Optimization for Mathematical Ecology

1 Summary

This report serves to provide an elegant single non-linear mathematical model that follows the small island effect that is widely discussed in mathematical ecology. The Rydin 1988 data set was being used as the primary data points for our curve fitting of our model. Our general model is : $Y = a(log(b + e^{cX-d}))$. We curve fit the model onto the data set using a PSO MATLAB Algorithm designed by S. Mostapha Kalami Heris[2]. We then conducted minor modifications and reconfigured the code such that it can optimize the parameter values of a, b, c, d with the least squares error. The optimized curve is $Y = 7.9506(log(1.8389 + e^{(2.0076X-6.3317)}))$ and has a least squares error of ≈ 3710.307. We also provided a possible explanation of the small island effect based on climate and biodiversity.

2 Data Set

We extracted the Rydin 1988 Data Set and used Excel to convert values of Area from km^2 to m^2. Then we take the natural logarithm of the Area and imported it into MATLAB via (*.csv) file with commands `fileopen` and `textscan`. (The data in Excel can be found in Appendix A). Our data set based on Rydin 1988 was used to observe the scatter plot in order to conjecture a model to fit the data set itself. The scatter plot of 37 data points are plotted using our function `importCSV2.m` in MATLAB which uses the command `scatter` (The entire function code can be found in Appendix B) as shown below:

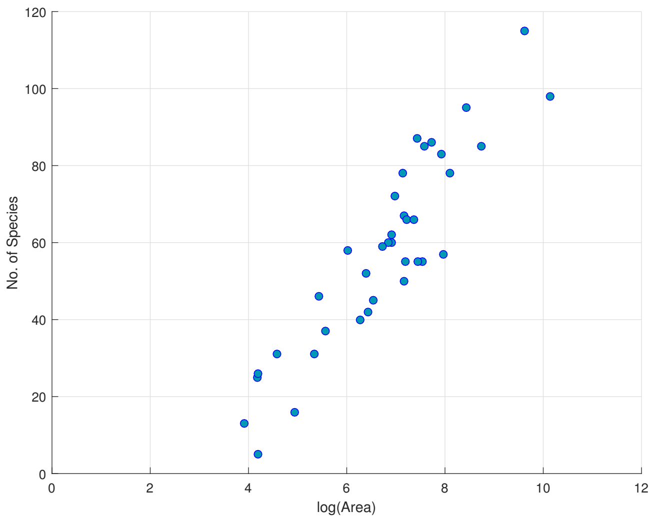

Figure 1: Scatter Plot of Rydin 1988 Data Set

3 Model Structure

Derivation

Notice that $\forall x \in \mathbb{R}$, $log(e^x) = x$. Furthermore, notice that $1 + e^x \approx e^x$ when $x \to \infty$. Hence, $log(1 + e^x) \approx log(e^x) = x$ as $x \to \infty$ This means that the function has a linear characteristic for large values of x. On the other hand, as $x \to -\infty$, $log(1 + e^x) \to log(1) = 0$. Hence, the function is constant for small values of x. Thus, the model that we would want to curve fit should be about the form $log(1 + e^x)$.
Hence, by assigning parameters into the model, we arrive at:

$$Y = a(log(b + e^{cX-d})) \tag{1}$$

Proof:
We want to prove that our model satisfies the conditions mentioned above:

- $\lim_{X\to-\infty}(Y - \alpha) = 0$ for some $\alpha \in \mathbb{R}$ (i.e. $Y \approx \alpha$ when X is small)

- $\lim\limits_{X \to \infty} (Y - \beta X - \gamma) = 0$ for some $\beta, \gamma \in \mathbb{R}$
  (i.e. $Y \approx \beta X + \gamma$ when X is large)

Consider $\alpha = alog(b)$.

$$\begin{aligned}\lim_{X \to -\infty} (Y - \alpha) &= \lim_{X \to -\infty} [a(log(b + e^{cX-d})) - alog(b)] \\ &= a(log(b + 0)) - alog(b) \\ &= 0\end{aligned}$$

Consider $\beta = ac$ and $\gamma = -ad$. We first consider $\lim\limits_{X \to \infty} \frac{Y}{\beta X + \gamma}$.

$$\begin{aligned}\lim_{X \to \infty} \frac{Y}{\beta X + \gamma} &= \lim_{X \to \infty} \frac{a(log(b + e^{cX-d})}{a(cX - d)} \left(\frac{\infty}{\infty}\right) \\ &= \lim_{X \to \infty} \frac{\frac{a}{b+e^{cX-d}} \cdot e^{cX-d} \cdot c}{ac} \text{ (By L'Hôpital Rule)} \\ &= \lim_{X \to \infty} \frac{1}{\frac{b}{e^{cX-d}} + 1} \\ &= \frac{1}{0+1} \\ &= 1\end{aligned}$$

Hence, $Y \approx \beta X + \gamma$ as $X \to \infty$ since $\lim\limits_{X \to \infty} \frac{Y}{\beta X + \gamma} = 1$.

$$\therefore \lim_{X \to \infty} (Y - \beta X - \gamma) = 0 \text{ (By Limit Law)}$$

Therefore, we proposed the model $Y = a(log(b + e^{cX-d}))$, for some non-negative $a, b, c, d \in \mathbb{R}$. Notice that we want to achieve a graph of our model that concaves upwards for some interval I. It is easy to check that if any values of a, b, c were to be negative, the shape of the function will not look the same as the hypothetical smooth model in the handout. Since our data set requires the model to be translated to the right from the origin (refer to Figure 2 below), our model then requires a translation of d units towards the positive x-axis (i.e. $cx - d$ for the model) where c, d is some non-negative real number. Hence, we restrain all our parameters to non-negative values.

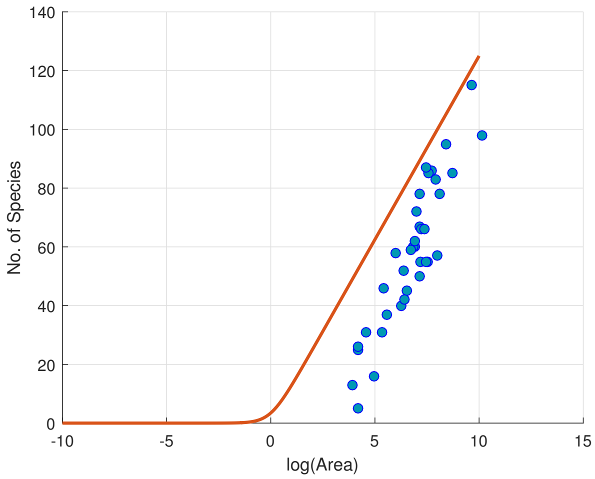

Figure 2: General Plot of $Y = a(log(b + e^{cX-d}))$ with Rydin 1988 Data Set

4 Particle Swarm Optimization

Background Information

Particle Swarm Optimization (PSO) is a computational algorithm that makes use of social behaviour stimulation to obtain a optimized solution via improvement of the produced solution at every iteration. It is created by Dr. Eberhart and Dr. Kennedy in 1995 based on the idea of a swarm of bees or flock of birds together[1].

