Comparative Study of Population-Based Metaheuristic Methods in Global Optimization

Master's Thesis

Teemu Peltonen

May 25, 2015

University of Jyväskylä
Department of Physics

Supervisors: Prof. Dr. Ferrante Neri
and Dr. Pekka Koskinen

Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's?

Alan Turing

Abstract

While global optimization problems are very common when working in computational nanoscience, they also constitute a set of computationally very demanding problems that lack efficient all-round optimizers. In this thesis I do a small survey on algorithms that try to be exactly like this; general methods that can be applied to any problem off the shelf. Constraining myself to so-called population-based metaheuristics, especially Nature-inspired evolutionary algorithms and swarm intelligence, I study their capabilities of solving a difficult real-world optimization problem, the Lennard-Jones cluster structure problem. I also use one of the algorithms, CCPSO2, to study the Lennard-Jones problem's separability, that is, how well it can be solved by dividing the problem into smaller subproblems.

An analysis on the Lennard-Jones problem of up to 200 atoms shows that these methods can be surprisingly efficient optimizers even in high-dimensional real-world problems. Especially one algorithm, namely CCPSO2, provides quite good and robust approximations for the global minima of the Lennard-Jones clusters with an error of around 10–20 %, and all that with a very small number of heavy energy function evaluations. I also show that they are superior to two versions of simulated annealing algorithms, a popular class of global optimization methods among physicists.

By using CCPSO2 to divide the Lennard-Jones problem into smaller pieces of different dimensionalities, I show that the most efficient way to solve it is to divide the problem into subproblems of only 4–12 dimensions, regardless of the problem dimension. This is a remarkable result since this division dramatically decreases the difficulty of the Lennard-Jones problem and it serves as a guide for people trying to develop even more efficient algorithms for this problem.

Tiivistelmä

Vaikka globaalit optimointiongelmat ovat hyvin yleisiä laskennallisen nanotieteen alalla, ne ovat myös laskennallisesti erittäin vaativia ongelmia, joille tehokkaita ja yleisiä ratkaisualgoritmeja ei ole saatavilla. Tässä työssä teen yleiskatsauksen algoritmeihin, joiden tavoitteena on olla juuri tällaisia yleisiä menetelmiä, joita voi soveltaa tehokkaasti mihin ongelmaan tahansa. Rajoittuessani niin kutsuttuihin populaatiopohjaisiin metaheuristiikkoihin, erityisesti luonnosta ideansa saaneisiin evolutiivisiin algoritmeihin ja parviälyyn, tutkin niiden kyvykkyyttä ratkaista eräs vaikea todellisen maailman ongelma, Lennard-Jones-atomiryppään rakenneongelma. Käytän lisäksi yhtä algoritmeista, CCPSO2:ta, Lennard-Jones ongelman separoituvuusanalyysiin eli siihen, kuinka hyvin kyseisen ongelman voi ratkaista jakamalla se pienempiin osaongelmiin.

Lennard-Jones-ongelman tutkiminen 200 atomiin asti osoittaa, että kyseiset menetelmät voivat olla hämmästyttävän tehokkaita optimoijia jopa korkeadimensioisissa todellisen maailman ongelmissa. Erityisesti yksi menetelmistä, CCPSO2, kykenee arvioimaan Lennard-Jones-atomiryppäiden globaaleja minimejä 10–20 % virheellä dimensiosta riippumatta, ja vieläpä todella vähällä määrällä raskaita energiafunktiokutsuja. Näytän lisäksi, että nämä menetelmät ovat ylivoimaisia verrattuna kahteen simuloidun jäähdytyksen versioon. Simuloitu jäähdytys on fyysikoiden keskuudessa suosittu luokka globaalin optimoinnin ratkaisualgoritmeja.

Käyttämällä CCPSO2:ta jakamaan Lennard-Jones-ongelma pienempiin, dimensioiltaan erikokoisiin osaongelmiin, näytän että tehokkain tapa ratkaista kyseinen ongelma on jakaa se erittäin pieniin 4–12 dimension osaongelmiin riippumatta ongelman dimensiosta. Tämä on merkittävä tulos, sillä jako näin pieniin ongelmiin helpottaa vaikean Lennard-Jones-ongelman ratkaisua merkittävästi, ja se toimii suuntaviivana kyseiselle ongelmalle vieläkin tehokkaampia algoritmeja kehittäville tutkijoille.

Acknowledgements

Here I wish to thank all those people who have helped me both in putting together this thesis and surviving the long studies I have done.

First of all, I would like to thank both my supervisors, Prof. Dr. Ferrante Neri and Dr. Pekka Koskinen. Thanks to Ferrante Neri for introducing me to a whole new field and for giving a different perspective for this interdisciplinary study. Thanks to Pekka Koskinen for being a fantastic supervisor in all my larger works, namely bachelor's thesis, research training, and this master's thesis. You are a supervisor who has always time to give valuable opinions and to help solving numerous problems, as well as to provide comments on a thesis in less than a day. Your guide for seminar presentations also helped me a lot in giving a good and understandable seminar.

I would also wish to thank my lovely partner Anna for supporting me when having difficulties in my studies and with this thesis, and for understanding all those weekends and evenings I spent with this thesis instead of you.

And last but certainly not least, I must thank my dear friends and people at my second home, FYS4, for both spending evenings doing the homework exercises with me and for giving valuable help in uncountably many problems I encountered doing this thesis. I also very much liked discussing more or less lightweight subjects with you, with everything from politics to the essence of black holes and worm holes, in the middle of stressing out with this thesis.

Contents

1 Introduction

What is the shortest route between two cities? What is the best way to manufacture cars to maximize income? What are the optimal weights of an artificial neural network? What is the minimum energy structure of an atomic cluster? Optimization problems are everywhere. You encounter them almost in daily basis both in everyday life and as a scientist, if not consciously, then at least unconsciously. Even though approximate solutions to some of these problems might be obvious to a person having a good intuition or education, solving them mathematically might give even better solutions. However, often when increasing the number of possible routes, number of perceptrons in a neural network, or number of atoms, the problems might become so difficult that mathematics and computer algorithms are the only viable method for providing even the crudest approximate solutions. Nevertheless, no efficient standard methods exist and the study of new global optimization algorithms is very active.

