Optimización por Cúmulo de Partículas

Estudiantes: Cázares Trejo Leonardo Damián / Rivera Gálvez Ernesto
Materia MATEMATICAS DISCRETAS

Intuición

La optimización por cúmulo de partículas (*Particle Swarm Optimization*, *PSO*) es un algoritmo inspirado por el agrupamiento natural de organismos individuales, por ejemplo, las bandadas de aves o los bancos de peces, ver figuras 1 y 2 respectivamente.

Figura 1: Bandada de aves (*Sturnus unicolor*).

Figura 2: Banco de peces (*Sardina pilchardus*).

En estos agrupamientos naturales de organismos, cada uno de los individuos sin ser supervisados ni controlados por un solo individuo trabajan juntos para alcanzar un objetivo en común. Este comportamiento grupal inspiro un algoritmo computacional llamado, optimización por cúmulo de partículas, con el objetivo de optimizar un determinado problema, para esto usaremos un grupo de soluciones candidatas representadas por partículas individuales, análogamente a como los grupos de animales de los que hablamos arriba trabajan juntos para lograr un objetivo.

Cada una de las partículas dentro del cumulo se moverán dentro de un espacio de búsqueda previamente definido, y sobre el que queremos optimizar un problema dado, buscando paso a paso la mejor solución posible. El movimiento del enjambre de individuos está controlado por reglas simples que involucran su posición y velocidad

durante cada iteración.

El *PSO* es un algoritmo iterativo, en cada iteración la posición de cada partícula se evalúa, según la función o problema de nuestro interés, y se guarda su mejor ubicación hasta el momento, así como la mejor ubicación dentro de todo el grupo de partículas, actualizando si es necesario. Después la velocidad de cada partícula se actualiza acorde a la siguiente información:

1. La velocidad actual de la partícula y la dirección en la que se mueve, esto se conoce como inercia.
2. La mejor posición (local) de la partícula encontrada hasta el momento, representando la fuerza local.
3. La mejor posición del grupo (global) encontrada hasta el momento, representando la fuerza social del enjambre.

El paso siguiente es una actualización de la posición de cada una de las partículas, basada en la última velocidad calculada.

Este proceso iterativo continua hasta alguna condición de paro, por ejemplo, un límite del número de iteraciones. En este punto del algoritmo la mejor posición grupal se toma como la solución del problema.

Modelo matemático

Una vez que comprendimos los pasos detrás del *PSO*, plantemos el modelo matemáticamente y el pseudocódigo del algoritmo. Son tres las principales ecuaciones con las que podemos describir el modelo *PSO*, la primera de ellas se refiere el proceso de inicialización de las posiciones de las partículas del enjambre,

$$x_k^{i,t} = l_k + rand \cdot (u_k - l_k),\ x_k^{i,t} \in x^t, \tag{1}$$

donde $x_k^{i,t}$ es la i-ésima ($i = 1, 2, ..., Part\_N$), partícula de la población al tiempo t, además que el número de dimensiones en las que consideramos viven las partículas viene dado por k. Por otro lado, para cada una de las dimensiones en el espacio de búsqueda de la solución, el límite inferior y superior serán respectivamente l_k y u_k, *rand* es número aleatorio bajo la distribución uniforme de 0 a 1. Las posiciones iniciales de las partículas se eligen de forma aleatoria a lo largo del espacio de búsqueda, con el fin de que las partículas inicien la búsqueda desde todos los sitios permitidos.

Inicialmente las partículas inician con velocidad cero, pero para la segunda iteración deben de calcularse las nuevas velocidades bajo los parámetros que enlistamos antes (inercia, fuerza local y fuerza social),

$$v^{t+1} = v^t + rand_1 \times (P - x^t) + rand_2 \times (G - x^t), \tag{2}$$

donde v^{t+1} es el valor de la velocidad que se calcula en la iteración t, x^t es el vector que contiene las posiciones de las partículas, P tiene las mejores posiciones actuales asociadas a la vecindad de cada partícula, G es la mejor posición global. Además, $rand_1$ y $rand_2$ son valores aleatorios distintos de distribuciones aleatorias uniformes entre 0 y 1. A diferencia de la primera ecuación donde mostramos entrada a entrada cada partícula, el cálculo de la posición inicial, las cantidades de la segunda expresión son matrices, y las operaciones de suma y producto mostradas son de elemento a elemento.

