*Revista Tecnocientífica URU*
Universidad Rafael Urdaneta
Facultad de Ingeniería
Nº 5 Julio - Diciembre 2013
Depósito legal: ppi 201402ZU4464
ISSN: 2343 - 6360

# Biblioteca de diferenciación automática para la máquina virtual de Java

**María E. Portillo Montiel[1], Nelson Arapé[2] y Gerardo Pirela Morillo[1]**

[1]Departamento de Computación. Facultad Experimental de Ciencias.
Universidad del Zulia. Maracaibo, Estado Zulia, Venezuela.
mariaeportillo@fec.luz.edu.ve, gpirela@fec.luz.edu.ve

[2]Instituto de Cálculo Aplicado. Facultad de Ingeniería. Universidad del Zulia.
Maracaibo, Estado Zulia, Venezuela. narape@ica.luz.edu.ve

Recibido: 27/07/2012 Aceptado: 21/06/2013

**Resumen**

Los métodos de optimización juegan un papel fundamental en el diseño de sistemas complejos en ingeniería. Por lo general, los métodos de optimización iterativos requieren el cálculo repetitivo de la matriz Jacobiana, elemento crítico en lo que a eficiencia se refiere. Existen diferentes maneras de obtener la derivada: derivación analítica, requiere gran inversión de tiempo y esfuerzo; diferenciación numérica, método propenso a errores de truncamiento; y diferenciación automática, que permite obtener la derivada de funciones representadas en un programa, implica un tiempo de cómputo razonable y el resultado es preciso. Actualmente, no se dispone de herramientas de diferenciación automática para la máquina virtual de Java. El objetivo de la presente investigación fue desarrollar una Biblioteca de Diferenciación Automática para la máquina virtual de Java. Inicialmente se procedió al estudio de los fundamentos de la diferenciación automática, seleccionando un lenguaje de programación adecuado para la codificación de las clases que conforman la biblioteca. Finalmente, y posterior a la selección de programas diferenciables, se ejecutaron diversas pruebas a fin de documentar la exactitud de los resultados. Al finalizar la presente investigación se cuenta con una biblioteca de diferenciación automática que puede ser incorporada a cualquier proyecto que lo requiera.

**Palabras clave:** Diferenciación automática, Java, optimización, Jacobiano

# Automatic differentiation library for the Java virtual machine

**Abstract**

Optimization methods play an important role in the design of complex engineering systems. Generally, iterative optimization methods require repetitive computing of the Jacobian matrix, which is critical for efficiency. There are different ways to compute the derivative of a function: analytic differentiation (which supposes great manual effort), numerical differentiation (which is susceptible to approximation and round-up/truncation errors), and automatic differentiation (which allows to compute the derivative of a function coded in a computer program and thus incurs in reasonable computing time and accurate results). There are no readily-available tools for automatic differentiation in Java currently. The main purpose of this research was to develop such a tool in the form of a library for the Java Virtual Machine. First, the authors documented the basics of automatic differentiation in

order to select the most appropriate and suitable tool to code the classes that would eventually conform the library. Then, a few differentiable programs were selected to test the library and document the accuracy and precision of the results. Finally, the authors demonstrated that the resulting automatic differentiation library may be readily used as integral part of any optimization project that so requires, without increasing the computational complexity of the resulting program, yielding a level of accuracy and precision matching and even surpassing that of analytic and numerical methods, and avoiding repetitive computation of the Jacobian matrix.

**Key words:** Automatic differentiation, Java virtual machine, Java library, optimization, Jacobian matrix

## Introducción

La derivada de una función en un punto mide el coeficiente de variación o cambio de dicha función en este punto. Es una herramienta de cálculo fundamental en distintas disciplinas, tales como electricidad, electrónica, termodinámica, mecánica, economía y biología en las cuales resulta de vital importancia no sólo saber que determinada magnitud o cantidad varía respecto a otra, sino conocer cuán rápido se produce dicha variación. Por ejemplo, en el campo de la ingeniería, muchas leyes y otras generalidades se basan en predecibles en las cuales el cambio mismo se manifiesta en el mundo físico, tal es el caso de la segunda ley de Newton, que no está dirigida en términos de la posición de un objeto, sino en su cambio de posición con respecto al tiempo [1].

Existen diferentes maneras de obtener la derivada de una función:

A. Diferenciación Analítica, se utilizan tablas y técnicas bien conocidas del cálculo para obtener la derivada de una función de acuerdo con su definición. Esta es la estrategia de diferenciación más exacta y la precisión de los resultados dependerá de la precisión misma del computador en que se programen tanto la función original como su derivada. Sin embargo, en muchas aplicaciones de ingeniería, como simulación y optimización, es necesario conocer la derivada de funciones muy complejas, cuya solución manual supone gran inversión de tiempo y de esfuerzo en el desarrollo de cualquier proyecto, además de ser una fuente importante de errores.

B. Diferenciación Simbólica, se utilizan técnicas de manipulación de cadenas para transformar la representación textual de la función en la correspondiente representación textual de su derivada; por lo general, se usan técnicas de análisis sintáctico (e.g., traducción dirigida por sintaxis) o lenguajes de programación simbólica para lograr tal transformación. En teoría, la exactitud de los resultados es la misma que la Diferenciación Analítica; sin embargo, los lenguajes de manipulación simbólica pueden llegar a dar tiempos de respuesta altos y devolver la función diferenciada en una expresión innecesariamente compleja, con descomposición de término y sin reducciones o simplificaciones que un humano implementando técnicas analíticas sí pueda aplicar durante los pasos intermedios para controlar la complejidad de la función resultante. Este crecimiento de complejidad en la expresión resultante impacta directamente la precisión de los resultados del programa que calcule dicha expresión (debido a la propagación de errores de cálculo asociado a todo computador) y aumenta proporcionalmente a la composición de funciones en la expresión original (debido a que la diferenciación simbólica aplicaría recursivamente la regla de la cadena de derivación) y a la cantidad de variables de la expresión original (debido a la necesidad de calcular el Jacobiano – o matriz de derivadas parciales).

A. Diferenciación Numérica, Se parte de tablas de valores de la función evaluada en distintos puntos y se calcula, a través de técnicas de aproximación numérica, el valor estimado de la derivada de los mismos puntos. Con técnicas de ajuste de curvas y de interpolación, se puede estimar la derivada de la función en puntos que no estén incluidos en las tablas de valores originales. Sin embargo, a pesar de que los métodos numéricos disponibles se pueden implementar con algoritmos de complejidades computacionales (tanto temporal como espacial) aceptables: polinómicas de bajo grado respecto al tamaño de las tablas de entrada, estos algoritmos son susceptibles a errores de aproximación y precisión inherentes

a no utilizar la función analítica original (sino su representación tabular) y a la propagación natural del error de cálculo asociado a cualquier computador. Esto se ve exacerbado cuando la dimensionalidad de la función original aumenta debido al cálculo de las derivadas parciales.

