Splitting for Optimization

Qibin Duan[a,*], Dirk P. Kroese[a]

[a]*School of Mathematics and Physics, The University of Queensland, Brisbane 4072, Australia*

Abstract

The splitting method is a well-known method for rare-event simulation, where sample paths of a Markov process are split into multiple copies during the simulation, so as to make the occurrence of a rare event more frequent. Motivated by the splitting algorithm we introduce a novel global optimization method for continuous optimization that is both very fast and accurate. Numerical experiments demonstrate that the new splitting-based method outperforms known methods such as the differential evolution and artificial bee colony algorithms for many bench mark cases.

Keywords:
Evolutionary computation, Splitting method, Continuous optimization, Artificial bee colony, Differential evolution

1. Introduction

Randomized algorithms have shown to be of significant benefit for solving complicated optimization problems. In particular, such methods are of great use in finding (near) optimal solutions to highly multi-modal functions, "black-box" problems where gradients are difficult to obtain, and problems with complicated constraints. Since the 1960s many well-known random algorithms for optimization have been proposed. Many of these algorithms can be viewed as population Monte Carlo algorithms, where a sample (population) of individuals is modified randomly over time in order to produce a high-performing sample according to some chosen objective. Often such algorithms are nature-inspired. Examples include evolution strategy (ES) [1], evolutionary programming (EP) [2], genetic algorithms (GA) [3] and, more recently, the cross-entropy (CE) method [4], differential evolution (DE) [5], particle swarm optimization (PSO) [6], ant colony optimization(ACO)[7], fast EP (FEP)[8], artificial bee colony (ABC) [9] and many other inventive methods based on the principle of exploration and exploitation.

The splitting method is a well-known method for rare-event simulation, where sample paths of a Markov process are split into multiple copies during the simulation, so as to make the occurrence of a rare event more frequent. The purpose of this paper is to introduce the "splitting" idea to the optimization toolbox for continuous optimization, and to show that the approach, when reduced to its core elements, can outperform other well-known methods in terms of accuracy and speed.

To motivate the splitting technique, we draw on various ideas from rare-event simulation. It has been realized for some time that the problem of minimizing a complicated continuous or discrete

*Corresponding author.
Email addresses: `q.duan@uq.edu.au` (Qibin Duan), `kroese@maths.uq.edu.au` (Dirk P. Kroese)

function $S(\mathbf{x})$, $\mathbf{x} \in \mathscr{X}$ is closely related to the efficient estimation of rare-event probabilities of the form $\mathbb{P}(S(\mathbf{X}) \leqslant \gamma)$, where $\mathbf{X}$ is a random element of $\mathscr{X}$, distributed according to a given probability density function (pdf), e.g., the uniform pdf on $\mathscr{X}$. The latter requires efficient sampling from the level set $\{\mathbf{x} \in \mathscr{X} : S(\mathbf{x}) \leqslant \gamma\}$. By gradually decreasing γ the level set becomes smaller and smaller until it only contains elements that lie close to the minimizer of S. For γ close to the minimum, the event $\{S(\mathbf{X}) \leqslant \gamma\}$ will be very rare. Hence, it is useful to apply rare event simulation techniques to optimization problems. This is, for example, the premise of the cross-entropy (CE) method, which aims to find a sequence of pdfs $f_1, f_2, f_3, \ldots$ that converges to the pdf that concentrates all its mass in the set of points where S is minimal. In the CE method the densities $f_1, f_2, \ldots$ are parameterized by a fixed-dimensional parameter vector, which is updated at each iteration using the cross-entropy (or Kullback–Leibler) distance. If instead a non-parametric approach is taken, the densities can be represented by a collection of particles, whose distribution is updated at each iteration. This is where the splitting method enters the scene.

The splitting method was first put forward by [10] for time-dependent Markovian models and later generalized in [11] to both static (that is, not involving time) and non-Markovian models. The latter modification is called *Generalized Splitting* (GS), which will be the focus of our discussion below.

The purpose of GS method is to estimate the rare-event probability $\mathbb{P}(S(\mathbf{X}) \leqslant \gamma)$ for some (small) γ, where $\mathbf{X}$ has a specified nominal distribution. This is done by first defining a sequence of levels $\{\gamma_t\}$ decreasing to γ and then constructing a sequential sampling scheme that samples from the conditional distribution of $\mathbf{X}$ given $\{S(\mathbf{X}) \leqslant \gamma_t\}$. Note that if γ is equal to the minimum of S, then sampling $\mathbf{X}$ conditional on $\{S(\mathbf{X}) \leqslant \gamma\}$ is equivalent to sampling from the minimizer of S. However, the problem is that in general the minimum value is not known, and hence the intermediate values $\{\gamma_t\}$ have to be determined adaptively. This is the motivation for the ADAptive Multilevel splitting algorithm (ADAM) in [12, 13]. The ADAM algorithm has be applied to mostly combinatorial optimization problems. For continuous optimization, where the nominal distribution is taken to be uniform, the ADAM algorithm is generally more difficult to apply, as sampling $\mathbf{X}$ conditional on $\{S(\mathbf{X}) \leqslant \gamma_t\}$ may be too time-consuming or complicated.

In this paper we propose to replace the complicated sampling step in the ADAM algorithm with a simpler one, while retaining the other features. Instead of sampling (at stage t) from the uniform distribution on the difficult "level set" $\{\mathbf{x} : S(\mathbf{x}) \leqslant \gamma_t\}$, our sampling scheme involves sampling from a collection of multi-variate normal distributions, using a Gibbs sampler. The mean vector and covariance matrix of the normal distributions are determined by the current population of individuals. This simplification greatly increases the applicability of the ADAM method, making it competitive for continuous optimization. We compare the method with the best performing algorithms in this area and demonstrate that it can outperform them for a suite of established test functions.

The rest of the paper is organized as follows. In Section 2, we review the mathematical framework of the GS and ADAM algorithms, and put forward the new splitting idea for continuous optimization. For easy comparison we summarize two well-performing algorithms, DE and ABC, in Section 3. In Section 4, we employ a popular suite of test functions to evaluate the performance of the proposed optimization technique. We describe the precise settings of the numerical experiments and show the comparison between DE, ABC, and the new splitting algorithm for continuous optimization (SCO). Finally, in Section 5, we further discuss the results of the numerical experiments, and compare the proposed algorithm with other algorithms via existing comparative studies.

2. Mathematical Framework and Algorithms

2.1. Mathematical Framework

Let $S(\mathbf{x})$ be a continuous function on $\mathbb{R}^n$. We wish to find the minimum $\gamma^* = \min_{\mathbf{x}} S(\mathbf{x})$ and the global minimizer $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x}} S(\mathbf{x})$, assuming for simplicity that there is only one minimizer. Let f be some "nominal" pdf f, e.g., the uniform pdf on some bounded subset of $\mathbb{R}^n$. Suppose first that γ^* is known. To find the corresponding $\mathbf{x}^*$ we could sample a random vector $\mathbf{X}$ conditional on the rare event $\{S(\mathbf{X}) \leqslant \gamma^*\}$, which basically means sampling from the argmin set $\{\mathbf{x}^*\}$. This can be done using the GS method by sampling iteratively from intermediate (increasingly rare) events $\{S(\mathbf{X}) \leqslant \gamma_t\}$, for levels $\infty = \gamma_0 \geqslant \gamma_1 \geqslant \ldots \geqslant \gamma_{T-1} \geqslant \gamma_T = \gamma^*$. Define the *level set* of S corresponding to level γ_t to be the set $\{\mathbf{x} : S(\mathbf{x}) \leqslant \gamma_t\}$. We call it the γ_t*-level set* for short. Let f_t be the conditional pdf of $\mathbf{X} \sim f$ given $\{S(\mathbf{X}) \leqslant \gamma_t\}$; that is, $f_t(\mathbf{x})$ is proportional to $f(\mathbf{x}) I_{\{S(\mathbf{x}) \leqslant \gamma_t\}}$. In particular, we are interested in sampling from $f^*(\mathbf{x}) = f_T(\mathbf{x}) \propto f(\mathbf{x}) I_{\{S(\mathbf{x}) \leqslant \gamma^*\}}$. The GS method works as follows.

Given the sequence of intermediate levels $\gamma_t, t = 0, \ldots, T$, and an initial sample (population) $\mathcal{X}_0$ from $f_0 = f$, execute the following two phases at each iteration t, from $t = 0$ to $t = T - 1$:

(a) Let $\mathcal{E}_{t+1} = \{\mathbf{x} \in \mathcal{X}_t : S(\mathbf{x}) \leqslant \gamma_{t+1}\}$, which is referred as the *elite set* of $\mathcal{X}_t$. Its size is denoted by N_{t+1}. Note that the elite elements are distributed according f_{t+1}.

(b) Split the elite population in $\mathcal{E}_{t+1}$ to create the next population $\mathcal{X}_{t+1}$, distributed according to f_{t+1}. Increase t by one and go to Step (a).

The splitting step (b) can be implemented in many different ways; for example, in [12] it is done by running a Markov chain on the γ_{t+1}-level set starting from each point in the elite set $\mathcal{E}_{t+1}$ and storing each state in $\mathcal{X}_{t+1}$. The only requirement is that the Markov chain has stationary pdf f_{t+1}.

Figure 1 illustrates how the splitting is performed on a typical problem in 2-D space. Here, there are three levels, $\gamma_t, t = 1, 2, 3$, and the initial sample set is $\mathcal{X}_0 = \{\mathbf{X}_1, \ldots, \mathbf{X}_5\}$, where $\mathbf{X}_1, \ldots, \mathbf{X}_5 \overset{\text{iid}}{\sim} f_0$. Since two of the five initial points, namely $\mathbf{X}_1$ and $\mathbf{X}_2$, are such that $S(\mathbf{X}_1)$ and $S(\mathbf{X}_2)$ are below the γ_1 threshold, we have that $N_1 = 2$. The elite points $\mathbf{X}_1$ and $\mathbf{X}_2$ are the starting points of two Markov chains, whose stationary pdf is f_1. The length of each Markov chain is called the *splitting factor*. In this case, the GS algorithm uses the *same* splitting factor, 5, for each chain. This is called GS with *Fixed Splitting*.

Thus, we have two Markov chains on the γ_1-level set that start from $\mathbf{X}_1$ and $\mathbf{X}_2$ respectively and run for 5 steps, which are plotted in thicker lines. For the Markov chain starting from the point $\mathbf{X}_1$, two of five points have entered the γ_2-level set, say $\mathbf{X}_{1,3}, \mathbf{X}_{1,4}$, while only one point of the Markov chain starting at $\mathbf{X}_2$ has reached the next level, namely, $\mathbf{X}_{2,2}$. So, $N_2 = 3$. In the final stage, we start three independent Markov chains (of length 5) on the γ_2-level set from points $\mathbf{X}_{1,3}, \mathbf{X}_{1,4}$ and $\mathbf{X}_{2,2}$ with the stationary pdf f_2. Of all the points generated in the last stage, four have reached the final level set, so $N_3 = 4$.

