Optimize Resource Scheduling in Cloud using PSO

Data Structure and Algorithms Final Project

Astha Shah
Nishad Ranadive
Ankita Mhatre

Introduction

Particle Swarm Optimization is a population based optimization technique inspired by social behavior of bird flocking. It is a computational method that optimizes a problem iteratively by trying to improve a solution with regard to a given measure of quality which is treated as the target. It solves the problem by having a population of candidate solutions i.e the particles which are guided towards better positions found by other particles. This behavior moves the entire swarm towards the best solution for the problem. In PSO, over a number of iterations, a group of variables have their values adjusted closer to the member whose value is closest to the target at any moment.

PSO has become popular due to its simplicity and its effectiveness in a wide range of applications with low computational cost.

In Cloud Computing, resources are dynamically allocated to facilitate applications. Users are charged as per use i.e the time required to execute their application on cloud. Cloud computing is an emerging technology used by many companies for computing tasks if large data.

Problem Statement

As stated above in cloud computing, the distribution of computing tasks in cloud environment is from the resource pool where users get these resources on demand. However what makes users more concerned is the cost of using these resources. User applications may incur large data retrieval and execution costs when they are scheduled taking into account only the ‘execution time’. In addition to optimizing execution time, the cost arising from data transfers between resources as well as execution costs must also be taken into account. Hence to minimize the computation cost of tasks in Cloud computing we will use Particle Swarm Optimization technique.

Proposed Solution

We present a particle swarm optimized approach to schedule applications to cloud resources that takes into account both computation cost and data transmission cost. We experiment with a workflow application by varying its computation and communication costs. We compare the cost savings when using PSO. Our results show that PSO can achieve: a) optimum cost , and b) good distribution of workload onto resources.

The mapping of tasks of an application workflow to distributed resources can have several objectives. We focus on minimizing the total cost of computation of an application workflow.

Steps:

1: Set particle dimension as equal to the size of tasks in {ti} ∈ T

2: Initialize particles position randomly from C = 1, ..., j and velocity vi randomly.

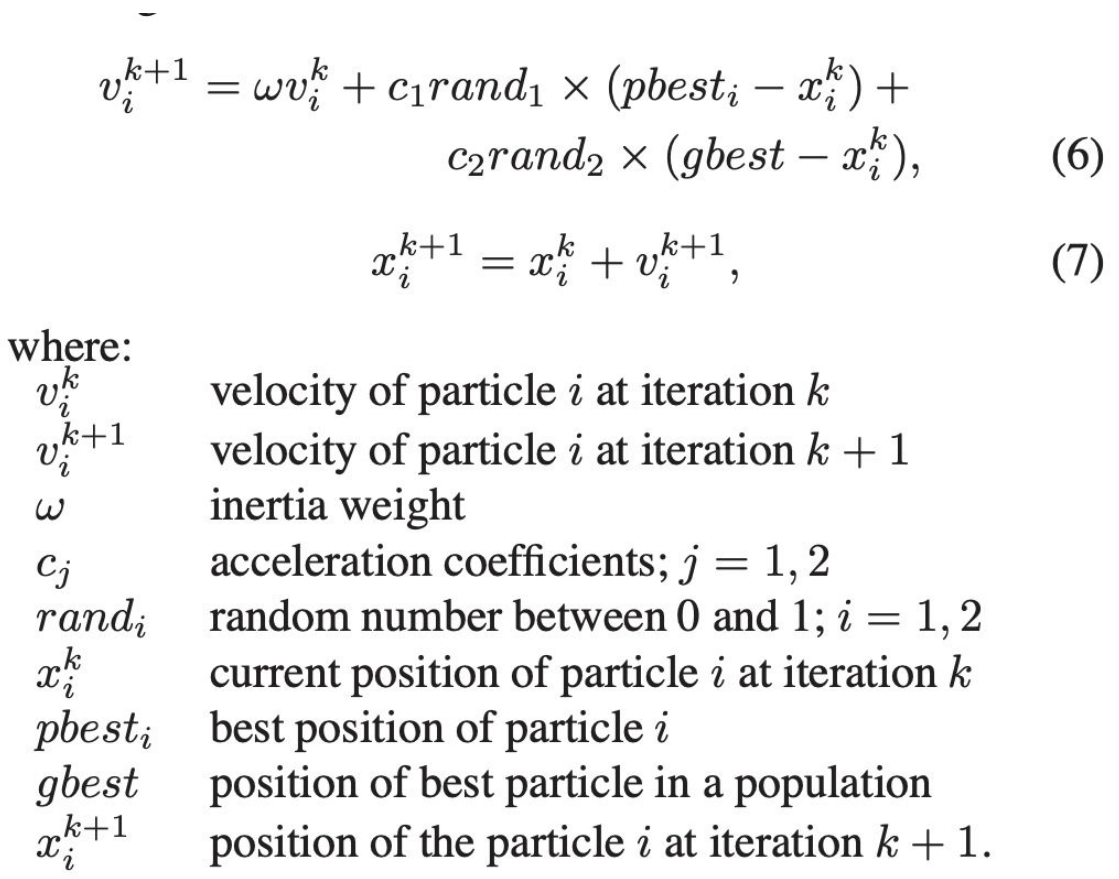
3: For each particle, calculate its fitness value as in Equation 4.

