Trabajo Práctico Nº3: Heurísticas.

Optimización – Primer Cuatrimestre 2017

Maia Numerosky y Matías Zylbersztejn

26 de mayo de 2017

1. Introducción

El objetivo de este trabajo es estudiar los algoritmos heurísticos vistos en la materia. Estudiaremos los siguientes:

1. *a*) Descenso aleatorio
   b) Descenso aleatorio con búsqueda local
   c) Descenso aleatorio con búsqueda local iterada
2. Algoritmo genético
3. Recocido simulado
4. Algoritmo híbrido evolutivo y de descenso aleatorio con búsqueda local

2. Implementación y uso

El usuario debe ingresar como *input* una función para iniciar el algoritmo. En cada sección están especificados los argumentos adicionales. Los argumentos opcionales se ingresan entre comillas en cualquier orden. Por ejemplo, `metodo(f,'MaxIter','10000','A','[0,2;0,2]')`. Si se quiere testear una función que esté guardada en una carpeta, se debe ingresar con `@`. Por ejemplo, si quiero estudiar `funcion.m`, debo ingresar `metodo(@funcion,opcionesExtra)`.

Dentro de los archivos enviados incluimos la función `rosenbruck` que sirve para testear.

Además de eso, incluimos un archivo llamado `corredor.m`. El mismo sirve para correr cualquiera de los algoritmos y graficar los resultados, tanto el valor de la función iteración por iteración como las curvas de nivel de la función y la evolución de las iteraciones. En el caso de tener un dominio mayor a dos dimensiones, se grafican las componentes correspondientes a las primeras dos coordenadas. En el *script* sólo hay que ir modificando el algoritmo que se quiere testear junto con los argumentos que el mismo requiere.

Este *script* también arroja el minimizador y el mínimo que encontró junto con el tiempo que demoró.

3. Algoritmos

3.1. Descenso aleatorio (con sus variantes)

3.1.1. Descenso aleatorio

Input: `descenso_aleatorio(f,x0)`

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0,1]^d$), `n` (la cantidad de puntos aleatorios a generar, por defecto es 100), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite de tiempo, por defecto es 300 segundos).

Funcionamiento: Este algoritmo intenta minimizar una función $f : A \to \mathbb{R}^d$ donde A es un rectángulo ($A = [a_1, b_1] \times \ldots \times [a_d, b_d]$ y se ingresa como una matriz de $d \times 2$).

El punto x_0 en A que ingresa el usuario será el candidato a mínimo (`minimizador_viejo`). Luego, se generan n puntos al azar en A, se toma el mejor de ellos (el que minimiza la función) y se compara con `minimizador_viejo`. El mejor entre esos dos será el nuevo `minimizador_viejo`, y se vuelve a comenzar generando n puntos al azar en A. Repite este proceso *MaxIter* veces o hasta que se acabe el tiempo.

3.1.2. Descenso aleatorio con búsqueda local

Input: `busqueda_local(f,x0)`

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0,1]^d$), `n` (la cantidad de puntos aleatorios a generar, por defecto es 100), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite de tiempo, por defecto es 300 segundos), `alpha` (la magnitud en que achico el dominio, por defecto es 0.05).

Funcionamiento: El objetivo de este algoritmo es el mismo que el del anterior.

El punto x_0 en A que ingresa el usuario será el candidato a mínimo (`minimizador_viejo`) . Luego, se construye una nueva "caja" (una nueva A) alrededor de este punto (sumándole y restándole *alpha*) y se realiza descenso aleatorio con esta nueva A. Repite *MaxIter* veces o hasta que se acabe el tiempo.

3.1.3. Descenso aleatorio con búsqueda local iterada

Input: `busqueda_local_iterada(f,x0)`

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0,1]^d$), `n` (la cantidad de puntos aleatorios a generar, por defecto es 100), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite de tiempo, por defecto es 300 segundos), `alpha` (la magnitud en que achico el dominio, por defecto es 0.05).

