Kudu Herd Optimization

Julien Boelaert
CES
Universite Paris 1 Pantheon-Sorbonne
Maison des Sciences Economiques
106-112 boulevard de l'Hopital, 75013 Paris
julien.boelaert@gmail.com

***Abstract*—This work proposes a new and simple algorithm for unconstrained numeric optimization over continuous spaces. A population of candidate solutions styled as a herd of kudus performs a succession of jumps through the search space in order to find the best solution (the kudu is a species of antelope). The logic of this algorithm is quite different from that of most population-based algorithms, as the individual solutions are moved together in a herd-like fashion. Performance comparisons are conducted with the Artificial Bee Colony, Differential Evolution, the Genetic Algorithm and Particle Swarm Optimization on benchmark functions. The kudu herd seems to perform well in the early stages and on high-dimensional problems.**

I. Introduction

Numerical optimization is an active research field with a wide range of applications. Ever since the emergence of evolutionary computation in the 1960s, much progress has been made in stochastic optimization techniques that make few assumptions about the function being optimized, in particular not requiring it to be continuous or differentiable. Exciting recent developments include Particle Swarm Optimization (PSO) [9], the Artificial Bee Colony algorithm (ABC) [8], both based on nature-inspired behaviour, and Differential Evolution (DE) [13]. These evolutionary algorithms have proved to yield good results on benchmark and real-life problems.

The present contribution introduces a new algorithm named Kudu Herd Optimization. It can be thought of as mimicking the behaviour of a herd of kudus that jump their way through a solution space to find the optimal point. The kudu is a species of antelope present in sub-Saharan Africa, that mostly lives in herds and is known for its ability to jump high and far [3]. The kudus were not the original inspiration for the algorithm, but provide a convenient and entertaining way of presenting its workings and logic. The algorithm tries to find an optimal point by iteratively changing a population of candidate solutions. However it does not rely on evolutionary principles or swarm intelligence, but rather on herd-like behaviour with centralized decisions. Results indicate that it performs well on a number of classic benchmark problems taken from the literature.

The rest of this paper is organized as follows. Section 2 describes the KHO algorithm, first through an illustrative story, then more formally. Section 3 presents the experimental protocol and the results of performance comparison between KHO, ABC, DE, Genetic Algorithm (GA), and PSO. Section 4 contains some concluding comments.

II. Description of the algorithm

A. Illustrative story

The logic of KHO is best understood through a story. Let us state first that the behaviour of the kudu herd we describe is purely fictional - apart from the fact that they are often found in herds and that they are good jumpers, we claim no knowledge of kudu behaviour. Let us then imagine a herd of kudus jumping around their habitat in search of the location containing the best food - a kudu finding itself in any point in space can give it a real-valued mark, indicating food quality. Most kudus in our herd are fairly unimaginative, and they all follow one intelligent leader-kudu in their jumps. The herd moves in the following way:

- The leader jumps to a given location, and all the other kudus jump to random positions around the leader.
- The kudus then report their new positions and the corresponding quality to the leader. Based on this information, the leader determines the direction of its next jump (by computing the covariance of a function of picnic-quality with the position coordinates).
- The leader's jump distance grows smaller if two consecutive jumps are in opposite directions, otherwise it grows larger.
- Several jumps are performed in this fashion, and the size of the scatter region around the leader can be reduced over time to exploit information from narrower regions.
- The herd remembers the single best location it has found so far.

B. Formal description and pseudo-code

Formally, we try to find the point that minimizes a real-valued cost function over a given bounded D-dimensional real-valued search space:

$$\arg\min_x \text{cost}(x), x \in S$$
$$S = [lb_1, ub_1] \times \ldots \times [lb_D, ub_D]$$

For this purpose we use a population of P kudus (D-dimensional vectors representing candidate solutions), of which one is the leader. The algorithm iterates over four steps: place the leader in the search space, randomly place

the other kudus in an area around the leader, evaluate the cost function for each kudu, and use these evaluations to compute the leader's next position.

Introducting some notations, let:

- $\overrightarrow{leader}$ be an D-dimensional vector containing the leader's position,
- A be a $P \times D$ matrix whose first row is $\overrightarrow{leader}$ and whose remaining rows contain the other kudus' positions,
- $\overrightarrow{rank}$ be a P-dimensional vector containing the cost-ranking of each of the kudus,
- $\overrightarrow{jump}$ be an D-dimensional vector giving the direction of the leader's jump,
- $jlength$ be a scalar value representing the leader's jump's length,
- $scatter$ be a scalar value controlling how close to the leader the kudus will be scattered,
- η^+ and η^- be scalar values used to automatically lengthen or shorten the jumps,
- $mlength$ be a scalar value representing the minimal allowed jump length.

