Consigna: Automatic differentiation in Matlab + aplicación a gradientes y Newton. El cálculo analítico de derivadas y su implementación es tediosa, y está sujeta a error. Se propone estudiar los sistemas de diferenciación automática para cálculo de gradientes y hessianos, implementar una versión simple e integrarlos a la optimización con métodos clásicos (gradiente, gradientes conjugados, Newton, etc.).

Automatic differentiation en Matlab y aplicaciones

Introducción

Existen dos métodos clásicos para el cálculo de derivadas: derivar simbólicamente la función obteniendo una expresión y evaluarla en un punto dado o utilizar derivación numérica (diferencias finitas). Pero estas técnicas presentan varios problemas. La diferenciación simbólica puede conducir a un código ineficiente y enfrenta la dificultad de convertir un programa en una sola expresión, mientras que la diferenciación numérica puede introducir errores de redondeo en el proceso de discretización. Ambos tienen problemas para calcular derivadas de mayor orden, donde la complejidad y los errores aumentan y son lentos para calcular derivadas parciales de una función con respecto a muchas entradas, como es necesario para los algoritmos de optimización basados en gradientes. La diferenciación automática o Autodiff, del inglés *Automatic Differentiation* (AD) es distinta de la diferenciación simbólica y la diferenciación numérica y resuelve estos problemas.

AD se basa en el hecho de que un programa que implementa una función se puede descomponer en una secuencia de operaciones aritméticas elementales (suma, resta, multiplicación, división, etc.) y funciones elementales (exp, log, sin, cos, etc.), siendo cada una trivialmente diferenciable. Estas derivadas parciales básicas, evaluadas utilizando los argumentos, se combinan de acuerdo a la regla de la cadena para calcular automáticamente derivadas de orden arbitrario (como gradientes, Jacobianos, polinomios de Taylor, etc.). Este proceso obtiene derivadas exactas (según la precisión numérica de la máquina), sin errores de aproximación como en la derivación numérica y debido a que la transformación simbólica ocurre sólo en el nivel más básico, DA evita los problemas computacionales inherentes al cálculo simbólico complejo. Por lo tanto es una forma precisa y eficiente de calcular derivadas que luego pueden usarse en otros métodos numéricos como el método de Newton o el método del gradiente.

Forward vs. Reverse

Para Autodiff es fundamental la descomposición de diferenciales proporcionada por la regla de la cadena. Para la simple composición $y = f(g(x)) = f(g(w_0)) = f(w_1) = w_2$, con $w_0 = x, w_1 = g(w_0)$ y $w_2 = f(w_1) = y$, aplicando regla de la cadena se tiene:

$$\frac{dy}{dx} = \frac{dy}{dw_1}\frac{dw_1}{dx} = \frac{df(w_1)}{dw_1}\frac{dg(w_0)}{dx}$$

Por lo general, se presentan dos formas distintas de derivación automática: acumulación *forward* (o hacia adelante) y acumulación *reverse* (o hacia atrás). La acumulación hacia adelante especifica que uno recorre la regla de la cadena de adentro hacia afuera (es decir, primero se calcula $\frac{dw_1}{dx}$ y luego $\frac{dy}{dw_1}$), mientras que en la acumulación hacia atrás se recorre del exterior al interior (primero $\frac{dy}{dw_1}$y luego $\frac{dw_1}{dx}$). Es decir,

- La acumulación hacia adelante computa la relación recursiva:

  $$\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$$

  con $w_n = y$, si se quiere computar la composición de *n* funciones.

- La acumulación hacia atrás computa la relación recursiva:

  $$\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$$

  con $w_0 = x$.