Funcionamiento: El objetivo de este algoritmo no cambia, pero lidia con el siguiente asunto: el algoritmo anterior corre el riesgo de quedarse cerca de mínimos locales, por eso puede ser una buena idea recomenzar cada cierta cantidad de iteraciones en un punto aleatorio y lejano. En [1] se entiende muy bien:

> *"If the size is very small, then Hill-Climbing will march right up a local hill and be unable to make the jump to the next hill because the bound is too small for it to jump that far. Once it's on the top of a hill, everywhere it jumps will be worse than where it is presently, so it stays put. Further, the rate at which it climbs the hill will be bounded by its small size. On the other hand, if the size is large, then Hill-Climbing will bounce around a lot. Importantly, when it is near the top of a hill, it will have a difficult time*

converging to the peak, as most of its moves will be so large as to overshoot the peak." [1]

Lo que hace entonces este algoritmo es ejecutar *MaxIter* veces (o hasta que se acabe el tiempo), comenzando con un punto nuevo aleatorio cada vez, `busqueda_local(f,minimizador_viejo,iter,n,A,tlim,alpha)` .

3.2. Algoritmo genético

Input: `GA(f,popsize,d)` donde `popsize` es el tamaño de la población y `d` es la dimensión del dominio.

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0,1]^d$), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite de tiempo, por defecto es 300 segundos), `ParamSelec` (la cantidad de individuos seleccionados para hacer *crossover*, por defecto es 4), `crossover` (la función usada para hacer *crossover*, por defecto es `coverl`, se explicará más adelante), `padres` (la cantidad de padres que sobreviven, por defecto es 4).

Funcionamiento: Este algoritmo intenta minimizar una función $f : A \to \mathbb{R}^d$ donde A es un rectángulo ($A = [a_1, b_1] \times \ldots \times [a_d, b_d]$ y se ingresa como una matriz de $d \times 2$). Implementamos la versión con elitismo del algoritmo genético que explica [1].

Empezamos con una población aleatoria en A de una cantidad *popsize* de individuos. *Best* es, por ahora, el que minimiza la función entre esos individuos. Seleccionamos, además, los mejores (esta cantidad es la que llamamos `padres`). Después de eso, seleccionamos (usando la función `seleccion` que explicaremos después) dos padres y les hacemos *crossover*. A esos dos hijos los mutamos (cuidando que no se salgan del dominio) y ellos reemplazarán a sus padres. Repetimos este proceso `(popsize - padres)` $\div 2$ veces así nuestra nueva población está toda compuesta de estos hijos más los mejores de la generación anterior. Se reitera esto `MaxIter` veces o hasta que se supera el tiempo máximo.

Selección: Utilizamos lo que [1] llama *Tournament Selection* por su simpleza y rapidez. Se fija una cantidad `ParamSelec` que es menor que la cantidad total de la población. Se elige esa cantidad de individuos al azar con reposición y se toma el mejor de ellos (el que minimiza la función). Obviamente, en el algoritmo genético utilizamos este método una vez por cada padre que queremos elegir.

Crossover: Hay muchas maneras de hacer *crossover*. Programamos tres de ellas que encontramos en [1]:

- *Two-Point Crossover*, función `cover2`: si $P_a = [a_1, \ldots, a_d]$ y $P_b = [b_1, \ldots, b_d]$ se eligen al azar dos números entre 1 y d (por ejemplo k y l) y se intercambian las coordenadas comprendidas entre esos índices. Es decir, $C_a = [a_1, \ldots, b_k, \ldots, b_l, \ldots, a_d]$ y $C_b = [b_1, \ldots, a_k, \ldots, a_l, \ldots, b_d]$.

- *Uniform Crossover*, función `coveru`: se fija una probabilidad $P = \frac{1}{d}$ y para cada índice se genera un número al azar entre 0 y 1. Si ese número es menor que P, intercambiamos esa coordenada de P_a con la de P_b. La ventaja de esto con respecto al método anterior es el hecho de que todas las coordenadas tienen la misma chance de ser cambiadas.

- *Line recombination*, función `coverl`: se toman dos números al azar entre 0 y 1, α y β. $C_a = P_a + \alpha\,(P_b - P_a)$ y $C_b = P_a + \beta\,(P_b - P_a)$. Este método de *crossover* tiene la ventaja

[1] Hay que tener en cuenta que en este libro se buscan máximos en lugar de mínimos, por lo que toda la terminología está al revés.

de aprovechar el hecho de que trabajamos con vectores de números reales y no de ceros y unos.

