SOBRE LA NECESIDAD DEL DESARROLLO DE ALGORITMOS INFORMÁTICOS

Francisco Salamanca González[1]

Universidad Nacional de Educación a Distancia,UNED, España

RESUMEN. Este artículo surge como el resumen del trabajo fin de grado de Matemáticas del mismo autor bajo la tutela del profesor Miguel Delgado Pineda[2] en el curso 2022-2023. El objetivo de este documento es reflejar un algoritmo desarrollado en Python para facilitar la búsqueda del valor mínimo de una función cualquiera continua de n variables definida en una caja. La optimización de funciones convexas se puede apoyar en varias propiedades que facilitan el cálculo y el diseño de los algoritmos, situación de la que las funciones no convexas no se ven beneficiadas. Esto dificulta el cálculo de la búsqueda de los minimizadores ya que la obtención de mínimo local no garantiza en absoluto que se corresponda con el mínimo global, como sí sucede en la optimización de funciones convexas [11]. En estas páginas se plantearán diferentes casos con complejidad creciente para visibilizar la necesidad de programas informáticos en su resolución. Después se planteará, junto con su respectivo código, un algoritmo con este objetivo: e1 algoritmo cúbico [3]. Se ha planteado el desarrollo en el lenguaje de programación Python por la facilidad que supone tanto en la lectura de su código, como en las librerías que implementa con diversas herramientas matemáticas y el alcance que tiene por ser un lenguaje libre.

PALABRAS CLAVE: Optimización global, optimización con restricciones, optimización no convexa, optimización en Python, algoritmo cúbico

---

ABSTRACT. This article emerges as the summary of the end-of-degree project in the Mathematics degree by the same author under the tutelage of Professor Miguel Delgado[2] Pineda in the 2022-2023 academic year. The aim of this document is to reflect an algorithm developed in Python to enable the search for the minimum value of any continuous function in R^n with box constraints. Convex functions optimization can be supported by several properties that make the calculation and design of algorithms easier, which don't apply for non-convex functions. This makes the search for the minimizers more difficult since obtaining a local minimum does not guarantee at all that it corresponds to the global minimum, as it does in the optimization of convex functions [11]. In this document, different increasing-complexity examples will be considered to show the need for computer programs in their resolution. Later, together with its code, there will be considered an algorithm with that goal: the cubic algorithm [3]. Development in Python programming language has been proposed due to the ease it entails, both in reading its code as well as in the libraries that it implements with several mathematical tools.

KEY WORDS: Global optimization, constrained optimization, non-convex optimization, Python optimization, cubic algorithm

[1] fsalamanc8@alumno.uned.es
[2] miguel@mat.uned.es

1. INTRODUCCIÓN Y OBJETIVO

La optimización de procesos y recursos es un problema recurrente en casi todos los ámbitos. Además, en aquellos casos en los que podamos generar un modelo matemático para estudiar dichos comportamientos mediante una función, surge la necesidad de disponer de herramientas que permitan trabajar con ellas.

Cuando las funciones a las que nos enfrentamos son suficientemente “asequibles” podemos trabajar con ellas analíticamente sin necesitar el apoyo de herramientas informáticas (siendo incluso más lento el proceso de diseñar un código informático que el cálculo manual). Sin embargo, al ir incrementando la dificultad de estas funciones y operaciones, y aunque el concepto de “asequibilidad” o “dificultad” sean un tanto subjetivos, es fácil entender cómo llega un momento en el que el cálculo manual o analítico resulta imposible siendo las herramientas de programación no sólo recomendables sino imprescindibles.

En los siguientes apartados se plantearán varios ejemplos de optimización de funciones no convexas y no diferenciables en los que se pretende obtener el valor mínimo global que alcanza cada una de esas funciones, así como el conjunto de minimizadores, esto es, los puntos del dominio en la que la función alcanza ese mínimo. Partiremos de un ejemplo sencillo e incluso trivial para ir incrementando la complejidad hasta poner de manifiesto la necesidad de apoyo informático.

Para ello se ha decidido desarrollar un código que permita dicho cálculo en Python, lenguaje libre con multitud de librerías que facilitan su programación y simplifican las rutinas. Se ha trabajado con la versión 3.11.2 [8], la más actualizada en la fecha de la realización de este documento. Para la ejecución de los códigos que aquí se muestran es además necesaria la instalación de las librerías *numpy* [6], *matplotlib* [4], *scipy* [10] e importar las librerías *functools* e *itertools*. El código mendionado se muestra al completo en el apéndice A-1.

2. OPTIMIZACIÓN GLOBAL SIN RESTRICCIONES.

Recordemos antes de empezar [1] que $f(x)$ alcanza el mínimo absoluto o mínimo global en $x = a$ si y solo si $f(a) \leq f(x)$ para cualquier valor x en el que esté definida la función f , siendo:

$$a, x \in R^n$$

$$f(x): R^n \to R$$

2.1. Ejemplo 1

Comenzaremos determinando el valor mínimo absoluto (o mínimo global) de la siguiente función y los puntos en los cuales se alcanza ese mínimo:

$$f(x) = 1 + |x^3 - 3x| \qquad \text{[Ec.1]}$$

En caso de trabajar con funciones que hacen uso de la norma (comúnmente conocida como valor absoluto cuando es aplicada a una variable real) podemos ayudarnos en su propiedad básica de no negatividad, es decir:

$$|g(x)| \geq 0 \qquad \forall x \in \mathrm{R} \qquad \text{[Ec.2]}$$

por lo que siendo k una constante real, tendríamos:

$$f(x) = \mathrm{k} + |g(x)| \geq \mathrm{k} \qquad \text{[Ec.3]}$$

Como ambos términos son positivos e independientes, encontrar el mínimo de $f(x)$ pasa por encontrar el mínimo de $|g(x)|$:

$$Min\{f(x)\} = k + Min\{|g(x)|\} \qquad \text{[Ec.4]}$$

Es intuitivo[3] suponer que en caso de que exista algún valor x_0 que cumpla:

$$g(x_0) = 0 \qquad \text{[Ec.5]}$$

Se tendrá que el mínimo de $|g(x)|$ será 0 por lo expresado en [Ec.2], perteneciendo x_0 al conjunto de minimizadores buscados. Retomando el ejemplo de la expresión [Ec.1], deducimos que el mínimo de $f(x)$ tomará el valor 1 en caso de que existan valores en los que se cumpla $x^3 - 3x = 0$. En caso afirmativo, los minimizadores estarán formados por dichos puntos.

Evidentemente $x^3 - 3x = 0$ tiene las soluciones $x = \{-\sqrt{3}, 0, \sqrt{3}\}$ Por lo que $f(x)$ alcanzará el valor mínimo 1 en dichos puntos.