  Veamos un poco más en detalle cómo funciona cada sistema con la función de ejemplo

$$f(x_{1,} x_2) = x_1 x_2 + sen(x_1)$$

En la acumulación hacia adelante, primero se fija la variable independiente con respecto a la cual se realiza la diferenciación y se calcula la derivada de cada subexpresión de forma recursiva con la fórmula que vimos antes. Esto se puede generalizar a múltiples variables como un producto matricial de los jacobianos. En comparación con la acumulación hacia atrás, la acumulación hacia adelante resulta más natural y fácil de implementar, ya que el flujo de información de las derivadas coincide con el orden de evaluación. Cada variable *w* se enriquece

con su derivada $\dot{w}=\frac{dw}{dx}$ (almacenada como un valor numérico, no como una expresión simbólica, ya veremos más detalles de la implementación de esto en la próxima sección). Luego, las derivadas se calculan en sincronía con los pasos de evaluación y se combinan con otras derivadas mediante la regla de la cadena. Para calcular esto en nuestra función, renombramos las variables de la siguiente manera:

$$f(x_1, x_2) = x_1 x_2 + sen(x_1) = w_1 w_2 + sen(w_1) = w_3 + w_4 = w_5$$

La elección de la variable independiente respecto a la cual se va a diferenciar afecta los valores de inicialización $\dot{w}_1$ y $\dot{w}_2$. Entonces, si queremos, por ejemplo, calcular la derivada de f respecto a x_1, los valores de inicialización deben establecerse en:

$$\dot{w}_1 = \frac{dx_1}{dx_1} = 1 \text{ y } \dot{w}_2 = \frac{dx_2}{dx_1} = 0$$

Luego, usando la regla de la cadena tenemos:

$$\dot{w}_1 = 1 \text{ (valor inicial)}$$

$$\dot{w}_2 = 0 \text{ (valor inicial)}$$

$$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2$$

$$\dot{w}_4 = cos(w_1)\, \dot{w}_1$$

$$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$$

Y por lo tanto se puede calcular $\dot{w}_5$ propagando los valores de $\dot{w}_1$ y $\dot{w}_2$:

$$1 * x_2 + x_1 * 0 + cos(x_1) * 1 = x_2 + cos(x_1) = \frac{df}{dx_1}$$

Si quisiéramos calcular el gradiente de esta función de ejemplo, que requiere las derivadas de f no solo con respecto a x_1, sino también a x_2, se debe realizar el mismo procedimiento pero con valores iniciales $\dot{w}_1 = 0$ y $\dot{w}_2 = 1$.

Por otro lado, en la acumulación hacia atrás, primero se fija la variable dependiente a ser diferenciada y se calcula la derivada con respecto a cada subexpresión de forma recursiva con la fórmula correspondiente. La cantidad de interés es el *adjunto*, denotado con una barra ($\bar{w}$). Esto es la derivada de la variable dependiente elegida con respecto a una subexpresión w: $\bar{w} = \frac{dy}{dw}$.

La función de ejemplo es escalar y, por lo tanto, solo se necesita un valor inicial para el cómputo de la derivada y solo se necesita un recorrido para calcular el gradiente (de dos componentes). Esto es solo la mitad del trabajo en comparación con la acumulación hacia adelante, pero la acumulación hacia atrás requiere el almacenamiento de las variables intermedias w_i, así como las instrucciones que las produjeron en una estructura de datos conocida como lista de Wengert, que puede consumir una cantidad significativa de memoria si el grafo computacional es grande, aunque existen algunas optimizaciones.

Las operaciones para calcular la derivada usando acumulación hacia atrás son las siguientes (notar el orden inverso):

$$\bar{w}_5 = 1 \text{ (valor inicial)}$$

$$\bar{w}_4 = \bar{w}_5$$

$$\bar{w}_3 = \bar{w}_5$$

$$\bar{w}_2 = \bar{w}_3 * w_1$$

$$\bar{w}_1 = \bar{w}_3 * w_2 + \bar{w}_4 * cos(w_1)$$

La retropropagación o *backpropagation*, una técnica muy utilizada hoy en día en el aprendizaje automático, es un caso especial de acumulación hacia atrás. Al ajustar una red neuronal, la retropropagación calcula el gradiente de la función de pérdida con respecto a los pesos de la red para un solo ejemplo de entrada-salida, y lo hace de manera eficiente, a diferencia de un cálculo directo del gradiente con respecto a cada peso individualmente.