Problem Definition

In any PSO problem, there exist an objective function whereby this function is the function to be optimized by PSO. In general, there could exist more than one objective required for a PSO problem. In our problem, we only have one objective which is to minimize our objective function. We thus define our single-objective PSO problem as to minimize the sum of squares formula for our model. Hence, we would like to obtain the solution for the global minimum of:

$$E(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) = \sum_{i=0}^{37}[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}})) - y_i]^2 \tag{2}$$

where $E(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d})$ is the sum of squares formula for our model.
We implemented a MATLAB function `LinNonLin.m` that takes in `[a b c d]` as input and outputs the sum of squares error (can also be found in Appendix D):

```
function SqError = LinNonLin(X)
a=X(1); b=X(2); c=X(3); d=X(4);
T=importCSV(); % importCSV() imports the data set (*.csv) into MATLAB
Y=a*(log(b+exp(c*(T.Area)-d)));
SqError = sum((Y-T.Species).^2);
end
```

By adapting a PSO algorithm implemented by S. Mostapha Kalami Heris[2], we would be able to obtain the parameter values $(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d})$ of that least squares error to obtain a solution of our model for the data set. Note that the system of nonlinear equations for the partial derivatives (refer to Appendix C) of E with respect to the parameters cannot be solved explicitly hence the requirement of an alternative method such as PSO.

General Idea

The intuitive understanding of PSO in our problem goes as follows: PSO initializes by randomly generating particles (i.e. data points in $\mathbb{R}^n$) and swarm around $\mathbb{R}^n$ (where n is the no. of parameters in our model) at every iteration. Based on our objective function, we have 4 parameters hence our particles are defined in $\mathbb{R}^4$ (i.e. with values of `a`, `b`, `c`, `d` that indicates the position of the particle in $\mathbb{R}^4$). During each iteration, the particles have a certain cost (i.e. Sum of Squares Error) which evaluates the particles with a value calculated based on our objective function using the particle's position as input. The particles also have a velocity which is contributed from social, cognitive and inertia components (will be discussed in the Velocity section) in order to assess and travel to a better position for the next iteration. Ideally, all particles should swarm to a common position (i.e. Least Squares Error) eventually.

Velocity Updating & Constriction Factor

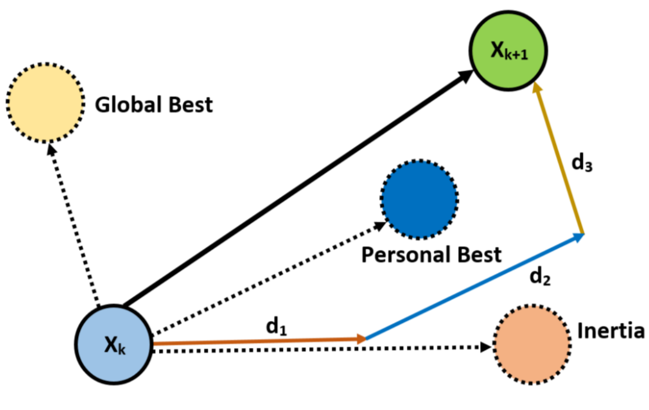

Figure 3: Components that affect Particle X_k's velocity

The velocity vector describes the movement of the particle by taking into account its direction and distance. On every iteration i of PSO, the velocity vector causes the position vector of the particle to change from its initial position X_k to its new position X_{k+1}. Based on Figure 3, the velocity vector is the sum of the 3 components; inertia, cognitive and social. Each particle moves towards the inertia vector in order to stay in its original direction, with an inertia weight of ω. Hence, the inertia component results in this: $\omega v_i(t)$ where i is the iteration and t is time. The cognitive component comprises of the personal best velocity vector, acceleration coefficient (c_1) and a random number (r_1) where r_1 is randomly generated by using `rand(VarSize)`. Here, `VarSize` is the matrix size of the no. of parameters (i.e. 1x4 matrix containing coordinates (position) in $\mathbb{R}^4$). In general, the cognitive component acts as the memory of the particle such that the particle can return to the personal best position. Hence, the cognitive component results in the following term: $c_1 r_1[X_i^*(t) - X_i(t)]$ where $X_i^*(t)$ is the personal best position of the particle at iteration i at time t and $X_k = X_i(t)$. Lastly, the social component comprises of the global best velocity vector, acceleration coefficient (c_2) and another random number (r_2) where where r_2 is randomly generated by using `rand(VarSize)` as well. Hence, we have the following term: $c_2 r_2[G(t) - X_i(t)]$ where $G(t)$ is the global best position and and $X_k = X_i(t)$. Note that since the personal best is the best experience of each particle and the global best is the common best experience of all particles in the swarm, the particle's velocity and position as well as the global best and the personal best are constantly updated at every iteration i of the PSO (will be shown in the next section). Thus, summing up all 3 components, we arrived at the equation 3 below:

$$v_i(t+1) = \omega v_i(t) + c_1 r_1[X_i^*(t) - X_i(t)] + c_2 r_2[G(t) - X_i(t)] \tag{3}$$

Following equation 3, it also known as the velocity updating equation in the PSO Main Algorithm and the following below shows the code segment that implements equation 3 and the position updating of the particle in the MATLAB Code by S. Mostapha Kalami Heris[2].

```
% Update Velocity
particle(i).Velocity = w*particle(i).Velocity ...
+ c1*rand(VarSize).*(particle(i).Best.Position - particle(i).Position)
    ...
+ c2*rand(VarSize).*(GlobalBest.Position - particle(i).Position);
% Update  Position
particle(i).Position = particle(i).Position + particle(i).Velocity;
```

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \tag{4}$$

The inertia weight ω can be calculated based on equation 4. Although PSO is a generalised method and has different pre-defined configurations for the various variables shown in this section, Clerk and Kennedy in 2002 [3], however discovered that there are predefined good configurations for the coefficients ω, c_1 and c_2. According to the equation, κ is taken as 1 and $\phi = \phi_1 + \phi_2 = 4.10$ where $\phi_1 = \phi_2 = 2.05$, $\phi > 4$. Hence, we arrived with $\chi = 0.7298$. We then take $\omega = \chi = 0.7298$, $c_1 = \chi\phi_1$ and $c_1 = \chi\phi_2$. Moreover, the inertia weight ω has a linear coefficient `wdamp` which acts as a damping ratio of ω. By setting `wdamp` at a certain value between 0.8 and 1.2 (inclusive), we can control the rate of convergence of the PSO to the least squares error. In our case, we found that setting `wdamp` as 1.0 is most optimal for our problem.

