

|  |  |
| --- | --- |
|  | DESIGN AND ANALYSIS OF ALGORITHM |
|  |  |
|  | PAPER CODE: CS 591 |

MEMBER DETAILS

|  |  |
| --- | --- |
| **MD. Yusuf Idrishi** | **3312016009001685** |
| **Anupam Shankar Dey** | **3312016009001083** |

INTRODUCTION

Inspired by the flocking and schooling patterns of birds and fish, Particle Swarm Optimization (PSO) was invented by Russell Eberhart and James Kennedy in 1995. Originally, these two started out developing computer software simulations of birds flocking around food sources, and then later realized how well their algorithms worked on optimization problems.

Particle Swarm Optimization might sound complicated, but it's really a very simple algorithm. 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 given moment. Imagine a flock of birds circling over an area where they can smell a hidden source of food. The one who is closest to the food chirps the loudest and the other birds swing around in his direction. If any of the other circling birds comes closer to the target than the first, it chirps louder and the others veer over toward him. This tightening pattern continues until one of the birds happens upon the food. It's an algorithm that's simple and easy to implement.

The algorithm keeps track of three global variables:

* **Target value or condition**
* **Global best (gBest) value indicating which particle's data is currently closest to the Target**
* **Stopping value indicating when the algorithm should stop if the Target isn't found**

Each particle consists of:

* **Data representing a possible solution**
* **A Velocity value indicating how much the Data can be changed**
* **A personal best (pBest) value indicating the closest the particle's Data has ever come to the Target**

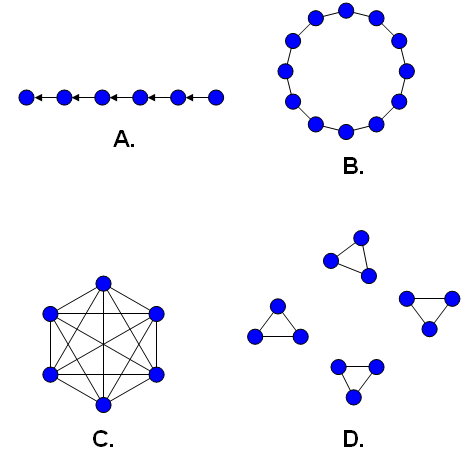
The particles' data could be anything. In the flocking birds’ example above, the data would be the X, Y, Z coordinates of each bird. The individual coordinates of each bird would try to move closer to the coordinates of the bird which is closer to the food's coordinates (gBest). If the data is a pattern or sequence, then individual pieces of the data would be manipulated until the pattern matches the target pattern.

The velocity value is calculated according to how far an individual's data is from the target. The further it is, the larger the velocity value. In the birds example, the individuals furthest from the food would make an effort to keep up with the others by flying faster toward the gBest bird. If the data is a pattern or sequence, the velocity would describe how different the pattern is from the target, and thus, how much it needs to be changed to match the target.

Each particle's pBest value only indicates the closest the data has ever come to the target since the algorithm started.

The gBest value only changes when any particle's pBest value comes closer to the target than gBest. Through each iteration of the algorithm, gBest gradually moves closer and closer to the target until one of the particles reaches the target.

It's also common to see PSO algorithms using population topologies, or "neighborhoods", which can be smaller, localized subsets of the global best value. These neighborhoods can involve two or more particles which are predetermined to act together, or subsets of the search space that particles happen into during testing. The use of neighborhoods often helps the algorithm to avoid getting stuck in local minima.