Implementación forward

En los archivos de MATLAB adjuntos (y disponibles también en github) está implementada una versión simple de diferenciación automática por medio de acumulación hacia adelante. La idea es que, si queremos calcular la derivada (o gradiente, si es una función escalar de varias variables) de f en x, en lugar de tomar como argumento solo el valor de x, el programa toma como argumento un vector de 1×2 que contiene el valor de x y el de su derivada (o su gradiente) y se modifican las operaciones básicas para producir el valor numérico de la derivada además del valor de la función. Veamos un ejemplo con una función lineal de una variable: $f(x) = 3x + 7$. Si $x = 3$, y calculamos $f(x)$, MATLAB devuelve 16. En cambio,

definiendo $x = [3,\ 1]$ (pues la derivada de x es 1) y $f(x) = 3x + [7, 0]$ (pues la derivada de la constante 7 es 0), tenemos ahora que calcular $f(x)$ devuelve $[16, 3]$, el valor de la función y el valor de su derivada en 3!

Para funciones no lineales, no es tan directo, hay que además redefinir las operaciones. Esto se hace siguiendo la regla de la cadena, por ejemplo, para la función $f(x) = x * x$, queremos que se compute $[3 * 3,\ 1 * 3 + 3 * 1] = [9,\ 6]$. Lo que hacemos entonces, para cada operación básica (división, multiplicación, seno, coseno, exponencial, logaritmo, potencias y raíz cuadrada) es en un archivo .m definirlas para que tomen uno de estos pares $[valor,\ derivada]$ y devuelvan el par $[valor,\ derivada]$ resultante de haber aplicado esa operación, siempre calculando la derivada con la regla de la cadena. Veamos un par de ejemplos:

- AD_cos toma un vector $[v,\ d]$ y devuelve el vector $[cos(v),\ -sin(v) * d]$, donde cos y sin son las funciones nativas de MATLAB para calcular senos y cosenos.
- AD_mult toma dos vectores $[v_1,\ d_1]$, $[v_2,\ d_2]$ y devuelve el vector $[v_1 * v_2,\ d_1 * v_2 + v_1 * d_2]$.

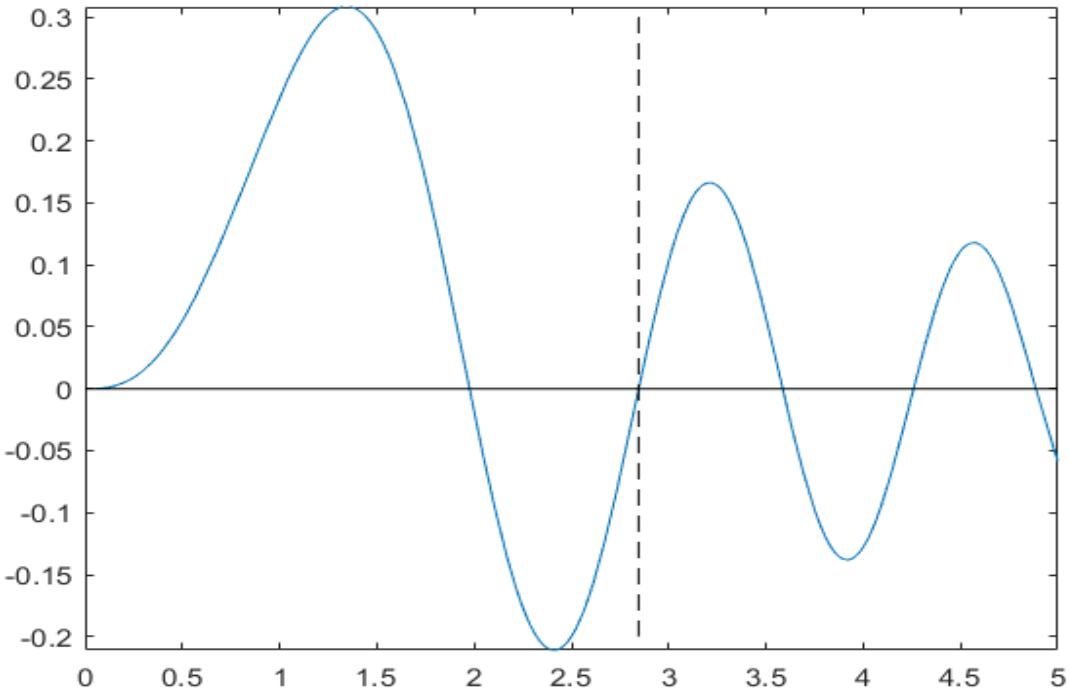
El resto de las operaciones están implementadas en los archivos AD_div, AD_exp, AD_log, $AD\_pot\_escalar$, AD_resta, AD_sin, AD_sqrt, AD_suma. Luego, si uno quiere, por ejemplo, la función $f(x) = sen(x * log(1 + x^2))$ y calcular $f'(a)$, se debe implementar $AD\_sin(AD\_mult(x,\ AD\_log([1,\ 0]+AD\_pot\_escalar(x,\ 2))))$ y pasarle como argumento $x = [a,\ 1]$. Observemos que en ningún momento se calcula la expresión simbólica para $f'(x)$, pero aún así se obtiene la evaluación numérica $f'(a)$.