Mutación: se fija una probabilidad $P = \frac{1}{d}$ y para cada índice se genera un número al azar entre 0 y 1. Si ese número es menor que P, se perturba esa coordenada sumándole un número aleatorio distribuido como $\mathcal{N}(0, 1)$.

3.3. Recocido simulado

Input: `recocido_simulado(f,x0)`

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0, 1]^d$), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite de tiempo, por defecto es 300 segundos), `temp` (temperatura inicial, por defecto es 1000), `k` (fracción de temperatura, por defecto es 1,05), `paso` (longitud de paso según la temperatura, por defecto es $\sqrt{t}$), `desc` (función para que la temperatura vaya descendiendo, por defecto es $t \div k$), `tempTol` (mínimo de temperatura, por defecto es 10^{-10}), `n` (cantidad de iteraciones en el ciclo interno, por defecto es 1000).

Funcionamiento: En este algoritmo podemos acotar el dominio de la función o no. Comenzamos con un x_0 que ingresa el usuario, y por ahora este es nuestro candidato a mínimo, lo llamamos S. Ahora repetimos n veces el siguiente proceso: generamos otro punto (R) perturbando el anterior en una cantidad `paso(temp)` $\cdot$ `randn(d)`, es decir que lo modificamos mediante una $\mathcal{N}(0, 1)$ pero si la temperatura es alta este número aumentará (siempre tenemos en cuenta que R no se salga del dominio). Después, generamos un número al azar y nos fijamos si es menor que $e^{\frac{f(S)-f(R)}{temp}}$. Si es así o si $f(R) < f(S)$, cambiamos S por R. Es decir que, si R es mejor que S, los intercambiamos sí o sí. Pero si es peor, podríamos cambiarlos igual, con una probabilidad $e^{\frac{f(S)-f(R)}{temp}}$. La razón de esto la explica [1] claramente:

> *If R is much worse than S, the fraction is larger, and so the probability is close to 0. If R is very close to S, the probability is close to 1. Thus if R isn't much worse than S, we'll still select R with a reasonable probability.*
>
> *Second, we have a tunable parameter t. If t is close to 0, the fraction is again a large number, and so the probability is close to 0. If t is high, the probability is close to 1. The idea is to initially set t to a high number, which causes the algorithm to move to every newly-created solution regardless of how good it is. We're doing a random walk in the space. Then t decreases slowly, eventually to 0, at which point the algorithm is doing nothing more than plain Hill-Climbing.* [2]

Luego de haber hecho esto n veces, hacemos descender la temperatura (por defecto, dividiéndola por k, pero podríamos usar otra función).

Repetimos todo esto $MaxIter$ veces o hasta que se acabe el tiempo.

3.4. Algoritmo híbrido evolutivo y de descenso aleatorio con búsqueda local

Input: `hibrido(f,popsize,d)` donde `popsize` es el tamaño de la población y `d` es la dimensión del dominio.

Argumentos opcionales: `A` (el dominio rectangular donde realizar la búsqueda, por defecto es $[0, 1]^d$), `MaxIter` (la cantidad límite de iteraciones a realizar, por defecto es 10000), `tlim` (límite

[2] La variable que llama t para nosotros es *temp*.

de tiempo, por defecto es 300 segundos), `ParamSelec` (la cantidad de individuos seleccionados para hacer *crossover*, por defecto es 4), `crossover` (la función usada para hacer *crossover*, por defecto es `coverl`, se explicará más adelante), `IterBusqueda` (la cantidad de iteraciones en la búsqueda local, por defecto es 10).

Funcionamiento: Este algoritmo intenta minimizar una función $f : A \to \mathbb{R}^d$ donde A es un rectángulo ($A = [a_1, b_1] \times \ldots \times [a_d, b_d]$ y se ingresa como una matriz de $d \times 2$). Combina el algoritmo genético y la búsqueda local de la siguiente manera:

Empieza con una población inicial de *popsize* individuos. A cada uno de ellos los mejora corriendo el algoritmo de búsqueda local una cantidad $IterBusqueda$ de iteraciones empezando desde ese punto, para que nuestra población inicial para el algoritmo genético sea buena. Después de eso, sigue con el algoritmo genético en su versión sin elitismo, es decir que cada vez que pasamos de una generación a la otra "matamos" a todos los padres. Pero no sólo eso sino que cuando empezamos una nueva generación mejoramos a todos los hijos usando búsqueda local.