Finalmente, luego de haber calculado la velocidad, las partículas avanzaran según la última velocidad calculada,

$$x^{t+1} = x^t + v^{t+1}, \tag{3}$$

donde x^{t+1} es el vector donde se guardan las nuevas posiciones para la iteración t. Donde v^{t+1} es la velocidad calculada en la ecuación (2). Un esquema donde mostramos el comportamiento de una partícula bajo las tres ecuaciones descritas antes la encontramos en la figura 3.

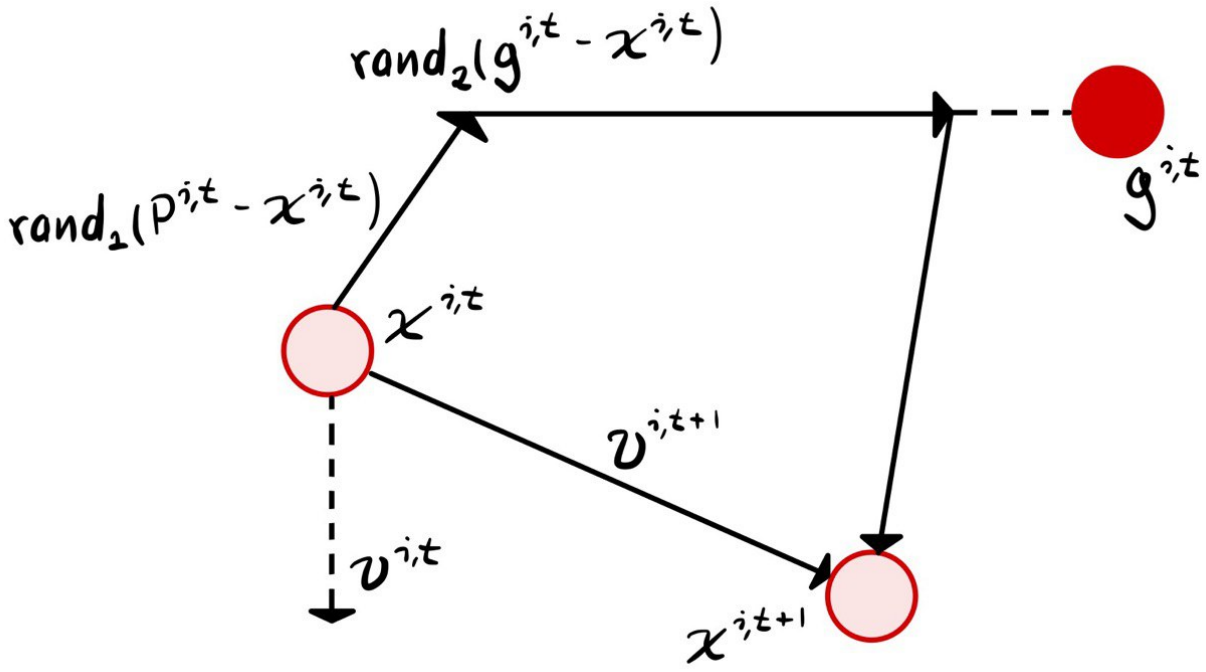

Figura 3: Comportamiento de una partícula en el *PSO*.

Pseudocodigo

Ya conocida la intuición dentras del método y las principales ecuaciones que gobiernan el *PSO*, veamos el pseudocodigo:

```
para cada particula i=1,2,...,Part_N
    inicializamos la posicion segun x_k^{i,t}=l_k+rand*(u_k-l_k)
    inicializamos la mejor posicion local P_i = x_k^{i,t}
    si f(P_i) < f(g) entonces
        actualizamos la mejor posicion global G = P_i
    inicializamos las velocidades de las particulas v_i = 0
mientras no se exceda el maximo de iteraciones
    actualizamos las velocidades v^{t+1}=v^t+rand_1*(P-X^t)+rand_2*(G-X^t)
    actualizamos las posiciones x^{t+1} = x^{t} + v^{t+1}
    para cada particula i=1,2,...,Part_N
        si f(x_k^{i,t})<f(P_i) entonces
            actualizamos la mejor posici n local P_i = x_k^{i,t}
            si f(P_i)<f(g) entonces
                actualizamos la mejor posici n local g = P_i
```

Pruebas

Ahora que también conocemos el modelo matemático y el pseudocódigo detras del *PSO*, hagamos la implementación en *Julia*, para dos funciones,

$$f_1(x) = 3x^4 - 8x^3 + 12x^2 - 48x + 25, \tag{4}$$

con mínimo en $x = 2$ y $f(2) = -39$, como vemos en la figura 4.

