Testbed Problem Optimization

Jon-Paul Boyd (P17231743)
School of Computer Science and Informatics
De Montfort University
United Kingdom

I. Introduction

This short report presents 4 popular testbed problems for optimization in Section II, implemented within the SOS (Stochastic Optimization Software) framework developed by Fabio Caraffini [1]. Section III introduces a single solution simulated annealing (SA) algorithm, implemented to optimize the 4 testbed problems. This SA optimizer is supported by developed helper functions to generate random solutions and perform toroidal correction, as detailed in Sections IV and V. Section VI presents the experiment definition, followed by details of experiment execution in Section VII. Experiment results and conclusion are presented in Section VIII.

II. Testbed Problem Implementation

As the author of [2] states, "*New optimization algorithms should be tested using at least a subset of functions with diverse properties so as make sure whether or not the tested algorithm can solve certain type of optimization efficiently*" [2]. Therefore, the remainder of this section presents the implementation of 4 functions which the SA algorithm will optimize. They are coded as Java classes in *SOS/src/benchmarks/BaseFunctions*.

A. *De Jong's Sphere Function*

The sphere (Fig. 1), or De Jong's function, is "*continuous, convex and unimodal*" [3]. The only minimum is global. Note that boundaries are set to [-5.12, 5.12]. The function definition is as follows:

$$f(x) = \sum_{i=1}^{n} x_i^2, \ -5.12 \leq x_i \leq 5.12$$

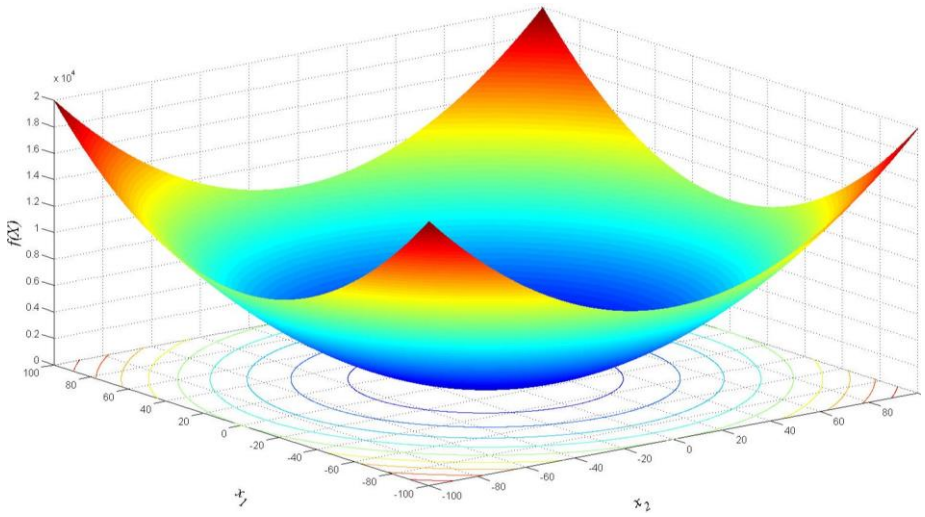

Fig. 1 De Jong's Sphere Function

Boundaries are set in the constructor of the function, followed by the implementation itself:

```
public DeJongSphere(int dimension){ super(dimension, new double[] {-5.12, 5.12});}
…
for (int i = 0; i < n; i++)
    y += Math.pow((x[i]),2);
```

B. *Schwefel's Function*

"*Schwefel's function (Fig. 2) is deceptive in that the global minimum is geometrically distant, over the parameter space, from the next best local minima. Therefore, the search algorithms are potentially prone to convergence in the wrong direction.*" [4]. The function definition is below, evaluated with the boundary [-500, 500].

$$f(x) = 418.9829d - \sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right), \ -500 \leq x_i \leq 500$$