In this thesis I take a small journey to the field of computer science, trying to introduce methods that might be standard for computer scientists but not known to physicists, to the field of computational nanoscience. The standard method for a physicist to tackle a new global optimization problem is usually to use all available knowledge of the problem to create a (partially) new *heuristic* algorithm, a so-called problem-tailored algorithm, that exploits the problem peculiarities. This method often produces very good results but it has the obvious drawback that your new sophisticated algorithm will only work for the single problem, and then for the next one you have to start from scratch. This, of course, also takes a lot of time. In this work I take a different approach and introduce a few *metaheuristic* methods, inspired by Darwinian evolution and swarm intelligence in animals, that try to be general all-round optimizers that can be used without much tuning. These methods have shown to be efficient optimizers in many benchmark problems [1, 2, 3, 4] as well as in many real-world problems [5, 6, 7, 8, 9, 10, 11, 12]. I show that in the case of a Lennard-Jones potential, the difficult atomic cluster structure problem is no exception and that these methods offer an attractive and quick alternative to the problem-tailored methods.

On the other hand, one of the methods, namely CCPSO2, also provides a way to study the separability of a given problem, *i.e.* how well it can be solved by dividing it into smaller subproblems. If this division is possible, the difficulty of high-dimensional problems can be dramatically decreased. As an example, I show that the Lennard-Jones cluster problem is best solved by dividing it into very small subproblems of only 4–12 dimensions, giving a hint that if developing new algorithms for this problem, one should stick to methods perturbing only a small subset of the variables at a time.

The thesis is structured as follows. In Section 2 I define the general optimization problem and give a few specific physical examples, including Lennard-Jones cluster problem, which can be modeled as optimization problems. In Section 3 I introduce a few specific metaheuristics that serve as a basis for the more advanced methods, jDE and CCPSO2, presented in Section 4. Then in Section 5.1 I show that these Nature-inspired metaheuristics can be very efficient optimizers even in the difficult Lennard-Jones cluster problem also in high dimensions. Finally in Section 5.2 I show a way to use CCPSO2 to analyze the separability of the Lennard-Jones problem to get ideas for further algorithmic development.

Finally, a word about the mathematical notation. Throughout this thesis, vectors in $\mathbb{R}^n$ are denoted by boldface-font. Furthermore, as the amount of different indices grows rather high from time to time, unless otherwise stated I use a convention where the main loop (usually the generation) is indexed by an upper index, while the vector component or individual of the population *etc.* is indexed by a lower index. As an example, at the generation G, the i'th individual could be denoted $\mathbf{x}_i^G$, while its j'th component would be $x_{i,j}^G$. It should always be clear from the context whether the upper index means an exponent or a loop index.

2 The problem of global optimization in physics

2.1 The general optimization problem

Let us first define what we mean by global and local minima and maxima of a general function with $\mathbb{R}$ as the codomain, equipped with the Euclidean metric.

Definition 1 (Minima and maxima)**.** Let D be a nonempty set and $f : D \to \mathbb{R}$. The point $x^* \in D$ is said to be a

(i) *global minimum point* of f if $f(x^*) \leq f(x)$ for all $x \in D$ and
(ii) *global maximum point* of f if $f(x^*) \geq f(x)$ for all $x \in D$.

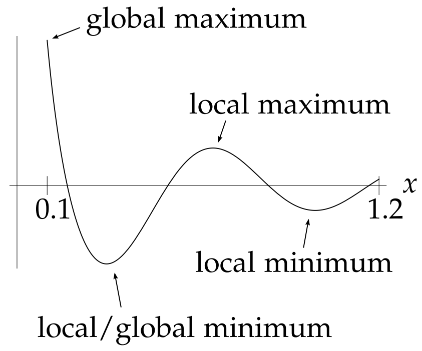
Moreover, if D is a topological space, the point $x^* \in D$ is said to be a

(iii) *local minimum point* of f if $f(x^*) \leq f(x)$ for all $x \in U \cap D$ for some neighbourhood U of x and
(iv) *local maximum point* of f if $f(x^*) \geq f(x)$ for all $x \in U \cap D$ for some neighbourhood U of x.

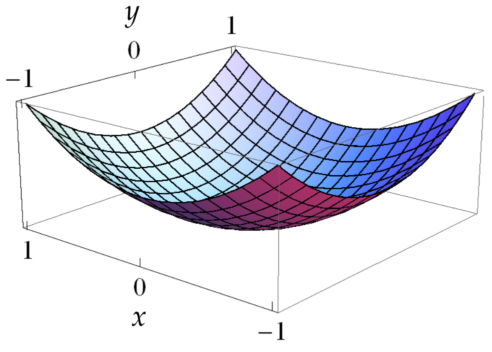
In these cases the point x^* is said to be an *extremum point* while the corresponding value $f(x^*)$ is known as an *extremum*. If the inequality is strict for all $x \neq x^*$, the corresponding extremum is *strict*.

Examples of different kind of minima and maxima are shown in Fig. 1.

By the term *optimization problem* we mean finding the (local or global) minimum or maximum points of a given *objective function* f. Of course, depending on the function, these might not be unique even if they exist. It depends on the



(a)



(b)

Figure 1. Examples of local and global minima when the space is equipped with the Euclidean metric. **(a)** The function $f : [0.1, 1.2] \to \mathbb{R}$, $f(x) = \cos(3\pi x)/x \ \ \forall x \in [0.1, 1.2]$ has two local minima, a local maximum, a global minimum, and a global maximum. **(b)** The function $f :]-1, 1[^2 \to \mathbb{R}$, $f(x, y) = x^2 + y^2 \ \ \forall (x, y) \in]-1, 1[^2$ has one and only one local minimum at the origin, which is also a global minimum.

particular problem at hand whether one wants to find only one of them, all of them, or if one is only studying their existence. In the case of minimization (maximization), if f has only a single minimum (maximum) it is called *unimodal* and in the case where it has more than one minima (maxima) it is called *multimodal*. Moreover, the domain D in the above definition is said to be the *search space*, and if it is a vector space, its dimension is called the *problem dimension*.

Although the definition above was very general, usually for practical solving methods the search space D needs to be a subset of $\mathbb{R}^n$. In fact, in this thesis I will mostly consider methods that assume D to be a *rectangular* subset of $\mathbb{R}^n$.

Definition 2 (Rectangular set)**.** A set $D \subset \mathbb{R}^n$ is called *rectangular* if for each $i \in \{1, \dots, n\}$ there are $a_i, b_i \in \mathbb{R}$ such that $a_i < b_i$ and

$$D = \prod_{i=1}^{n} [a_i, b_i]. \tag{1}$$

In other words, this means that each coordinate x_i has lower and upper bounds as

$$a_i \leq x_i \leq b_i. \tag{2}$$

A common optimization problem in real life is minimizing the cost of some process, yielding the term *cost function* for f in the case of minimization. Respectively, in the case of maximization the objective function f is called the *fitness function*. However, since the minimization and maximization problems are very closely related, these terms are often used interchangeably. So if someone tells you "the fitness is better", he means the function value is higher in the case of maximization and that the function value is lower in the case of minimization, respectively.