Then in pseudo-code the KHO algorithm operates as follows:

1) Initialization
   a) Initialize the leader to a random point in the search space.
   b) Initialize the $(P-1)$ other kudus to random points in a region around the leader, with parameter $scatter$ controlling the size of this region.
   c) Evaluate the cost function at each of the P points.
   d) Rank the costs and store the result in $\overrightarrow{rank}$.
   e) Store the lowest cost and the associated position.
   f) Compute the jump direction according to the following formula:

$$\forall i \in [1, D], \overrightarrow{jump_i} = \mathrm{cov}(\overrightarrow{rank}, A_{.i}) \qquad (1)$$

   where $\overrightarrow{jump_i}$ denotes the i-th element of the jump vector and $A_{.i}$ denotes the i-th column of matrix A.
   g) Initialize jump-length variable $jlength$ to the maximal distance between the leader and the other kudus.
2) Loop (until termination criterion is met)
   a) Update the leader's position according to the following formula:

$$\overrightarrow{leader^{t+1}} = \overrightarrow{leader^{t}} - \frac{jlength^t}{\|\overrightarrow{jump^t}\|}\overrightarrow{jump^t}$$

   b) Update the other kudus' positions by randomly placing them in a region around the leader, the size of which is controlled by parameter $scatter$.
   c) Evaluate the cost function at each of the P points.
   d) Rank the costs and store the result in $\overrightarrow{rank}$.
   e) Store the lowest cost and the associated position if it is lower than the stored best.
   f) Compute the jump direction according to formula (1).
   g) Update the jump-length variable: if the new jump is made a direction opposite to that of the last jump (*ie* if $\overrightarrow{jump^{t+1}} \cdot \overrightarrow{jump^t} < 0$) then multiply $jlength$ by η^-, else multiply it by η^+. If this makes $jlength$ smaller than $mlength$, set it to $mlength$.
   h) Update the $scatter$ parameter.

C. Parameters and taxonomy

The algorithm essentially takes 3 parameters: the population size P, the scatter-range $scatter$, and the minimal jump-length $mlength$. Their role is easily understood: P is the number of cost-function evaluations at each iteration, which should affect the quality of jump direction (more evaluations at each iteration means a better "understanding" of the topology of the cost-surface around the leader) and the intensity of exploitation of each visited area (more evaluations give a better probability to pick a good point in the local search-area). The scatter-range controls "how local" the search is at each iteration: a lower $scatter$ value will let the random evaluations occur in a narrower region around the leader. The minimum jump-length is used to avoid convergence to local optima. In section 3 we try to apprehend the influence of P, $scatter$ and number of iterations on performance.

Parameters η^+ and η^- are directly inspired by those of Riedmiller and Braun's RPROP algorithm [12] for the training of feedforward neural networks. The idea, paraphrasing the original article, is that two consecutive jumps in opposite directions (which in KHO is detected by a negative dot product) indicate that the last jump was too long and the algorithm has jumped over a local minimum; jump-length is then decreased by factor η^-. Otherwise, jump-length is slightly increased (by factor η^+) in order to accelerate convergence in shallow regions. Although our setting is quite different, we used the original values of both parameters, $\eta^- = 0.5$ and $\eta^+ = 1.2$. No attempt has been made as yet to assess their influence on performance.

We chose to use the rank of costs instead of costs to compute the jump direction. This makes the algorithm invariant to any increasing transformation of the cost function. Preliminary results seem to indicate that using ranks instead of cost does not significantly affect performance on benchmark functions. Other choices are of course possible, for instance using log-rank, exp-rank or squared rank to give more importance to lower or higher costs, and this choice might affect performance. Another discussable choice concerns the random placements around the leader. For the experiments of section III the kudus were uniformly distributed inside a hyperparallelepiped centered on the leader, according to the following formula:

$$\forall (i,j) \in [2, P] \times [1, D],$$
$$a_{ij} = \overrightarrow{leader}_j + (r_{ij} - 0.5) \times scatter \times (ub_j - lb_j)$$

where a_{ij} denotes element (i, j) of matrix A, $\overrightarrow{leader}_j$ is the leader-vector's j-th element, $scatter$ is a scalar value chosen