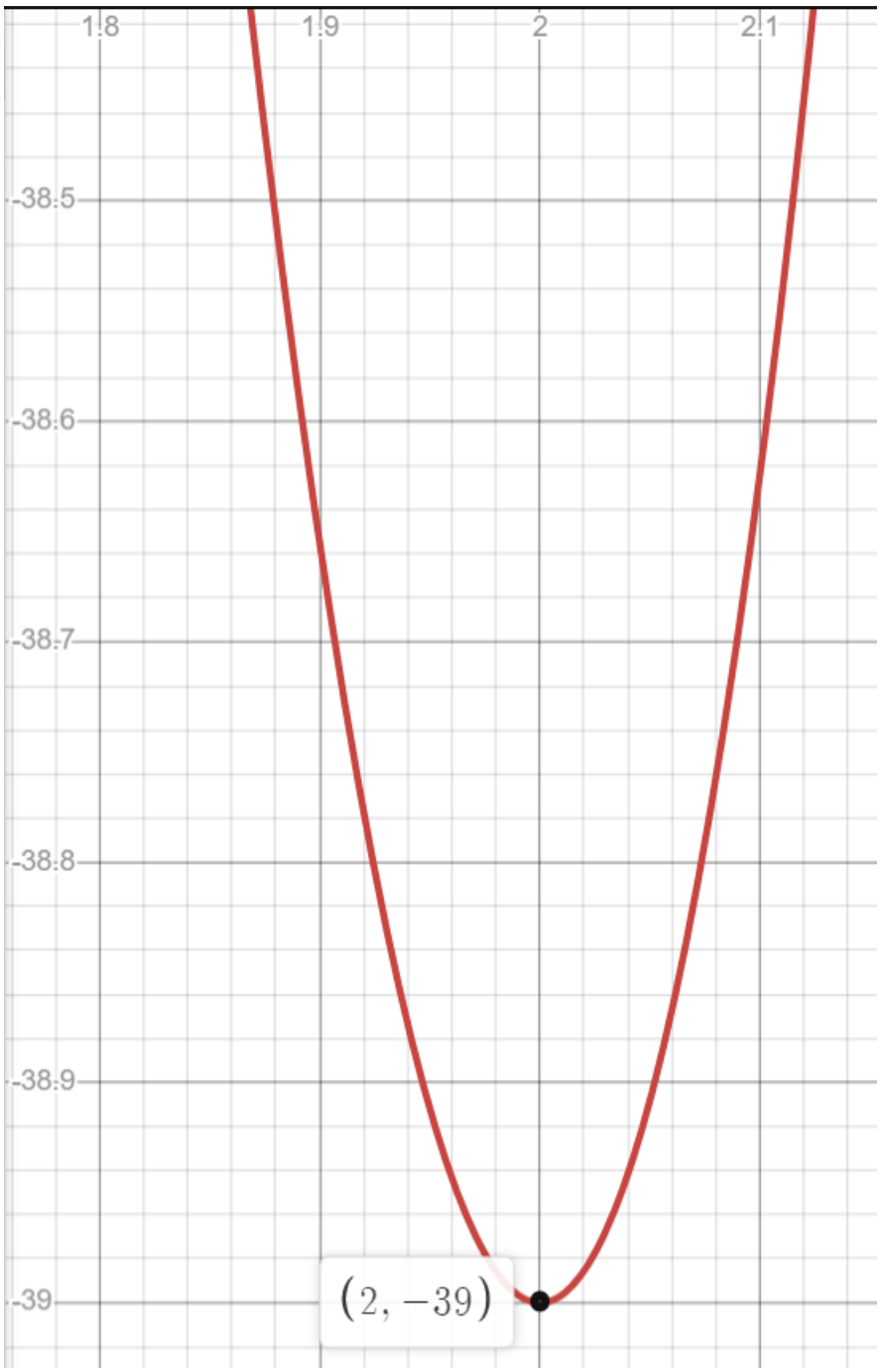

Figura 4: Primera función, $f_1(x) = 3x^4 - 8x^3 + 12x^2 - 48x + 25$, con su minimo en $x = 2$.