Let us define a few more general terms regarding optimization.

Definition 3. Let $x \in D \subset \mathbb{R}^n$ and $f : D \to \mathbb{R}$ be differentiable. The point x is said to be a *critical point* of f if $\nabla f(x) = 0$. Furthermore, x is said to be a *saddle point* of f if it is a critical point but is neither a minimum nor a maximum.

Two examples of saddle points are shown in Fig. 2.

Finding the local extrema is easy in principle by using the following theorem, which gives us a necessary condition for a point to be a local extremum [13].

Theorem 1. *Let $f : D \to \mathbb{R}$ be a differentiable function on the open set $D \subset \mathbb{R}^n$. If $x \in D$ is a local extremum point of f then x is a critical point,* i.e. $\nabla f(x) = 0$.

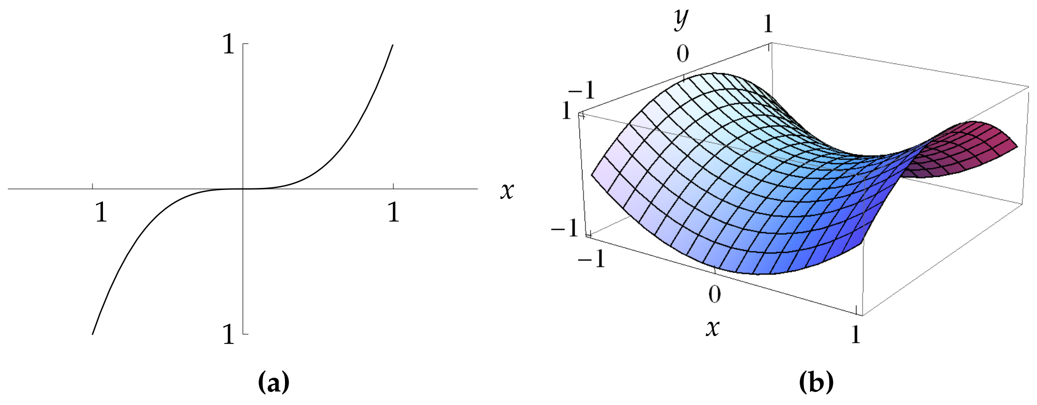


Figure 2. Two examples of saddle points. Both the functions **(a)** $f : \mathbb{R} \to \mathbb{R},\ f(x) = x^3\ \ \forall x \in \mathbb{R}$ and **(b)** $f : \mathbb{R}^2 \to \mathbb{R},\ f(x, y) = x^2 - y^2\ \ \forall (x, y) \in \mathbb{R}^2$ have one and only one saddle point, which is located at the origin.

With the help of Theorem 1, one could find all the local extrema of a given *differentiable* function f by first calculating its gradient and solving $\nabla f(x) = 0$ for x. This yields all the critical points which can be either local minima, maxima, or saddle points, so the quality of these critical points have to be examined one at a time. If one wanted to study the qualities, one should solve the eigenvalues of the so-called Hessian matrix [13].

Although in principle the above procedure for finding the local extrema of a given function f seems straightforward, there might be many problems in using it in real life optimization problems. First of all f might not be differentiable at all, let alone speaking of its continuous second partial derivatives that are needed in the Hessian matrix. Even worse, in the case of optimization of a procedure *etc.*, one may not even know the analytical expression of f. Also, the amount of critical points can be astronomically large as we will see in the next section in the Lennard-Jones cluster problem. However, what the term *local optimization* usually means in numerical physics is finding the nearest local optimum from a given initial point.

Now moving on to global optimization, a similar route can be followed. It is clear that the global extremum points can be found only from the critical points, the domain boundary, or from the points where f is not differentiable. The process sounds easy since one must only compare the function values at each of these points and pick the ones that give the highest or lowest values, but this is usually not the case. Again the same problems arise in real life as in the case of local optimization. Moreover, calculating the objective function value itself might be very time-consuming. Methods for optimizing this kind of difficult "black box" functions, whose analytical expressions need not be known, are presented in Section 3.

One important factor determining the difficulty of an optimization problem is its dimension since the volume of the search space increases exponentially: a cube with a side of length L has a volume of L^n in an n-dimensional space. This phenomenon is called *the curse of dimensionality* [14, 15, 16].

As can be seen from the results in the papers describing the algorithms CCGA [17] and CCPSO2 [18], a very efficient way to tackle *some* high-dimensional problems is to subdivide them into smaller pieces. The important property of a problem that tells how well it can be solved this way is called separability.

Definition 4 (Separable function)**.** Let $D = \prod_{i=1}^{n} D_i \subset \mathbb{R}^n$. The function $f : D \to \mathbb{R}$ is called *separable* if there are functions $\{f_i : D_i \to \mathbb{R}\}_{i=1}^{n}$ such that

$$f(x_1, \dots, x_n) = \sum_{i=1}^{n} f_i(x_i) \quad \forall (x_1, \dots, x_n) \in D. \tag{3}$$

It is obvious that if a function can be written as a sum of functions of only one variable as above, its optimization becomes way easier since instead of solving one n-dimensional problem it is enough to solve n one-dimensional problems. In some sense the effective dimensionality of the problem has decreased from L^n to nL, which is a dramatic change. A simple example of a separable function would be

$$f : \mathbb{R}^3 \to \mathbb{R}, \quad f(x_1, x_2, x_3) = x_1^2 + 2x_2 - \mathrm{e}^{-x_3} \quad \forall (x_1, x_2, x_3) \in \mathbb{R}^3. \tag{4}$$

But how about functions that cannot be written as a sum this way only because a few of the variables interact? For example, the function

$$\begin{aligned} &f : \mathbb{R}^5 \to \mathbb{R}, \\ &f(\mathbf{x}) = f(x_1, x_2, x_3, x_4, x_5) = x_1 x_2 + 2x_3 + \mathrm{e}^{x_4} - x_5^2 \quad \forall \mathbf{x} \in \mathbb{R}^5 \end{aligned} \tag{5}$$

is not separable because of the interaction term $x_1 x_2$, but still algorithms that optimize each variable separately should be quite effective. These kind of functions are called *partially separable*.