**Figure 1.** A few common population topologies (neighborhoods).

**(A)** Single-sighted, where individuals only compare themselves to the next best.

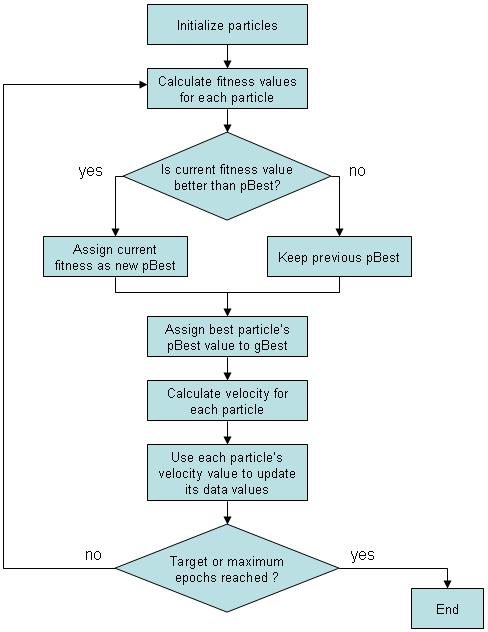
**(B)** Ring topology, where each individual compares only to those to the left and right.

**(C)** Fully connected topology, where everyone is compared together.

**(D)** Isolated, where individuals only compare to those within specified groups.

Neighborhood definitions and how they're used have different effects on the behavior of the algorithm.

FLOWCHART



**Figure 2.** Flow diagram illustrating the particle swarm optimization algorithm.

PSEUDO CODE



SOURCE CODE

import java.util.\*;

import java.io.\*;

import java.util.Map.Entry;

import java.util.Collection;

import java.io.BufferedWriter;

import java.io.IOException;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileWriter;

class PSO\_java{

// Function to be optimized:

public static double ProblemFunc(double[] buffer){

double val = 0;

for(int i=0;i<buffer.length;i++) {

val = val + Math.pow(buffer[i], 2);

}

return val;

}

public static void main(String[] args) {

double INF = Double.NEGATIVE\_INFINITY;

double maxVal = INF;

double weight = 0.8, r1 = 0.0 , r2 = 0.0; // Inertia weight

double p1 = 2.06, p2 = 2.06;

double posMin = -5.05, posMax = 5.05;

double velocityMin = 0 , velocityMax = 1;

double weightMin = 0.3 , weightMax = 0.8;

double phiVal = p1 + p2;

double chiVal = 2.0/Math.abs(2.0-phiVal-Math.sqrt(Math.pow(phiVal, 2)-4\*phiVal));

String[] str = new String[100000];

int lenn = 0;

//int no\_p = 100, no\_s = 1000 , no\_d = 2;

//Input taken from user

Scanner sc = new Scanner(System.in);

System.out.printf("\nEnter Number of Particle: ");

int no\_p = sc.nextInt();

System.out.printf("\nEnter Number of Timesteps: ");

int no\_s = sc.nextInt();

System.out.printf("\nEnter Number of Dimensions: ");

int no\_d = sc.nextInt();

Random rd = new Random();

double[] pOptimumVal = new double[no\_p];

double[] gOptimumPrevious = new double[no\_s]; // Previous fitness stats

double[] gOptimumPos = new double[no\_d]; // Best particle position

double[] M = new double[no\_p];

double[][] pOptimumPos = new double[no\_p][no\_d]; // Individual Particle position

double[][] R = new double[no\_p][no\_d];

double[][] V = new double[no\_p][no\_d];

// Initialisation

for(int i=0; i<no\_p; i++) {

pOptimumVal[i] = INF;

}

// Populate particle velocity table with random value between (0 - (max-min+1)) range

for(int i=0; i<no\_p; i++){

for(int j=0; j<no\_d; j++){

R[i][j] = posMin + (posMax-posMin)\*rd.nextDouble();

V[i][j] = velocityMin + (velocityMax-velocityMin)\*rd.nextDouble();

// Global best error checkpoint

if(rd.nextDouble() < 0.5){

V[i][j] = -V[i][j];

R[i][j] = -R[i][j];

}

}

}

for(int i=0; i<no\_p; i++){

M[i] = ProblemFunc(R[i]);

M[i] = -M[i];

}

// Updation and finding optimum position

for(int j=0; j<no\_s; j++){

for(int p=0; p<no\_p;p++){

for(int i=0; i<no\_d; i++){

R[p][i] = R[p][i] + V[p][i];

If(R[p][i] > posMax) { R[p][i] = posMax;}

else if(R[p][i] < posMin) { R[p][i] = posMin;}

}

}

// Iterating through all particle and calculating the fitness stats

for(int p=0; p<no\_p; p++){

M[p] = ProblemFunc(R[p]);