4: If the fitness value is better than the previous best pbest, set the current fitness value as the new pbest.

5: After Steps 3 and 4 for all particles, select the best particle as gbest.

6: For all particles, calculate velocity using Equation 6 and update their positions using Equation 7.

7: If the stopping criteria or maximum iteration is not satisfied, repeat from Step 3.



$$v_i^{k+1} = \omega v_i^k + c_1 rand_1 \times (pbest_i - x_i^k) + c_2 rand_2 \times (gbest - x_i^k), \qquad (6)$$

$$x_i^{k+1} = x_i^k + v_i^{k+1}, \qquad (7)$$

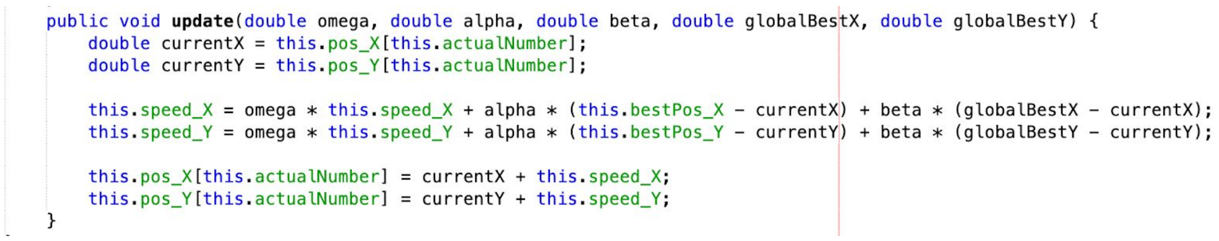
where:

v_i^k	velocity of particle i at iteration k
v_i^{k+1}	velocity of particle i at iteration $k+1$
ω	inertia weight
c_j	acceleration coefficients; $j = 1, 2$
$rand_i$	random number between 0 and 1; $i = 1, 2$
x_i^k	current position of particle i at iteration k
$pbest_i$	best position of particle i
$gbest$	position of best particle in a population
x_i^{k+1}	position of the particle i at iteration $k+1$.

We calculate total execution cost by adding the cost of execution and cost of transfer where cost of transfer is calculated using total transfers between resources and price of each transfer. Using PSO we find the optimal total cost.