Initialisation

Before PSO starts to optimize our equation 2, the MATLAB Script `pso2.m` by S. Mostapha Kalami Heris[2] initializes the necessary variables in order to start the algorithm (Refer to Appendix E lines 25-39 of the code). A MATLAB structure `params` is being created to store all the necessary constants for the algorithm. Based on the previous section, we have constants related to velocity such as `w` $= \chi$ (Inertia Coefficient), `wdamp`=1 (Damping Ratio), `c1`=Personal Acceleration Coefficient and `c2`=Social Acceleration Coefficient. As mentioned previously, PSO is a very generalised method hence the constants might be unique for specific problems. For this problem, we found that the following values set for the remaining constants yield the best results: `nPop`=30 (Population Size of the Swarm), `MaxIt` = 500 (No. of Iterations), `VarMin` = 0 (Since parameters are non-negative) and `VarMax` = 10. In this MATLAB Code used, `MaxVelocity` is set to `0.2*(VarMax-VarMin)` and `MinVelocity=-MaxVelocity`. The particle's initial positions are uniformly random generated using `unifrnd(VarMin,VarMax,VarSize)` which generates continuous uniform random numbers of `VarSize`. Lastly, the global best is initialised as `GlobalBest.Cost=inf` because ∞ is the largest global minimum. (If the problem is to find global maximum instead, then `GlobalBest.Cost=-inf`). After configuring the constants needed, the function `pso2.m` calls upon `PSO.m` at line 43.

Main Algorithm

Based on most of the concepts and variables have been discussed above, hence the main algorithm `PSO.m` comprises of the following at every iteration i:

- Updating the particles' velocities (via equation 3) and the position

- Ensuring velocity and position of particles are within their bounds
- Evaluation of Particle's Sum of Squares Error (Using `LinNonLin.m`)
- Update Personal Best
- Update Global Best

Additionally, the Main Algorithm can be referenced from lines 85-122 of `PSO.m` which is in Appendix F.

5 Results

Our PSO found that the least squares error(LSE) for our model is ≈ 3710.307 after confirmation through several rounds of executing the PSO Algorithm, each round having 500 iterations. (The best round that generated the LSE of ≈ 3710.307 is shown in the Appendix G). Following this least squares error, we obtained the values of our parameters `a` ,`b`, `c` and `d`. (i.e. `a` = 7.9506 and `b` = 1.8389, `c` = 2.0076 and `d` = 6.3317). The following output is also shown below:

```
>> BestSol =
     Position: [7.9506 1.8389 2.0076 6.3317]
     Cost: 3.7103e+03
```

Using the following parameter values, the scatter plot is now plotted superimposed with the curve of our non-linear model:

$$Y = 7.9506(log(1.8389 + e^{(2.0076X-6.3317)})) \tag{5}$$

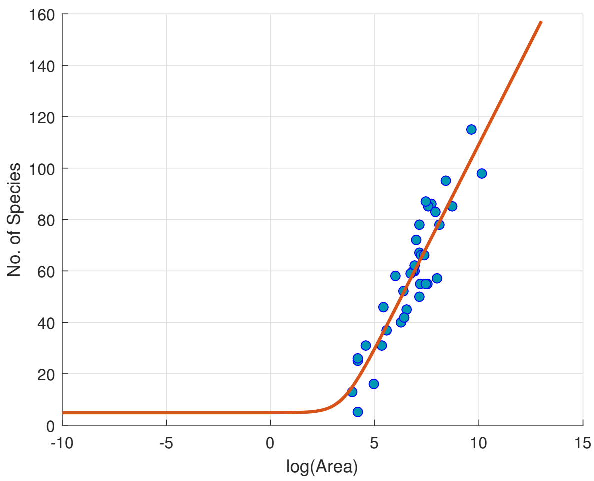

Figure 4: $Y = a(log(b + e^{cX-d}))$ fitted to Rydin 1988 Data Set

6 Possible Explanation of Small Island Effect

Based on the research article written by William A. Niering , small islands are easily vulnerable to climate disasters such as heavy storms and huge waves[4]. This actually can result in the destruction of the food supply such as fruits and trees that are available on the island for the taxons. With an insufficient abundance of food and water, taxons can fail to survive and their no. of species can fall to a certain threshold. Below the threshold, there is only a remaining fixed no. of species on the small island such that that number can sustain survival with that low amount of food and habitat. Hence, this explains the trend for small values of $log(Area)$ from our model. On the other hand, it follows that there could be an abundance of taxons available given the $log(Area)$ is above the threshold creating a stable eco system on the island itself between the taxon and other taxons on the island. This could then promote a constant growth rate (Since the derivative of the linear straight line is a constant function). Hence, the gradient $\beta = ac$ from section 3 could represent the maximum growth rate attainable for that taxon on an island with a given $log(Area)$ that is large enough.

References

[1] PSO Tutorial
`http://www.swarmintelligence.org/tutorials.php`

[2] Particle Swarm Optimization in MATLAB
`http://yarpiz.com/50/ypea102-particle-swarm-optimization`

[3] Clerk, M.; Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6, 1, (2002) 58-73.

[4] William A. Niering. *Terrestrial Ecology of Kapingamarangi Atoll, Caroline Islands.* Ecological Monographs, Vol. 33, No. 2 (Spring, 1963), pp. 131-160

Appendix A: Rydin 1988 Data Set

Data Set	Area (km^2)	Area (m^2)	log(Area)	No. of Species
Rydin 1988	0.000528	528	6.269096284	40
Rydin 1988	0.00226	2260	7.723120092	86
Rydin 1988	0.00197	1970	7.585788822	85
Rydin 1988	0.001882	1882	7.54009032	55
Rydin 1988	0.001264	1264	7.142036575	78
Rydin 1988	0.00129	1290	7.162397497	67
Rydin 1988	0.001072	1072	6.977281342	72
Rydin 1988	0.001371	1371	7.22329568	66
Rydin 1988	0.000261	261	5.564520407	37
Rydin 1988	0.00014	140	4.941642423	16
Rydin 1988	0.000066	66	4.189654742	5
Rydin 1988	0.001329	1329	7.192182059	55
Rydin 1988	0.001006	1006	6.913737351	60
Rydin 1988	0.000596	596	6.390240667	52
Rydin 1988	0.00069	690	6.536691598	45
Rydin 1988	0.002755	2755	7.921172722	83
Rydin 1988	0.000943	943	6.849066283	60
Rydin 1988	0.000229	229	5.433722004	46
Rydin 1988	0.006232	6232	8.737452588	85
Rydin 1988	0.001292	1292	7.163946684	50
Rydin 1988	0.000999	999	6.906754779	62
Rydin 1988	0.000065	65	4.17438727	25
Rydin 1988	0.00005	50	3.912023005	13
Rydin 1988	0.000619	619	6.428105273	42
Rydin 1988	0.001576	1576	7.36264527	66
Rydin 1988	0.000209	209	5.342334252	31
Rydin 1988	0.000066	66	4.189654742	26
Rydin 1988	0.000097	97	4.574710979	31
Rydin 1988	0.000409	409	6.013715156	58
Rydin 1988	0.00083	830	6.721425701	59
Rydin 1988	0.0017	1700	7.43838353	87
Rydin 1988	0.00459	4590	8.431635303	95
Rydin 1988	0.01512	15120	9.62377365	115
Rydin 1988	0.02517	25170	10.13340809	98
Rydin 1988	0.0033	3300	8.101677747	78
Rydin 1988	0.0029	2900	7.972466016	57
Rydin 1988	0.001715	1715	7.44716836	55

