División de Inenierías Campus Irapuato Salamanca
Universidad de Guanajuato Clase: Inteligencia artificial

Profesor: Dr. Carlos Hugo García Capulín
Estudiante: Brandon Marquez Salazar

Tarea 1

Uso de PSO Velocity Clamping

Entrega 25 de Abril del 2023

Introducción

La La optimización de enjambre de partículas (PSO) se considera importante en la inteligencia basada en enjambre. El PSO está relacionado con el estudio de los enjambres; donde se trata de una simulación de bandadas de pájaros. Se puede utilizar para resolver una amplia variedad de problemas de optimización. Es un método heurístico el cual, esencialmente, busca la solución a un modelo, basado en elementos específicos y un conjunto de estructuras, que emulan el comportamiento de un conjunto de individuos quienes buscan una posición espefífica, que se evalúa según el modelo planteado.

Planteamiento del problema

El problema que se nos plantea, es la búsqueda del valor más alto, utilizando el método de PSO con truncamiento de velocidad, d'a siguiente ecuación:

$$y(x,y) = \left(\frac{-((x+1)^2+(y-3{,}14)^2)}{5^2}\right) + \cos(2x) + \sin(2y)$$

Descripción del programa

El programa consta de una estructura PARTICULA y una estructura ENJAMBRE. La estructura partícula tiene los valores importantes de posición y evaluación respecto al modelo matemático. También, se tienen funciones que permiten la operación d'os elementos del enjambre, su inicialización, actualización e interacción. El código está dividido en una cabecera que define las operaciones esenciales del algoritmo, las estructuras y el prototipo de las dos funciones q'el usuario puede definir: función objetivo (el cual albergará al modelo matemático), función proceso (que albergará'l proceso que llevará, desde la creación del

enjambre, las evaluaciones, entre otros elementos del procesamiento, hasta su limpieza).