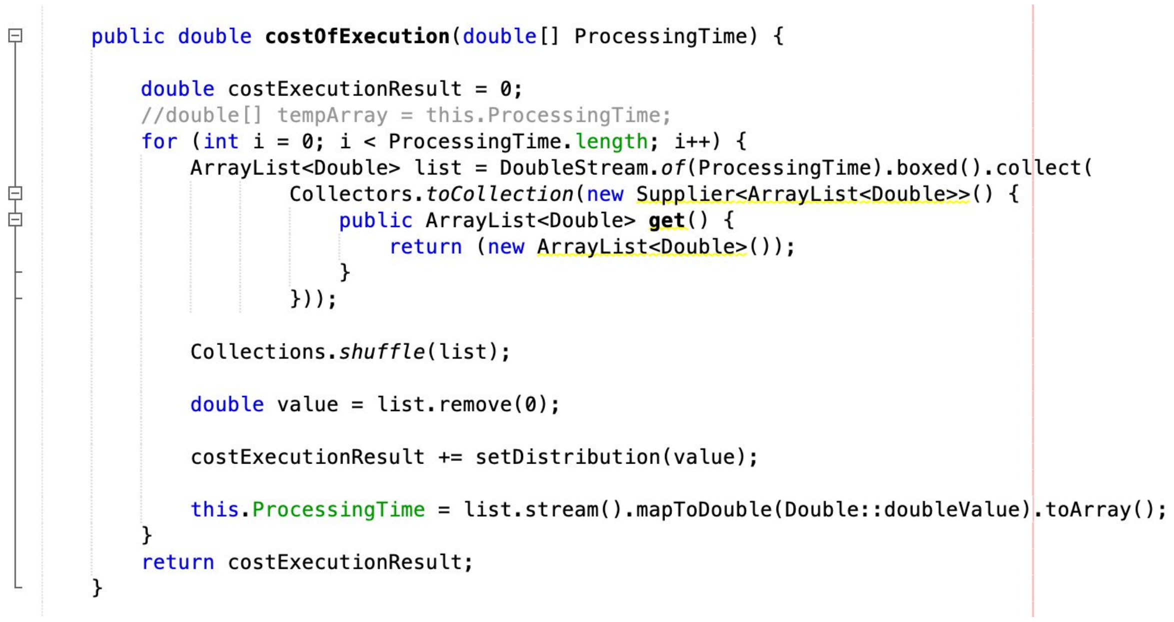
Some important functions

PSO Function



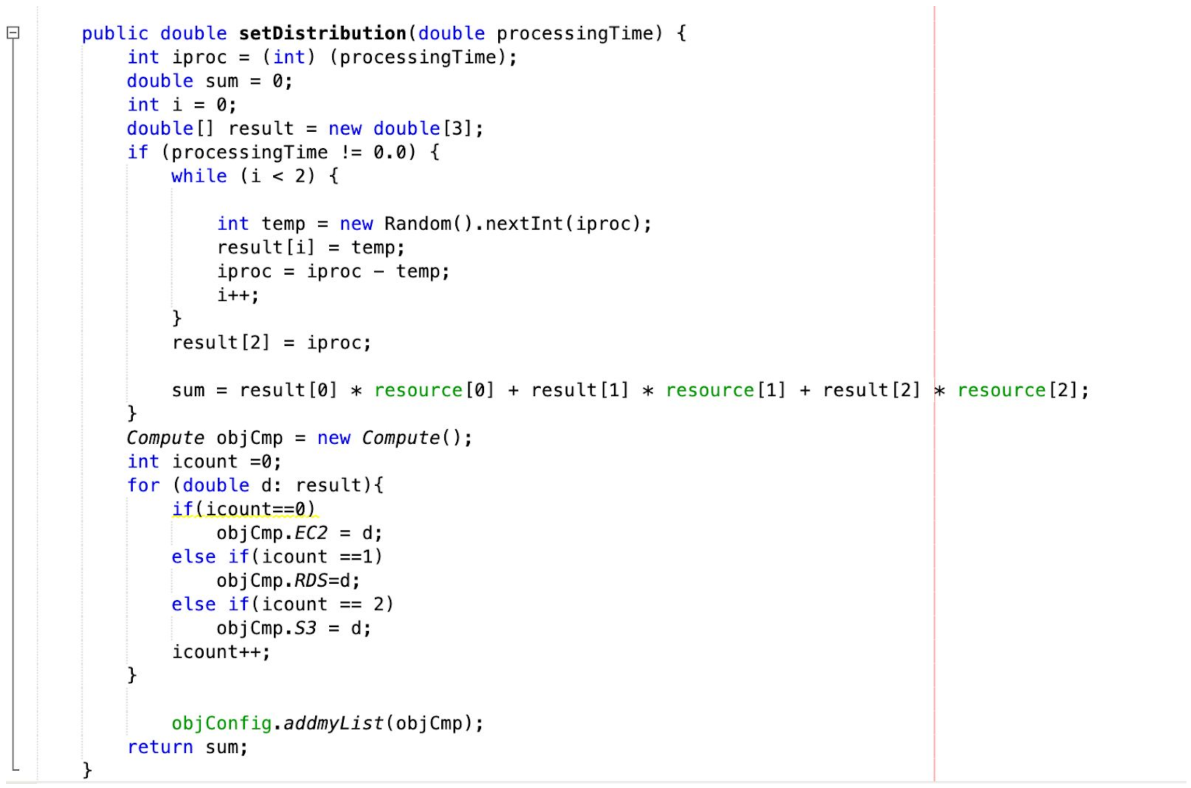
```java
public void update(double omega, double alpha, double beta, double globalBestX, double globalBestY) {
    double currentX = this.pos_X[this.actualNumber];
    double currentY = this.pos_Y[this.actualNumber];

    this.speed_X = omega * this.speed_X + alpha * (this.bestPos_X - currentX) + beta * (globalBestX - currentX);
    this.speed_Y = omega * this.speed_Y + alpha * (this.bestPos_Y - currentY) + beta * (globalBestY - currentY);

    this.pos_X[this.actualNumber] = currentX + this.speed_X;
    this.pos_Y[this.actualNumber] = currentY + this.speed_Y;
}
```

Cost of Execution



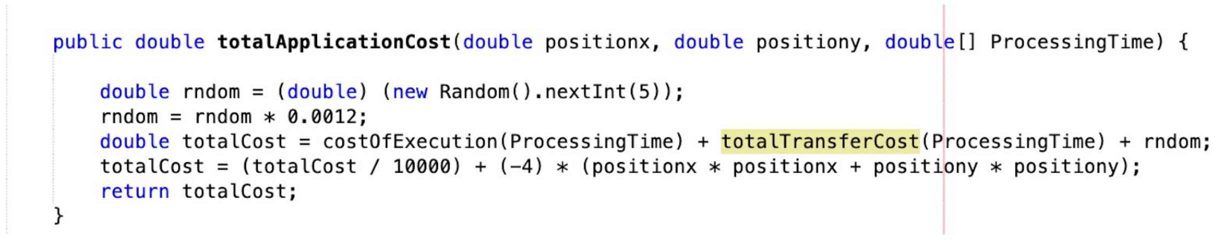
```java
public double costOfExecution(double[] ProcessingTime) {

    double costExecutionResult = 0;
    //double[] tempArray = this.ProcessingTime;
    for (int i = 0; i < ProcessingTime.length; i++) {
        ArrayList<Double> list = DoubleStream.of(ProcessingTime).boxed().collect(
                Collectors.toCollection(new Supplier<ArrayList<Double>>() {
                    public ArrayList<Double> get() {
                        return (new ArrayList<Double>());
                    }
                }));

        Collections.shuffle(list);

        double value = list.remove(0);

        costExecutionResult += setDistribution(value);

        this.ProcessingTime = list.stream().mapToDouble(Double::doubleValue).toArray();
    }
    return costExecutionResult;
}
```

