



University of Paderborn

Faculty EIM

Assignment

Particle Swarm Optimization

Thomas-Ivo Heinen

presented to
Prof. Dr. Hans Kleine Büning

Contents

1. Introduction	2
1.1. Swarm Intelligence	2
1.2. Particle Swarm Optimization (PSO)	2
2. Swarm Intelligence in Nature	3
2.1. Fish Schools and Bird Flocks	3
2.2. Ants	3
2.3. Termites	4
3. History	6
3.1. Reynolds, Boids and the Lord of the Rings	6
3.1.1. Behavioral Animation: Lord of the Rings	6
3.2. Heppner and the Roosting area	7
3.3. Hoffmeyer: Semiotics	7
3.4. Kennedy and Eberhart: Ignition of the PSO	8
4. The Algorithm	9
4.1. Abstracting the principles	9
4.2. Adjusting the velocity	9
4.3. Neighbourhoods	10
4.4. Topologies	11
4.5. The Optimisation Process	11
5. Summary	13
5.1. Comparison to other EAs	13
5.2. Applications	13
5.3. Conclusion	14
A. An PHP Swarm optimizer	15

1. Introduction

This paper is about particle swarm optimisation (PSO), certainly a term which sounds like theoretical stuff with little practical relevance - but this isn't true. PSO instead is a very practically-oriented way of optimizing functions and emerged from biological sciences. It is probably one of the newest research topics in the field of evolutionary algorithms (EA).

1.1. Swarm Intelligence

In literature about PSO the term of swarm intelligence appears rather often, sometimes being used as replacement word for the more technical term of PSO. So where does this emerge from?

Early research which led into the theory of particle swarms was done by non-computer scientists: ornithologists, biologists, psychologists. In this areas the term of swarm intelligence is well-known and characterizes the possibility of a large number of individuals being able to do quite complex tasks. So this is more of the biological side of PSO theory.

1.2. Particle Swarm Optimization (PSO)

Later some basic simulations of swarms were abstracted into the mathematical field. The usage of swarms for solving simple tasks in nature became an intriguing idea in algorithmics and function optimisation.

2. Swarm Intelligence in Nature

The context out of which PSO arose can be found in nature. Many species employ some form of swarm intelligence where rather simple individuals in large numbers solve astonishingly complex tasks.

The knowledge of some of these examples is a good base for extending into the more theoretical particle swarms later on.

2.1. Fish Schools and Bird Flocks

A rather simple and generic approach to the task of solving problems can be found in fish schools or bird flocks. By forming a swarm of hundreds or thousands of animals an effective protection is given from enemies which are scared off by the huge size of this organism which is moving around.



Figure 2.1.: Bird Flock + Fish School

Although this is not exactly what motivated the computer sciences equivalent, this is possibly the simplest possible example for the benefits of a swarm.

2.2. Ants

A well-known example for swarm intelligence is the principle by which ants learn how to find their way around newly appeared obstacles.

