<html><head></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">/*
Standard PSO version 2006

Motivation
Quite often some authors say they compare their PSO versions
to the "standard one" ... which is never the same!
So the idea is to define a real standard at least for one year, validated by some
researchers of the field, in particular James Kennedy and Maurice Clerc.
This PSO version does not intend to be the best one on the market (in particular there is
no adaptation of the swarm size nor of the coefficients) but simply very near of the
original version (1995) with just a few improvements based on some recent works.

So referring to "standard PSO 2006" would mean exactly this version with the default
values
detailed below as,for example, referring to "standard PSO 2006 (w=0.8)" would mean almost
this version but with a non standard first cognitive/confidence coefficient.

Parameters
S := swarm size
K := maximum number of particles _informed_ by a given one
T := topology of the information links
w := first cognitive/confidence coefficient
c := second cognitive/confidence coefficient
R := random distribution of c
B := rule "to keep the particle in the box"

Equations
For each particle and each dimension
v(t+1) = w*v(t) + R(c)*(p(t)-x(t)) + R(c)*(g(t)-x(t))
x(t+1) = x(t) + v(t+1)
where
v(t) := velocity at time t
x(t) := position at time t
p(t) := best previous position of the particle
g(t) := best previous position of the informants of the particle

Default values
S = 10+2*sqrt(D) where D is the dimension of the search space
K = 3
T := randomly modified after each step if there has been no improvement
w = 1/(2*ln(2))
c = 0.5 + ln(2)
R = U(0..c), i.e. uniform distribution on [0, c]
B := set the position to the min. (max.) value and the velocity to zero

About information links topology
A lot of works have been done about this topic. The main result is there is no
"best" topology. Hence the random approach used here. Note that is does not mean that
each particle is informed by K ones: the number of particles that informs a given one
may be any value between 1 (for each particle informs itself) and S.

About initialisation
Initial positions are chosen at random inside the search space (which is
supposed to be a hyperparallelepid, and even often a hypercube), according to an uniform
distribution. This is not the best way, but the one of the original PSO.

Each initial velocity is simply defined as the difference of two random
positions. It is simple, and needs no additional parameter.
However, again, it is not the best approach. The resulting distribution is not even
uniform, as for any method that uses an uniform distribution independently for each
</pre></body></html>

component. The mathematically correct approach needs to use an uniform distribution inside a hypersphere. It is not that difficult, and indeed used in some PSO versions, but quite different from the original one.

Some results with the standard values
You may want to recode this program in another language. Also you may want to modify it for your own purposes. Here are some results on classical test functions to help you to check your code.
Dimension D=30
Acceptable error eps=0
Objective value f_min=0
Number of runs n_exec_max=50
Number of evaluations for each run eval_max=30000

Problem	Mean best value	Standard deviation
Parabola/Sphere on [-100, 100]^D	0	0
Griewank on [-600, 600]^D	0.018	0.024
Rosenbrock/Banana on [-30, 30]^D	50.16	36.9
Rastrigin on [-5.12, 5.12]^D	48.35	14.43
Ackley on [-32, 32]^D	1.12	0.85