Las operaciones están implementadas de forma que también sirven para funciones escalares de varias variables. En ese caso, en lugar del par $[valor,\ derivada]$, vamos a tener el par $[valor,\ gradiente]$, donde $gradiente$ es a su vez un vector. Luego, si queremos, por ejemplo, el gradiente de $f(x, y) = x * y + y^2$ en $(a,\ b)$, implementamos $AD\_mult(x,\ y) + AD\_pot\_escalar(y,\ 2)$, le pasamos como argumento $x = [a,\ [1,\ 0]]$ e $y = [b,\ [0,\ 1]]$ y nos va a devolver el vector $[f(a,\ b),\ [\frac{df}{dx}(a,\ b),\ \frac{df}{dy}(a,\ b)]]$.

Esta implementación sirve entonces para calcular derivadas de primer orden de funciones escalares (de una o varias variables). Veremos ahora ejemplos de métodos clásicos de optimización utilizando Autodiff para el cálculo de derivadas y gradientes y lo compararemos con la utilización de diferencias finitas. Veremos luego cómo se podría modificar la implementación para calcular derivadas de orden mayor y derivadas de funciones a varias variables.

Aplicación al método de Newton en una variable para encontrar raices

Consideramos la función $f(x) = e^{-\sqrt{x}} * sen(x * log(1 + x^2))$, que tiene varios ceros, pero que inicializando el método de Newton en $x = 1,5$, encuentra la raíz 2.8456 .