B. La Diferenciación Automática, Se calcula la derivadas (incluyendo las parciales) de una función en paralelo a la ejecución de un algoritmo que implementa de dicha función. Se basa en el hecho de que cualquier programa de computación que implemente una función vectorial se puede descomponer en una secuencia de asignaciones elementales, siendo cada una trivialmente diferenciable. Combina entonces técnicas simbólicas y técnicas numéricas, evadiendo la transformación simbólica (por lo cual no resulta de evaluaciones de expresiones innecesariamente complejas) así como la inclusión de errores por aproximación; las complejidades computacionales son las mismas que las del algoritmo que implementa la función original, aumentada apenas en un término constante: la cantidad de variables para el cálculo del Jacobiano, la cual es conocida a priori y por lo tanto puede ser despreciada en la notación asintótica de dichas complejidades. Quedando los resultados de esta técnica susceptibles solamente a los errores de propagación de cálculo asociados a cualquier computador, los cuales son bien conocidos y asumidos o tratados en aplicaciones de ingeniería.

Diversas investigaciones se han orientado al desarrollo de herramientas de diferenciación automática; tal es el caso de OpenAD/F [6] y ADIFOR [2] que permiten la evaluación de derivadas definidas por un programa en Fortran; ADIC, que permite aplicar la funcionalidad de la diferenciación automática a programas escritos en ANSI-C [3]; ADOL-C, un paquete para diferenciación automática de algoritmos escritos en C/C++ [4]; ADiJAC, herramienta de diferenciación automática de archivos .class de Java [5], de la cual no es posible obtener el código fuente.

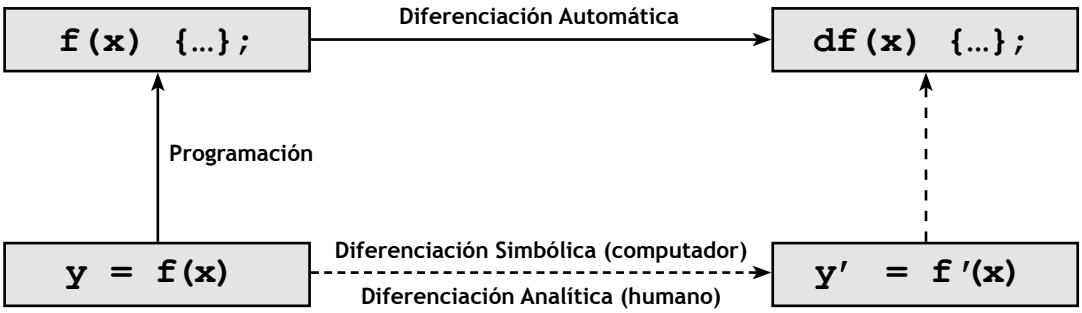
Ello representa un problema por cuanto, en lo últimos años, Java se ha convertido en uno de los lenguajes más utilizados por sus numerosas ventajas para el desarrollo de grandes aplicaciones. Mediante el uso de la máquina virtual de Java (JVM) se ofrece la independencia de plataforma y ello mejora la portabilidad. Además, este lenguaje incorpora una serie de tecnologías tales como manejo de excepciones, subprocesamiento múltiple, invocación de objetos remotos (RMI), servlets.

El objetivo de esta investigación fue desarrollar una solución en lenguaje de programación Java que permita la diferenciación automática. El presente documento presenta los resultados de una solución desarrollada para incorporar diferenciación automática para la JVM.

## Técnicas de Diferenciación Automática

La diferenciación automática corresponde a una técnica de diferenciación que consiste en calcular tanto el valor de una función en un punto dado como la derivada de dicha función, a partir de la definición en un lenguaje de programación de dicha función; la figura 1 resume este proceso en contraste con la derivación analítica y simbólica.

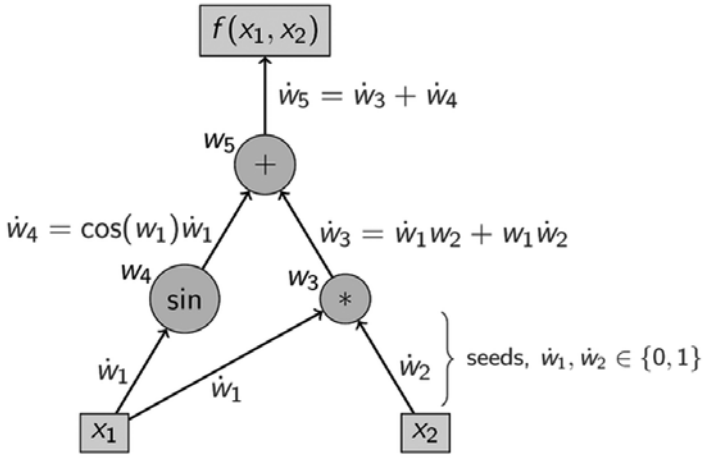
**Figura 1. Diferenciación automática**



Existen dos técnicas para implementar la diferenciación automática:

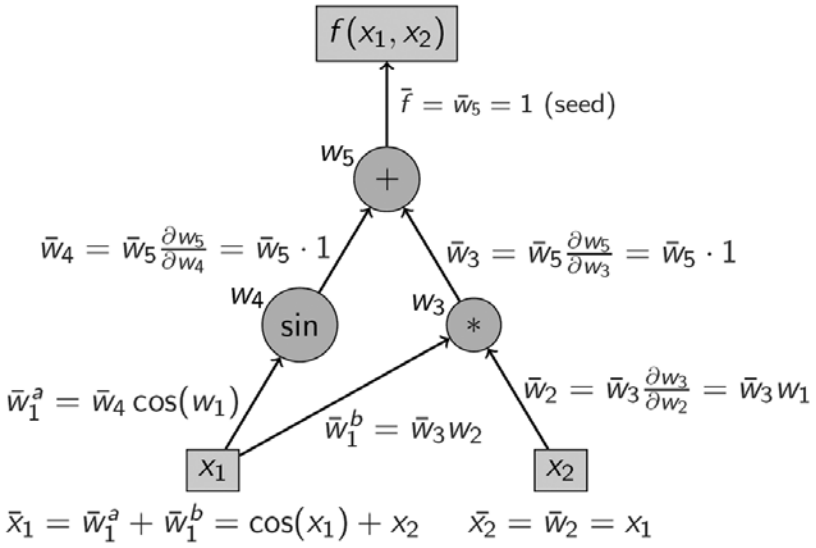
*Forward mode:* va acumulando las derivadas intermedias obtenidas a través de cálculos repetitivos desde las entradas hasta las salidas. Esto se logra asociando un objeto *gradiente* a cada una de las variables miembro de la función que se desea diferenciar, de tal manera que las derivadas parciales de la función se van acumulando en dicho objeto, esto permite que los valores intermedios de las mismas estén disponibles a lo largo de toda la ejecución del programa; al finalizar se tendrá una variable que almacena el valor de la función evaluada en el punto dado al iniciar el proceso y el gradiente de la función (véase figura 2).

**Figura 2. Propagación hacia delante de los valores de las derivadas intermedias**




*Reverse mode:* acumula las derivadas desde la salida hasta la entrada, en dirección opuesta al *forward mode*. Este requiere una inversión del modo de ejecución del programa original, puesto que es preciso completar la evaluación de la función para poder comenzar el cálculo de la derivada; adicionalmente, deben ser almacenados todos los valores obtenidos a partir los cálculos intermedios efectuados, ya que son requeridos para el cálculo de la derivada de la función (véase figura 3).

**Figura 3. Propagación hacia atrás de los valores de las derivadas intermedias**



Se seleccionó el *forward mode*, como técnica para implementar diferenciación automática por su aparente simplicidad de aplicación y porque, en contraste con lo establecido en la teoría del *reverse mode*, la cantidad de recursos necesarios es menor (la verificación de la validez de esta sentencia escapa al alcance de la presente investigación).