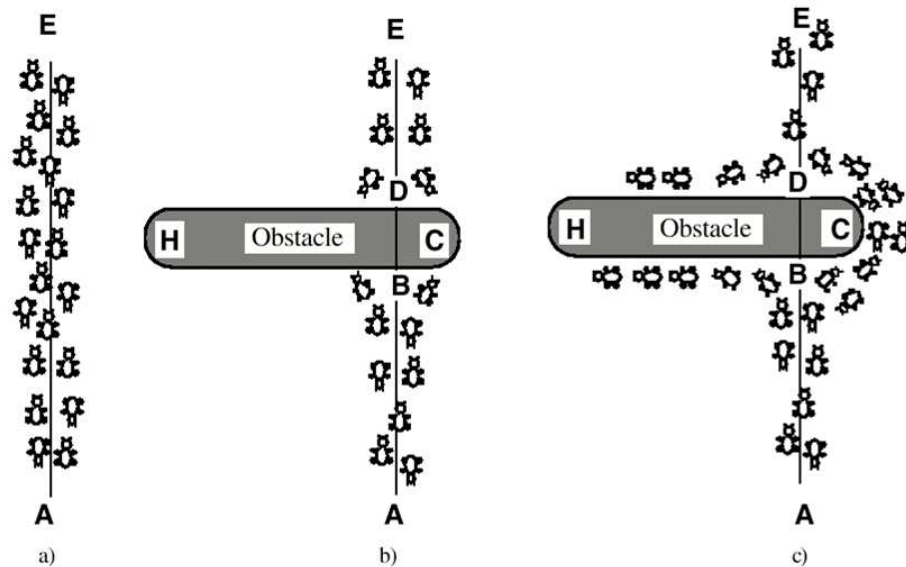


Figure 2.2.: Ants moving around an obstacle

It is important to note that ants, while moving, leave a trail of pheromones behind them. So when an obstacle occurs, half of the ants try to get around it on the right and half on the left (this is basically some random action and not part of any rule). If the shortest path around the obstacle is found this increases the in pheromone concentration over time as more and more ants use it and after some time all ants will choose it.

There are similar ways of solving other problems in ant colonies and it is worth a mention that there are other forms of communication present, too. While most people who read this example may think that ants primarily communicate using pheromones, this is totally wrong. Ants also have an expressive form of communication by touching each other and their main paths to food use geometrical properties (angle sizes) as replacement for direction signs [2].

2.3. Termites

An astonishing example of one of the most complex results created by swarms can be found in the world of termites.

Termite nests are huge compared with the size of a single individual, about factor 1:300 between termite and its nest. Even now, humans have yet to build skyscrapers as tall as 600 meters, which would be needed to be on par with these little insects.

Even more interesting is the climatisation of these nests, as they cool down air using tunnels below the earth to suck air into the nest. Then this cooled air gets directed



Figure 2.3.: A termite nest

into the nest, thus cooling it down even in the hottest regions of earth. With colony sizes ranging between 1,000,000 and 70,000,000 insects [3] the termites even grow mushrooms in some chambers of their nest or dig down up to 40 meters for water supply [4].

So if now thinking about the basic nest-building principle, some much more sophisticated swarm intelligence than that of fish or birds is found.

Grassé [5] discovered the basic rules of this to be:

Move at random + build heaps First all termites move at random and drop pellets of earth on elevated positions. This results in forming small heaps.

Put more pellets on top of existing heaps If a termite meets a heap it starts putting other pellets of earth onto its top, forming a vertical column. This process is stopped at a species-specific height. This effectively is indirect communication using stigmergy (markers which are put into environment and trigger actions of other individuals).

Search for clusters and connect them In the next step the termites look for heaps which are grouped next to each other. Isolated heaps get disregarded but the heap clusters get chosen for being connected with each other by building walls.

In later literature this was renamed to sematectonic communication as it is connected with building up structures.

3. History

Although the principle of Particle Swarm Optimization is a quite new approach, its ancestors reach back into history as it emerged from biological research and simulation on swarming animals.

3.1. Reynolds, Boids and the Lord of the Rings

The first computer-related work in this area was provided by Craig Reynolds which published a paper about simulating bird swarms on SigGraph 1986 [6]. His idea was to simulate realistic swarms mainly for computer graphics and movies.

The result was some simulated swarm of whose the individuals he called “Boids“ [7]. These were directed by three simple rules, which were implemented and caused a near-realistic swarming behaviour:

- Separation: Do not run into flockmates
- Alignment: Align the own heading to the average of the neighbours
- Cohesion: Move toward the average position of neighbours.

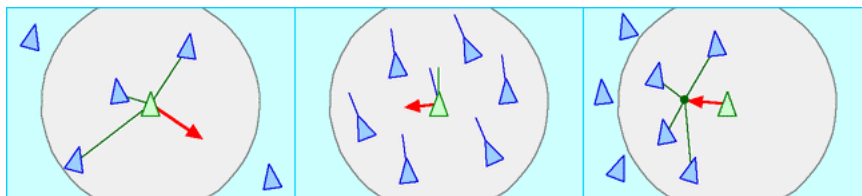


Figure 3.1.: Separation, Alignment, Cohesion

3.1.1. Behavioral Animation: Lord of the Rings

The concepts embedded into the Boids were refined and later led into some new area of computer graphics which is called behavioral animation. Examples for this technology can be found in Batman Returns (1992), The Lion King (1994) and From Dusk Till Dawn (1996). The most impressive usage are probably the immense battle sequences in the trilogy Lord of the Rings where about 250,000 individual

3. History

fighters. For making this possible a new software was written named MASSIVE which controls this mass of agent technology-equipped computer actors (CGIs) and their states.