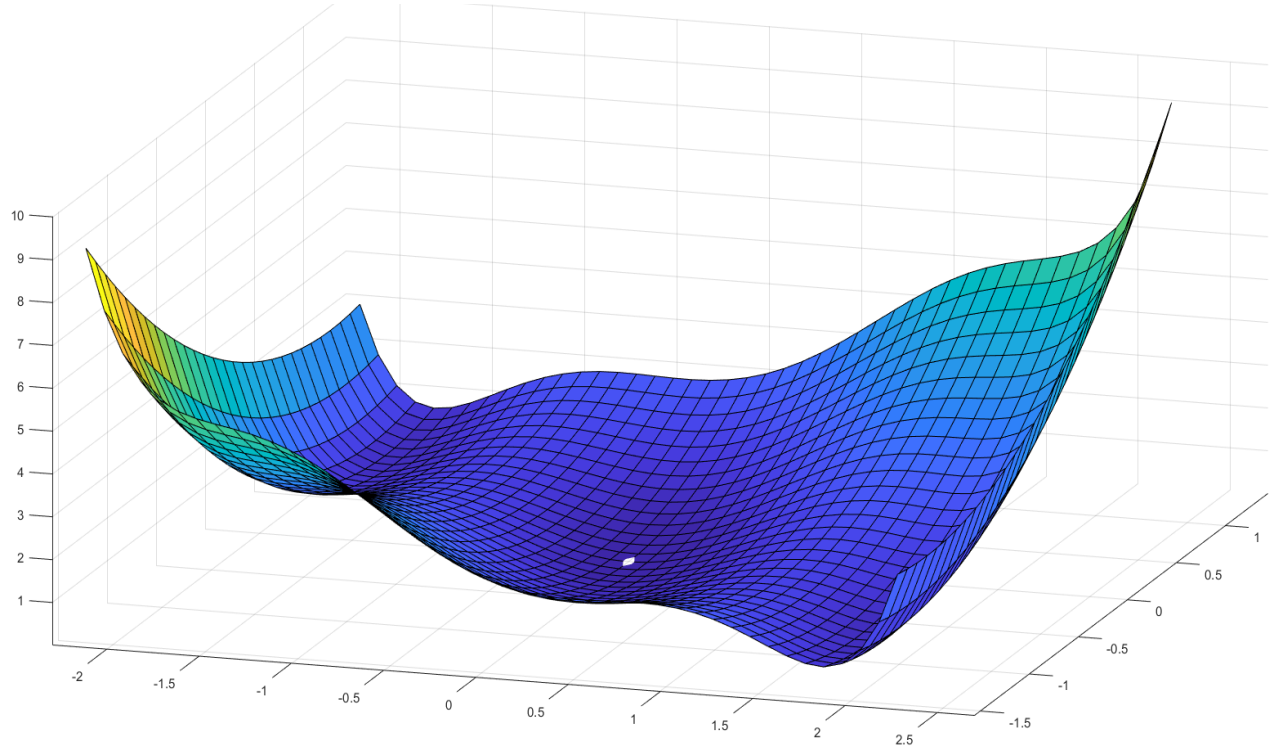


Recordemos que el método de Newton consiste básicamente en elegir un valor inicial x_0 y en cada paso calcular $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, por un cierto número de iteraciones o hasta que la

diferencia entre dos iteraciones sea suficientemente chica. Como se puede observar, para este método es necesario el cálculo de la derivada de f. En los archivos se encuentra implementado el método y la función de dos formas: una donde la derivada se aproxima utilizando la fórmula de derivadas centradas $f'(x) \approx [f(x+h) - f(x-h)]/2h$, haciendo $h \to 0$, y otra donde se utiliza Autodiff. Al correr el primer bloque del archivo "main.m" se puede observar, que si bien de ambas formas se llega a la misma raíz, utilizar Autodiff reduce el tiempo que tarda el método. Esto es porque para calcular las derivadas centradas, se deben hacer varias iteraciones, mientras que al implementar f con las operaciones modificadas para AD, la derivada se calcula de una vez al llamar a la función.

Aplicación al método del gradiente y gradientes conjugados

Consideremos ahora la función $f(x_1, x_2) = 2 * x_1^2 - 1.05 * x_1^4 + x_1^6 /6 + x_1 * x_2 + x_2^2$



Esta función (conocida como *three hump camel*) tiene mínimos locales y un mínimo absoluto en $(0,\ 0)$.

Los métodos del gradiente y de gradientes conjugados son algoritmos de optimización irrestricta y en su implementación incluyen el cálculo del gradiente de f en distintos puntos. En los archivos se encuentran implementados los métodos y la función de dos formas. Lo que cambia es cómo se calcula el gradiente: por un lado se calculan las derivadas parciales con el método iterativo de centradas y por otro lado, dada la función implementada con Autodiff, calcular el gradiente es solo tomar los valores a partir de la segunda coordenada de la función evaluada en un punto.