Repetimos todo esto $MaxIter$ veces o hasta que se acabe el tiempo.

4. Resultados y observaciones

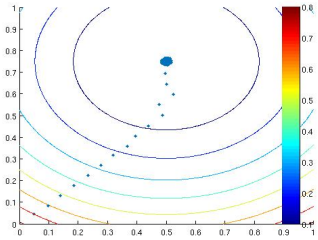
4.1. Función cuadrática

Nuestra primera función de *testeo* fue una muy simple y con buenas propiedades para ver si los algoritmos funcionaban bien en este caso: $f(x) = (x - 0{,}5)^2 + (y - 0{,}75)^2$. Obviamente, en este caso el mínimo es 0 y se alcanza en $(0{,}5; 0{,}75)$. El dominio en todos los casos fue el $[0, 1]^2$ y el $x_0 = (0; 0)$.

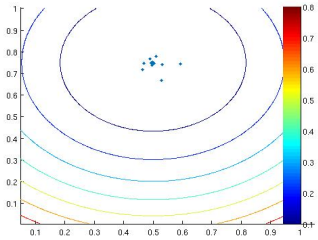
Decidimos correr todos los algoritmos durante 30 segundos para poder compararlos.

Seremos breves con esta función porque todos los algoritmos funcionaron bastante bien.

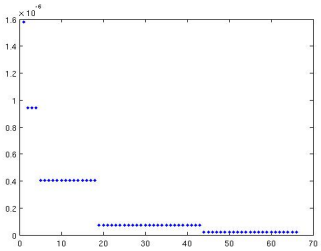
El que más lejos estuvo del mínimo fue el descenso aleatorio, quedando a 10^{-3} de distancia del 0. El que más cerca estuvo fue el recocido simulado, quedando a 10^{-14}. El algoritmo genético y el híbrido quedaron a 10^{-8}. Búsqueda local y búsqueda local iterada se comportaron de manera similar, quedando a 10^{-6} de distancia al 0 aproximadamente.



(a) Búsqueda local



(b) Descenso aleatorio



(c) Híbrido

Figura 1: Tres métodos con una función cuadrática

Es notable la dispersión de los puntos en el descenso aleatorio y cómo esto mejora cuando pasamos a búsqueda local, en el que el dominio de búsqueda va mejorando progresivamente. El gráfico 1c muestra cómo el valor de la función va mejorando generación a generación.

4.2. Función de Rosenbruck

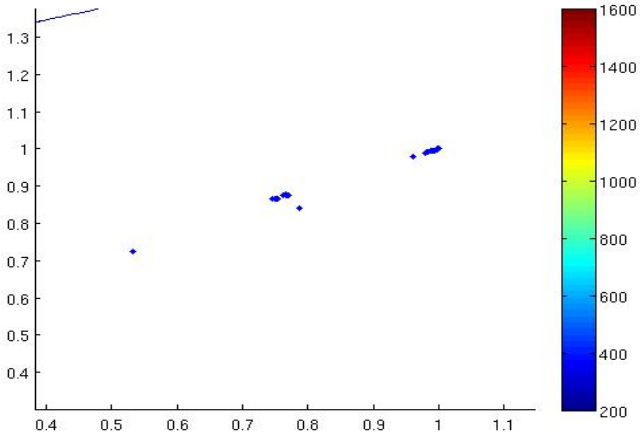
Para esta función utilizamos $[0, 2]^d$ como dominio y todos los parámetros por defecto a menos que indiquemos lo contrario.