Set Distribution



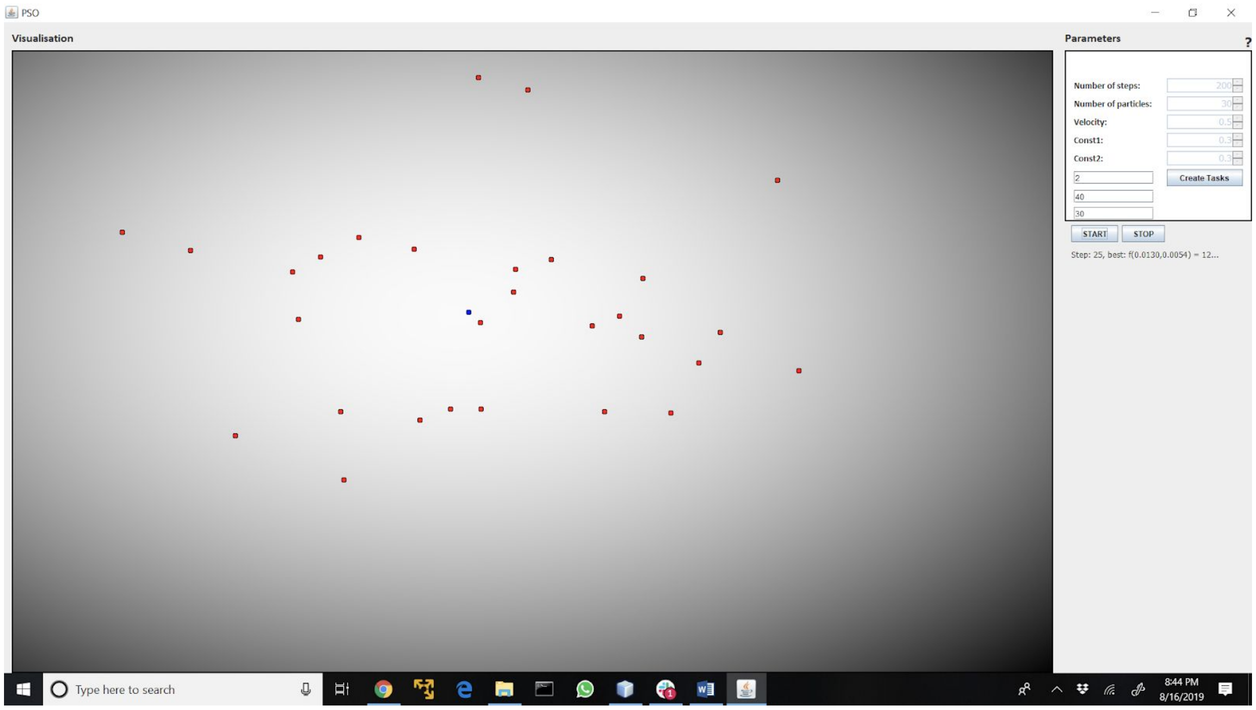
```java
public double setDistribution(double processingTime) {
    int iproc = (int) (processingTime);
    double sum = 0;
    int i = 0;
    double[] result = new double[3];
    if (processingTime != 0.0) {
        while (i < 2) {

            int temp = new Random().nextInt(iproc);
            result[i] = temp;
            iproc = iproc - temp;
            i++;
        }
        result[2] = iproc;

        sum = result[0] * resource[0] + result[1] * resource[1] + result[2] * resource[2];
    }
    Compute objCmp = new Compute();
    int icount =0;
    for (double d: result){
        if(icount==0)
            objCmp.EC2 = d;
        else if(icount ==1)
            objCmp.RDS=d;
        else if(icount == 2)
            objCmp.S3 = d;
        icount++;
    }

        objConfig.addmyList(objCmp);
    return sum;
}
```