Y la función,

$$f_2(x) = 3cos(x) + sin^2(x), \tag{5}$$

con mínimo en $x = \pi$ y $f(\pi) = -3$, como en la figura 5.

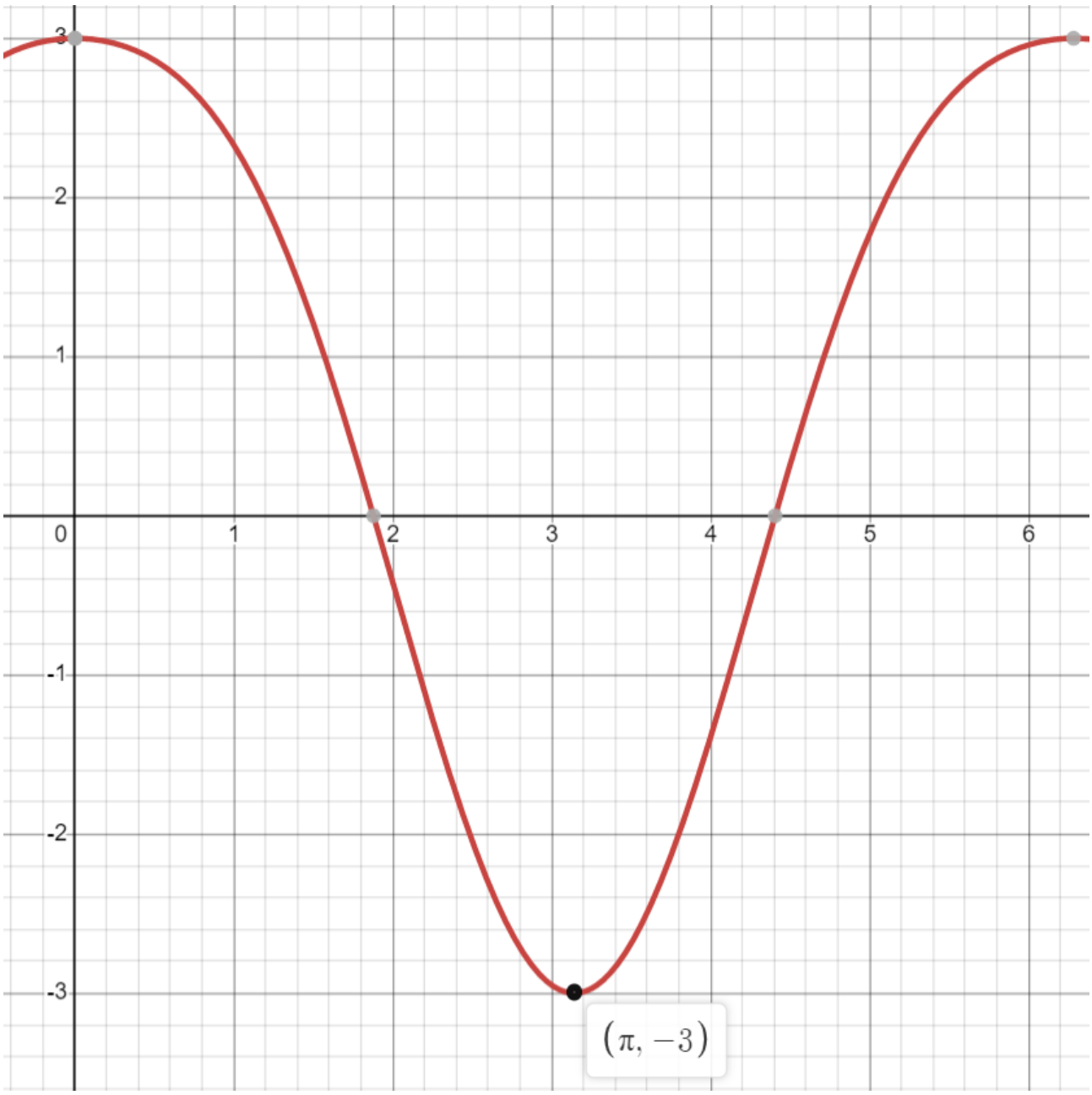

Figura 5: Segunda función, $f_2(x) = 3cos(x) + sin^2(x)$, con su minimo en $x = \pi$.