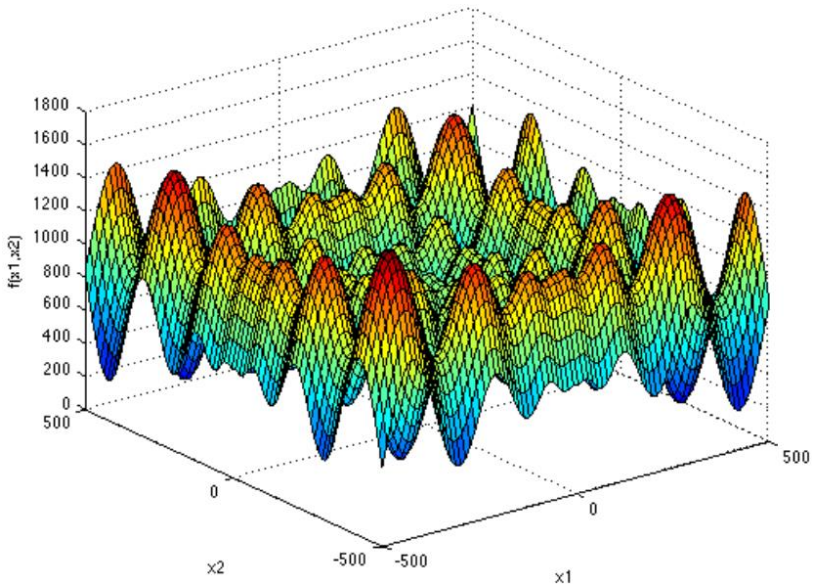

Fig. 2 Schwefel's Function

Again, boundaries are set in the constructor of the function, followed by the implementation itself:

```
public Schwefel(int dimension) { super(dimension, new double[] {-500, 500});}
...
for (int i = 0; i < n; i++)
    sum += x[i] * Math.sin(Math.sqrt(Math.abs(x[i])));
y = (418.9829 * n) - sum;
```

C. Rastrigin's Function

This function (Fig. 3) exhibits many local minima, thus is highly multimodal, with the location of minima regularly distributed [5]. The function is defined as follows:

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) \right], \; -5.12 \leq x_i \leq 5.12$$

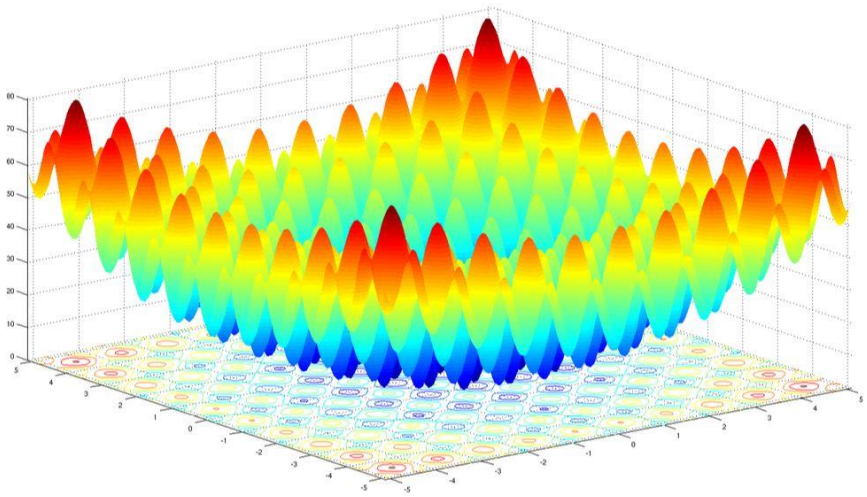

Fig. 3 Rastrigin's Function

The constructor definition of boundaries and implementation of the function itself is as below:

```
public Rastrigin(int dimension){ super(dimension, new double[] {-5.12, 5.12});}
…
for (int i = 0; i < n; i++)
    sum += ((x[i] * x[i]) - (10 * Math.cos(2 * Math.PI * x[i])));
y = (10 * n) + sum;
```

D. *Michalewicz's Function*

"*The Michalewicz function (Fig. 4) is a multimodal test function. The parameter m defines the "steepness" of the valleys or edges. Larger m leads to more difficult search. For very large m the function behaves like a needle in the haystack (the function values for points in the space outside the narrow peaks give very little information on the location of the global optimum).*" [6]. The function definition is as follows:

$$f(x) = -\sum_{i=1}^{n} \sin(x_i) \cdot \left[\sin(\frac{ix_i^2}{\pi})\right]^{2m}, \; i = 1:n, \; m = 10, \; 0 \leq x_i \leq \pi$$