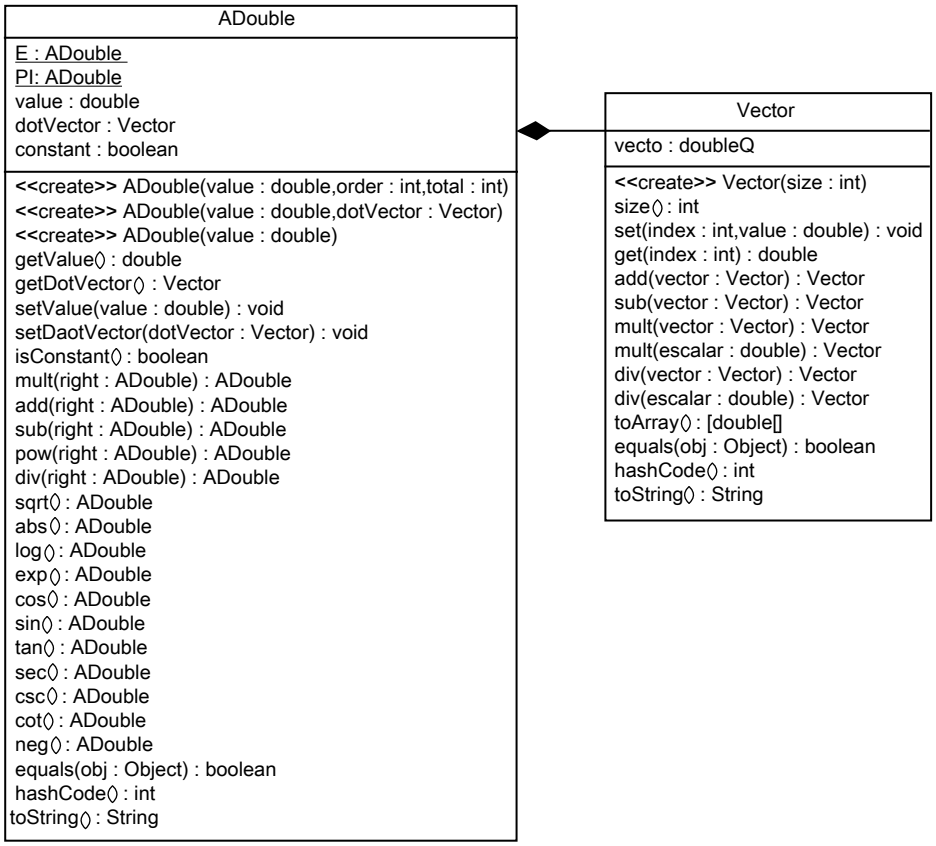
## Diseño de la biblioteca

En primera instancia, se desarrolló una clase llamada ADouble. Esta clase tiene como atributos: un double (*value*); para almacenar el valor correspondiente a la variable que se está declarando y un valor de tipo Vector (*dotVector*), para almacenar los valores correspondientes a las derivadas parciales de la función. Adicionalmente han sido definidos dos parámetros de tipo ADouble constantes, cuyos valores corresponden a Math.E y Math.PI respectivamente.

En la Figura 4, puede observarse el diagrama de clases conformado por la clase ADouble y la clase Vector; nótese que la clase Vector corresponde a una clase auxiliar definida por el autor, esta contiene un arreglo de doubles y los respectivos métodos para la ejecución de algunas operaciones aritméticas sobre vectores: adición, sustracción, producto y división, requeridas para acumular las respectivas derivadas parciales de $f(x)$. De esta manera, es posible, desde la clase ADouble, derivar en forma automática una función multivariable.

El comportamiento de la clase viene dado por los métodos diseñados para cada una de las operaciones matemáticas soportadas. Cada método calcula, no sólo el valor de la función a partir de los datos previamente almacenados en el atributo *value*, sino también las derivadas parciales con respecto a cada una de las variables miembro de la función, éstas se van acumulando en el atributo *dotVector*.

**Figura 4. Clases ADouble y Vector**



A continuación se describe un ejemplo de código que utiliza la clase ADouble para calcular la función $f(x) = \cos(x_1 + a)/x_2$ y sus respectivas derivadas parciales (Figura 5).

**Figura 5. Código ejemplo del uso de la clase ADouble**

```
1.   ADouble a = new ADouble(1);
2.   ADouble X1 = new ADouble(2.141592654,1,2);
3.   ADouble X2 = new ADouble(2,2,2);
4.   ADouble term1 = X1.add(a);
5.   ADouble term2 = term1.cos();
6.   ADouble term3 = term2.div(X2);
7.   System.out.println("a+X = "+ term1.getValue() +
                 " DX= " + term1.getDotVector().get(0)+
                 " Da= " + term1.getDotVector().get(1));
8.   System.out.println("X2 = "+ X2.getValue() +
                          " DX2 = " + X2.getDotVector() .get (1));
9.   System.out.println("cos(a+X1) = "+term2.getValue() +
                          " DX1="+ term2.getDotVector().get(0));
10.  System.out.println("cos(a+X1)/X2 = "+ term3.getValue() +
                          "DX1=" +term3.getDotVector().get(0) +
                          "DX2=" +term3.getDotVector().get(1));
```

Para ilustrar el ejemplo se han asignado un conjunto de valores, una constante $a = 1$, y dos variables $x_1 = 2.141592654$ y $x_2 = 2$.

Inicialmente debe instanciarse un objeto de la clase ADouble por cada miembro de la función (líneas 1-3). Sin embargo, la definición varía si corresponde a una variable o a una constante; en la línea 1 puede observarse la definición de una constante puesto que sólo es necesario indicar su valor numérico. En contraste, las líneas 2 y 3 definen las variables involucradas, para ello es necesario indicar el valor numérico, la posición dentro de la función (primera para $x_1$ y segunda para $x_2$), además del total de variables que participan en la función, dos en este caso.

Una vez definidas las constantes y variables, se procede a efectuar las operaciones correspondientes (invocando los métodos definidos en la clase ADouble) para obtener no sólo las derivadas parciales sino el valor de la función. Para mayor claridad, la función se construyó en partes: $term1$, $term2$ y $term3$, siendo este último el objeto que almacena los valores finales. La variable $term1$ almacena el valor de $(x_1 + a)$ y su correspondiente derivada; $term2$ almacena el valor y la derivada de $\cos(term1) \equiv \cos(x_1 + a)$; finalmente $term3$ calcula el valor y la derivada para $term2/x_2$, equivalente a $\cos(term1)/x_2 \equiv \cos(x_1 + a)/x_2$, que corresponde a la función $f(x)$.

Las líneas 7-10 imprimen por pantalla los valores correspondientes a cada cálculo efectuado para construir $f(x)$; lo que demuestra que es posible conocer los valores intermedios en cualquier momento de la ejecución de las operaciones necesarias para obtener los valores definitivos de $f(x)$ y $f'(x)$.

Como puede observarse en la línea 10, se imprime el valor de la función y dos valores adicionales, estos corresponden a las derivadas parciales con respecto a $x_1$ y $x_2$ respectivamente, que han sido calculadas y almacenadas en el atributo *dotVector*.