Podríamos haber llegado a la misma solución definiendo $f(x)$ por intervalos y estudiando su monotonía:

$$f(x) = \begin{cases} 1 - (x^3 - 3x) = -x^3 + 3x + 1 & si\; x^3 - 3x < 0 \\ 1 + (x^3 - 3x) = x^3 - 3x + 1 & si\; x^3 - 3x \geq 0 \end{cases} \qquad \text{[Ec.6]}$$

Que puede escribirse como sigue:

$$f(x) = \begin{cases} -x^3 + 3x + 1 & si\; x \in (-\infty, -\sqrt{3}) \cup (0, \sqrt{3}) \\ x^3 - 3x + 1 & si\; x \in [-\sqrt{3}, 0) \cup [\sqrt{3}, \infty) \end{cases} \qquad \text{[Ec.7]}$$

Con más o menos soltura, podemos ir estudiando cada uno de los intervalos anteriores, así como los puntos extremos de cada uno, llegando a que la función está acotada inferiormente por 1 (valor mínimo absoluto) y que el conjunto de minimizadores está formado por los puntos $x = -\sqrt{3}, x = 0$ y $x = \sqrt{3}$. Además, sabríamos que $f(x)$ no está acotada superiormente por lo que no tiene máximo absoluto, pero presenta dos máximos locales fuertes en $x = \pm 1$ de valor 3.

---

[3] Podríamos desarrollar con más precisión estas demostraciones, pero se desviaría del objetivo de este artículo.

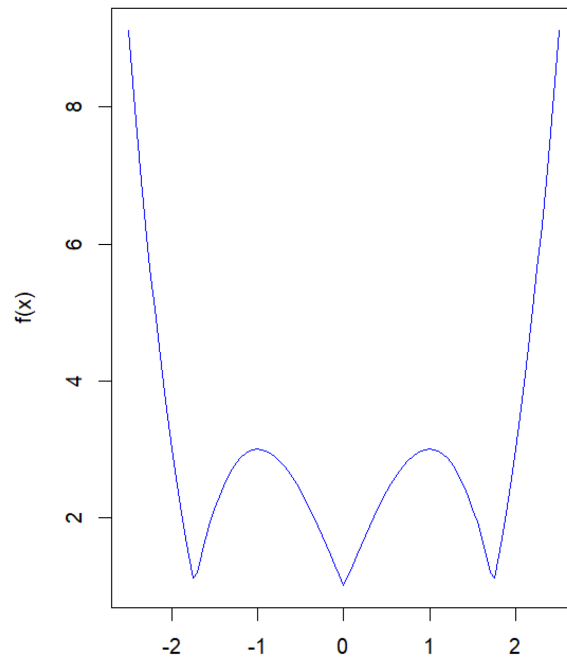

Imagen 1: Representación gráfica de la función f(x) en el intervalo [-2.5, 2.5]. Fuera de ese intervalo la función continúa creciendo indefinidamente.

2.2. Ejemplo 2

Juguemos ahora añadiendo más variables independientes para determinar el mínimo global y conjunto de minimizadores de la siguiente función:

$$f(x,y,z) = 1 + |(x^2 + y^2 + z^2)^3 - 3x^2 - 3y^2 - 3z^2| \qquad \text{[Ec.8]}$$

En primer lugar, realizamos el cambio de variables a coordenadas esféricas para facilitar su trabajo:

$$\rho^2 = x^2 + y^2 + z^2,\ \rho \in [0, +\infty) \qquad \text{[Ec.9]}$$

Y definimos la función $g(x,y)$ como sigue:

$$g(x,y,z) = (x^2 + y^2 + z^2)^3 - 3x^2 - 3y^2 - 3z^2 \qquad \text{[Ec.10]}$$

Por lo que podemos definir entonces $f(x,y)$ de la siguiente forma:

$$f(\rho) = 1 + |g(\rho)| \qquad \text{[Ec.11]}$$

Siendo

$$g(\rho) = \rho^6 - 3\rho^2 = \rho^2(\rho^4 - 3) \qquad \text{[Ec.12]}$$

Por lo expuesto previamente, el mínimo absoluto de $f(\rho)$ se alcanzará en los puntos que minimicen $|g(\rho)|$, así que comprobaremos si existe algún valor que anule $g(\rho)$.

$$g(\rho) = 0 \ \rightarrow \ \rho^2(\rho^4 - 3) = 0 \ \begin{cases} \rho = 0 \\ \rho^4 = 3 \end{cases} \qquad \text{[Ec.13]}$$

Por tanto, el mínimo de $|g(\rho)|$ es 0 y el de $f(\rho)$ es 1 y se alcanzan en los puntos:

$$\rho = \{0, \sqrt[4]{3}\}.$$

La interpretación de dicha solución parte por deshacer el cambio de variables inicial de la expresión [Ec.9]:

$\rho = 0$ se corresponde con el punto $(x, y, z) = (0,0,0)$

$\rho = \sqrt[4]{3}$ equivale a los puntos del espacio correspondientes a $x^2 + y^2 + z^2 = \sqrt{3}$, es decir aquellos puntos situados en R^3 en la superficie de la esfera de centro 0 y radio $\sqrt[4]{3}$.

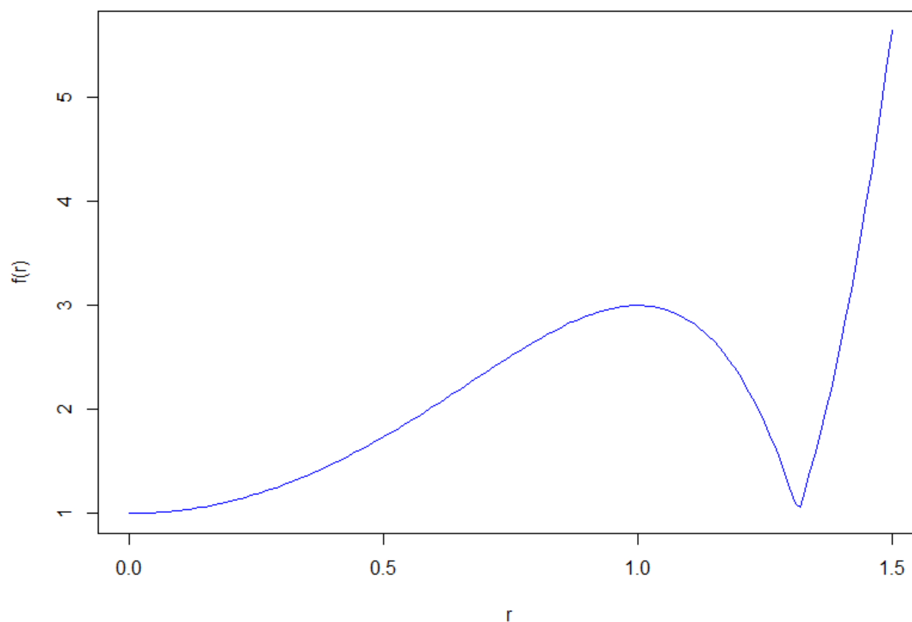

Imagen 2: Representación gráfica de la función f(   ) en el intervalo [0, 1.5].