Table I
BENCHMARK FUNCTIONS F1-F10 DEFINITION, RANGE AND CHARACTERISTICS

Function	Name	Definition	Range	Unimodal	Separable	Easily optimized dimension by dimension
F_1	Ackley	$20 + e - 20\exp(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}) - \exp(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i))$	$[-32, 32]^D$	N	Y	Y
F_2	Bohachevsky	$\sum_{i=1}^{D-1}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7)$	$[-15, 15]^D$	Y	N	N
F_3	Griewank	$\sum_{i=1}^{D}\frac{x_i^2}{4000} - \prod_{i=1}^{D}\cos(\frac{x_i}{\sqrt{i}}) + 1$	$[-600, 600]^D$	N	N	N
F_4	Rastrigin	$\sum_{i=1}^{D}(x_i^2 - 10\cos(2\pi x_i) + 10)$	$[-5, 5]^D$	N	Y	Y
F_5	Rosenbrock	$\sum_{i=1}^{D-1}(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2)$	$[-100, 100]^D$	N	N	Y
F_6	Schaffer	$\sum_{i=1}^{D-1}(x_i^2 + x_{i+1}^2)^{0.25}(\sin^2(50(x_i^2 + x_{i+1}^2)^{0.1}) + 1)$	$[-100, 100]^D$	Y	N	Y
F_7	Schwefel 1.2	$\sum_{i=1}^{D}(\sum_{j=1}^{i} x_j)^2$	$[-65.536, 65.536]^D$	Y	N	N
F_8	Schwefel 2.21	$\max_i\{\lvert x_i\rvert, 1 \le i \le D\}$	$[-100, 100]^D$	Y	N	N
F_9	Schwefel 2.22	$\sum_{i=1}^{D}\lvert x_i\rvert + \prod_{i=1}^{D}\lvert x_i\rvert$	$[-10, 10]^D$	Y	Y	Y
F_{10}	Sphere	$\sum_{i=1}^{D} x_i^2$	$[-100, 100]^D$	Y	Y	Y

in $[0, 1]$, ub_j and lb_j are respectively the upper and lower bounds of dimension j, and r_{ij} is a random value uniformly drawn from $[0, 1]$. A better choice might be to generate normal deviates from $\overrightarrow{leader}$.

The KHO algorithm uses a population of candidate solutions that it changes and evaluates at each iteration. This property, which is shared by population-based algorithms such as ABC, DE, GA and PSO, makes parallelization of these algorithms easy, a very desirable property when evaluations are time-consuming. Whether KHO can be called a population-based algorithm is, however, discussable. In KHO the individual solutions that make up the population have no autonomy, as all main decisions are centralized (jump direction and length are computed by the leader using the information provided by the whole population). At each iteration these individuals are randomly placed around the leader, so that contrary to the algorithms cited above different individuals never simultaneously explore wildly different regions. Furthermore their new coordinates do not depend on previous individual coordinates. This lack of autonomy sets KHO apart from most population-based algorithms, as its population moves through the solution-space in a herd-like fashion. This and the lack of inheritance in the transformation process also means that KHO does not belong to the family of evolutionary algorithms, and does not make use of swarm intelligence (as defined in [10]).

III. EXPERIMENT AND RESULTS

A. Experimental design

In order to test the performance of the KHO algorithm we use the unshifted expression of 10 classic benchmark functions as specified in [6]. As can be seen in table I their expressions and properties differ, but they are all scalable to any number D of dimensions, and all offer at least one global minimum of value 0.

Performance was compared to that of ABC, DE, GA, PSO and a random search. Comparing optimization algorithms in a rigorous way is a notoriously difficult and time-consuming exercise, but we have tried to make the results as objective and reproducible as possible. Each algorithm was tried with varying population size (20, 50, 100, 200) and number of iterations (50, 100, 200, 1000), and each cost function was tried with varying number of dimensions (10, 20, 50, 100, 200). Each combination was run 100 times. The algorithms are compared through cost medians and standard deviations, on a basis of same problem dimension, population size and number of iterations. Lack of space forbids the presentation of all the detailed results, but a summary is given in subsection III-C.

B. Algorithms and settings

All trials were run using R version 2.13.1. Following is a brief description of the algorithms and settings used.

1) Artificial Bee Colony: The ABC algorithm was proposed by Karaboga in 2007 [8] as an optimization technique inspired by the foraging behaviour of honey bees. Candidate solutions are called food sources. The population is divided into employed bees and onlooker bees. In each iteration each employed bee tries to find one better food source than its previous one, applying a greedy selection if it succeds. Each onlooker bee is then directed towards one of the existing food sources, with a probability depending on the fitness of these sources, and tries to improve it, again applying a greedy selection. Each food source can only be exploited a given number of times, after which it is abandoned and replaced by a new random source.

The algorithm was implemented by plugging the C code found on the official ABC website [1] into the Rcpp package [5]. The $limit$ parameter, controlling the number of maximal exploitations per food source, was set to 100 for all runs.