TotalApplication Cost



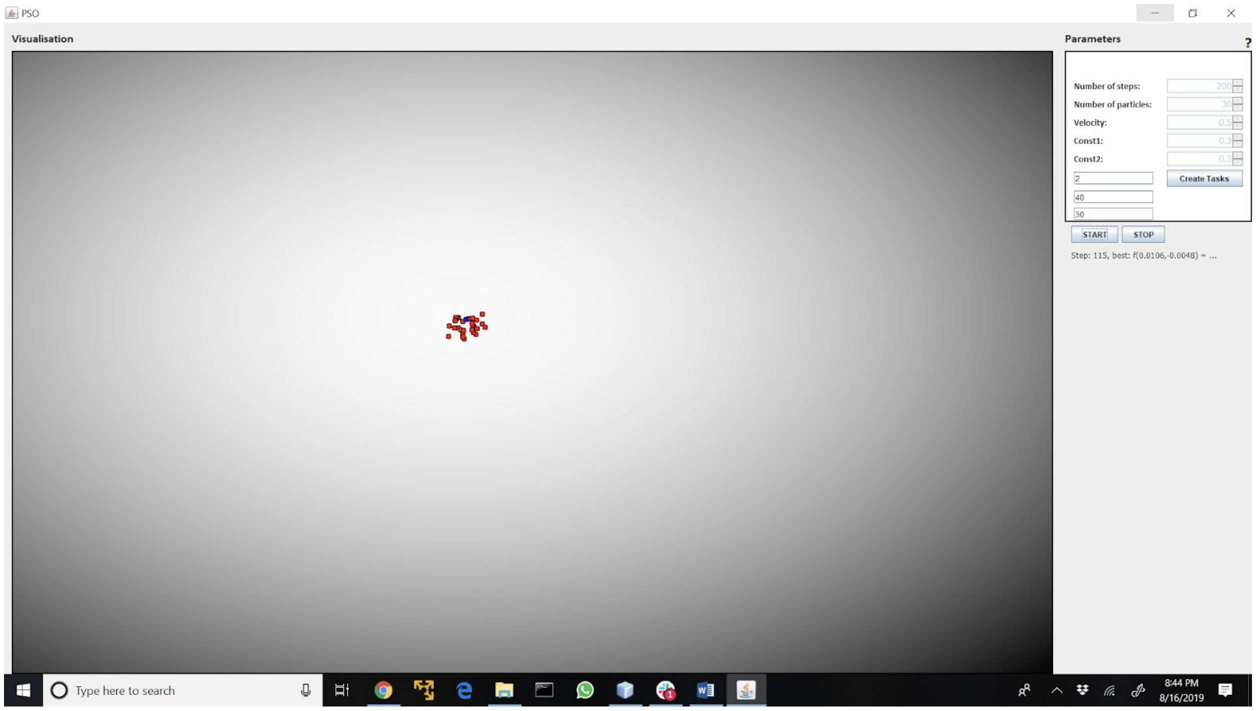
```java
public double totalApplicationCost(double positionx, double positiony, double[] ProcessingTime) {

    double rndom = (double) (new Random().nextInt(5));
    rndom = rndom * 0.0012;
    double totalCost = costOfExecution(ProcessingTime) + totalTransferCost(ProcessingTime) + rndom;
    totalCost = (totalCost / 10000) + (-4) * (positionx * positionx + positiony * positiony);
    return totalCost;
}
```

Output Console:



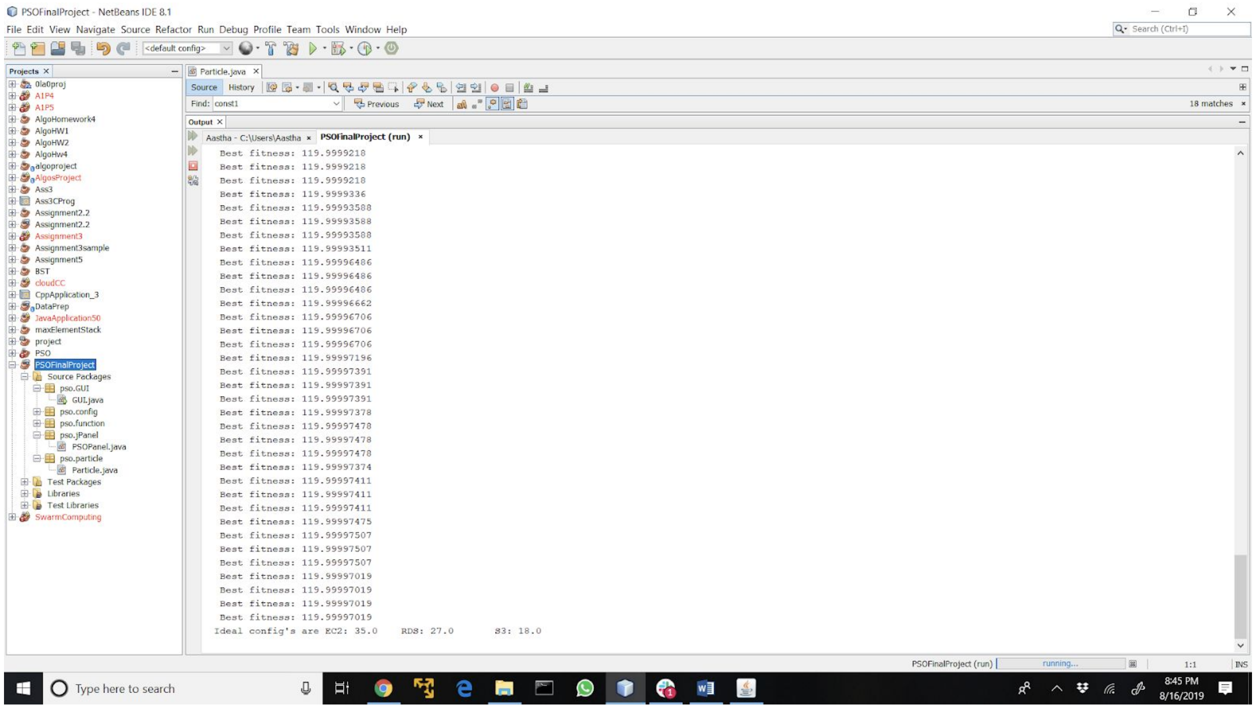
Number of steps: 200
Number of particles: 30
Velocity: 0.5
Const1: 0.3
Const2: 0.3
2
40
30

Step: 25, best: f(0.0130,0.0054) = 12...



Number of steps: 200
Number of particles: 30
Velocity: 0.5
Const1: 0.3
Const2: 0.3
2
40
30

Step: 115, best: f(0.0106,-0.0048) = ...



```
 Best fitness: 119.9999218
 Best fitness: 119.9999218
 Best fitness: 119.9999218
 Best fitness: 119.9999336
 Best fitness: 119.99993588
 Best fitness: 119.99993588
 Best fitness: 119.99993588
 Best fitness: 119.99993511
 Best fitness: 119.99996486
 Best fitness: 119.99996486
 Best fitness: 119.99996486
 Best fitness: 119.99996662
 Best fitness: 119.99996706
 Best fitness: 119.99996706
 Best fitness: 119.99996706
 Best fitness: 119.99997196
 Best fitness: 119.99997391
 Best fitness: 119.99997391
 Best fitness: 119.99997391
 Best fitness: 119.99997378
 Best fitness: 119.99997478
 Best fitness: 119.99997478
 Best fitness: 119.99997478
 Best fitness: 119.99997374
 Best fitness: 119.99997411
 Best fitness: 119.99997411
 Best fitness: 119.99997411
 Best fitness: 119.99997475
 Best fitness: 119.99997507
 Best fitness: 119.99997507
 Best fitness: 119.99997507
 Best fitness: 119.99997019
 Best fitness: 119.99997019
 Best fitness: 119.99997019
 Best fitness: 119.99997019
Ideal config's are EC2: 35.0    RDS: 27.0       S3: 18.0
```