3. OPTIMIZACIÓN CON RESTRICCIONES.

Hasta ahora hemos obtenido los mínimos absolutos de las diferentes funciones en todo su dominio de definición. Veamos ahora qué pasaría si las restringimos a un intervalo cerrado [a,b][4]. Como norma general, se puede tomar el siguiente procedimiento para obtener el valor mínimo de una función en una región concreta:

1. Obtener mínimos globales y locales de la función (sin limitar el dominio).
2. Eliminar los puntos anteriores que no pertenezcan a la región [a,b].
3. Evaluar la función en la frontera.
4. De entre los valores resultantes de los puntos anteriores, determinar el menor.

[4] Siendo $f: R^n \rightarrow R$ y los puntos a y b pertenecientes a R^n.

3.1. Ejemplo 3

Partamos de la función del ejemplo anterior según la expresión [Ec.8]. Tomemos el dominio de definición

$$\bar{X}=[1,5]x[0,3]x[1,3] \quad \text{[Ec.14]}$$

De acuerdo con el listado anterior, tendríamos que obtener la intersección entre la esfera de centro 0 y radio $\sqrt[4]{3}$ con el cubo definido por las coordenadas de [Ec.14] para comprobar si algunos de los puntos del conjunto de minimizadores globales pertenece al dominio $\bar{X}$. Posteriormente, tendríamos que evaluar $f(x,y,z)$ en cada uno de los planos que delimitan el dominio para obtener los mínimos en ellos, posteriormente en las intersecciones de estos planos y acabando con los vértices de $\bar{X}$, tarea que se presenta tediosa e innecesariamente compleja de realizar de manera analítica.

Podría entonces interesarnos no tanto la obtención del valor o valores exactos sino algunos suficientemente aproximados. Para ello planteamos el concepto de algoritmo heurístico el cual, según la Real Academia de Ingeniería ([10]), se define como aquel algoritmo que «*descarta el encontrar soluciones óptimas debido a la dificultad que ello conlleva en aras de encontrar soluciones suficientemente satisfactorias, a pesar de que la solución encontrada pudiera ser arbitrariamente errónea en algunos casos*».

4. ALGORITMO CÚBICO

4.1. Planteamiento

En [3] se propone un algoritmo que va descomponiendo el espacio de búsqueda en subregiones disjuntas que se van subdividiendo en caso de que cumplan la condición de ser susceptibles de contener a algún minimizador, o que se descartan si se cumple cierta condición.

La única condición que ha de cumplir la función f, además de la continuidad, es que debe ser Lipschitziana sobre $\bar{X}$ [2], es decir, que exista una constante $A \geq 0$ [7] que verifique:

$$|f(x) - f(x')| \leq A|x - x'|, \ \forall x, x' \in \bar{X} \quad \text{[Ec.15]}$$

Por ser $\bar{X}$ un conjunto cerrado, existe un valor L que verifica la desigualdad anterior:

$$L = sup\left\{\frac{|f(x) - f(x')|}{|x - x'|} : \ x, x' \in \bar{X}, \qquad x \neq \ x'\right\} \quad \text{[Ec.16]}$$

Al valor $L = max\|\nabla f(x)\|; x \in \bar{X}$ se le conoce como constante de Lipschitz de f en $\bar{X}$.

4.2. Paso previo. Transformación al cubo unidad

Antes de comenzar a desarrollar el mallado de $\bar{X}$, se realizará una transformación afín [2] para llevar el cubo o n-rectángulo de definición de la función al cubo unidad $\bar{U}$ con un vértice en el origen de coordenadas para simplificar el cálculo:

$$x_i = a_i + (b_i - a_i)z_i, \qquad i = 1, \dots, n \qquad \text{[Ec.17]}$$

Siendo $z_i \in [0, 1]$ y recordando que el conjunto sobre el que actuará el algoritmo será:

$$\bar{X} = \{x \in R^n : a_i \leq x_i \leq b_i, \qquad i = 1, \dots, n\} \qquad \text{[Ec.18]}$$

Trabajaremos entonces con la función g [2] definida como:

$$g(z) = f[a + (b - a)z]; \quad z \in \bar{U} = \{z \in R^n : 0 \leq z_i \leq 1, \qquad i = 1, \dots, n\} \qquad \text{[Ec.19]}$$

Por tanto, el problema de encontrar

$$s^0 = \min f(x), \qquad x \in \bar{X} \qquad \text{[Ec.20]}$$

Se transforma en buscar

$$s^0 = \min g(z), \qquad z \in \bar{U} \qquad \text{[Ec.21]}$$

Con el conjunto de minimizadores

$$U^0 = \{z \in \bar{U} : g(z) = s^0\} \qquad \text{[Ec.22]}$$

Para obtener el conjunto de minimizadores equivalente en $\bar{X}$ basta con deshacer la transformación afín indicada previamente.

4.3. Pasos del algoritmo

Definimos en primer lugar como generador de particiones al proceso de dividir $X \subset R^n$ en N^n subconjuntos disjuntos $X_i \subset \mathrm{X}$ del mismo tamaño y tales que $\cup X_i \subset \bar{\mathrm{X}}$. Es decir, estos elementos tendrán una longitud de $1/N$ en cada lado y por tanto una diagonal del cubo de valor $\sqrt{3}/N$ (si estamos trabajando en R^3), una diagonal del cuadrado de valor $\sqrt{2}/N$ (en R^3), o una diagonal del n-cubo de valor $\sqrt{n}/N$ (en R^n).

Se elije una cota A de la pendiente de g según se indicó previamente (ver ecuación [Ec.15]), de manera que la pendiente de la función en cualquier punto del dominio de definición no sea superior a este. Esta cuestión es determinante ya que un valor excesivamente grande ralentizará enormemente el proceso de computación, pero un valor menor a la constante de Lipschitz de f en $\bar{X}$ puede suponer la eliminación de algunos puntos minimizadores (o todos) como se pondrá de manifiesto más adelante.

Para este valor A definimos el parámetro r que se utilizará para descartar nodos y no seguir explorando los subconjuntos que este pudiese generar. Al valor r se le denomina constante de eliminación [3] y se define, para la iteración j:

$$r_j = A\frac{\sqrt{n}}{N^j} \qquad \text{[Ec.23]}$$

Además, utilizaremos el mecanismo generador de mallado [3] de forma que asociaremos a cada elemento X_i anterior con uno de sus vértices. Concretamente, siguiendo el planteamiento en [2], se asigna a cada cubo el nodo correspondiente en el que se transformaría el origen tras realizar una traslación del cubo ubicado en $\boldsymbol{z} = 0$ hasta el cubo X_i.