## Comparación de técnicas de diferenciación

Con la finalidad de comparar la exactitud de los valores obtenidos mediante la derivación automática en Java, se calculo la derivada de la función Branin-Hoo en forma analítica, para posteriormente programarla en Java. Así mismo, se programo la derivada numérica mediante la implementación de diferencias divididas hacia delante, hacia atrás y centrales, los resultados obtenidos con los 5 métodos se muestran en la Tabla 2.

Se ejecutaron las pruebas con 7 puntos; los primeros 3 puntos corresponden a los mínimos de la función evaluada, para los cuales se esperan valores muy cercanos a cero de sus respectivas derivadas, en correspondencia con la definición de puntos mínimos. Los 4 puntos restantes fueron escogidos puesto que, a partir de ellos, se tiene la certeza de que un buen algoritmo de optimización converja a un punto mínimo (en la siguiente sección de detallan las pruebas realizadas).

En la Tabla 2 se han listado los valores correspondientes a la evaluación de la función en cada punto y su correspondiente derivada. Como puede observarse, el valor de la función es el mismo para todos los métodos; por el contrario, el valor de la derivada varia de acuerdo al método empleado. En todos los casos el valor del gradiente obtenido empleando la diferenciación automática en Java es casi exacto con respecto al valor obtenido en forma analítica, lo cual deja claro que con esta técnica se garantiza exactitud en los resultados obtenidos, minimizando el error absoluto a valores iguales o muy cercanos a cero como se ilustra en la Tabla 3.

**Tabla 2. Valores de la función Branin-Hoo con Derivación analítica, automática y numérica**

| Método | F(x) | Gradiente |
|---|---|---|
| **Punto: [9,42477796076938; 2,475]** | | |
| Analítica | [0,39788735772973816] | [-3,5277589537225827E-15; 0,0] |
| ADouble | [0,39788735772973816] | [-3,5277589537225827E-15; 0,0] |
| Dif, Div, Adelante | [0,39788735772973816] | [5,5125787268295305E-5; 1,000000082740371E-5] |
| Dif, Div, Atrás | [0,39788735772973816] | [-5,512574285937432E-5; -1,000000082740371E-5] |
| Dif, Div, Central | [0,39788735772973816] | [4,4408920985006255E-11; 0,0] |
| **Punto: [3,141592653589793; 2,25]** | | |
| Analítica | [0,39851235772973814] | [-0,03899296105751471; -0,049999999999998934] |
| ADouble | [0,39851235772973814] | [-0,038992961057516096; -0,05000000000000071] |
| Dif, Div, Adelante | [0,39851235772973814] | [-0,0389388041976968; -0,04999000000061037] |
| Dif, Div, Atrás | [0,39851235772973814] | [-0,03904711796121063; -0,05001000000226518] |
| Dif, Div, Central | [0,39851235772973814] | [-0,0389929610794537; -0,0500000000014378] |
| **Punto: [-3,141592653589793; 12,275]** | | |
| Analítica | [0,39788735772973816] | [1,175919651240861E-15; 0,0] |
| ADouble | [0,39788735772973816] | [1,175919651240861E-15; 0,0] |
| Dif, Div, Adelante | [0,39788735772973816] | [1,0576602926803956E-4; 1,000000082740371E-5] |
| Dif, Div, Atrás | [0,39788735772973816] | [-1,0576615694368739E-4; -1,000000082740371E-5] |
| Dif, Div, Central | [0,39788735772973816] | [-1,27675647831893E-10; 0,0] |
| **Punto: [0,0; 10,0]** | | |
| Analítica | [35,602112642270264] | [12,732395447351628; 8,0] |
| ADouble | [35,602112642270264] | [12,732395447351628; 8,0] |
| Dif, Div, Adelante | [35,602112642270264] | [12,732362432643638; 8,000009999165059] |
| Dif, Div, Atrás | [35,602112642270264] | [12,732428461958987; 7,999989999518674] |
| Dif, Div, Central | [35,602112642270264] | [12,73239544730131; 7,999999999341865] |
| **Punto: [-1,0; 3,0]** | | |
| Analítica | [37,473372534686355] | [-9,386046437711835; -9,441467880125868] |
| ADouble | [37,473372534686355] | [-9,386046437711835; -9,441467880125868] |
| Dif, Div, Adelante | [37,473372534686355] | [-9,386025959656763; -9,441457880399184] |
| Dif, Div, Atrás | [37,473372534686355] | [-9,386066916050595; -9,44147788075611] |
| Dif, Div, Central | [37,473372534686355] | [-9,386046437853675; -9,441467880577645] |
| **Punto: [7,0; 12,0]** | | |
| Analítica | [134,1125605939252] | [-11,001077310524934; 21,621610136755244] |
| ADouble | [134,1125605939252] | [-11,001077310524938; 21,62161013675525] |
| Dif, Div, Adelante | [134,1125605939252] | [-11,001140973121435; 21,621620135192643] |
| Dif, Div, Atrás | [134,1125605939252] | [-11,001013649547529; 21,621600137677888] |
| Dif, Div, Central | [134,1125605939252] | [-11,00107731133448; 21,62161013643527] |
| **Punto: [6,0; 10,0]** | | |
| Analítica | [98,40571082188518] | [3,418636551411606; 17,797308512660834] |
| ADouble | [98,40571082188518] | [3,418636551411606; 17,797308512660834] |
| Dif, Div, Adelante | [98,40571082188518] | [3,418567476387579; 17,797318513146365] |
| Dif, Div, Atrás | [98,40571082188518] | [3,4187056229484365; 17,797298511368354] |
| Dif, Div, Central | [98,40571082188518] | [3,418636549668008; 17,79730851225736] |