Para la versión en dimensión 2 de esta función no tenemos mucho para decir, ya que se comportó de manera muy similar a la función anterior. Esto de todos modos ya es un poco sorpresivo, ya que esta función es bastante patológica. Algo a remarcar (aunque predecible) es que el método de descenso aleatorio es muy sensible a corridas, es decir que da resultados bastante distintos cada vez que lo corremos. En general la distancia al mínimo es siempre la misma (del orden de 10^{-1}) pero el valor mínimo que nos da va cambiando. Esto obviamente tiene que ver con que el proceso es aleatorio. Además de eso, parece ir mejorando inversamente proporcional al tiempo (es decir, aumentamos el tiempo al doble y la distancia al mínimo se reduce aproximadamente a la mitad).

Otro comentario es que, en una primera versión, implementábamos la versión sin elitismo para el algoritmo genético, y se comportaba de manera desastrosa, quedando a distancia 2 del mínimo constantemente, aunque variara todos los parámetros (la forma de hacer *crossover*, el tiempo, el tamaño de la población). Cuando incorporamos la supervivencia de los mejores cuatro padres por generación hubo una mejora importante, quedando a 10^{-13} de distancia del mínimo en 30 segundos.

Lo último a remarcar es que a partir de esta función fue que decidimos quedarnos con *line crossover* para el algoritmo genético, ya que los otros métodos hacían que quedara más lejos del mínimo.

Figura 2: Algoritmo genético con la función de Rosenbruck en dimensión 2.



Acá se ven con bastante claridad las distintas generaciones y cómo se van acercando al $(0;0)$ que es el máximo.

Para la dimensión 4 los resultados son muy distintos.

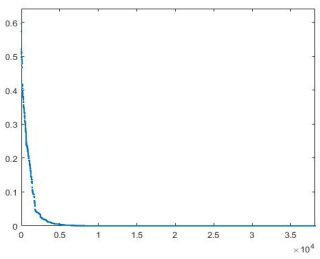
El descenso aleatorio arrojaba resultados a distancia 12 del mínimo y, al realizar más pruebas aumentando el tiempo de corrida, los resultados eran incluso peores.

La búsqueda local mejoró esto considerablemente, arrojando resultados a 10^{-1} de distancia del mínimo (con 30 segundos de corrida) y 10^{-2} de distancia (aumentando al doble el tiempo).

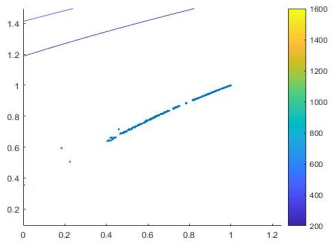
Obtuvimos prácticamente los mismos resultados con búsqueda local iterada, con la única diferencia de que la distancia al mínimo era del orden de 10^{-2} tanto con 30 como con 60 segundos.

El algoritmo híbrido quedó a 10^{-3} del mínimo y a 10^{-4} el algoritmo genético. Al aumentar en este último la población y el tiempo al doble, se redujo a 10^{-8} la distancia al mínimo.

La sorpresa fue el recocido simulado, que en 30 segundos llegó a un orden de 10^{-10} del mínimo.

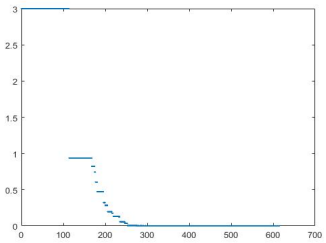
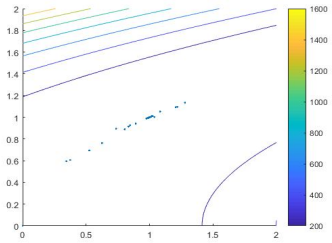


(a) Valor de la función en cada iteración



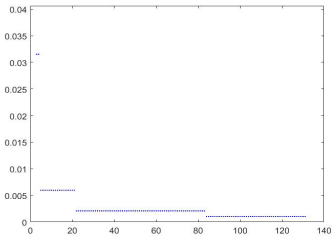
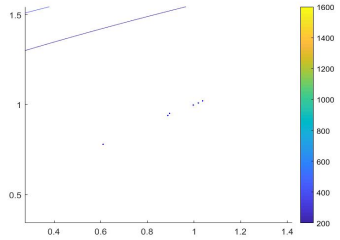
(b) Curvas de nivel