Para la f_1, podemos ver la implementación secuencial en el código siguiente:

```
using Distributions
using Printf
using Statistics

# Numero de dimensiones en que se mueve nuestra particula
d = 1

# Limite inferior del dominio en que se mueve cada particula
l = [0]

# Limite superior del dominio en que se mueve cada particula
u = [3]

# Numero maximo de iteraciones (condicion de paro)
Max_iter = 500
```

```
# Numero de particulas
Part_N = 100

# Inicializamos la posicion inicial de cada particula, ecuacion (1)
x = l[1] .+ rand(Uniform(0,1), Part_N,d) .* (u[1] - l[1])

# Evaluamos la funcion objetivo en cada particula
obj_func = zeros(1,Part_N)
for i = 1:Part_N
    obj_func[i] = funcion1d_1(x[i])
end

# Mejor valor global (minimo), valor e indice
glob_opt = minimum(obj_func)
ind = argmin(obj_func)[2]

# Vector de valores optimos
G_opt = x[ind] .* ones(Part_N,d)

# Mejor posicion global
Mejor_pos = [x[ind]]

# Mejor local de cada particula
Loc_opt = x

# Velocidades iniciales
v = zeros(Part_N,d)

# Contador de iteraciones
t = 1

# Arreglos para evaluaciones de las funciones y su evolucion a lo largo de las
# iteraciones
Nva_obj_func = zeros(1,Part_N)
Evol_func_obj = zeros(1,Max_iter)

# Mientras no se alcance la condicion de paro
while t < Max_iter

# Calcula la nueva velocidad, ecuacion (2)
global v = v .+ rand(Uniform(0,1), Part_N,d) .* (Loc_opt - x) + rand(Part_N,d)
.* (G_opt - x);

# Obtenemos nueva posicion, ecuacion (3)
global x = x .+ v;

# Para cada particula se verifica que no salgan de los limites l y u
for i=1:Part_N
if x[i] > u[1]
global x[i] = u[1];
elseif x[i] < l[1]
global x[i] = l[1];
end

# Se vuelven a evaluar las nuevas posiciones en la funcion objetivo
global Nva_obj_func[i] = funcion1d_1(x[i]);
```

```
# Se comprueba si se actualizan los optimos locales
if Nva_obj_func[i] < obj_func[i]

# Actualiza optimo local
global Loc_opt[i] = x[i]

# Actualiza funcion objetivo
global obj_func[i] = Nva_obj_func[i]
end
end

# Obtiene el mejor valor global
global Nvo_glob_opt = minimum(obj_func)
global ind = argmin(obj_func)[2]

# Se verifica si se actualiza el optimo global
if Nvo_glob_opt < glob_opt
    global glob_opt = Nvo_glob_opt
    # Se actulizan los valores optimos
    global G_opt[:] = x[ind] .* ones(Part_N,d)
    global Mejor_pos = [x[ind]];
end

# Almacena valores de funcion objetivo
global Evol_func_obj[t] = glob_opt;

# Incrementa la iteracion
global t = t + 1;
end
```

La función f_1, también se realizo en *Julia*:

```
function funcion1d_1(x)
    """
    Esta funcion calcula el resultado de
    evaluar x -> 3*x^4 - 8*x^3 + 12*x^2 -
    48*x + 25

    El minimo de esta funcion en el
    intervalo de [0,3] es -39 y se encuentra
    en x = 2

    Recibe:
    ----------
    x = (numeric) valor numerico

    Devuelve
    ----------
    (numeric) el resultado de evaluar
    la funcion
    """
    return 3*x^4 - 8*x^3 + 12*x^2 - 48*x + 25
end
```

Para ver el resultado de nuestro primer conjunto de pruebas ejecutemos el programa $funcion1d\_1.jl$ y luego $pso1d.jl$, los cuales contienen los dos programas anteriores, los resultados los podemos ver en la figura 6. Donde el valor donde encontramos el mínimo es $x_{min} \approx Mejor\_pos$, y se aproxima con error porcentual de 0.002 % al valor real.