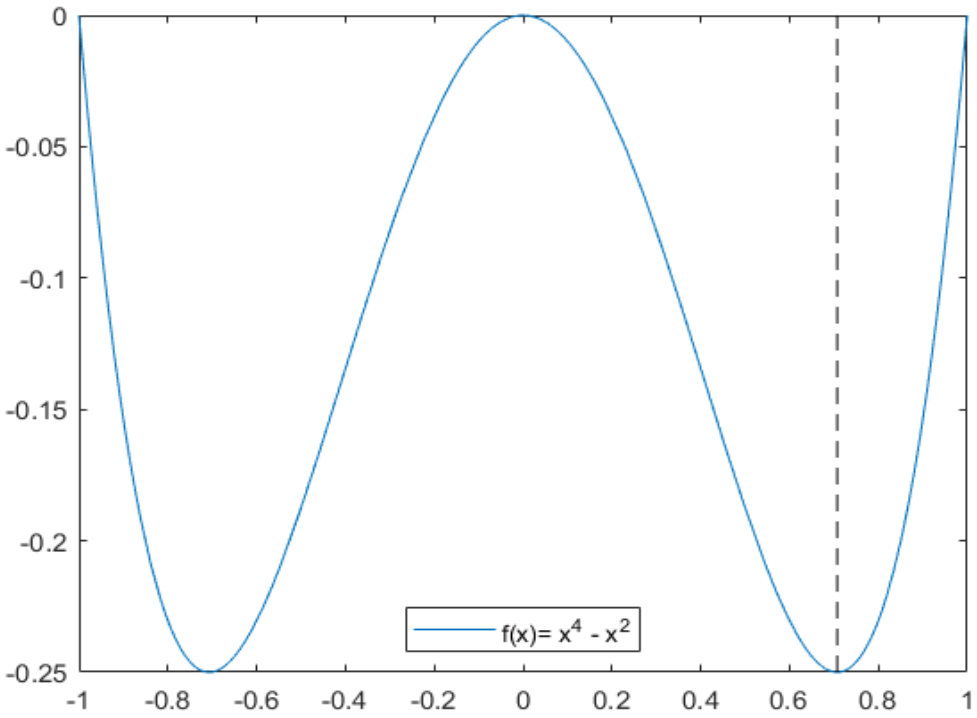
Corriendo el tercer y cuarto bloque del archivo "main.m" se pueden observar los resultados. En este caso, el error cometido con los métodos utilizando Autodiff para el cálculo del gradiente es levemente mejor que con diferencias finitas ya que a diferencia de éstas, Autodiff no involucra aproximaciones (salvo las inevitables por el hecho de que la computadora tiene precisión finita).

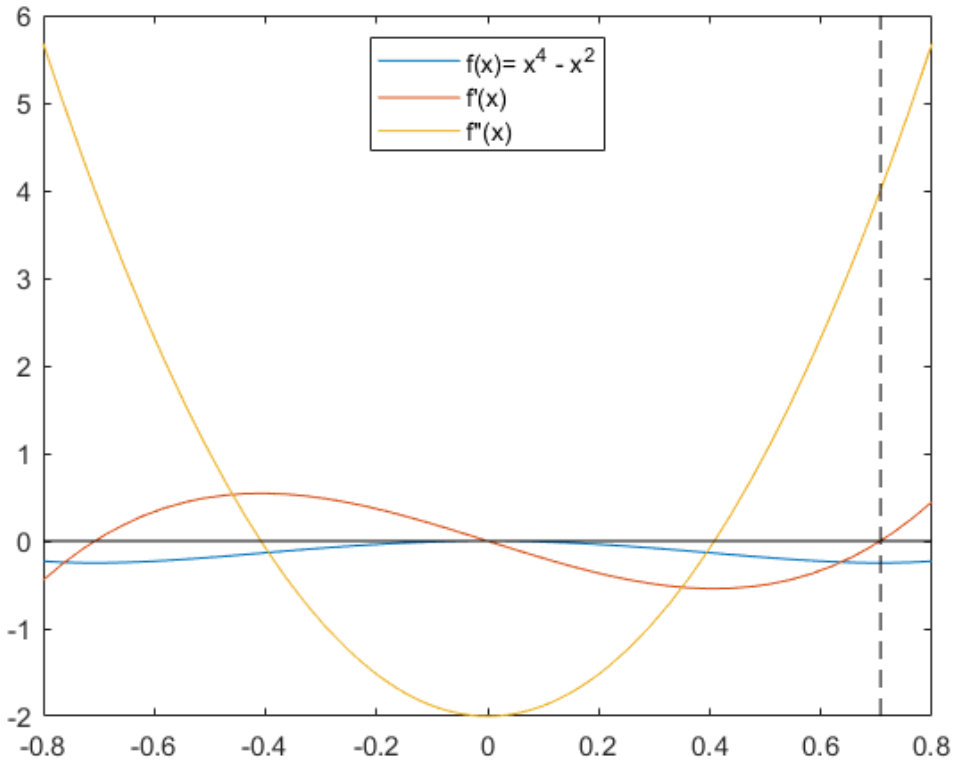
Aplicación al método de Newton para encontrar mínimos