2) Differential Evolution: The DE algorithm, introduced by Storn and Price in 1997 [13], is an evolutionary algorithm in which the evolution of individual agents is based on sums and

differences between coordinates of existing agents. Its most simple expression is the following: at each iteration, for each member of the population x, three existing agents a,b and c are chosen at random, and a new candidate solution is created by crossover between x and $a+F\times(b-c)$, using a crossover probability CR and ensuring that at least one coordinate of x is conserved in the new candidate. A greedy selection is then applied between x and the new candidate solution.

The tests were run using the R package DEoptim version 2.1-2 [11]. We used the classical DE/rand/1/bin strategy, and the package's standard values for F and CR, respectively set to 0.8 and 0.5.

3) Genetic Algorithm: GA has a long history of usage in many different settings since its introduction by Holland in 1975 [7]. The basic principle is to evolve children solutions from a population of parent solutions by the use of selection, crossover and mutation operators.

We used the real-vector version of GA implemented in function rbga of R package genalg version 0.1.1 [14]. Except for the population size and number of iterations, we used the package's standard parameters: mutation probability of $1/(D+1)$ where D is the number of dimensions, and elitism of about 20% of the population size.

4) Particle Swarm Optimization: PSO was introduced by Kennedy and Eberhart in 1995 [9] and is inspired by the flocking behaviour of birds and fish. A population of candidate solutions called particles is flown around the search space in the following way. At each iteration, each particle's position is changed by adding to it a random velocity vector which is a function of the previous velocity, of the difference vector between the particle's present position and its remembered best position, and of the difference vector between the particle's present position and the remembered best position of a set of informant particles.

We used the psoptim function provided by R package pso [2], which follows the "standard pso 2007" specification by Clerc et al. [4]. Except for the population size and number of iterations, the default settings were used: exploitation constant of $1/(2\log(2))$, local and global exploitation constants both set to $0.5+\log(2)$, random order of particle processing, no restarting, no clamping of velocity, and average percent of informants for each particle of $1-(1-1/popsize)^3$.

5) Random Search: In order to detect bad performance we included a random search algorithm, which is simply the best solution found among $iter \times popsize$ random guesses uniformly distributed over the search space.

6) KHO: The algorithm was run as described in section II-B, with a minimal jump length of $1e-20$. It should be noted that for a same population size and number of iterations the KHO algorithm performs more operations than the other algorihtms, as for each iteration it has to compute covariances of the solution ranks with the population's coordinates.

The algorithm was coded in C++ and integrated in the R environment using the Rcpp package. Five different $scatter$ parameters were tried: in four implementations the parameter is constant over all the iterations (0.3, 0.1, 0.01, 0.001) and in the fifth implementation the parameter is linearly decreased from 0.1 to 0.001 over the iterations.

C. Results

Tables II through V show some of the results obtained on the benchmark functions for 10 and 200 dimensions. These tables reflect the main results of the experiment: *KHO finds a relatively good solution very fast* (as can be seen in tables II and IV it often gets much better results than the other algorithms after 50 iterations), but seems *unable to make significant progress in subsequent iterations*. After many iterations (typically 200 in low dimensions and 1000 in high dimensions) the other algorithms are able to catch up with it and attain better solutions. This double result is observed in nearly all settings for functions F_1, F_2, F_3, F_5, F_6, F_8 and F_{10}. It is unclear whether KHO gets stuck in local optima or whether it is unable to exploit the good regions it has found to the fullest. The example of function F_{10}, which is unimodal and has no local "traps", and on which, in low dimensions, KHO is bested by PSO and ABC for 200 iterations or more, indicates that good exploitation is indeed an issue.

The results clearly indicate that the $scatter$ parameter plays a crucial role in KHO's performance, and that the optimal value depends on the considered problem. An extreme example of this is function F_1, on which KHO performs well when the parameter is set to 0.1 or linearly decreased from 0.1 to 0.001, but gives very bad results - equivalent to or worse than those of a random search - when it is set to 0.01 or 0.001. On the other hand, on F_{10} the best parameter is always the lowest, 0.001. Letting the parameter decrease with each iteration seems to be a good choice in many settings. An auto-adaptive scheme for this parameter might give good results.

KHO has relatively bad results on functions F_7 and F_9. Interestingly, on function F_4 KHO obtains very bad results in 10 to 50 dimensions (similar to random search or worse), but gets much better in 100 and 200 dimensions (again finding good solutions comparatively fast, but being bested after many iterations). More generally, *the algorithm performs comparatively better in high dimensions*: it still finds good solutions after only a small number of iterations, and keeps its relative advantage to other algorithms much longer than in lower dimensions (most algorithms struggle with high dimensions and require a great number of iterations to attain good solutions).

