USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
FACULTADE DE MATEMÁTICAS

Trabajo Fin de Grado

Diferenciación Automática mediante grafos computacionales

Lucía Expósito García

2024/2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRADO DE MATEMÁTICAS

Trabajo Fin de Grado

Diferenciacion Automática mediante grafos computacionales

Lucía Expósito García

Julio, 2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Coñecemento: Matemática Aplicada
Título: Diferenciación Automática mediante grafos computacionales
Breve descrición do contido
El término diferenciación automática engloba diversas técnicas que usan la representación computacional de una función matemática para obtener el valor de sus derivadas. Los principales enfoques, los métodos progresivo y retrógrado (en inglés, forward y backward), se basan en mayor o menor medida en el concepto de grafo computacional. Para ello se buscará la forma de implementar la creación del grafo computacional de una función en MATLAB. Posteriormente se utilizará para el objetivo de la diferenciación automática, indagando sus posibles extensiones a varios grados de derivación y/o varias variables independientes.
Recomendacións
Outras observacións

Índice

Resumen

La Diferenciación Automática es una técnica de derivación que combina las ideas de la diferenciación simbólica y de la diferenciación numérica con el objetivo de evaluar de manera exacta derivadas de una función en un punto dado. Existen dos enfoques principales para su aplicación: el método progresivo y el método retrógrado. En este trabajo, nos centraremos principalmente en el segundo, aunque también se tratará el método progresivo, pero de manera introductoria, para contextualizar este nuevo tipo de diferenciación. Se estudiará el beneficio de utilizar el enfoque retrógrado en el cálculo de gradientes de funciones que dependan de multitud de variables, la extensión del método a varias variables independientes y a diferentes grados de derivación, así como la creación del grafo computacional necesario para proporcionar un código que permita su implementación en MATLAB. Por último, se explorarán sus aplicaciones prácticas como, por ejemplo, el cálculo de gradientes en los procesos de optimización de redes neuronales.

Abstract

Automatic Differentiation is a differentiation technique that combines the ideas of symbolic and numerical differentiation with the goal of accurately evaluating derivatives of functions at given points. Two main approaches are typically used: forward mode and reverse mode. Although both will be discussed, the primary focus of this work is on the reverse mode, due to its advantages in scenarios involving functions with many input variables. The study explores the benefits of reverse mode for gradient computation, its extension to multiple independent variables and higher-order derivatives, and the construction of the computational graph required for its implementation in MATLAB. Finally, practical applications are examined, particularly in the context of gradient-based optimization processes, such as those used in training neural networks.

Introducción

La Diferenciación Automática es una técnica de diferenciación que utiliza las reglas de derivación analíticas para hallar la derivada de una función en un punto de forma exacta. Esta técnica se diferencia de la derivación simbólica en que no es necesario calcular la expresión explícita de la derivada. Los principales enfoques, el método progresivo y el método regresivo, serán explicados en las siguientes páginas, pero el objetivo principal de este trabajo es desarrollar una implementación para el método regresivo.

En el primer capítulo se describen las ideas fundamentales de los dos enfoques anteriores. Comenzaremos analizando los métodos para una función escalar de una variable y, posteriormente, indagaremos en sus posibles extensiones a varias variables independientes y/o funciones vectoriales.

El segundo capítulo se adentra en la implementación de la Diferenciación Automática. Los enfoques progresivo y regresivo se basan en mayor o menor medida en el concepto de grafo computacional, que será de gran utilidad a lo largo de este capítulo.

En la primera sección, introducimos la implementación del método progresivo, comentando brevemente sus nociones más importantes. En esta parte no será necesario construir un grafo computacional, pero veremos que está incluido implícitamente.

A continuación, en la segunda sección, se desarrolla la implementación del método regresivo. La particularidad de este método es que realiza las operaciones en dos sentidos diferentes. Se dividirá en tres partes, las dos primeras avanzando en uno de los sentidos y la última en el otro. Por este motivo, será necesario construir un grafo dirigido que le diga al método en qué orden debe realizar las operaciones. Una gran parte de esta sección se centrará en dicha construcción. En primer lugar, se realizará la creación de la estructura de relaciones entre los términos de la función. Después se utilizará esta estructura para construir el grafo dirigido que necesitamos. Por último, ilustraremos cómo conseguir que el grafo computacional guíe los pasos del método en cada uno de los sentidos en los que avanza.

En el Capítulo 3, se examinará la eficiencia de los métodos de diferenciación simbólico, au-

tomático progresivo y automático regresivo. Para ello se seleccionarán varias funciones que se consideran representativas del conjunto de funciones a las que es posible aplicar cada uno de los métodos y se compararán los tiempos de ejecución para cada uno de ellos.
Finalmente, se estudiarán las repercusiones de estos resultados en las aplicaciones de las derivadas, gradientes o matrices jacobianas.

Capítulo 1

Problema matemático y métodos de resolución

La evaluación de la derivada de una función suficientemente regular en un punto arbitrario de su dominio es un problema matemático que ha sido ampliamente estudiado.
Supongamos que se quiere hallar el valor de la derivada en el punto $x_0 = 0$ de la función $f : \mathbb{R} \longrightarrow \mathbb{R}$, $x \longrightarrow f(x) = x^2 \cdot \sin(x)$.

Una posible forma de resolver el problema es calcular la expresión analítica de la derivada manualmente mediante las reglas de derivación habituales de la suma, la multiplicación y la cadena. En este caso se llegaría a la expresión

$$f'(x) = 2 \cdot x \cdot \sin(x) + x^2 \cdot \cos(x).$$

A continuación, se evalúa para $x_0 = 0$, de donde se obtiene el valor buscado mediante la serie de operaciones

$$f'(0) = 2 \cdot 0 \cdot \sin(0) + 0^2 \cdot \cos(0) = 2 \cdot 0 \cdot 0 + 0 \cdot 1 = 0 + 0 = 0.$$

Sin embargo, este método, aunque exacto, puede ser largo y propenso a errores humanos cuando se trabaja con funciones cuya expresión es complicada.

La derivación simbólica permite la obtención de una expresión analítica que se evalúa en el punto de interés. En este caso, se genera computacionalmente una expresión explícita para la derivada. Esto resuelve el problema del error humano, pero requiere un coste computacional elevado y es posible que el resultado final sea una expresión demasiado grande o difícil de interpretar.

Un tercer planteamiento podría ser utilizar una fórmula de aproximación numérica. Por ejemplo, seleccionamos la fórmula

$$f'(x) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h)$$

con paso h y calculamos el valor de la derivada en $x_0 = 0$,

$$f'(0) \approx \frac{f(0 + 0.1) - f(0)}{0.1} = \frac{(0.1)^2 \cdot \sin(0.1) - 0^2 \cdot \sin(0)}{0.1} = 0.00998.$$

De esta forma no necesitamos una expresión explícita para la derivada y se puede calcular el valor de la derivada de cualquier función suficientemente regular utilizando la misma fórmula, que sólo involucra una suma y una división, aparte de las operaciones involucradas en el cálculo de la función.

Si bien es cierto que hemos reducido los cálculos y evitado el uso de expresiones grandes, solucionando los problemas que presentaban los anteriores métodos, no se puede ignorar el hecho de que el valor obtenido no es exacto, lo cual conlleva a una pérdida de precisión. Se podría reducir el paso h para obtener valores cada vez más cercanos al real, pero no se puede reducir este h sin tener en cuenta los errores de redondeo de la aritmética en punto flotante, que terminarían por aumentar el error de aproximación hasta producir valores completamente distintos al verdadero para h muy cercanos a 0.

Un método que combina las ideas de la diferenciación simbólica y de la derivación numérica es lo que se conoce como Diferenciación Automática. La Diferenciación Automática es un método alternativo que se sirve de las reglas de diferenciación analíticas para aprovechar las ventajas del cálculo simbólico y utiliza valores previamente obtenidos para continuar los cálculos del valor de la derivada. El resultado es una evaluación exacta sin necesidad de utilizar expresiones analíticas.

En lo que sigue, las ideas acerca de la Diferenciación Automática se han extraído de [4] y [5]

1.1. Diferenciación Automática progresiva

Calcular la expresión para la derivada es el resultado de realizar operaciones concretas sobre ciertas partes que componen la función f, según las reglas de derivación. Así, para funciones $v, w : D \subseteq \mathbb{R} \longrightarrow \mathbb{R}$, $v, w \in \mathcal{C}^1(\mathbb{R})$, tenemos las siguientes reglas:

- $(k)' = 0$, para $k \in \mathbb{R}$.
- $(x)' = 1$.

- $(x^k)' = k \cdot x^{k-1}$, para $k \in \mathbb{R}$.
- $(v \pm w)'(x) = v'(x) \pm w'(x)$.
- $(v \cdot w)'(x) = v'(x) \cdot w(x) + v(x) \cdot w'(x)$.
- $(\frac{v}{w})'(x) = \frac{v'(x) \cdot w(x) - v(x) \cdot w'(x)}{(w(x))^2}$, para un punto en el que no se anule w.
- Regla de la cadena: $(v \circ w)'(x) = v'(w(x)) \cdot w'(x)$, si $v : D \subseteq \mathbb{R} \longrightarrow \mathbb{R}$, $w : G \subseteq \mathbb{R} \longrightarrow \mathbb{R}$ y $v(D) \subseteq G$.

La expresión de la derivada para las funciones elementales y estas siete reglas permiten hallar la expresión analítica para la derivada de casi cualquier función suficientemente regular de variable real y serán fundamentales en la construcción del método de la Diferenciación Automática.

Obsérvense de nuevo la función $f(x) = x^2 \cdot \sin(x)$ y su derivada $f'(x) = 2 \cdot x \cdot \sin(x) + x^2 \cdot \cos(x)$. Para evaluar la función f en x_0, lo primero que se necesita es saber el valor del punto arbitrario y sustituirlo en la expresión dada, es decir, realizamos, de abajo a arriba, el procedimiento indicado por el árbol de la Figura 1.1, de modo que para calcular el valor $f(x_0)$ con $x_0 = 0$ se realizan las

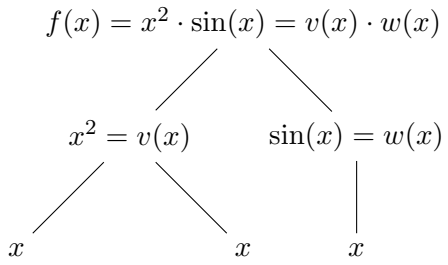


Figura 1.1: Árbol asociado a la función $f(x) = x^2 \cdot \sin(x)$.

operaciones $v(0) = 0^2 = 0$, $w(0) = \sin(0) = 0$ y $f(0) = v(0) \cdot w(0) = 0 \cdot 0 = 0$.

El sentido de "abajo hacia arriba" se considera el sentido progresivo porque avanzamos desde las partes más elementales con valor conocido hacia el valor de la expresión completa. También se puede ver como realizar las operaciones "de dentro hacia fuera" de la expresión de la función. Así, los valores $v(0)$ y $w(0)$, que ya se habían obtenido previamente, han sido los que han permitido calcular finalmente el valor $f(0)$.

Podemos pensar que es posible proceder de manera análoga para calcular la derivada $f'(0)$ a la vez que se obtiene $f(0)$.

Si se calculan las derivadas de v y w en x_0 a la vez que calculamos $v(x_0)$ y $w(x_0)$ en el sentido progresivo anterior a través del árbol, se obtienen los valores $[g(0) = 0, g'(0) = 1]$ para $g(x) = x$,

$[v(0) = 0, v'(0) = 2 \cdot 0 = 0]$, $[w(0) = 0, w'(0) = \cos(0) = 1]$ y $[f(0) = v(0) \cdot w(0) = 0, f'(0) = v'(0) \cdot w(0) + v(0) \cdot w'(0) = 0 \cdot 0 + 0 \cdot 1 = 0]$.

La notación $[f(x_0), f'(x_0)]$ denota la 2-tupla compuesta por el valor de una función f y el de su derivada en un punto arbitrario. Estos valores para la función f y sus funciones intermedias se recogen en la siguiente tabla.

Función	Valor función	Valor derivada
$g(x) = x$	0	1
$v(x) = x^2$	0	0
$w(x) = \sin(x)$	0	1
$f(x) = v(x) \cdot w(x)$	0	0

Además, para ilustrar los pasos del cálculo de ambas evaluaciones, se incluye el árbol que aparece a continuación. Recuérdese que los cálculos avanzan en sentido progresivo.

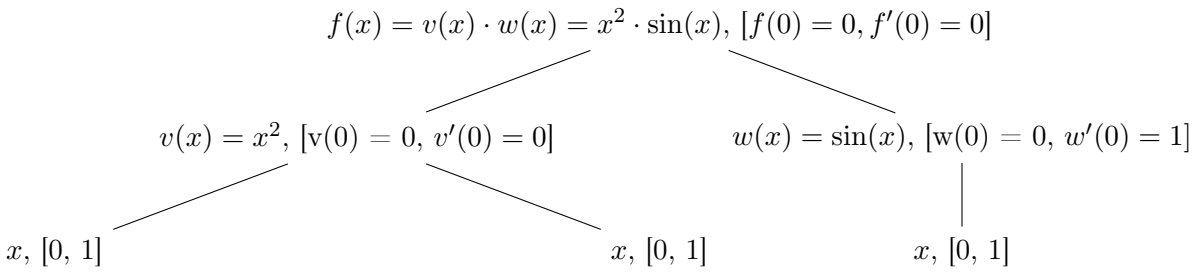


Figura 1.2: Árbol asociado a la función $f(x) = x^2 \cdot \sin(x)$ con los valores de la función y derivada en cada nodo.

Esta vez el resultado es exacto y no hay necesidad de generar una expresión analítica explícita, aunque se ha incluido por claridad en la explicación. A este método se le llama Diferenciación Automática progresiva y se caracteriza por realizar una evaluación de las derivadas intermedias de la función hasta hallar el valor de la derivada de la función completa.

Una observación importante a tener en cuenta y que será útil más adelante es que el árbol que ejemplifica la evaluación de las derivadas intermedias ya está implementado implícitamente en los sistemas de cómputo numérico y no hace falta construirlo.

1.1.1. Caso multivariable

A continuación se estudiará la extensión de la Diferenciación Automática progresiva a funciones que dependan de varias variables. A lo largo de esta subsección se trabajará con funciones $f : D \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$ suficientemente regulares en su dominio y un punto arbitrario $x_0 \in \mathring{D}$.

Definición 1.1. Sean $f : D \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$, $x_0 \in \mathring{D}$ e y un punto arbitrario de $\mathbb{R}^n$. La derivada de f en x_0 con respecto a y se define como el límite

$$\lim_{h \to 0} \frac{f(x_0 + h \cdot y) - f(x_0)}{h}$$

cuando éste existe. Si y es un vector unitario, el límite anterior se llama derivada direccional de f en x_0 en la dirección de y. Si, además, el vector y es el k-ésimo vector coordenado, la derivada direccional se denomina derivada parcial respecto de e_k. Utilizaremos la notación $\frac{\partial f}{\partial x_k}$ para referirnos a la derivada parcial de f respecto del k-ésimo vector coordenado [1].

Definición 1.2. Se define el gradiente de f y se denota por ∇f como $\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})^t$ [1].