El método de Newton también puede usarse para encontrar el máximo o mínimo de una función, encontrando los ceros de su primera derivada. Para esto entonces, es necesario calcular las derivadas de segundo orden de f. Acá la implementación de la forma en que veníamos haciendola se complica un poco. Supongamos que tenemos la función simple $f(x) = x^4 + x^2$, que tiene dos mínimos absolutos: $\frac{1}{\sqrt{2}}$ y $-\frac{1}{\sqrt{2}}$. Lo que podemos hacer es modificar la función $AD\_pot\_escalar$ para que si recibe algo de la forma $[v, d]$ funcione como antes devolviendo un vector con el valor de la función en v y el valor de su derivada evaluada en v y si recibe algo de la forma $\{[v, d_1], d_2\}$ devuelva un vector con los valores de la derivada primera evaluada en v y de la derivada segunda evaluada en v, como si hubiera "vuelto a derivar". Este sería, aplicando la regla de la cadena como corresponde, el vector $n * AD\_pot\_escalar([v, d_1], n - 1) * d_2$. Veamos un ejemplo con números, para entender mejor. Supongamos que queremos calcular la derivada segunda de x^4 en $x = 3$. Luego, al pasarle a la función $AD\_pot\_escalar(x, 4)$, como argumento $\{[3, 1], 1\}$, se

calcula: 4 * $AD\_pot\_escalar$([3, 1], 4-1) * 1 = 4 * [3^{4-1} , (4-1) * 3^{4-2} * 1] * 1 = [108, 108], que es justamente el valor de las derivadas primera y segunda de x^4 en 3.

De esta forma, ejecutando el segundo bloque del archivo "main.m" se puede observar que iniciando en $x = 3$, tanto el método con Autodiff como con derivadas centradas encuentran el mínimo $\frac{1}{\sqrt{2}}$, pero con Autodiff es mucho más rápido.





Conclusiones

Esta implementación, sin embargo, es muy difícil de generalizar para funciones más complejas. Lo que se propone en [1], en lugar de utilizar vectores e implementar las operaciones para Autodiff con otro nombre, es implementar una clase que represente estos vectores y sobreescribir las funciones nativas de MATLAB para los elementos de esa clase. De esta forma, se puede aprovechar mejor código de MATLAB ya existente y extenderlo simplemente cambiando la clase de la variable de entrada. Esto también permite calcular Jacobianos, sin hacer prácticamente ningún cambio. De todos modos, si bien definir la clase y sobreescribir operadores permite calcular derivadas de cualquier orden para funciones escalares de una variable, no es la forma más eficiente de hacerlo y en [1] se propone también otra solución utilizando series de Taylor truncadas. Para derivadas de mayor orden de funciones de varias variables, se presentan además desafíos prácticos debido a la gran cantidad de derivadas parciales que son necesarias calcular.

Pero, si bien la implementación puede ser un poco más engorrosa que la de diferencias finitas, Autodiff resulta mucho más estable, ya que no se realizan aproximaciones (salvo las inherentes a trabajar con una computadora). Por otro lado, AD requiere menos memoria que la diferenciación simbólica, donde deben almacenarse grandes expresiones, y también es menos costosa en tiempo. Esto hizo que, en el último tiempo, se vuelva una herramienta muy útil en áreas como Machine Learning donde se requiere calcular una gran cantidad de derivadas para optimizar funciones de pérdida y por lo tanto se necesita un método eficiente y estable.

Referencias

[1] Neidinger, Richard D. (2010). *"Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming*". SIAM Review. 52 (3): 545–563.
https://www.neidinger.net/SIAMRev74362.pdf

[2] https://en.wikipedia.org/wiki/Automatic_differentiation

[3] http://www.autodiff.org/