edition)*. Brooks/Cole Cengage Learning.

[5] LEVEQUE, R.J. (2007). *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, SIAM.

[6] NEIDINGER, R.D. (2010). *Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming*, SIAM Review (Vol. 52, No. 3, pp. 545–563).

[7] GRIEWANK, A.; WALTHER, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd ed.*, SIAM.

[8] GONZÁLEZ DUQUE, R. (2008). *Python para todos*, licencia Creative Commons (mundogeek.net/tutorial-python).