Last updates
2006-02-27 Fixed a bug about minimal best value over several runs
2006-02-16 Fixed a bug (S_max for P, V, X, instead of D_max), thanks to Manfred Stickel
2006-02-16 replaced k by i x by xs (in perf()), because of possible confusion with K and X
2006-02-13 added De Jong's f4
*/

```
#include "stdio.h"
#include "math.h"
#include &lt;stdlib.h&gt;
#include &lt;time.h&gt;

#define D_max 100  // Max number of dimensions of the search space
#define S_max 100 // Max swarm size
#define R_max 200 // Max number of runs


// Structures
struct velocity
{
  int size;
  double v[D_max];
};


struct position
{
  int size;
  double x[D_max];
  double f;
};



// Sub-programs
double alea( double a, double b );
int alea_integer( int a, int b );
double perf( int s, int function ); // Fitness evaluation

// Global variables
int best; // Best of the best position (rank in the swarm)
```

```
int D; // Search space dimension
double E; // exp(1). Useful for some test functions
double f_min; // Objective(s) to reach
int LINKS[S_max] [S_max]; // Information links
int nb_eval; // Total number of evaluations
double pi; // Useful for some test functions
struct position P[S_max]; // Best positions found by each particle
int S; // Swarm size
struct velocity V[S_max]; // Velocities
struct position X[S_max]; // Positions
double xmin[D_max], xmax[D_max]; // Intervals defining the search space

// File(s)
FILE * f_run;

// ================================================
int main()
{
  double c; // Second onfidence coefficient
  int d; // Current dimension
  double eps; // Admissible error
  double eps_mean; // Average error
  double error; // Error for a given position
  double error_prev; // Error after previous iteration
  int eval_max; // Max number of evaluations
  double eval_mean; // Mean number of evaluations
  int function; // Code of the objective function
  int g; // Rank of the best informant
  int init_links; // Flag to (re)init or not the information links
  int i;
  int K; // Max number of particles informed by a given one
  int m;
  double mean_best[R_max];
  double min; // Best result through several runs
  int n_exec, n_exec_max; // Nbs of executions
  int n_failure; // Number of failures
  int s; // Rank of the current particle
  double t1, t2;
  double variance;
  double w; // First confidence coefficient

  f_run = fopen( "f_run.txt", "w" );
  E = exp( 1 );
  pi = acos( -1 );

  //------------------------------------------------ PROBLEM
  function = 13; //Function code
  /*
  0 Parabola (Sphere)
  1 De Jong' f4
  2 Griewank
  3 Rosenbrock (Banana)
  4 Step
  6 Foxholes 2D
  7 Polynomial fitting problem 9D
  8 Alpine
  9 Rastrigin
  10 Ackley
  13 Tripod 2D
  17 KrishnaKumar
  18 Eason 2D
```

```
  */

  D = 2; // Search space dimension

  // D-cube data
  for ( d = 0; d &lt; D; d++ )
  {
    xmin[d] = -100; xmax[d] = 100;
  }

  eps = 0.9; // Acceptable error
  f_min = 0; // Objective value
  n_exec_max = 100; // Numbers of runs
  eval_max = 10000; // Max number of evaluations for each run

  if(n_exec_max&gt;R_max) n_exec_max=R_max;
  //--------------------------------------------------  PARAMETERS
  S = 10+( int )(2*sqrt(D)); if (S&gt;S_max) S=S_max;
  K = 3;
  w = 1 / ( 2 * log( 2 ) ); c = 0.5 + log( 2 );

printf("\n Swarm size %i", S);
printf("\n coefficients %f %f \n",w,c);

  //-------------------------------------------------- INITIALISATION
  t1 = clock(); // Init time
  // Initialisation of information variables
  n_exec = 0; eval_mean = 0; eps_mean = 0; n_failure = 0;

init:
n_exec = n_exec + 1;
for ( s = 0; s &lt; S; s++ )  // Positions and velocities
{
  X[s].size = D; V[s].size = D;

  for ( d = 0; d &lt; D; d++ )
  {
    X[s].x[d] = alea( xmin[d], xmax[d] );
   V[s].v[d] = (alea( xmin[d], xmax[d] ) - X[s].x[d])/2; // Non uniform
    // V[s].v[d] = ( xmin[d]-xmax[d] )*(0.5-alea(0,1)); //Uniform. 2006-02-24
  }
}

  // First evaluations
  nb_eval = 0;
  for ( s = 0; s &lt; S; s++ )