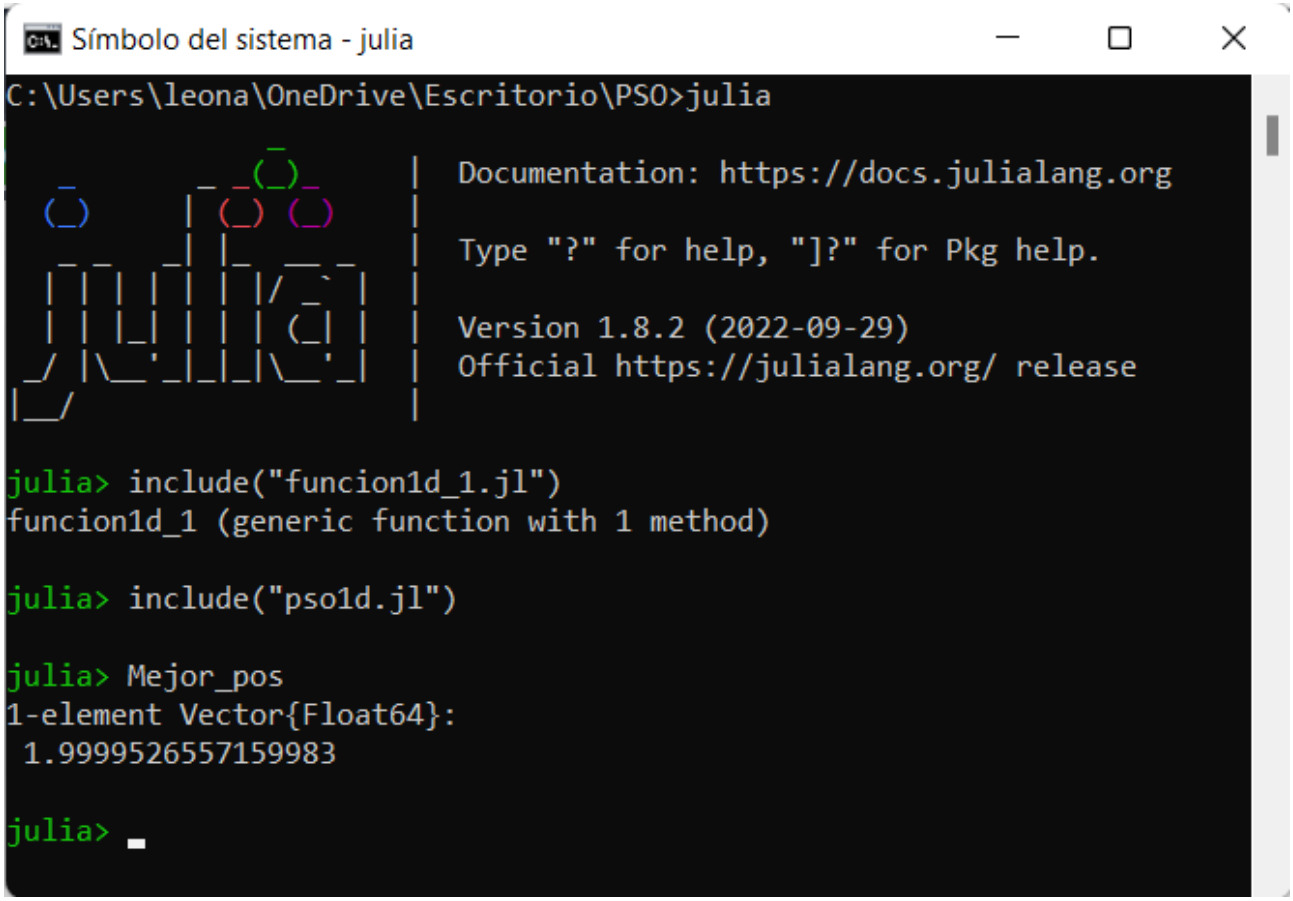

Figura 6: Valor minimo de f_1 usando *funcion1d_1.jl* y *pso1d.jl*.