Figure 3.2.: Animation phases within “Massive“

A quite funny anecdote about this battle sequence is that in the early testing-runs it was working way too good. The directors noticed some group of orcs which fled the battle because they were too scared. This was adjusted later on, as orcs are said to fear nothing at all...

3.2. Heppner and the Roosting area

In 1990 ornithologist Frank Heppner reviewed Reynold’s algorithms for doing more detailed bird flock animation and studies [8]. As he quickly realized, an important concept was missing: roosting areas. To introduce these he developed a desire factor per bird which is representing the need of roosting or swarming and which gets stronger if a defined roosting area is approached.

As a result of the roosting areas and the swarming rules of Reynolds the effect was very nature like and was good enough for Heppner to continue his studies.

3.3. Hoffmeyer: Semiotics

Jesper Hoffmeyer, a biologist, studies the area of semiotics and published some of his works in 1995[9]. He released one of the best definitions of a swarm in concepts of algorithmics:

A swarm has been defined as a set of (mobile) agents which are liable to communicate directly or indirectly (by acting on their local environment) with each other, and which collectively carry out a distributed problem solving.

3.4. Kennedy and Eberhart: Ignition of the PSO

Research about the computer-applicable Particle Swarm Optimizations was really started after Kennedy and Eberhart published their book in 2001[10].

These two are the real inventors of the PSO in terms of algorithmic implementation for solving optimization tasks of many kinds. Once again Kennedy and Eberhart are no computer scientists but come out of socio-psychology and engineering.

Their book reflects that, as it does not really mention details of computer sciences for the first 290 pages until it comes to implementation issues. The focus is placed more upon animal wildlife and psychology, detailing the underlying principles of swarms and collective intelligence.

Regardless of this, it was like an ignition in the field of EA and from year to year more papers on PSO get published.

4. The Algorithm

Finally it is time to cover the actual abstraction from simulated flocks, herds and swarms to mathematical problem solving. In this chapter the abstraction process is described as well as the most important factors in PSO.

4.1. Abstracting the principles

It is not too hard to abstract the combined concepts of Reynolds, Heppner and Kennedy/Eberhart. If the real-world parts are listed up it is fairly easy to get an abstracted version.

First of all, the world in nature becomes the solution space, which consists of all solution candidates. As the wanted effect is e.g. solving some equation, this solution space is an n-dimensional vector space where the variable n equals the number of unknowns in the equation to optimize.

In effect, the animals are abstracted into software agents which reside at a position in this n-dimensional space, having a velocity and a fitness at their position regarding the problem. This fitness is roughly comparable to the “desire“ in Heppner’s bird simulation.

As consequence the roosting areas out of the same model can be seen as the local optima in the problem that should be solved. It is worth a notice that of course the global optimum (which is searched for) is a local optimum in its environment, too.

The central question of implementation now is, how to avoid all particles to move towards some local optimum and leaving the global one by side. For achieving a convergence towards the global optimum the particles need some information about their environment. For this the ratio of exploration and exploitation needs to be adjusted.

4.2. Adjusting the velocity

This adjustment takes place in updating particle velocities each step. Of course there are many different ways of doing that, given is an example by using these individual properties of the particles:

4. The Algorithm

- $i \in [0..1]$ is the individuality of the particle, the amount of exploration it desires to do.
- $p_{id} \in \mathbb{R}_n$ is the best position of the particle so far
- $s \in [0..1]$ is the sociality factor; how much it is exploiting the knowledge of other particles
- $p_{gv} \in \mathbb{R}_n$ is the best neighbours best known position so far

For adjusting the velocity these values are combined:

$A_0 = i * p_{id}$ is called the *attitude* of the particle. It quantifies the portion of own desire to explore and the best solution so far.

$SN_0 = s * p_{gv}$ is the *subjective norm*, a number which measures the particle's connection to other particles knowledge and its tendency to move towards their best points.

Combined this leads to

$$\Delta v_i = A_0 + SN_0 = i * p_{id} + s * p_{gv}$$

and this is change in velocity per step. As it involves some swarming component in form of the subjective norm this is the “heart“ of the PSO.

As already mentioned there needs to be some connection between the particles to be defined, this is called a neighbourhood.

4.3. Neighbourhoods

There are two main types of neighbourhoods defined.