IV. Conclusion

Kudu herd optimization is a simple algorithm that seems to perform well on many difficult problems. Performance comparisons with well-known population-based algorithms give encouraging results: KHO seems to find good solutions faster than other agorithms, and to respond comparatively well to the problem of scalability. It seems, however, to suffer from a lack of exploiting behaviour when run with a large number of iterations.

Future research could improve the algorithm in many directions. We suspect that using normal deviations instead

Table II

MEDIAN COST OBTAINED BY ABC, DE, GA, PSO, KHO AND RANDOM SEARCH ON THE TEST SUITE, 100 TRIALS (FIVE DIFFERENT CHOICES FOR KHO'S SCATTER PARAMETER : FIXED AT 0.3, 0.1, 0.01, AND 0.001, AND LINEARLY DECREASED FROM 0.1 TO 0.001) DIM=10, POPSIZE= 50, ITER= 50 (STANDARD DEVIATIONS IN PARENTHESES, BEST MEDIAN IN BOLDFACE).

	f1	f2	f3	f4	f5
ABC	5.042 (1.645)	1.712 (1.133)	0.9627 (0.2855)	**10.76** (3.433)	1172 (6766)
DE	8.543 (0.8296)	23.58 (5.953)	3.54 (0.9653)	29.01 (4.866)	1.577e+06 (1.274e+06)
GA	10.31 (1.359)	48.52 (17.22)	7.829 (2.98)	23.52 (5.906)	1.422e+07 (2.02e+07)
PSO	3.639 (0.4723)	6.328 (1.156)	1.239 (0.1309)	40.54 (7.666)	2.213e+04 (2.443e+04)
Random	17.76 (1.042)	380.9 (83.04)	59.07 (12.65)	78.2 (9.346)	5.822e+08 (2.918e+08)
KHO 0.3	2.674 (0.5356)	4.903 (1.698)	1.074 (0.06662)	35.46 (10.16)	1.859e+05 (4.627e+05)
KHO 0.1	1.016 (5.283)	1.217 (0.562)	0.987 (0.07092)	84.21 (26.1)	1.033e+05 (2.977e+05)
KHO 0.01	19.58 (0.497)	5.209 (1.526)	0.08589 (0.06632)	84.94 (25.36)	**868.6** (7.693e+04)
KHO 0.001	19.53 (0.3751)	6.136 (1.762)	**0.07859** (0.293)	91.04 (23.19)	3887 (1.311e+05)
KHO 0.1-0.001	**0.2189** (7.238)	**0.2944** (0.2159)	0.4287 (0.1646)	78.84 (24.58)	6134 (1.895e+05)

	f6	f7	f8	f9	f10
ABC	**10.16** (1.415)	1177 (492.9)	28.33 (7.541)	**0.3464** (0.2068)	0.819 (5.719)
DE	12.56 (1.384)	1256 (378.7)	24.75 (4.241)	3.745 (0.6137)	309.2 (110.4)
GA	11.87 (2.02)	827.6 (371.8)	24.77 (5.08)	4.827 (1.224)	753.9 (396.3)
PSO	14.83 (2.538)	**171.9** (104.3)	5.09 (1.158)	1.366 (0.3425)	24.99 (12.02)
Random	22.84 (3.386)	2739 (657.9)	43.4 (5.171)	22.95 (3.975)	6463 (1473)
KHO 0.3	16.15 (4.581)	1559 (957.2)	0.9454 (0.3826)	8.75 (7.622)	8.853 (5.141)
KHO 0.1	19.6 (4.64)	388 (271.3)	0.3334 (0.5184)	29.59 (24.23)	1.015 (0.6104)
KHO 0.01	14.5 (5.1)	493 (1025)	45.42 (14.08)	63.53 (229.7)	0.01107 (0.006097)
KHO 0.001	14.44 (10.8)	956.4 (9190)	73.32 (12.44)	51.15 (28.71)	**0.0002638** (0.001495)
KHO 0.1-0.001	14.98 (4.579)	236.2 (462.2)	**0.08873** (5.924)	32.74 (17.65)	0.08066 (0.08757)

Table III

MEDIAN COST OBTAINED BY ABC, DE, GA, PSO, KHO AND RANDOM SEARCH ON THE TEST SUITE, 100 TRIALS (FIVE DIFFERENT CHOICES FOR KHO'S SCATTER PARAMETER : FIXED AT 0.3, 0.1, 0.01, AND 0.001, AND LINEARLY DECREASED FROM 0.1 TO 0.001) DIM=10, POPSIZE= 50, ITER= 1000 (STANDARD DEVIATIONS IN PARENTHESES, BEST MEDIAN IN BOLDFACE).