Código utilizado

Cabecera

```
#ifndef __pso__header__
#define __pso__header__

// Definición de la estructura Patícula
// Esta partícula representa a un individuo
// El individuo buscará tener la mejor posición
// Habrán dos criterios: En el primero, ve la mejor
// posición que ha tenido durante su existencia; en
// el segundo, ve qué partícula tiene la mejor posición
// en ese momento.
// Recalcula su valor de paso (velocidad) y, a partir de
// ahí, suma el paso a su posición actual, obteniendo el
// nuevo valor de posición.
// La partícula requiere saber en cuántas dimensiones
// estará moviéndose. Dichas dimensiones definirán al vector
// posición y al vector velocidad.
typedef struct {
  float *Xi;   //Posicion
  float *Vi;   //Velocidad
  float *Pi;   //Mejor Posicion Historica
  float  Xfit; //Valor de Fitnes para la posicion actual
  float  Pfit; //Valor de Fitnes para la Mejor Posicion Historica
}PARTICULA;

// Definición de la estructura Enjambre
// El enjambre es un conjunto de partículas
// Este conjunto actuará para encontrar soluciones
// Cada solución es repensada según los valores históricos
// y valores presentes.
typedef struct{
  PARTICULA *Part;                          // Partículas
  unsigned int CantidadDeParticulas;        // Número de partículas
  unsigned int CantidadDeParametros;        // Número de parámetros del
      problema
  unsigned int MejorParticulaDelGrupo;      // ID de la mejor partícula del
       grupo
  unsigned int MaximoDeIteraciones;         // Número máximo d'iteraciones
      a realizar
  float C1;                                 // Valor de peso C1
  float C2;                                 // Valor de peso C2
```

```
  const float *LimitesSuperiores;       // Limites Superiores
  const float *LimitesInferiores;       // Limites Inferiores
  float X;                              // Factor de constricción (
      convergencia)
}ENJAMBRE;


// Operadores del enjambre (métodos)

/* Creador de enjambres:
 * Recibe el número de partículas y el número de parámetros
 * (variables del problema)*/
ENJAMBRE* CrearEnjambre(
    //ENJAMBRE     *__Enjambre__,
    unsigned int  __CantidadDeParticulas__,
    unsigned int  __CantidadDeParametros__
  );
/* Inicializador de enajmbres:
 * Defie los valoes predeterminados (de inicio), de los individuos.
 * Recibe el enjambre, la posición inicial, las variables del problema,
 * y los límites*/
void InicializarEnjambre(
    ENJAMBRE    *__Enjambre__,
    float        __FactorConstriccion__,
    float        __ValorDePeso_C1__,
    float        __ValorDePeso_C2__,
    unsigned int __MaximoDeIteraciones__,
    const float *__LimitesInferiores__,
    const float *__LimitesSuperiores__
  );
/* Una vez terminado el programa, ésta función liberará la memoria
 * que se reservó durante la creación del enjambre. Como argumento,
 * recibe al apuntador del enjambre.*/
void EliminarEnjambre(
    ENJAMBRE *__Enjambre__
  );
/* Nos permite visualizar los parámetros de la partícula.*/
void ImprimeParticulaID(
    ENJAMBRE     *__Enjambre__,
    unsigned int  __ID_Particula__
  );
/*Imprime la particula sin enjambre*/
void ImprimeParticula(
  const PARTICULA    *__Particula__,
  const unsigned int  __CantidadDeParametros__
  );
/* Permite visualizar los parámetros del enjambre, y las partículas
 * que le componen.*/
void ImprimeEnjambre(
    ENJAMBRE *__Enjambre__
```

```
  );
/* Permite valorar al enjambre, según los criterios del PSO
 * y de la función objetivo*/
void EvaluarEnjambre(
    ENJAMBRE    *__Enjambre__,
    const float *__ParametrosDeOperacion__
  );
/* Similar a EvaluarEnjambre, con la particularidad de que Inicializa
 * los valores de Mejor Posicion Historica, de las particulas*/
void EvaluacionInicialEnjambre(
    ENJAMBRE    *__Enjambre__,
    const float *__ParametrosDeOperacion__
  );
/* Renueva la valocidad basado en los vectores de
 * Posición Actual,
 * Mejor Posicion Historica y
 * Mejor Posicion Global Actual*/
void ActualizarVelocidad(
    ENJAMBRE *__Enjambre__
  );
void ActualizarVelocidadInerciaW(
    ENJAMBRE *__Enjambre__
  );
/* Suma los valores de velocidad a la posición actual de
 * cada partícula.*/
void ActualizarPosicion(
    ENJAMBRE *__Enjambre__
  );
/* Valora, en cada partícula, si el valor actual es mejor que'l mejor
 * valor histórico; si los valores actuales son mejores, actualiza
 * los parametros. */
void ActualizarMejoresPosiciones(
    ENJAMBRE *__Enjambre__
  );

/* La función a evaluar, regresa el valor de fitness (precisión)
 * Requiere ser definida para l'evaluación d'as partículas
 * Valores De Parametros .... (arreglo float)
 * Cantidad De Parametros ... (int)
 * Parametros De Operacion .. (arreglo float)
*/
float FuncionObjetivo(
    float          *__ValoresDeParametros__,
    unsigned int    __CantidadDeParametros__,
    const float    *__ParametrosDeOperacion__
  );

/* Funcion que se puede definir para realizar el procesamiento pso,
 * puede ser ignorado o definido y consta d'os sig. elementos, en
 * el orden en que se presentan a continuación:
```

```
 * Numero De Particulas ..... (float)
 * Dimension ..... (float)
 * Límites Superiores ..... (arreglo float)
 * Límite Inferiores ..... (arreglo float)
 * Numero Máximo De Iteraciones ..... (int)
 * Factor De Constriccion O De Inercia .. (float)
 * Valor Peso C1: Mejor Personal ..... (float)
 * Valor Peso C2: Mejor Global ..... (float)
 * Parametros De Operacion ..... (arreglo float) */
PARTICULA ProcesoPSO(
    const float         __NumeroDeParticulas__,
    const float         __Dimension__,
    const float        *__LimiteSuperior__,
    const float        *__LimiteInferior__,
    const unsigned int  __NumeroMaximoDeIteraciones__,
    const float         __Factor_Constriccion_Inercia__,
    const float         __ValorPesoPersonalC1__,
    const float         __ValorPesoGlobalC2__,
    const float        *__ParametrosDeOperacion__
  );

#endif
```