Si tomamos $v : D \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ y $w : G \subseteq v(D) \subseteq \mathbb{R}^m \longrightarrow \mathbb{R}$, entonces se puede extender la regla de la cadena de la siguiente forma :

$$\nabla_x v(w(x)) = \sum_{i=1}^{m} \frac{\partial v}{\partial w_i} \cdot \nabla w_i(x).$$

La Diferenciación Automática se puede adaptar también a estas situaciones. Considerando la derivada direccional como un operador diferencial, podemos extender las operaciones con escalares para el caso univariable a operaciones con vectores y proceder de la misma manera, sin más que sustituir el gradiente por la derivada.

Consideremos la función $f(x, y) = \sin(x) \cdot e^{x \cdot y}$, cuyo gradiente es $\nabla f = (\cos(x) \cdot e^{x \cdot y} + \sin(x) \cdot e^{x \cdot y} \cdot y, e^{x \cdot y} \cdot x \cdot \sin(x))^t$, y el punto $(x_0, y_0) = (\frac{\pi}{2}, \frac{\pi}{2})$. Realizando el mismo procedimiento, pero teniendo en cuenta que en vez de una derivada ahora hay que operar con un vector gradiente de modo que $\nabla f(\frac{\pi}{2}, \frac{\pi}{2}) = \nabla v(\frac{\pi}{2}, \frac{\pi}{2}) \cdot w(\frac{\pi}{2}, \frac{\pi}{2}) + v(\frac{\pi}{2}, \frac{\pi}{2}) \cdot \nabla(\frac{\pi}{2}, \frac{\pi}{2}) = (e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2}, e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2})$, llegamos al valor del gradiente de f en (x_0, y_0).

Función	Valor función	Valor gradiente
$g(x, y) = x \cdot y$	$\frac{\pi^2}{4}$	$(\frac{\pi}{2}, \frac{\pi}{2})$
$v(x, y) = \sin(x)$	1	$(\cos(\frac{\pi}{2}), 0) = (0, 0)$
$w(x, y) = e^{g(x,y)} = e^{x \cdot y}$	$e^{g(\frac{\pi}{2}, \frac{\pi}{2})} = e^{\frac{\pi^2}{4}}$	$(e^{g(\frac{\pi}{2}, \frac{\pi}{2})} \cdot \frac{\pi}{2}, e^{g(\frac{\pi}{2}, \frac{\pi}{2})} \cdot \frac{\pi}{2}) = (e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2}, e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2})$
$f(x) = v(x) \cdot w(x)$	$v(\frac{\pi}{2}, \frac{\pi}{2}) \cdot w(\frac{\pi}{2}, \frac{\pi}{2}) = e^{\frac{\pi^2}{4}}$	$(0, 0) \cdot e^{x \cdot y} + 1 \cdot (e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2}, e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2}) = (e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2}, e^{\frac{\pi^2}{4}} \cdot \frac{\pi}{2})$

Los valores de las funciones y de sus gradientes se han incluido al lado de los nodos correspondientes en el árbol que aparece a continuación.

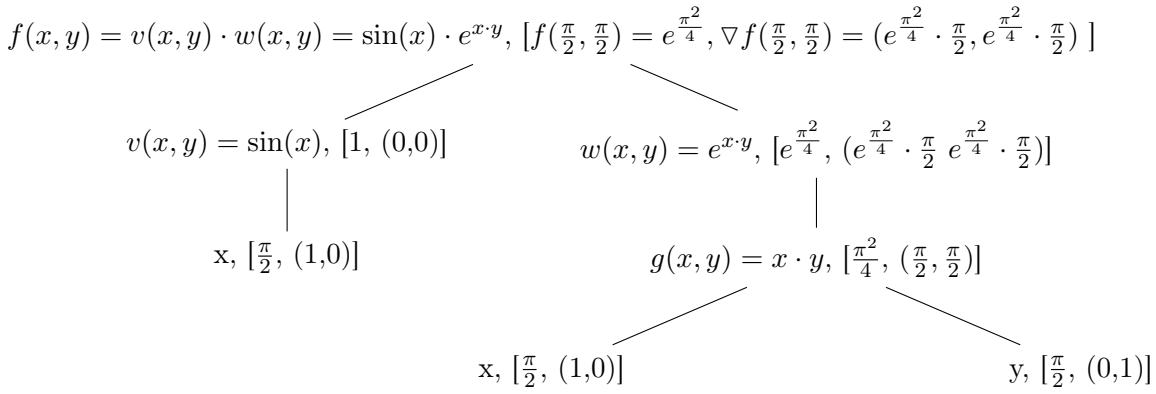


Figura 1.3: Árbol asociado a la función $f(x,y) = \sin(x) \cdot e^{x \cdot y}$ con los valores de la función y derivada en cada nodo.

La extensión a funciones vectoriales también es posible. Considérese una función vectorial f de $\mathbb{R}^n$ en $\mathbb{R}^m$ con $n, m \in \mathbb{N}$ suficientemente regular en un punto x_0.

Definición 1.3. Se define la matriz jacobiana de una función f suficientemente regular en un punto x_0 como la matriz asociada a la aplicación diferencial de f en x_0. La expresión de la matriz jacobiana para una función viene dada por

$$J = \left(\nabla f_1 \ ... \ \nabla f_m \right)^t.$$

En este caso, se puede calcular la jacobiana de una función aplicando la Diferenciación Automática progresiva a cada una de sus componentes f_i, $i \in \{1, ..., m\}$.

1.1.2. Derivación de orden superior

La forma más intuitiva de extender el método a derivadas de orden superior es volver a aplicar la Diferenciación Automática progresiva después de haber evaluado la primera derivada, repitiendo el paso todas las veces que fuese necesario. No obstante, esta idea no es nada eficiente, ya que acumula demasiados valores redundantes y genera árboles muy extensos.

Una propuesta más eficiente si $f : \mathbb{R} \longrightarrow \mathbb{R}$ es una función analítica y suficientemente regular, es utilizar la expansión de Taylor de f y truncar la serie en el orden de derivada que se desee. De esta manera se calcula $f^{n)}$ en $x_0 = a$ como sigue:

$$f(x) = \sum_{k=0}^{n-1} \frac{f^{k)}(a)}{k!} \cdot (x-a)^k$$

$$f^{n)}(a) = \left(f(x) - \sum_{k=0}^{n-1} \frac{f^{k)}(a)}{k!} \cdot (x-a)^k \right) \cdot \frac{k!}{(x-a)^n}.$$

Si guardamos los valores de los coeficientes de la serie en $f_k = \frac{f^{k)}(a)}{k!}$ entonces

$$f^{n)}(a) = \left(f(x) - \sum_{k=0}^{n-1} f_k \right) \cdot f_n^{-1}.$$

Utilizando las propiedades de la suma y producto de las series de Taylor podemos calcular los coeficientes intermedios de cualquier función de la que dependa f y llegar a la evaluación final de la derivada.

Para terminar esta sección, aclaramos que es posible extender esta idea a derivadas de orden superior de funciones de varias variables, pero no lo trataremos en este trabajo, pues no es nuestro objetivo realizar un estudio extensivo de la Diferenciación Automática progresiva. Más información sobre este tema se puede encontrar en [6]. Como aún no ha sido publicado en el repositorio institucional Minerva, citaremos el artículo [4] en el que se basa dicho trabajo.

1.2. Diferenciación Automática regresiva

Dada una función con la misma regularidad que en la sección anterior y su árbol asociado, la Derivación Automática regresiva calcula su derivada operando de forma progresiva para calcular las derivadas parciales de cada nodo respecto a las funciones definidas en un nivel inferior del árbol asociado a él y de forma regresiva para calcular las derivadas parciales de la función inicial respecto de la variable asociada en cada nodo $\frac{\partial f}{\partial x_i}$, $i \in 1, ..., N$. Estas derivadas parciales son productos de las derivadas calculadas en el paso anterior, siguiendo la expresión que proporciona la regla de la cadena para el gradiente de una función escalar multivariable.

A modo de ejemplo, sean $f(x, y, z) = (x \cdot z) \cdot \sin(x \cdot y)$ y su árbol asociado el que aparece a continuación.

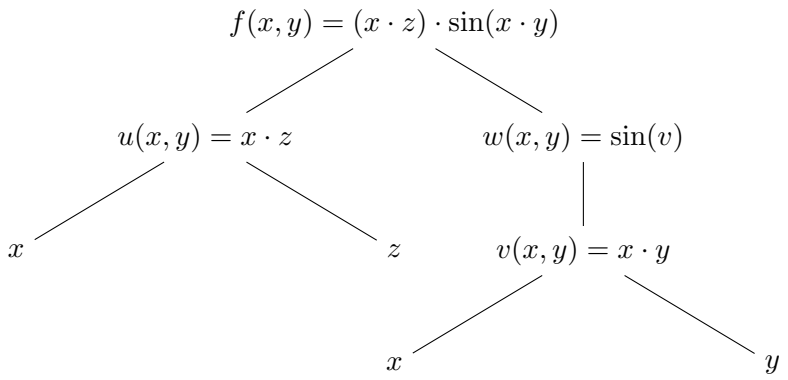


Figura 1.4: Árbol asociado a la función $f(x, y, z) = (x \cdot z) \cdot \sin(x \cdot y)$

En primer lugar se realizan los cálculos que corresponden a la parte progresiva del método.

- z = c, x = a, y = b
- $u_x = \frac{\partial u}{\partial x} = c$, $u_z = \frac{\partial u}{\partial z} = a$
- $v_x = \frac{\partial v}{\partial x} = b$, $v_y = \frac{\partial v}{\partial y} = a$
- $w_v = \frac{\partial w}{\partial v} = \cos(v)$
- $h_u = \frac{\partial h}{\partial u} = w$, $h_w = \frac{\partial h}{\partial w} = u$

A continuación se procede a realizar un camino inverso calculando las $\frac{\partial f}{\partial x_i}$ para cada nodo del árbol. Este paso es la razón por la que a este método se lo denomina método regresivo.

$$\overline{f} := \frac{\partial f}{\partial f} = 1 \tag{1.1}$$

$$\overline{u} := \frac{\partial f}{\partial u} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial u} = \overline{f} \cdot f_u \tag{1.2}$$

$$\overline{w} := \frac{\partial f}{\partial w} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial w} = \overline{f} \cdot f_w \tag{1.3}$$

$$\overline{v} := \frac{\partial f}{\partial v} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial v} = \overline{w} \cdot w_v \tag{1.4}$$

$$\overline{x} := \frac{\partial f}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x} = \overline{u} \cdot u_x + \overline{v} \cdot v_x \tag{1.5}$$

Para el punto $x_0 = (1, 0, 1)$ los valores serían:

1. Primero, de abajo hacia arriba:
   - $x = 1$, $y = 0$, $z = 1$
   - $u_x = \frac{\partial u}{\partial x} = 1$, $u_z = \frac{\partial u}{\partial z} = 1$

- $v_x = \frac{\partial v}{\partial x} = 0$, $v_y = \frac{\partial v}{\partial y} = 1$
- $w_v = \frac{\partial w}{\partial v} = \cos(v) = 1$
- $h_u = \frac{\partial h}{\partial u} = w = 0$, $h_w = \frac{\partial h}{\partial w} = u = 1$

2. Después de arriba hacia abajo (regresivo):

$$\overline{f} := \frac{\partial f}{\partial f} = 1 \tag{1.6}$$

$$\overline{u} := \frac{\partial f}{\partial u} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial u} = \overline{f} \cdot f_u = 0 \tag{1.7}$$

$$\overline{w} := \frac{\partial f}{\partial w} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial w} = \overline{f} \cdot f_w = 1 \tag{1.8}$$

$$\overline{v} := \frac{\partial f}{\partial v} = \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial v} = \overline{w} \cdot w_v = 1 \cdot 1 = 1 \tag{1.9}$$

$$\overline{x} := \frac{\partial f}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x} = \overline{u} \cdot u_x + \overline{v} \cdot v_x = 0 \cdot 1 + 1 \cdot 0 = 0 \tag{1.10}$$

$$\overline{y} := \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x} = \overline{v} \cdot v_y = 1 \cdot 1 = 1 \tag{1.11}$$

$$\overline{z} := \frac{\partial f}{\partial z} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial z} = \overline{u} \cdot u_z = 0 \cdot 1 = 0 \tag{1.12}$$

La evaluación de cada una de las derivadas parciales de f es

- $\frac{\partial f(1,0,1)}{\partial x} = z \cdot \sin(x \cdot y) + (x \cdot z) \cdot \cos(x \cdot y) \cdot y_{/(1,0,1)} = 0$
- $\frac{\partial f(1,0,1)}{\partial y} = (x \cdot z) \cdot \cos(x \cdot y) \cdot x_{/(1,0,1)} = \cos(0) = 1$
- $\frac{\partial f(1,0,1)}{\partial z} = x \cdot \sin(x \cdot y)_{/(1,0,1)} = 0.$

El método regresivo proporciona el valor correcto del gradiente $\nabla f = (0, 1, 0)$ sin calcular la expresión analítica de cada una de las componentes del vector gradiente. Nosotros las hemos incluido para comprobar que, efectivamente, los resultados son correctos, pero, igual que al aplicar la Diferenciación Automática progresiva, en ningún paso es necesario calcularlas.

Este método resulta más eficiente cuando se trabaja con funciones que dependen de varias variables, ya que no es necesario obtener el gradiente de cada nodo presente en el grafo y se puede acceder a la derivada parcial respecto de una variable independiente sin calcular las demás.
Si se hubiese utilizado el método progresivo, los cálculos pertinentes para $x_0 = (a, b, c) = (1, 0, 1)$ habrían sido:

- $x = a, y = b, z = c$
- $u = a \cdot c$, $v = a \cdot b$, $w = \sin(v)$, $f = u \cdot v$

- $\nabla x = [1, 0, 0]$
- $\nabla y = [0, 1, 0]$
- $\nabla z = [0, 0, 1]$
- $\nabla u = c \cdot [1, 0, 0] + a \cdot [0, 0, 1] = [c, 0, a]$
- $\nabla v = b \cdot [1, 0, 0] + a \cdot [0, 1, 0] = [b, a, 0]$
- $\nabla w = \cos(v) \cdot [b, a, 0] = [b \cdot \cos(v), a \cdot \cos(v), 0]$
- $\nabla f = w \cdot [c, 0, a] + u \cdot [b \cdot \cos(v), a \cdot \cos(v), 0]$

Es obvio que resulta un procedimiento más costoso ya que para calcular las parciales con respecto a cada variable, los cálculos no se pueden aprovechar. En efecto, ya difieren desde el inicio.

Con el procedimiento regresivo, son los últimos pasos del cálculo que difieren. Las variables $\overline{u}$, $\overline{v}$,$\overline{w}$ no se tienen que recalcular para evaluar $\overline{x}$, $\overline{y}$ y $\overline{z}$.

Capítulo 2

Implementación

En este capítulo se tratará fundamentalmente la implementación en MATLAB del método regresivo para funciones escalares en una variable real. Las ideas acerca de la implementación de este método se han extraído de [4]. La implementación del método progresivo será explicada con el objetivo de introducir los conceptos básicos que se utilizarán en la construcción del código de la Diferenciación Automática regresiva. Para ello se resumen las explicaciones y códigos desarrollados en [6], que el lector interesado puede consultar para más detalles. De nuevo, es preciso aclarar que este documento todavía no se ha publicado en el repositorio institucional Minerva, de modo que en lo que sigue citaremos el artículo [4] en su lugar.