Para un segundo conjunto de pruebas, modificamos el *PSO* como:

```
using Distributions
using Printf
using Statistics

#function funcion1d_2(x)
#    return 3*cos(x) + (sin(x))^2
#end

# Numero de dimensiones en que se mueve nuestra part cula
d = 1

# Limite inferior del dominio en que se mueve cada particula
l = [0]

# Limite superior del dominio en que se mueve cada particula
u = [6]

# Numero maximo de iteraciones (condicion de paro)
Max_iter = 500

# Numero de particulas
Part_N = 150

# Inicializamos la posicion inicial de cada particula, ecuacion (1)
x = l[1] .+ rand(Uniform(0,1), Part_N,d) .* (u[1] - l[1])

# Evaluamos la funcion objetivo en cada particula
```

```
obj_func = zeros(1,Part_N)
for i = 1:Part_N
    obj_func[i] = funcion1d_2(x[i])
end

# Mejor valor global (minimo), valor e indice
glob_opt = minimum(obj_func)
ind = argmin(obj_func)[2]

# Vector de valores optimos
G_opt = x[ind] .* ones(Part_N,d)

# Mejor posicion global
Mejor_pos = [x[ind]]

# Mejor local de cada particula
Loc_opt = x

# Velocidades iniciales
v = zeros(Part_N,d)

# Contador de iteraciones
t = 1

# Arreglos para evaluaciones de las funciones y su evolucion a lo largo de las
# iteraciones
Nva_obj_func = zeros(1,Part_N)
Evol_func_obj = zeros(1,Max_iter)

# Mientras no se alcance la condicion de paro
while t < Max_iter

# Calcula la nueva velocidad, ecuacion (2)
global v = v .+ rand(Uniform(0,1), Part_N,d) .* (Loc_opt - x) + rand(Part_N,d)
.* (G_opt - x);

# Obtenemos nueva posicion, ecuacion (3)
global x = x .+ v

# Para cada particula se verifica que no salgan de los limites l y u
for i=1:Part_N
if x[i] > u[1]
global x[i] = u[1];
elseif x[i] < l[1]
global x[i] = l[1];
end

# Se vuelven a evaluar las nuevas posiciones en la funcion objetivo
global Nva_obj_func[i] = funcion1d_2(x[i])

# Se comprueba si se actualizan los optimos locales
if Nva_obj_func[i] < obj_func[i]

# Actualiza optimo local
global Loc_opt[i] = x[i]

# Actualiza funcion objetivo
global obj_func[i] = Nva_obj_func[i]
```

```
end
end

# Obtiene el mejor valor global
global Nvo_glob_opt = minimum(obj_func)
global ind = argmin(obj_func)[2]

# Se verifica si se actualiza el optimo global
if Nvo_glob_opt < glob_opt

    # Se actulizan los valores optimos
    global glob_opt = Nvo_glob_opt;
    global G_opt[:] = x[ind] .* ones(Part_N,d)
    global Mejor_pos = [x[ind]]
end

# Almacena valores de funcion objetivo
global Evol_func_obj[t] = glob_opt

# Incrementa la iteracion
global t = t + 1;
end

function funcion1d_2(x)
    """
    Esta funci n calcula el resultado de
    evaluar x -> 3*cos(x) + (sin(x))^2

    El m nimo de esta funci n en el
    intervalo de [0,6] es -2 y se encuentra
    en x = pi

    Recibe:
    ----------
    x = (numeric) valor numerico

    Devuelve
    ----------
    (numeric) el resultado de evaluar
    la funcion
    """
    return 3*cos(x) + (sin(x))^2
end
```

Donde cambiamos la función objetivo con la que evaluamos (f_2), los extremos del dominio de busqueda y el número de particulas. El resultamos lo vemos en la figura 7, donde ejecutamos los programas $funcion1d\_2.jl$ y $pso1d\_2.jl$ que tienen los dos códigos mostrados arriba. Se obtuvo una buena aproximación de π, usando que $x_{min} \approx Mejor\_pos$, y se aproxima con error porcentual de 0.00007 % al valor real.