Definiciones

```
#include "pso.h"
#include <stdio.h>
#include <stdlib.h>

// Definición d'as funciones

ENJAMBRE* CrearEnjambre(
    //ENJAMBRE* __Enjambre__,
    unsigned int __CantidadDeParticulas__,
    unsigned int __CantidadDeParametros__
  ){
  ENJAMBRE *ptr=NULL;
  //Reservar la memoria para la estructura del enjambre
  ptr=(ENJAMBRE *)malloc(sizeof(ENJAMBRE));
  if(ptr==NULL){
    printf("Error al reservar la memoria para la estructura ENJAMBRE.")
        ;
    exit(0);
  }
  ptr->CantidadDeParticulas=__CantidadDeParticulas__;
  ptr->CantidadDeParametros=__CantidadDeParametros__;

  //Reservar la memoria para N particulas de M parametros
  ptr->Part=NULL;
  ptr->Part=(PARTICULA *)malloc(__CantidadDeParticulas__*sizeof(
```

```
    PARTICULA));
  if(ptr->Part==NULL){
    printf("Error al reservar la memoria para las Particulas.");
    exit(0);
  }
  //Reservar memoria para los 3 vectores de cada Particula
  for(unsigned int i=0; i<__CantidadDeParticulas__; ++i){
    ptr->Part[i].Xi=(float *)malloc(__CantidadDeParametros__*sizeof(
        float));
    ptr->Part[i].Vi=(float *)malloc(__CantidadDeParametros__*sizeof(
        float));
    ptr->Part[i].Pi=(float *)malloc(__CantidadDeParametros__*sizeof(
        float));
  }
  return ptr;
}


void InicializarEnjambre(
    ENJAMBRE     *__Enjambre__,
    float         __FactorConstriccion__,
    float         __ValorDePeso_C1__,
    float         __ValorDePeso_C2__,
    unsigned int __MaximoDeIteraciones__,
    const float *__LimitesInferiores__,
    const float *__LimitesSuperiores__
){
  if(__Enjambre__){
  float aux,rango;
  __Enjambre__->X                      = __FactorConstriccion__;
  __Enjambre__->C1                     = __ValorDePeso_C1__;
  __Enjambre__->C2                     = __ValorDePeso_C2__;
  __Enjambre__->MaximoDeIteraciones    = __MaximoDeIteraciones__;
  __Enjambre__->MejorParticulaDelGrupo = 0;
  __Enjambre__->LimitesInferiores      = __LimitesInferiores__;
  __Enjambre__->LimitesSuperiores      = __LimitesSuperiores__;
  //Inicializar cada vector de cada particula
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; ++i) //
      Para cada particula i
    for(unsigned int j=0; j<__Enjambre__->CantidadDeParametros; ++j) //
        Para cada parametro j de cada vector de la particula i
    { rango=__Enjambre__->LimitesSuperiores[j]-__Enjambre__->
        LimitesInferiores[j];
      aux= ((float)rand()/(float)RAND_MAX) * rango + __Enjambre__->
          LimitesInferiores[j];
      __Enjambre__->Part[i].Xi[j]=aux;
      __Enjambre__->Part[i].Vi[j]=0;
      __Enjambre__->Part[i].Pi[j]=aux;
    }
  }
```

```
}


void EliminarEnjambre(ENJAMBRE* __Enjambre__)
{ //Liberar la memoria para de los 3 vectores de cada Particula
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; ++i)
     { free(__Enjambre__->Part[i].Xi);
       free(__Enjambre__->Part[i].Vi);
       free(__Enjambre__->Part[i].Pi);
     }
  //Liberar la memoria de las estructuras particula
  free(__Enjambre__->Part);
  //Liberar la memoria de la estrcutura del enajmbre
  free(__Enjambre__);