M[p] = -M[p];

if(M[p] > pOptimumVal[p]){

pOptimumVal[p] = M[p];

for(int i=0; i<no\_d; i++){

pOptimumPos[p][i] = R[p][i];

}

}

if(M[p] > maxVal){

maxVal = M[p];

for(int i=0; i<no\_d; i++){

gOptimumPos[i] = R[p][i];

}

}

}

// Finally, (Previous) Current fitness stats is set.

gOptimumPrevious[j] = maxVal;

weight = weightMax - ((weightMax-weightMin)/no\_s) \* j;

// Updation of velocity : Cognitive and social constant

for(int p=0; p<no\_p; p++){

for(int i=0; i<no\_d; i++){

r1 = rd.nextDouble();

r2 = rd.nextDouble();

// Formula : v(next)=w()\*v(curr)+p1\*r1(pos(individual particle pos)−x(particle position))+p2\*r2(pos(best opt pos)−x(particle position))

V[p][i] = chiVal \* weight \* (V[p][i] + r1 \* p1 \* (pOptimumPos[p][i] - R[p][i]) + r2\*p2 \*(gOptimumPos[i] - R[p][i]));

if (V[p][i] > velocityMax) { V[p][i] = velocityMax; }

else if (V[p][i] < velocityMin) { V[p][i] = velocityMin; }

}

}

// Store global optimum for every step

str[lenn++] = "At i:" + j + " Global Optimum Value " + maxVal;

}

//TO PRINT THE OUTPUT INTO THE SYSTEM

/\*for(int i = 0; i < lenn; i++)

System.out.println(str[i]);\*/

//TO PRINT THE OUTPUT INTO THE TEXTFILE

File file = new File ("Output\_File.txt");

try

{

PrintWriter printWriter = new PrintWriter(file);

printWriter.println ("INPUT:\n");

printWriter.println ("Enter Number of Particle: " + no\_p + "\n");

printWriter.println ("Enter Number of Timesteps: " + no\_s + "\n");

printWriter.println ("Enter Number of Dimensions: " + no\_d + "\n");

printWriter.println ("OUTPUT:\n");

for(int i = 0; i < lenn; i++)

printWriter.println(str[i]);

printWriter.close();

}

catch (FileNotFoundException ex)