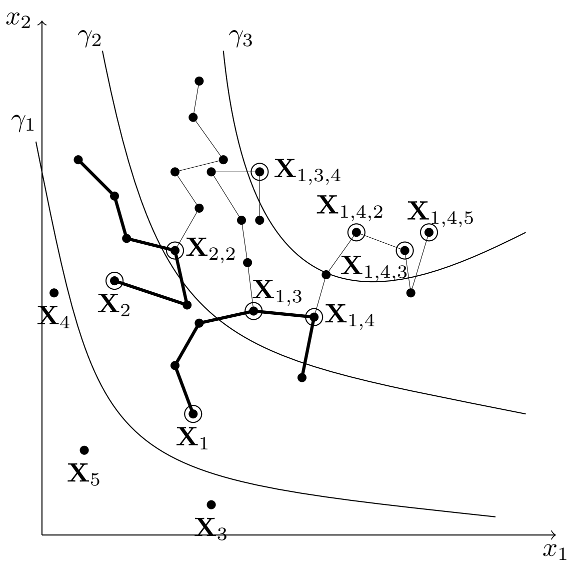

Figure 1: Illustration of the GS algorithm in 2-D space

In practice γ^* is not known and therefore one cannot determine the intermediate levels beforehand. Instead, one can determine them adaptively via the ADAM algorithm. This involves a *rarity parameter* ϱ. Having again an initial sample set $\mathcal{X}_0$ from f_0, the ADAM algorithm modifies the two steps of the GS algorithm as follows:

(a) Calculate the function value $S(\mathbf{x})$ for each $\mathbf{x} \in \mathcal{X}_t$ and sort these from smallest to largest: $S_1 \leqslant S_2 \leqslant \ldots \leqslant S_N$. For $N^{\mathrm{e}} = \lceil |\mathcal{X}_t|\, \varrho \rceil$, where $|\mathcal{X}_t|$ is the number of elements in $\mathcal{X}_t$, let γ_{t+1} be the N^{e}-th, smallest function value; that is, $\gamma_{t+1} = S_{N^{\mathrm{e}}}$. Note that N^{e} is the size of the elite set $\mathcal{E}_{t+1} = \{\mathbf{x} \in \mathcal{X}_t : S(\mathbf{x}) \leqslant \gamma_{t+1}\}$.

(b) Split the elite population in $\mathcal{E}_{t+1}$ to create the next population $\{\mathcal{X}_{t+1}\}$, distributed according to f_{t+1}. Increase t by one and repeat Steps (a) and (b) until some stopping condition is met.

Again, the splitting step can be implemented in different ways, e.g., by running a Markov chain from each of the elite elements. Instead of splitting each element into a fixed number of samples (fixed splitting factor) it is sometimes useful to keep the sample size (the number of elements of $\mathcal{X}_t$) constant, say N. This is called GS with a *Fixed Effort*. A way to "evenly" split the elite samples into N new samples is by defining random splitting factors as follows:

$$s_i = \left\lfloor \frac{N}{N^{\mathrm{e}}} \right\rfloor + B_i, \quad i = 1, \ldots, N^{\mathrm{e}}, \tag{1}$$

where $B_1, \ldots, B_{N^{\mathrm{e}}} \sim \mathsf{Ber}(1/2)$ and contingent on $\sum_{i=1}^{N^{\mathrm{e}}} B_i = N \mod N^{\mathrm{e}}$.

The following pseudo code in Algorithm 1 summarizes the ADAM algorithm for minimization. A possible terminating condition for Algorithm 1 is to stop when the overall best found solution does not improve over d iterations. Other possible stopping criteria include the maximum number of iterations or the CPU time exceeding a threshold. To keep the stopping criterion general, we use the customary control statement "**while** the stopping criterion is not met **do**".

Algorithm 1 The ADAM Algorithm for Minimization

```
Input: Sample size $N$, rarity parameter $\varrho \in (0,1)$.
Output: Final iteration number $t$ and sequence $(\mathbf{X}_{\text{best},1}, b_1), \ldots, (\mathbf{X}_{\text{best},t}, b_t)$ of best solutions and
    function values found at iteration $1, \ldots, t$, respectively.
 1: Generate $\mathcal{X}_0 = \{\mathbf{X}_1, \ldots, \mathbf{X}_N\}$ from the pdf $f_0 = f$. Set $t = 0$ and $N^{\text{e}} = \lceil N\varrho \rceil$.
 2: while the stopping criterion is not met do
 3:   For all $\mathbf{X} \in \mathcal{X}_t$, evaluate $S(\mathbf{X})$, and sort $\{S(\mathbf{X})\}$ from smallest to largest: $S_1 \leqslant \ldots \leqslant S_N$,
      then select the $N^{\text{e}}$ smallest ones and store the corresponding $\mathbf{X}$ in $\mathcal{E}_{t+1}$, Set $b_{t+1} = S_1$ and
      record the corresponding individual as $\mathbf{X}_{\text{best},t+1}$.
 4:   Compute the splitting factors $s_{t+1,i}$ for each $\mathbf{X}^{(i)} \in \mathcal{E}_{t+1}$ as in Eq. (1), $i = 1, \ldots, N^{\text{e}}$.
 5:   for $i = 1$ to $N^{\text{e}}$ do
 6:     Sample $\mathbf{Y}_{i,j}$ from the density $\kappa_{t+1}(\mathbf{y} \,|\, \mathbf{Y}_{i,j-1})$, where $\mathbf{Y}_{i,0} = \mathbf{X}^{(i)}$, $j = 1, \ldots, s_{t+1,i}$ and
        $\kappa_{t+1}(\mathbf{y} \,|\, \mathbf{Y}_{i,j-1})$ is a Markov transition density with stationary pdf $f_{t+1}$.
 7:     Add $\mathbf{Y}_{i,j}$ to $\mathcal{X}_{t+1}$.
 8:   end for
 9:   Set $t = t + 1$.
10: end while
11: return $\{(\mathbf{X}_{\text{best},k}, b_k), k = 1, \ldots, t\}$.
```

In Line 6, $\mathbf{Y}_{i,j}$ denotes the j-th state of the Markov chain that starts from the i-th sample, $\mathbf{X}^{(i)}$, of the elite set $\mathcal{E}_{t+1}$.

Note that in Algorithm 1 the Markov transition density κ_t has not been specified. One could use, for example, a Gibbs or Metropolis-Hastings sampler to sample from the stationary pdf f_t. A Gibbs sampler has the advantage that each component can be sampled from a one-dimensional distribution.

Example. We illustrate the potential use of the ADAM algorithm for continuous optimization via the minimization of the Rosenbrock function — a well-known difficult test function; see also [12]. The n-dimensional Rosenbrock function is given by

$$S(\mathbf{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2, \quad \mathbf{x} \in [-2,2]^n.$$

To employ the ADAM algorithm, we first need to specify the nominal sampling pdf. It is natural to let f be the uniform distribution on $[-2,2]^n$. This means that f_t is the uniform pdf on the γ_t-level set. The procedure is exactly as Algorithm 1, where the sampling step (Line 6) is now specified as follows.

Let $\mathbf{X} = (X_1, \ldots, X_n)$ be the "current" point of the Markov chain. For example $\mathbf{X}$ could be one of the elite points where the Markov chain starts (recall that there are N^{e} Markov chains starting from each of the elite points). To generate the "next" point $\mathbf{Y} = (Y_1, \ldots, Y_n)$ we apply a Gibbs sampler that samples each component uniformly on the corresponding level set. To identify the boundaries of the uniform distribution for each component on the level set, a quartic equation needs to be solved. For example, the distribution of Y_1 given $X_2, \ldots, X_n$ is uniformly distributed on the set $\{x_1 \in [-2,2] : S(x_1, X_2, \ldots, X_n) \leqslant \gamma_{t+1}\}$, which can be written as

$$\{x_1 \in [-2,2] : \quad S(\mathbf{X}) - 100(X_2 - X_1^2)^2 - (X_1 - 1)^2 + 100(X_2 - x_1^2)^2 + (x_1 - 1)^2 \leqslant \gamma_{t+1}\}.$$

The roots of the quartic polynomial (in x_1)

$$100(X_2 - x_1^2)^2 + (x_1 - 1)^2 - \gamma_{t+1} + S(\mathbf{X}) - 100(X_2 - X_1^2)^2 - (X_1 - 1)^2$$

identify the boundaries of the region on which Y_1 is uniformly distributed. More generally, $\mathbf{Y}$ is obtained via the following steps:

1. Set $\Sigma = 100(X_2 - X_1^2)^2 + (X_1 - 1)^2$.
   Generate $Y_1 \sim f_{t+1}(y_1 \,|\, X_2, \ldots, X_n)$; that is, Y_1 is a random variable uniformly distributed on the set $\{[r_1, r_2] \cup [r_3, r_4]\} \cap [-2, 2]$, where $r_1 < r_2, r_3 < r_4$ are the real roots of the quartic equation $a_1 x^4 + a_2 x^3 + a_3 x^2 + a_4 x + a_5 = 0$ with coefficients

   $$a_1 = 100, \;\; a_2 = 0, \;\;\; a_3 = 1 - 200X_2, \;\;\; a_4 = -2, \;\;\; a_5 = 1 + 100X_2 + S_1 - \Sigma - \gamma_{t+1},$$

   with $S_1 = S(\widetilde{\mathbf{Y}})$, where $\widetilde{\mathbf{Y}} = (X_1, \ldots, X_n)$ is the initial intermediate point.

   Depending on the coefficients, the quartic equation has either 2 or 4 real roots. Thus, in some cases $r_3 = r_1$ and $r_4 = r_2$ and Y_1 will be a random variable uniformly distributed on the set $[r_1, r_2] \cap [-2, 2]$.

2. For each $j = 2, \ldots, n-1$, set $\Sigma = 100(X_{j+1} - X_j^2)^2 - 100(X_j^2 + 2Y_{j-1}^2 X_j)$.
   Generate $Y_j \sim f_{t+1}(y_j \,|\, Y_1, \ldots, Y_{j-1}, X_{j+1}, \ldots, X_n)$; that is, Y_j is a random variable uniformly distributed on the set $\{[r_1, r_2] \cup [r_3, r_4]\} \cap [-2, 2]$, where $r_1 < r_2, r_3 < r_4$ are the real roots of the quartic equation $a_1 x^4 + a_2 x^3 + a_3 x^2 + a_4 x + a_5 = 0$ with coefficients

   $$a_1 = 100, \; a_2 = 0, \; a_3 = 101 - 200X_{j+1}, \; a_4 = -2 - 200Y_{j-1}^2, a_5 = 1 + 100X_{j+1} + S_j - \Sigma - \gamma_{t+1},$$

   with $S_j = S(\widetilde{\mathbf{Y}})$, where $\widetilde{\mathbf{Y}} = (Y_1, \ldots, Y_{j-1}, X_j, \ldots, X_n)$.