}


void ImprimeParticulaID(ENJAMBRE *__Enjambre__, unsigned int
    __ID_Particula__){
  printf("\nP%d,Xi:␣",__ID_Particula__);
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParametros; i++)
    printf("%f,␣",__Enjambre__->Part[__ID_Particula__].Xi[i]);
  printf("\nP%d,Vi:␣",__ID_Particula__);
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParametros; i++)
    printf("%f,␣",__Enjambre__->Part[__ID_Particula__].Vi[i]);
  printf("\nP%d,Pi:␣",__ID_Particula__);
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParametros; i++)
    printf("%f,␣",__Enjambre__->Part[__ID_Particula__].Pi[i]);
  printf("\nP%d,Xfit=%f",__ID_Particula__,__Enjambre__->Part[
     __ID_Particula__].Xfit);
  printf("\nP%d,Pfit=%f",__ID_Particula__,__Enjambre__->Part[
     __ID_Particula__].Pfit);
}

void ImprimeParticula(
  const PARTICULA     *__Particula__,
  const unsigned int  __CantidadDeParametros__
){
  printf("\nParticula:");
  for(unsigned int i=0; i<__CantidadDeParametros__; i++)
    printf("%f,␣",__Particula__->Xi[i]);
  printf("\nParticula␣Vi:␣");
  for(unsigned int i=0; i<__CantidadDeParametros__; i++)
    printf("%f,␣",__Particula__->Vi[i]);
  printf("\nParticula,Pi:␣");
  for(unsigned int i=0; i<__CantidadDeParametros__; i++)
    printf("%f,␣",__Particula__->Pi[i]);
  printf("\nParticula,Xfit=%f",__Particula__->Xfit);
  printf("\nParticula,Pfit=%f",__Particula__->Pfit);
}
```

```
void ImprimeEnjambre(ENJAMBRE *__Enjambre__)
{ for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; ++i) //
    Para cada particula i
    ImprimeParticulaID(__Enjambre__,i);
}


void EvaluarEnjambre(ENJAMBRE *__Enjambre__,const float*
    __ParametrosDeOperacion__){
  float BestFit;
  // Calcular el valor de Fitness de cada particula
  BestFit = FuncionObjetivo(
      __Enjambre__->Part[0].Xi,
      __Enjambre__->CantidadDeParametros,
      __ParametrosDeOperacion__
    );
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++){
    __Enjambre__->Part[i].Xfit = FuncionObjetivo(
        __Enjambre__->Part[i].Xi,
        __Enjambre__->CantidadDeParametros,
        __ParametrosDeOperacion__
      );
    // Almacena el indice de la mejor particula de todo en enjambre
    if(__Enjambre__->Part[i].Xfit>BestFit){
      BestFit = __Enjambre__->Part[i].Xfit;
      __Enjambre__->MejorParticulaDelGrupo =i;
    }
  }
}

void EvaluacionInicialEnjambre(ENJAMBRE *__Enjambre__,const float*
    __ParametrosDeOperacion__){
  if(__Enjambre__){
  float aux,BestFit;
  //Calcular el valor de fitness de cada Particula
  BestFit=FuncionObjetivo(
      __Enjambre__->Part[0].Xi,
      __Enjambre__->CantidadDeParametros,
      __ParametrosDeOperacion__
    );
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++){
    aux=FuncionObjetivo(
        __Enjambre__->Part[i].Xi,
        __Enjambre__->CantidadDeParametros,
        __ParametrosDeOperacion__
      );
    __Enjambre__->Part[i].Xfit=aux;
    __Enjambre__->Part[i].Pfit=aux;
    //Almacena el indice de la mejor particula de todo el enjambre
```

```
    if(aux>BestFit){
      BestFit=aux;
      __Enjambre__->MejorParticulaDelGrupo=i;