2.2 Structure optimization and the Lennard-Jones cluster problem

Let us now take a physical example of a global minimization problem: structure optimization. We have N atoms and a potential $U : \mathbb{R}^{3N} \to \mathbb{R}$ describing the

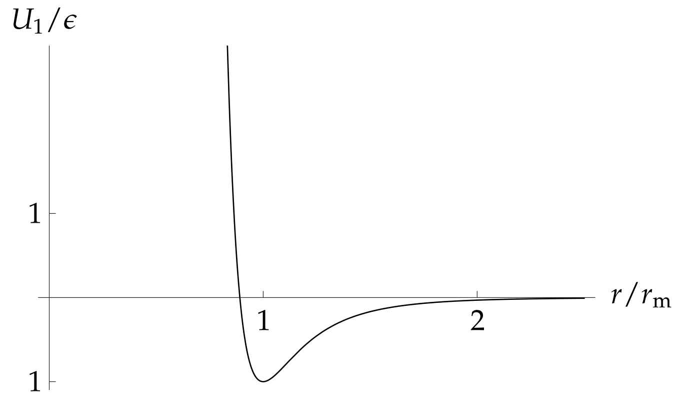


Figure 3. The graph of the Lennard-Jones potential function U_1 for a single pair of atoms separated by a distance r.

total potential energy between all these particles. Again, the explicit form of U might be unknown or very complex, which is the case of a quantum-mechanical potential. In the structure optimization problem one is interested in finding the coordinates $(x_i, y_i, z_i) \in \mathbb{R}^3$ for all the atoms $i \in \{1, \ldots, N\}$ such that the potential U gets minimized, because this is also the most stable structure in Nature.

In this thesis I will concentrate on a simple and easy-to-calculate *Lennard-Jones* potential because the main focus will be on studying the efficiency of algorithms instead of new physical phenomena. The Lennard-Jones potential is a so-called *pair-wise* potential where the potential is calculated by summing up all the potentials between each pair of atoms. The Lennard-Jones potential for a single pair of atoms separated by the Euclidean distance r is the function

$$U_1 : \mathbb{R}_+ \to \mathbb{R}, \quad U_1(r) = \epsilon \left[\left(\frac{r_\mathrm{m}}{r} \right)^{12} - 2 \left(\frac{r_\mathrm{m}}{r} \right)^{6} \right] \quad \forall r \in \mathbb{R}_+, \tag{6}$$

where $\epsilon \in \mathbb{R}_+$ is the well depth and $r_\mathrm{m} \in \mathbb{R}_+$ is the minimum point. A figure showing the behaviour of the function (6) is shown in Fig. 3, which shows a single global and local minimum point and a steep potential rise when the atoms are getting closer together.

In order to work with numbers in the order of unity, it is convenient to choose the energy and length units as $\epsilon = r_\mathrm{m} = 1$ in numerical calculations. The potential (6) then becomes

$$U_1 : \mathbb{R}_+ \to \mathbb{R}, \quad U_1(r) = \frac{1}{r^{12}} - \frac{2}{r^6} \quad \forall r \in \mathbb{R}_+, \tag{7}$$

or as a function from the coordinate-space of atoms i and j,

$$U_{ij} : G \to \mathbb{R}, \quad U_{ij}(\mathbf{r}_i, \mathbf{r}_j) = \frac{1}{||\mathbf{r}_i - \mathbf{r}_j||^{12}} - \frac{2}{||\mathbf{r}_i - \mathbf{r}_j||^{6}} \tag{8}$$

for all $(\mathbf{r}_i, \mathbf{r}_j) = (x_i, y_i, z_i, x_j, y_j, z_j) \in G := \{(\mathbf{r}, \mathbf{r}') \in \mathbb{R}^6 : \mathbf{r} \neq \mathbf{r}'\} \subset \mathbb{R}^6$, where

$$||\cdot|| : \mathbb{R}^3 \to \mathbb{R}, \quad ||(x,y,z)|| = \sqrt{x^2 + y^2 + z^2} \quad \forall (x,y,z) \in \mathbb{R}^3 \tag{9}$$

is the Euclidean norm. With the help of the pair potential (8) the total Lennard-Jones potential describing the whole system of N atoms is then written as a sum over all pairs of atoms, excluding the double counting, as

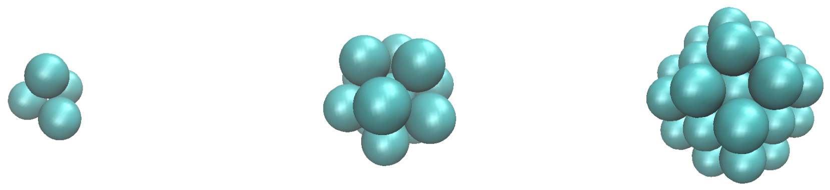
$$\begin{aligned} &\tilde{U} : \tilde{D} \to \mathbb{R}, \\ &\tilde{U}(\mathbf{r}_1, \dots, \mathbf{r}_N) = \sum_{j=1}^{N} \sum_{\substack{i=1 \\ (i<j)}}^{N} U_{ij}(\mathbf{r}_i, \mathbf{r}_j) \\ &\qquad = \sum_{j=1}^{N} \sum_{\substack{i=1 \\ (i<j)}}^{N} \left(\frac{1}{||\mathbf{r}_i - \mathbf{r}_j||^{12}} - \frac{2}{||\mathbf{r}_i - \mathbf{r}_j||^{6}} \right) \end{aligned} \tag{10}$$

for all $(\mathbf{r}_1, \dots, \mathbf{r}_N) \in \tilde{D} := \{(\mathbf{r}_1, \dots, \mathbf{r}_N) \in \mathbb{R}^{3N} : \mathbf{r}_i \neq \mathbf{r}_j \ \forall i \neq j\} \subset \mathbb{R}^{3N}$.

However, as mentioned previously, I will mostly consider methods that assume the search space to be rectangular, which $\tilde{D}$ is not because it is not bounded and it has the extra constraint $\mathbf{r}_i \neq \mathbf{r}_j$. Firstly, the constraint can simply be dropped out since it is highly unlikely that any algorithm utilizing random number generators, which I will only consider, will come up generating two exactly same vectors. Secondly, for the coordinate-wise bounds I used the following solution. To keep the rectangular search space small for small clusters and to increase its size for larger clusters I define the search space as

$$D := [0,4]^3 \times \prod_{k=4}^{3N} \left[-4 - \frac{1}{4} \left\lfloor \frac{k-3}{3} \right\rfloor, \ 4 + \frac{1}{4} \left\lfloor \frac{k-3}{3} \right\rfloor \right]. \tag{11}$$

So whenever I refer to “the Lennard-Jones problem”, I mean finding the mini-



(a) $N = 4$ **(b)** $N = 13$ **(c)** $N = 38$

Figure 4. Currently best known minimum configurations of Lennard-Jones clusters of 4, 13, and 38 atoms.

mum of

$$U : D \to \mathbb{R},$$
$$U(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \sum_{j=1}^{N} \sum_{\substack{i=1 \\ (i<j)}}^{N} \left(\frac{1}{||\mathbf{r}_i - \mathbf{r}_j||^{12}} - \frac{2}{||\mathbf{r}_i - \mathbf{r}_j||^{6}} \right). \tag{12}$$

For visualization, currently the best known [19] minimum configurations for 4, 13, and 38 atoms are shown in Figure 4.