3. Set $\Sigma = 100(X_n - X_{n-1}^2)^2$.
   Generate $Y_n \sim f_{t+1}(y_n \,|\, Y_l, \ldots, Y_{n-1})$; that is, Y_n is a random variable uniformly distributed on the set $[r_1, r_2] \cap [-2, 2]$, where $r_1 < r_2$ are the real roots of the quadratic equation $a_1 x^2 + a_2 x + a_3 = 0$ with coefficients

   $$a_1 = 100, a_2 = -200(Y_{n-1})^2, a_3 = 100Y_{n-1}^4 + S_n - \Sigma - \gamma_{t+1},$$

   with $S_n = S(\widetilde{\mathbf{Y}})$, where $\widetilde{\mathbf{Y}} = (Y_1, \ldots, Y_{n-1}, X_n)$.

The above ADAM procedure works well for minimizing the Rosenbrock function because sampling from the uniform distribution on a level set can be carried out by sampling 1-D uniform random variables from relatively easy regions (determined by solving a 4th degree polynomial). Unfortunately, such an approach may not be feasible for other continuous objective functions.

2.2. Splitting Algorithm for Continuous Optimization

To be able to use the ADAM framework for general continuous optimization, we provide a new method for splitting the samples. The idea is to "split" (at iteration t) each elite point $\mathbf{X}^{(i)} = (X_1^{(i)}, \ldots, X_n^{(i)}) \in \mathcal{E}_{t+1}$ on the γ_{t+1}-level set into $s_{t+1,i}$ points by using a multivariate normal distribution with mean vector $\boldsymbol{\mu}_i = \mathbf{X}^{(i)}$ and a diagonal covariance matrix with corresponding vector of standard deviations $\boldsymbol{\sigma}_i$. Hence, the components of the multivariate normal distribution are independent of each other. In theory, any distribution could have been used to sample from the level sets. The multivariate normal distribution was chosen for computational convenience and efficiency.

In the splitting step, to have most of the new samples still on the current γ_t-level set the components of $\boldsymbol{\sigma}_i$ should be chosen not too large. Neither should they be chosen too small, as otherwise there will be little exploration. To choose an appropriate sampling region (determined by $\boldsymbol{\sigma}_i$) we employ another elite point, in the following way. For each elite point $\mathbf{X}^{(i)}$ ("base point") choose another elite point $\mathbf{X}^{(R)}$ ("assistant point") from $\mathcal{E}_{t+1}$, where R is uniformly selected from $\{1, \ldots, N_t\} \setminus \{i\}$. We now choose the vector of standard deviations as

$$\boldsymbol{\sigma}_i = w\,|\mathbf{X}^{(i)} - \mathbf{X}^{(R)}| \overset{\text{def}}{=} w \begin{pmatrix} |X_1^{(i)} - X_1^{(R)}| \\ |X_2^{(i)} - X_2^{(R)}| \\ \cdots \\ |X_n^{(i)} - X_n^{(R)}| \end{pmatrix}, \tag{2}$$

where w is a *scale factor.* From our experience, $w = 0.5$ performs well in practice.

It is important to realize that the splitting step does not involve direct sampling from an n-D normal distribution. Instead, as in the GS algorithm, we sample the components of the normal distribution via the Gibbs sampler. To improve efficiency each component of the base point is updated in a *random order* $\mathbf{r} = (r_1, \ldots, r_n)$ instead of a fixed order $(1, \ldots, n)$. By updating the components in a random order, some components with better feasible regions can be updated before those with poor (e.g., very narrow) ones. This enhances the probability to improve the base point to a better point.

During the Gibbs sampler steps, an intermediate point $\mathbf{y}$ is used to store the component updates. Initially, $\mathbf{y} = \mathbf{X}^{(i)}$ for the current base point that is to be split. At step k of the Gibbs sampler, the r_kth component of $\mathbf{y}$ is sampled from the 1-D normal distribution with mean $\mathbf{y}(r_k)$ and standard deviation $\boldsymbol{\sigma}_i(r_k)$, and the other components remain the same, giving rise to a new trial vector $\widetilde{\mathbf{y}}$.

If $S(\widetilde{\mathbf{y}}) < S(\mathbf{y})$ then $\widetilde{\mathbf{y}}$ is accepted as the new intermediate point; otherwise, we repeat the sampling of the r_kth component up to `MaxTry` times (say 3 or 5 times). If still no improvement is found after this many tries, then the intermediate point $\mathbf{y}$ remains unchanged. Note that a successful update $\widetilde{\mathbf{y}}$ is determined by comparing with the previous intermediate point $\mathbf{y}$ instead of the base point $\mathbf{X}^{(i)}$. When all n components have been processed, the last intermediate point is taken as the outcome of the splitting of $\mathbf{X}^{(i)}$. This is repeated $s_{t+1,i}$ times, independently, for each base (elite) point $\mathbf{X}^{(i)}, i = 1, \ldots, N^{\mathrm{e}}$. We call our proposed method SCO, which is short for Splitting for Continuous Optimization. The exact algorithm is specified in Algorithm 2. It is also worth noting that, at iteration t, the proposed method is not sampling from the conditional distribution f_{t+1}, as in Algorithm 1. Instead, in the splitting step, we use multiple multivariate normal distributions to sample on the level set. This heuristic modification greatly reduces the algorithm's running time.

Algorithm 2 Splitting for Continuous Optimization (SCO)

```
Input: Objective function $S$, sample size $N$, rarity parameter $\varrho \in (0,1]$, and scale factor $w$,
    maximum number of attempts MaxTry
Output: Final iteration number $t$ and sequence $(\mathbf{X}_{\text{best},1}, b_1), \dots, (\mathbf{X}_{\text{best},t}, b_t)$ of best solutions and
    function values found at iteration $1, \dots, t$, respectively.
1: Generate $\mathcal{X}_0 = \{\mathbf{X}_1, \dots, \mathbf{X}_N\}$ via uniform sampling. Set $t = 0$ and $N^{\text{e}} = \lceil N\varrho \rceil$.
2: while the stopping criterion is not met do
3:   For all $\mathbf{X} \in \mathcal{X}_t$, evaluate $S(\mathbf{X})$, and sort $\{S(\mathbf{X})\}$ from smallest to largest: $S_1 \leqslant \dots \leqslant S_N$,
     then select the $N^{\text{e}}$ smallest ones and store the corresponding $\mathbf{X}$ in $\mathcal{E}_{t+1}$; set $b_{t+1} = S_1$ and
     record the corresponding individual as $\mathbf{X}_{\text{best},t+1}$.
4:   Compute the splitting factors $s_{t+1,i}$ for each individual element $\mathbf{X}^{(i)} \in \mathcal{E}_{t+1}, i = 1, \dots, N^{\text{e}}$
     according to Eq. (1).
5:   for $i = 1$ to $N^{\text{e}}$ do
6:     for $j = 1$ to $s_{t+1,i}$ do
7:       Select $R$ uniformly from the set $\{1, \dots, N^{\text{e}}\} \setminus \{i\}$.
8:       Compute $\boldsymbol{\sigma}_i$ as Eq. (2).
9:       Set $\mathbf{y} = \widetilde{\mathbf{y}} = \mathbf{X}^{(i)} \in \mathcal{E}_{t+1}$.
10:      Generate a random order $\mathbf{r} = (r_1, \dots, r_n)$.
11:      for $k = 1$ to $n$ do
12:        for Try = 1 to MaxTry do
13:          Update the $r_k$th component of $\mathbf{y}$: set $\widetilde{\mathbf{y}}(r_k) = \mathbf{y}(r_k) + \boldsymbol{\sigma}_i(r_k)Z$, $Z \sim \mathsf{N}(0,1)$.
14:          If $S(\widetilde{\mathbf{y}}) < S(\mathbf{y})$, then set $\mathbf{y}(r_k) = \widetilde{\mathbf{y}}(r_k)$, break the for loop.
15:        end for
16:      end for
17:      Add $\mathbf{y}$ to $\mathcal{X}_{t+1}$.
18:    end for
19:  end for
20: end while
21: Set $t = t + 1$.
22: return $\{(\mathbf{X}_{\text{best},k}, b_k), k = 1, \dots, t\}$.
```

In Figure 2, the sampling procedure is illustrated. Consider two base points $\mathbf{X}^{(1)}, \mathbf{X}^{(2)}$ from the elite set $\{\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \mathbf{X}^{(3)}, \mathbf{X}^{(4)}\}$ on the γ_1-level set. Suppose the corresponding assistant points for $\mathbf{X}^{(1)}$ and $\mathbf{X}^{(2)}$ are $\mathbf{X}^{(3)}$ and $\mathbf{X}^{(4)}$, respectively. So, $R_1 = 3$ and $R_2 = 4$. These assistant points determine the standard deviation for the sampling distribution of the base points $\mathbf{X}^{(1)}$ and $\mathbf{X}^{(2)}$. Also assume that `MaxTry` $= 2$ and $w = 0.5$.

Let $\mathbf{Y}^{(i)}_{u,v}$ be the trial values ($\mathbf{y}$ in the algorithm) starting from the base point $\mathbf{X}^{(i)}$ at the uth try for the first component and the vth try for the second component. Suppose for base point $\mathbf{X}^{(1)}$ the components are updated in order $\mathbf{r} = (1, 2)$ and for $\mathbf{X}^{(2)}$ they are updated in order $\mathbf{r} = (2, 1)$. Initially, $\mathbf{Y}^{(1)}_{0,0} = \mathbf{X}^{(1)}$ and $\mathbf{Y}^{(2)}_{0,0} = \mathbf{X}^{(2)}$.

As for the first base point $\mathbf{X}^{(1)}$, the horizontal dotted segment to its right indicates 1.5 times the length of standard deviation of the normal distribution of the first component. The first updating step is with respect to the the first component. The corresponding trial point is denoted by $\mathbf{Y}^{(1)}_{1,0}$. But this potential update is rejected by comparing with $\mathbf{Y}^{(1)}_{0,0} = \mathbf{X}^{(1)}$. As `MaxTry` $= 2$, we can

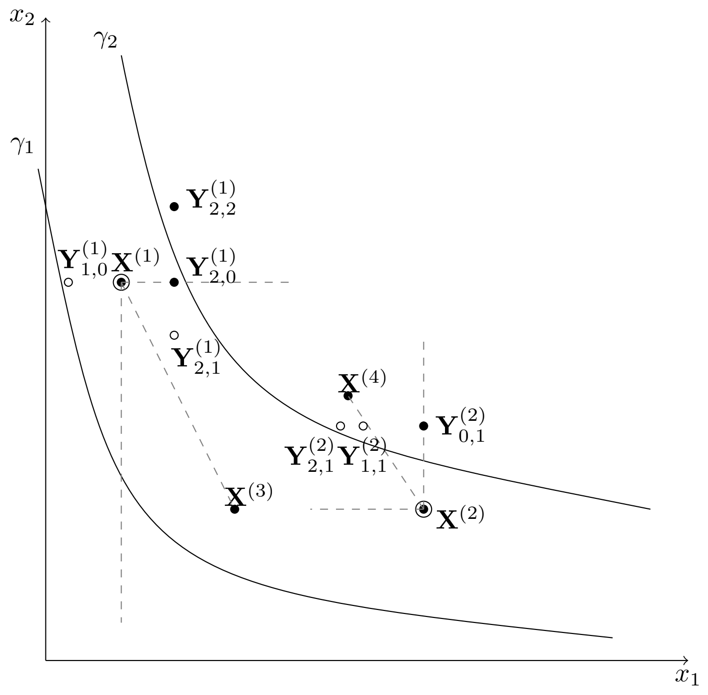

Figure 2: Illustration of splitting method for two base (elite) points $\mathbf{X}^{(1)}$ and $\mathbf{X}^{(2)}$, with assistant points $\mathbf{X}^{(3)}$ and $\mathbf{X}^{(4)}$, respectively. The relative distance between the base and assistant points determines the standard deviation of the sampling distribution. $\mathbf{Y}_{2,0}^{(1)}$, $\mathbf{Y}_{2,2}^{(1)}$ and $\mathbf{Y}_{0,1}^{(2)}$ are intermediate points, which are indicated by $\bullet$, while $\circ$ indicates the failed trial points.