On the one side the most intuitive neighbourhood is the global neighbourhood which is also called *gbest*. In it, every particle has access to the fitness and best value so far of all other particles in the swarm. But this can become problematic, as this way the first particle which meets a local optimum could cause the others to join it. So in multi-modal solution spaces there is a too strong convergence as too much exploitation is present and the exploration is too weak.

As an alternative there is a local neighbourhood, where a particle is just connected to a fragmentary number of particles of its swarm. This way these groups exchange information about local optima and because of the sheer number of groups the possibility is higher to find the global optimum of the problem/function. So the exploitation factor is weakened and the exploration component is getting stronger.

It is important that, from the algorithmical point, it is not a necessity to build the neighbourhoods by distance as this would be the case in a natural swarm. There

is a similar phenomenon in the world that show this: Chatters. Think of a group of people all over the world which are in the same internet chat room. Of course this group would also be able to solve some given puzzle. Same would be true for particle swarms with e.g. an age-based neighbourhood mechanism.

Local neighbourhoods get termed lbest-k neighbourhoods if every particle has access to k others. On another hand this is also called a “k+1“ neighbourhood (number of neighbours plus the central particle which is observed).

4.4. Topologies

It is a long-known fact that the structure of social structures primarily affects the communication abilities and thus the group’s performance.

In the case of a global neighbourhood the structure is a fully connected network where every particle has access to the others’ best position. But in local neighbourhoods there are more variants possible.

One common structure for a local neighbourhood (lbest-2) is the circle topology. In this one individuals which are far away from others (in terms of graph structure, not necessarily distance) are independent of each other but neighbours are closely connected. So it may be the case that one segment of the swarm converges on a local optimum but another segment converges on another one.

Another structure is called wheel topology and has a more hierarchical structure, because all members of the neighbourhood are connected to one leader individual. So all information has to be communicated through this focal individual, which then compares the performances of all others and adjusts its trajectory towards the best of them. So if these adjustments improve the leader’s performance, this will affect the rest of the group members (think of some sort of information filter which slows down communication speed). This is very similar to the hierarchy of companies and organizations. This buffering is meant to prevent too quick convergence on local optima and used as a way to preserve diversity of potential solutions.

4.5. The Optimisation Process

As in all EA the first step in the optimisation is the initialisation. Here, we initialize the particles with a random n-dimensional position and velocity, as well as a very low fitness value such as $-\infty$. The saved best fitness so far and the corresponding position can be left off uninitialised. It is important that the initial random values cover an appropriate area of the solution space, so that likely global optima are contained in this, otherwise it will be much harder to find them.

4. *The Algorithm*

On initialisation it is also necessary to initialise the neighbourhoods. If the design choice of non-distance oriented neighbours was made, this is the one and only position where the neighbourhoods get fixed.

For practical purposes it comes handy to define a maximum number of possible iterations on the problem, as well as a threshold of the fitness value where program flow is aborted as the solution is close enough.

The three steps of the iterations are then rather simple

- Calculate fitness at current position of all particles
- Adjust the velocity according to the exploration/exploitation rule
- Change the positions by adding the velocity to the current position
- If distance-based neighbourhoods are used adjust them accordingly to position changes

On each finished iteration then the results are checked against the values for maximum of iterations and for the threshold being met.

Further restrictions on the maximum movement distance per step are also advisable, as well as adjusting this limit according to the particle's current fitness value. This way the particle will not warp beyond an optimum but explore the area more carefully if it notices a raise in fitness.

5. Summary

5.1. Comparison to other EAs

PSO is a bit different from other evolutionary algorithms because there are no generations and the basic operators of selection, crossover and mutation seem to be absent. But this is just partially the case.

The basic EA term of *convergence* is represented in PSO by the exploitation/sociality factor which steers the particles' movements towards a good solution. It's EA contrary, the *diversity* in turn equals to the exploration/individuality factor.

A particle basically is comparable to the *individual* of another evolutionary algorithm although it remains "alive" during the whole run and has some embedded intelligence (software agent) whereas the individuals of e.g. GA are just representations of problem solving strategies. The particles form a swarm, which is then same as the *population* in classical EAs.

A term which is identical is the *fitness*.

Selection and *Crossover* are in some way comparable to the PSO's Neighbourhoods and Sociality factor as the formula for adjusting the particle's velocity mixes in information of other particles and form a crossover-equivalent. Selection is then simply the progress of choosing the best available information out of neighbourhood context.