Table 1: Rydin 1988 Data Set from Rydin1988.csv

Appendix B: Scatter Plot MATLAB Function

```
function T=importCSV2()
% This function imports (*.csv) and
% plots the Rydin 1988 scatter data.
fileID = fopen('Rydin1988.csv');
C = textscan(fileID,'%s %f %f %f %f',...
        'HeaderLines',1,'Delimiter',',');
fclose(fileID);
Area = C{1,4}; Species = C{1,5};
T=table(Area,Species);
figure;
s=scatter(T.Area,T.Species);
s.LineWidth = 0.6;
s.MarkerEdgeColor = 'b';
s.MarkerFaceColor = [0 0.6 0.7];
xlabel('log(Area)');
ylabel('No. of Species');
end
```

Appendix C: Partial Derivatives of E(a,b,c,d)

The least squares error of our model is:

$$E(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) = \sum_{i=0}^{37}[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))-y_i]^2$$

At a point of global minimum, we will have:

$$\frac{\partial E}{\partial \mathtt{a}} = \sum_{i=0}^{37} 2\cdot[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))-y_i]\cdot(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))$$

$$\frac{\partial E}{\partial \mathtt{b}} = \sum_{i=0}^{37} 2\cdot[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))-y_i]\cdot\mathtt{a}\cdot\frac{1}{\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}}$$

$$\frac{\partial E}{\partial \mathtt{c}} = \sum_{i=0}^{37} 2\cdot[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))-y_i]\cdot\mathtt{a}\cdot\frac{e^{\mathtt{c}x_i-\mathtt{d}}\cdot x_i}{\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}}$$

$$\frac{\partial E}{\partial \mathtt{d}} = \sum_{i=0}^{37} 2\cdot[\mathtt{a}(log(\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}))-y_i]\cdot\mathtt{a}\cdot\frac{e^{\mathtt{c}x_i-\mathtt{d}}\cdot(-1)}{\mathtt{b}+e^{\mathtt{c}x_i-\mathtt{d}}}$$

Similar to the example provided in the handout, the system of nonlinear equations for our model also cannot be solved explicitly.

Appendix D: Sum of Squares MATLAB Function

```
function SqError = LinNonLin(X)
a=X(1); b=X(2); c=X(3); d=X(4);

T=importCSV();

Y=a*(log(b+exp(c*(T.Area)-d)));

SqError = sum((Y-T.Species).^2);
end
```

```
function T=importCSV()
fileID = fopen('Rydin1988.csv');
C = textscan(fileID,'%s %f %f %f %
   f',...
'HeaderLines',1,'Delimiter',',');
fclose(fileID); Area = C{1,4};
Species = C{1,5};
T=table(Area,Species);
end
```

Appendix E: PSO MATLAB Script pso2.m

```
%
% Copyright (c) 2016, Yarpiz (www.yarpiz.com)
% All rights reserved. Please read the "license.txt" for license terms.
%
% Project Code: YTEA101
% Project Title: Particle Swarm Optimization Video Tutorial
% Publisher: Yarpiz (www.yarpiz.com)
%
% Developer and Instructor: S. Mostapha Kalami Heris (Member of Yarpiz Team)
%
% Contact Info: sm.kalami@gmail.com, info@yarpiz.com
%

clc;
clear;
close all;

%% Problem Definiton

problem.CostFunction = @(x) LinNonLin(x);  % Cost Function
problem.nVar = 4;       % Number of Unknown (Decision) Variables
problem.VarMin = 0;    % Lower Bound of Decision Variables
problem.VarMax = 10;    % Upper Bound of Decision Variables

%% Parameters of PSO

% Constriction Coefficients
kappa = 1;
phi1 = 2.05;
phi2 = 2.05;
phi = phi1 + phi2;
chi = 2*kappa/abs(2-phi-sqrt(phi^2-4*phi));

params.MaxIt = 500;         % Maximum Number of Iterations
params.nPop = 30;            % Population Size (Swarm Size)
params.w = chi;              % Intertia Coefficient
params.wdamp = 1;            % Damping Ratio of Inertia Coefficient
params.c1 = chi*phi1;        % Personal Acceleration Coefficient
params.c2 = chi*phi2;        % Social Acceleration Coefficient
params.ShowIterInfo = true; % Flag for Showing Iteration Informatin

%% Calling PSO

out = PSO(problem, params);

BestSol = out.BestSol;
BestCosts = out.BestCosts;

%% Results

figure;
% plot(BestCosts, 'LineWidth', 2);
semilogy(BestCosts, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Squares Error');
grid on;
```