**Tabla 3. Error absoluto del gradiente de F(x) obtenido durante la aplicación de cada uno de los métodos**

| Método | X | Y | Error Absoluto |
|---|---|---|---|
| **[9,42477796076938; 2,475]** | | | |
| Analitica | **-3,53E-15** | **0** | |
| Automatica | -3,53E-15 | 0 | [0;0] |
| Dif. Div. Adelante | 5,51E-05 | 1,00E-05 | [-5,51257872718231E-05;-1,00000008274037E-05] |
| Dif. Div. Atrás | -5,51E-05 | -1,00E-05 | [5,51257428558465E-05;1,00000008274037E-05] |
| Dif. Div. Central | 4,44E-11 | 0 | [-4,44124487439599E-11;0] |
| **[3,141592653589793; 2,25]** | | | |
| Analitica | **-0,038992961** | **-0,05** | |
| Automatica | -0,038992961 | -0,05 | [1,29757316003065E-15;1,79717352111197E-15] |
| Dif. Div. Adelante | -0,038938804 | -0,04999 | [-0,000054156859817904;-9,9999993885963E-06] |
| Dif. Div. Atrás | -0,039047118 | -0,05001 | [5,41569036959025E-05;1,00000022661972E-05] |
| Dif. Div. Central | -0,077985922 | -0,1 | [0,0389929611013927;0,0500000000028761] |
| **[-3,141592653589793; 12,275]** | | | |
| Analitica | **1,18E-15** | **0** | |
| Automatica | 1,18E-15 | 0 | [0;0] |
| Dif. Div. Adelante | 1,06E-04 | 1,00E-05 | [-0,000105766029266863;-1,00000008274037E-05] |
| Dif. Div. Atrás | -1,06E-04 | -1,00E-05 | [0,000105766156944863;1,00000008274037E-05] |
| Dif. Div. Central | -1,28E-10 | 0 | [1,27676823751544E-10;0] |
| **[0,0; 10,0]** | | | |
| Analitica | **1,27E+01** | **8,00E+00** | |
| Automatica | 1,27E+01 | 8,00E+00 | [0;0] |
| Dif. Div. Adelante | 1,27E+01 | 8,00E+00 | [3,30147080003229E-05;-9,99916504973442E-06] |
| Dif. Div. Atrás | 1,27E+01 | 8,00E+00 | [-0,00003301460730043;1,00004813301524E-05] |
| Dif. Div. Central | 2,55E+01 | 1,60E+01 | [-12,732395447251;-7,9999999986837] |
| **[-1,0; 3,0]** | | | |
| Analitica | **-9,39E+00** | **-9,44E+00** | |
| Automatica | -9,39E+00 | -9,44E+00 | [0;0] |
| Dif. Div. Adelante | -9,39E+00 | -9,44E+00 | [-2,04780550703276E-05;-9,99972667869997E-06] |
| Dif. Div. Atrás | -9,39E+00 | -9,44E+00 | [0,000020478338759844;0,000010000630251028] |
| Dif. Div. Central | -1,88E+01 | -1,89E+01 | [9,38604643799547;9,44146788102934] |
| **[7,0; 12,0]** | | | |
| Analitica | **-1,10E+01** | **2,16E+01** | |
| Automatica | -1,10E+01 | 2,16E+01 | [0;0] |
| Dif. Div. Adelante | -1,10E+01 | 2,16E+01 | [6,36625964993698E-05;-9,99843739890593E-06] |
| Dif. Div. Atrás | -1,10E+01 | 2,16E+01 | [-6,36609773998487E-05;9,9990774025116E-06] |
| Dif. Div. Central | -22,00215462 | 43,24322027 | [11,001077312144;-21,6216101361153] |
| **[6,0; 10,0]** | | | |
| Analitica | **3,418636551** | **17,79730851** | |
| Automatica | 3,418636551 | 17,79730851 | [0;0] |
| Dif. Div. Adelante | 3,418567476 | 17,79731851 | [6,90750240299742E-05;-1,00004855028146E-05] |
| Dif. Div. Atrás | 3,418705623 | 17,79729851 | [-6,90715368296679E-05;1,00012924981741E-05] |
| Dif. Div. Central | 6,837273099 | 35,59461702 | [-3,41863654792441;-17,7973085118539] |

## Diferenciación automática en optimización

Una de las principales aplicaciones de la diferenciación automática se encuentra en el área de la optimización, en la cual la exactitud de la derivada es crucial para garantizar la convergencia de los métodos empleados en el hallazgo de los mínimos de cualquier función.

Para demostrar la eficiencia en el cálculo de derivadas exactas de la diferenciación automática, se ha seleccionado el algoritmo de optimización conocido como *Levenberg-Marquardt* [8,9]. Este algoritmo proporciona una solución numérica al problema de minimizar una función generalmente no lineal.

El algoritmo de *Levenberg-Marquardt* interpola entre el algoritmo de Newton y el método de descenso más rápido, característica esta que lo hace más robusto puesto que, aunque comience a iterar a partir de puntos muy lejanos al mínimo final, generalmente consigue una solución.

La biblioteca *Commons Math*, distribuida bajo licencia Apache 2.0, proporciona funcionalidades matemáticas relacionadas con estadística, algebra lineal, interpolación, matrices, optimización, entre otros. Entre las clases para optimización que provee se encuentra *LevenbergMarquardtOptimizer*, que utiliza el algoritmo antes descrito para resolver problemas de optimización.

La clase *LevenbergMarquardtOptimizer* tiene una función *optimize* que ejecuta el algoritmo de optimización; esta requiere de objetos diferenciables para ejecutarse, es decir, es preciso definir clases que implementen la interfaz *DifferentiableMultivariateVectorialFunction* (incluida en la misma biblioteca) cuya implementación exige la definición de dos métodos: *value* y *jacobiano*, el primero retorna el valor y el segundo el jacobiano de la función a diferenciar.

## Comparación de técnicas de diferenciación con una función bivariada

En la Figura 6 se muestra la definición de una clase para la evaluación de la función de Branin-Hoo (descrita en secciones previas).

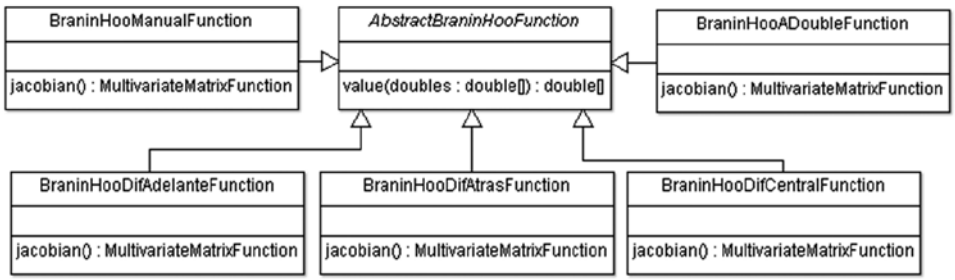
En la línea 1 puede observarse que la clase implementa la interfaz *DifferentiableMultivariateVectorialFunction*, lo cual permite que cualquier objeto de esta clase pueda ser pasado al método *optimize* de la clase *LevenbergMarquardtOptimizer*. Además, la clase es abstracta lo que significa que las clases derivadas deberán implementar sólo el método faltante correspondiente al cálculo del jacobiano.

**Figura 6. Clase AbstractBraninHooFunction**

```
1.  public abstract class AbstractBraninHooFunction
                          implements DifferentiableMultivariateVectorialFunction{
2.      public final double[] value(double[ ] doubles) throws
                                                    FunctionEvaluationException,
3.                                                  IllegalArgumentException {
4.          double x = doubles[0];
5.          double y = doubles[1];
6.          double term1 = y - (5.1*x*x) / (4*PI*PI) + (5*x)/PI - 6;
7.          term1 = term1 * term1;
8.          double term2 = 10 * (1 - 1/(8*PI)) * cos(x) + 10;
9.          final double res = term1 + term2;
10.           return new double [] {res}; 10.
            } }
```

A partir de la clase *AbstractBraninHooFunction*, fue posible definir 5 clases para implementar los diferentes métodos de diferenciación a fin de ejecutar las pruebas, el diagrama de clases correspondiente puede observarse en la Figura 7.

**Figura 7. Diagrama de clases definidas para la ejecución de la Función BraninHoo**




En la Figura 8 se muestra la clase *BraninHooManualFunction*, en la cual el jacobiano es calculado a partir de la codificación de la derivada analítica de la función. Obsérvese que no se utiliza la clase *ADouble*. Al ser una clase derivada de la clase *AbstractBraninHooFunction*, implementa el método *ja-*

*cobian,* este último utiliza la clase interna *Jacobian* (véase líneas 5-18), que es quien tiene definido el método para calcular la derivada.