Although the Lennard-Jones potential is not a very realistic model for describing general atomic clusters, in the case of noble gas atoms it works quite well [20, 21] by modeling the dipole–dipole interactions properly. The main reason for choosing it for this cluster problem, however, is its computational conveniency and the number of previous studies regarding its optimization [22, 23, 24, 25, 26, 27]. A comprehensive database for comparison is thus readily available.

Even though the potential (7) is trivial to minimize, the case is very different when increasing the number of atoms. It has been suggested, based on empirical evidence, that the number of local minima increases with the number of atoms N as $O(e^{N^2})$ [28], which renders the problem a really hard one. Even if one could by any mean solve the equation $\nabla U(\mathbf{r}_1, \ldots, \mathbf{r}_N) = 0$, the amount of solutions (critical points) is so large that only calculating the potential function values for these points would take an astronomical amount of time. For example, if $N = 100$, the order of the amount of local minima* is about 10^{136} [28]. When comparing this to the number of atoms in the observable universe, 10^{80}, it is clear that only trying out all the possible critical points is not a viable method

*. Note that there might be a huge amount of local maxima and saddle points as well.

for solving this global minimization problem.

2.3 Examples of other physical optimization problems

The cluster problem above was an example of a "traditional" physical optimization problem in the sense that optimization is the traditional tool used to solve the problem. Let us next take an example of another traditional physical/chemical optimization problem, the problem of finding the minimum energy path of a chemical reaction.

2.3.1 Minimum energy path of a chemical transition

If one wanted to gain deeper understanding of a chemical reaction, knowing the initial and the final molecules and their configurations, it would be beneficial to know the *minimum energy path* connecting these initial and final states, that is, the path it usually follows in Nature. As in the Lennard-Jones cluster problem, let $G \subset \mathbb{R}^{3N}$ be the coordinate-space of N atoms, and denote the initial configuration by $A \in G$ and the final configuration by $B \in G$. For example, A could be the situation where the two molecules are separated, and B could be the situation where the two molecules have formed a new single molecule. The space of all the possible paths between the initial and final states, the search space, now becomes

$$\tilde{\Gamma} := \{\gamma : [0,1] \to G : \ \gamma \text{ is continuous}, \ \gamma(0) = A, \ \gamma(1) = B\}. \qquad (13)$$

Then what is the actual minimum energy path of all the paths in $\tilde{\Gamma}$? Here we need to fix the potential describing the system, $V : G \to \mathbb{R}$, that tells the potential energy of a given configuration. In the very simple case it could be, for instance, the Lennard-Jones potential in Eq. (12). We can now think of the graph of V, $\mathcal{G}_V \subset \mathbb{R}^{3N} \times \mathbb{R}$, as a "landscape" (the *potential energy surface*) where at a given position (configuration of all the atoms) we are at a certain height (the potential energy). Following this analogy, it is clear that water running down a hill will most likely follow a path that goes at the bottom of a *valley*. It is easy to convince oneself that a chemical reaction will similarly most likely follow "a valley bottom path", with the obvious difference caused by inertia of mass in water. For visualization, three example paths that follow a valley bottom on a simple artificial low-dimensional potential energy surface are shown in Fig. 5.

Obviously, the valley bottom path is not unique and we thus need more constraints. Making the well-justified assumption that the initial and final states A and B are local minimum points, the valley bottom path *must* go through at

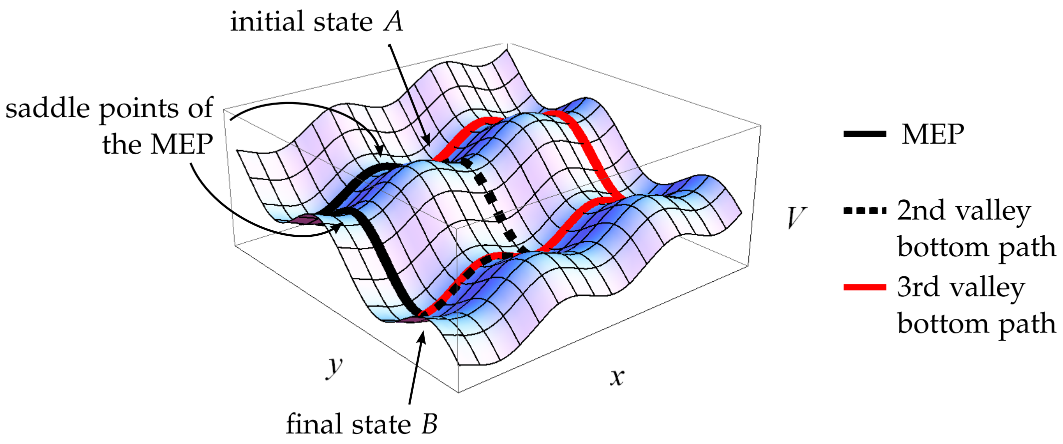


Figure 5. Three example paths from an initial configuration A to a final configuration B that follow a valley bottom on a simple artificial potential energy surface. Every path has saddle points, but out of the highest saddle points, the minimum energy path (MEP) has the lowest one.

least one saddle point. The highest of these saddle points is called the *transition state*. As is well known from chemistry, the reaction with the lowest energy transition state is most likely to occur, meaning that this is the path we are looking for. As a summary, the minimum energy path can be defined by the following two requirements:

1. The path follows the bottom of a valley.
2. Out of these valley bottom paths, the minimum energy path (MEP) is the one whose highest saddle point has the lowest energy.

In Fig. 5, out of the three example valley bottom paths, the MEP is shown as a black, continuous path.

The two requirements above are not enough to model the problem as a global optimization problem, so let us make approximations next. Looking at the definition of MEP, finding it feels more or less equivalent to minimizing the path integral of V along all the possible paths. This, of course, does not exactly provide the correct MEP, but it should provide good results for well-behaved potentials, as is discussed in Ref. [29]. Mathematically, finding the MEP has (approximately) reduced down to minimizing the path integral function

$$\begin{aligned} &\tilde{f} : \tilde{\Gamma} \to \mathbb{R}, \\ &\tilde{f}(\gamma) = \int_\gamma V \mathrm{d}s = \int_0^1 V(\gamma(t)) ||\gamma'(t)|| \, \mathrm{d}t \quad \forall \gamma \in \tilde{\Gamma}, \end{aligned} \tag{14}$$

which still needs to be discretized.