Appendix F: PSO Main Algorithm

```
%
% Copyright (c) 2016, Yarpiz (www.yarpiz.com)
% All rights reserved. Please read the "license.txt" for license terms.
%
% Project Code: YTEA101
% Project Title: Particle Swarm Optimization Video Tutorial
% Publisher: Yarpiz (www.yarpiz.com)
%
% Developer and Instructor: S. Mostapha Kalami Heris (Member of Yarpiz
    Team)
%
% Contact Info: sm.kalami@gmail.com, info@yarpiz.com
%

function out = PSO(problem, params)

    %% Problem Definiton

    CostFunction = problem.CostFunction;  % Cost Function

    nVar = problem.nVar;        % Number of Unknown (Decision) Variables

    VarSize = [1 nVar];         % Matrix Size of Decision Variables

    VarMin = problem.VarMin;    % Lower Bound of Decision Variables
    VarMax = problem.VarMax;    % Upper Bound of Decision Variables


    %% Parameters of PSO

    MaxIt = params.MaxIt;   % Maximum Number of Iterations

    nPop = params.nPop;     % Population Size (Swarm Size)

    w = params.w;           % Intertia Coefficient
    wdamp = params.wdamp;   % Damping Ratio of Inertia Coefficient
    c1 = params.c1;         % Personal Acceleration Coefficient
    c2 = params.c2;         % Social Acceleration Coefficient

    % The Flag for Showing Iteration Information
    ShowIterInfo = params.ShowIterInfo;

    MaxVelocity = 0.2*(VarMax-VarMin);
    MinVelocity = -MaxVelocity;

    %% Initialization

    % The Particle Template
    empty_particle.Position = [];
    empty_particle.Velocity = [];
    empty_particle.Cost = [];
    empty_particle.Best.Position = [];
    empty_particle.Best.Cost = [];

    % Create Population Array
    particle = repmat(empty_particle, nPop, 1);

    % Initialize Global Best
```

```
    GlobalBest.Cost = inf;

    % Initialize Population Members
    for i=1:nPop

        % Generate Random Solution
        particle(i).Position = unifrnd(VarMin, VarMax, VarSize);

        % Initialize Velocity
        particle(i).Velocity = zeros(VarSize);

        % Evaluation
        particle(i).Cost = CostFunction(particle(i).Position);

        % Update the Personal Best
        particle(i).Best.Position = particle(i).Position;
        particle(i).Best.Cost = particle(i).Cost;
        % Update Global Best
        if particle(i).Best.Cost < GlobalBest.Cost
            GlobalBest = particle(i).Best;
        end
    end

    % Array to Hold Best Cost Value on Each Iteration
    BestCosts = zeros(MaxIt, 1);


%% Main Loop of PSO

    for it=1:MaxIt

        for i=1:nPop

            % Update Velocity
            particle(i).Velocity = w*particle(i).Velocity ...
                + c1*rand(VarSize).*(particle(i).Best.Position - particle
                   (i).Position) ...
                + c2*rand(VarSize).*(GlobalBest.Position - particle(i).
                   Position);

            % Apply Velocity Limits
            particle(i).Velocity = max(particle(i).Velocity, MinVelocity)
                ;
            particle(i).Velocity = min(particle(i).Velocity, MaxVelocity)
                ;

            % Update Position
            particle(i).Position = particle(i).Position + particle(i).
                Velocity;

            % Apply Lower and Upper Bound Limits
              particle(i).Position = max(particle(i).Position, VarMin);
              particle(i).Position = min(particle(i).Position, VarMax);

            % Evaluation
            particle(i).Cost = CostFunction(particle(i).Position);

            % Update Personal Best
            if particle(i).Cost < particle(i).Best.Cost

                particle(i).Best.Position = particle(i).Position;
```

```
                    particle(i).Best.Cost = particle(i).Cost;

                    % Update Global Best
                    if particle(i).Best.Cost < GlobalBest.Cost
                        GlobalBest = particle(i).Best;
                    end

                end
            end

            % Store the Best Cost Value
            BestCosts(it) = GlobalBest.Cost;

            % Display Iteration Information
            if ShowIterInfo
                disp(['Iteration ' num2str(it) ': Best Cost = ' num2str(BestCosts(it))]);
            end

            % Damping Inertia Coefficient
            w = w * wdamp;
        end

        out.pop = particle;
        out.BestSol = GlobalBest;
        out.BestCosts = BestCosts;

    end
```