Una vez tenemos la malla de N^n nodos, comenzamos a operar iterativamente los siguientes pasos:

1. Evaluamos la función g en todos ellos y obtenemos el valor mínimo, siendo s_j el correspondiente a la iteración j.
2. Comparamos la imagen de g en cada nodo con el valor s_j obtenido antes y con r_j según se definió en la expresión [Ec.23], de manera que eliminaremos aquellos nodos en los que $g\left(z_j{}^i\right) - s_j > r_j$ por tener constancia de que el mínimo no se puede encontrar en el cubo $\bar{X}_i{}^j$ (elemento al que se asoció el nodo $z_j{}^i$ en la iteración j [3].
3. En el nuevo subconjunto de elementos $\bar{X}_i{}^j$ se vuelve a realizar un mallado como el inicial, de manera que cada elemento $\bar{X}_i{}^j$ se descompone en N^n elementos obteniendo un nuevo conjunto de $\bar{X}_i{}^{j+1}$ cubos, volviendo a asociar a cada uno el nodo correspondiente $z_i{}^{j+1}$.
4. Repetimos los pasos 1 a 3 hasta llegar a un número prefijado de iteraciones o hasta obtener la precisión deseada, siendo esta precisión $\eta_j = 3r_j = 3A\frac{\sqrt{n}}{N^j}$.
5. El conjunto resultante de elementos $\bar{X}_i$ se devuelve como conjunto de minimizadores deshaciendo la transformación afín de los nodos z_i para volver a los equivalentes x_i en el n-rectángulo definido inicialmente.

Es interesante además incorporar un punto de control al finalizar para eliminar los puntos susceptibles de ser considerados el mismo al igual que se propone en [1] como procedimiento de limpieza.

4.4. Código en Python

El código de salida y entrada de datos se muestra en el apéndice A-1 en la función *Cubico()*, siendo las variables de entrada para este procedimiento las siguientes:

```
def Cubico(f, intervalo, tol, N, Ag, Escala=1):
```

f: función a minimizar

intervalo: n-rectángulo de búsqueda. Se define mediante los dos vértices opuestos de la región $[a, b]$ tales que

$$a, b \in R^n: \quad a_i < b_i \;\; \forall \;\; i = 1, \dots, n$$

tol: tolerancia deseada.

N: Número de divisiones de cada cubo en cada nueva iteración.

Ag: factor de pendiente según se ha indicado en el planteamiento del algoritmo.

Escala: Cuando la pendiente de la función alcanza valores muy grandes se pueden producir problemas de convergencia. Para solucionarlo, se divide la función objetivo por este valor durante la búsqueda de mínimos para poder trabajar con valores de Ag menores. Por defecto (si el usuario no indica un valor que lo sobrescriba) el valor predeterminado se tomará como 1.

Junto a la salida numérica de resultados se reflejará la evolución gráfica del conjunto de minimizadores (en los casos de 1 o 2 variables de entrada) para observar su evolución en cada iteración.

4.5. Aplicación

Ejecutemos el código anterior al caso del ejemplo 3 mencionado en el punto 0. La entrada de datos en el programa sería la siguiente:

```
'''
---------------------------------------------------------------------------------
DEFINICIONES DE LOS PARÁMETROS A UTILIZAR EN EL ALGORITMO
---------------------------------------------------------------------------------
'''

#definimos el intervalo en el que vamos a buscar el mínimo y que representaremos
gráficamente
#Valores mínimos X, Y, Z...
X0=[1,0,1]
#Valores máximos X, Y, Z...
X1=[5,3,3]
#Indicamos la tolerancia deseada para detener la búsqueda
Tolerancia=1e-7
#Indicamos el número de divisiones que se realizarán de cada elemento con cada
iteración
N=10
#Parámetro para determinar la constante de eliminación (solo para algoritmo cúbico)
Ag=0.1
#Factor de escala de la función
valorescala=100
#Indicamos qué función vamos a minimizar de las anteriormente definidas
funcion_optimizar = Ejemplo2
nombrefuncion="Ejemplo 2"

MostrarGrafico=True
```

Habiendo definido la función Ejemplo2 como sigue:

```
def Ejemplo2(x):    #3 dimensiones
    return 1+abs((x[0]**2+x[1]**2+x[2]**2)**3-3*(x[0]**2+x[1]**2+x[2]**2))
```

La salida de datos que nos brindaría la ejecución del código[5] es la siguiente:

```
Se ejecutará el algoritmo cúbico en el intervalo ([1, 0, 1], [5, 3, 3]) con los
siguientes parámetros:
Numero de divisiones por iteración= 10, Ag=0.1, valor de escala de la función=100,
Tolerancia= 1e-07
```

[5] El código completo se muestra en el apéndice A-1

```
Se prevén inicialmente 8 iteraciones
Iteracion: 1    r = 0.017320508076      long.intervalo: 0.1
NºNodos al inicio de la iteración: 1000
Iteracion: 2    r = 0.0017320508        N=10    long.intervalo: 0.01
NºNodos al inicio de la iteración: 2
Iteracion: 3    r = 0.0001732051        N=10    long.intervalo: 0.001
NºNodos al inicio de la iteración: 5
Iteracion: 4    r = 1.73205e-05 N=10    long.intervalo: 0.0001
NºNodos al inicio de la iteración: 15
Iteracion: 5    r = 1.7321e-06  N=10    long.intervalo: 1e-05
NºNodos al inicio de la iteración: 47
Iteracion: 6    r = 1.732e-07   N=10    long.intervalo: 1.0000000000000002e-06
NºNodos al inicio de la iteración: 147
Iteracion: 7    r = 1.73e-08    N=10    long.intervalo: 1.0000000000000002e-07
NºNodos al inicio de la iteración: 463
El mínimo de la función mediante el algoritmo cúbico es 3.0 y se alcanza en el conjunto de minimizadores:
[(1.0, 0.0, 1.0)]
El tiempo total de ejecución han sido 1.64 segundos
```

Nos indica, además del valor mínimo de la función y el punto en el que se alcanza, que el tiempo de procesamiento ha sido inferior a 2 segundos, algo impensable a la hora de plantear el procedimiento analítico del apartado 0.

5. OTRAS VENTAJAS

Además de haber justificado la necesidad a la hora de ejecutar los cálculos, las herramientas informáticas nos pueden proporcionar un sinfín de ayudas para interpretar o procesar la información resultante.