sample one more time the first component, giving the second try $\mathbf{Y}_{2,0}^{(1)}$. Because the corresponding function value is better, this trial point is accepted as an intermediate point. Then based on $\mathbf{Y}_{2,0}^{(1)}$, the second component is updated. The vertical dotted segment starting from $\mathbf{X}^{(1)}$ is 1.5 times the length of standard deviation of normal distribution of the second component. Here, also the first try $\mathbf{Y}_{2,1}^{(1)}$ is rejected and the second try $\mathbf{Y}_{2,2}^{(1)}$ is accepted. As this is the final intermediate point, $\mathbf{Y}_{2,2}^{(1)}$ is added to $\mathcal{X}_2$. As for the second base point $\mathbf{X}^{(2)}$, the second component is updated before the first one. The first try for the second component $\mathbf{Y}_{0,1}^{(2)}$ has resulted in a good intermediate value, so that this change is accepted and an update the first component is attempted next. However, both two attempts $\mathbf{Y}_{1,1}^{(2)}$ and $\mathbf{Y}_{2,1}^{(2)}$ are rejected (as they have worse function value that $\mathbf{Y}_{0,1}^{(2)}$, so that $\mathbf{Y}_{0,1}^{(2)}$ is the last intermediate point, which is added to $\mathcal{X}_2$.

In Algorithm 2, apart from the maximum number of attempts, `MaxTry`, there are three parameters: the sample size N, the rarity parameter ϱ, and the scaling factor w. The rarity parameter ϱ is the proportion of the samples to be selected for the elite set, which is the most important control parameter for the SCO algorithm. When the objective function has relatively few modes, setting ϱ small significantly increases the convergence speed. However, when the surface of the objective function is very convoluted, like for the Rosenbrock function, or has many local optima, a larger ϱ

(even $\varrho = 1$) might be required. The reason is that when $\varrho < 1$ the splitting method discards "bad" samples from previous level sets. For complicated functions this could lead to an under-exploration of the space and therefore failure to converge to a global optimum.

To handle both situations with the same algorithm, we recommend three values for ϱ: $0.4, 0.8$ and 1. In particular, 0.4 and 0.8 are preferred for uni-modal problems, while 0.8 and 1 are preferred for multi-modal ones. In any case, $\varrho = 1$ is a safe choice for finding the global optimum, but may not lead to the fastest convergence. For $\varrho < 1$, the worst $(1 - \varrho)N$ particles are discarded, and the remaining population is split into N new particles. For $\varrho = 1$, no particle is discarded, and the whole population is split into N new particles.

The choice of the sample size N depends on the choice of ϱ and the dimension of the problem. In general, a larger dimension requires a larger N. For $\varrho = 1$, N could be taken a small multiple of the dimension. For example, in a 10-D problem one could take $N = 20$. When decreasing ϱ , N should be increased correspondingly.

The parameter w scales the standard deviations of each component of the independent multivariate normal distribution. In most cases, $w = 0.5$ works well. Although the proposed SCO algorithm has four parameters, all of them are easy to choose. Even simpler, one can take `MaxTry` = 5 and $w = 0.5$ by default, and then N and ϱ are the only parameters to be adjusted.

3. The DE and ABC Algorithms

For ease of comparison we summarize the DE and ABC algorithms.

The DE Algorithm

The DE algorithm, introduced in [5] in 1995, is a stochastic population-based optimization algorithm. It belongs to the class of evolutionary algorithms (EAs), which also includes genetic algorithms, evolutionary strategies and evolutionary programming. All evolutionary algorithms use similar operations, such as crossover, mutation and selection. The main difference between DE and the traditional EAs is the generation of new candidate solutions. In the DE algorithm, the mutation operation is used as a search mechanism. Then a selection operation with a greedy scheme is used to direct the search toward the prospective regions in the search space. The DE works as follows. Initially, a population of individuals is generated randomly and evaluated via the fitness (objective) function. Afterwards, for each individual in the population, an offspring candidate is constructed using the weighted difference of a pair of parent solutions. Then greedy selection is used to determine if an offspring candidate is accepted or rejected.

In our study, we only use the *DE/rand/1/bin* scheme, which is the classical version. This involves the following mutation step at each iteration. For the ith individual (out of a population of N) $\mathbf{X}^{(i)} = (X_1^{(i)}, \ldots, X_n^{(i)})$, construct a new vector $\mathbf{Y}^{(i)} = \left(Y_1^{(i)}, \ldots, Y_n^{(i)}\right)$ as

$$\mathbf{Y}^{(i)} = \mathbf{X}^{(R_1)} + F\,\left(\mathbf{X}^{(R_2)} + \mathbf{X}^{(R_3)}\right), \tag{3}$$

where $R_1 \neq R_2 \neq R_3$ are uniformly sampled from $\{1, \ldots, N\} \setminus \{i\}$, and F is a fixed scaling factor. After mutation a binary crossover between $\mathbf{Y}^{(i)}$ and $\mathbf{X}^{(i)}$ is applied, to obtain a trial vector $\widetilde{\mathbf{X}}^{(i)} = \left(\widetilde{X}_1^{(i)}, \ldots, \widetilde{X}_n^{(i)}\right)$; that is,

$$\widetilde{\mathbf{X}}^{(i)} = \left(B_1 Y_1^{(i)} + (1 - B_1)X_1^{(i)}, \ldots, B_n Y_n^{(i)} + (1 - B_n)X_n^{(i)}\right), \tag{4}$$

where $B_1, \ldots, B_n \overset{\text{iid}}{\sim} \mathsf{Ber}(p)$ and p is the crossover factor. The DE algorithm is summarized in Algorithm 3.

Algorithm 3 The DE Algorithm for Minimization

```
Input: Objective function $S$, population size $N$, scaling factor $F$ and crossover factor $p$.
Output: Final iteration number $t$ and sequence $(\mathbf{X}_{\text{best},1}, b_1), \ldots, (\mathbf{X}_{\text{best},t}, b_t)$ of best solutions and
    function values found at iteration $1, \ldots, t$, respectively.
 1: Generate a population $\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}$ via uniform sampling. Set iteration counter $t = 0$.
 2: while the stopping criterion is not met do
 3:   for $i = 1$ to $N$ do
 4:     Mutation: Construct a vector $\mathbf{Y}^{(i)}$ as in Eq. (3).
 5:     Crossover: Construct a trial vector $\widetilde{\mathbf{X}}^{(i)}$ by using binary crossover between $\mathbf{Y}^{(i)}$ and $\mathbf{X}^{(i)}$,
        as indicated in Eq. (4).
 6:     Selection: If $S(\widetilde{\mathbf{X}}^{(i)}) \leqslant S(\mathbf{X}^{(i)})$, set $\mathbf{X}^{(i)} = \widetilde{\mathbf{X}}^{(i)}$; otherwise, keep $\mathbf{X}^{(i)}$.
 7:   end for
 8:   $t = t + 1$
 9:   Set $b_t = \min\{S(\mathbf{X}^{(i)}), i = 1, \ldots, N\}$ and record the corresponding individual as $\mathbf{X}_{\text{best},t}$.
10: end while
11: return $\{(\mathbf{X}_{\text{best},k}, b_k), k = 1, \ldots, t\}$.
```

Although the DE algorithm is one of the most successful randomized optimization algorithms, its performance quite relies on the choice of the parameters. In the *DE/rand/1/bin* version, there are 3 parameters: the population size N, the scaling factor F, and the crossover factor p. In many comparative studies for DE performance, the *same* choice of parameters for all test functions is used. For example, in [14] all experiments use the setting $N = 100, F = 0.5$ and $p = 0.9$.

Even in such settings, DE can outperform many other algorithms in terms of the mean and variance of the final optimum. In our study, we not only compare the final optimum, but also the CPU time. To make a fair comparison, it is required to tune the DE parameters as efficiently as possible. For each experiment, we choose the parameters based on highly cited references on the DE algorithm, e.g., [5, 15], and then adjust them to the best we can find.

The ABC Algorithm

The ABC algorithm is a recently introduced evolutionary algorithm which is based on the intelligent foraging behavior of honey bees. The idea was first proposed in [16] and then published in [9]. In the ABC algorithm the objective value is interpreted as the amount of nectar at a food source (position). Artificial bees fly around in a multidimensional search space to seek out food sources. The colony contains three groups of bees who do different tasks: *employed bees* are associated with a specific food source, *onlookers* watch the dance of employed bees within the hive to choose a food source, and *scouts bees* search for food sources randomly.

Initially, a population of food source positions is generated randomly. Then, the food sources are exploited by employed bees and onlooker bees, and their continual exploitation will ultimately cause the food sources to become exhausted. Once this happens, an employed bee that was associated with that food source now becomes a scout bee and searches randomly again. The employed and onlooker bees choose new food sources depending on their own experience and that of their nest mates, and adjust their positions accordingly. In the basic form of the ABC algorithm, the number of employed bees and onlooker bees is equal to the number of food source positions.

There are three main step involved in each search cycle (iteration) of ABC the algorithm:

1. The employed bees are sent to a food source within the neighborhood of their previous food sources and evaluate the fitness (nectar amounts) of these food sources.

2. The employed bees share this information with the onlooker bees, who will select one of food sources. Onlooker bees choose a food source within the neighborhood of their selected food sources and, after visiting, evaluate the nectar amounts at these positions.

3. An employed bee of a food source that has not been replaced for many times becomes a scout and will search for a new food source randomly in the search space.

An artificial onlooker bee chooses a food source depending on the information (nectar amount) of each food source, which is communicated by dance by corresponding employed bees in the previous cycle. To determine to which food source an onlooker bee should go, the information is transformed into a probability

$$p_i = \frac{S_i}{\sum_{k=1}^{N} S_k}, \tag{5}$$

where S_i is the fitness of the ith food source and $i = 1, \ldots, N$, N is the total number of food sources. Note that each probability must be positive. If not, it must be converted to a positive number in such a way that a better food source always has a larger probability to be visited than a poorer one.

To construct a new food source position $\mathbf{Y}^{(i)} = \left(Y_1^{(i)}, \ldots, Y_n^{(i)}\right)$ from a previous position $\mathbf{X}^{(i)} = \left(X_1^{(i)}, \ldots, X_n^{(i)}\right)$, the ABC algorithm uses the following mechanism:

$$Y_j^{(i)} = X_j^{(i)} + \phi_{ij}(X_j^{(i)} - X_j^{(k)}), \tag{6}$$

where j is uniformly sampled from $\{1, \ldots, n\}$, k is uniformly sampled from $\{1, \ldots, N\} \setminus \{i\}$ and ϕ_{ij} is a uniform random number on the interval $[-1, 1]$.