Mutation does also occur in a subliminal way, as the individuality and sociality factors are adjusted randomly in each iteration and thus vary the results.

Basically all the classical operations of evolutionary algorithms are done on the level of information with PSOs instead on some species' genotype. Thus they serve another dimension of evolution by evolving memes rather than genes.

5.2. Applications

PSO are best-suited for function optimisation tasks. Their structure makes them some sort of plug-and-play solution, while it's still needed to tune some parameters such as initialisation area, swarm size and neighbourhoods.

5. Summary

There have also been proofs of PSO being able to solve the Travelling Salesman Problem and doing multi-objective optimisation tasks.[10]

On the other side the ability to optimise functions makes PSO effective for using as a meta-EA algorithm for e.g. adjusting neural net weights or some parameters to other evolutionary algorithms and techniques.[10]

5.3. Conclusion

As we have seen PSO is a mighty tool for function optimization which in addition is quite easy to implement. They offer a robust environment for various optimization tasks and don't even fail completely if some subset of particles converges on a local optimum instead of the global one. For PSO's one statement is very important:

The groups abilities are more than the sum of its particles

A. An PHP Swarm optimizer

I am including the PHP code which I wrote for illustrating my example in the presentation slides. It is just a rudimentary example implementation of simple PSO without special properties.

The reasons for the unusual choice of PHP were:

- I am most used to this language
- PHP is a rapid development language without memory management etc
- I wanted to present it live during presentation without installation
- It is probably the first and only EA written in PHP

Please forgive the kludgy formatting in some areas, it was meant to be a proof of concept and no reusable library of some sort.

```
<?php
```

```
    ini_set("max_execution_time", 300);
```

```
    //=====
    // Swarm parameters
```

```
    // Control variables
    $numberOfParticles = 200;
    $numberOfNeighbors = 25;
    $maxIterations = 200;
    $threshold = 0.001;
```

```
    // Solution Space Size (-sss .. +sss)
    $sss = 50;
```

```
    // Limits for location changes
    $deltaMin = -4.0;
    $deltaMax = 4.0;
```

A. An PHP Swarm optimizer

```
// Set individuality and sociality
$iWeight = 2.0;
$iMin = 0.0;
$iMax = 1.0;
$sWeight = 2.0;
$sMin = 0.0;
$sMax = 1.0;

// Whether to use distance-based model or chatter model
$distanced = false;

// Related to problem
$initialFitness = -1000000;
$targetFitness = 0;
$dimensions = 4;

// If given, take parameters from request
if (count($_GET) > 0) {
    $numberOfParticles = $_GET["p"];
    $numberOfNeighbors = $_GET["k"];
    $maxIterations = $_GET["i"];
    $sss           = $_GET["s"];
}

//=====
// Swarm problem

/**
 * This tests the particle against the function wanted.
 *
 * @param Particle Particle to test
 * @returns Integer Fitness of particle
 */
function test($particle) {
    global $initialFitness;

    // Extract current position
    list($x, $y, $z, $w) = $particle->current;

    // Take care of divisions by zero
```

A. An PHP Swarm optimizer

```
    if ($w == 0) $w = 0.00001;

    // The problem which we wanna solve
    $f = 5*pow($x, 2) + 2*pow($y, 3)
        - 2*($z / $w) + 4;

    /* fitness value; 0 = optimal */
    return (0 - abs($f));
}

//=====
// Helper methods

/**
 * Generate floating point random number.
 *
 * @param Float Minimum
 * @param Float Maximum
 * @returns Float Random number within limits
 */
function frand($min, $max) {
    $fac = 1000000.0;

    return rand($min * $fac, $max * $fac) / $fac;
}

/**
 * Take care of delta not exceeding bounds.
 * (This restricts 'warping')
 *
 * @param Float Value to constrict
 * @param Float .. constricted value. Wooho
 */
function constrict($delta) {
    global $deltaMin, $deltaMax;

    if ($delta < $deltaMin) return $deltaMin;
    if ($delta > $deltaMax) return $deltaMax;

    return $delta;
}
```