	f1	f2	f3	f4	f5
ABC	7.105e-15 (2.672e-15)	**0** (0)	6.874e-08 (0.006576)	**0** (0)	0.8951 (2.486)
DE	7.416e-14 (3.253e-14)	**0** (0)	**1.127e-08** (1.331e-05)	**0** (0)	**0.5597** (0.6756)
GA	0.405 (0.4804)	0.2636 (0.4649)	0.2788 (0.1285)	0.06823 (0.2336)	309.8 (560.2)
PSO	**3.109e-15** (0)	**0** (0)	0.02987 (0.02224)	2.985 (1.95)	0.8642 (8.623)
Random	15.36 (1.047)	206.5 (49.24)	33.45 (6.986)	60.85 (6.156)	1.502e+08 (7.715e+07)
KHO 0.3	2.755 (0.5298)	4.997 (1.529)	1.077 (0.07309)	25.42 (6.288)	1.262e+05 (3.999e+05)
KHO 0.1	0.9013 (4.576)	1.153 (0.5725)	0.9808 (0.08427)	82.34 (28.73)	3.638e+04 (1.986e+05)
KHO 0.01	19.54 (0.3989)	4.828 (1.667)	0.06546 (0.04697)	86.67 (24.38)	156.1 (2742)
KHO 0.001	19.47 (0.4189)	5.781 (1.45)	0.03901 (0.05073)	84.57 (22.1)	9.54 (6.669e+07)
KHO 0.1-0.001	0.1539 (5.095)	0.3468 (0.2488)	0.09367 (0.1925)	78.67 (24.82)	322.4 (1.712e+05)

	f6	f7	f8	f9	f10
ABC	**2.659** (0.7143)	10.94 (12.08)	0.2239 (0.1311)	3.246e-16 (9.051e-17)	9.679e-17 (3.789e-17)
DE	3.572 (0.5385)	0.1085 (0.07433)	0.0002765 (0.0001027)	2.29e-15 (1.079e-15)	1.287e-26 (1.796e-26)
GA	4.604 (1.579)	8.19 (7.973)	1.813 (0.4921)	0.07912 (0.03888)	0.1653 (0.3744)
PSO	5.491 (1.258)	**2.014e-14** (4.147e-13)	**1.208e-15** (1.34e-15)	**1.286e-21** (2.208e-21)	**4.933e-41** (1.806e-40)
Random	16.42 (1.715)	1563 (318.9)	32.29 (4.444)	15.47 (2.288)	3414 (822.9)
KHO 0.3	12.52 (3.498)	662.9 (882.3)	0.8498 (0.3285)	8.441 (7.556)	8.929 (5.28)
KHO 0.1	16.21 (5.381)	145.6 (78.9)	0.2931 (0.1077)	21.72 (15.31)	0.9661 (0.5751)
KHO 0.01	13.45 (3.956)	2.344 (6085)	0.02826 (0.01003)	48.29 (35.19)	0.009647 (0.006323)
KHO 0.001	12.76 (8.224)	0.02768 (1.761e+04)	23.98 (11.83)	47.81 (10.38)	9.704e-05 (5.153e-05)
KHO 0.1-0.001	13.12 (5.149)	1.089 (2.798)	0.04958 (0.04955)	0.07108 (17.41)	0.006017 (0.04688)

of the uniform scattering might improve convergence and exploitation. The role of parameters η^+ and η^-, and of the score-ranking scheme should be investigated. Auto-adaptive schemes for the adjustment of crucial parameter $scatter$ might enhance performance on many problems. The algorithm could be adapted relatively easily to problems of constrained optimization. Finally, the results seem to indicate a certain complementarity of KHO with other population-based algorithms, which should make it a good candidate for hybridization.

REFERENCES

[1] Artificial Bee Colony Algorithm Homepage, http://mf.erciyes.edu.tr/abc/

[2] Bendtsen, C. 2011 "pso: Particle Swarm Optimization" R package version 1.0.1, http://CRAN.R-project.org/package=pso

[3] Burton, M., Burton, R. 1970 *The international wildlife encyclopedia, Volume 1*, Marshall Cavendish

[4] Clerc, M. et al. 2010 "Standard particle swarm optimisation 2007", URL: http://www.particleswarm.info/standard_pso_2007.c

[5] Eddelbuettel, D., Francois, R. 2011 "Rcpp: Seamless R and C++ Integration" *Journal of Statistical Software* 40(8), pp. 1-18, http://www.jstatsoft.org/v40/i08/