To determine whether an employed bee becomes a scout, a parameter, referred to as "limit", is used, indicating the maximum number of cycles that a food source can not be improved. If the food source can not be improved within "limit" trials, the corresponding employed bee becomes a scout, and is sent to a new food source, whose location is generated randomly. As indicated in [17], the "limit" can be specified as the product of the number of food sources and the dimension of the objective function; that is,

$$\text{limit} = N \times n\ . \tag{7}$$

The ABC algorithm is summarized in Algorithm 4.

Since the invention of ABC in 2005, it has been well studied. For example, [9] evaluates its performance and compares it to GA, PSO, and particle swarm inspired evolutionary algorithm on a limited number of test functions, and [18] compares ABC to GA, DE, PSO and evolution strategies on a comprehensive set of problems. In the above the studies ABC has been found to be very effective and able to obtain very good results with a low computational cost. Therefore, involving ABC in our numerical experiments is meaningful. Note that although several modified versions of ABC algorithms are available, e.g., [19, 20], only the standard ABC is involved in our research.

Algorithm 4 The ABC Algorithm for Minimization.

```
Input: Objective function $S$, the number of food sources $N$.
Output: Final iteration number $t$ and sequence $(\mathbf{X}_{\text{best},1}, b_1), \ldots, (\mathbf{X}_{\text{best},t}, b_t)$ of best solutions and
    function values found at iteration $1, \ldots, t$, respectively.
 1: Generate a set of food sources $\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}$ via uniform sampling. Set iteration counter $t = 0$.
 2: while the stopping criterion is not met do
 3:   for $i = 1$ to $N$ do
 4:     Produce a new position $\mathbf{Y}^{(i)}$ for the employed bee from $\mathbf{X}^{(i)}$ by using (6).
 5:     Apply the greedy selection for employed bees: if $S(\mathbf{Y}^{(i)}) < S(\mathbf{X}^{(i)})$, set $\mathbf{X}^{(i)} = \mathbf{Y}^{(i)}$;
        otherwise, keep $\mathbf{X}^{(i)}$.
 6:   end for
 7:   Calculate the probability $p_i$ for each $\mathbf{X}^{(i)}$ according to Eq. (5) and $i = 1, \ldots, N$.
 8:   for $i = 1$ to $N$ do
 9:     Draw $I$ from $\{1, \ldots, N\}$ according to probability distribution $\{p_i\}$. Produce a new position
        $\mathbf{Y}^{(I)}$ for the $i$th onlooker bee from $\mathbf{X}^{(I)}$, according to Eq. (6).
10:     Apply the greedy selection for onlooker bees: if $S(\mathbf{Y}^{(I)}) < S(\mathbf{X}^{(I)})$, set $\mathbf{X}^{(I)} = \mathbf{Y}^{(I)}$;
        otherwise, keep $\mathbf{X}^{(I)}$.
11:   end for
12:   If there exists $\mathbf{X}^{(k)} = \left(X_1^{(k)}, \ldots, X_n^{(k)}\right)$ whose fitness did not improve for a "limit" number
      of trials, then set

          $$X_j^{(k)} = \text{Lower}_j + U(\text{Upper}_j - \text{Lower}_j), \quad U \sim \mathsf{U}(0,1),$$

      for $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, N\}$. $\text{Lower}_j$ and $\text{Upper}_j$ are given fixed lower and upper
      bound on the $j$th component of the food position.
13:   Set $t = t + 1$.
14:   Set the best position $\mathbf{X}_{\text{best},t}$ and the corresponding fitness $b_t = S(\mathbf{X}_{\text{best},t})$.
15: end while
16: return $\{(\mathbf{X}_{\text{best},k}, b_k), k = 1, \ldots, t\}$.
```

In our experiments, we used the `MATLAB` code of the ABC algorithm version 2, which was released on Dec.14, 2009 and can be downloaded from the ABC homepage `http://mf.erciyes.edu.tr/abc/`. In this implementation, ϕ_{ij} in Eq. (6) is set to $\phi_{ij} = 2(U - 0.5)$, with $U \sim \mathsf{U}(0,1)$. Thus, there is only one parameter left, that is, the number of food source positions N, since the "limit" can be determined by Eq. (7). Similar with DE, for each problem, a good parameter will be chosen to guarantee a fair comparison.

4. Experiments

4.1. Benchmark functions

To test our proposed algorithm SCO, we used a well-known suite of 23 benchmark functions that were first introduced in [8] to compare the classical EP and the FEP algorithms. The same functions were used to compare the performances of DE, PSO, the attractive and repulsive PSO (arPSO), and a simple EA in [14]. Also, [21] has compared four different Evolutionary strategies (namely, Canonical ES, Fast ES (FES), Covariance Matrix Adaptation ES (CMAES) and ES Learned with Automatic Termination (ESLAT)) using these 23 functions. From these comparative works, the overall performance of DE was found to be much better than that of other methods on this test suite. Recently in [17], the same functions were used to compare the ABC algorithm with many popular evolutionary algorithm, where the results of [21] were also included in the comparison. In these comparisons, ABC was found to be superior to the other methods. Because of the widespread use of this test suite, by directly comparing our algorithm with DE and ABC only, we can indirectly compare with many other algorithms using the existing comparative studies. In the numerical experiments we do not make reference to any probability density function pdf f. Consequently, below we have used the conventional notation f for the objective function, rather than S.

The suite of test problems contains a diverse set of functions, including unimodal and multimodal functions. Note that we only consider minimization problems. We have conducted two sets of numerical experiments. In the first set, we have divided the 23 test functions into three groups: Group A contains functions $f_1, \ldots, f_7$, which are all unimodal, Group B are functions $f_8, \ldots, f_{13}$, which are multimodal functions with many local minima and Group C ($f_{14}, \ldots, f_{23}$) are multimodal functions with only a few local minima. Functions in Group B are such that the number of local minima increases exponentially with the dimension, so they appear to be more difficult than functions in Group C. Functions in Group A and B are all in 30 dimensions and functions in Group C are in 2, 4 or 6 dimensions. In the second set, we use 100-D f_5, $f_8, \ldots, f_{13}$ in order to test the performance of DE, ABC and SCO in high dimensions. The problems in this set of experiments are much more difficult.

4.2. Setting of Experiments

For each function, a level of accuracy is specified. Suppose that the algorithm returns a value $\widehat{\gamma}$. For functions whose minimum value is 0, if $\widehat{\gamma} < 10^{-10}$ we consider that the minimum solution has been found. For other functions, whose minimum value is not equal to 0, we consider that the minimum has been found if $\widehat{\gamma}$ agrees with the true minimum in at least 8 digits after the decimal point. The stopping criterion for the three methods in the experiments is to terminate when the minimum value of the population has reached the accuracy level or the running time of algorithm exceeds the predefined maximum CPU time, which is 600s for the Experiment Set 1 and 1,800s for the Experiment Set 2.

In terms of the algorithms, all the parameters are specified in a parameter table before displaying the experimental outcomes. Since [14] has shown that DE can solve all the 23 problems, we first select a good set of parameters for DE. These parameters are chosen to make DE work in a stable and efficient manner. The parameters of the ABC and SCO algorithms are easier to select. For example, the population sizes will be similar to those of DE. If ABC and SCO fail to converge toward the global minimum within a reasonable CPU time, we regard the experiment to have failed for such functions and we do not display the parameters and outcomes. It is worth noting that we mainly compare for speed, rather than accuracy, when all the methods converge to the global minimum. For each experiment, the average CPU time of best performing algorithm is highlighted in **gray**.

All the experiments have been coded in MATLAB R2014b, and conducted on a desktop personal computer with Intel(R) Core(TM) i7-4970 CPU @ 3.60GHz. Each experiment is repeated 10 times independently, and the minimum (Min), mean (Mean), and maximum (Max) of the function values are reported, as well as the CPU time in seconds (CPU) and the average number of iterations ($\bar{T}$).

4.3. Experiment Set 1: 30-D or Less Functions

A. 30-D Unimodal Functions

1. The First De Jong function (Sphere):

$$f_1(\mathbf{x}) = \sum_{i=1}^{n-1} x_i^2, \quad \mathbf{x} \in [-100, 100]^n. \tag{8}$$

2. The Schwefel Problem 2.22:

$$f_2(\mathbf{x}) = \sum_{i=1}^{n} |x_i| + \prod_{i=1}^{n} |x_i|, \quad \mathbf{x} \in [-10, 10]^n. \tag{9}$$

3. The Schwefel Problem 1.2:

$$f_3(\mathbf{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2, \quad \mathbf{x} \in [-100, 100]^n. \tag{10}$$

4. The Schwefel Problem 2.21:

$$f_4(\mathbf{x}) = \max\{|x_i|\}, \quad \mathbf{x} \in [-100, 100]^n. \tag{11}$$

5. The Second De Jong function (Rosenbrock):

$$f_5(\mathbf{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2, \quad \mathbf{x} \in [-30, 30]^n. \tag{12}$$

6. The Step function:

$$f_6(\mathbf{x}) = \sum_{i=0}^{n-1} \left(\left\lfloor x_i + \frac{1}{2} \right\rfloor \right)^2, \quad \mathbf{x} \in [-100, 100]^n. \tag{13}$$

7. The Fourth De Jong function (Quartic with random noise):

$$f_7(\mathbf{x}) = \sum_{i=1}^{n} (i\, x_i^4 + \eta), \quad \mathbf{x} \in [-1.28, 1.28]^n, \quad \eta \sim \mathsf{U}(0, 1). \tag{14}$$

The experiments in Group A are aimed at testing and comparing the convergence of the three methods. The parameters of DE are tuned based on the work in [5, 15]. With the selected values, it works efficiently for all 10 runs. A slight decrease in population size may speed up DE, but may affect convergence. ABC and SCO use the same population size as DE, for fair comparison.

In our experiments with the function f_5, the ABC algorithm can get a solution near the global minimum, but does not converge toward it within our specified tolerance level, which is in accordance with [17, 9], so the results were not recorded. For f_3 we observed the same phenomenon, so we omitted the results. The parameters used in these experiments are summarized in Table 1 and the outcomes of the experiments are given in Table 2. The parameters of the unrecorded results are represented with a $\times$ in the parameters tables.

Table 1: The parameters for the three algorithms for 30-D $f_1, \ldots, f_7$.

Function	n	DE			ABC	SCO		
		N	F	p	N	N	ϱ	w
f_1	30	30	0.5	0.2	30	30	0.4	0.5
f_2	30	30	0.5	0.9	30	30	0.4	0.5
f_3	30	30	0.7	0.9	$\times$	30	0.4	0.5
f_4	30	30	0.5	0.2	30	30	0.8	0.5
f_5	30	50	0.7	0.9	$\times$	50	0.8	0.5
f_6	30	30	0.5	0.7	30	30	0.4	0.5
f_7	30	30	0.5	0.2	30	30	0.4	0.5

Table 2: Results for the 30-dimensional functions $f_1, \ldots, f_7$. The experiments were repeated 10 times and the average, minimum, maximum of the objective function were recorded. CPU is the average time and $\bar{T}$ is the average numbers of iterations.