The first easy refinement is to consider only piecewise linear paths. Defining a (dense) partition $P := \{t_0 = 0, t_1, \ldots, t_{K-1}, t_K = 1\}$ of $[0,1]$, the search space can be written as

$$\begin{aligned}\Gamma := \{\gamma : [0,1] \to G :\ & \gamma \text{ is continuous, } \gamma(0) = A,\ \gamma(1) = B, \\ & \gamma \text{ is piecewise linear with the partition P}\}. \end{aligned} \tag{15}$$

But because of continuity

$$\begin{aligned}\tilde{f}(\gamma) = \int_0^1 V(\gamma(t))||\gamma'(t)||\,\mathrm{d}t &\approx \sum_{j=1}^{K} V(\gamma(t_j))||\gamma'(t_j)||(t_j - t_{j-1}) \\ &\approx \sum_{j=1}^{K} V(\gamma(t_j))||\gamma(t_j) - \gamma(t_{j-1})|| \quad \forall \gamma \in \Gamma, \end{aligned} \tag{16}$$

we may define the function to be minimized as

$$\begin{aligned} & f : \Gamma \to \mathbb{R}, \\ & f(\gamma) = \sum_{j=1}^{K} V(\gamma(t_j))||\gamma(t_j) - \gamma(t_{j-1})|| \quad \forall \gamma \in \Gamma, \end{aligned} \tag{17}$$

where $t_j \in P$ for all j. In order to solve this optimization problem by a computer, however, one yet needs to identify the path γ as a finite set of points $\{(t_0, \gamma(t_0)), \ldots, (t_K, \gamma(t_K))\}$. Note that at this point I have only shown that it is possible to model the approximate MEP problem as an optimization problem, but I will not discuss how to solve it in practice. For practical algorithms the objective function in Eq. (17) might need some serious further refinements.

Let us next take an example of yet another physical optimization problem, the problem of finding the ground state of a 1-dimensional quantum-mechanical system.

2.3.2 Ground state of a quantum-mechanical system

Another interesting physical problem is finding the ground state of a quantum-mechanical system. If we restrict ourselves to study the stationary states of a single particle in one dimension, the problem is to solve the Schrödinger equation

$$\left(-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right)\psi(x) = E\psi(x), \tag{18}$$

where $\hbar$ is the reduced Planck's constant, m is the mass of the particle, V is the potential describing the system, E is the (unknown) energy of the system, and ψ is the (unknown) wavefunction of the system. Solving this differential equation yields all the possible states, but if we are only interested in finding the lowest-energy state (the ground state), the problem can also be formulated as a global minimization problem.

Assume now that $V : [a, b] \to \mathbb{R}$ is the given arbitrary (smooth) potential and that the system is in an infinite potential well, *i.e.* $V(a) = 0 = V(b)$. Because we are interested in stationary states it is enough to study real-valued wavefunctions in the space L^2. However, due to numerical convenience, we restrict ourselves to the search space

$$D := \{\psi : [a, b] \to \mathbb{R} : \psi(a) = 0 = \psi(b),\ \exists \psi'',\ \int_a^b \psi(x)^2 \,\mathrm{d}x = 1\}. \tag{19}$$

The objective function can be found by using a fact from the variational method of quantum mechanics: If E_0 is the ground state energy of a given system, then

$$E_0 \leq \langle \psi | H\psi \rangle \tag{20}$$

for any normalized state ψ, where $H = -\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V$ is the Hamiltonian and $\langle \cdot | \cdot \rangle$ is the L^2 inner product. By defining the function

$$\begin{aligned} &E : D \to \mathbb{R}, \\ &E(\psi) = \langle \psi | H\psi \rangle = \int_a^b \psi(x) \left(-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x) \right) \psi(x) \,\mathrm{d}x \\ &\qquad\quad = \int_a^b \left(-\frac{\hbar^2}{2m}\psi(x)\psi''(x) + V(x)\psi(x)^2 \right) \mathrm{d}x \end{aligned} \tag{21}$$

for all $\psi \in D$, the problem of finding the ground state of a one-dimensional single particle QM-system has reduced down to minimizing the function E.

Again, I have shown that a specific problem can be formulated as an optimization problem. But the above form for the objective function is of little practical help, especially when solving it numerically, because the search space is a rather abstract function space. It is now up to the researcher how he or she wants to proceed and how to search the search space in practice. One possible algorithm-dependent approach is presented by Grigorenko *et al.* in Ref. [30], where they used a specific genetic algorithm that combined Gaussian-like functions in a nonlinear way to form quite arbitrary functions in D.

In the next two sections I will describe population-based, metaheuristic algorithms that are designed to solve general optimization problems. I have splitted the discussion in two parts: in Section 3 I discuss basic methods that are good in explaining the foundational ideas behind the state-of-the-art algorithms, presented in Section 4, that are nowadays the algorithms actually in use.

3 Basic population-based, metaheuristic algorithms for global optimization

In the previous section we saw a few physical problems that can be modeled as global optimization problems. In this section I will present selected algorithms that try to solve these problems. As the real-life physical problems are often very difficult and high-dimensional, *stochastic* methods [31] are usually the tool of choice. These methods are based in generating new possible solutions by random number generators. A few of the most popular stochastic methods among physicists are called *simulated annealing* [32, 33, 23] and *basin hopping* [34, 35, 36]. The first one tries to simulate annealing and cooling of a physical object: we start from a random solution, we have a high probability of accepting worse random solutions in the beginning, but during the run we gradually decrease the acceptance probability (by decreasing the temperature) and hope that in the end the solution has converged into the global optimum. The latter one, on the other hand, simply does hopping between adjacent local optimum points, where the local optima are found by some local optimizer.

Although the methods above are outside the scope of this thesis, a generic version of simulated annealing is introduced in Algorithm 1 because I will use two versions of it as a comparison later in Section 5.1. The details of the actual versions I used in the comparison study are also presented in Section 5.1, where I present the unspecified parts, $\mathbf{r}^k$ and A, of the pseudocode.

In this thesis I will mainly concentrate in global optimization methods that have a very different approach: Nature-inspired *evolutionary algorithms* (EA) and algorithms from the field of *swarm intelligence*. While EA's apply the ideas of Darwinian evolution of species to global optimization by using the familiar mechanisms reproduction, mutation, recombination, and selection, swarm intelligence models collective behaviour of individual agents working together towards a common goal, *e.g.* a bird flock searching for food. These algorithms belong to a class of *population-based metaheuristics* that have a simple generic outline:

1. Initialize a random population of individuals
2. Apply the search (*e.g.* mutation and crossover) operators
3. Select the best individuals to the next generation
4. Repeat steps 2–3 until a good enough individual is present in the population.