[6] Herrera, F., Lozano, M., Molina, D. 2010 "Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems" February 9, 2010

[7] Holland, J. 1975 *Adaptation in Natural and Artificial Systems*, University of Michigan Press

[8] Karaboga, D., Basturk, B. 2007 "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm", *Journal of Global Optimization*, Volume 39, Number 3, pp. 459-471

[9] Kennedy, J., Eberhart, R. 1995 "Particle swarm optimization" *Proceed-*

Table IV

MEDIAN COST OBTAINED BY ABC, DE, GA, PSO, KHO AND RANDOM SEARCH ON THE TEST SUITE, 100 TRIALS (FIVE DIFFERENT CHOICES FOR KHO'S SCATTER PARAMETER : FIXED AT 0.3, 0.1, 0.01, AND 0.001, AND LINEARLY DECREASED FROM 0.1 TO 0.001) DIM=200, POPSIZE= 200, ITER= 50 (STANDARD DEVIATIONS IN PARENTHESES, BEST MEDIAN IN BOLDFACE).

	f1	f2	f3	f4	f5
ABC	20.73 (0.04959)	3.237e+04 (982.5)	4292 (146.9)	2859 (74.64)	2.467e+11 (1.394e+10)
DE	20.71 (0.04798)	3.058e+04 (1026)	4097 (120)	2943 (49.89)	2.246e+11 (1.082e+10)
GA	19.96 (0.1113)	1.953e+04 (883.3)	2644 (126.4)	2268 (56.54)	1.053e+11 (8.466e+09)
PSO	15.3 (0.3637)	4311 (386.9)	561.9 (49.5)	2070 (47.98)	6.91e+09 (1.391e+09)
Random	20.88 (0.02924)	3.455e+04 (716.5)	4602 (110.6)	3124 (44.72)	2.643e+11 (8.832e+09)
KHO 0.3	11.05 (4.591)	183.8 (697.7)	6.467 (87.2)	1786 (177.3)	4.904e+05 (5.947e+05)
KHO 0.1	9.545 (4.205)	64.98 (308.5)	1.648 (44.18)	2015 (207.4)	1.747e+05 (3.289e+05)
KHO 0.01	19.58 (0.07194)	114.5 (9.739)	**0.4737** (0.1034)	1721 (119.5)	**1.326e+04** (1.684e+07)
KHO 0.001	19.5 (0.06086)	145.6 (13.54)	0.9237 (0.1494)	1666 (117.2)	6.002e+04 (5.411e+09)
KHO 0.1-0.001	**7.961** (7.598)	**19.71** (201.5)	0.7366 (32.49)	**1577** (99.78)	1.367e+04 (3.207e+05)

	f6	f7	f8	f9	f10
ABC	1160 (44.43)	6.197e+05 (9.622e+04)	97.01 (0.6387)	6.556e+31 (4.402e+46)	4.805e+05 (1.703e+04)
DE	1240 (28.7)	7.196e+05 (8.821e+04)	96.17 (0.8543)	1.95e+79 (3.91e+83)	4.568e+05 (1.493e+04)
GA	912.6 (30.42)	**3.654e+05** (4.737e+04)	90.65 (1.38)	3.762e+35 (3.526e+44)	2.93e+05 (1.533e+04)
PSO	942.8 (35.79)	3.839e+05 (6.097e+04)	**66.17** (4.024)	**308.2** (340)	6.288e+04 (5492)
Random	1356 (25.8)	8.975e+05 (1.196e+05)	95.3 (0.6393)	5.436e+87 (3.848e+90)	5.117e+05 (1.228e+04)
KHO 0.3	1144 (101.1)	7.291e+05 (1.674e+05)	93.21 (7.536)	486.8 (66.4)	649.3 (1.08e+04)
KHO 0.1	966.8 (98.15)	5.673e+05 (1.183e+05)	78.97 (3.307)	755.9 (1.826e+29)	73.04 (3759)
KHO 0.01	**470.7** (53.11)	6.567e+05 (3.093e+05)	89.88 (1.741)	1.888e+87 (5.761e+107)	1.186 (0.2198)
KHO 0.001	818.8 (258.1)	3.044e+06 (4.718e+06)	95.2 (1.189)	3.373e+105 (5.354e+119)	**0.8473** (0.8737)
KHO 0.1-0.001	474.7 (96.66)	5.083e+05 (1.144e+05)	80.08 (3.064)	811.3 (8.129e+16)	2.354 (2196)