Function	n	Method	Min	Mean	Max	CPU	$\bar{T}$
		DE	8.3296×10^{-11}	9.2307×10^{-11}	9.6917×10^{-11}	1.07	861.4
f_1	30	ABC	3.8247×10^{-11}	7.8605×10^{-11}	9.8754×10^{-11}	1.10	931.1
		SCO	1.1478×10^{-12}	2.8822×10^{-11}	7.2469×10^{-11}	0.20	12.6
		DE	8.3879×10^{-11}	9.6256×10^{-11}	9.9970×10^{-11}	3.96	1636.1
f_2	30	ABC	7.3659×10^{-11}	8.9666×10^{-11}	9.9947×10^{-11}	1.93	1582.5
		SCO	2.5659×10^{-11}	5.5492×10^{-11}	8.3617×10^{-11}	0.50	22.8
f_3		DE	8.9752×10^{-11}	9.5390×10^{-11}	9.9600×10^{-11}	19.79	8640.4
	30	**SCO**	9.5598×10^{-11}	9.7618×10^{-11}	9.9978×10^{-11}	15.42	848.9
		DE	9.1888×10^{-11}	9.5490×10^{-11}	9.9530×10^{-11}	8.50	4794.5
f_4	30	ABC	9.2581×10^{-11}	9.7329×10^{-11}	9.9771×10^{-11}	34.63	32224.3
		SCO	8.1875×10^{-11}	9.1788×10^{-11}	9.9736×10^{-11}	2.48	299.8
		DE	6.8419×10^{-11}	8.5554×10^{-11}	9.6158×10^{-11}	37.94	9455.8
f_5	30	SCO	9.9630×10^{-11}	9.9726×10^{-11}	9.9871×10^{-11}	304.22	6772.7
		DE	0	0	0	0.68	280.6
f_6	30	ABC	0	0	0	0.31	222.4
		SCO	0	0	0	0.086	8.7
		DE	13.72410384	13.72410384	13.72410384	0.95	381.4
f_7	30	ABC	13.72410383	13.72410384	13.72410384	0.58	359.9
		SCO	13.72410383	13.72410383	13.72410383	0.22	6.8

Table 2 shows that the DE algorithm and our proposed SCO algorithm converge to the correct minimum for the seven functions. Moreover, the SCO algorithm outperforms both the DE and ABC algorithm for functions $f_1, \ldots, f_4, f_6, f_7$ in terms of average CPU time. Even though the ABC fails in f_3 and f_5 and perform badly in f_4, it is almost twice faster than DE on functions f_2, f_6 and f_7. DE has best performance in function f_5, converging to the global minimum much faster than the SCO algorithm. Function f_5 is the Rosenbrock function, whose global minimum lies inside a long, narrow and parabolic shaped flat valley and is very difficult to find. As indicated in Figure 3, although SCO can find the valley region faster than DE, the convergence rate to the global minimum is slower.

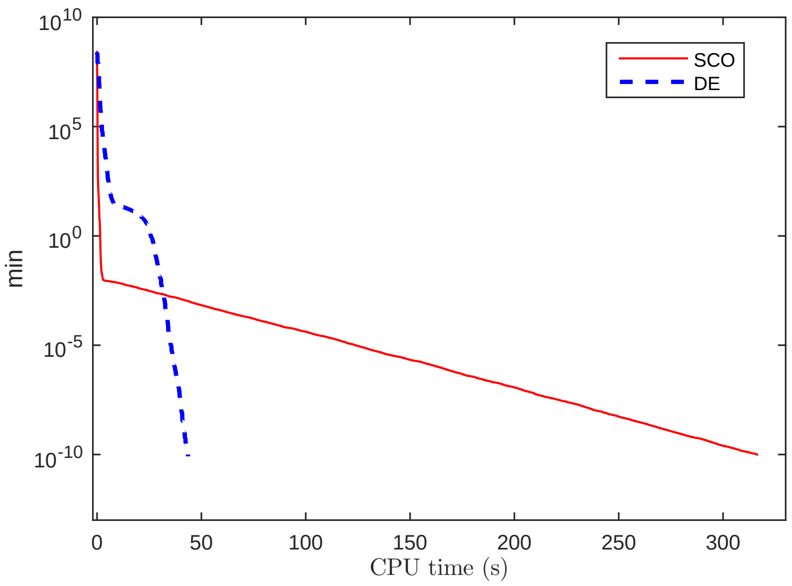

Figure 3: Evolution of DE and SCO algorithms for the Rosenbrock function f_5

Note that again for some of the functions, decreasing the population size might speed up the algorithms. However, if the convergence is not affected, all the three methods can work with a same population size, so the the comparison in CPU time-consuming is still similar to that of Table 2.

B. 30-D Multimodal Functions

The functions $f_8, \ldots, f_{13}$ are multimodal with many local minima, and the number of local minima increases exponentially as the dimension of the function increases. In this group, the dimensions of $f_8, \ldots, f_{13}$ are 30 and the level of accuracy is set to 10^{-11}. The parameters of the three methods are indicated in Table 3. The parameters of DE algorithm are adapted from [5, 15]. For the ABC algorithm, the size of the population is 30 in all cases, where the "limit" is $N \times n = 90$. The SCO algorithm uses the same size of population, and ϱ takes value 1 for $f_8, \ldots, f_{11}$ and 0.8 for f_{12} and f_{13}. The Table 4 summarizes the results of ten runs.

1. The Schwefel Problem 2.26:

$$f_8(\mathbf{x}) = \sum_{i=0}^{n-1} -x_i \sin\left(\sqrt{|x_i|}\right), \quad \mathbf{x} \in [-500, 500]^n. \tag{15}$$

2. The Rastrigin function:

$$f_9(\mathbf{x}) = \sum_{i=1}^{30} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right], \quad \mathbf{x} \in [-5.12, 5.12]^n. \tag{16}$$

3. The Ackley function:

$$f_{10}(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + \mathrm{e}, \quad \mathbf{x} \in [-30, 30]^n. \tag{17}$$

4. The Griewank function:

$$f_{11}(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1, \quad \mathbf{x} \in [-600, 600]^n. \tag{18}$$

5. The Penalized function 1:

$$\begin{aligned} f_{12}(\mathbf{x}) = \frac{\pi}{n}&\left\{10(\sin(\pi y_1))^2 + \sum_{i=1}^{n-1}(y_i - 1)^2[1 + 10\sin^2(\pi y_i + 1)] + (y_n - 1)^2\right\} \\ &+ \sum_{i=1}^{n} u(x_i, 10, 100, 4), \quad \mathbf{x} \in [-50, 50]^n, \end{aligned} \tag{19}$$

where $y_i = 1 + \frac{1}{4}x_i$ and

$$u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a, \\ 0, & -a \leqslant x_i \leqslant a, \\ k(-x_i - a)^m, & x_i < -a. \end{cases}$$

6. The Penalized function 2:

$$\begin{aligned} f_{13}(\mathbf{x}) = 0.1&\left\{\sin^2(3\pi x_1) + \sum_{i=1}^{n-1}(x_i - 1)^2[1 + \sin^2(3\pi x_{i+1})] + (x_n - 1)[1 + \sin^2(2\pi x_n)]\right\} \\ &+ \sum_{i=1}^{n} u(x_i, 5, 100, 4), \quad \mathbf{x} \in [-50, 50]^n, \end{aligned} \tag{20}$$

where y_i and u are defined as in f_{12}.

Table 3: The parameters selection for the three algorithms for 30-D $f_8, \ldots, f_{13}$.

Function	n	DE			ABC	SCO		
		N	F	p	N	N	ϱ	w
f_8	30	30	0.5	0	30	30	1	0.5
f_9	30	25	0.5	0	30	30	1	0.5
f_{10}	30	20	0.5	0.1	30	30	1	0.5
f_{11}	30	20	0.5	0.1	30	30	1	0.5
f_{12}	30	30	0.5	0.2	30	30	0.8	0.5
f_{13}	30	30	0.5	0.2	30	30	0.8	0.5

Table 4: Results for the 30-dimensional functions $f_8, \ldots, f_{13}$. The experiments were repeated 10 times and the average, minimum, maximum of the objective function were recorded. CPU is the average time and $\bar{T}$ is the average numbers of iterations.

Function	n	Method	Min	Mean	Max	CPU	$\bar{T}$
f_8	30	DE	-12569.48661817	-12569.48661817	-12569.48661816	2.30	1087.1
		ABC	-12569.48661817	-12569.48661817	-12569.48661817	3.13	2015.8
		SCO	-12569.48661817	-12569.48661817	-12569.48661817	1.69	95.7
f_9	30	DE	7.9837×10^{-11}	9.2210×10^{-11}	9.9522×10^{-11}	2.41	$1,171.5$
		ABC	2.1149×10^{-11}	5.8284×10^{-11}	9.9918×10^{-11}	1.58	$1,273.7$
		SCO	4.2633×10^{-14}	3.6740×10^{-11}	8.9090×10^{-11}	1.08	93.2
f_{10}	30	DE	9.5295×10^{-11}	9.8002×10^{-11}	9.9953×10^{-11}	2.84	$1,401.9$
		ABC	7.6178×10^{-11}	8.9128×10^{-11}	9.9267×10^{-11}	5.86	$1,634.6$
		SCO	2.1953×10^{-11}	6.5364×10^{-11}	9.3430×10^{-11}	1.66	61.7
f_{11}	30	**DE**	8.2883×10^{-11}	9.1398×10^{-11}	9.5228×10^{-11}	1.52	938.4
		ABC	1.6721×10^{-11}	6.2046×10^{-11}	9.9988×10^{-11}	2.15	$1,165.8$
		SCO	3.8907×10^{-12}	4.3107×10^{-11}	8.5327×10^{-11}	1.42	43.8
f_{12}	30	DE	7.6932×10^{-11}	9.0782×10^{-11}	9.9077×10^{-11}	3.57	840.6
		ABC	4.2338×10^{-11}	7.9264×10^{-11}	9.8755×10^{-11}	2.43	834.9
		SCO	1.3005×10^{-11}	6.6929×10^{-11}	9.6349×10^{-11}	2.58	33.5
f_{13}	30	DE	7.4773×10^{-11}	8.8599×10^{-11}	9.8990×10^{-11}	3.54	853.5
		ABC	3.5438×10^{-11}	7.1386×10^{-11}	9.8533×10^{-11}	2.61	947.3
		SCO	2.3145×10^{-11}	5.6150×10^{-11}	9.6540×10^{-11}	2.60	35.0

It is obvious that all the three methods have reached the global optimum for all the 6 functions. When comparing the average CPU time, the SCO is superior to both DE and ABC on functions $f_8, \ldots, f_{11}$. As for functions f_{12} and f_{13}, the ABC and SCO perform better than DE. We used $\varrho = 0.8$ in SCO for functions f_{12} and f_{13}, where $\varrho = 1$ also performs well but is a little slower in convergence. If so, in terms of CPU time, ABC will outperform SCO on functions f_{12} and f_{13}. Thus, for multimodal functions, we recommend to use $\varrho = 1$ for a safe choice, but also $\varrho = 0.8$ as a more efficient choice.