2.1. Implementación del método progresivo

La implementación para el método de la Diferenciación Automática progresiva se puede realizar de varias maneras, pero en este trabajo se explicará la implementación realizada en [4] que utiliza lo que se conoce como programación orientada a objetos. El uso de la programación orientada a objetos es una herramienta para la implementación de este método, pero no es fundamental para su definición.

En la programación orientada a objetos, a diferencia de en la programación estructurada que se imparte en el grado, los datos y los procedimientos no están separados. La base de este paradigma de programación está en la definición de objetos o estructuras que contienen información en campos. Estos objetos poseen una serie de métodos asociados que pueden modificar los valores contenidos en sus campos. Por ejemplo, se puede definir en MATLAB el objeto `flores` con la información del tipo de flor y la cantidad.

```
flores = struct()
flores.rosa = 26
flores.geranio = 17
```

El objeto flores devuelve los campos que contiene la estructura. Se puede ver en la Figura 2.1

```
>> flores

flores =

  struct with fields:

      rosa: 26
    geranio: 17
```

Figura 2.1: Información de la estructura `flores`.

Una de las aplicaciones de la programación orientada a objetos es que permite modificar funciones definidas previamente, como la función que realiza la suma o la evaluación de una función elemental, para que actúen sobre nuevos tipos de datos. Esta es la idea fundamental de la implementación del método progresivo en [4].

Las funciones elementales y los operadores de la suma, resta, multiplicación, división y exponenciación operan sobre datos reales. No obstante, utilizando la programación orientada a objetos, estas funciones se pueden modificar definiendo un objeto que actúe sobre un nuevo tipo de dato. A este procedimiento se lo denomina sobrecarga de operadores. En el caso de la Derivación Automática progresiva, este nuevo dato, al que se le llamará `valder`, contiene el valor de una función y el de su derivada en un punto. Lo que buscamos es sobrecargar las funciones elementales y los operadores anteriores para que actúen sobre `valder` devolviendo un objeto del mismo tipo.

Conocido el valor de una función f cualquiera y de su derivada en un punto x_0, podemos construir el objeto `valder` utilizando como argumentos de tipo double $a = f(x_0)$ y $b = f'(x_0)$ de la siguiente forma.

```
function obj = valder(a,b)
    if nargin == 0
        obj.val == [];
        obj.der == [];

    elseif nargin == 1
```

```
        obj.val = a;
        obj.der = 0;

    else
        obj.val = a;
        obj.der = b;
    end
end
```

Por ejemplo, `x = valder(9,6)` representa el valor de $f(x) = x^2$ y de su derivada en $x_0 = 3$. La clave en la implementación es sobrecargar las funciones elementales y los operadores básicos $+, -, \times, \div, \wedge$ para que actúen sobre un objeto `valder` como el que hemos definido devolviendo un objeto del mismo tipo. De esta manera al evaluar la expresión de una función arbitraria obtendremos directamente su valor y el de su derivada en un punto dado.

Obsérvese que sería posible empezar el esquema por $f(x) = x^2$ para una función $h(x) = x^2 \cdot \sin(x^2)$ en vez de por $f(x) = x$. Aún así, comenzar por la función identidad es más eficiente, ya que $f'(x) = 1$ y $f(x_0) = x_0$, con lo que de esta forma es inmediato y sencillo inicializar el algoritmo, ya que solo se necesita conocer el punto en el que queremos evaluar la derivada.

Las funciones elementales se sobrecargan utilizando la expresión de su derivada cuando se define el campo `.der`. A modo de ejemplo, para la función elemental $f(x) = \cos(x)$ el código que sobrecarga la función se puede ver en el Apéndice I.1.1.

La sobrecarga de un operador básico se realiza de manera análoga, analizando cuáles de los argumentos son de tipo `valder` y, en el caso de que lo sean, añadiendo la expresión de su derivada en el campo `.der`. Los ejemplos de la sobrecarga de los operadores suma y producto se han incluido en el Apéndice I.1.2 y I.1.3

Implementando el código para todos los operadores básicos y funciones elementales, la derivada de una función f se calcula inicializando `x = valder(x0,1)` y evaluando su expresión. Al ser x un objeto tipo `valder`, el valor de la función y de su derivada en x_0 será automáticamente proporcionado por los operadores sobrecargados previamente.

El objeto `valder` también puede operar con gradientes para funciones que dependen de varias variables. El hecho de no haber especificado qué tipo de clase deben tener los argumentos permite sustituir el valor de la derivada por las coordenadas del vector gradiente.
Por ejemplo, si la función depende de dos variables x e y, entonces se puede definir
`x = valder(3,[1,0])`
y

`y = valder(5,[0,1])`,
de forma que al ejecutar `x*y` obtendremos `[1,0]*5 + 3*[0,1] = [5,3]`. La salida en la ventana de comandos se puede ver en la Figura 2.2

```
>> x = valder(3,[1,0]); y = valder(5,[0,1]);
>> x*y

ans =

  valder with properties:

    val: 15
    der: [5 3]
```

Figura 2.2: Ejemplo del uso de `valder` para una función de dos variables.

2.2. Implementación del método regresivo

En esta sección trabajaremos con una función $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ suficientemente regular. La implementación del caso univariable puede verse como un caso particular para $n = 1$.

El método regresivo se puede dividir en tres partes.

- La primera es la propia evaluación de la función en el punto de interés.
- La segunda es el cálculo de las derivadas parciales de cada función intermedia respecto a sus variables independientes más inmediatas.
- La última parte es la evaluación de las derivadas parciales de la función original considerada respecto de cada función intermedia. Estas funciones intermedias se corresponden con los nodos del árbol asociado a la evaluación de una función matemática. Ejemplos de estos árboles se pueden encontrar en el Capítulo 1. La evaluación comienza en la derivada parcial respecto de ella misma y termina en las derivadas parciales respecto de las variables independientes.

Las dos primeras partes funcionan en sentido progresivo, es decir, comienzan en las funciones más elementales que componen la función inicial. La última parte de este método actúa en sentido regresivo.

Al estilo de lo mostrado en la Sección 2.1, dada una función matemática $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ de variables independientes $x_1, ..., x_n$ debemos construir un árbol de expresión binaria. Un árbol

de expresión binaria es un tipo de árbol binario que se usa para representar expresiones [8]. Para que este árbol se adapte a los dos sentidos en los que avanza el método, podemos construir un grafo dirigido con nodos numerados que lo represente.

Un *grafo* G es un par ordenado $G = (V, A)$ que se compone de V un conjunto de vértices o nodos y A un conjunto de aristas que representan la relación entre estos nodos.
El *orden* de un grafo se define como el número de vértices que posee y el tamaño como el número de aristas que tiene.

Un grafo se dice que es *dirigido* u *orientado* si las aristas tienen una orientación definida, es decir, se sabe que van de un nodo inicial a un nodo final. Cuando un grafo es dirigido también se dice que está definido por un conjunto de arcos en lugar de por un conjunto de aristas.

Cada grafo se puede representar por medio de una matriz. Las formas más comunes de representarlo son usando las matrices de adyacencia o incidencia. La *matriz de adyacencia* es una matriz cuyo elemento de la fila i y columna j es el número de aristas que unen el nodo n_i con el nodo n_j. Para un grafo orientado $M_{n\times n} = (a_{ij})$, donde n es el orden del grafo, $(a_{ij}) =$ número de arcos entre n_i y n_j y $(a_{ij}) = 0$ en otro caso. La *matriz de incidencia* es una matriz con número de filas el orden del grafo y número de columnas su tamaño. El elemento correspondiente a la fila i y a la columna j es 1 si el nodo i es el nodo inicial del arco que une el nodo i y el nodo j, -1 si el nodo n_i es un extremo final, 2 si en el nodo i hay un bucle y 0 en otro caso [2].

Ejemplo 2.1. Considérese la función $f(x) = x^2 \cdot \sin(x)$. El árbol asociado se puede ver representado en la Figura 1.1 Los siguientes grafos están asociados al árbol de esta función. El de la izquierda está numerado siguiendo el orden progresivo y el de la derecha el regresivo.

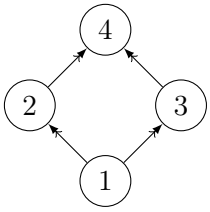


Figura 2.3: Grafo numerado en el orden progresivo

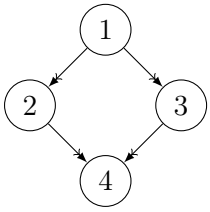


Figura 2.4: Grafo numerado en el orden regresivo

Las matrices de adyacencia e incidencia para el grafo de la Figura 2.4 son, respectivamente,

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \tag{2.1}$$

y

$$Q = \begin{pmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & -1 & 0 \end{pmatrix} \tag{2.2}$$

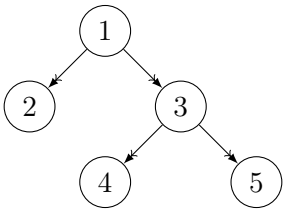
MATLAB permite implementar un grafo dirigido mediante la función *digraph*. También existen funciones propias de MATLAB para calcular el orden del recorrido progresivo y regresivo, así como pintar sobre el grafo información asociada a los nodos y las aristas. Para la implementación, necesitamos conocer esta información. Estas funciones evitan que tengamos que programarlo nosotros mismos. En realidad, la Diferenciación Automática regresiva tan solo requiere de un grafo muy específico, un árbol binario, pero la herramienta de MATLAB permite realizar el recorrido del grafo. Por este motivo y ya que está disponible, resulta conveniente usarla.

La manera de crear el grafo dirigido es proporcionar a la función la matriz de adyacencia del grafo o el conjunto de aristas. El conjunto de aristas se representa por medio de una matriz de dos columnas.

- En la primera columna se encuentran los nodos que son extremos iniciales de las aristas.
- En la segunda columna se encuentran los extremos finales.
- El número de filas indica el número de aristas del grafo.

En este trabajo se usará el conjunto de aristas para definir el grafo computacional y no será necesario calcular la matriz de adyacencia.

Ejemplo 2.2. A modo ilustrativo, considérese el siguiente grafo



cuyo conjunto de aristas es $A^t = \left(\begin{smallmatrix} 1 & 1 & 3 & 3 \\ 2 & 3 & 4 & 5 \end{smallmatrix}\right)$. La construcción en MATLAB de este grafo sería la siguiente.

```
mt = [1 1 3 3; 2 3 4 5];
m = [1 2; 1 3; 3 4; 3 5];
G = digraph(mt(1,:),mt (2,:)) % usando mt
G = digraph(m(:,1), m(:,2)) % usando m
```

La idea para nuestra implementación es representar la fórmula matemática mediante un árbol binario que recorreremos utilizando las herramientas para grafos de MATLAB.
Implementamos este árbol en dos fases:

- La primera es la creación de una estructura de Matlab en la que estén definidos los nodos y sus relaciones.
- En la segunda fase creamos una numeración para los nodos del árbol.

La numeración obtenida en la segunda fase permitirá definir un grafo con *digraph* y usar las funciones de MATLAB para recorrer automáticamente el árbol de forma progresiva y regresiva.

La implementación de las dos fases se separará en dos funciones diferentes para conseguir una buena estructuración del código.

La función que implementa la primera fase se llama *analiza*. Su finalidad es conseguir analizar la expresión de cada nodo y agruparla dentro de otras expresiones con las que guarde relación para, después, acceder a cada una de ellas y numerarlas. Esta función debe continuar con este proceso hasta llegar a una variable independiente o a una constante. En este momento, la función debe identificar que se encuentra al final del árbol y detener el proceso. Iremos comentando paso a paso las líneas del código de *analiza*. La función *analiza* completa se puede consultar en el Apéndice I.2.

Una función que se llama a sí misma durante la ejecución se denomina función recursiva. El factorial de un número natural se puede calcular utilizando una función recursiva. La parada de esta función se realiza cuando se calcula el factorial de uno.

```
function y = fact(n)
    if n == 1
        y = 1
    else
        y = n*fact(n-1)
    end
end
```

Para conseguir el objetivo expuesto en los párrafos previos, tenemos que definir *analiza* como una función recursiva que realice un procedimiento similar al de la función *fact* para agrupar las expresiones relacionadas con cada nodo del árbol.

2.2.1. Identificación de expresiones regulares

En primer lugar, la función *analiza* debe identificar qué símbolos, funciones y variables independientes componen la variable `formula`. Las variables independientes y las funciones elementales se separan en dos celdas de caracteres distintas. Sólo se han incluido dos variables independientes para que el código resulte más sencillo de comprender, pero podrían añadirse todas las que se quisieran.

Para poder identificar los símbolos de la expresión, usamos una expresión regular, que llamaremos `patron`. Una expresión regular es una secuencia de caracteres que actúa como patrón de búsqueda. En [3] se puede encontrar más información acerca de estos objetos.

```
patron = ['(' strjoin(fcs, '|') '|' strjoin(unkws, '|') '|\d
   +\.\d+|\d+|[a-zA-Z]+\([^\)]*\)|[+\-*/^()]|[a-zA-Z_][a-zA-Z_0
   -9]*)'];
```

Los símbolos de la función serán aquellos que coincidan con alguno de los incluidos en esta expresión regular. La función *regexp* puede utilizarse para comparar dos expresiones regulares y devolver las subcadenas que coincidan en ambas en una lista de expresiones. Si se utiliza para comparar las expresiones `formula` y `patron` se obtiene una lista de celdas. A esta lista la llamaremos `partes`. En cada celda se encuentra uno de los símbolos o funciones de `formula`.

```
partes = regexp(formula, patron, 'match');
```

Ejemplo 2.3. Considérese la expresión '$2 + \sin(x + y)$'. Los símbolos de `formula` en este caso son

$$\texttt{partes} = [\ \{2\}\ \ \{+\}\ \ \{\sin\}\ \ \{(\}\ \ \{x\}\ \ \{+\}\ \ \{y\}\ \ \{)\}].$$

2.2.2. Creación de la estructura de relaciones

Una vez obtenida la lista de cadenas y caracteres, la función *analiza* cuenta con la información necesaria para su verdadero objetivo: construir el árbol para la expresión `formula`. Separaremos

esta parte del código en una función interna que se llamará *crea_ arbol*. Se puede consultar el código de la función completa en el Apéndice I.2.

La función *crea_ arbol* tiene que identificar en la lista `partes` incógnitas, paréntesis, operadores básicos y funciones siguiendo las reglas del orden de prioridad en las operaciones. De esta manera, primero intentaremos identificar cuáles son las variables independientes, después localizaremos los paréntesis, seguidos por las funciones elementales. Por último, analizaremos qué operadores básicos contiene la expresión, también con el orden de preferencia usual. Para cada uno de estos casos, vamos a crear una variable `nodo` de tipo estructura que indique con cuál de estos casos se corresponde y cuál es su valor. Esta variable sustituirá a las componentes de la lista que representa.

En suma, se irán agrupando uno o varios elementos dentro de otros hasta que todos ellos formen parte de una única estructura que encierre toda la información.

Por simplicidad, se ha dividido el análisis de las variables independientes, los paréntesis, las funciones elementales y los operadores básicos en cuatro funciones definidas dentro de *crea_ arbol*. Estas cuatro funciones son *incognitas*, *parentesis*, *operadores* y *funcion*. En las búsquedas de nuevos elementos, las funciones devuelven un nodo vacío en caso de que no se encuentre tal elemento.