Figura 3: Algoritmo genético con función de Rosenbruck en dimensión 4

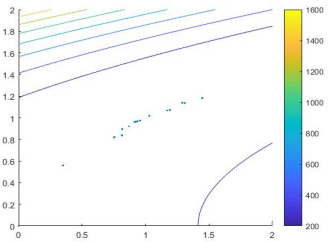
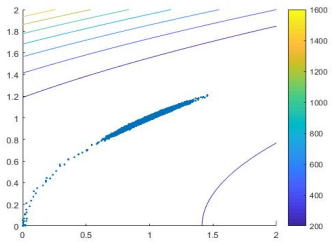
(a) Valor de la función en cada iteración (b) Curvas de nivel

Figura 4: Recocido simulado con función de Rosenbruck en dimensión 4

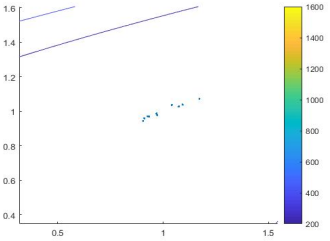



(a) Valor de la función en cada iteración (b) Curvas de nivel

Figura 5: Algoritmo híbrido con función de Rosenbruck en dimensión 4

(a) Descenso aleatorio (b) Búsqueda local

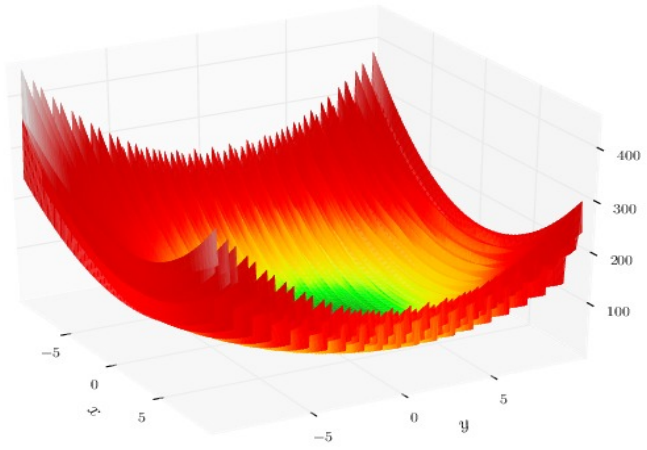


(c) Búsqueda local iterada

Figura 6: Descenso aleatorio y sus variantes con función de Rosenbruck en dimensión 4

4.3. Función de Lévi

Figura 7: Función de Lévi



Esta función tiene como fórmula

$$f(x,y) = \sin^2(3\pi x) + (x-1)^2 \cdot (1 + \sin^2(3\pi y)) + (1-y)^2 \cdot (1 + \sin^2(2\pi y))$$

y, como se ve en el gráfico 7, tiene muchos mínimos y máximos locales. El mínimo global se alcanza en $(1; 1)$.

Cuando *testeamos* descenso aleatorio, búsqueda local y búsqueda local iterada los resultados fueron desastrosos, quedando a distancias grandes del mínimo (entre 20 y 100). Algo sorprendente es que incluso funcionó mejor el descenso aleatorio, que quedó a distancia 1 del mínimo. Lo que también es raro es que el gradiente en los puntos que arrojan estos algoritmos está lejos de ser nulo.

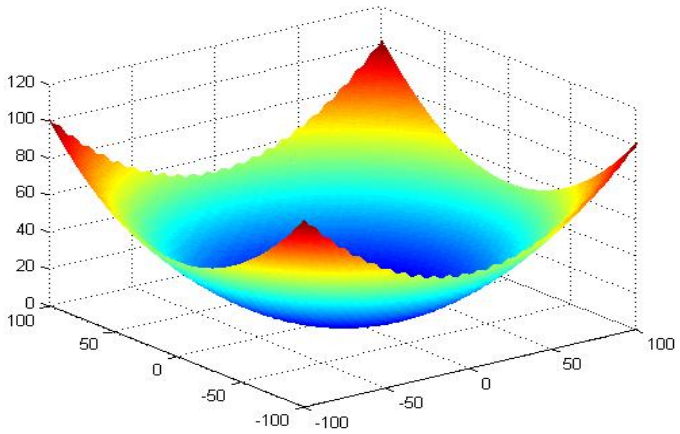
Una posible explicación a este hecho se deba a que Lévi es una función con numerosos extremos locales. Tiene sentido que cualquier proceso que tenga como motor fundamental una búsqueda local aleatoria termine en alguno de estos mínimos locales, en un tiempo dado. En el algoritmo de búsqueda local iterada el mayor tiempo de las iteraciones es consumido justamente por la parte de búsqueda local. Mientras que en descenso aleatorio, todo el tiempo se van generando nuevos candidatos de manera independiente de los anteriores, y uno se queda con el mejor conseguido.