C. Low-Dimensional Multimodal Functions

1. The Fifth De Jong function (Shekel's Foxholes):

$$f_{14}(\mathbf{x}) = \frac{1}{0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}}, \quad \mathbf{x} \in [-65.536, 65.536]^2, \tag{21}$$

where

$$(a_{1j}) = (\underbrace{\mathbf{c}, \ldots, \mathbf{c}}_{5 \text{ times}}), \quad \text{with} \quad \mathbf{c} = (-32, -16, 0, 16, 32),$$

and

$$(a_{2j}) = (\underbrace{-32, \ldots, -32}_{5 \text{ times}}, \underbrace{-16, \ldots, -16}_{5 \text{ times}}, \underbrace{0, \ldots, 0}_{5 \text{ times}}, \underbrace{16, \ldots, 16}_{5 \text{ times}}, \underbrace{32, \ldots, 32}_{5 \text{ times}}).$$

2. The Kowalik function:

$$f_{15}(\mathbf{x}) = \sum_{j=0}^{10} \left(a_j - \frac{x_1(b_j^2 + b_j x_2)}{b_j^2 + b_j x_3 + x_4} \right)^2, \quad \mathbf{x} \in [-5, 5]^4, \tag{22}$$

where

$$(a_j) = (0.1957, 0.1947, 0.1735, 0.16, 0.0844, 0.0627, 0.0456, 0.0342, 0.0323, 0.0235, 0.0246),$$

and

$$(b_j) = (4, 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{10}, \frac{1}{12}, \frac{1}{14}, \frac{1}{16}).$$

3. The Six-hump Camel function:

$$f_{16}(\mathbf{x}) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4, \quad \mathbf{x} \in [-5, 5]^2. \tag{23}$$

4. The Branin–Hoo function:

$$f_{17}(\mathbf{x}) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10, \quad x_1 \in [-5, 10], \quad x_2 \in [0, 15]. \tag{24}$$

5. The Goldstein–Price function:

$$\begin{aligned} f_{18}(\mathbf{x}) = &\left[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right] \times \\ &\left[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right], \quad \mathbf{x} \in [-2, 2]^2. \end{aligned} \tag{25}$$

6. The 3-dimensional Hartmann function:

$$f_{19}(\mathbf{x}) = -\sum_{j=1}^{4} c_j \exp\left(-\sum_{i=1}^{3} a_{ij}(x_i - p_{ij})^2\right), \quad \mathbf{x} \in [0, 1]^4, \tag{26}$$

where

$$(a_{ij}) = \begin{pmatrix} 3.0 & 0.1 & 3.0 & 0.1 \\ 10 & 10 & 10 & 10 \\ 30 & 35 & 30 & 35 \end{pmatrix}, \quad (c_j) = (1, 1.2, 3, 3.2), \quad \text{and} \quad (p_{ij}) = 10^{-4}\begin{pmatrix} 3689 & 4699 & 1091 & 381 \\ 1170 & 4387 & 8732 & 5743 \\ 2673 & 7470 & 5547 & 8828 \end{pmatrix}.$$

7. The 6-dimensional Hartmann function:

$$f_{20}(\mathbf{x}) = -\sum_{j=1}^{4} c_j \exp\left(-\sum_{i=1}^{6} a_{ij}(x_i - p_{ij})^2\right), \quad \mathbf{x} \in [0, 1]^6. \tag{27}$$

where

$$(a_{ij}) = \begin{pmatrix} 10 & 0.05 & 3 & 17 \\ 3 & 10 & 3.5 & 8 \\ 17 & 17 & 1.7 & 0.05 \\ 3.5 & 0.1 & 10 & 10 \\ 1.7 & 8 & 17 & 0.1 \\ 8 & 14 & 8 & 14 \end{pmatrix}, \quad (c_j) = (1, 1.2, 3, 3.2), \quad \text{and} \quad (p_{ij}) = 10^{-4}\begin{pmatrix} 1312 & 2329 & 2348 & 4047 \\ 1696 & 4135 & 1451 & 8828 \\ 5569 & 8307 & 3522 & 8732 \\ 124 & 3736 & 2883 & 5743 \\ 8283 & 1004 & 3047 & 1091 \\ 5886 & 9991 & 6650 & 381 \end{pmatrix}.$$

8. The Shekel function 5:

$$f_{21}(\mathbf{x}) = -\sum_{j=1}^{5} \left((\mathbf{x}-\mathbf{a}_j)^\top(\mathbf{x}-\mathbf{a}_j)+c_j\right)^{-1}, \quad \mathbf{x} \in [0,1]^4, \tag{28}$$

where $(c_j) = (0.1, 0.2, 0.2, 0.4, 0.4, 0.6, 0.3, 0.7, 0.5, 0.5)$ and

$$(\mathbf{a}_1 \ldots, \mathbf{a}_{10}) = (a_{ij}) = \begin{pmatrix} 4 & 1 & 8 & 6 & 3 & 2 & 5 & 8 & 6 & 7 \\ 4 & 1 & 8 & 6 & 7 & 9 & 3 & 1 & 2 & 3 \\ 4 & 1 & 8 & 6 & 3 & 2 & 5 & 8 & 6 & 7 \\ 4 & 1 & 8 & 6 & 7 & 9 & 3 & 1 & 2 & 3 \end{pmatrix}.$$

Notice that the Shekel function 5 only uses the first 5 columns of the matrix (a_{ij}). The Shekel functions 7 and 10 below use respectively 7 and 10 columns of the same matrix.

9. The Shekel function 7:

$$f_{22}(\mathbf{x}) = -\sum_{j=1}^{7} \left((\mathbf{x}-\mathbf{a}_j)^\top(\mathbf{x}-\mathbf{a}_j)+c_j\right)^{-1}, \quad \mathbf{x} \in [0,1]^4. \tag{29}$$

10. The Shekel function 10:

$$f_{23}(\mathbf{x}) = -\sum_{j=1}^{10} \left((\mathbf{x}-\mathbf{a}_j)^\top(\mathbf{x}-\mathbf{a}_j)+c_j\right)^{-1}, \quad \mathbf{x} \in [0,1]^4. \tag{30}$$

Functions $f_{14}, \ldots, f_{23}$ have low dimensions (2, 4, or 6), but some of them are hard to optimize. The ABC algorithm fails on f_{15}, which has been shown in [17], so we will not compare the performance of ABC algorithm for function f_{15}. The parameters of DE algorithm are adapted from existing experiments in [14, 15] or chosen by us based on the best recommendations we could find. Table 5 summarizes the values of parameters and the outcomes are in Table 6.

Table 5: The parameters for the three algorithms for $f_{14}, \ldots, f_{23}$.

Function	n	DE			ABC	SCO		
		N	F	p	N	N	ϱ	w
f_{14}	2	20	0.5	0.2	30	30	1	0.5
f_{15}	4	50	0.5	0.9	$\times$	50	0.8	0.5
f_{16}	2	20	0.5	0.9	20	20	0.8	0.5
f_{17}	2	20	0.5	0.9	20	20	0.8	0.5
f_{18}	2	20	0.5	0.9	40	30	0.8	0.5
f_{19}	4	20	0.5	0.9	20	20	0.8	0.5
f_{20}	6	30	0.5	0.2	30	30	0.8	0.5
f_{21}	4	50	0.5	0.7	30	50	0.8	0.5
f_{22}	4	50	0.5	0.9	30	50	0.8	0.5
f_{23}	4	50	0.5	0.9	30	50	0.8	0.5

Table 6: Results for functions $f_{14}, \dots, f_{23}$. The experiments were repeated 10 times and the average, minimum, maximum of the objective function were recorded. CPU is the average time and $\bar{T}$ is the average numbers of iterations.

Function	n	Method	Min	Mean	Max	CPU	$\bar{T}$
f_{14}	2	**DE**	0.99800384	0.99800388	0.99800399	0.041	35.4
		ABC	0.99800384	0.99800385	0.99800388	0.086	37.4
		SCO	0.99800384	0.99800386	0.99800396	0.058	21.6
f_{15}	4	**DE**	3.0749×10^{-4}	3.0750×10^{-4}	3.0750×10^{-4}	0.24	109.9
		SCO	3.0750×10^{-4}	3.0750×10^{-4}	3.0750×10^{-4}	7.74	$1,737.7$
f_{16}	2	DE	-1.03162843	-1.03162813	-1.03162793	0.040	30.2
		ABC	-1.03162844	-1.03162828	-1.03162802	0.066	38.6
		SCO	-1.03162837	-1.03162822	-1.03162802	0.031	11.7
f_{17}	2	**DE**	0.39788974	0.39789434	0.39789878	0.026	32.1
		ABC	0.39788757	0.39789190	0.39789544	0.063	50.6
		SCO	0.39788844	0.39789201	0.39789541	0.022	14.0
f_{18}	2	**DE**	3	3	3	0.10	58.4
		ABC	3	3	3	2.29	643.4
		SCO	3	3	3	0.12	28.8
f_{19}	4	**DE**	-3.86277976	-3.86277974	-3.86277971	0.040	39.3
		ABC	-3.86277977	-3.86277974	-3.86277971	0.094	115.2
		SCO	-3.86277978	-3.86277976	-3.86277973	0.033	12.6
f_{20}	6	DE	-3.32236570	-3.32236532	-3.32236502	0.14	175.7
		ABC	-3.32236708	-3.32236603	-3.32236516	0.098	121.2
		SCO	-3.32236751	-3.32236642	-3.32236522	0.040	13.9
f_{21}	4	**DE**	-10.15319812	-10.15319325	-10.15319018	0.21	75.3
		ABC	-10.15319844	-10.15319479	-10.15319066	0.17	109.6
		SCO	-10.15319937	-10.15319537	-10.15319157	0.13	14.0
f_{22}	4	**DE**	-10.40294054	-10.40294021	-10.40294002	0.22	76.9
		ABC	-10.40294055	-10.40294031	-10.40294001	0.27	171.6
		SCO	-10.40294054	-10.40294035	-10.40294001	0.16	17.0
f_{23}	4	**DE**	-10.53640740	-10.53640300	-10.53640021	0.19	63.8
		ABC	-10.53640776	-10.53640391	-10.53640010	0.27	168.2
		SCO	-10.53640930	-10.53640507	-10.53640002	0.16	16.5

Among the 10 functions, f_{15} is challenging for all three methods: ABC fails it; DE and SCO have risks to get in a poor local minimum. Nevertheless, we still get ten continuous successful runs for both DE and SCO. However, SCO consumes much more CPU time than DE. Similar to Figure 3, SCO can quickly find good solutions close to the global optimizer, but has a smaller convergence rate than DE.