**Figura 8. Clase BraninHooManualFunction**

```
1.  public class BraninHooManualFunction extends AbstractBraninHooFunction{
2.      public MultivariateMatrixFunction jacobian() {
3.         return new Jacobian();
4.      }
5.      private static class Jacobian implements MultivariateMatrixFunction
6.      {
7.         public double[][] value(double[] doubles) throws FunctionEvaluationException,
                                                            IllegalArgumentException {
8.           double x1 = doubles [0];
9.           double x2 = doubles [1];
10.          double term2 = ((-10.2*x1)/(4*PI*PI)) +5/PI;
11.          double term3 = (10* (1-(1/ (8*PI))) * Math.sin (x1) )*(-1) ;
12.          double dfx = (term1 * term2) + term3;
13.          double dfy = term1;
14.          double[] dfxy = {dfx,dfy};
15.          return new double[][]{dfxy};
16.       }
17.     }
18. }
```

Una vez definidas las clases necesarias para pasar la función de Branin-Hoo al optimizador de LevenbergMarquardt, se ejecutaron las pruebas con 4 puntos (los últimos puntos de las pruebas efectuadas en la sección anterior), obteniendo los siguientes resultados.

**Tabla 4. Puntos mínimos para la función de Branin-Hoo implementando el algoritmo de optimización de LevenbergMarquardt**

| Método | Mínimo | F(x) |
|---|---|---|
| **Punto: (0.0, 10.0)** | | |
| Analítica | [-3.141589028919808, 12.274984047878512] | [0.39788735784524937] |
| ADouble | [-3.141589028918824, 12.274984047875765] | [0.3978873578452494] |
| Dif. Div. Adelante | [-3.1415636725149994, 12.274872641901215] | [0.39788736509261796] |
| Dif. Div. Atrás | [-3.141573576279439, 12.274917431899981] | [0.3978873608262052] |
| Dif. Div. Central | [-3.1415892588546503, 12.274985022849362] | [0.39788735783156226] |
| **Punto: (-1.0, 3.0)** | | |
| Analítica | [-3.1415895074733693, 12.274987138363889] | [0.3978873578053574] |
| ADouble | [-3.141589507473393, 12.274987138364008] | [0.3978873578053574] |
| Dif. Div. Adelante | [-3.141563406219107, 12.274871470257809] | [0.39788736522870877] |
| Dif. Div. Atrás | [-3.141575073147592, 12.274924366647962] | [0.39788736032990857] |
| Dif. Div. Central | [-3.1415887556707682, 12.274982907228726] | [0.397887357862362] |
| **Punto: (7.0, 12.0)** | | |
| Analítica | [9.42477785250463, 2.47499742989686] | [0.39788735773593936] |
| ADouble | [9.424777852504628, 2.4749974298968533] | [0.39788735773593936] |
| Dif. Div. Adelante | [9.424778812836497, 2.475013675483021] | [0.39788735790457985] |
| Dif. Div. Atrás | [9.424784188376814, 2.4750156408579103] | [0.39788735803651964] |
| Dif. Div. Central | [9.424778905178037, 2.475009374574176] | [0.3978873578076021] |
| **Punto: (6.0, 10.0)** | | |
| Analítica | [3.1415928484833477, 2.2749986481798175] | [0.3978873577313607] |
| ADouble | [3.1415928484833473, 2.27499864817982] | [0.3978873577313607] |
| Dif. Div. Adelante | [3.1415939087661284, 2.2749895101946405] | [0.39788735782949397] |
| Dif. Div. Atrás | [3.141598236408053, 2.274996311572142] | [0.3978873578834072] |
| Dif. Div. Central | [3.14159312524573, 2.2749954927839005] | [0.3978873577479403] |

Como puede observarse, el punto mínimo de la función obtenido utilizando la diferenciación automática en Java es casi exacto con respecto al punto mínimo obtenido utilizando diferenciación analítica. Ello demuestra la exactitud del resultado, no sólo en la obtención del punto mínimo, sino también en la evaluación de la función (véase columna F(x) en la Tabla 4). El resto de los métodos empleados (dife-

rencias divididas hacia atrás, hacia adelante y centrales) obtienen valores muy cercanos, sin embargo no logran igualar, y mucho menos superar, los arrojados con la diferenciación automática en Java.

## Comparación de técnicas de diferenciación de funciones multivariadas

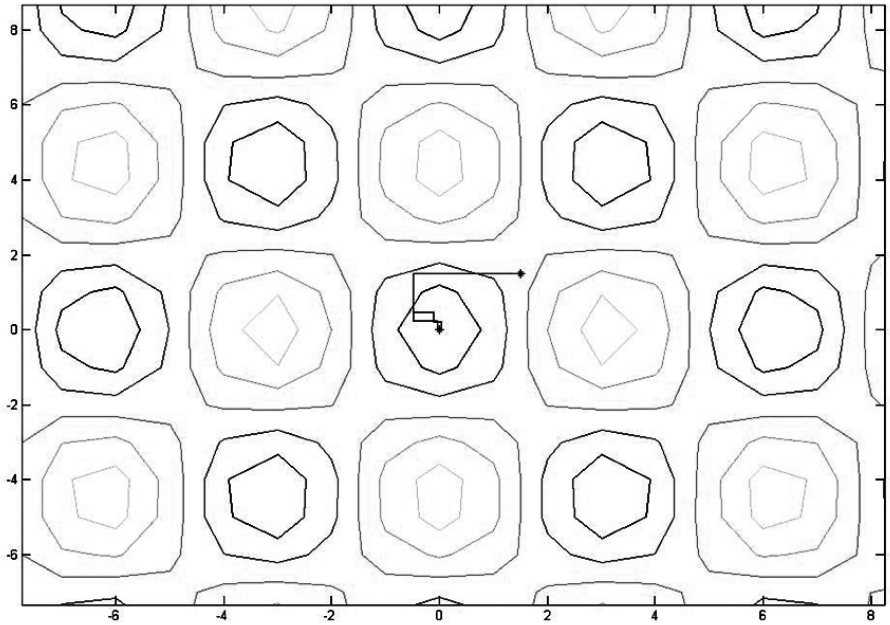
En esta sección se detallan las pruebas realizadas a la biblioteca, también en el área de optimización, esta vez aplicada a tres funciones previamente seleccionadas por su relevancia y frecuente uso en aplicaciones de ingeniería: Griewank, Shubert y Zakharov [7].

En la tabla 5 se listan los resultados obtenidos para la función Griewank, cuyo punto mínimo se ubica en (0,0). Se han tomado como puntos iniciales: (-1.0,-1.0), (1.0,1.0) y (1.5, 1.5); obsérvese que, aun cuando el mínimo obtenido no es exacto, el algoritmo de Levenberg Maquartd se detiene puesto tiene una tolerancia de 1.0e-10. Para los dos primeros puntos el valor obtenido con la Librería de Diferenciación Automática es más exacto que el obtenido con la derivación numérica. No ocurre lo mismo, para el tercero, para el cual el mejor valor es el obtenido en forma numérica. La gráfica 1 muestra a través de curvas de nivel, la trayectoria seguida por el algoritmo de optimización en la consecución del óptimo local resultante para la función Griewank.