Lo que cabe preguntarse, si son extremos locales de una función derivable, es por qué el gradiente es tan distinto de cero. Lo que se conjetura es que se debe a errores numéricos debido a que es una función que tiene "paredes muy empinadas", es decir que cerca de los extremos locales el gradiente es realmente muy distinto de cero. Si a esto le sumamos que la cuenta se aproxima de manera numérica, es esperable que no se cumpla que el gradiente sea igual a cero en uno de estos extremos locales.

Como prueba a favor de esto, se analizó la función Griewank (analizada también en el Trabajo Práctico N° 2), la cual tiene muchos extremos locales, pero es mucho más suave (como se puede ver abajo). En estos casos los gradientes de los valores obtenidos son mucho más cercanos a cero.

4.4. Función de Griewank

Figura 8: Función de Griewank



Esta función tiene como fórmula

$$f(x,y) = \frac{x^2+y^2}{200} - \cos(x) \cdot \cos\left(\frac{y}{\sqrt{2}}\right) + 1$$

y tiene muchos mínimos y máximos locales. El mínimo global se alcanza en $(0;0)$. A diferencia de lo que se ve en la función anterior, esta parcería ser más suave, en el sentido en que en los entornos de los extremos locales la función se mantiene pareja.

Con esta función se observa en los metodos de descenso aleatorio, búsqueda local y búsqueda local iterada un comportamiento similar al de la función anterior. Pero debido a la suavidad que tiene, si esos puntos alcanzados son algún mínimo local, deberían tener un gradiente cercano a cero. Y esto fue lo que efectivamente ocurrió. Por ejemplo, en la figura 9a podemos ver que a través del método de búsqueda local se alcanzó el punto $(9{,}3037; 47{,}8114)$, muy lejano al $(0,0)$, sin embargo, el gradiente evaluado en ese punto es $(0{,}0057; -0{,}0002)$. Si consideramos que el gradiente fue calculado numéricamente, y que el método de búsqueda local es heurístico, es razonable considerar que el método nos acercó a un mínimo local. Esto permite corroborar que, en casos donde las funciones tienen numerosos extremos locales, no es buena idea usar en métodos cuyo principal "motor" sea una búsqueda local. El caso totalmente opuesto fue el resultado obtenido con el algoritmo genético, con el cual se obtuvieron los mejores resultados, llegando a valores a distancia del orden de 10^-7 del $(0;0)$ como se puede ver en la figura 9b.

(a) Búsqueda Local (b) Algoritmo Genético

Figura 9: Pruebas realizadas con la función Griewank

5. Conclusiones y perspectivas

La principal conclusión que se obtuvo realizando el presente trabajo práctico es la utilidad de los métodos heurísticos para un montón de casos en donde se busca optimizar una función de la cual se conoce poca información. Como contraparte, el conocer poco de la función impide saber qué tan bueno es el valor obtenido. Aunque según el método utilizado es el mejor de todos los puntos analizados. También se mostraron muy útiles en los casos donde la función a analizar tiene numerosos extremos locales, donde aunque se conozca bien la expresión de la función es dificil de analizar, o donde incluso los métodos tradicionales (como por ejemplo método del gradiente) no funcionan bien. Sin embargo en estos casos hay que tener cuidado con los métodos que buscan localmente.

Como posibles líneas a profundizar podemos pensar en analizar con más detalle todas las variantes y opciones que ofrecen el recocido simulado y los algoritmos genéticos. Probando en el primer caso distintas formas de descenso de la temperatura por ejemplo, o en el segundo distintas formas de seleccionar a los padres de la siguiente generación.

Referencias

[1] Luke, Sean. 2015. Essentials of Metaheuristics International Series in Operations Research & Management Science. Third edition