Por ejemplo, tomemos la función siguiente:

$$f(x,y) = 1 + |(4x^2 + y^2)^3 - 3x^2 - 3y^2| \qquad \text{[Ec.24]}$$

A priori parece presentar una estructura similar a los ejemplos 1 y 2 (según las expresiones [Ec.1] y [Ec.8], respectivamente), pero ese factor 4 multiplicando a la primera x imposibilita el cambio de variables realizado en el punto 2.2 para simplificar las operaciones.

Como ya hemos comprobado, podemos definir esta función en Python para ejecutar el algoritmo desarrollado con el objetivo de encontrar el conjunto de minimizadores. Tendríamos la siguiente entrada de datos:

```
def Ejemplo3(x):     #2 dimensiones
    return 1+abs((4*x[0]**2+x[1]**2)**3-3*(x[0]**2+x[1]**2))
```

```
#definimos el intervalo en el que vamos a buscar el mínimo y que representaremos gráficamente
#Valores minimos X, Y, Z...
X0=[-1.5,-1.5]
#Valores máximos X, Y, Z...
X1=[1.5,1.5]
#Indicamos la tolerancia deseada para detener la búsqueda
Tolerancia=1e-4
#Indicamos el número de divisiones que se realizarán de cada elemento con cada iteración
N=10
#Parámetro para determinar la constante de eliminación (solo para algoritmo cúbico)
Ag=0.25
```

```
#Factor de escala de la función
valorescala=300
#Indicamos qué función vamos a minimizar de las anteriormente definidas
funcion_optimizar = Ejemplo3
nombrefuncion="Ejemplo 3"

MostrarGrafico=True
```

Con ella, se obtendría la siguiente salida de resultados, en la cual se han eliminado bastantes líneas del texto que se devuelven en la solución ya que el conjunto de minimizadores en este caso se trata de un continuo formado por infinitos puntos[6]:

```
Se ejecutará el algoritmo cúbico en el intervalo ([-1.5, -1.5], [1.5, 1.5]) con los
siguientes parámetros:
Numero de divisiones por iteración= 10, Ag=0.25, valor de escala de la función=300,
Tolerancia= 0.0001
Se prevén inicialmente 5 iteraciones
Iteracion: 1    r = 0.035355339059      long.intervalo: 0.1
N°Nodos al inicio de la iteración: 100
Iteracion: 2    r = 0.0035355339        N=10    long.intervalo: 0.01
N°Nodos al inicio de la iteración: 42
Iteracion: 3    r = 0.0003535534        N=10    long.intervalo: 0.001
N°Nodos al inicio de la iteración: 1988
Iteracion: 4    r = 3.53553e-05 N=10    long.intervalo: 0.0001
N°Nodos al inicio de la iteración: 24331
Iteracion: 5    r = 3.5355e-06  N=10    long.intervalo: 1e-05
N°Nodos al inicio de la iteración: 240407
El mínimo de la función mediante el algoritmo cúbico es 1.0 y se alcanza en el
conjunto de minimizadores:
[(-0.387, 0.915), (0.451, -0.644), (0.47, -0.22), (-0.465, -0.035), (-0.164,
1.258), (0.159, 1.262), (0.454, -0.621), (-0.451, -0.648), (-0.468, -0.14), (0.004,
-0.002), (0.47, -0.427), (0.457, -0.598), (0.091, 1.298), (0.002, -0.016), (0.471,
0.334),
```

[…]

```
, (0.099, 1.295), (-0.468, 0.143), (-0.265, -1.154), (0.47, 0.231), (0.469, -
0.439), (0.002, 0.017), (-0.439, 0.719), (-0.459, -0.581), (-0.47, -0.408), (0.346,
-1.016), (-0.465, 0.044), (0.345, -1.019), (0.091, -1.299), (0.448, -0.667)]
El tiempo total de ejecución han sido 61.65 segundos
```

Sin embargo, y aquí viene lo interesante, podemos pedir que, una vez finalizado el algoritmo, nos muestre no sólo los puntos que forman este conjunto de minimizadores buscados sino también la evolución de este en cada una de las iteraciones. Para el ejemplo anterior, esta salida se muestra en la Imagen 3.

En este caso, la representación gráfica de los puntos puede ser de mucha más ayuda que el listado de puntos, al menos de cara a la interpretación de los resultados.

[6] En estos casos, reducir la tolerancia no supone una convergencia a la solución final sino una mayor partición en los puntos que conforman la solución e incluso un posible colapso del sistema. Hay que tantear con diferentes valores de Ag, factor de escala y N para ir aproximándonos a la solución deseada con eficacia.

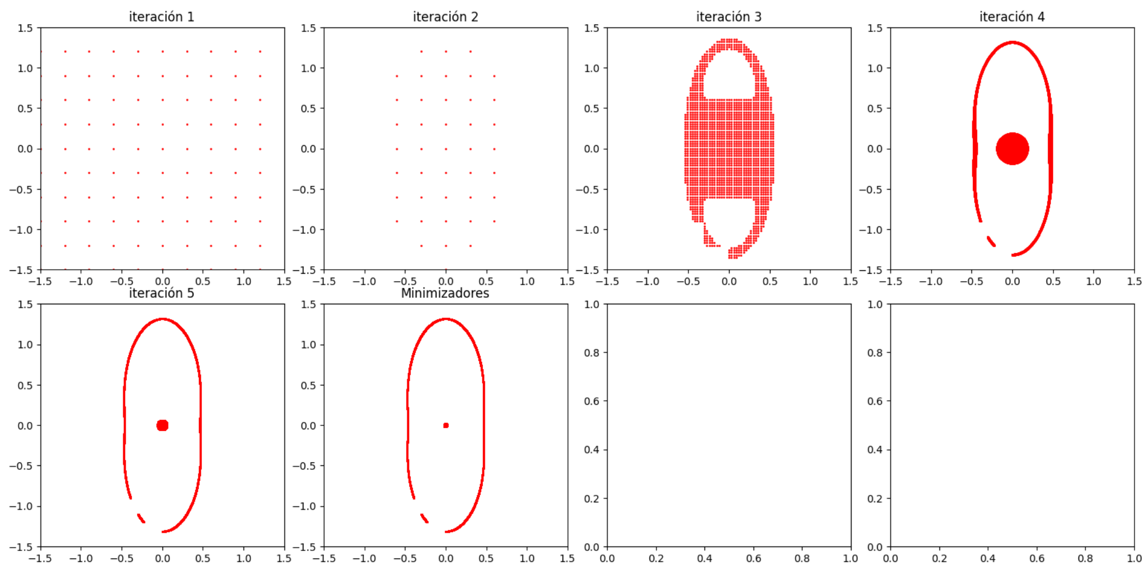

Imagen 3: Representación gráfica de la evolución del conjunto de minimizadores en cada iteración.