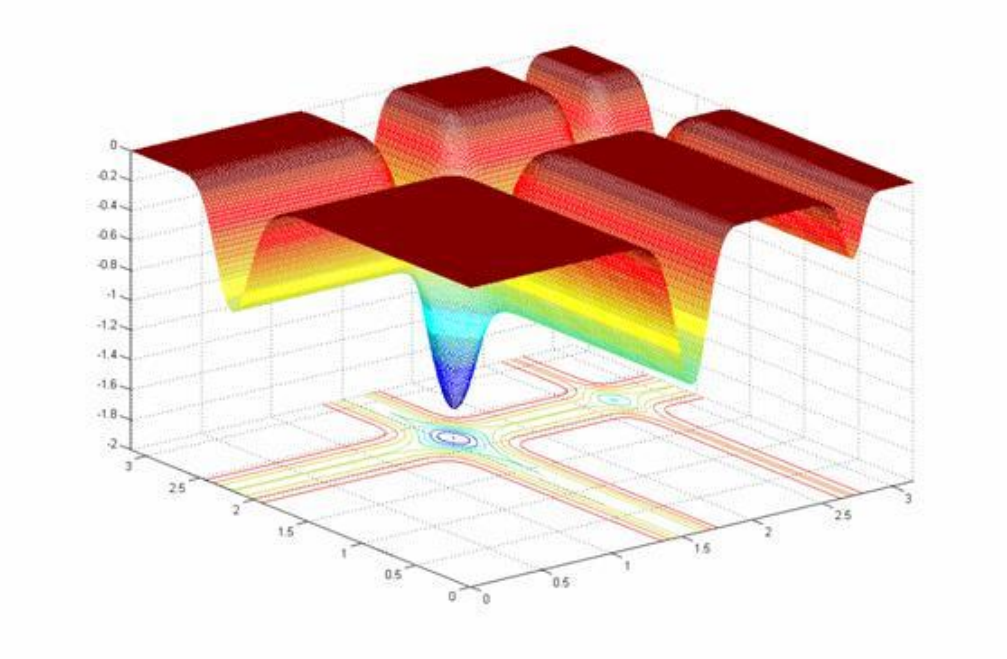

Fig. 4 Michalewicz's Function

The following presents the coded implementation where search space boundaries are defined in the class constructor:

```
public Michalewicz(int dimension){ super(dimension, new double[] {0.0, Math.PI});}
…
for (int i = 0; i < n; i++)
    sum -= Math.sin(x[i]) * Math.pow(Math.sin((i + 1) * (Math.sqrt(x[i]) / Math.PI)), 2 * m);
y = sum;
```

III. Simulated Annealing Single Solution Optimizer Implementation

A simulated annealing (SA) stochastic algorithm is now implemented to optimize the 4 testbed problems detailed in the previous section. As described by the author of [7], "*the term 'annealing' refers to the process in which a solid, that has been brought into liquid phase by increasing its temperature, is brought back to a solid phase by slowly reducing the temperature in such a way that all the particles are allowed to arrange themselves in a perfect crystallized state. Such a crystallized state represents the global minimum of certain energy function*". It is coded as a Java class in directory *SOS/src/algorithms*.

Two configuration parameters are of particular interest in the design of this optimizer. The first is the initial maximum temperature from where the process of cooling begins. Initially this maximum was set heuristically through trial and error. However, as guided by [7], "*One common way to compute a good initial temperature is to compute the values of the objective function for a set of models selected at random; then the energy variations among all of them are computed and a value of To is estimated such that, according to the acceptance criterion, the maximum energy variation is accepted with some probability close to the unit.*". Therefore, method *setInitialTemp* is designed to sample the bounded search space 10% of the number of *maxEvaluations* iterations defined then take the 95th percentile as the starting maximum temperature. With such a design a reasonable starting point should eliminate wasted random search and computing resource had too high a starting temperature been used. Furthermore, it should provide a sizeable search space that would otherwise have been restricted had too low a temperature been specified. The second parameter of interest is *coolingRate* which determines how the temperature is decreased though each iteration. After trialling various cooling schedule methods including logarithmic, quadratic, geometric and exponential where temperature reduction and fitness was observed, it was found the simple method decaying current temperature through multiplication by a fixed cooling factor (0.9998) worked best so as to cool slowly enough for sufficient search and minimisation of fitness function.