A. An PHP Swarm optimizer

```
/**
 * Comparator for sorting by distance.
 * Just used in distance-based neighborhoods.
 *
 * @private
 */
function _distanceSorter($a, $b) {
    list($adist, $aobj) = $a;
    list($bdist, $bobj) = $b;

    // Sort by distance
    return ($adist < $bdist) ? -1 : 1;
}

//=====
// Our particle class

/**
 * Simple class for basic PSO
 */
class Particle {

    // Dimension dependant
    var $next;
    var $velocity;
    var $current;
    var $best;

    // Best fitness + Neighbors
    var $best_so_far;
    var $neighbor;

    /** Constructor */
    function Particle() {
        global $dimensions, $deltaMin, $deltaMax;
        global $initialFitness, $ssss;

        // Init arrays
```

A. An PHP Swarm optimizer

```
$next      = array ();
$velocity = array ();
$current   = array ();
$best      = array ();
$neighbor  = array ();

// Insert first position + velocity
for ($d = 0; $d < $dimensions; $d++) {
    $this->current[$d] = 0;
    $this->next[$d]     = frand(-$sss,$sss);
    $this->velocity[$d] = frand($deltaMin,$deltaMax);
}

// Predefine a very low fitness
$this->best_so_far = $initialFitness;
}

/**
 * Return the n-th neighbor.
 *
 * @param Integer Index of neighbor
 * @returns Particle Corresponding particle
 */
function getNeighbor($n) {
    $neighborParticle = $this->neighbor[$n];

    return $neighborParticle;
}

/**
 * Return best particle in neighbourhood
 *
 * @returns Particle Neighbour with best fitness
 */
function getNeighborWithBestFitness() {

    // Get neighbors
    $neighbors = $this->neighbor;
    $neighborParticle = NULL;
    $best = $initialFitness;
```

A. An PHP Swarm optimizer

```
// Search best one
foreach ($neighbors as $neighbor) {
    if ($neighbor->best_so_far > $best) {
        $best = $neighbor->best_so_far;
        $neighborParticle = $neighbor;
    }
}

return $neighborParticle;
}

/**
 * Calculate distance between me + the other.
 *
 * @param Particle      Other particle
 * @returns Float       Euclidian distance
 */
function distanceTo(&$otherParticle) {
    global $dimensions;

    // Get other's and my position
    $otherPos = $otherParticle->current;
    $myPos     = $this->current;

    // Calculate vector between particles
    $delta = array();
    for($d = 0; $d < $dimensions; $d++) {
        $delta[$d] = abs($otherPos[$d] - $myPos[$d]);
    }

    // Calculate Euclidian distance
    $temp = 0;
    for($d = 0; $d < $dimensions; $d++) {
        $temp += pow($delta[$d], 2);
    }
    $temp = sqrt($temp);

    return ($temp);
}
```

A. An PHP Swarm optimizer

```
/**
 * Update list of neighbors.
 *
 * Just to be used if working with
 * distance-based neighborships.
 */
function updateNeighbors() {
    global $particles, $numberOfParticles;
    global $numberOfNeighbors;

    $distances = array();

    // Get all distances
    for($i = 0; $i < $numberOfParticles; $i++) {
        $p = &$particles[$i];

        // Calculate distance to other
        $thisdist = $p->distanceTo($this);

        // Probably seen myself, next one!
        if ($thisdist == 0) continue;

        array_push($distances, array($thisdist, $p));
    }

    // Sort array by user-defined sorter
    usort($distances, "_distanceSorter");

    // Pick first k entries of form (distance, particle)
    $temp = array_slice($distances, 0,
                        $numberOfNeighbors);

    // Extract particles + set as neighbors
    $this->neighbor = array();
    for($i = 0; $i < count($temp); $i++) {
        list($dist, $particle) = $temp[$i];

        array_push($this->neighbor, $particle);
    }
}
```

A. An PHP Swarm optimizer

```
}

/**
 * Move particle to next wanted location.
 */
function move() {
    $this->current = $this->next;
}

/**
 * Evaluate fitness at this position.
 * If best fitness so far, memorize value + position
 */
function evalFitness() {

    // Get fitness at this position
    $fitness = test($this);

    // If better than before, memorise
    if ($fitness > $this->best_so_far) {
        $this->best_so_far = $fitness;
        $this->best = $this->current;
    }

    return ($fitness);
}
}

//=====
// Display some data first

printf("<html><head </><body><pre>\n");
printf("<br </><br </>%s<br </>", str_repeat("=", 80));
printf("==== Initialization <br </><br </>", $iterations);
printf(" Particles : %8d<br </>", $numberOfParticles);
printf(" Neighborhood : %8s<br </>",
    ($numberOfParticles > $numberOfNeighbors+1 ?
    "lbest - " . $numberOfNeighbors : "gbest")
);
```