6. CONCLUSIONES

A lo largo de este documento se ha planteado la búsqueda del mínimo absoluto de una función cualquiera, conocida su expresión algebraica independientemente de si presenta o no convexidad.

Se ha visto en el apartado 2 que para expresiones sencillas de la función este proceso puede hacerse manualmente haciendo uso de las herramientas habituales del cálculo. No obstante, ya se refleja en el apartado 0 cómo esta complejidad aumenta cuando comenzamos a establecer condiciones de contorno a los límites del dominio de la función objetivo.

Surge entonces la necesidad de emplear herramientas informáticas para ayudarnos en esta tarea, como el denominado algoritmo [3][11] que se desarrolló en el punto 4. Con él, dividimos el espacio en el que está definida la función en subespacios más pequeños que quedan representados por uno de sus vértices. Mediante un criterio de eliminación que depende de un parámetro Ag relacionado con la constante de Lipschitz y con el módulo del gradiente, se descartarán algunas de esas divisiones y en las restantes se volverán a subdividir, evaluar y eliminar de manera sucesiva.

En varios ejemplos se ha comprobado la eficacia de esta herramienta, tanto de cara al tiempo de cálculo como a la representación gráfica de los resultados obtenidos (como se reflejó en los apartados 4.5 y 5) siendo esa representación inabordable en la mayoría de los casos.

Con todo lo anterior se manifiesta lo determinante que es el conocimiento de lenguajes de programación que se puedan emplear para facilitar este tipo de operaciones complejas, repetitivas o simplemente imposibles de abordar de otro modo, en cualquiera de las ramas laborales en las que las matemáticas tengan un peso determinante.

BIBLIOGRAFÍA

[1] Delgado M. y Galperin E.A. (2003): *Global optimization in Rn with box constraints and applications: A Maple code*. Mathematical and Computer Modelling.

[2] Delgado M. (2008): *Optimizacion global de funciones continuas no diferenciables*. 100cias@uned. Vida científica. Colaboraciones en matemáticas.

[3] Galperin, E (1985): *The Cubic Algorithm*. Journal of Mathematical Analysis and Applications.

[4] Matplotlib (2012–2023) https://matplotlib.org/stable/users/getting_started/

[5] Nocedal J. y Wright S.J. (1999): *Numerical Optimization*. Springer Series in Operations Research.

[6] Numpy (2023) https://numpy.org/install/

[7] Payá R. *Continuidad uniforme*. Universidad de Granada: Apuntes Cálculo II (Tema 6).

[8] Python (2001-2023) https://www.python.org/downloads/

[9] Ramos, E. (2020): *Modelización*, Editorial Sanz y Torres.

[10] Real Academia de Ingeniería (2023) https://diccionario.raing.es/es/lema/algoritmo-heur%C3%ADstico#:~:text=Definici%C3%B3n%3A,arbitrariamente%20err%C3%B3nea%20en%20algunos%20casos.

[11] Scipy (2008-2023) https://docs.scipy.org/doc/scipy/

BIBLIOGRAFÍA COMPLEMENTARIA

Baudin, M (2011): Introduction to Unconstrained Optimization.

Campos, J - C.P.S. Esquemas algorítmicos - Ramificación y acotación.

Delgado Pineda M. Galperin E.A. (2007) *Maple code of gamma algorithm for global optimization of uncertain functions over compact robust sets*. ELSEVIER

Delgado Pineda M. et al. (2013): *Maple code of the cubic algorithm for multiobjective optimization with box constraints*. Numerical algebra, control and optimization.

Liberti, L (2008): *Introduction to Global Optimization*. LIX, École Polytechnique, Palaiseau F-91128, France

Moreno, C. (2014): Introducción al cálculo numérico (Edición digital), Universidad Nacional de Educación a Distancia.

Palacios Gutiérrez, F (2008). *Diseño óptimo aerodinámico a través del método adjunto continuo*. Universidad Autónoma de Madrid.

Ramos, E. (2012): Programación lineal y entera, Ediciones académicas.

William H. Press et al. (2002) *Numerical Recipes in C. The Art of Scientific Computing*. Press Syndicate of the University of Cambridge.