Note that variable *temperatureThreshold* sets a temperature lower limit. A new fitness value is always accepted if lower than the current fitness value. However, this is not always the case, and so the local area is stochastically searched. Loss is measured as the delta between current and new fitness. A probability is calculated as the exponential value of the loss divided by temperature. The greater the difference between current and new fitness, or the lower the current temperature, then the probability is lower when compared with random value [0, 1] that the stochastic fitness is accepted. This thermal jumping ability helps avoid getting trapped in local minima.

```
public class SimulatedAnnealing extends Algorithm
{
    @Override
    public FTrend execute(Problem problem, int maxEvaluations) throws Exception
    {
        double temperature = setInitialTemp(problem, maxEvaluations);  // Set max temp from rand sol subset
```

```
        double coolingRate = getParameter("coolingRate");
        double temperatureThreshold = 1;

        int problemDimension = problem.getDimension();
        double[][] bounds = problem.getBounds();

        FTrend FT = new FTrend();

        // Current solution and fitness
        double[] xcb;
        double fcb;

        // New solution and fitness
        double[] xnew;
        double fnew;

        double loss;
        double probability;

        int i = 0;
        // Begin with random solution configuration
        if (initialSolution != null)
        {
            xcb = initialSolution;
            fcb = initialFitness;
        }
        else
        {
            xcb = generateRandomSolution(bounds, problemDimension);
            fcb = problem.f(xcb);
            i++;
        }

        FT.add(0, fcb); // Store the initial guess

        //Decrease until budget comnsumed or cool until minimum temperature reached
        while ((i < maxEvaluations) && (temperature > temperatureThreshold)) {  // Prevent t < tmin
            i++;
            xnew = generateRandomSolution(bounds, problemDimension);
            xnew = Misc.toro(xnew, bounds);  // Always saturate solution within search space
            fnew = problem.f(xnew);

            loss = fcb - fnew;
            probability = Math.exp(loss / temperature);

            // Pairwise comparison - if change in energy is decreasing then accept the new solution
            //                     - or move to random new point nearby
            if ((fnew < fcb) || (RandUtils.random() < probability)) {
                FT.add(i, fnew);
                fcb = fnew;
                xcb = xnew;
            }

            // Exponential cooling schedule implemented as current temperature multiplied by fixed factor
            temperature = temperature * coolingRate;

        }
        finalBest = xcb;
        return FT; // Return the fitness trend

}

private double setInitialTemp (Problem problem, int maxEvaluations)  throws Exception {
    int problemDimension = problem.getDimension();
    double[][] bounds = problem.getBounds();
    int i = 0;

    // Sample 10% of search space to determine starting max temperature
    int maxIter = (int)(maxEvaluations * 0.1);

    double[] fcurrSubset = new double[maxIter + 1];
    double[] xcurr;
    double fcurr;

    // Build up sample subset of temps
    while (i < maxIter) {
        i++;
        xcurr = generateRandomSolution(bounds, problemDimension);
        fcurr = problem.f(xcurr);
        fcurrSubset[i] = fcurr;
    }

    // Use 95th percentile as starting max temperature
    return StatUtils.percentile(fcurrSubset, 95);
}
```

IV. Random Solution Generation

The method *generateRandomSolution* implemented in *SOS/src/utils.algorithms/Misc* generates a random solution point within the specified bounds of the search space and is executed from within the SA implementation. Note the code for general and hyper-parallel piped variants is effectively the same.

```
public static double[] generateRandomSolution(double[][] bounds, int n)
{
    double[] r = new double[n];
    for (int i = 0; i<n; i++)
        r[i]= (int)(Math.random() * (bounds[i][1]-bounds[i][0])) + bounds[i][0];
    return r;
}
```

V. Toroidal Correction

The toroidal correction saturation scheme of the random solution is implemented in *SOS/src/utils.algorithms/Misc* as below. Note the code for general and hyper-parallel piped variants is effectively the same.

```
public static double[] toro(double[] x, double[][] bounds)
{
    double[] xCor = new double[x.length];
    double[] xOut = new double[x.length];

    for (int i = 0; i < x.length; i++) {
        xCor[i] = (x[i] - bounds[i][0]) / (bounds[i][1] - bounds[i][0]);
        if (xCor[i] > 1) {
            xCor[i] = xCor[i] - fix(xCor[i]);
        } else if (xCor[i] < 0) {
            xCor[i] = 1 - Math.abs(xCor[i] - fix(xCor[i]));
        }
        xOut[i] = bounds[i][0] + (xCor[i] * (bounds[i][1] - bounds[i][0]));
    }
    return xOut;
}
```