**Tabla 5. Error absoluto para puntos mínimos hallados de la función Griewank implementando el algoritmo de optimización de LevenbergMarquardt**

| Método | Mínimo | Error Absoluto |
|---|---|---|
| **Punto inicial: (-1.0, -1.0)** | | |
| Analítica | [-7,33E-06; -1,36E-08] | [ 7,33E-06;1,36E-08] |
| ADouble | [ 2,69E-07; -1,25E-08] | [2,69E-07;1,25E-08] |
| **Punto inicial: (1.0, 1.0)** | | |
| Analítica | [7,33E-06; 1,36E-08] | [7,33E-06;1,36E-08] |
| ADouble | [-2,69E-07;1,25E-08] | [ 2,69E-07;1,25E-08] |
| **Punto inicial: (1.5, 1.5)** | | |
| Analítica | [1,93E-07; 6,46E-07] | [1,93E-07;6,46E-07] |
| ADouble | [4,55E-06;1,37E-08] | [4,55E-06;1,37E-08] |

**Gráfica 1. Trayectoria descrita por el algoritmo de optimización en la consecución del óptimo local resultante para la función Griewank**

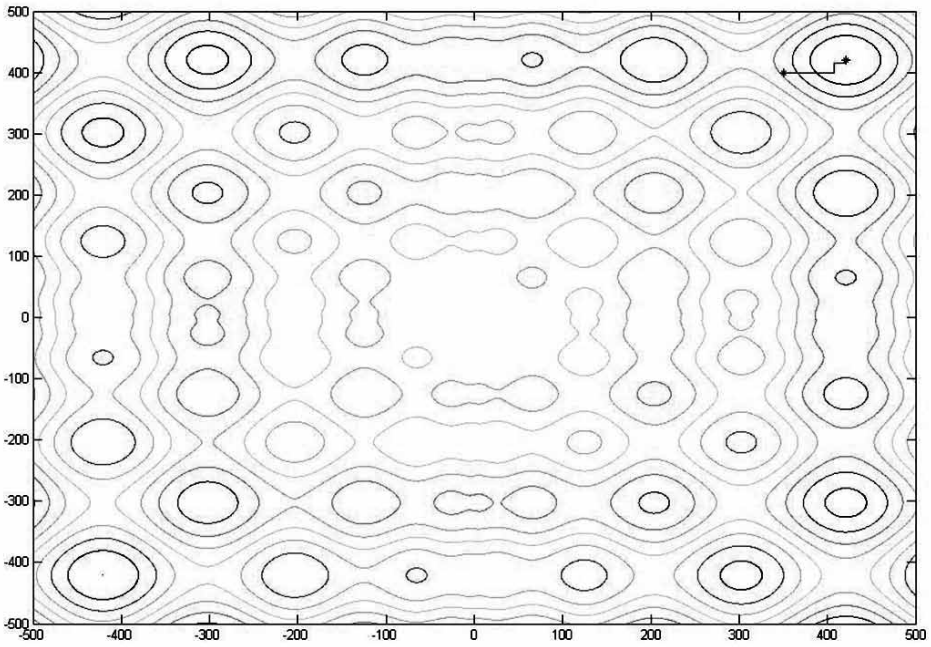


La tabla 6 lista los resultados obtenidos para la función Scwefel cuyo punto mínimo se encuentra en (420.9687, 420.9687). Se tomaron como puntos iniciales: (400, 400), (350, 350) y (350, 400). Para los tres puntos con ambos métodos se obtienen los mismos resultados, con lo que se valida la exactitud de los resultados arrojados por la biblioteca diseñada. La gráfica 2 muestra a través de curvas de nivel, la trayectoria seguida por el algoritmo de optimización en la consecución del óptimo local resultante para la función Scwefel.

**Tabla 6. Error absoluto para puntos mínimos hallados de la función Schwefel implementando el algoritmo de optimización de LevenbergMarquardt**

| Método | Mínimo | Error Absoluto |
|---|---|---|
| **Punto inicial: (400, 400)** | | |
| Analítica | [4,21E+02;4,21E+02] | [4,66E-05;4,66E-05] |
| ADouble | [4,21E+02;4,21E+02] | [4,66E-05;4,64E-05] |
| **Punto inicial: (350, 350)** | | |
| Analítica | [4,21E+02;4,21E+02] | [4,64E-05;4,64E-05] |
| ADouble | [4,21E+02;4,21E+02] | [4,64E-05;4,64E-05] |
| **Punto inicial: (350, 400)** | | |
| Analítica | [4,21E+02;4,21E+02] | [4,62E-05;4,64E-05] |
| ADouble | [4,21E+02;4,21E+02] | [4,62E-05;4,64E-05] |

**Gráfica 2. Trayectoria descrita por el algoritmo de optimización en la consecución del óptimo local resultante para la función Schwefel**




Finalmente, las tablas 7 y 8 listan los resultados de las pruebas para las funciones Zakharov y Power Sum, respectivamente. En estas se repite el patrón observado para la función Scwefel: se obtuvieron los mismos resultados con ambos métodos. Vale la pena acotar que para la función PowerSum se ha trabajado con 4 variables, lo que indica la funcionalidad de la biblioteca diseñada para más de dos dimensiones. La gráfica 3 muestra a través de curvas de nivel, la trayectoria seguida por el algoritmo de optimización en la consecución del óptimo local resultante para la función Zakharov. No se generaron curvas de nivel para PowerSum por la alta dimensionalidad de la misma, lo cual dificulta su visualización.

**Tabla 7. Error absoluto para puntos mínimos hallados de la función Zakharov implementando el algoritmo de optimización de LevenbergMarquardt**

| Método | Mínimo | Error Absoluto |
|---|---|---|
| **Punto inicial: (0, 10)** | | |
| Analítica | [0,00E+00;4,46E-67] | [0,00E+00;4,46E-67] |
| ADouble | [0,00E+00;4,46E-67] | [0,00E+00;4,46E-67] |
| **Punto inicial: (50, 50)** | | |
| Analítica | [6,12E-67; -2,81E-66] | [6,12E-67;2,81E-66] |
| ADouble | [6,12E-67;-2,81E-66] | [6,12E-67;2,81E-66] |
| **Punto inicial: (-5, 6)** | | |
| Analítica | [-6,15E-67;1,23E-67] | [6,15E-67;1,23E-67] |
| ADouble | [-6,15E-67;1,23E-67] | [6,15E-67;1,23E-67] |

**Gráfica 3. Trayectoria descrita por el algoritmo de optimización en la consecución del óptimo local resultante para la función Zakharov**