A-1 APÉNDICE 1. CÓDIGO EN PYTHON DEL PROGRAMA DE OPTIMIZACIÓN

```
'''
==========================================================================================
            Fragmento del código empleado en el trabajo fin de grado:
                "OPTIMIZACIÓN DE FUNCIONES NO NECESARIAMENTE CONVEXAS NI DIFERENCIABLES.
                                DESARROLLO DE ALGORITMOS EN PYTHON."

            Alumno: Francisco Salamanca González
            Universidad Nacional de Educación a Distancia. Grado en Matemáticas
            Curso 2022-2023
==========================================================================================
'''

import numpy as np
import math
import matplotlib.pyplot as plt
import itertools
import functools
import time

'''
==========================================================================================
                                    ALGORITMO CÚBICO
==========================================================================================
Función que tratará de encontrar el mínimo de la función f en el intervalo establecido.
El siguiente algoritmo divide el dominio de definición en una malla de NxN cubos. Asignará a
cada cubo el valor de la función
en uno de sus vértices y descartará aquellos para los que su imagen sea mayor que una
constante de eliminación que se
calculará para cada iteración. Con los cubos restantes se volverá a proceder a su subdivisión
y así sucesivamente hasta
obtener la precisión deseada.
==========================================================================================
'''
def Cubico(f, intervalo, tol, N, Ag, Escala=1):

    #Precision de los valores que devolveremos:
    Decimales=int(round(-math.log(tol, 10),0))

    #obtenemos los valores extremos del intervalo
    a, b = intervalo

    #n es la dimensión del espacio en el que trabajamos
    n=(len(a))

    #vértices del cubo inicial
    x0=[0]*n
    x1=[1]*n

    #Determinamos el número de iteraciones prevista inicialmente en función de la tolerancia
deseada
    Lados_Box=list(map(lambda x1,x2: x2-x1 ,a,b))
    Long_max=functools.reduce(lambda a, b: a if a > b else b, Lados_Box)
    N_total_iteraciones=min(math.trunc(np.log(Long_max/tol)/np.log(N)+1), 15)
    EstadoActual=(f"Se prevén inicialmente {N_total_iteraciones} iteraciones")
    print(EstadoActual)

    #Transformacion afín    x=a+(b-a)z
    def g(z):
        x=list(map(lambda ai,bi,zi: ai+(bi-ai)*zi ,a,b,z))
        return f(x)/Escala

    #generamos una lista con el mallado en N segmentos en cada dirección en la que se haya
definido la función
    mallado=[]
    Long_intervalo=1/N
    for j in range(n):
        mallado_direccion=[]
        for e in range(N):
            #aseguramos que pertenece al dominio
            mallado_direccion.append(Long_intervalo*e)
        mallado.append(mallado_direccion)

    #Establecemos el listado inicial de nodos con esta primera división
    nodos = list(itertools.product(*mallado))
```

```
    #definimos un listado para ir almacenando las diferentes iteraciones de los nodos y 
mostrarlas gráficamente
    iteraciones=[]
    iteraciones.append(nodos)

    #Constante de eliminación
    r=Ag*np.sqrt(n)/N

    #Indicamos por pantalla y registramos el estado inicial
    EstadoActual=(f"Iteracion: 1\tr = {round(r, 12)}\tlong.intervalo: {1/N}\nN°Nodos al inicio 
de la iteración: {len(nodos)}")
    print(EstadoActual)

    #Calculamos todas las imágenes de los nodos de la partición
    N_iter=1
    imagenes = list(map(lambda nodo : g(nodo), nodos))

    #Valores mínimos obtenidos en cada iteración se almacenan en la lista 's'
    s=[]
    f_min=functools.reduce(lambda a, b: a if a < b else b, imagenes)
    s.append(round(f_min,Decimales))

    #eliminamos los nodos en los que no se encontrarán los mínimos globales aplicando el 
criterio de eliminación
    #Almacenamos en la lista nuevos_nodos a aquellos nodos en curso
    nuevos_nodos=[]
    nuevos_nodos=list(filter(lambda nodo : not(g(nodo)-f_min>r), nodos))

    #Eliminamos posibles nodos duplicados
    nodos = list(set(nuevos_nodos))

    #Almacenamos el conjunto de nodos al listado que va guardando la evolución de 
minimizadores
    iteraciones.append(nodos)

    #repetimos el bucle hasta que la distancia a la que creamos los nuevos nodos sea menor que 
la tolerancia
    while N_iter<15 and 3*r>tol:

        N_iter+=1

        #Si el número de nodos es mayor a 10.000.000 se sale del proceso para evitar colapso 
del sistema
        if len(nodos)>10000000:
            EstadoActual=(f"La cantidad de nodos en este momento es mayor a 10.000.000. Se han 
interrumpido las iteraciones para evitar colapso del equipo.")
            (EstadoActual)
            break

        #Volvemos a inicializar la variable nuevos nodos para almacenar la nueva partición que 
generemos en esta iteración
        nuevos_nodos=[]

        #Constante de eliminación
        r=r/N

        #Definimos el nuevo tamaño de intervalo para la iteración
        Long_intervalo=Long_intervalo/N

        #secuencia de control:
        EstadoActual=(f"Iteracion: {N_iter}\tr = {round(r,10)}\tN={N}\tlong.intervalo: 
{Long_intervalo}\nN°Nodos al inicio de la iteración: {len(nodos)}")
        (EstadoActual)

        #por cada nodo almacenado, generamos una nueva partición
        for nodo in nodos:
            nodos_temporales=[]
            mallado=[]
            #bucle por cada variable en la que esté definida el dominio
            for j in range(n):
                mallado_direccion=[]
                for e in range(N):
                    mallado_direccion.append(nodo[j] + Long_intervalo*e)
                mallado.append(mallado_direccion)
            nodos_temporales=list(itertools.product(*mallado))
            for nodo_temp in nodos_temporales:
```

```
                nuevos_nodos.append(nodo_temp)

        #obtenemos el valor mínimo en la iteración
        imagenes = list(map(lambda nodo : g(nodo), nodos))
        f_min=functools.reduce(lambda a, b: a if a < b else b, imagenes)
        s.append(round(f_min,Decimales))

        #actualizamos el listado 'nodos' eliminando los que cumplan g(z)-s>r
        nodos=list(filter(lambda nodo : not(g(nodo)-f_min>r), nuevos_nodos))

        #Almacenar los puntos en un vector para representar al final del proceso
        #en caso de que sean muchos puntos, de cara a la visualización de resultado, se
tomarán hasta quedarnos con un máximo de 100.000
        if len(nodos)>100000:
            factorEscala=max(math.trunc(len(nodos)/100000)+1, 2)
            nodos_temporales=[]
            for i in np.arange(1, len(nodos), factorEscala):
                nodos_temporales.append(nodos[i])
            iteraciones.append(nodos_temporales)
        else:
            iteraciones.append(nodos)

    #Deshacer transformación afín para devolver la solución del conjunto de minimizadores
    minimizadores=[]
    for nodo in nodos:
        nodo_temp=list(map(lambda ai,bi,zi: ai+(bi-ai)*zi ,a,b,nodo))
        nodo_temp=tuple(round(x,3) for x in nodo_temp)
        minimizadores.append(nodo_temp)

    #eliminar poribles duplicados y calcular el mínimo absoluto encontrado
    minimizadores = list(set(minimizadores))
    imagenes = list(map(lambda nodo : f(nodo), minimizadores))
    f_min=functools.reduce(lambda a, b: a if a < b else b, imagenes)

    #Devolvemos el conjunto de minimizadores obtenido, el valor mínimo encontrado en la caja
de definición,
    # una lista s con la evolución de dicho valor mínimo, el error en la última iteración y el
número de iteraciones finales.