{

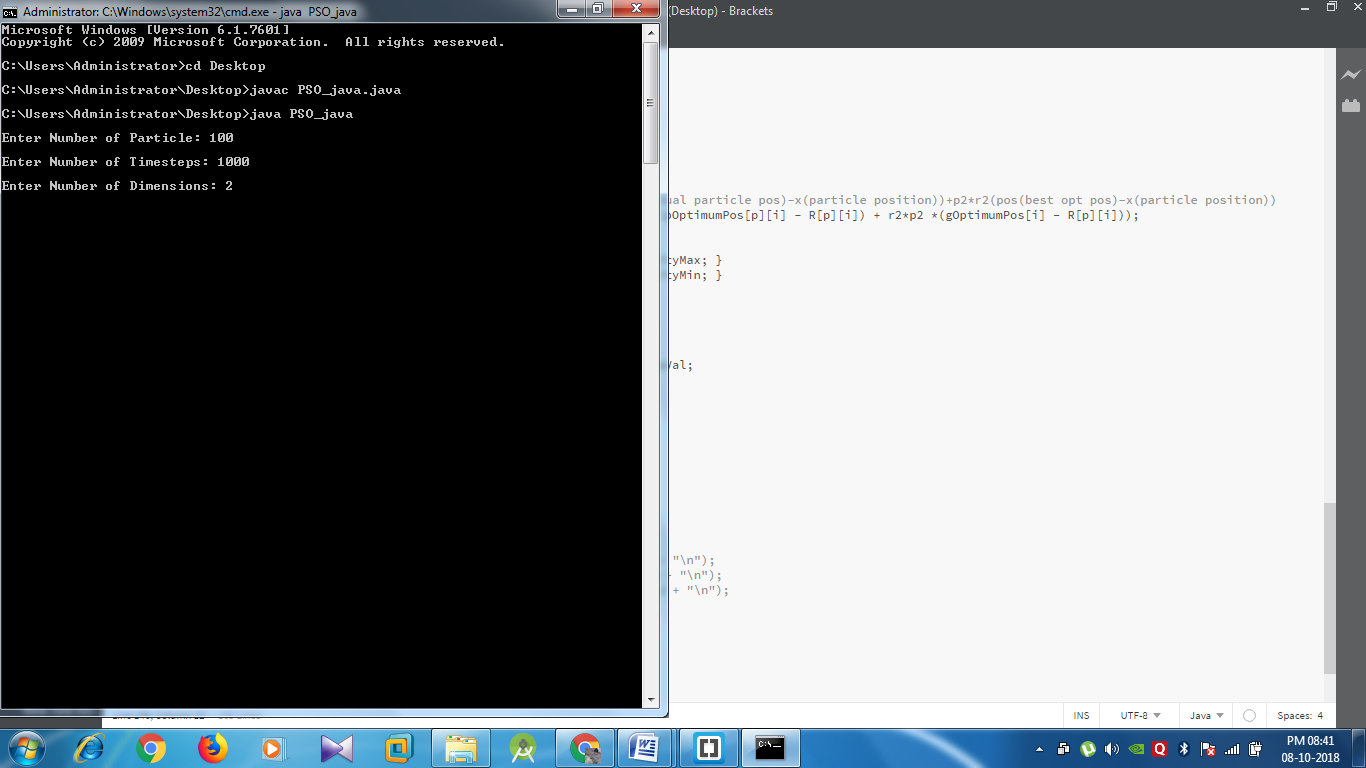
System.err.print ("Something went wrong");