    }
  }
  }
}


void ActualizarVelocidad(ENJAMBRE *__Enjambre__){
  float Y1,Y2;
  //Actualizar cada vector velocidad Vi de cada particula
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++) //
      Para cada particula i
    for(unsigned int j=0; j<__Enjambre__->CantidadDeParametros; j++) //
       Para cada parametro j de cada vector Vi de la particula i
    {
      Y1=(float)rand()/(float)RAND_MAX;
      Y2=(float)rand()/(float)RAND_MAX;
      __Enjambre__->Part[i].Vi[j] =(
        __Enjambre__->Part[i].Vi[j]+
        (__Enjambre__->C1*Y1*(__Enjambre__->Part[i].Pi[j]-__Enjambre__
            ->Part[i].Xi[j]))+
        (__Enjambre__->C2*Y2*(__Enjambre__->Part[__Enjambre__->
            MejorParticulaDelGrupo].Pi[j]-__Enjambre__->Part[i].Xi[j]))
      );
    }
}

void ActualizarVelocidadInerciaW(ENJAMBRE *__Enjambre__){
  float Y1,Y2;
  //Actualizar cada vector velocidad Vi de cada particula
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++) //
      Para cada particula i
    for(unsigned int j=0; j<__Enjambre__->CantidadDeParametros; j++) //
       Para cada parametro j de cada vector Vi de la particula i
    {
      Y1=(float)rand()/(float)RAND_MAX;
      Y2=(float)rand()/(float)RAND_MAX;
      __Enjambre__->Part[i].Vi[j] =(
          (__Enjambre__->Part[i].Vi[j]*__Enjambre__->X)+
        (__Enjambre__->C1*Y1*(__Enjambre__->Part[i].Pi[j]-__Enjambre__
            ->Part[i].Xi[j]))+
        (__Enjambre__->C2*Y2*(__Enjambre__->Part[__Enjambre__->
            MejorParticulaDelGrupo].Pi[j]-__Enjambre__->Part[i].Xi[j]))
      );
    }
}

void ActualizarPosicion(ENJAMBRE *__Enjambre__){
```

```
  // Acutailzsr cada vector Posicion XI de cada particula
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++) //
      Para cada particula i
    for(unsigned int j=0; j<__Enjambre__->CantidadDeParametros; j++) //
        Para cada parametro j de cada vector de la particula i
      __Enjambre__->Part[i].Xi[j] += __Enjambre__->Part[i].Vi[j];
}

void ActualizarMejoresPosiciones(ENJAMBRE *__Enjambre__){
  for(unsigned int i=0; i<__Enjambre__->CantidadDeParticulas; i++)
    if(__Enjambre__->Part[i].Xfit > __Enjambre__->Part[i].Pfit){
      __Enjambre__->Part[i].Pfit = __Enjambre__->Part[i].Xfit;
      for(unsigned int j=0; j<__Enjambre__->CantidadDeParametros; j++)
          //Para cada parametro j de cada vector de la particula i
        __Enjambre__->Part[i].Pi[j] = __Enjambre__->Part[i].Xi[j];
    }
}


/*float FuncionObjetivo(float *__ValoresDeParametros__, unsigned int
    __CantidadDeParametros__){
  unsigned int k;
  float fit, aux = 0;
  // Maximar la siguiente funcion:
  //    f(x,y)=50-(x-5)^2-(y-5)^2;
  //fit=250-pow(Xi[0]+7,2)-pow(Xi[1]-3,2)-pow(Xi[2]-3,2)-pow(Xi[3]-5,2)
      -pow(Xi[4]-8,2);

  // Funcion Rastriging (Buscamos el valor 0)
  for (k=0; k<__CantidadDeParametros__; k++)
    aux += pow(__ValoresDeParametros__[k],2)-10*cos(6.283185*
        __ValoresDeParametros__[k])+10;
  fit = 100 - aux; // El valor d'a precisión se ponderará en una escala
      del 1 al 100
  return fit;
}*/
```