Por ejemplo, lo primero en el orden de prioridad sería buscar incógnitas, pero el código llama a *incongnitas* para que las busque una por una.

```
for ui = 1:length(unkws)
    nodo = true;
    while ~isempty(nodo)
        [partes, nodo] = incognitas(partes, unkws{ui});
    end
end
```

Para cada variable independiente especificada, la función *incognitas* busca la primera ocurrencia. Si la encuentra, crea una estructura nodo con los campos `.tipo` y `.valor`. A estos campos se asocia '?' y el valor de la incógnita ('x' o 'y'), respectivamente. Una vez que el nodo ha sido creado, se sustituye la entrada del índice correspondiente en la lista por este nodo.

```
function [partes, nodo] = incognitas(partes, ui)
ind = find(strcmp(partes, ui), 1);
    if isempty(ind)
        nodo = []; % No hay incognitas
```

```
            return;
        end
        nodo = struct('tipo', '?', 'valor', ui);
        partes = [partes(1:ind-1), {nodo}, partes(ind+1:end)];
    end
```

El siguiente paso es buscar paréntesis en la expresión, según el orden de prioridades. La función *parentesis* compara los string hasta encontrar el índice de la cadena '('. Si lo encuentra, buscará el cierre correspondiente.

```
ind = find(strcmp(partes, '('), 1);
        if isempty(ind)
            nodo = []; % No hay parentesis
            return;
        end

        % Buscar el cierre correspondiente
        nivel = 0;
        for i = ind:numel(partes)
            if strcmp(partes{i}, '(')
                nivel = nivel + 1;
            elseif strcmp(partes{i}, ')')
                nivel = nivel - 1;
                if nivel == 0
```

Una vez encontrado el cierre, crea una nueva lista con las partes del interior del paréntesis y vuelve a llamar a la función *crea_arbol* para generar el árbol del interior del paréntesis. Cuando ha sido creado, se crea una estructura nodo con los campos anteriores, pero con `.tipo` '()' y `.valor` el nuevo árbol asociado al interior. Este nodo se sustituye en la entrada en la que aparecía el paréntesis izquierdo y se eliminan el resto de entradas correspondientes a los índices en los que se encuentran el interior y el paréntesis derecho.

```
    if nivel == 0
        interior = partes(ind + 1:i - 1);
        valor_nodo = crea_arbol(interior);
        nodo = struct('tipo', '()', 'valor', valor_nodo);

        % reemplazar parentesis con nodo
        partes = [partes(1:ind-1), {nodo}, partes(i+1:end)];
```

```
    return;
end
```

De manera análoga a lo explicado para buscar paréntesis, se procede para la búsqueda de funciones elementales.

Por último, se realiza la búsqueda de operadores básicos por orden de prioridad. En esta función separamos las celdas que sean caracteres de las que son cadenas.

```
es_char = cellfun(@ischar, partes);
partes_op = false(size(partes));
partes_op(es_char) = ismember(partes(es_char),operador);
```

Una vez separados los caracteres en `partes_op` buscamos los índices en los que aparecen. Como hay operadores que se leen de izquierda a derecha y otros de derecha a izquierda, ha sido necesario incluir el argumento `izqda_decha` para indicarle al programa el sentido de búsqueda. Esta variable es de tipo lógico y es verdadera si el operador se lee de izquierda a derecha.

```
if izqda_dcha
    ind = find(partes_op, 1, 'first');
else
    ind = find(partes_op, 1, 'last');
end
```

Si se encuentra el índice para el caracter, se crea un nuevo árbol para las partes izquierda y derecha llamando de nuevo a *crea_ arbol*. Estos dos árboles se usan para definir un nodo estructura con campos `.tipo`, `.izqda` y `.dcha`, a cada uno de los cuales se asocia el tipo de operador `op` y los árboles izquierdo y derecho, respectivamente. Finalmente, se sustituyen el operador y las celdas en los índices anterior y posterior por el nuevo nodo.

```
if isempty(ind)
        nodo = [];
        return;
else
    % Construir nodo
    op = partes{ind};
    izqda = crea_arbol(partes(ind-1));
    dcha  = crea_arbol(partes(ind+1));
    nodo = struct('tipo', op, 'izqda', izqda, 'dcha', dcha);
    partes = [partes(1:ind-2), {nodo}, partes(ind+2:end)];
```

La función *crea_ arbol* debe parar cuando hayamos llegado a una variable independiente o a una constante. El hecho de llegar a una de estas dos opciones significa que la construcción de la estructura ha finalizado. En este supuesto, al igual que en la función factorial, necesitamos que haya algún tipo de cierre que termine las llamadas de la función. Veamos cómo se construye este cierre para *crea_ arbol*.

Si la lista de partes tiene un único elemento, sabemos que debe ser una variable independiente o una constante.

- Si el único elemento de la lista es una estructura, entonces es una variable independiente. En este caso la estructura es el nodo.
- En el caso de que el elemento no sea una estructura, entonces es una constante. Para este caso se crea una estructura nodo con campos `.tipo` 'cte.' y `.valor` el único elemento de la lista.

El siguiente fragmento de código se corresponde a lo explicado en este párrafo.

```
if isempty(nodo) && isscalar(partes)
        if isstruct(partes{1})
            nodo = partes{1};
        else
            nodo = struct('tipo', 'cte.', 'valor', partes{1});
        end
end
```

Ejemplo 2.4. Retomemos la función anterior $f(x, y) = 2 + sin(x + y)$. La lista para esta función se ha calculado previamente en el ejemplo anterior. Comentaremos los pasos que realiza la función *analiza* en la construcción del árbol.

En esta expresión hay dos variables independientes x, y. El programa busca el índice donde aparece x, que en este caso es cinco, crea la estructura para esta variable y hace que ocupe el lugar del string anterior. Como el nodo no es vacío, repite el proceso. No hay otro x en la expresión, con lo cual la variable nodo ahora es una lista vacía y se detiene el proceso. El proceso para la variable y es análogo. Ya tenemos las dos variables independientes como estructura dentro de la lista.

Ahora la función se encarga de buscar paréntesis. Encuentra el paréntesis izquierdo en la posición cuatro y el cierre correspondiente en la posición ocho. Las partes para el interior del paréntesis son

$$[\{x\} \ \{+\} \ \{5\}].$$

En este punto la función *crea_ arbol* se llama a sí misma. Análogamente a la llamada anterior de *crea_ arbol*, se buscan las variables independientes. No hay paréntesis ni funciones en esta lista. La función encuentra un operador suma para el índice dos y genera una estructura de tipo '+' con campos `.tipo`, `.izqda` y `.dcha`. El valor de cada uno de los dos últimos campos corresponde a una nueva llamada de la función para crear sendos árboles.

Las partes del árbol de la izquierda tienen una única componente que es una variable independiente. Esta variable es de tipo estructura porque se ha definido así en la primera llamada, con lo que la función *crea_ arbol* para el árbol izquierdo finaliza. El proceso es igual para el árbol derecho, cuya única componente es la variable y.

Regresamos a la primera llamada de la función. Las partes que nos han quedado después de analizar los paréntesis de la expresión son

$$\texttt{partes} = [\{2\} \quad \{+\} \quad \{\sin\} \quad \{\text{`()'}\}].$$

El siguiente paso es identificar las funciones elementales. La única función elemental es sin. Como está antes de un paréntesis se crea una estructura nodo con tipo 'sin' y valor la estructura interior del paréntesis.

Para el resto de funciones de la lista no se puede encontrar ningún índice, con lo que el nodo es vacío y no hay cambios.

Hasta ahora las modificaciones sobre la lista `partes` han dado como resultado

$$\texttt{partes} = [\{2\} \quad \{+\} \quad \{\sin\}].$$

Retomando la primera ejecución de *crea_ arbol*, el siguiente paso es buscar operadores básicos. Se encuentra '+' en el índice dos y se llama a sí misma para crear los árboles de las partes izquierda y derecha.

El árbol de la derecha trabaja con una lista que tiene un único elemento. Dicho elemento es de tipo estructura porque ya se había definido así en la etapa de búsqueda de constantes y la función llega a un cierre. El árbol de la izquierda también tiene que trabajar con una lista de un único elemento, que no es una estructura. La única posibilidad es que sea una constante, así que el programa crea una variable `nodo` tipo estructura con campos `.tipo` y `.valor` con los valores 'cte' y 2, respectivamente, y la ejecución termina.

En la Figura 2.5 se puede ver la salida en MATLAB que obtiene `arbol = analiza('2+sin(x+y)',{'x','y')}`.

```
arbol =

  struct with fields:

     tipo: '+'
    izqda: [1×1 struct]
     dcha: [1×1 struct]

>> arbol.dcha

ans =

  struct with fields:

     tipo: 'sin'
    valor: [1×1 struct]
```

Figura 2.5: Campos del árbol para la expresion '$2 + sin(x + y)$'.

Se puede ver que cuando accedemos a uno de los campos de `arbol` llegamos a otra estructura interior de tipo 'sin'.

Si reflexiona sobre el tema, el lector se dará cuenta de que sólo estamos implementando el proceso natural que se sigue al dibujar el árbol en un papel. Lo primero que haría una persona que quisiera dibujar el árbol para una expresión concreta, sería razonar cómo se agrupan las operaciones. De este procedimiento previo es del que se encarga *crea_ arbol*. En realidad, cuando se dibuja el árbol con las expresiones intermedias, lo que se está haciendo es dibujar un grafo dirigido con los nodos sin numerar, pero siguiendo un orden que no está escrito dado implícitamente por información anterior. El siguiente paso en la implementación del método regresivo es conseguir asociar esta numeración a los nodos y aristas del árbol. Una vez conseguido este objetivo, se podrá extraer la matriz que representa al conjunto de aristas y usarla para construir el grafo dirigido.

2.2.3. Creación de la numeración del árbol

Para numerar los nodos y aristas del árbol hay que saber cómo se relacionan los nodos entre sí, es decir, qué padres y qué hijos tiene cada uno de ellos; para ello utilizaremos el árbol creado. Este árbol está compuesto por estructuras encadenadas. Cada estructura tiene dentro una o varias estructuras, que serán hijos de la estructura anterior. Una vez numerados los hijos, podremos decir qué nodo es su padre y construir la arista correspondiente. Obsérvese que de esta manera comenzamos numerando con 1 el nodo correspondiente a la expresión completa y avanzamos de arriba hacia abajo en la numeración del resto de nodos. Este orden es el que buscábamos para programar el camino regresivo del método.

La información que se necesita obtener de cada nodo es:

1. Número, nombre y valor asociado a ese nodo.

2. Hijos del nodo.

3. Padres del nodo.

4. El tipo de estructura que es. Esto es necesario para conocer si debemos continuar analizando el árbol.

5. Aristas que relacionan al nodo con sus padres.

6. Aristas que salen de ese nodo.

7. Constantes en cada nodo.

Estas propiedades se almacenarán en una estructura con nombre `prop`. La estructura se puede definir de la siguiente manera y se actualizará cada vez que pasemos por un nodo distinto.

```
prop = struct();
prop.arista  = {}; %cell(aristas, 1) con vector 1x2
prop.hijo    = {}; %cell(nodos,   1) con vector 2x1
prop.padre   = {}; %cell(nodos,   1) con vector nx1
prop.ari     = {}; %cell(nodos,   1) con vector 2x1
prop.nombre  = {}; %cell(nodos,   1) con string
prop.tipo    = {}; %cell(nodos,   1) con string
prop.cte     = {}; %cell(nodos,   1) con double
```

Se pueden inicializar los campos de `prop` como celdas vacías porque no es necesario conocer la longitud, pues pararán de añadirse elementos al llegar a los nodos sin hijos.

También crearemos una estructura que asigne el valor indefinido (en Matlab `NaN`) a las variables independientes.

```
cnan = num2cell(NaN(1, length(incog_var)));    % cell(1, incog)
   con NaN
prop.incog = cell2struct(cnan, incog_var, 2);
```

Modificaremos el valor indefinido la primera vez que encontremos una variable independiente. Esto permite distinguir si es la primera vez que se llega a esa variable o si no ha sido su primera ocurrencia. Si se numerase cada ocurrencia, habría nodos innecesarios que aumentarían el coste de la función y dificultarían su implementación, pues tendríamos que estar buscando qué nodos contienen la variable independiente para calcular la derivada parcial respecto de ella.

Para organizar la información, separamos de nuevo el código que construye el grafo dirigido en una función *crea_grafo*. Esta función vuelve a ser una función recursiva. Los argumentos que necesita son una estructura tipo árbol, que ya hemos construido, los padres de cada nodo del árbol y las propiedades anteriores.

Nótese que si la estructura es de tipo '()' lo que realmente es el nodo no es el paréntesis, sino lo que hay en el interior. Por este motivo, lo primero que hace nuestra función *crea_grafo* es eliminar estos paréntesis, si existen, y cambiarlos por la estructura que contienen en su interior.

```
while strcmp(estr.tipo, '()')
    estr = estr.valor;
end
```

Después de eliminar los paréntesis, la función tiene que analizar de qué tipo es la estructura. Recordemos que es una función recursiva y actúa para cada nodo, no para un árbol completo. Queremos ver de qué tipo es la estructura del nodo y clasificarlo.

Si la estructura es una variable independiente, comprobamos si todavía está asociada a un valor indefinido. Si lo está, es la primera vez que aparece esa variable. Actualizamos los campos de la estructura propiedades.

```
if strcmp(estr.tipo, '?') % solo variables
        if isnan(prop.incog.(estr.valor)) % por ejemplo, incog.x =
           NaN
            % En este caso no he pasado por esta incognita ninguna
               vez
            prop.tipo   = [prop.tipo; {[estr.valor estr.tipo]}]; %
               tipo, 'x?'
            prop.cte    = [prop.cte;  {NaN}]; % cte, NaN
            prop.nombre = [prop.nombre; ...
                {[num2str(length(prop.nombre)+1) ': ' estr.valor
                   estr.tipo]}]; % nombre, '6: x?'
            prop.incog.(estr.valor) = length(prop.nombre); %incog.x
               , 6
            prop.padre{length(prop.nombre)}  = [];

        end
        actual = prop.incog.(estr.valor);
```

Obsérvense los campos `prop.nombre` y `prop.incog`. El primero se ha actualizado como la lon-

gitud de `prop.nombre` más uno y el segundo como el valor de la estructura. De esta manera numeramos el nodo actual como el siguiente a los que ya estaban incluidos e identificamos la primera ocurrencia de la variable independiente. En el caso de que la variable independiente no esté asociada a un valor indefinido, entonces ya ha sido almacenada como estructura y tiene un número asignado. De esta forma se puede identificar qué variables independientes han sido numeradas.

Si la estructura es una constante actualizamos la estructura `prop` igual que antes, aunque esta vez con el valor de la constante en `prop.cte`.

En otro caso, actualizamos la estructura de propiedades y en el campo `.cte` escribimos `NaN`.

```
else
    prop.tipo   = [prop.tipo; {estr.tipo}];
    prop.cte    = [prop.cte;  {NaN}];
    prop.nombre = [prop.nombre; {[num2str(length(prop.nombre)
       +1) ': ' estr.tipo]}];
    actual = length(prop.nombre);
    prop.padre{length(prop.nombre)}  = [];
end
```

Obviamente, para estos dos últimos escenarios no se actualiza `prop.incog`.