Table V

MEDIAN COST OBTAINED BY ABC, DE, GA, PSO, KHO AND RANDOM SEARCH ON THE TEST SUITE, 100 TRIALS (FIVE DIFFERENT CHOICES FOR KHO'S SCATTER PARAMETER : FIXED AT 0.3, 0.1, 0.01, AND 0.001, AND LINEARLY DECREASED FROM 0.1 TO 0.001) DIM=200, POPSIZE= 200, ITER= 1000 (STANDARD DEVIATIONS IN PARENTHESES, BEST MEDIAN IN BOLDFACE).

	f1	f2	f3	f4	f5
ABC	10.03 (0.6114)	107.8 (78.39)	5.939 (6.538)	397.9 (24.85)	2.341e+05 (6.826e+06)
DE	11.78 (0.1911)	1304 (75.26)	158.3 (8.036)	978.1 (23.85)	7.215e+08 (9.416e+07)
GA	8.981 (0.2693)	813.6 (57.45)	92.1 (8.218)	**348.8** (19.64)	2.618e+08 (4.978e+07)
PSO	**2.818** (0.3042)	73.87 (7.768)	1.522 (0.1487)	497.1 (192.3)	9.009e+05 (4.291e+05)
Random	20.8 (0.02844)	3.271e+04 (672.2)	4351 (93)	3027 (38.65)	2.424e+11 (7.862e+09)
KHO 0.3	11 (4.709)	174.3 (750)	6.098 (95.79)	1464 (190)	3.511e+05 (5.297e+05)
KHO 0.1	9.102 (4.457)	50.13 (266.9)	1.464 (30.6)	1991 (185.6)	2.27e+05 (4.271e+05)
KHO 0.01	19.56 (0.07455)	112.2 (10.32)	0.2245 (0.08447)	1695 (103.6)	1158 (2.821e+08)
KHO 0.001	19.5 (0.05831)	130.6 (8.636)	0.1065 (0.03238)	1685 (101.5)	**302.3** (1.684e+09)
KHO 0.1-0.001	7.293 (3.932)	**20.6** (221)	**0.06891** (27.87)	1572 (115.3)	306.6 (3.013e+05)

	f6	f7	f8	f9	f10
ABC	317 (10.32)	2.666e+05 (1.785e+04)	94.97 (0.7385)	**2.231** (0.4175)	526.4 (770.1)
DE	446.3 (9.36)	2.888e+05 (2.226e+04)	85.73 (1.305)	129.1 (3.543)	1.753e+04 (992.8)
GA	**271.4** (10.79)	**9.43e+04** (7620)	55.23 (2.548)	76.5 (4.063)	9815 (793.9)
PSO	520.3 (92.83)	1.139e+05 (2.101e+04)	**29.78** (2.105)	4.263 (1.028)	57.61 (16.55)
Random	1302 (23.28)	6.701e+05 (6.33e+04)	93.91 (0.5761)	5.921e+82 (7.099e+84)	4.85e+05 (1.066e+04)
KHO 0.3	1049 (117.1)	6.282e+05 (1.479e+05)	90.96 (12.93)	432.3 (75.88)	522.6 (8681)
KHO 0.1	972.7 (83.12)	5e+05 (8.644e+04)	76.48 (3.061)	682.1 (65.01)	53.2 (3588)
KHO 0.01	431.3 (63.92)	1.526e+05 (5.307e+05)	82.54 (2.476)	2.751e+87 (1.383e+109)	0.5515 (0.1279)
KHO 0.001	753.4 (260.3)	3.215e+06 (7.006e+06)	88.05 (1.75)	1.191e+104 (2.923e+116)	**0.005412** (0.001316)
KHO 0.1-0.001	375.7 (57.4)	2.134e+05 (4.698e+04)	76.57 (3.311)	144 (289.2)	0.03702 (2877)

ings of the IEEE International Conference on Neural Networks IV pp. 19421948

[10] Kennedy, J., Eberhart, R., Shi, Y. 2001 *Swarm Intelligence* Morgan Kaufmann

[11] Mullen, K., Ardia, D., Gil, D., Windover, D., Cline, J. 2011 "'DEoptim': An R Package for Global Optimization by Differential Evolution", *Journal of Statistical Software*, 40(6), pp. 1-26, http://www.jstatsoft.org/v40/i06/

[12] Riedmiller, M., Braun, H. 1993 "A direct adaptive method for faster backpropagation learning: the RPROP algorithm", in Ruspini, H. (ed) *Proceedings of the IEEE International Conference on Neural Networks (ICNN) 1993* pp. 586-591

[13] Storn, R., Price, K. 1997 "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces". *Journal of Global Optimization* 11, pp. 341359

[14] Willighagen, E. 2005 "genalg: R Based Genetic Algorithm", R package version 0.1.1. http://CRAN.R-project.org/package=genalg