In contrast to simulated annealing and basin hopping where we had only a single solution that was perturbed, there is now a *population* of *individuals*, which is why they are called population-based methods. The reason to prefer

Algorithm 1: A generic pseudocode of different versions of simulated annealing (SA). Here, $\mathrm{rand}(0,1)$ is a uniform random number from the interval $[0,1]$. The pseudocode has two components that must be specified by a specific implementation: $\mathbf{r}^k$ is a random vector of some kind, while A is the rule for adaptively updating θ^k. The termination criterion is usually a lower bound for the temperature, but it can also be *e.g.* a maximum number of fitness function evaluations.

```
Data: Rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, fitness function
      $f : D \to \mathbb{R}$, initial temperature $T^0$, temperature reduction parameter $\chi$,
      initial step size $\theta^0$, number of steps $l^k$ in each temperature
Generate a random solution $\mathbf{x}^0 \in D$;
Initialize $k = 0$;
while termination criterion is not met do
    for $l = 0, \ldots, l^k$ do
        Generate a new neighbour $\mathbf{u}^k = \mathbf{x}^k + \theta^k \mathbf{r}^k$;
        Move to the new neighbour with the rule
        $\mathbf{x}^{k+1} = \begin{cases} \mathbf{u}^k & \text{if } f(\mathbf{u}^k) \le f(\mathbf{x}^k) \text{ or } \mathrm{e}^{(f(\mathbf{x}^k) - f(\mathbf{u}^k))/T^k} > \mathrm{rand}(0,1) \\ \mathbf{x}^k & \text{otherwise} \end{cases}$;
    Adaptively update the step size: $\theta^{k+1} = A(\theta^k)$;
    Reduce the temperature: $T^{k+1} = \chi T^k$;
    $k$++;
```

population-based methods over single-solution methods in this thesis is that there are multiple arguments on why they usually perform better, as argued by Prügel-Bennett [37].

On the other hand, the reason to prefer *metaheuristics* over *heuristics* is simple. While heuristics, problem-tailored approximate methods that use problem-dependent "rule-of-thumbs", are often used because of their power in tackling difficult problems, I focus on metaheuristics. They are problem-independent methods that are based on more abstract, higher-level rules [38]. They do not need any information about the gradient of the fitness function or even the explicit formula of it. The fitness function is thus only treated as a "black box", an object that just provides a value for a given point. This is advantageous if the fitness function is very complex, noisy, not differentiable, or if it is a procedure or a simulation that yields the fitness value. They also have the obvious advantage that a metaheuristic algorithm is not restricted to a single problem only, but can be used for any problem instead.

In the following I will discuss three basic population-based metaheuristics. The first, *genetic algorithm* (GA), is a class of algorithms that can be considered as a father of all the evolutionary algorithms, as it most closely models Darwinian evolution by using bit-string representations as a model of the chromosomes representing individuals. The second, *differential evolution* (DE), is similar to GA's but uses vectors of real numbers instead of the bit-strings to represent individuals. The third, *particle swarm optimization* (PSO), is an example of swarm intelligence, even though it has very similar components in it.

The methods presented here are all minimization algorithms. However, it is clear that the maximization of f is equivalent to the minimization of $-f$, so the restriction to minimization problems can be done without loss of generality.

3.1 Genetic algorithms

In 1950, while tackling with the problem of artificial intelligence, Alan Turing proposed [39] a "learning machine" that would parallel the principles of Darwinian evolution to simulate the learning happening in a child's brain. The idea was put to practice by Barricelli in 1954 [40, 41] when he wrote the first successful computer experiments regarding artificial evolution. However, it was not until 1975 when these methods were applied to function optimization by John Holland [42, 43], yielding the so-called *genetic algorithms*. The genetic algorithms can be viewed as the father of all the evolutionary algorithms, as they are all somehow derived from the GA approach.

The genetic algorithms follow the basic metaheuristic outline with some very identifiable peculiarities. Assuming a population size N_{pop} and a rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, the population at the G'th generation is

$$P^G := \left\{\mathbf{x}_i^G\right\}_{i=1}^{N_{\text{pop}}} \tag{22}$$

where $\mathbf{x}_i^G \in D$ for all i. The important point is that the individuals $\mathbf{x}_i^G$ are usually *bit-string representations* of the corresponding vectors. For example, if we had a 4-bit system, for a vector

$$\tilde{\mathbf{x}} := (3_{10}, 11_{10}, 5_{10}) = (0011_2, 1011_2, 0101_2) \in \mathbb{R}^3 \tag{23}$$

we could have a representation

$$\mathbf{x} := (001110110101) \tag{24}$$

where the components are just glued together to form a bit-string. To keep the notation easier, we identify the vectors $\tilde{\mathbf{x}}$ and $\mathbf{x}$ and use the bit-string representation for the genetic operators and the vector representation for the fitness calculations.

The idea of the genetic algorithm starts to become apparent now: the bit-string in (24) is a simple model of a chromosome, representing an individual, and the chromosomes then form the population.

3.1.1 Crossover

Following the analogy to Darwinian evolution, first k *parents* $Q^G := \{\mathbf{q}_i^G\}_{i=1}^k$ are selected from the population P^G either randomly or by some fitness-based rule, and the parents are then mated to form an offspring $C^G := \{\mathbf{c}_i^G\}_{i=1}^l$. As in Darwinian evolution, this information-exchanging process is called *crossover*. A schematic picture of the process is shown in Fig. 6, where two parent chromosomes are crossed over to form two children chromosomes. There are several possibilities of how this can be done in genetic algorithms, but one simple and common example is the *sexual* (two parents at a time form two children) version of the *one-point* crossover. Here, we cut the two parents from a random place into two parts and exchange their latter parts. For example, the parents

$$\begin{aligned} \mathbf{q}_1^G &= (0011101|10101) \quad \text{and} \\ \mathbf{q}_2^G &= (1001110|00100) \end{aligned}$$

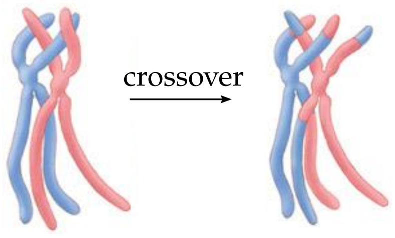


Figure 6. Crossover of two chromosomes in Darwinian evolution.

would produce the children

$$\mathbf{c}_1^G = (0011101|00100) \quad \text{and}$$
$$\mathbf{c}_2^G = (1001110|10101)$$

if the random cut-place happens to be in the position denoted by $|$. This procedure is then repeated for as many parent-pairs as necessary to form an offspring of l children.