En todos estos casos, una vez definido el nombre del nodo, definimos el nodo en el que estamos con la longitud de `prop.nombre`: `actual = length(prop.nombre)`. La longitud de los nombres será la de los nodos que ya hemos recorrido, contando con el nodo actualizado.

Una vez identificado el nodo actual definimos el padre y los hijos. Veamos cómo lo hacemos. Si para el nodo actual la lista padre no está vacía, comenzamos las asignaciones de aristas del grafo, aristas que salen del nodo, padres e hijos. A las aristas incidentes añadimos el vector que va desde el padre hasta el nodo actual `actual`.

```
    prop.arista = [prop.arista; {[padre,actual]}];
    prop.ari{padre}    = [prop.ari{padre};    length(prop.
       arista)];
```

En este punto conocemos el padre del nodo actual. Entonces podemos asignar el hijo al nodo padre y el padre al nodo actual.

```
prop.hijo{padre}   = [prop.hijo{padre};   actual];
prop.padre{actual} = [prop.padre{actual}; padre];
```

Finalmente, si fuese necesario continuar clasificando nodos, llamamos a la función recursiva. Será

necesario continuar en estos casos:

- Si existe algún operador básico en la estructura, hacemos la llamada para los lados izquierdo y derecho.
- Si no existe y la estructura no es de tipo variable ni constante, hacemos la llamada para el valor de la estructura.

El hecho de no incluir las constantes ni las variables independientes es lo que construye el cierre de la función, ya que al llegar a uno de estos dos supuestos estaremos en el nivel más bajo del árbol y la función debe terminar su ejecución.

```
if any(ismember({'^', '*', '/', '+', '-'}, estr.tipo))
    prop = crea_grafo(estr.izqda, prop, actual);
    prop = crea_grafo(estr.dcha,  prop, actual);
elseif ~strcmp(estr.tipo, '?') && ~strcmp(estr.tipo, 'cte.')
    prop = crea_grafo(estr.valor, prop, actual);
end
```

Ejemplo 2.5. Habíamos construido anteriormente el árbol de una expresión concreta

$$f(x, y) = 2 + \sin(x + y).$$

Los pasos que seguiría la función *crea_grafo* se recogen a continuación.

1. **PASO 1:**
   `estr.tipo = '+'`
   `prop.tipo = ['+']`
   `prop.cte = [{NaN}]`
   `prop.nombre = [1: '+']`
   `actual = 1`
   `prop.padre{length(prop.nombre)} = prop.padre {1} = []`
   Como el padre de este nodo es vacío y tipo es '+', actualizamos la lista de propiedades con la función recursiva:
   - `prop = crea_grafo(estr.izqda,prop,1)`
   - `prop = crea_grafo(estr.dcha,prop,1)`

2. **PASO 2:**
   `estr.tipo = 'cte'`

`prop.tipo = ['+'; 'cte']`
`prop.cte = [{NaN}, {2}]`
`prop.nombre = [1:'+'; 2: 'cte' 2]`
`actual = 2`
`prop.padre {2} = []`
`prop.hijo{actual} = []`
`prop.ari{actual} = []`
Como el padre de este nodo no es vacío:

- `prop.arista = [{[1,2]}]`
- `prop.ari{1} = [1]`;
- `prop.hijo {1} = [actual] = [2]`;
- `prop.padre{2} = [padre] = [1]`;

Como es una constante no se realiza ninguna llamada.

3. **PASO 3:**
`estr.tipo = 'fcs'`
`prop.tipo = ['+'; 'cte'; 'fcs']`
`prop.cte = [{NaN}, {2}; {NaN}]`
`prop.nombre = [1:'+'; 2: 'cte' 2; 3: 'fcs']`
`actual = 3`
`prop.padre{3} = []`
`prop.hijo{actual} = []`
`prop.ari{actual} = []`
Como el padre de este nodo no es vacío:

- `prop.arista = [{[1,2]}, {[1,3]}]`
- `prop.ari{1} = [1; 2]`;
- `prop.hijo{1} = [2; 3]`;
- `prop.padre{3} = [padre] = [1]`;

Como no hay operadores y no es una constante ni una variable: `prop = crea_grafo(estr.valor, prop,3)`.

4. **PASO 4:**
`estr.tipo = '()'` → `estr = estr.valor`, estructura de tipo '+' (elimina el paréntesis)
`prop.tipo = ['+'; 'cte'; 'fcs';'+']`

```
prop.cte = [{NaN}, {2}; {NaN}; {NaN}]
prop.nombre = [1:'+'; 2: 'cte' 2; 3: 'fcs'; 4: '+']
actual = 4
prop.padre{4} = []
prop.hijo{actual} = []
prop.ari{actual} = []
```

Como el padre de este nodo no es vacío:

- `prop.arista = [{[1,2]}, {[1,3]}, {[3,4]}]`
- `prop.ari{3} = [1; 2; 3]`
- `prop.hijo{3} = [4]`
- `prop.padre{4} = [padre] = [3]`

  Como existe un operador '+' en la expresión actualizamos la lista de propiedades:

  - `prop = crea_grafo(estr.izqda, prop, 4)`
  - `prop = crea_grafo(estr.dcha, prop, 4)`

5. **PASO 5:**

```
estr.tipo = '?'
prop.tipo = ['+'; 'cte', 'fcs', '+'; '?']
prop.cte = [{NaN}, {2}, {NaN},{NaN},{NaN}]
prop.nombre = [1:'+'; 2: 'cte' 2; 'fcs'; 4: '+'; 5: '?' ]
actual = 5
prop.padre{5} = []
prop.hijo{actual} = []
prop.ari{actual} = []
```

Como el padre de este nodo no es vacío:

- `prop.arista = [{[1,2]}, {[1,3]}, {[3,4]}, {[4,5]}]`
- `prop.ari{4} = [4]`
- `prop.hijo{4} = [5]`
- `prop.padre{5} = [padre] = [4]`

Como es una variable no se realiza ninguna llamada.

6. **PASO 6:**

```
estr.tipo = '?'
prop.tipo = ['+'; 'cte', 'fcs', '+'; '?'; '?']
prop.cte = [{NaN}, {2}, {NaN},{NaN},{NaN},{NaN}]
```

```
prop.nombre = [1:'+'; 2: 'cte' 2; 'fcs'; 4: '+'; 5: '?' ; 6: '?']
actual = 6
prop.padre{6} = []
prop.hijo{actual} = []
prop.ari{actual} = []
```

Como el padre de este nodo no es vacío:

- `prop.arista = [{[1,2]}, {[1,3]}, {[3,4]}, {[4,5]}, {[4,6]}]`
- `prop.ari{4} = [4,5]`
- `prop.hijo{4} = [5,6]`
- `prop.padre{5} = [padre] = [4]`

Como es una variable no se realiza ninguna llamada.

Ahora que ya tenemos terminada la lista de propiedades para un árbol, se extraen las aristas y se utilizan para construir el grafo con la función *digraph* de MATLAB. El grafo se construye de la misma manera que la mostrada en el Ejemplo 2.2. La matriz que representa el conjunto de aristas se corresponde con el campo de la estructura `.arista` convertido previamente a matriz. Si representamos el grafo generado para el ejemplo anterior con la función *plot*, se obtiene la gráfica representada a continuación.

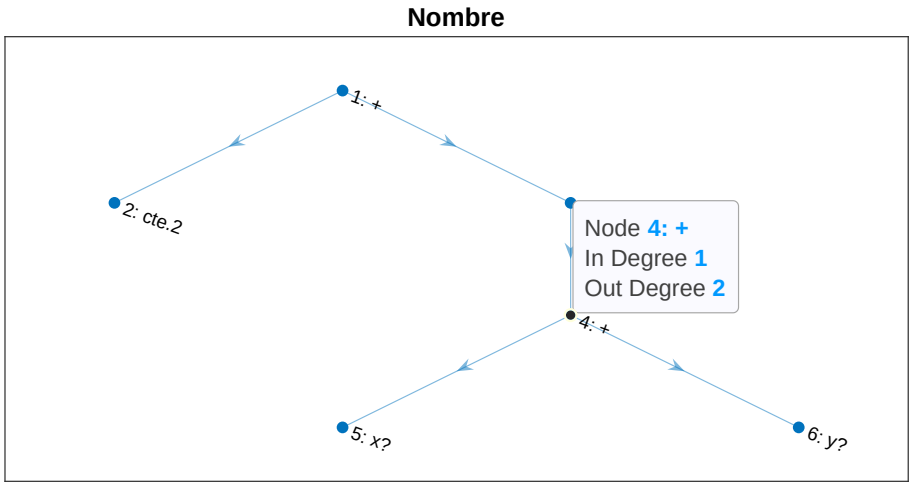


Figura 2.6: Grafo computacional asociado a la expresión $2 + \sin(x + y)$.

2.2.4. Cálculo de las variables `barra`.

La construcción del grafo computacional ha sido laboriosa, pero una vez terminado, el código para implementar el método regresivo no resulta demasiado complicado. Recordemos que se puede dividir en tres pasos:

1. Evaluación de la función en el punto.
2. Cálculo de las derivadas parciales de las funciones de cada nodo respecto de sus variables independientes más inmediatas.
3. Cálculo de las derivadas parciales de la función del primer nodo con respecto de las variables de cada nodo, es decir, las ecuaciones 1.1, 1.2, 1.3, 1.4 y 1.5 de la Sección 1.2.

A las derivadas parciales del tercer ítem, siguiendo la notación de [4], las llamaremos `barra`. El orden en sentido progresivo es la numeración de los nodos del grafo recorrido en sentido inverso. Si almacenamos la numeración del grafo en una variable, que llamaremos `regresivo`, se puede utilizar la función *flip* para invertir el orden. Así, podemos almacenar el sentido en el que queremos recorrer el grafo en dos variables diferentes y cambiar de uno a otro cuando sea necesario.

```
regresivo  = toposort(G);  % ordena los nodos
progresivo = flip(regresivo)
```

Para los dos primeros pasos podría usarse otro tipo de implementación, como, por ejemplo, el `valder` definido en la sección anterior. En este trabajo se ha realizado utilizando el grafo computacional por dos motivos. El primero es que ya se ha programado una función para construirlo y el segundo, que se quiere estudiar la aplicación de éstos a la Diferenciación Automática.

A continuación se detalla la programación efectuada para los tres pasos anteriores.

En primer lugar se efectúa la evaluación de todas las funciones intermedias, es decir, de las funciones asociadas a los nodos. Para almacenar los valores se inicializa la variable `val` con tamaño el orden del grafo. Estas evaluaciones se realizan analizando de qué tipo es el nodo y asignándole un valor con algún criterio. Los primeros nodos en sentido progresivo son constantes o variables independientes. Estos nodos llevan el valor que hay en la entrada de `prop.cte` correspondiente a su número y los valores del punto x_0, respectivamente.

```
if strcmp(prop.tipo(in), 'cte.')
     val(in) = prop.cte{in};
elseif endsWith(prop.tipo(in), '?') % variable
     val(in) = punto.(prop.tipo{in}(1:end-1));
```

Los nodos que sean operadores básicos son nodos que tienen dos hijos. Su valor será la combinación correspondiente del valor de sus hijos. Un operador nunca será uno de los primeros nodos, por lo que estas asignaciones están bien definidas. Por ejemplo, para el operador división tenemos el siguiente fragmento de código.

```
elseif strcmp(prop.tipo(in), '/')
     valh = val(prop.hijo{in});
     val(in) = valh(1)/valh(2);
```

Las funciones elementales tienen un único hijo. Su valor se obtiene evaluándolas en el punto dado por el valor de su hijo.

```
elseif strcmp(prop.tipo(in), 'log')
     val(in) = log(val(prop.hijo{in}));
```

Sólo se han incluido las funciones elementales seno, coseno, valor absoluto, logaritmo y secante hiperbólica por falta de tiempo en este estudio, pero podrían añadirse las restantes funciones con una definición análoga. Por ejemplo, si se quisiese añadir la función raíz cuadrada, se modificaría el código añadiendo las líneas

```
elseif strcmp(prop.tipo(in), 'sqrt')
val(in) = sqrt(val(prop.hijo{in}));
```

en el interior del bucle que recorre los nodos.

Se pueden representar los valores de cada uno de los nodos con la función *plot*, aunque se debe hacer un pequeño cambio, que es convertir los valores numéricos a string para que la función trabaje correctamente.

```
G.Nodes.val = val;
subplot(2,2,2)
plot(G, 'NodeLabel', arrayfun(@num2str, G.Nodes.val, '
   UniformOutput', false))
title('Valor')
```

Sin ir más lejos, para la función $f(x, y) = 2 + \sin(x + y)$ del ejemplo anterior se obtienen los valores representados en la Figura 2.7.

Antes de continuar con la implementación de los pasos del método, obsérvese que la función *digraph* modifica el número con el que las aristas son almacenadas. Valga por caso el Ejemplo 2.2 que usamos para ilustrar el uso de *digraph*. Para el grafo con el conjunto de aristas dado por

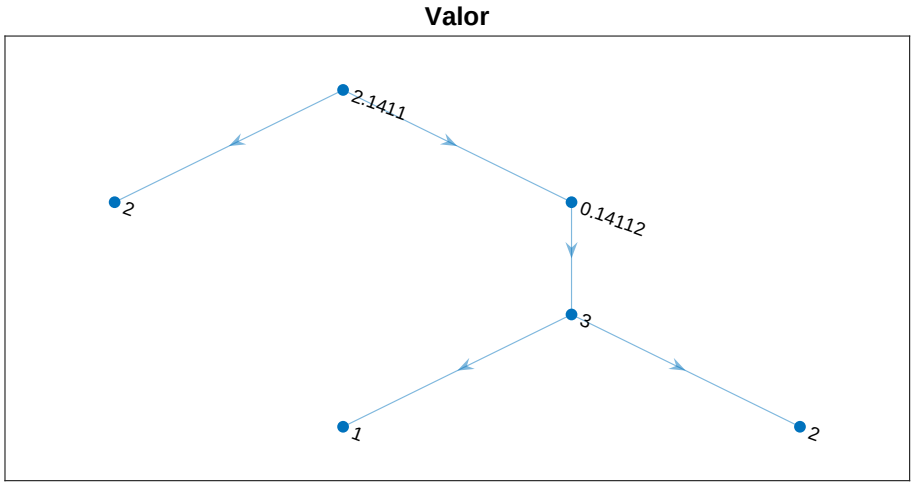


Figura 2.7: Valores de los nodos del grafo computacional asociado a la expresión $2 + \sin(x + y)$.

la matriz $M^t = \left(\begin{smallmatrix} 1 & 1 & 3 & 3 & 2 \\ 2 & 3 & 4 & 5 & 5 \end{smallmatrix}\right)$, nosotros hemos numerado con un 2 la arista que va del nodo 3 al nodo 4, pero si buscamos las aristas con *findedge*,

```
G = digraph([1 1 3 3 2], [2 3 4 5 5]);
[s,t] = findedge(G);
```

MATLAB devuelve `s = [1 1 3 3 2], t =[2 3 4 5 5]`.