    return minimizadores, f_min, iteraciones, s, round(3*r, Decimales+2), N_iter

'''
-----------------------------------------------------------------------------------------------
FUNCIONES PARA REPRESENTAR RESULTADOS
-----------------------------------------------------------------------------------------------
'''

#función para mostrar la evolución del conjunto de minimizadores con cada iteración
#Sólo se ejecutará cuando estemos hablando de funciones con una o dos variables independientes
def representar_iteraciones(puntos, intervalo, f, MostrarGrafico=True, NombreGuardado=""):


    a, b = intervalo

    #Deshacer transformación afín
    iteraciones_unitarias=puntos
    iteraciones=[]
    for iteracion in iteraciones_unitarias:
        nodos_iteracion=[]
        for nodo in iteracion:
            nodo_temp=list(map(lambda ai,bi,zi: ai+(bi-ai)*zi ,a,b,nodo))
            nodo_temp=tuple(round(x,3) for x in nodo_temp)
            nodos_iteracion.append(nodo_temp)
        iteraciones.append(nodos_iteracion)
    puntos=iteraciones

    serie1=puntos[0]
    punto1=serie1[0]
    #Dimensión del espacio definido
    Dim=len(punto1)

    if Dim <=2:

        #definimos número de gráficos de salida
        NumeroColumnas=4
        NumeroIteraciones=len(puntos)
        NumeroFilas=NumeroIteraciones/NumeroColumnas
        if not(NumeroIteraciones%NumeroColumnas ==0):
```

```
            NumeroFilas=math.trunc(NumeroFilas)+1
        else:
            NumeroFilas=int(NumeroFilas)

        #Configurar ventanas con gráficos
        fig, axs = plt.subplots(NumeroFilas, NumeroColumnas, figsize=(5*NumeroColumnas,
5*NumeroFilas))

        if Dim==2:
            # Iterar sobre cada conjunto de datos y graficarlos en un subplot
            for i, ax in enumerate(axs.flat):
                if(i<len(puntos)):

                    x = [punto[0] for punto in puntos[i]]
                    y = [punto[1] for punto in puntos[i]]

                    ax.scatter(x, y, 1, color='red')

                    ax.set_xlim(a[0],b[0])
                    ax.set_ylim(a[1],b[1])

                    if i==len(puntos)-1:
                        ax.set_title('Minimizadores')
                    else:
                        ax.set_title(f'iteración {i+1}')
        elif Dim==1:

            #Gráfica de la función
            x_f = np.arange(a[0], b[0], (int(b[0]-a[0])+1)/500)
            y_f = [f([i,]) for i in x_f]

            #límites verticales de los ejes
            f_max = functools.reduce(lambda a, b: a if a > b else b, y_f)
            f_min=f_max

            for i, ax in enumerate(axs.flat):
                if(i<len(puntos)):
                    #Dispersión de los nodos de cada iteración
                    x = [punto[0] for punto in puntos[i]]
                    y = [f([i,]) for i in x]
                    f_min = functools.reduce(lambda a, b: a if a < b else b, y)
                    ax.scatter(x, y, 15, color='red')

                    #Gráfica de la función
                    ax.plot(x_f,y_f,color='grey')

                    #Título de la gráfica
                    if i==len(puntos)-1:
                        ax.set_title('Minimizadores')
                    else:
                        ax.set_title(f'iteración {i+1}')


            for i, ax in enumerate(axs.flat):
                #Límites de los ejes
                ax.set_xlim(a[0]-0.1*(b[0]-a[0]), b[0]+0.1*(b[0]-a[0]))
                ax.set_ylim(f_min-0.1*(f_max-f_min), f_max+0.1*(f_max-f_min))

        #Guardar gráfico
        if not(NombreGuardado==""):
            plt.savefig(NombreGuardado)

        # Mostrar la figura
        plt.tight_layout()
        if MostrarGrafico:
            plt.show()
        else:
            plt.close()
'''
FUNCIONES AUXILIARES DE SALIDA DE RESULTADOS
'''

def
Registrar_inicio_algoritmo_CU(algoritmo,intervalo,N,Tolerancia,Ag,valorescala,nombrefuncion):
    EstadoActual=(f"\n\nFunción {nombrefuncion}\nSe ejecutará el algoritmo {algoritmo} en el
intervalo {intervalo} con los siguientes parámetros:\n"+
```

```
    f"Numero de divisiones por iteración= {N}, Ag={Ag}, valor de escala de la
función={valorescala}, Tolerancia= {Tolerancia}\n")
    print(EstadoActual)

def
Registrar_solucion_algoritmo(tiempofinal,tiempoinicio,x_min,f_min,Ag,N,valorescala,Tolerancia,
errorfinal,Niteraciones,nombrefuncion,iteraciones,intervalo,funcion_optimizar,MostrarGrafico=T
rue, Unitario=False):
    tiempoTotal=round(tiempofinal-tiempoinicio,2)
    if len(x_min)>50:
        Minimizadores="Muchos puntos"
    else:
        Minimizadores=x_min

    ' indicar solución'
    EstadoActual=(f"El mínimo de la función mediante el algoritmo cúbico es {round(f_min,4)} y
se alcanza en el conjunto de minimizadores: \n{x_min}\n" +
    f"El tiempo total de ejecución han sido {round(tiempofinal-tiempoinicio,2)} segundos")
    print(EstadoActual)

    'Representar gráfica de la evolución de minimizadores'
    NombreArchivo=(f'N={N}_Ag={Ag}_tol_{Tolerancia}.jpg')
    try:
        representar_iteraciones(iteraciones, intervalo, funcion_optimizar, True,
NombreArchivo)
    except:
        print('No se pudo generar la gráfica ' + NombreArchivo)

'''
-----------------------------------------------------------------------------------------------
A continuación se definen las funciones que se van a estudiar
-----------------------------------------------------------------------------------------------
'''

def Ejemplo1(x):    #1 dimensión
    return 1+abs(x[0]**3+-3*x[0])

def Ejemplo2(x):    #2 dimensiones
    return 1+abs((x[0]**2+x[1]**2)**3-3*(x[0]**2+x[1]**2))

def Ejemplo3(x):    #3 dimensiones
    return 1+abs((x[0]**2+x[1]**2+x[2]**2)**3-3*(x[0]**2+x[1]**2+x[2]**2))

'''
-----------------------------------------------------------------------------------------------
DEFINICIONES DE LOS PARÁMETROS A UTILIZAR EN EL ALGORITMO
-----------------------------------------------------------------------------------------------
'''

#definimos el intervalo en el que vamos a buscar el mínimo y que representaremos gráficamente
#Valores minimos X, Y, Z...
X0=[1,0,1]
#Valores máximos X, Y, Z...
X1=[5,3,3]
#Indicamos la tolerancia deseada para detener la búsqueda
Tolerancia=1e-7
#Indicamos el número de divisiones que se realizarán de cada elemento con cada iteración
N=10
#Parámetro para determinar la constante de eliminación (solo para algoritmo cúbico)
Ag=0.1
#Factor de escala de la función
valorescala=100
#Indicamos qué función vamos a minimizar de las anteriormente definidas
funcion_optimizar = Ejemplo3
nombrefuncion="Ejemplo 3"

MostrarGrafico=True

'''
-----------------------------------------------------------------------------------------------
EJECUCIÓN DEL ALGORITMO
-----------------------------------------------------------------------------------------------
'''

#Obtener minimizadores de la función f elegida
intervalo=(X0,X1)
tiempoinicio=time.time()
```

```
print(f"Se ejecutará el algoritmo cúbico en el intervalo {intervalo} con los siguientes parámetros:")
print(f"Numero de divisiones por iteración= {N}, Ag={Ag}, valor de escala de la función={valorescala}, Tolerancia= {Tolerancia}")
x_min, f_min, iteraciones, s, errorfinal, Niteraciones = Cubico(funcion_optimizar, intervalo, Tolerancia, N, Ag, valorescala)
DeshacerAfinidad=True

tiempofinal=time.time()
Registrar_solucion_algoritmo(tiempofinal,tiempoinicio,x_min,f_min,Ag,N,valorescala,Tolerancia,
errorfinal,Niteraciones,nombrefuncion,iteraciones,intervalo,funcion_optimizar,MostrarGrafico,
DeshacerAfinidad)

print("\n\n\nPROCESO FINALIZADO")
```