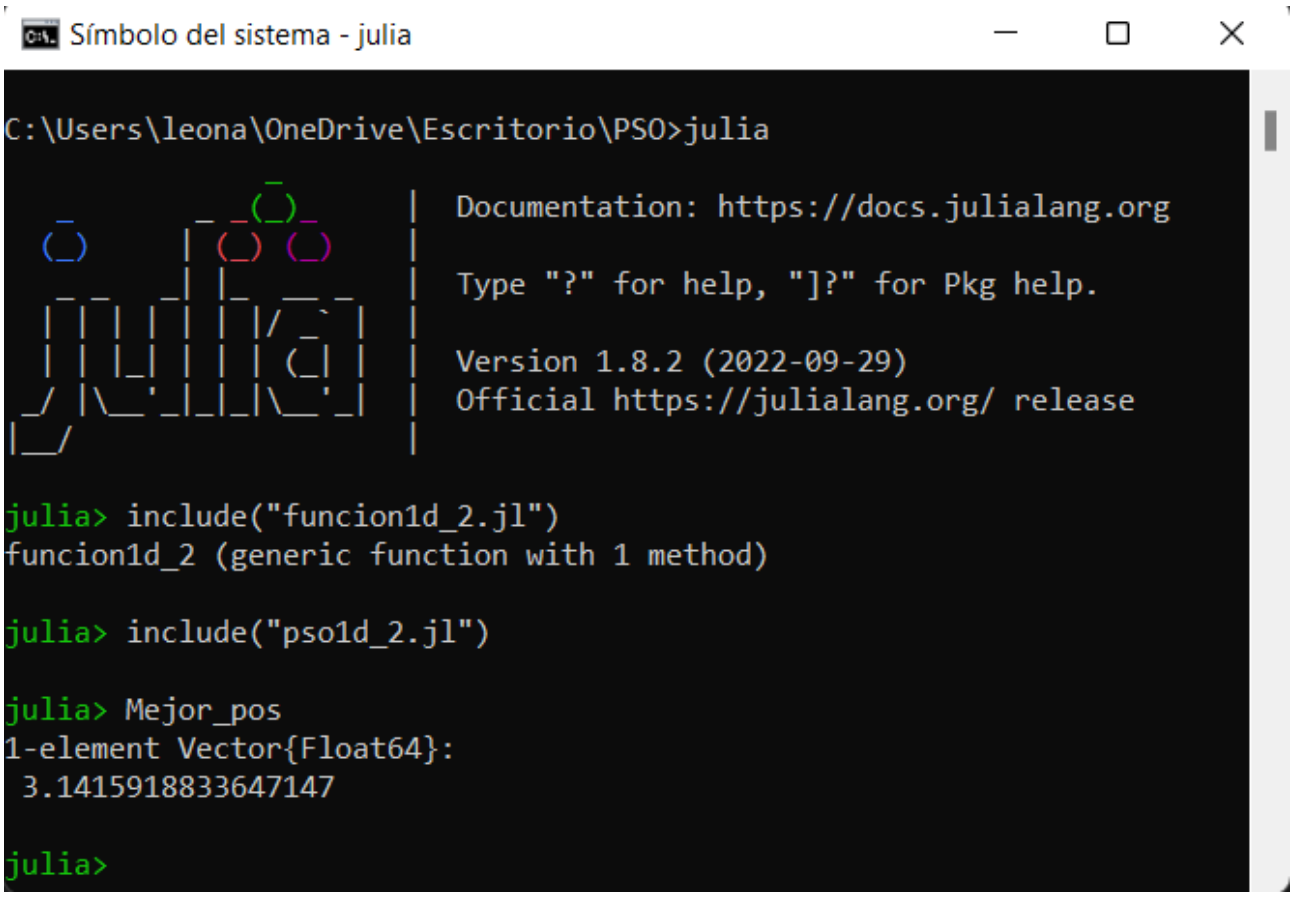

Figura 7: Valor minimo de f_2 usando $funcion1d\_2.jl$ y $pso1d\_2.jl$.

Referencias

1. Wirsansky, E. (2020). *Hands-on genetic algorithms with python: Applying genetic algorithms to solve real-world deep learning and Artificial Intelligence Problems*. Packt Publishing Ltd.
2. Gridin, I. (2021). *Learning genetic algorithms with python*. BPB Publications.
3. Cuevas, O. E. V.(2016). Optimización de Algoritmos programados con MATLAB (1.a ed.). Marcombo.
4. Maurice Clerc. Standard Particle Swarm Optimisation. date. ffhal-00764996ff