La forma en la que se almacenan las aristas es importante porque se usarán para guardar las derivadas parciales, pero esto no supone una complicación. En lugar de acceder a las aristas por el orden en el que se han construido, se busca el orden que les ha dado *digraph* con *findedges* y se guarda en una nueva variable. Esta variable se llamará `m_recuperado`.

```
[s, t] = findedge(G); % s = nodos origen, t = nodos destino.
m_recuperado = [s, t];
```

Los valores que se necesitan conocer en cada nodo son el valor de sus hijos, las aristas que lo unen con sus hijos y si los hijos son constantes. Para cada nodo se crean tres variables `v`, `ar` y `cte` que almacenan estos datos, respectivamente.

```
parc = zeros(size(m_recuperado,1),1);
for in = progresivo
    for ia = 1:length(prop.ari{in})
        v   = zeros(size(prop.hijo{in})); % valor de los hijos
        cte = zeros(size(prop.hijo{in})); % si son constantes
            los hijos
```

```
ar  = zeros(size(prop.hijo{in})); % indice de arista de
    los hijos
for ih = 1:length(prop.hijo{in})
    v(ih)   = val(prop.hijo{in}(ih));
    cte(ih) = strcmp(prop.tipo{prop.hijo{in}(ih)}, 'cte
       .');
 ar(ih)  = find(ismember(m_recuperado, [in, prop.hijo{
    in}(ih)], 'rows'));
end
```

Las funciones elementales sólo tienen un hijo. Las derivadas parciales respecto de su hijo se calculan evaluando la expresión de la derivada en el valor de los hijos. A modo de ejemplo, a la derivada de secante hiperbólica le corresponde el siguiente fragmento de código.

```
elseif strcmp(prop.tipo(in), 'sech')
        parc(ar(1)) = -sinh(v(1))/cosh(v(1))^2;
```

Los nodos que son operadores tendrán dos hijos, es decir, dependen de dos variables. Por este motivo, estos nodos tienen dos derivadas parciales, que se almacenan en un vector de dos componentes. Sea $g(u, v)$ la función asociada a un nodo del tipo de los anteriores.

- Si el nodo es una suma, se guardan las parciales (1, 1).
- Si el nodo es una resta, se guardan las parciales (1, -1).
- Si el nodo es una multiplicación, se guardan las parciales (v(2), v(1)).
- Si el nodo es una división, se guardan las parciales $\left(\frac{1}{v(2)}, \frac{-v(1)}{v(2)^2}\right)$.

Por ejemplo, si el nodo es una suma las parciales se calcularían así.

```
if strcmp(prop.tipo(in), '+')
        parc(ar(1)) = 1;
        parc(ar(2)) = 1;
```

Una vez calculadas las parciales, en sentido regresivo, se calculan las variables `barra`. El caso en el que el nodo no tiene padres se ha separado del resto, pues es el primer nodo del grafo y la entrada de `barra` en ese nodo siempre vale uno.

```
if isempty(prop.padre{in})
     barra(in) = 1;
```

Así como un nodo puede tener uno o dos hijos, un nodo puede tener más de dos padres, salvo el raíz. Para calcular los valores de las variables `barra`, hay que tener en cuenta cuántos padres tienen. Tal y como hemos expuesto en un ejemplo en la Sección 1.2, ecuaciones 1.10, 1.11 y 1.12, la variable `barra` de un nodo con función g, es el resultado de sumar para cada uno de sus padres la derivada parcial de la arista que lo une con su hijo por la variable `barra` del nodo padre.

```
barra(in) = 0;
     for ip = prop.padre{in}'
        ar  = find(ismember(m_recuperado, [ip, in], 'rows'));
        barra(in) = barra(in) + barra(ip)*parc(ar);
     end
```

A continuación, mostramos los resultados que proporcionan las funciones *analiza* y *barras* para la función $f(x, y) = 2 + \sin(x + y)$ de los Ejemplos 2.4 y 2.5
Las parciales `parc` se pueden ver en la Figura 2.8. Nótese que los valores aparecen en las aristas del grafo. La Figura 2.9 representa los valores de las variables `barra` en cada nodo. Los valores que se encuentran al lado de los nodos asociados a las variables independientes son las derivadas parciales de f: $\overline{x} = \frac{\partial f}{\partial x}$ y $\overline{y} = \frac{\partial f}{\partial y}$.

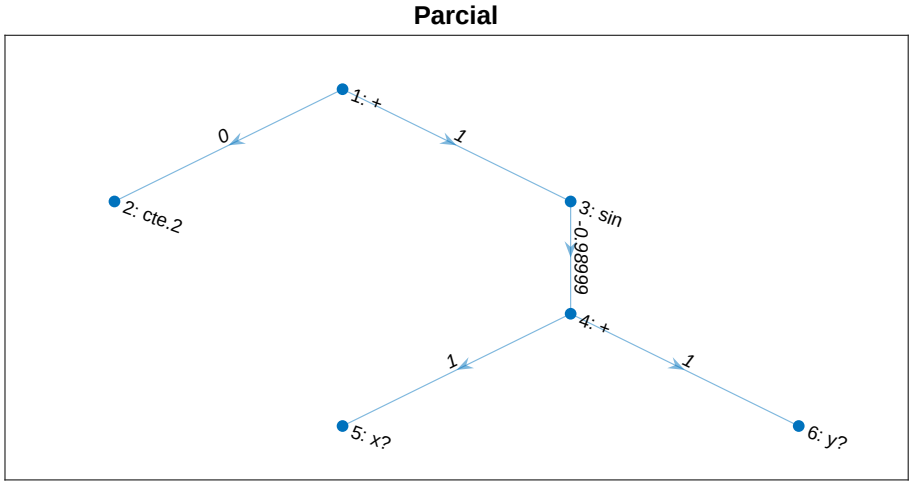


Figura 2.8: Valores de las parciales del grafo computacional asociado a la expresión $2+\sin(x+y)$.

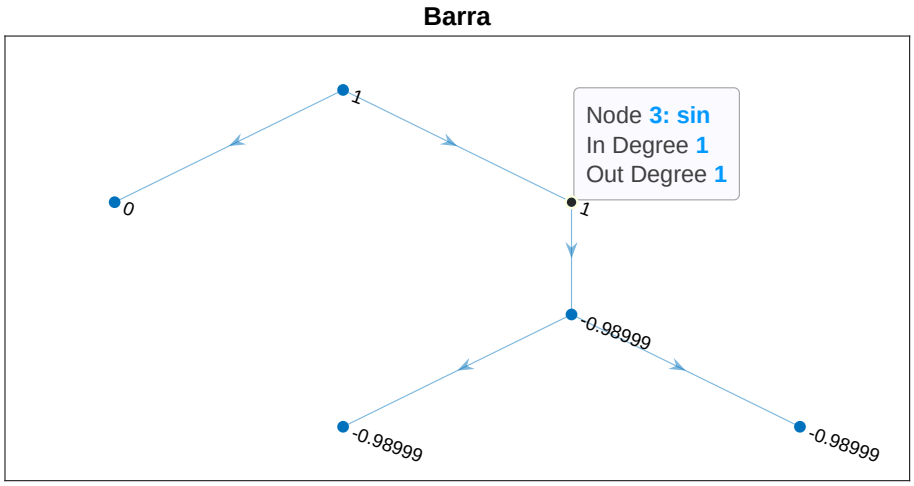


Figura 2.9: Valores de las barras del grafo computacional asociado a la expresión $2 + \sin(x + y)$.

Para concluir este capítulo, se destacan las siguientes observaciones.

- La implementación de las dos primeras partes puede realizarse de manera similar a la implementación de la Diferenciación Automática progresiva, pero esto es inviable para la última parte. El motivo es que, al avanzar en sentido regresivo, no somos capaces de identificar cuál es el nodo que va después de cada derivada parcial. Esto hace que sea necesario construir un grafo dirigido u orientado con nodos numerados que indique el orden en el que se deben hacer las operaciones.
- En realidad, como se observó en el capítulo anterior, para avanzar en sentido progresivo también estamos siguiendo un orden predeterminado que se puede representar con un grafo, pero para la implementación de la Diferenciación Automática progesiva no fue necesario construirlo porque es el orden natural en el que está implementada la evaluación en una máquina.

Capítulo 3

Resultado numérico

Este capítulo reflexionará sobre la eficiencia de los métodos tratados en capítulos anteriores. El objetivo es intentar realizar un estudio muestral que relacione el tiempo de cálculo de la máquina con el método de diferenciación utilizado: simbólico, Diferenciación Automática progresiva y Diferenciación Automática regresiva. Para ello se ha escogido una selección de expresiones que consideramos representativas del conjunto de posibles funciones a las que podrían aplicarse los métodos progresivo y regresivo. No se compararán la diferenciación numérica y la Diferenciación Automática porque la segunda es exacta y la primera no.

Los cálculos para todas las derivadas parciales del último ejemplo no han sido incluidos porque serían demasiados, pero, al igual que en los casos anteriores, están bien calculadas y sus valores coinciden para todos los métodos[1]. El lector interesado puede comprobarlo ejecutando el código del Anexo I.4 para $f(x, y) = 2 \cdot (x \cdot z) + \sin(x + y)^2$.

Calcular todas las derivadas parciales con cada método, permite validar la implementación del método. Los valores de las derivadas parciales coinciden para todos los métodos. Estos datos se pueden ver el Cuadro 3.1. Es obvio que no hemos comprobado el funcionamiento del código para todas las posibles combinaciones de símbolos y caracteres que pueden aparecer en la expresión de una función, pero el haber obtenido resultados correctos para las cinco funciones anteriores nos da confianza en que los métodos para la selección de operadores y funciones que hemos hecho estén bien implementados. Asimismo, se pueden añadir más funciones elementales al código del Anexo I.3. Si se definen adecuadamente, los resultados deberían ser correctos.

El Cuadro 3.1 también recoge los tiempos que emplea la máquina en hacer los cálculos con

[1] El tiempo de cálculo puede variar dependiendo de la máquina y de la ejecución.

[2] Las funciones y objetos de la clase `valder` deben haber sido implementados previamente para que la función f esté sobrecargada.

Función	Variables independientes / Punto	Simbólico	Progresivo	Regresivo
$f(x,y) = \log(5 \cdot x) + \cos(4 - \frac{1}{\lvert x \rvert})$	$x = 1$	0.8588	0.8588	0.8588
	TIEMPO(s)	0.021434	0.8588	0.222721
$f(x,y) = 2 + \sin(x + y)$	$x = 1$	-0.9899	-0.9899	-0.9899
	$y = 2$	-0.9899	-0.9899	-0.9899
	TIEMPO (s)	0.084846	0.007950	0.201128
$f(x,y,z) =$ $2 \cdot \cos(x \cdot z) + \sin(x + y)$	$x = 1$	-1.8367	-1.8367	-1.8367
	$y = 2$	-0.9899	-0.9899	-0.9899
	$z = 3$	-0.2822	-0.2822	-0.2822
	TIEMPO (s)	0.065042	0.009830	0.149966
$f(x,y,z,k,r,t) = \log(x \cdot z +$ $y \cdot x) \cdot cos\left(\frac{k}{(z - \sin(r+t))}\right)$	$x = 1$	0.9577	0.9577	0.9577
	$y = 1$	0.0871	0.0871	0.0871
	$z = 10$	0.1067	0.1067	0.1067
	$k = 3$	-0.0671	-0.0671	-0.0671
	$r = 5$	-0.0188	-0.0188	-0.0188
	$t = 1$	-0.0188	-0.0188	-0.0188
	TIEMPO(s)	0.180255	0.010579	0.218995
$f(x,y,z,r,s,t,w,u,v,p,q,l,$ $m,n,h,f,g) = \cos(\text{sech}(x +$ $\cos(y) \cdot r \cdot w))/t \cdot \sin(s +$ $\log(x/z)) + 2 \cdot \text{sech}(u \cdot v) -$ $(n \cdot f)/g \cdot \log((l + m)/p)$ $\cdot(5 \cdot h \cdot l - n \cdot q)$	TIEMPO(s)	0.636728	0.110039	0.202837

Cuadro 3.1: Valores y tiempos de ejcución para los tres métodos

cada uno de los métodos.

La máquina tarda más tiempo en calcular las derivadas parciales con el método regresivo cuando las funciones dependen de pocas variables. El cálculo regresivo acumula el tiempo de hacer la estructura de relaciones, la numeración y el tiempo de evaluar las derivadas parciales y las variables `barra`. Sin embargo, si hay que evaluar en varios puntos, la estructura de relaciones con su respectiva numeración se crearía una sola vez. En las primeras cuatro funciones la diferenciación simbólica o Automática progresiva son más eficientes. Sin embargo, para el ejemplo en la fila cinco se observa que el tiempo del cálculo simbólico comienza a igualarse con el del cálculo de la diferenciación automática regresiva y cuando se analiza el ejemplo para 17 variables

independientes el tiempo que tarda la diferenciación simbólica triplica al del método regresivo. En cuanto a la Diferenciación Automática progresiva, funciona con mucha eficiencia en los cinco casos. Sería la opción más recomendable, pues el orden para el tiempo que tarda en realizar los cálculos es del orden de centésimas o milésimas de segundo. La única función para la que tarda décimas de segundo es la última y, aún en este supuesto, es la que menos tiempo emplea.

Estos datos quizás puedan hacer que el lector se pregunte por qué se ha trabajado en hacer un código para la Diferenciación Automática regresiva, pues los tiempos de ejecución parecen indicar que el método progresivo es el más eficiente, al menos para estos ejemplos.

La respuesta es que todavía estamos trabajando con una cantidad de variables independientes manejable. El objetivo de ilustrar el método regresivo con claridad nos impide escribir aquí una función con 100 o 1000 variables independientes, pero en esta situación sería más aconsejable utilizar el método regresivo, pues como se indica en [4], el número de operaciones depende del coste de la función multiplicado por una constante. Para una función de n variables independientes, el coste del método progresivo sería $(n + 1)$ veces el coste de evaluar la función. Puede que en los cinco ejemplos considerados no se aprecie esta consecuencia, pero si hubiese una cantidad de variables independientes demasiado grande, la Diferenciación Automática regresiva sería más eficiente que las demás.

Una observación que ayuda a confiar en este resultado teórico es que el tiempo de ejecución del método regresivo es más estable. Se puede ver que no presenta grandes cambios de un ejemplo a otro, mientras que el tiempo que tarda el método progresivo varía de milésimas de segundo hasta décimas de segundo desde el primer hasta el último caso.

3.1. Aplicaciones

La necesidad de obtener soluciones a problemas físicos constituye un amplio campo de las posibles aplicaciones de la Diferenciación Automática. Estos problemas suelen ser de tipo no lineal, lo que complica el análisis y obtención de soluciones para ellos. La Diferenciación Automática permite implementar de forma más eficiente los métodos numéricos que necesitan del cálculo de una derivada, gradiente o matriz jacobiana. Por ejemplo, uno de los métodos numéricos para calcular soluciones de funciones con varias variables independientes es el método de Newton-Raphson.

Supongamos que queremos encontrar una solución de una función $f : [a, b] \subset \mathbb{R} \longrightarrow \mathbb{R}$ suficientemente regular. El método de Newton-Raphson aproxima la solución $x_0 \in \mathbb{R}$ mediante la

construcción de la sucesión

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

para $n \in \mathbb{N}$.