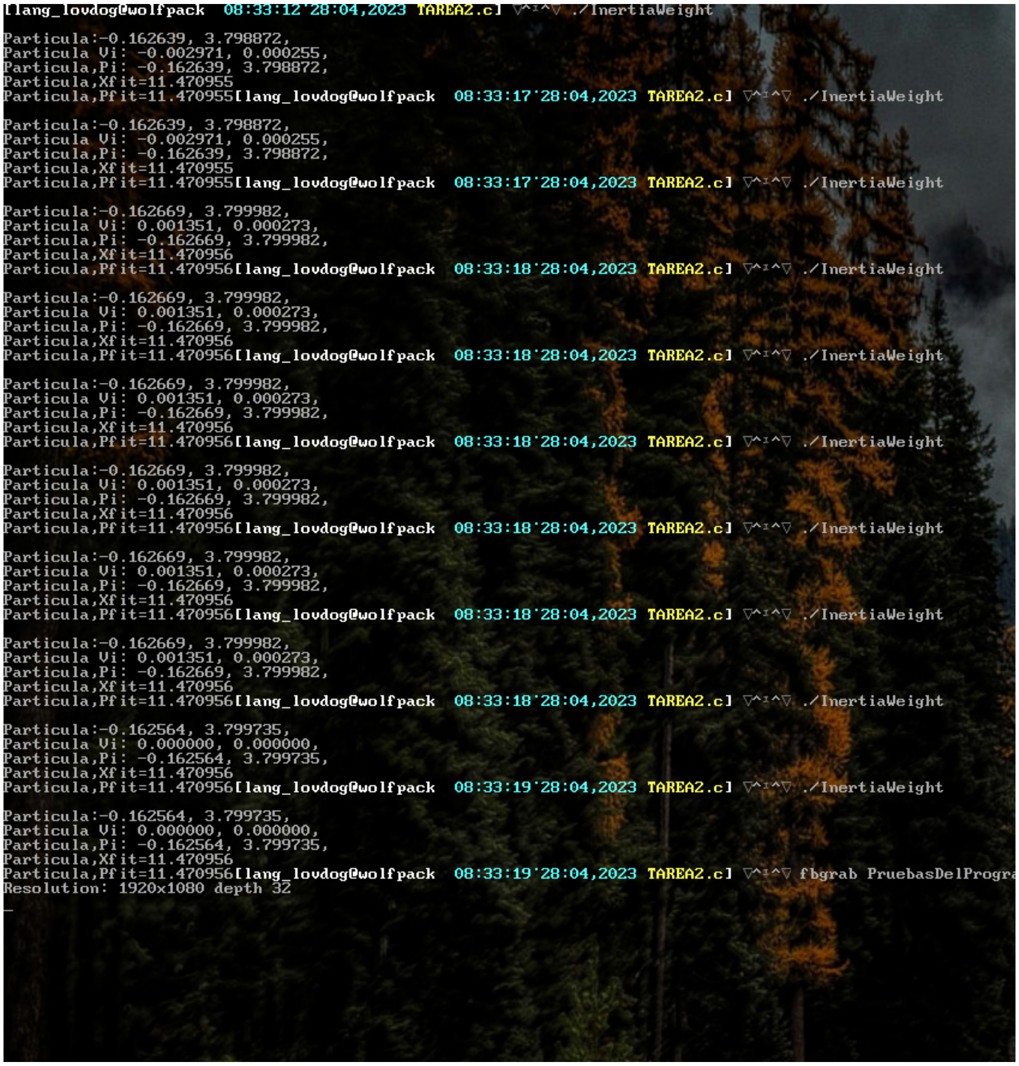
Programa principal, definición de modelo y proceso pso

```
#include <stdio.h>
#include <math.h>
#include "../pso.h"
#include <time.h>
#include <stdlib.h>

int main (void){
  //Programa que obtenga los valores
  // Limites={InfX, InfY, SupX, SupY}
  PARTICULA LaSelecta;
  const float Limites[4]={-6.28,-6.28,6.28,6.28};
```

```
  LaSelecta= ProcesoPSO(
    12, 2,
    Limites+2, Limites,
    20, 0, 2, 2,
    NULL
  );
  ImprimeParticula(&LaSelecta,2);
  free(LaSelecta.Pi);
  free(LaSelecta.Vi);
  free(LaSelecta.Xi);
}

  /*
     --------------------------------------------------------------------------
     */

float FuncionObjetivo(
    float         *__ValoresDeParametros__,
    unsigned int   __CantidadDeParametros__,
    const float   *__ParametrosDeOperacion__
){
  float f_xy=
    10*exp(
      -(pow(*(__ValoresDeParametros__)+1,2)+pow(*(
          __ValoresDeParametros__+1)-3.14,2))/25
    )+
    cos(*(__ValoresDeParametros__  )*2)+
    sin(*(__ValoresDeParametros__+1)*2);
  return f_xy;
}

PARTICULA ProcesoPSO(
    const float          __NumeroDeParticulas__,
    const float          __Dimension__,
    const float         *__LimiteSuperior__,
    const float         *__LimiteInferior__,
    const unsigned int   __NumeroMaximoDeIteraciones__,
    const float          __Factor_Constriccion_Inercia__,
    const float          __ValorPesoPersonalC1__,
    const float          __ValorPesoGlobalC2__,
    const float         *__ParametrosDeOperacion__
){
  ENJAMBRE *Enj=NULL;
  PARTICULA Particle;
  srand(time(NULL));
  unsigned int t=0;