Appendix G: PSO Result

```
Iteration 1: Best Cost = 4529.7507
Iteration 2: Best Cost = 4103.2849
Iteration 3: Best Cost = 3809.6049
Iteration 4: Best Cost = 3809.6049
Iteration 5: Best Cost = 3809.6049
Iteration 6: Best Cost = 3809.6049
Iteration 7: Best Cost = 3809.6049
Iteration 8: Best Cost = 3772.6319
Iteration 9: Best Cost = 3772.6319
Iteration 10: Best Cost = 3772.6319
Iteration 11: Best Cost = 3772.6319
Iteration 12: Best Cost = 3772.6319
Iteration 13: Best Cost = 3772.6319
Iteration 14: Best Cost = 3772.6319
Iteration 15: Best Cost = 3753.8686
Iteration 16: Best Cost = 3753.8686
Iteration 17: Best Cost = 3744.4635
Iteration 18: Best Cost = 3744.4635
Iteration 19: Best Cost = 3732.1147
Iteration 20: Best Cost = 3721.6442
Iteration 21: Best Cost = 3721.6442
Iteration 22: Best Cost = 3721.6442
Iteration 23: Best Cost = 3721.6442
Iteration 24: Best Cost = 3721.6442
Iteration 25: Best Cost = 3721.6442
Iteration 26: Best Cost = 3721.6442
Iteration 27: Best Cost = 3721.6442
Iteration 28: Best Cost = 3721.6442
Iteration 29: Best Cost = 3721.6442
Iteration 30: Best Cost = 3721.6442
Iteration 31: Best Cost = 3721.6442
Iteration 32: Best Cost = 3721.6442
Iteration 33: Best Cost = 3721.6442
Iteration 34: Best Cost = 3718.4412
Iteration 35: Best Cost = 3718.4412
Iteration 36: Best Cost = 3718.4412
Iteration 37: Best Cost = 3718.4412
Iteration 38: Best Cost = 3718.4412
Iteration 39: Best Cost = 3718.4412
Iteration 40: Best Cost = 3718.4412
Iteration 41: Best Cost = 3718.4412
Iteration 42: Best Cost = 3717.509
Iteration 43: Best Cost = 3716.7885
Iteration 44: Best Cost = 3716.7885
Iteration 45: Best Cost = 3716.6145
Iteration 46: Best Cost = 3714.8838
Iteration 47: Best Cost = 3714.6819
Iteration 48: Best Cost = 3714.6725
Iteration 49: Best Cost = 3714.6725
Iteration 50: Best Cost = 3714.6725
Iteration 51: Best Cost = 3714.6725
Iteration 52: Best Cost = 3714.6725
Iteration 53: Best Cost = 3714.5476
Iteration 54: Best Cost = 3714.5476
Iteration 55: Best Cost = 3714.5476
Iteration 56: Best Cost = 3714.5476
Iteration 57: Best Cost = 3710.6065
Iteration 58: Best Cost = 3710.6065
Iteration 59: Best Cost = 3710.6065
Iteration 60: Best Cost = 3710.6065
Iteration 61: Best Cost = 3710.6065
Iteration 62: Best Cost = 3710.6065
Iteration 63: Best Cost = 3710.6065
Iteration 64: Best Cost = 3710.6065
Iteration 65: Best Cost = 3710.4496
Iteration 66: Best Cost = 3710.4496
Iteration 67: Best Cost = 3710.3639
Iteration 68: Best Cost = 3710.3639
Iteration 69: Best Cost = 3710.3639
Iteration 70: Best Cost = 3710.339
Iteration 71: Best Cost = 3710.339
Iteration 72: Best Cost = 3710.339
Iteration 73: Best Cost = 3710.339
Iteration 74: Best Cost = 3710.339
Iteration 75: Best Cost = 3710.3242
Iteration 76: Best Cost = 3710.3213
Iteration 77: Best Cost = 3710.3213
Iteration 78: Best Cost = 3710.3213
Iteration 79: Best Cost = 3710.3213
Iteration 80: Best Cost = 3710.3213
Iteration 81: Best Cost = 3710.3213
Iteration 82: Best Cost = 3710.3139
Iteration 83: Best Cost = 3710.3139
Iteration 84: Best Cost = 3710.3139
Iteration 85: Best Cost = 3710.3083
Iteration 86: Best Cost = 3710.3083
Iteration 87: Best Cost = 3710.3083
Iteration 88: Best Cost = 3710.3083
Iteration 89: Best Cost = 3710.3083
Iteration 90: Best Cost = 3710.3083
Iteration 91: Best Cost = 3710.3083
Iteration 92: Best Cost = 3710.3083
Iteration 93: Best Cost = 3710.3076
Iteration 94: Best Cost = 3710.3076
Iteration 95: Best Cost = 3710.3076
Iteration 96: Best Cost = 3710.307
Iteration 97: Best Cost = 3710.307
Iteration 98: Best Cost = 3710.307
Iteration 99: Best Cost = 3710.307
Iteration 100: Best Cost = 3710.307
Iteration 101: Best Cost = 3710.307
Iteration 102: Best Cost = 3710.307
Iteration 103: Best Cost = 3710.307
Iteration 104: Best Cost = 3710.307
Iteration 105: Best Cost = 3710.307
Iteration 106: Best Cost = 3710.307
Iteration 107: Best Cost = 3710.307
Iteration 108: Best Cost = 3710.307
Iteration 109: Best Cost = 3710.307
Iteration 110: Best Cost = 3710.307
Iteration 111: Best Cost = 3710.307
Iteration 112: Best Cost = 3710.307
Iteration 113: Best Cost = 3710.307
Iteration 114: Best Cost = 3710.307
Iteration 115: Best Cost = 3710.307
Iteration 116: Best Cost = 3710.307
Iteration 117: Best Cost = 3710.307
Iteration 118: Best Cost = 3710.307
Iteration 119: Best Cost = 3710.307
Iteration 120: Best Cost = 3710.307
Iteration 121: Best Cost = 3710.307
Iteration 122: Best Cost = 3710.307
Iteration 123: Best Cost = 3710.307
Iteration 124: Best Cost = 3710.307
Iteration 125: Best Cost = 3710.307
Iteration 126: Best Cost = 3710.307
Iteration 127: Best Cost = 3710.307
Iteration 128: Best Cost = 3710.307
```

```
Iteration 129: Best Cost = 3710.307
Iteration 130: Best Cost = 3710.307
Iteration 131: Best Cost = 3710.307
Iteration 132: Best Cost = 3710.307
Iteration 133: Best Cost = 3710.307
Iteration 134: Best Cost = 3710.307
Iteration 135: Best Cost = 3710.307
Iteration 136: Best Cost = 3710.307
Iteration 137: Best Cost = 3710.307
Iteration 138: Best Cost = 3710.307
Iteration 139: Best Cost = 3710.307
Iteration 140: Best Cost = 3710.307
Iteration 141: Best Cost = 3710.307
Iteration 142: Best Cost = 3710.307
Iteration 143: Best Cost = 3710.307
Iteration 144: Best Cost = 3710.307
Iteration 145: Best Cost = 3710.307
Iteration 146: Best Cost = 3710.307
Iteration 147: Best Cost = 3710.307
Iteration 148: Best Cost = 3710.307
Iteration 149: Best Cost = 3710.307
Iteration 150: Best Cost = 3710.307
Iteration 151: Best Cost = 3710.307
Iteration 152: Best Cost = 3710.307
Iteration 153: Best Cost = 3710.307
Iteration 154: Best Cost = 3710.307
Iteration 155: Best Cost = 3710.307
Iteration 156: Best Cost = 3710.307
Iteration 157: Best Cost = 3710.307
Iteration 158: Best Cost = 3710.307
Iteration 159: Best Cost = 3710.307
Iteration 160: Best Cost = 3710.307
Iteration 161: Best Cost = 3710.307
Iteration 162: Best Cost = 3710.307
Iteration 163: Best Cost = 3710.307
Iteration 164: Best Cost = 3710.307
Iteration 165: Best Cost = 3710.307
Iteration 166: Best Cost = 3710.307
Iteration 167: Best Cost = 3710.307