  {
    X[s].f = fabs( perf( s, function ) - f_min );
    P[s] = X[s]; // Best position = current one
  }

  // Find the best
  best = 0;
  for ( s = 1; s &lt; S; s++ )
    if ( P[s].f &lt; P[best].f ) best = s;

  error =  P[best].f ; // Current min error
  if(n_exec==1) min=error;
  error_prev=error; // Previous min error

  init_links = 1; // So that information links will be initialized
```

```
  //--------------------------------------------- ITERATIONS
loop:
  if ( init_links == 1 )
  {
    // Who informs who, at random
    for ( s = 0; s &lt; S; s++ )
    {
      for ( m = 0; m &lt; S; m++ ) LINKS[m] [s] = 0; // Init to "no link"
      LINKS[s] [s] = 1; // Each particle informs itself
    }

    for ( m = 0; m &lt; S; m++ ) // Other links
    {
      for ( i = 0; i &lt; K; i++ )
      {
        s = alea_integer( 0, S - 1 );
        LINKS[m] [s] = 1;
      }
    }
  }

  // The swarm MOVES
  for ( s = 0; s &lt; S; s++ ) // For each particle ...
  {
    // .. find the best informant
    g=s;
  for ( m = 0; m &lt; S; m++ )
  {
  if ( LINKS[m] [s] == 1 &amp;&amp; P[m].f&lt;P[g].f  ) g = m;
  }

  // ... compute the new velocity, and move
      for ( d = 0; d &lt; D; d++ )
      {
        V[s].v[d] = w * V[s].v[d] + alea( 0, c ) * ( P[s].x[d] - X[s].x[d] );
        V[s].v[d] = V[s].v[d] + alea( 0, c ) * ( P[g].x[d] - X[s].x[d] );
        X[s].x[d] = X[s].x[d] + V[s].v[d];
      }

  // ... interval confinement (keep in the box)
  for ( d = 0; d &lt; D; d++ )
   {
     if ( X[s].x[d] &lt; xmin[d] )
     {
       X[s].x[d] = xmin[d]; V[s].v[d] = 0;
     }
     if ( X[s].x[d] &gt; xmax[d] )
     {
       X[s].x[d] = xmax[d]; V[s].v[d] = 0;
     }
   }


  // ... evaluate the new position
    X[s].f = fabs( perf( s, function ) - f_min );


  // ... update the best previous position
  if ( X[s].f&lt;P[s].f )
  {
```

```
    P[s] = X[s];
  // ... update the best of the bests
  if (  P[s].f&lt;P[best].f  ) best = s;
   }
}

// Check if finished
// If no improvement, information links will be reinitialized
error=P[best].f;
if ( error &gt;= error_prev ) init_links = 1;
else init_links = 0;
error_prev = error;

if ( error &gt; eps &amp;&amp; nb_eval &lt; eval_max ) goto loop;
if ( error &gt; eps ) n_failure = n_failure + 1;

// Result display
printf( "\nExec %i Eval %i. Error %f ", n_exec, nb_eval, error );
printf( "\n Position :\n" );
for ( d = 0; d &lt; D; d++ ) printf( " %f", P[best].x[d] );

// Save result
fprintf( f_run, "\n%i %i %f ", n_exec, nb_eval,error );
fprintf( f_run, " Position: " );
for ( d = 0; d &lt; D; d++ ) fprintf( f_run, " %f", P[best].x[d] );

// Compute some statistical information
if ( error &lt; min ) min = error;
eval_mean = eval_mean + nb_eval;
eps_mean = eps_mean + error;
  mean_best[n_exec - 1] = error;

  if ( n_exec &lt; n_exec_max ) goto init;

  // END. Display some statistical information
  t2 = clock();
  printf( "\n\n Total clocks %.0f", t2 - t1 );
  eval_mean = eval_mean / ( double )n_exec;
  eps_mean = eps_mean / ( double )n_exec;
  printf( "\n\n Eval. (mean)= %f", eval_mean );
  printf( "\n Error (mean) = %f", eps_mean );

  // Variance
  variance = 0;
  for ( d = 0; d &lt; n_exec_max; d++ ) variance = variance + ( mean_best[d] - eps_mean
) * ( mean_best[d] - eps_mean );
  variance = sqrt( variance / n_exec_max );
  printf( "\n Std. dev. %f", variance );

  // Success rate and minimum value
  printf( "\n Success rate = %.2f%%", 100 * ( 1 - n_failure / ( double )n_exec ) );
  if ( n_exec &gt; 1 ) printf( "\n Best min value = %f", min );

end:;
  return 0;
}

//==========================================================
double alea( double a, double b )
{ // random number (uniform distribution) in [a b]
  double r;
```

```
   r=(double)rand(); r=r/RAND_MAX;
  return a + r * ( b - a );
}
//============================================================
int alea_integer( int a, int b )
{ // Integer random number in [a b]
  int ir;
  double r;