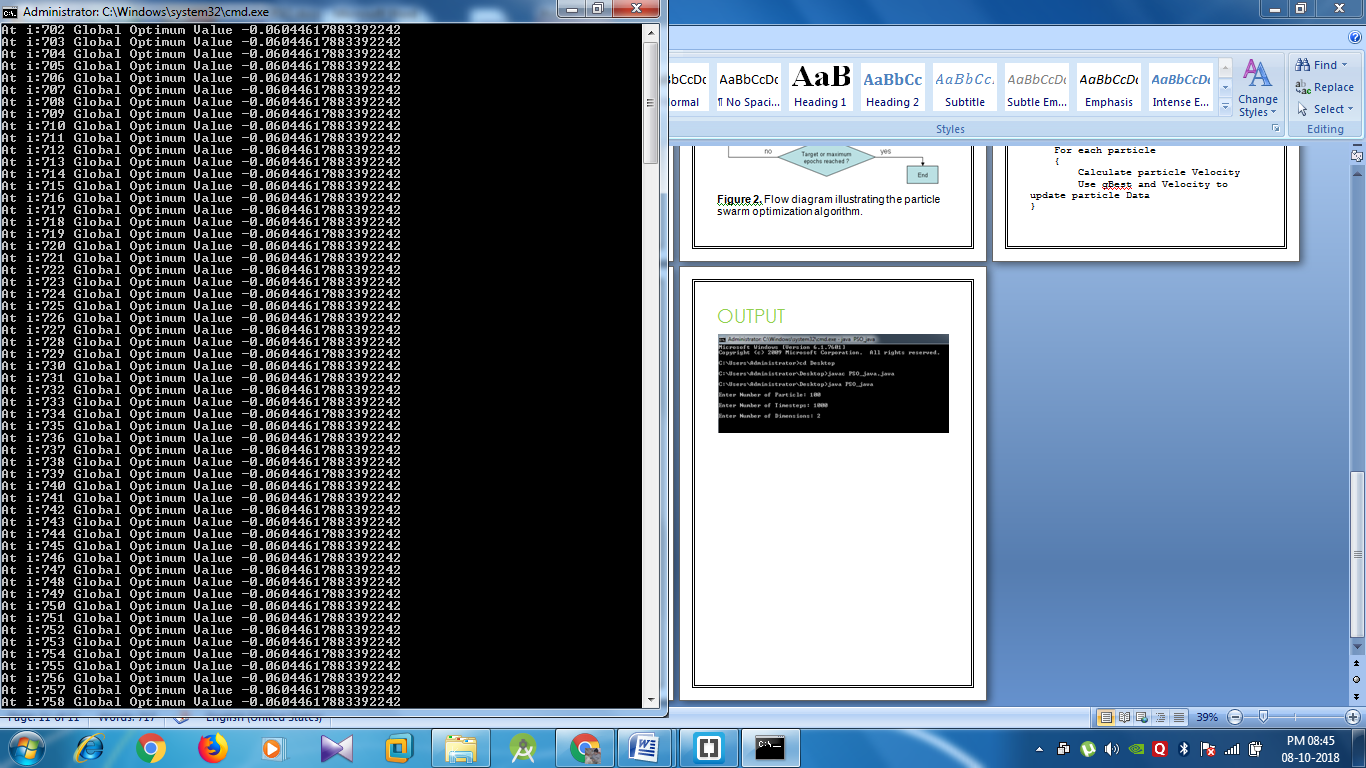
}

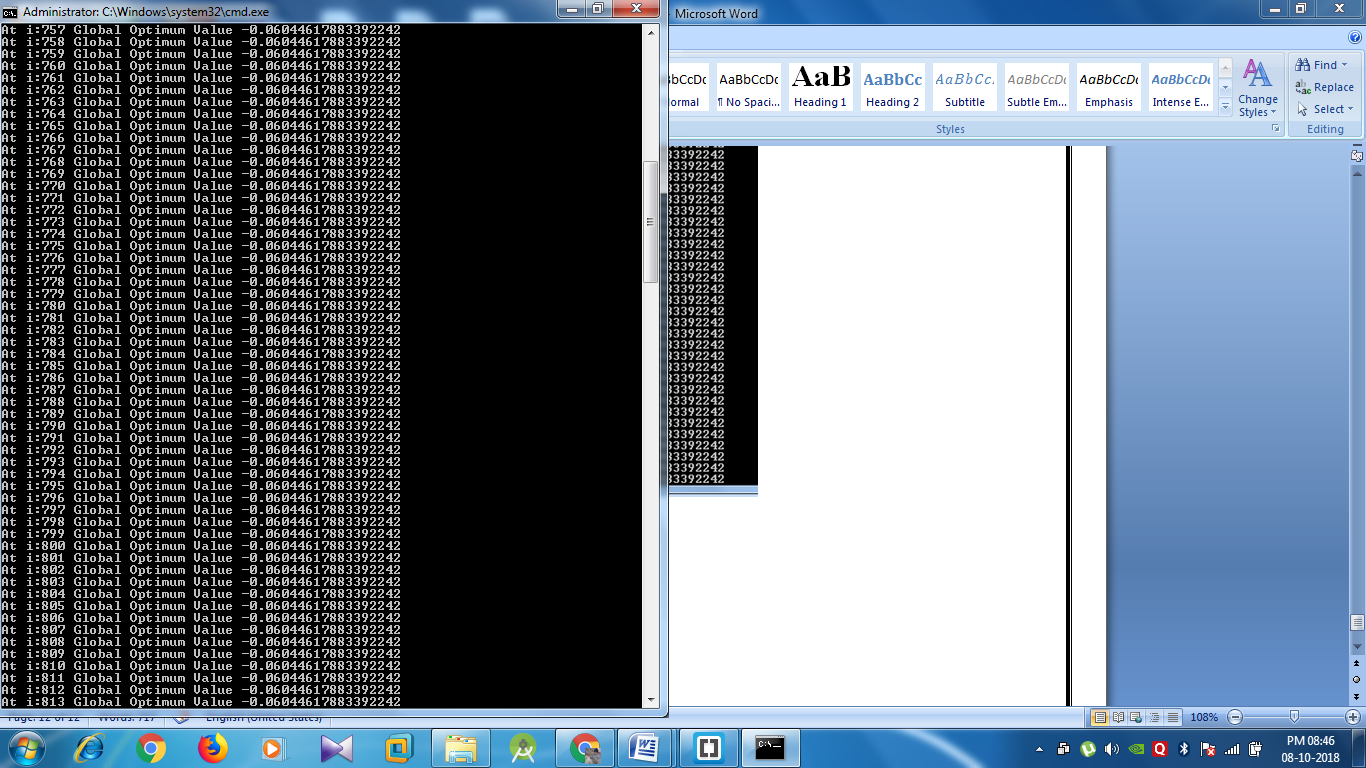
}

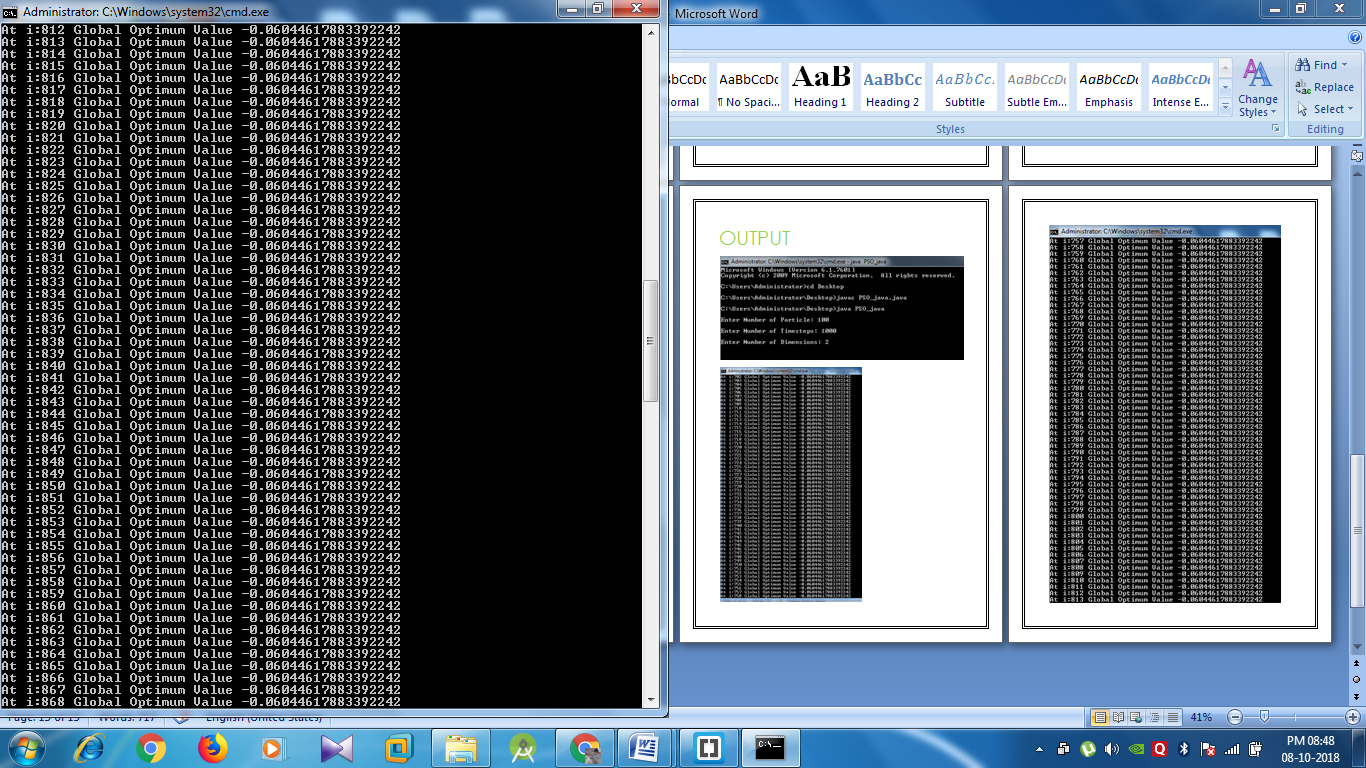
}

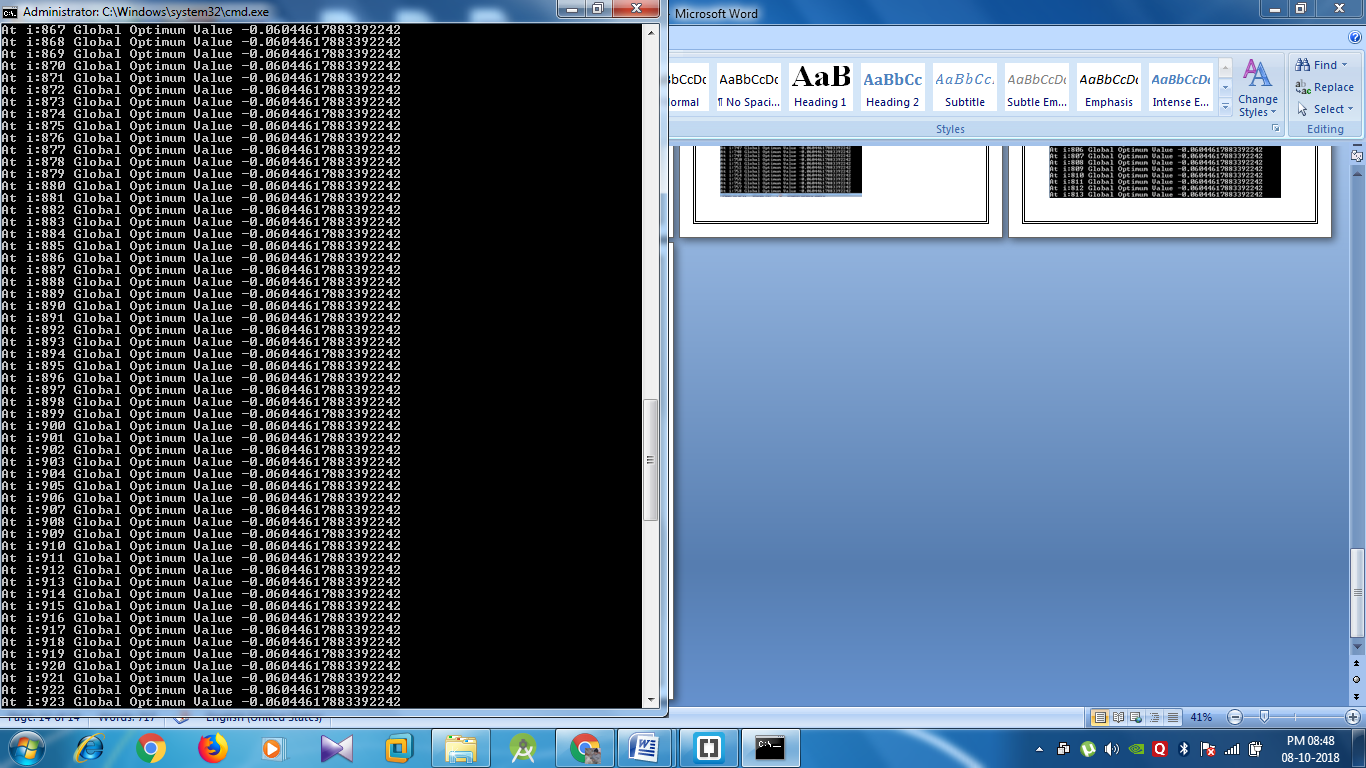
OUTPUT

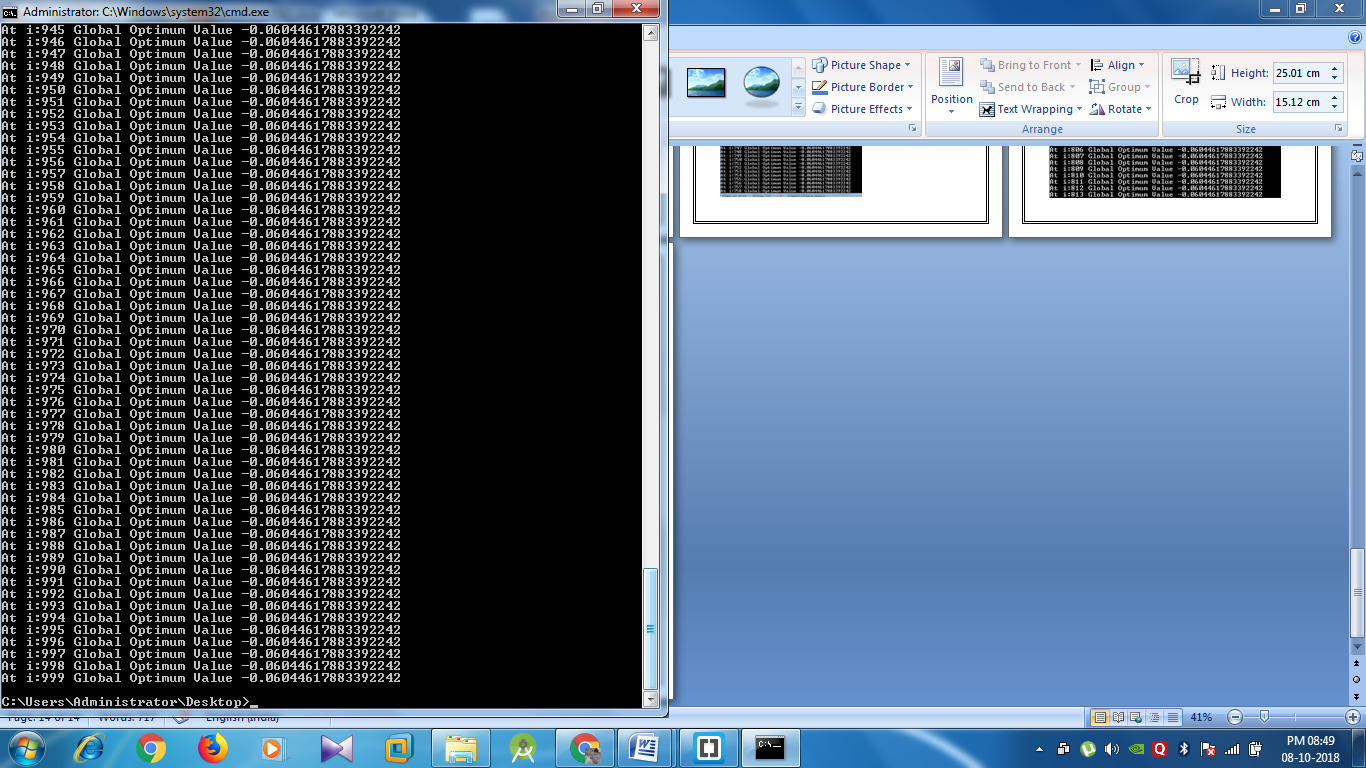
****











APPLICATION

In order to improve the pressure sensor's current stability and temperature drift performance, a soft sensor regression model was modeled based on **least square support vector machine (LS-SVM).** According to the difficulty in selecting penalty factor and kernel parameter which are called hyper-parameters in LS-SVM when modeling, **particle swarm optimization (PSO) algorithm** and **ergodicity search algorithm (ESA)** are proposed to optimize it.

Then a soft sensor model would be reconstructed according to the optimal hyper-parameters. The experiment results show that the time of optimizing the hyper-parameters by PSO (about 1676 second) is decreased to that of ESA (about 211884 second), and **mean squared error (MSE)** of the prediction model with optimal parameters got by **PSO** **(about 1.25times10-6)** is reduced to one sixth of that by **ESA (about 8.35times10-6)**.

So PSO algorithm has more superior performance on global optimization and convergence speed. The optimal soft pressure sensor model got by PSO has more superior current stability and temperature drift performance, and the disadvantage influence of the change of working current and circumstance temperature on the sensor is decreased greatly. This method is feasible that combining **PSO** algorithm with **LS-SVM** in soft sensor modeling. It has definite development space and practical application value.