A. An PHP Swarm optimizer

```
printf("# Iterations: %8d<br />", $maxIterations);
printf("Threshold: %4.3f<br />", $threshold);
printf("<br />");

//=====
// Initialization of particles

$particles = array();
for($i = 0; $i < $numberOfParticles; $i++) {

    // Create particle
    array_push($particles, new Particle());
}

// Initialize neighbors (non-distance based neighborhood)
if (! $distanced)
    for($i = 0; $i < $numberOfParticles; $i++) {
        $p = &$particles[$i];

        for ($n = 0; $n < $numberOfNeighbors; $n++) {
            $p->neighbor[$n] = &$particles[(1 + $i + $n)
                % $numberOfParticles];
        }
    }

// Initialize neighbors (distance based neighborhood)
else
    for($i = 0; $i < $numberOfParticles; $i++) {
        $p = &$particles[$i];
        $p->updateNeighbors();
    }

//=====
// PSO

$iterations = 0;
$pbest = NULL;
while ($iterations <= $maxIterations) {
```

A. An PHP Swarm optimizer

```

if ($_GET["debug"]) {
    printf("<br_/>%s<br_/>", str_repeat("=", 80));
    printf("====_Iteration_%%03d<br_/><br_/>",
        $iterations
    );
}

// Move particles and test fitness
for($i = 0; $i < $numberOfParticles; $i++) {
    $p = &$particles[$i];

    // Move particle to next position
    $p->move();

    // Evaluate fitness at new position
    $fitness = $p->evalFitness();

    // Remember best particle so far
    if ($pbest == NULL) $pbest = $p;
    if ($p->best_so_far > $pbest->best_so_far) {
        $pbest = $p;
    }

    // Abort simulation if we exactly hit the target
    if ( abs($fitness - $targetFitness) <= $threshold ) {
        printf("THRESHOLD_MATCH<br_/>");
        printf("Matching_solution_found:_%%6.4f_<br_/>",
            $pbest->best_so_far
        );

        list ($x, $y, $z, $w) = $pbest->best;
        printf("x:_%%3.4f,_y:_%3.4f,_z:_%3.4f,_" .
            "w:_%3.4f<br_/>",
            $x, $y, $z, $w
        );
        printf(" f(%%3.4f,%%3.4f,%%3.4f,%%3.4f) =_%%3.4f" ,
            $x, $y, $z, $w,
            $f = 5*pow($x, 2) + 2*pow($y, 3) -
            2*($z / $w) + 4
        );

        printf("</pre></body></html>");
    }
}

```

A. An PHP Swarm optimizer

```
        exit;
    }
}

// Update neighbors (distance-based neighborhood)
if ($distanced)
    for($i = 0; $i < $numberOfParticles; $i++) {
        $p = &$particles[$i];
        $p->updateNeighbors();
    }

// Adjust the particle's velocities
for($i = 0; $i < $numberOfParticles; $i++) {
    $p = &$particles[$i];

    // Get best neighbor
    $n = $p->getNeighborWithBestFitness();

    for ($d = 0; $d < $dimensions; $d++) {

        // Get the mood of particle
        $iFactor = $iWeight * frand($iMin, $iMax);
        $sFactor = $sWeight * frand($sMin, $sMax);

        // Personal distance between best known/current
        $pDelta[$d] = $p->best[$d] - $p->current[$d];

        // Distance to best neighbor's and own position
        $nDelta[$d] = $n->best[$d] - $p->current[$d];

        // Combine these two and adjust
        $delta = ($iFactor * $pDelta[$d]) +
            ($sFactor * $nDelta[$d]);
        $delta = $p->velocity[$d] + $delta;

        // Keep it within limits, no warping particles
        $p->velocity[$d] = constrict($delta);
    }
}
}
```