  //Crear un enjambre de NumeroParticulas de Numero de parametros igual
      a Dimension
  Enj=CrearEnjambre(
```

```
    __NumeroDeParticulas__,
    __Dimension__
  );

  InicializarEnjambre(
    Enj,
    __Factor_Constriccion_Inercia__,
    __ValorPesoPersonalC1__,
    __ValorPesoGlobalC2__,
    __NumeroMaximoDeIteraciones__,
    __LimiteInferior__,
    __LimiteSuperior__
  );

  EvaluacionInicialEnjambre(Enj,__ParametrosDeOperacion__);

  while((t++)<Enj->MaximoDeIteraciones){
    ActualizarVelocidadInerciaW(Enj);
    ActualizarPosicion(Enj);
    EvaluarEnjambre(Enj,__ParametrosDeOperacion__);
    ActualizarMejoresPosiciones(Enj);
  }

  Particle.Xi=(Enj->Part+Enj->MejorParticulaDelGrupo)->Xi;
  (Enj->Part+Enj->MejorParticulaDelGrupo)->Xi=NULL;
  Particle.Vi=(Enj->Part+Enj->MejorParticulaDelGrupo)->Vi;
  (Enj->Part+Enj->MejorParticulaDelGrupo)->Vi=NULL;
  Particle.Pi=(Enj->Part+Enj->MejorParticulaDelGrupo)->Pi;
  (Enj->Part+Enj->MejorParticulaDelGrupo)->Pi=NULL;
  Particle.Xfit=(Enj->Part+Enj->MejorParticulaDelGrupo)->Xfit;
  Particle.Pfit=(Enj->Part+Enj->MejorParticulaDelGrupo)->Pfit;

  EliminarEnjambre(Enj);
  return Particle;
}
```

Pruebas y resultados



Conclusión

El PSO con inercia es un método que impide la disperión de los elementos, ya que, el primer modelo planteado, no tenía un control sobre'l movimiento d'as partículas. Este modelo, con un solo coeficiente, cambia la forma en que las partículas se comportan, ocasionalmente son más eficientes q'otros modelos. Sin embargo, es posible que haya ocasiones en las que'l metodo base (truncamiento de velocidad) sea

más eficiente.