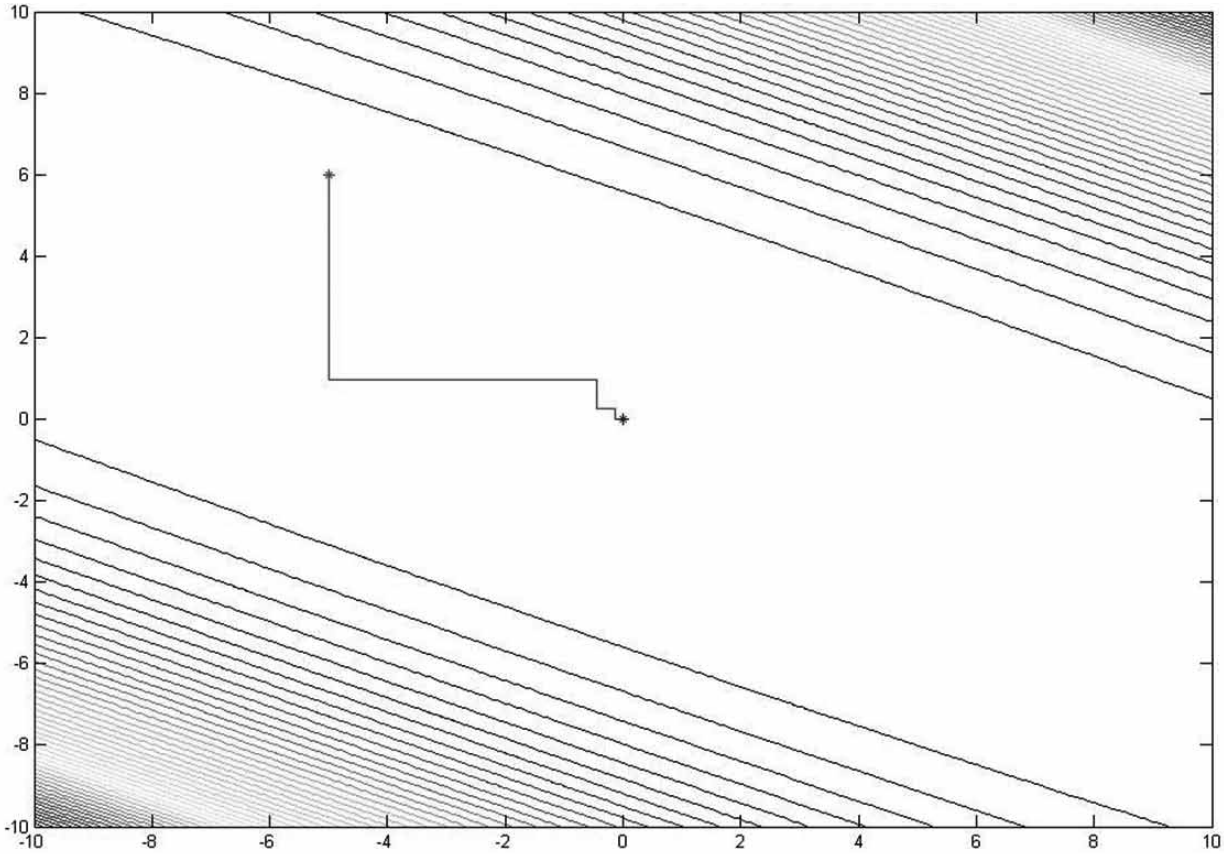


**Tabla 8. Error absoluto para puntos mínimos hallados de la función Power implementando el algoritmo de optimización de LevenbergMarquardt**

| Método | Mínimo | Error Absoluto |
|---|---|---|
| **Punto inicial: (1, 2, 2, 2.9)** | | |
| Analítica | [1,00E+00;2,00E+00; 2,00E+00; 3,00E+00] | [0,00E+00; 0,00E+00; 0,00E+00; 0,00E+00] |
| ADouble | [1,00E+00;2,00E+00; 2,00E+00; 3,00E+00] | [0,00E+00; 0,00E+00, 0,00E+00; 0,00E+00] |
| **Punto inicial: (1, 2, 2, 2)** | | |
| Analítica | [1,00E+00; 2,00E+00; 2,00E+00; 3,00E+00] | [0,00E+00; 0,00E+00; 0,00E+00; 8,43E-11] |
| ADouble | [1,00E+00; 2,00E+00; 2,00E+00; 3,00E+00] | [0,00E+00; 0,00E+00; 0,00E+00; 8,43E-11] |
| **Punto inicial: (1, 2, 2, 3)** | | |
| Analítica | [1,00E+00; 3,00E+16; 2,00E+00; 2,00E+00] | [0,00E+00; 3,00E+16; 0,00E+00; 0,00E+00] |
| ADouble | [1,00E+00; 3,00E+16; 2,00E+00; 2,00E+00] | [0,00E+00; 3,00E+16; 0,00E+00; 0,00E+00] |

## Conclusiones

Mediante la diferenciación automática es posible aplicar la regla de la cadena a secuencias de operaciones elementales, tales como sumas, multiplicaciones, funciones trigonométricas, entre otras, representadas en un programa, para de esta manera obtener valores precisos para las derivadas parciales.

Al implementar la diferenciación automática en la presente investigación se ha logrado desarrollar un mecanismo con el cual es posible el cálculo exacto de las derivadas parciales de una función definida en lenguaje de programación Java, de una manera sencilla, sin necesidad de intervención del usuario en el cálculo de la misma. Ello facilita considerablemente el cálculo de derivadas parciales, además de garantizar la minimización del error a lo largo de todo el proceso y en los resultados finales.

Además, se ha demostrado, a través de varios ejemplos, la simplicidad de adaptación de la clase ADouble para el cálculo del Jacobiano; normalmente, este proceso resulta costo computacionalmente ($\Theta(n^2C)$, n=#variables, C=costo promedio de obtener la derivada parcial analítica correspondiente), además de ser susceptible a errores de aproximación y redondeo, en el caso de diferenciación numérica. Con la solución presentada, basta incorporar algunas líneas de código y se tendrá una nueva clase a partir de la cual con la simple invocación de un método se tendrá el Jacobiano de la función dada, dato indispensable para la ejecución de la mayoría de los métodos de optimización.

Posteriores pruebas de la librería ADouble demostraron la efectividad y eficiencia de la misma como herramienta fundamental en la implementación de técnicas de optimización. Al usar ADouble como tipo de dato central en la implementación del bien conocido algoritmo de Levenberg-Marquardt para hallar mínimos a las funciones Griewank, Shubert y Zakharov se obtuvo resultados iguales o mejores en cuanto al error absoluto respecto a técnicas analíticas, con excepción de un caso, Griewank para punto inicial (1,5;1,5), en el que la susceptibilidad a la escogencia del punto inicial y los errores de redondeo produjeron resultados marginalmente mejores para el caso de diferenciación analítica (< 1E-07).

## Referencias bibliográficas

1. Chapra S. y y Canale R. (2008). “Métodos Numéricos para ingenieros”. Tercera edición. Mc Graw Hill.
2. Bischof C., Carle A., Khademi P. y Mauer A. (1996). “ADIFOR 2.0: Automatic differentiation of Fortran 77 programs”. IEEE Computational Science & Engineering.
3. Bischof C., Roh L., Mauer-Oats A. (1997). “ADIC: An extensible automatic differentiation tool for ANSI-C”. IEEE Computational Science & Engineering.

4. Kowarz A. (2008). “Advanced Concepts for Automatic Differentiation based on Operator Overloading”. Trabajo presentado para obtener el grado académico de Doctor. Universidad Técnica de Dresde - Alemania.
5. Slusanschi E. (2008). “Algorithmic Differentiation of Java Programs”. Trabajo presentado para obtener el grado académico de Doctor en Ciencias. Universidad Técnica de Aquisgrán – Alemania.
6. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., and Wunsch, C. (2008). “OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes”, ACM Trans. Math. Softw. 34, 4, Article 18 (July 2008), 36 pages. DOI = 10.1145/1377596.1377598. ttp://doi.acm.org/10.1145/1377596.1377598.
7. Cagnina, Leticia (2010). “Optimización Mono y Multiobjetivo a través de una Heurística de Inteligencia Colectiva”. Tesis doctoral. Doctorado en Ciencias de la Computación. Universidad Nacional de San Luis. Argentina.
8. Levenberg, K. "A Method for the Solution of Certain Problems in Least Squares". Quart. Appl. Math. 2, 164-168, 1944.
9. Marquardt, D. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." SIAM J. Appl. Math. 11, 431-441, 1963.