Este método se puede extender a funciones de varias variables. Si la función anterior f fuese una función que dependiese de una o más variables y quisiésemos calcular una solución en un conjunto de su dominio, el método de Newton permite aproximar dicha solución mediante la sucesión

$$J(x_n) \cdot s_n = -f(x_n)$$
$$x_{n+1} = x_n + s_n$$

para $n \in \mathbb{N}$ y siendo J la matriz jacobiana y s_n el paso utilizado en cada iteración.
Los dos métodos de Diferenciación Automática son válidos para calcular la matriz J, pero si la función f depende de una gran cantidad de variables, es más recomendable utilizar el enfoque regresivo. Un ejemplo más concreto de todo esto se puede consultar en [4].

En el entrenamiento de redes neuronales, la Diferenciación Automática regresiva es particularmente útil. La existencia de una gran cantidad de nodos hace que sea más beneficioso utilizar este enfoque en la minimización del error y de las funciones de coste de la red.

Los métodos numéricos para resolver EDO también suponen un campo de aplicación para la Diferenciación Automática. Los métodos de Euler Explícito, Euler Implícito y Runge-Kutta son algunos ejemplos de métodos para la resolución de EDO ordinarias más conocidos. En particular, el método de Runge-Kutta de orden cuatro (ver [7]) para el problema de valor inicial $y' = f(x, y)$, $y(x_0) = y_0$, viene dado por la ecuación

$$y_{i+1} = y_i + \frac{1}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4),$$

donde

$$k_1 = f(x_i, y_i) \tag{3.1}$$
$$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h) \tag{3.2}$$
$$k3 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h) \tag{3.3}$$
$$k_4 = f(x_i + h, y_i + k_3 h) \tag{3.4}$$

para $i \in 1, ...N, N = \frac{b-x_0}{h}, h > 0, b > x_0$.

Si quisiéramos aproximar la solución de una EDO de segundo orden

$$\begin{cases} y'' = f(x, y, y') \\ y(x_0) = y_0 \\ y'(x_0) = y'_0 \end{cases}$$

basta con realizar la transformación $y' = t$ para obtener un sistema de dos ecuaciones de primer orden.

$$\begin{cases} y'(x) = t, \ y(x_0) = y_0 \\ t'(x) = f(x, y, t), \ t(x_0) = t_0 \end{cases}$$

La Diferenciación Automática permite obtener directamente la derivada $y'(x) = t$ en $y_0 = y(x_0)$, simplificando el sistema y evitando tener que utilizar un método para resolver un sistema de ecuaciones diferenciales. En este caso, se podría aplicar directamente el método Runge-Kutta de orden cuatro a $t'(x) = f(x, y, t)$ en $t_0 = t(x_0)$.

Capítulo 4

Conclusiones y trabajo futuro

El objetivo principal de este trabajo era desarrollar una implementación para el método de Diferenciación Automática regresiva.
La idea fundamental para conseguirlo ha sido la construcción de un grafo computacional que permitiese ordenar los pasos que el algoritmo debía seguir. Además, las particularidades del proceso también han enriquecido este trabajo.

La Diferenciación Automática ha resultado ser una alternativa eficiente a la diferenciación simbólica. En líneas generales, reduce el coste computacional y elimina la necesidad de introducir las expresiones explícitas para las derivadas.

El método regresivo es más efectivo cuando las funciones dependen de una gran cantidad de variables independientes. Sin embargo, si esta cantidad es relativamente pequeña, es más aconsejable utilizar el método progresivo.

Por este motivo, ambos métodos se pueden aplicar en la resolución de modelos físicos que requieran del cálculo de gradientes o matrices jacobianas y suponen una mejor gestión de la capacidad de cálculo de la máquina.

Aún así, todo aprendizaje crea nuevas preguntas. Algunas de las posibles ideas para ampliar este trabajo se recogen en las siguientes líneas.

- La función para calcular las derivadas parciales funciona correctamente, pero no se han incluido todas las funciones elementales. Una posible línea de continuación podría ser perfeccionar el código incluyendo otros operadores básicos y funciones elementales. En particular, sería interesante añadir la función 'nthroot'. Esta función elemental se diferencia de las demás en que depende de dos argumentos. La complicación subyace en detectar respecto de cuál de los dos es necesario que derivar.

- El desarrollo de un programa para calcular derivadas de orden superior con Diferenciaicón Automática regresiva es una posible manera de continuar este trabajo, ya que el hecho de volver a utilizar el código de la implementación del método regresivo para calcular la derivada segunda no es posible. El motivo es que habría que proporcionar la expresión de la derivada primera, que es lo que en un principio se quería evitar.
- Como se había comentado en la Sección 3.1, el uso de la Diferenciación Automática regresiva en los procesos de optimización de redes neuronales es de gran utilidad. Un estudio más meticuloso y enfocado en este ámbito sería interesante.

Anexo I

Códigos asociados a la implementación en MATLAB

I.1. Ejemplos de código para la sobrecarga de algunos operadores

I.1.1. Sobrecarga del operador de la función coseno

```
function h = cos(x)
    if isa(x, 'valder')
        h = valder(cos(x.val), -sin(x.val)*x.der)
    end
end
```

I.1.2. Sobrecarga del operador suma

```
function h = plus(x,y)

    if ~isa(x,'valder')
        h = valder(x + y.val, y.der);

    elseif ~isa(y,'valder')
        h = valder(x.val + y, x.der);
    else
        h = valder(x.val + y.val, x.der + y.der);
    end
```

```
end
```

I.1.3. Sobrecarga del operador multiplicación

```
function h = mtimes(x,y)

    if ~isa(x,'valder')
        h = valder(x * y.val, x * y.der);

    elseif ~isa(y,'valder')
        h = valder(x.val * y, x.der * y);
    else
        h = valder(x.val * y.val, x.der * y + x.val * y.der);
    end
end
```

I.2. Función *analiza*

```
function arbol = analiza(formula, unkws)
    %% Descripcion
    % Dada una expresion matematica, devuelve su arbol de expresion
        en
    % forma de arbol de estructuras
    %% Sintaxis
    %   arbol = analiza(formula)
    %   arbol = analiza(formula, unkws)
    %% Entradas:
    %   formula (char) : Cadena de caracteres con la expresion
       matematica.
    %   unkws (cell, opcional) : Celda de cadenas con las
       incognitas de la expresion.
    %% Salida:
    %   arbol (struct) : Arbol de expresion en termino de
       estructuras.
    %% Descripcion detallada:
    %   La funcion utiliza expresiones regulares para analizar la
       formula
```

```
%   y descomponerla en partes que luego son organizadas en un
    arbol de expresion.
%   El patron de la expresion regular captura funciones
    matemoticas comunes,
%   numeros, variables, operadores y parentesis.
%% Ejemplo de uso:
%   arbol = analiza('sin(x) + cos(y) * 2', {'x', 'y'})

arguments
    formula (1,:) char
    unkws         cell = {'x', 'y'}
end
% Funciones matematicas conocidas
fcs = {'asinh', 'sinh', 'asin', 'sin', ...
       'acosh', 'cosh', 'acos', 'cos', ...
       'atanh', 'tanh', 'atan', 'tan', ...
       'asech', 'sech', 'asec', 'sec', ...
       'acoth', 'coth', 'acot', 'cot', ...
       'acsch', 'csch', 'hcsc', 'csc', ...
       'exp', 'log', 'sqrt', 'abs'};
% Patron para identificar funciones, numeros, variables y
    operadores
patron = ['(' strjoin(fcs, '|') '|' strjoin(unkws, '|') '|\d
    +\.\d+|\d+|[a-zA-Z]+\([^\)]*\)|[+\-*/^()]|[a-zA-Z_][a-zA-Z_0
    -9]*)'];
% Encuentra las partes coincidentes de la formula
partes = regexp(formula, patron, 'match');
% Crea el arbol de expresion basado en las partes
arbol = crea_arbol(partes);

function nodo = crea_arbol(partes)
    %% Descripcion
    % Construye un arbol de expresion identificando variables,
       funciones, parentesis y operadores.
    %% Sintaxis
    %   nodo = crea_arbol(partes)
    %% Entradas:
    %   partes (cell) : Lista de partes de una formula
```

```
   matematica.
%% Salida:
%   nodo (struct) : Nodo raiz del arbol de expresion.
%% Descripcion detallada:
%   La funcion recorre la lista de partes de la formula,
   identificando y
%   clasificando las incognitas, parentesis, funciones
   matematicas, y operadores
%   (potencia, multiplicacion, division, suma y resta).
   Despues de procesar
%   todos los elementos, construye un arbol jerarquico que
   refleja la estructura
%   de la formula.
%   Si al final solo queda un elemento, este se convierte
   en una hoja del arbol,
%   que puede ser un valor constante o un nodo que
   representa una expresion mas compleja.

for ui = 1:length(unkws)
    nodo = true;
    while ~isempty(nodo)
        [partes, nodo] = incognitas(partes, unkws{ui});
    end
end
nodo = true;
while ~isempty(nodo)
    [partes, nodo] = parentesis(partes);
end
for fi = 1:length(fcs)
    nodo = true;
    while ~isempty(nodo)
        [partes, nodo] = funcion(partes, fcs{fi});
    end
end
% Construccion jerarquica: ^, *, /, +, -
nodo = true;
while ~isempty(nodo)
    [partes, nodo] = operadores(partes, {'^'}, false);
```

```
        end
        nodo = true;
        while ~isempty(nodo)
            [partes, nodo] = operadores(partes, {'*', '/'}, true);
        end
        nodo = true;
        while ~isempty(nodo)
            [partes, nodo] = operadores(partes, {'+', '-'}, true);
        end

        % Si solo queda un parte, debe ser un operando (hoja)
        if isempty(nodo) && isscalar(partes)
            if isstruct(partes{1})
                nodo = partes{1};
            else
                nodo = struct('tipo', 'cte.', 'valor', partes{1});
            end
        end
    end

    function [partes, nodo] = incognitas(partes, ui)
        %% Descripcion
        % Busca una incognita en la lista de partes de la formula y
            la reemplaza
        % por un nodo en el arbol. Si encuentra la incognita, crea
           un nodo con su tipo y valor.
        %% Sintaxis
        %   [partes, nodo] = incognitas(partes, ui)
        %% Entradas:
        %   partes (cell) : Lista de partes de la formula, que
           contiene los diferentes
        %                   elementos de la expresion matematica.
        %   ui (char) : Incognita que se busca en la lista de
           partes.
        %% Salida:
        %   partes (cell) : Lista de partes modificada con el nodo
           reemplazado por
        %                   la incognita en caso de encontrarla.
```

```
    %   nodo (struct) : Nodo de tipo '?', que representa la
        incognita, si se encuentra.
    %% Descripcion detallada:
    %   La funcion busca la primera ocurrencia de la incognita
        especificada en la lista
    %   de partes de la formula. Si la encuentra, crea un nodo
        con tipo '?' y el valor
    %   de la incognita, luego reemplaza la parte
        correspondiente en la lista con este nodo.
    %   Si no se encuentra la incognita, la lista de partes
        permanece sin cambios y el nodo
    %   es vacio.

    ind = find(strcmp(partes, ui), 1);
    if isempty(ind)
        nodo = []; % No hay incognitas
        return;
    end
    nodo = struct('tipo', '?', 'valor', ui);
    partes = [partes(1:ind-1), {nodo}, partes(ind+1:end)];
end

function [partes, nodo] = funcion(partes, fci)
    %% Descripcion
    % Busca una funcion matematica en la lista de partes de la
        formula y, si esta seguida
    % de parentesis, crea un nodo con su tipo, valor (el
        contenido de los parentesis) y
    % una funcion anonima para su derivada.
    %% Sintaxis
    %   [partes, nodo] = funcion(partes, fci)
    %% Entradas:
    %   partes (cell) : Lista de partes de la formula, que
        contiene los diferentes
    %                   elementos de la expresion matematica.
    %   fci (char) : Nombre de la funcion matematica que se
        busca (por ejemplo, 'sin').
    %% Salida:
```

```
    %   partes (cell) : Lista de partes modificada con el nodo
       de la funcion reemplazado.
    %   nodo (struct) : Nodo con el tipo de la funcion, su
       valor (el contenido de los
    %                   parentesis) y su derivada.
    %% Descripcion detallada:
    %   La funcion busca la primera ocurrencia de una funcion
       matematica en la lista
    %   de partes. Si esta seguida de parentesis, se crea un
       nodo estructurado con:
    %   - Tipo: el nombre de la funcion.
    %   - Valor: el contenido de los parentesis.
    %   - Derivada: una funcion anonima para calcular la
       derivada simbolica.
    %   Luego, la funcion y su contenido se sustituyen en la
       lista de partes.
    %   Si no hay parentesis tras la funcion, se lanza un error
       .

    ind = find(strcmp(partes, fci), 1);
    if isempty(ind)
        nodo = []; % No hay funcion
        return;
    end

    if isstruct(partes{ind+1}) && strcmp(partes{ind+1}.tipo, '
       ()')
        nodo = struct('tipo', fci, 'valor', partes{ind+1}); %,
           'derivada', @(v) cos(v.valor)*v.derivada );
        partes = [partes(1:ind-1), {nodo}, partes(ind+2:end)];
    else
        error('No hay parentesis tras funcion.')
    end
end

function [partes, nodo] = parentesis(partes)
    %% Descripcion
    % Busca una funcion matematica en la lista de partes de la
```

```
    formula y, si encuentra
% que esta seguida de parentesis, crea un nodo
    correspondiente con el tipo de funcion
% y el valor de la expresion dentro de los parentesis.
%% Sintaxis
%   [partes, nodo] = funcion(partes, fci)
%% Entradas:
%   partes (cell) : Lista de partes de la formula, que
    contiene los diferentes
%                   elementos de la expresion matematica.
%   fci (char) : Nombre de la funcion matematica que se
    busca (por ejemplo, 'sin').
%% Salida:
%   partes (cell) : Lista de partes modificada con el nodo
    reemplazado por
%                   la funcion y su contenido, si se
    encuentra.
%   nodo (struct) : Nodo con el tipo de la funcion y su
    valor (expresion dentro
%                   de los parentesis), y la derivada
    asociada.
%% Descripcion detallada:
%   La funcion busca la primera ocurrencia de una funcion
    matematica en la lista
%   de partes y verifica que este seguida de parentesis. Si
     es asi, crea un nodo
%   con el tipo de la funcion, el valor de la expresion
    dentro de los parentesis y
%   una funcion anonima para su derivada. Si no se
    encuentra la funcion seguida de
%   parentesis, lanza un error.

%   El nodo creado se reemplaza en la lista de partes.
ind = find(strcmp(partes, '('), 1);
if isempty(ind)
    nodo = []; % No hay parentesis
    return;
end
```

```
        % Buscar el cierre correspondiente
        nivel = 0;
        for i = ind:numel(partes)
            if strcmp(partes{i}, '(')
                nivel = nivel + 1;
            elseif strcmp(partes{i}, ')')
                nivel = nivel - 1;
                if nivel == 0
                    % analizar el interior
                    interior = partes(ind + 1:i - 1);
                    valor_nodo = crea_arbol(interior);
                    nodo = struct('tipo', '()', 'valor', valor_nodo
                        );

                    % reemplazar parentesis con nodo
                    partes = [partes(1:ind-1), {nodo}, partes(i+1:
                        end)];
                    return;
                end
            end
        end
    end

    function [partes, nodo] = operadores(partes, operador,
        izqda_dcha)
        %% Descripcion
        % Busca un operador en la lista de partes de la formula y,
            segun el orden de
        % evaluacion (izquierda a derecha o derecha a izquierda),
            construye un nodo
        % con el operador y sus operandos a la izquierda y derecha.
        %% Sintaxis
        %   [partes, nodo] = operadores(partes, operador,
            izqda_dcha)
        %% Entradas:
        %   partes (cell) : Lista de partes de la formula, que
            contiene los diferentes
```

```
%                   elementos de la expresion matematica.
%   operador (cell) : Lista de operadores que se buscan (
    por ejemplo, {'+', '-', '*', '/'}).
%   izqda_dcha (logical) : Direccion de evaluacion del
    operador. Si es 'true',
%                          se evalua de izquierda a derecha
    ; si es 'false',
%                          de derecha a izquierda.
%% Salida:
%   partes (cell) : Lista de partes modificada con el nodo
    del operador y sus operandos.
%   nodo (struct) : Nodo con el tipo de operador y los
    nodos de los operandos izquierdo y derecho.
%% Descripcion detallada:
%   La funcion busca el primer operador de la lista de
    partes y, dependiendo de la
%   direccion de evaluacion (izquierda a derecha o derecha
    a izquierda), crea un nodo
%   que contiene el operador y los subarboles
    correspondientes a los operandos a la
%   izquierda y derecha del operador. El nodo se reemplaza
    en la lista de partes,
%   permitiendo que la expresion se estructure
    jerarquicamente.

es_char = cellfun(@ischar, partes); % Identificar celdas
    que contienen caracteres
% Aplicar ismember solo a las celdas con caracteres
partes_op = false(size(partes));
partes_op(es_char) = ismember(partes(es_char),operador);
if izqda_dcha
    ind = find(partes_op, 1, 'first');
else
    ind = find(partes_op, 1, 'last');
end

if isempty(ind)
    nodo = [];
```

```
            return;
        else
            % Construir nodo
            op = partes{ind};
            izqda = crea_arbol(partes(ind-1));
            dcha  = crea_arbol(partes(ind+1));
            nodo = struct('tipo', op, 'izqda', izqda, 'dcha', dcha)
               ;
            partes = [partes(1:ind-2), {nodo}, partes(ind+2:end)];
        end
    end
end
```