  r = alea( 0, 1 ); ir = ( int )( a + r * ( b + 1 - a ) );
  if ( ir &gt; b ) ir = b;
  return ir;
}


//============================================================
double perf( int s, int function )
{ // Evaluate the fitness value for the particle of rank s
  double c;
  int d, d1;
  int i, j, k;
  double f, f1, p, xd, x1, x2, x3, x4;
  double min;
  double sum1, sum2;
  double t0, tt, t1;
  struct position xs;

  // For Foxholes problem
  static int a[2] [25] =
  {
    {
      -32, -16, 0, 16, 32, -32, -16, 0, 16, 32, -32, -16, 0, 16, 32, -32, -16, 0, 16,
32, -32, -16, 0, 16, 32
    },
    {
      -32, -32, -32, -32, -32, -16, -16, -16, -16, -16, 16, 16, 16, 16, 16, 32, 32, 32,
32, 32
    }
  };
  // For polynomial fitting problem
  int const M = 60;
  double py, y = -1, dx = ( double )M;

  nb_eval = nb_eval + 1;
  xs = X[s];

  switch ( function )
  {
    case 0: // Parabola (Sphere)
      f = 0;
      p = 0; // Shift
      for ( d = 0; d &lt; D; d++ )
      {
        xd = xs.x[d] - p;
        f = f + xd * xd;
      }
    break;

    case 1: // De Jong's f4
      f = 0;
      p = 0; // Shift
      for ( d = 0; d &lt; D; d++ )
```

```
  {
    xd = xs.x[d] - p;
    f = f + (d+1)*xd*xd*xd*xd;
  }
break;


case 2: // Griewank
  f = 0;
  p = 1;
  for ( d = 0; d &lt; D; d++ )
  {
    xd = xs.x[d];
    f = f + xd * xd;
    p = p * cos( xd / sqrt( d + 1 ) );
  }
  f = f / 4000 - p + 1;
break;

case 3: // Rosenbrock
  f = 0;
  t0 = xs.x[0];
  for ( d = 1; d &lt; D; d++ )
  {
    t1 = xs.x[d];
    tt = 1 - t0;
    f += tt * tt;
    tt = t1 - t0 * t0;
    f += 100 * tt * tt;
    t0 = t1;
  }
break;

case 4: // Step
  f = 0;
  for ( d = 0; d &lt; D; d++ ) f = f + ( int )xs.x[d];
break;


case 6: //Foxholes 2D
  f = 0;
  for ( j = 0; j &lt; 25; j++ )
  {
    sum1 = 0;
    for ( d = 0; d &lt; 2; d++ )
    {
      sum1 = sum1 + pow( xs.x[d] - a[d] [j], 6 );
    }
    f = f + 1 / ( j + 1 + sum1 );
  }
  f = 1 / ( 0.002 + f );
break;

case 7: // Polynomial fitting problem
  // on [-100 100]^9
  f = 0;
  dx = 2 / dx;
  for ( i = 0; i &lt;= M; i++ )
  {
    py = xs.x[0];
    for ( d = 1; d &lt; D; d++ )
```

```
    {
      py = y * py + xs.x[d];
    }
    if ( py &lt; -1 || py &gt; 1 ) f += ( 1 - py ) * ( 1 - py );
    y += dx;
  }
  py = xs.x[0];
  for ( d = 1; d &lt; D; d++ ) py = 1.2 * py + xs.x[d];
  py = py - 72.661;
  if ( py &lt; 0 ) f += py * py;
  py = xs.x[0];
  for ( d = 1; d &lt; D; d++ ) py = -1.2 * py + xs.x[d];
  py = py - 72.661;
  if ( py &lt; 0 ) f += py * py;
break;

case 8: // Clerc's f1, Alpine function, min 0
  f = 0;
  for ( d = 0; d &lt; D; d++ )
  {
    xd = xs.x[d];
    f += fabs( xd * sin( xd ) + 0.1 * xd );
  }
break;

case 9: // Rastrigin. Minimum value 0. Solution (0,0 ...0)
  k = 10;
  f = 0;
  for ( d = 0; d &lt; D; d++ )
  {
    xd = xs.x[d];
    f += xd * xd - k * cos( 2 * pi * xd );
  }
  f += D * k;
break;

case 10: // Ackley
  sum1 = 0;
  sum2 = 0;
  for ( d = 0; d &lt; D; d++ )
  {
    xd = xs.x[d];
    sum1 += xd * xd;
    sum2 += cos( 2 * pi * xd );
  }
  y = D;
  f = ( -20 * exp( -0.2 * sqrt( sum1 / y ) ) - exp( sum2 / y ) + 20 + E );
break;


case 13: // 2D Tripod function (Louis Gacogne)
  // Search [-100, 100]^2. min 0 on (0  -50)
  x1 = xs.x[0];
  x2 = xs.x[1];
  if ( x2 &lt; 0 )
  {
    f = fabs( x1 ) + fabs( x2 + 50 );
  }
  else
  {
    if ( x1 &lt; 0 )
```

```
          f = 1 + fabs( x1 + 50 ) + fabs( x2 - 50 );
        else
          f = 2 + fabs( x1 - 50 ) + fabs( x2 - 50 );
      }
    break;

    case 17: // KrishnaKumar
      f = 0;
      for ( d = 0; d &lt; D - 1; d++ )
      {
        f = f + sin( xs.x[d] + xs.x[d + 1] ) + sin( 2 * xs.x[d] * xs.x[d + 1] / 3 );
      }
    break;

    case 18: // Eason 2D (usually on [-100,100]
      // Minimum -1  on (pi,pi)
      x1 = xs.x[0]; x2 = xs.x[1];
      f = -cos( x1 ) * cos( x2 ) / exp( ( x1 - pi ) * ( x1 - pi ) + ( x2 - pi ) * ( x2 -
pi ) );
    break;
   }
   return f;
  }
</pre></body></html>
```