Iteration 168: Best Cost = 3710.307
Iteration 169: Best Cost = 3710.307
Iteration 170: Best Cost = 3710.307
Iteration 171: Best Cost = 3710.307
Iteration 172: Best Cost = 3710.307
Iteration 173: Best Cost = 3710.307
Iteration 174: Best Cost = 3710.307
Iteration 175: Best Cost = 3710.307
Iteration 176: Best Cost = 3710.307
Iteration 177: Best Cost = 3710.307
Iteration 178: Best Cost = 3710.307
Iteration 179: Best Cost = 3710.307
Iteration 180: Best Cost = 3710.307
Iteration 181: Best Cost = 3710.307
Iteration 182: Best Cost = 3710.307
Iteration 183: Best Cost = 3710.307
Iteration 184: Best Cost = 3710.307
Iteration 185: Best Cost = 3710.307
Iteration 186: Best Cost = 3710.307
Iteration 187: Best Cost = 3710.307
Iteration 188: Best Cost = 3710.307
Iteration 189: Best Cost = 3710.307
Iteration 190: Best Cost = 3710.307
Iteration 191: Best Cost = 3710.307
Iteration 192: Best Cost = 3710.307
Iteration 193: Best Cost = 3710.307
Iteration 194: Best Cost = 3710.307
Iteration 195: Best Cost = 3710.307
Iteration 196: Best Cost = 3710.307
Iteration 197: Best Cost = 3710.307
Iteration 198: Best Cost = 3710.307
Iteration 199: Best Cost = 3710.307
Iteration 200: Best Cost = 3710.307
Iteration 201: Best Cost = 3710.307
Iteration 202: Best Cost = 3710.307
Iteration 203: Best Cost = 3710.307
Iteration 204: Best Cost = 3710.307
Iteration 205: Best Cost = 3710.307
Iteration 206: Best Cost = 3710.307
Iteration 207: Best Cost = 3710.307
Iteration 208: Best Cost = 3710.307
Iteration 209: Best Cost = 3710.307
Iteration 210: Best Cost = 3710.307
Iteration 211: Best Cost = 3710.307
Iteration 212: Best Cost = 3710.307
Iteration 213: Best Cost = 3710.307
Iteration 214: Best Cost = 3710.307
Iteration 215: Best Cost = 3710.307
Iteration 216: Best Cost = 3710.307
Iteration 217: Best Cost = 3710.307
Iteration 218: Best Cost = 3710.307
Iteration 219: Best Cost = 3710.307
Iteration 220: Best Cost = 3710.307
Iteration 221: Best Cost = 3710.307
Iteration 222: Best Cost = 3710.307
Iteration 223: Best Cost = 3710.307
Iteration 224: Best Cost = 3710.307
Iteration 225: Best Cost = 3710.307
Iteration 226: Best Cost = 3710.307
Iteration 227: Best Cost = 3710.307
Iteration 228: Best Cost = 3710.307
Iteration 229: Best Cost = 3710.307
Iteration 230: Best Cost = 3710.307
Iteration 231: Best Cost = 3710.307
Iteration 232: Best Cost = 3710.307
Iteration 233: Best Cost = 3710.307
Iteration 234: Best Cost = 3710.307
Iteration 235: Best Cost = 3710.307
Iteration 236: Best Cost = 3710.307
Iteration 237: Best Cost = 3710.307
Iteration 238: Best Cost = 3710.307
Iteration 239: Best Cost = 3710.307
Iteration 240: Best Cost = 3710.307
Iteration 241: Best Cost = 3710.307
Iteration 242: Best Cost = 3710.307
Iteration 243: Best Cost = 3710.307
Iteration 244: Best Cost = 3710.307
Iteration 245: Best Cost = 3710.307
Iteration 246: Best Cost = 3710.307
Iteration 247: Best Cost = 3710.307
Iteration 248: Best Cost = 3710.307
Iteration 249: Best Cost = 3710.307
Iteration 250: Best Cost = 3710.307
Iteration 251: Best Cost = 3710.307
Iteration 252: Best Cost = 3710.307
Iteration 253: Best Cost = 3710.307
Iteration 254: Best Cost = 3710.307
Iteration 255: Best Cost = 3710.307
Iteration 256: Best Cost = 3710.307
Iteration 257: Best Cost = 3710.307
Iteration 258: Best Cost = 3710.307
Iteration 259: Best Cost = 3710.307
Iteration 260: Best Cost = 3710.307
```

```
Iteration 261: Best Cost = 3710.307
Iteration 262: Best Cost = 3710.307
Iteration 263: Best Cost = 3710.307
Iteration 264: Best Cost = 3710.307
Iteration 265: Best Cost = 3710.307
Iteration 266: Best Cost = 3710.307
Iteration 267: Best Cost = 3710.307
Iteration 268: Best Cost = 3710.307
Iteration 269: Best Cost = 3710.307
Iteration 270: Best Cost = 3710.307
Iteration 271: Best Cost = 3710.307
Iteration 272: Best Cost = 3710.307
Iteration 273: Best Cost = 3710.307
Iteration 274: Best Cost = 3710.307
Iteration 275: Best Cost = 3710.307
Iteration 276: Best Cost = 3710.307
Iteration 277: Best Cost = 3710.307
Iteration 278: Best Cost = 3710.307
Iteration 279: Best Cost = 3710.307
Iteration 280: Best Cost = 3710.307
Iteration 281: Best Cost = 3710.307
Iteration 282: Best Cost = 3710.307
Iteration 283: Best Cost = 3710.307
Iteration 284: Best Cost = 3710.307
Iteration 285: Best Cost = 3710.307
Iteration 286: Best Cost = 3710.307
Iteration 287: Best Cost = 3710.307
Iteration 288: Best Cost = 3710.307
Iteration 289: Best Cost = 3710.307
Iteration 290: Best Cost = 3710.307
Iteration 291: Best Cost = 3710.307
Iteration 292: Best Cost = 3710.307
Iteration 293: Best Cost = 3710.307
Iteration 294: Best Cost = 3710.307
Iteration 295: Best Cost = 3710.307
Iteration 296: Best Cost = 3710.307
Iteration 297: Best Cost = 3710.307
Iteration 298: Best Cost = 3710.307
Iteration 299: Best Cost = 3710.307
Iteration 300: Best Cost = 3710.307
Iteration 301: Best Cost = 3710.307
Iteration 302: Best Cost = 3710.307
Iteration 303: Best Cost = 3710.307
Iteration 304: Best Cost = 3710.307
Iteration 305: Best Cost = 3710.307
Iteration 306: Best Cost = 3710.307
Iteration 307: Best Cost = 3710.307
Iteration 308: Best Cost = 3710.307
Iteration 309: Best Cost = 3710.307
Iteration 310: Best Cost = 3710.307
Iteration 311: Best Cost = 3710.307
Iteration 312: Best Cost = 3710.307
Iteration 313: Best Cost = 3710.307
Iteration 314: Best Cost = 3710.307
Iteration 315: Best Cost = 3710.307
Iteration 316: Best Cost = 3710.307
Iteration 317: Best Cost = 3710.307
Iteration 318: Best Cost = 3710.307
Iteration 319: Best Cost = 3710.307
Iteration 320: Best Cost = 3710.307
Iteration 321: Best Cost = 3710.307
Iteration 322: Best Cost = 3710.307
Iteration 323: Best Cost = 3710.307
Iteration 324: Best Cost = 3710.307
Iteration 325: Best Cost = 3710.307
Iteration 326: Best Cost = 3710.307
Iteration 327: Best Cost = 3710.307
Iteration 328: Best Cost = 3710.307
Iteration 329: Best Cost = 3710.307
Iteration 330: Best Cost = 3710.307
Iteration 331: Best Cost = 3710.307
Iteration 332: Best Cost = 3710.307
Iteration 333: Best Cost = 3710.307