VI. Experiment Definition

As the author of [2] states, it is "*important to validate new optimization algorithms and to compare the performance of various algorithms*". Therefore, Java class *Test* is created in *SOS/src/experiments*, defining the experiment "*CIO_Task2"* with fitness function test results per testbed problem/dimension stored in the same named sub folder of "*results*". In addition to the SA optimizer, the ISPO (Intelligent Single Particle Optimizer) and CMAES (Co-variance Matrix Adaptation Evolution Strategy) optimizers have been added for result comparison. Each optimizer will try to solve the 4 testbed problems implemented. There will be 200 runs for each single problem/optimizer combination, as defined by setNrRuns(200).

```
package experiments;

import algorithms.SimulatedAnnealing;
import algorithms.CMAES;
import algorithms.ISPO;
import benchmarks.BaseFunctions;
import interfaces.Experiment;
import interfaces.Algorithm;
import interfaces.Problem;

public class Test extends Experiment
{
    public Test(int probDim)
    {
        super(probDim,"CIO_Task2");
        setNrRuns(200);

        Algorithm a;  // A generic optimiser
        Problem p;    // A generic problem

        a = new SimulatedAnnealing();
        a.setParameter("coolingRate", 0.9998);
        add(a); //add it to the list

        a = new ISPO();
        a.setParameter("p0", 1.0);
        a.setParameter("p1", 10.0);
        a.setParameter("p2", 2.0);
        a.setParameter("p3", 4.0);
        a.setParameter("p4", 1e-5);
        a.setParameter("p5", 30.0);
        add(a); //add it to the list

        a = new CMAES();
        add(a);

        // Add DeJong Sphere problem to test
        p = new BaseFunctions.DeJongSphere(probDim);
```

```
        p.setFID("DeJongSpehere");
        add(p);

        // Add Schwefel problem to test
        p = new BaseFunctions.Schwefel(probDim);
        p.setFID("Schwefel");
        add(p);

        // Add Rastrigin problem to test
        p = new BaseFunctions.Rastrigin(probDim);
        p.setFID("Rastrigin");
        add(p);

        // Add Michalewicz problem to test
        p = new BaseFunctions.Michalewicz(probDim);
        p.setFID("Michalewicz");
        add(p);
    }
}
```

VII. Experiment Execution

The experiment runtime profiles are defined in Java class *SOS/src/RunExperiments*. Here the experiment configured as shown in Section VI is set to run for 10, 30 and 50 dimensions.

```
public static void main(String[] args) throws Exception
{
    resultsFolder();

    Vector<Experiment> experiments = new Vector<Experiment>();////!< List of problems

    //@@@ MODIFY THIS PART @@@
    experiments.add(new Test( 10));
    experiments.add(new Test( 30));
    experiments.add(new Test( 50));
    //@@@@@@
…
```

VIII. Experiment Results And Conclusion

Tables of results comparing performance of the SA, ISPO and CMAES optimizers when solving each testbed problem with 10, 30 and 50 dimensions are presented. The optimizers execute each problem 200 times, with the Avg and standard deviation (StDev) of the fitness values given. Additionally, the result of the Wilcoxon statistical analysis signed test is shown. It pairs observations, in this case the collected fitness results of each optimizer and infers a + where the first observation outperforms the second of the pair, otherwise a -. This testing considers SA the reference optimizer and first observation of any pair, hence a + indicates SA is better and – that SA poorer.

Considering De Jong's sphere, the SA optimizer shows inferior performance to ISPO and CMAES in all dimensionality tests, with performance increasingly degraded as dimensionality increases (higher Avg and StDev). This is in contrast to ISPO and CMAES which are far more stable regardless of dimensionality. Given that the sphere has only one minimum that is global, it is hypothesised that either SA is converging (cooling) too slowly or jumping around too much between solutions in this highly convex function.

Evaluating performance of Schwefel function optimization, SA outperforms both ISPO and CMAES in the 10 dimensionality test, evidence that it is capable of escaping local minima thanks to its stochastic thermal jumping ability. However, in the higher dimension tests performance significantly degrades whilst the other optimizers remain both comparable and stable. It is suggested that in higher dimensional, more complex search space a greater number of fitness values and thermal jumps are being rejected.