Beside function f_{15}, both DE and SCO could solve all the other functions $f_{14}, \dots, f_{23}$. For functions $f_{14}, f_{16}, \dots, f_{19}$, f_{22} and f_{23}, the performance of DE and SCO are close and better than that of ABC. In particular, for function f_{18}, DE and SCO are almost 20 times faster than ABC. For functions f_{20} and f_{21}, ABC and SCO are slightly superior to DE. Overall, in Experiment Set 1, SCO has a better performance than DE and ABC for all the functions except f_{14} and f_{15}.

4.4. Experiment Set 2: 100-D Functions

This experiment set is to compare the performance of the three methods for high-dimensional functions. The test functions are some of the most difficult problems in the test suite, namely f_5,

$f_8, \ldots, f_{13}$ in 100 dimensions.

For function 100-D f_5, the population sizes for DE and SCO are increased to 100, while for the other functions the population sizes for the three methods are exactly the same as in the 30-D case (Table 3). Regarding DE, for the 100-D functions f_5 and f_8, the selection of the other parameters is slightly different from that in the 30-D case, because DE did not perform well with the original parameters in terms of finding the minimum and CPU time. All the parameters are summarized in Table 7. The results of the experiments are shown in Table 8.

Table 7: The parameters for the three algorithms for 100-D $f_5, f_8 \ldots, f_{13}$.

Function	n	DE			ABC	SCO		
		N	F	p	N	N	ϱ	w
f_5	100	100	0.5	0.8	$\times$	100	0.8	0.5
f_8	100	30	0.7	0.2	30	30	1	0.5
f_9	100	25	0.5	0	30	30	1	0.5
f_{10}	100	20	0.5	0.1	30	30	1	0.5
f_{11}	100	20	0.5	0.1	30	30	1	0.5
f_{12}	100	30	0.5	0.2	30	30	0.8	0.5
f_{13}	100	30	0.5	0.2	30	30	0.8	0.5

Table 8: Results for the functions $f_5, f_8, \ldots, f_{13}$ in 100-dimension. The experiments were repeated 10 times and the average, minimum, maximum of the objective function were recorded. CPU is the average time and $\bar{T}$ is the average numbers of iterations.

Function	n	Method	Min	Mean	Max	CPU	$\bar{T}$
f_5	100	**DE**	9.2893×10^{-11}	9.7321×10^{-11}	9.9908×10^{-11}	965.42	48320.8
		SCO	6.8320×10^{-11}	9.6005×10^{-11}	9.9983×10^{-11}	$1,464.85$	$5,516.6$
f_8	100	DE	-41898.28872722	-41898.28872722	-41898.28872722	17.73	4192.8
		ABC	-41898.28872724	-41898.28872723	-41898.28872722	16.29	9675.3
		SCO	-41898.28872724	-41898.28872723	-41898.28872722	9.19	140.0
f_9	100	DE	8.3554×10^{-11}	9.4213×10^{-11}	9.9936×10^{-11}	15.28	$4,276.0$
		ABC	7.6178×10^{-11}	8.9128×10^{-11}	9.9267×10^{-11}	5.77	4342.7
		SCO	3.7002×10^{-11}	5.3482×10^{-11}	9.7948×10^{-11}	5.01	92.7
f_{10}	100	DE	9.3967×10^{-11}	9.7685×10^{-11}	9.9900×10^{-11}	12.90	4494.7
		ABC	8.9572×10^{-11}	9.4292×10^{-11}	9.9637×10^{-11}	21.69	5675.7
		SCO	5.3107×10^{-11}	7.7458×10^{-11}	9.8656×10^{-11}	6.48	72.3
f_{11}	100	DE	9.3395×10^{-11}	9.6480×10^{-11}	9.9961×10^{-11}	9.04	$2,904.5$
		ABC	6.0127×10^{-11}	7.8212×10^{-11}	9.8108×10^{-11}	7.11	$3,548.3$
		SCO	2.5731×10^{-11}	5.9148×10^{-11}	8.8098×10^{-11}	4.33	46.3
f_{12}	100	DE	9.6326×10^{-11}	9.7523×10^{-11}	9.9364×10^{-11}	25.67	$3,665.0$
		ABC	8.3713×10^{-11}	8.9108×10^{-11}	9.8624×10^{-11}	9.32	$2,991.0$
		SCO	3.4134×10^{-11}	6.9295×10^{-11}	9.4202×10^{-11}	10.38	36.0
f_{13}	100	DE	9.2595×10^{-11}	9.6520×10^{-11}	9.9168×10^{-11}	26.39	$3,748.0$
		ABC	8.4791×10^{-11}	9.2497×10^{-11}	9.7289×10^{-11}	10.03	$3,230.0$
		SCO	5.8721×10^{-11}	7.9251×10^{-11}	9.4134×10^{-11}	10.79	40.0

For function 100-D f_5, both DE and SCO converged toward the optimum exactly, but SCO consumes more CPU time. Since ABC failed on 30-D f_5, it is not included in the 100-D case. Overall, for the 100-D functions $f_8, \ldots, f_{13}$, all three methods can find the global minimum. As for the CPU time, the SCO and ABC algorithms perform better than DE, except for f_{10}. Specifically, on functions f_8 and f_{11}, ABC is a little better than DE, but both of them have worse performance than SCO. For functions f_9, f_{12} and f_{13}, SCO performs quite close to ABC and they are almost twice faster than DE in CPU time. On function f_{10}, ABC has a lower performance than SCO and DE, where SCO is twice faster than DE.

5. Discussion and Conclusion

In this paper we introduced the splitting method for continuous optimization (SCO) and compared its performance with that of the DE and ABC algorithms through two sets of numerical experiments based on a widely used suite of test functions. All the 23 functions used in first set of experiments were of dimension 30 or less and these were divided into three groups (A,B,C), depending on their number of modes. Overall, both SCO and DE could reliably converge toward the minimum values for all 23 functions, while ABC failed on f_3, f_5 and f_{15} for various reasons. From this aspect, SCO outperforms ABC. When also considering the CPU time consumption, SCO performs favorably compared with DE and ABC. In particular, for functions in Group A, SCO has the fastest convergence rate among the three methods, except for function 30-D f_5. Similarly, for functions in Group B, although all three methods work well, SCO is consistently the best performing. In Group C, except for function f_{15} where SCO consumes more CPU time than DE, SCO is the most efficient. The second set of experiments has shown than SCO also can perform well on high dimensional multi-modal problems, in terms of finding the optimal solution and CPU time consumption. From the results, it can be concluded that SCO is competitive with both DE and ABC algorithm on this test suite.

Besides DE and ABC, other algorithms are also worthy to be mentioned and compared. We do this indirectly via existing comparative studies on the same test suite, e.g., [14, 17, 21]. In [14] the means and standard deviations of the final function values are compared. If the average of a method is close to the real minimum and the standard deviation is small, then the method is regarded well-performing. Among the four methods that were compared, DE was found to be the best-performing algorithm, finding the optimal solution in all cases. In contrast, the standard PSO could only converge to the minimum value exactly for 3 functions. Also, arPSO solved 6 of them, while SEA solved 8 functions. Therefore, from the perspective of accuracy of the final solutions, SCO performs better than PSO, arPSO, since SCO can solve all 23 problems.

In [21] various advanced evolution strategies are compared. Again, only means and standard deviations are compared. The study finds that canonical ES fails on 12 functions, and both ESLAT and CMA-ES fail on 10 functions, while FEP fails on 9 functions. Also, the performance of ABC has also been compared with these methods in [17], where ABC is superior. Consequently, in this aspect, SCO also has better performance than these methods.

Our work compared the performance of SCO with that of DE and ABC on a large suite of test functions. In addition to comparing whether an algorithm can find the optimal solution reliably, we also compared the speed of the algorithm. From the numerical results we conclude that SCO is competitive with DE and ABC algorithm in terms of accuracy and speed, at least for this test suite. When comparing with other methods through some comparative studies indirectly, the performance of SCO is better than that of the methods mentioned above. SCO is simple, robust, and converges fast to the true global minimum. Besides, it has only a few control parameters which are easy to set and tune.

Acknowledgement

We thank the referees for their helpful comments. This work was supported by the Australian Research Council Centre of Excellence for Mathematical and Statistical Frontiers, under grant number CE140100049. Qibin Duan would also like to acknowledge the support from the University of Queensland through the UQ International Scholarships scheme.

References

[1] Rechenberg I. Evolution Strategy: Optimization of Technical Systems by Means of Biological Evolution. Stuttgart: Fromman-Holzboog; 1973.

[2] Fogel LJ, Owens AJ, Walsh MJ. Artificial Intelligence through Simulated Evolution. New York: John Wiley & Sons; 1966.

[3] Holland JH. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial intelligence. U Michigan Press; 1975.

[4] Rubinstein RY, Kroese DP. The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning. Berlin: Springer; 2004.

[5] Storn R, Price K. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 1997;11(4):341–59.

[6] Kennedy J, Eberhart R. Particle swarm optimization. In: Proceedings. IEEE International Conference on Neural Networks; vol. 4. 1995, p. 1942–1948 vol.4.

[7] Dorigo M, Stützle T. Ant Colony Optimization. Cambridge, MA, USA: MIT Press; 2004.

[8] Yao X, Liu Y, Lin G. Evolutionary programming made faster. IEEE T Evolut Comput 1999;3(2):82–102.

[9] Karaboga D, Basturk B. A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. J Glob Optim 2007;39(3):459–71.

[10] Kahn H, Harris TE. Estimation of particle transmission by random sampling. National Bureau of Standards applied mathematics series 1951;12:27–30.

[11] Botev ZI, Kroese DP. Efficient Monte Carlo simulation via the generalized splitting method. Stat Comput 2012;22(1):1–16.

[12] Kroese DP, Taimre T, Botev ZI. Handbook of Monte Carlo Methods. New York: John Wiley & Sons; 2011.

[13] Botev ZI. The Generalized Splitting Method for Combinatorial Counting and Static Rare-Event Probability Estimation. Ph.D. thesis; The Unverisity of Queensland; 2009.

[14] Vesterstrom J, Thomsen R. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In: IEEE Congress on Evolutionary Computation (CEC2004); vol. 2. Piscataway, New Jersey; 2004, p. 1980–7.

[15] Price K, Storn RM, Lampinen JA. Differential Evolution: A Practical Approach to Global Optimization. Berlin: Springer; 2006.

[16] Karaboga D. An idea based on honey bee swarm for numerical optimization. Tech. Rep.; Technical Report-TR06, Erciyes University, Engineering Faculty, Computer Engineering Department; 2005.

[17] Karaboga D, Akay B. A comparative study of artificial bee colony algorithm. Appl Math Comput 2009;214(1):108–32.

[18] Karaboga D, Basturk B. On the performance of artificial bee colony (ABC) algorithm. Appl Soft Comput 2008;8(1):687–97.

[19] Gao W, Liu S. A modified artificial bee colony algorithm. Comput Oper Res 2012;39(3):687–97.

[20] Karaboga D, Gorkemli B. A quick artificial bee colony (qABC) algorithm and its performance on optimization problems. Appl Soft Comput 2014;23:227–38.

[21] Hedar A, Fukushima M. Evolution strategies learned with automatic termination criteria. In: SCIS & ISIS 2006. Tokyo, Japan; 2006, p. 1126–34.