I.3. Función para el cálculo de las variables barra

```
    function barra = barras(expr,incog_var,punto)

% creacion de la estructura
estr = analiza(expr, incog_var);  % Este es el arbol de crea-arbol
%muestra(estr, 3);
estr.tipo
estr.dcha
estr.izqda

% propiedades para el grafo:
% arista = [1 2]
%          [1 3]
% donde 1 es el primer nombre
prop = struct();
prop.arista  = {}; %cell(aristas, 1) con vector 1x2
prop.hijo    = {}; %cell(nodos,   1) con vector 2x1
prop.padre   = {}; %cell(nodos,   1) con vector nx1
prop.ari     = {}; %cell(nodos,   1) con vector 2x1
prop.nombre  = {}; %cell(nodos,   1) con string
prop.tipo    = {}; %cell(nodos,   1) con string
prop.cte     = {}; %cell(nodos,   1) con double
```

```
% Creo una estructuctura para asignar el valor 'NaN' a las
   variablles x e y
cnan = num2cell(NaN(1, length(incog_var)));   % cell(1, incog) con
   NaN
prop.incog = cell2struct(cnan, incog_var, 2); % struct('x', NaN, 'y
   ', NaN, ...)

% creacion del grafo
prop = crea_grafo(estr, prop, []);     % por ahora tengo que
   entender esta funcion,
                                       % salto a la linea 153

m = cell2mat(prop.arista); % convertir a matriz
G = digraph(m(:,1), m(:,2));
[s, t] = findedge(G); % s = nodos origen, t = nodos destino,   % el
    grafo es el mismo pero las aristas no se guardan en el orden
   que heemos hecho
m_recuperado = [s, t]; % Matriz de indices de las aristas
G.Nodes.Name = prop.nombre;  % Los nodos del grafo ya estan
   guardado como una estructura, incorporamos un campo nuevo
close all
subplot(2,2,1)
plot(G)
title('Nombre')
% recorrido
regresivo  = toposort(G);  % ordena los nodos
progresivo = flip(regresivo); % el orden progresivo es el regresivo
    al reves
% evaluacion
val = zeros(size(prop.cte));
 for in = progresivo    % a cada nodo en sentido progresivo le
    ponemos su valor
     if strcmp(prop.tipo(in), 'cte.')
         val(in) = prop.cte{in};
     elseif endsWith(prop.tipo(in), '?') % variable
         val(in) = punto.(prop.tipo{in}(1:end-1));
     elseif strcmp(prop.tipo(in), '+')
         val(in) = sum(val(prop.hijo{in}));
```

```
      elseif strcmp(prop.tipo(in), '-')
          valh = val(prop.hijo{in});
          val(in) = valh(1)-valh(2);
      elseif strcmp(prop.tipo(in), '*')
          val(in) = prod(val(prop.hijo{in}));
      elseif strcmp(prop.tipo(in), '/')
          valh = val(prop.hijo{in});
          val(in) = valh(1)/valh(2);
      elseif strcmp(prop.tipo(in), '^')
          valh = val(prop.hijo{in});
          val(in) = valh(1)^valh(2);
      elseif strcmp(prop.tipo(in), 'sin')
          val(in) = sin(val(prop.hijo{in}));
      elseif strcmp(prop.tipo(in), 'cos')
          val(in) = cos(val(prop.hijo{in}));
      elseif strcmp(prop.tipo(in), 'abs')
          val(in) = abs(val(prop.hijo{in}));
      elseif strcmp(prop.tipo(in), 'log')
          val(in) = log(val(prop.hijo{in}));
      elseif strcmp(prop.tipo(in), 'sech')
          val(in) = sech(val(prop.hijo{in}));
      end
 end
G.Nodes.val = val;
subplot(2,2,2)
plot(G, 'NodeLabel', arrayfun(@num2str, G.Nodes.val, 'UniformOutput
   ', false))
title('Valor')

%Siguiente, hacer una estructura de operaciones, tipo:
% '+': @(in) sum(cellfun(@(x) x, val(prop.hijo{in})));

% parciales
parc = zeros(size(m_recuperado,1),1);
 for in = progresivo
    for ia = 1:length(prop.ari{in})
        v   = zeros(size(prop.hijo{in})); % valor de los hijos
        cte = zeros(size(prop.hijo{in})); % si son constantes los
```

```
   hijos
ar  = zeros(size(prop.hijo{in})); % indice de arista de los
    hijos
for ih = 1:length(prop.hijo{in})
    v(ih)   = val(prop.hijo{in}(ih));
    cte(ih) = strcmp(prop.tipo{prop.hijo{in}(ih)}, 'cte.');
    ar(ih)  = find(ismember(m_recuperado, [in, prop.hijo{in
       }(ih)], 'rows'));
end
if strcmp(prop.tipo(in), '+')
    parc(ar(1)) = 1;
    parc(ar(2)) = 1;
elseif strcmp(prop.tipo(in), '-')
    parc(ar(1)) =  1;
    parc(ar(2)) = -1;
elseif strcmp(prop.tipo(in), '*')
    parc(ar(1)) = v(2);
    parc(ar(2)) = v(1);
elseif strcmp(prop.tipo(in), '/')
    parc(ar(1)) = 1/v(2);
    parc(ar(2)) = -v(1)/v(2)^2;
elseif strcmp(prop.tipo(in), '^')
    parc(ar(1)) = v(2)*v(1)^(v(2)-1);
    parc(ar(2)) = v(1)^v(2)*log(v(1));
elseif strcmp(prop.tipo(in), 'sin')
    parc(ar(1)) = cos(v(1));
elseif strcmp(prop.tipo(in), 'cos')
    parc(ar(1)) = -sin(v(1));
elseif strcmp(prop.tipo(in), 'abs')
    parc(ar(1)) = sign(v(1));
elseif strcmp(prop.tipo(in), 'log')
    parc(ar(1)) = 1/v(1);
elseif strcmp(prop.tipo(in), 'sech')
    parc(ar(1)) = -sinh(v(1))/cosh(v(1))^2;
end
for ih = 1:length(prop.hijo{in})
    parc(ar(ih)) = parc(ar(ih))*(~cte(ih));
end
```

```
     end
 end

G.Edges.parc = parc;
subplot(2,2,3)
plot(G, 'EdgeLabel', arrayfun(@num2str, G.Edges.parc, '
   UniformOutput', false))
title('Parcial')

% barra
barra = zeros(size(prop.cte));

 for in = regresivo
     if isempty(prop.padre{in})
         barra(in) = 1;
     else
         barra(in) = 0;
         for ip = prop.padre{in}'
             ar  = find(ismember(m_recuperado, [ip, in], 'rows'));
                %Estamos buscando la posicion en M-recuperado de la
                 arista
             % seria mejor que cada nodo conociese de que arista
                viene
             % respecto de cada padre que tenga, en vez de buscar
             % es decir, cuando se crea cada nodo y se tiene el
                padre,
             % buscar esta informacion en m_recuperado. el problema
                 es que
             % m_recuperado se hace una vez terminado el proceso,
                por lo
             % menos, habria que hacerlo una sola vez para parcial
                y barra
             barra(in) = barra(in) + barra(ip)*parc(ar);
         end
     end
 end
G.Nodes.barra = barra;
subplot(2,2,4)
```

```
plot(G, 'NodeLabel', arrayfun(@num2str, G.Nodes.barra, '
   UniformOutput', false))
title('Barra')

function prop = crea_grafo(estr, prop, padre)

% Funcion para crear un grafo dirigido del arbol de evaluacion de
   una
% expresion (str) f
% Argumentos:
    % estr: arbol de la espresion(ya creado con 'analiza')
    % prop: estructura con las propiedades del grafo
        % aristas,hijos de cada nodo, padres de cada nodo, aristas
           que salen de
        % cada nodo, valor de cada nodo, tipo de cada nodo, nombre
           de cada
        % nodo, constantes de cada nodo (se evaluan igualemente en
           cada
        % nodo, aunque no sea un nodo cte, porque asi dejamos el
           sitio para
        % la evaluacion posterior)
    % padre: padre DE CADA ESTRUCTURA DEL ARBOL


    % Eliminar parentesis iniciales
    while strcmp(estr.tipo, '()')
        estr = estr.valor;
    end
    if strcmp(estr.tipo, '?') % solo variables
        if isnan(prop.incog.(estr.valor)) % por ejemplo, incog.x =
           NaN
            % En este caso no he pasado por esta incognita ninguna
               vez
            prop.tipo   = [prop.tipo; {[estr.valor estr.tipo]}]; %
               tipo, 'x?'
            prop.cte    = [prop.cte;  {NaN}]; % cte, NaN
            prop.nombre = [prop.nombre; ...
                {[num2str(length(prop.nombre)+1) ':␣' estr.valor
```

```
                estr.tipo]}]; % nombre, '6: x?'
        prop.incog.(estr.valor) = length(prop.nombre); %incog.x
            , 6
        prop.padre{length(prop.nombre)}  = [];    % Por ahora no
            le doy valor
                                                  % al padre
                                                     porque lo
                                                  % hago al
                                                     final para
                                                  % todos los
                                                     nodos a la
                                                  % vez
    end
    actual = prop.incog.(estr.valor);

elseif strcmp(estr.tipo, 'cte.')
    prop.tipo   = [prop.tipo; {estr.tipo}]; % tipo, 'cte.'
    prop.cte    = [prop.cte;  {str2double(estr.valor)}]; % cte,
        8.
    prop.nombre = [prop.nombre; {[num2str(length(prop.nombre)
       +1) ':␣' estr.tipo estr.valor]}];
    actual = length(prop.nombre);
    prop.padre{length(prop.nombre)}  = [];
else
    prop.tipo   = [prop.tipo; {estr.tipo}];
    prop.cte    = [prop.cte;  {NaN}];
    prop.nombre = [prop.nombre; {[num2str(length(prop.nombre)
       +1) ':␣' estr.tipo]}];
    actual = length(prop.nombre);
    prop.padre{length(prop.nombre)}  = [];
end
prop.hijo{actual} = [];
prop.ari{actual}  = [];
if ~isempty(padre)
    prop.arista = [prop.arista; {[padre,actual]}];
    prop.ari{padre}    = [prop.ari{padre};    length(prop.
       arista)];
    prop.hijo{padre}   = [prop.hijo{padre};   actual];
```

```
        prop.padre{actual} = [prop.padre{actual}; padre];
    end
    % Hijos
    if any(ismember({'^', '*', '/', '+', '-'}, estr.tipo))
        prop = crea_grafo(estr.izqda, prop, actual);
        prop = crea_grafo(estr.dcha,  prop, actual);
    elseif ~strcmp(estr.tipo, '?') && ~strcmp(estr.tipo, 'cte.')
        prop = crea_grafo(estr.valor, prop, actual);
    end
end

end
```

I.4. Código para la aplicación de los métodos de diferenciación del Capítulo 3 a la función $f(x, y, z) = 2 \cdot \cos(x \cdot z) + \sin(x + y)$.

```
format long

% Simbolico
syms x y z
fun = @(x,y,z) 2*cos(x*z) + sin(x+y);

tic
derivx = matlabFunction(diff(fun,x));
derivy = matlabFunction(diff(fun,y));
derivz = matlabFunction(diff(fun,z));

valor = {};
valor{1} = derivx(punto.x, punto.y,punto.z);
valor{2} = derivy(punto.x, punto.y);
valor{3} = derivz(punto.x, punto.z);
toc

disp(valor)

% Progresivo
x = valder(1, [1,0,0]);
```

```
y = valder(2, [0,1,0]);
z = valder(3, [0,0,1]);

tic
2*cos(x*z) + sin(x+y)
toc



%Regresivo
expr = '2*cos(x*z) + sin(x+y)';
incog_var = {'x','y','z'};
punto = struct('x', 1, 'y', 2,'z', 3);

tic
barra = barras(expr, incog_var, punto);
toc
```

Bibliografía

[1] T. M. Apóstol. *Cálculus: Cálculo con funciones de una variable, con una introducción al álgebra lineal*, volume 2. Editorial Reverté, 2ª edition, 1980.

[2] Q. Martín Martín, M. T. Santos Martín, and Y. d. R. d. Paz Santana. *Investigación operativa : problemas y ejercicios resueltos /.* Pearson, Madrid, 2005.

[3] MathWorks. Expresiones regulares. `https://es.mathworks.com/help/matlab/matlab_prog/regular-expressions.html`, 2024. Accedido el 24 de junio de 2025.

[4] R. D. Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM review*, 52(3):545–563, 2010.

[5] J. Nocedal and S. Wright. *Numerical Optimization.* Springer Series in Operations Research and Financial Engineering. Springer New York, New York, NY, 2nd ed. 2006. edition, 2006.

[6] V. P. Pérez. Introdución a la diferenciación automática a través de la programación orientada a objetos. En proceso de publicación: https://hdl.handle.net/10347/11381, 2024. Trabajo de Fin de Grado, Universidade de Santiago de Compostela.

[7] J. M. V. Rey. *Memoria de análisis numérico.* Tórculo Artes Gráficas, 1995.

[8] Wikipedia contributors. Binary expression tree — wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Binary_expression_tree`, Feb. 2024. Last edited on 24 February 2024. Accessed on 25 June 2025.