Iteration 334: Best Cost = 3710.307
Iteration 335: Best Cost = 3710.307
Iteration 336: Best Cost = 3710.307
Iteration 337: Best Cost = 3710.307
Iteration 338: Best Cost = 3710.307
Iteration 339: Best Cost = 3710.307
Iteration 340: Best Cost = 3710.307
Iteration 341: Best Cost = 3710.307
Iteration 342: Best Cost = 3710.307
Iteration 343: Best Cost = 3710.307
Iteration 344: Best Cost = 3710.307
Iteration 345: Best Cost = 3710.307
Iteration 346: Best Cost = 3710.307
Iteration 347: Best Cost = 3710.307
Iteration 348: Best Cost = 3710.307
Iteration 349: Best Cost = 3710.307
Iteration 350: Best Cost = 3710.307
Iteration 351: Best Cost = 3710.307
Iteration 352: Best Cost = 3710.307
Iteration 353: Best Cost = 3710.307
Iteration 354: Best Cost = 3710.307
Iteration 355: Best Cost = 3710.307
Iteration 356: Best Cost = 3710.307
Iteration 357: Best Cost = 3710.307
Iteration 358: Best Cost = 3710.307
Iteration 359: Best Cost = 3710.307
Iteration 360: Best Cost = 3710.307
Iteration 361: Best Cost = 3710.307
Iteration 362: Best Cost = 3710.307
Iteration 363: Best Cost = 3710.307
Iteration 364: Best Cost = 3710.307
Iteration 365: Best Cost = 3710.307
Iteration 366: Best Cost = 3710.307
Iteration 367: Best Cost = 3710.307
Iteration 368: Best Cost = 3710.307
Iteration 369: Best Cost = 3710.307
Iteration 370: Best Cost = 3710.307
Iteration 371: Best Cost = 3710.307
Iteration 372: Best Cost = 3710.307
Iteration 373: Best Cost = 3710.307
Iteration 374: Best Cost = 3710.307
Iteration 375: Best Cost = 3710.307
Iteration 376: Best Cost = 3710.307
Iteration 377: Best Cost = 3710.307
Iteration 378: Best Cost = 3710.307
Iteration 379: Best Cost = 3710.307
Iteration 380: Best Cost = 3710.307
Iteration 381: Best Cost = 3710.307
Iteration 382: Best Cost = 3710.307
Iteration 383: Best Cost = 3710.307
Iteration 384: Best Cost = 3710.307
Iteration 385: Best Cost = 3710.307
Iteration 386: Best Cost = 3710.307
Iteration 387: Best Cost = 3710.307
Iteration 388: Best Cost = 3710.307
Iteration 389: Best Cost = 3710.307
Iteration 390: Best Cost = 3710.307
Iteration 391: Best Cost = 3710.307
Iteration 392: Best Cost = 3710.307
```

```
Iteration 393: Best Cost = 3710.307
Iteration 394: Best Cost = 3710.307
Iteration 395: Best Cost = 3710.307
Iteration 396: Best Cost = 3710.307
Iteration 397: Best Cost = 3710.307
Iteration 398: Best Cost = 3710.307
Iteration 399: Best Cost = 3710.307
Iteration 400: Best Cost = 3710.307
Iteration 401: Best Cost = 3710.307
Iteration 402: Best Cost = 3710.307
Iteration 403: Best Cost = 3710.307
Iteration 404: Best Cost = 3710.307
Iteration 405: Best Cost = 3710.307
Iteration 406: Best Cost = 3710.307
Iteration 407: Best Cost = 3710.307
Iteration 408: Best Cost = 3710.307
Iteration 409: Best Cost = 3710.307
Iteration 410: Best Cost = 3710.307
Iteration 411: Best Cost = 3710.307
Iteration 412: Best Cost = 3710.307
Iteration 413: Best Cost = 3710.307
Iteration 414: Best Cost = 3710.307
Iteration 415: Best Cost = 3710.307
Iteration 416: Best Cost = 3710.307
Iteration 417: Best Cost = 3710.307
Iteration 418: Best Cost = 3710.307
Iteration 419: Best Cost = 3710.307
Iteration 420: Best Cost = 3710.307
Iteration 421: Best Cost = 3710.307
Iteration 422: Best Cost = 3710.307
Iteration 423: Best Cost = 3710.307
Iteration 424: Best Cost = 3710.307
Iteration 425: Best Cost = 3710.307
Iteration 426: Best Cost = 3710.307
Iteration 427: Best Cost = 3710.307
Iteration 428: Best Cost = 3710.307
Iteration 429: Best Cost = 3710.307
Iteration 430: Best Cost = 3710.307
Iteration 431: Best Cost = 3710.307
Iteration 432: Best Cost = 3710.307
Iteration 433: Best Cost = 3710.307
Iteration 434: Best Cost = 3710.307
Iteration 435: Best Cost = 3710.307
Iteration 436: Best Cost = 3710.307
Iteration 437: Best Cost = 3710.307
Iteration 438: Best Cost = 3710.307
Iteration 439: Best Cost = 3710.307
Iteration 440: Best Cost = 3710.307
Iteration 441: Best Cost = 3710.307
Iteration 442: Best Cost = 3710.307
Iteration 443: Best Cost = 3710.307
Iteration 444: Best Cost = 3710.307
Iteration 445: Best Cost = 3710.307
Iteration 446: Best Cost = 3710.307
Iteration 447: Best Cost = 3710.307
Iteration 448: Best Cost = 3710.307
Iteration 449: Best Cost = 3710.307
Iteration 450: Best Cost = 3710.307
Iteration 451: Best Cost = 3710.307
Iteration 452: Best Cost = 3710.307
Iteration 453: Best Cost = 3710.307
Iteration 454: Best Cost = 3710.307
Iteration 455: Best Cost = 3710.307
Iteration 456: Best Cost = 3710.307
Iteration 457: Best Cost = 3710.307
Iteration 458: Best Cost = 3710.307
Iteration 459: Best Cost = 3710.307
Iteration 460: Best Cost = 3710.307
Iteration 461: Best Cost = 3710.307
Iteration 462: Best Cost = 3710.307
Iteration 463: Best Cost = 3710.307
Iteration 464: Best Cost = 3710.307
Iteration 465: Best Cost = 3710.307
Iteration 466: Best Cost = 3710.307
Iteration 467: Best Cost = 3710.307
Iteration 468: Best Cost = 3710.307
Iteration 469: Best Cost = 3710.307
Iteration 470: Best Cost = 3710.307
Iteration 471: Best Cost = 3710.307
Iteration 472: Best Cost = 3710.307
Iteration 473: Best Cost = 3710.307
Iteration 474: Best Cost = 3710.307
Iteration 475: Best Cost = 3710.307
Iteration 476: Best Cost = 3710.307
Iteration 477: Best Cost = 3710.307
Iteration 478: Best Cost = 3710.307
Iteration 479: Best Cost = 3710.307
Iteration 480: Best Cost = 3710.307
Iteration 481: Best Cost = 3710.307
Iteration 482: Best Cost = 3710.307
Iteration 483: Best Cost = 3710.307
Iteration 484: Best Cost = 3710.307
Iteration 485: Best Cost = 3710.307
Iteration 486: Best Cost = 3710.307
Iteration 487: Best Cost = 3710.307
Iteration 488: Best Cost = 3710.307
Iteration 489: Best Cost = 3710.307
Iteration 490: Best Cost = 3710.307
Iteration 491: Best Cost = 3710.307
Iteration 492: Best Cost = 3710.307
Iteration 493: Best Cost = 3710.307
Iteration 494: Best Cost = 3710.307
Iteration 495: Best Cost = 3710.307
Iteration 496: Best Cost = 3710.307
Iteration 497: Best Cost = 3710.307
Iteration 498: Best Cost = 3710.307
Iteration 499: Best Cost = 3710.307
Iteration 500: Best Cost = 3710.307
```