A. An PHP Swarm optimizer

```

        // Adjust next position
        $p->next[$d] = $p->current[$d] +
                    $p->velocity[$d];
    }

    if ($_GET["debug"]) {
        printf("p = (%3.4f,%3.4f,%3.4f,%3.4f) \n",
            $p->current[0], $p->current[1],
            $p->current[2], $p->current[3]
        );
        printf("v = (%3.4f,%3.4f,%3.4f,%3.4f) \n",
            $p->velocity[0], $p->velocity[1],
            $p->velocity[2], $p->velocity[3]
        );
        printf("f = (%6.4f) \n", $p->evalFitness());
        printf("<br />");
    }
}

// Give our debug output
list($x,$y,$z,$w) = $pbest->best;
printf("it = %6d \n fbest = %6.4f \n pbest = "
    "(%3.4f,%3.4f,%3.4f,%3.4f)<br />"
    , $iterations, $pbest->best_so_far,
    $x, $y, $z, $w
);
$iterations++;
}

//=====
// Output of this run

printf("Best solution found: %6.4f<br />"
    , $pbest->best_so_far
);

list($x,$y,$z,$w) = $pbest->best;
printf("x: %3.4f , y: %3.4f , z: %3.4f , w: %3.4f<br />"
    , $x, $y, $z, $w
);

```

A. An PHP Swarm optimizer

```
printf(" f(%3.4f,%3.4f,%3.4f,%3.4f) = %3.4f" ,  
    $x, $y, $z, $w,  
    $f = 5*pow($x, 2) + 2*pow($y, 3)  
    - 2*($z / $w) + 4  
);  
  
printf("</pre></body></html>" );  
?>
```

Bibliography

- [1] J.Kennedy, R.Eberhart: Swarm Intelligence; Morgan Kaufmann Publishers; 2001
- [2] D.Jackson/CBC: Angles guide ants back to home; 2004; <http://www.cbc.ca/story/science/national/2004/12/15/ants-angles041215.html>
- [3] J.McQuaid: A silent invader bursts into view; 1998; <http://www.nola.com/speced/homewreckers/day1mainstory1.html>
- [4] Unknown: Termite towers; http://www.exchangedlife.com/Creation/termite_towers.htm
- [5] P.P.Grasse; La Reconstruction du Nid et les Coordinations Interindividuelles Chez Bellicositermes Natalensis et Cubitermes SP. La Theorie de la Stigmergie: Essai D'interpretation du Comportement des Termites Constructeurs; Insectes Sociaux, Tome V1, No1; 1959
- [6] C.W.Reynolds; Flocks, herds and schools: A distributed behavioral model; Computer Graphics, Volume 21, Number 4, July 1987;
- [7] C.W.Reynoldy; Homepage on Boids; <http://www.red3d.com/cwr/boids/>
- [8] F.Heppner, U.Grenander; A stochastic nonlinear model for coordinated bird flocks; The Ubiquity of Chaos, AAAS, Washington; 1990
- [9] J.Hoffmeyer; The Swarming body; in: Semiotics Around the World; Proceedings of the Fifth Congress of the International Association for Semiotic Studies; Berkley; 1994
- [10] J.Kennedy, R.Eberhard; Swarm Intelligence; Morgan Kaufmann; 2001

Index

- agent, 7, 9, 13
- ants, 3–4
- attitude, 10
- behavioral animation, 6–7
- Boids, 6
- CGI, 7
- convergence, 13
- crossover, 13
- diversity, 13
- Eberhart, Russel, 8, 9
- evolutionary algorithm, 2, 8, 11, 13
 - meta EA, 14
- exploration / exploitation, 9, 10, 12, 13
- genetic algorithm, 13
- genotype, 13
- Heppner, Frank, 7, 9
- Hoffmeyer, Jesper, 7
- individual, 13
- Kennedy, James, 8, 9
- memes, 13
- mutation, 13
- neighbourhood, 10–13
 - circle topology, 11
 - global, 10, 11
 - local, 10, 11
 - topology, 11
- wheel topology, 11
- optimum
 - global, 9
 - local, 9
- particle
 - desire, 7, 9
 - fitness, 9, 10, 12
 - position, 9
 - velocity, 9–10
- Reynolds, Craig, 6, 7, 9
- roosting area, 7, 9
- selection, 13
- sematectonic communication, 5
- semiotics, 7
- SigGraph, 6
- solution space, 9
 - multi modal, 10
- stigmergy, 5
- subjective norm, 10
- swarm intelligence, 2–5
- termites, 4–5