SA responds well to the Rastrigin function, outperforming ISPO and CMAES by quite some margin in all dimension tests. In the tested configuration SA appears to manage well the highly multi-model, uniformly distributed nature of this testbed problem, again supported by its thermal jumping capability.

In all dimensionality tests SA performs poorer when optimizing Michalewicz's function. An interpretation of the results seems to suggest that although degradation of performance is less severe when dimensionality is increased compared with other testbed problems, SA is however stalling in some local valley.

In conclusion, SA seems to perform best when optimizing non-convex, lower dimensionality but highly multi-modal, non-plateauing problems, when cooled gradually. It is straight-forward to implement and comprehend but does require effort in fine tuning maximum starting temperature and cooling rate.

TABLE I
Optimized testbed problems with 10 dimensions

Function	SA Optimizer		ISPO Optimizer			CMAES Optimizer		
	Avg	StDev	Avg	StDev	Wilcoxon	Avg	StDev	Wilcoxon
De Jong	1.410e+01	± 3.777e+00	5.153e-35	± 3.475e-35	-	5.219e-04	± 7.362e-03	-
Schwefel	1.760e+03	± 1.670e+02	1.987e+03	± 3.818e+02	+	1.922e+03	± 3.598e+02	+
Rastrigin	4.095e+01	± 3.633e+00	8.462e+01	± 2.538e+01	+	8.716e+01	± 2.523e+01	+
Michalewicz	-1.033e+00	± 6.365e-01	-4.762e+00	± 3.513e-01	-	-4.185e+00	± 6.724e-01	-

TABLE II
Optimized testbed problems with 30 Dimensions

Function	SA Optimizer		ISPO Optimizer			CMAES Optimizer		
	Avg	StDev	Avg	StDev	Wilcoxon	Avg	StDev	Wilcoxon
De Jong	1.123e+02	± 1.145e+01	1.580e-34	± 6.196e-35	-	2.621e-01	± 2.608e+00	-
Schwefel	8.218e+03	± 2.937e+02	5.968e+03	± 7.166e+02	-	5.816e+03	± 6.909e+02	-
Rastrigin	1.940e+02	± 1.081e+01	2.649e+02	± 4.369e+01	+	2.716e+02	± 4.369e+01	+
Michalewicz	-2.866e+00	± 9.540e-01	-1.939e+01	± 1.179e+00	-	-1.633e+01	± 1.908e+00	-

TABLE III
Optimized testbed problems with 50 Dimensions

Function	SA Optimizer		ISPO Optimizer			CMAES Optimizer		
	Avg	StDev	Avg	StDev	Wilcoxon	Avg	StDev	Wilcoxon
De Jong	2.371e+02	± 1.462e+01	2.837e-34	± 1.007e-34	-	1.704e+00	± 6.463e+00	-
Schwefel	1.531e+04	± 4.147e+02	9.852e+03	± 9.046e+02	-	9.579e+03	± 8.386e+02	-
Rastrigin	3.733e+02	± 1.482e+01	4.431e+02	± 5.604e+01	+	4.478e+02	± 5.945e+01	+
Michalewicz	-4.866e+00	± 1.175e+00	-3.633e+01	± 1.400e+00	-	-2.848e+01	± 2.502e+00	-

REFERENCES

[1] F. Caraffini, “Fabio Caraffini - Research,” [Online]. Available: https://sites.google.com/site/facaraff/research.

[2] X. S. Yang, Test problems in optimization, in: Engineering Optimization: An Introduction with Metaheuristic Applications, John Wiley & Sons, 2010.

[3] P. Gdanska, “GEATbx: Examples of Objective Functions - De Jong's function,” [Online]. Available: http://www.pg.gda.pl/~mkwies/dyd/geadocu/fcnfun1.html.

[4] P. Gdanska, “Schwefel's Function,” [Online]. Available: http://www.geatbx.com/docu/fcnindex-01.html#P150_6749.

[5] P. Gdanska, “Rastrigin's Function,” [Online]. Available: http://www.geatbx.com/docu/fcnindex-01.html#P140_6155.

[6] P. Gdanska, “Michalewicz's function,” [Online]. Available: http://www.geatbx.com/docu/fcnindex-01.html#P204_10395.

[7] R. Banchs, “Simulated Annealing,” The University of Texas, Austin, 1997.