3.1.2 Mutation

Again as in evolution, the resulting children are *mutated* to form the mutated children $M^G := \{\mathbf{m}_i^G\}_{i=1}^l$. A simple method is to pick each child $\mathbf{c}_i^G$ and perform bit-flipping in each bit with a probability p, as defined by the user. For example, if $p = 0.2$, on average 2.4 bits get flipped and thus the mutated version of the child $\mathbf{c}_1^G$ above could be

$$\mathbf{m}_1^G = (00\bar{1}11\bar{0}10010\bar{0}) = (000111100101).$$

3.1.3 Selection

The final very important phase in evolution is *survival of the fittest*. Similarly in GA, the population P^{G+1} of the next generation is formed somehow from the mutated children M^G and their parents Q^G. There exists a whole range of different schemes, including *fitness-proportionate selection* and *greedy selection* as two common examples. When comparing between two individuals, in fitness-proportionate selection the probability of being selected to the next generation is proportionate to fitness, while in greedy selection the better individual gets selected with a probability of 1.

3.1.4 Summary of the algorithm and further notes

A pseudocode of the genetic algorithm described above is shown in Algorithm 2. The description here is actually called a *generational genetic algorithm* [44]. There is also a simpler version, called a *steady-state genetic algorithm* [44], where no generations are present. Instead, usually only two parents are selected from the population and they are crossed-over to form two children. The children are then mutated and the resulting individuals replace their parents if they are better. This way the population is updated on-the-fly and two individuals at a time, yielding a more suitable version for parallel implementations. A careful reader can see that, in fact, the steady-state version is only a special case of the generational one.

Algorithm 2: A pseudocode of a generic genetic algorithm (GA). The termination criterion is usually a maximum number of fitness function evaluations or a maximum number of generations.

```
Data: Rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, fitness function
      $f : D \to \mathbb{R}$, population size $N_{\text{pop}}$, mutation probability $p$
Initialize a random population $P^0 = \{\mathbf{x}_i^0\}_{i=1}^{N_{\text{pop}}}$: for each individual $i$, generate a
random position $\mathbf{x}_i^0 \in D$;
Initialize the generation $G = 0$;
while termination criterion is not met do
    Select $k$ parents $Q^G = \{\mathbf{q}_i^G\}_{i=1}^{k} \subset P^G$;
    Create offspring $C^G = \{\mathbf{c}_i^G\}_{i=1}^{l}$ by crossing over the parents in $Q^G$;
    Mutate the children $C^G$ to form $M^G$;
    Select individuals staying alive: form $P^{G+1}$ out of $M^G$ and $Q^G$ by selecting
    "the best" individuals;
    $G$++;
return the best known individual
```

It is important to note that in the above form genetic algorithms can be used only in problems where the search space is a rectangular subset of R^n. But if one is willing to take this restriction, then the same algorithm can be used for every problem having a search space a rectangular subset of R^n. However, if one is willing to develop a different variant for each problem, genetic algorithms can be used also in more abstract search spaces such as the function space in Eq. (19). In Ref. [30] a specific genetic algorithm has been developed for this particular problem. The individuals are originally Gaussian-like functions, but by going through specifically engineered mutation and crossover operators they form more and more arbitrary functions.

3.2 Differential evolution

First introduced by Storn and Price in 1997 [1], *differential evolution* (DE) is yet another example of an evolutionary algorithm. Comparing to genetic algorithms, the solutions are not represented by bit-strings but by vectors of real numbers instead. Having a rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$ and a population size N_{pop}, the population at the G'th generation is

$$P^G := \left\{\mathbf{x}_i^G\right\}_{i=1}^{N_{\text{pop}}} \tag{25}$$

where $\mathbf{x}_i^G \in D$ for all i.

3.2.1 Mutation

The way DE examines the landscape of the search space is encoded in the mutation operator, which adds *weighted differences* of random individuals to other individuals, yielding the term differential evolution. To be exact, for each *target vector* $\mathbf{x}_i^G \in P^G$, a *mutant vector*

$$\tilde{\mathbf{v}}_i^G := \mathbf{x}_{r_1}^G + F(\mathbf{x}_{r_2}^G - \mathbf{x}_{r_3}^G) \tag{26}$$

is formed. Here $F \in [0, 2]$ is the *mutation strength parameter* and $r_1, r_2, r_3 \in \{1, \ldots, N_{\text{pop}}\} \setminus \{i\}$ are *mutually distinct* random integers that are generated again for each i. This also means that the population size must be at least four.

3.2.2 Saturation

The resulting mutant vector $\tilde{\mathbf{v}}_i^G$ might very well reside outside the rectangular search space D. To transform it back to the feasible region D, each of its components j are transformed to D by a user-defined function as

$$v_{i,j}^G := \text{saturate}_{a_i,b_i}(\tilde{v}_{i,j}^G), \tag{27}$$

if the j'th coordinate is bounded by the lower and upper bounds a_i and b_i, respectively. The original version, in fact, does not have the saturation component at all. But since most of the following algorithms have it, and to keep the algorithms comparable, I included it also in this algorithm.

3.2.3 Crossover

To further increase the diversity of the population, crossover is applied after the mutation process. Here a new parameter has to be introduced, the *crossover* *rate* $C \in [0,1]$, which tells the probability to pick a chosen component from the mutant vector. Using the mutant vector $\mathbf{v}_i^G = (v_{i,1}^G, \ldots, v_{i,n}^G)$, the *trial vector*

$$\mathbf{u}_i^G = (u_{i,1}^G, \ldots, u_{i,n}^G) \tag{28}$$

is formed, where the component $j \in \{1, \ldots, n\}$ is

$$u_{i,j}^G := \begin{cases} v_{i,j}^G & \text{if } s_j \leq C \text{ or } j = t_i \\ x_{i,j}^G & \text{otherwise} \end{cases}. \tag{29}$$

In other words, each component is checked independently for update, in contrast to the one-point crossover presented in Section 3.1.1 in regard of genetic algorithms. Here $s_j \in [0,1]$ is the j'th evaluation of a uniform random number generator and $t_i \in \{1, \ldots, n\}$ is a uniformly chosen random index to ensure that the new trial vector gets at least a single component from the mutant vector.

3.2.4 Selection

Again to implement the idea of survival of the fittest, differential evolution uses the greedy criterion to decide which individual survives to the next generation. The individual for the next generation is chosen as

$$\mathbf{x}_i^{G+1} = \begin{cases} \mathbf{u}_i^G & \text{if } f(\mathbf{u}_i^G) < f(\mathbf{x}_i^G) \\ \mathbf{x}_i^G & \text{otherwise} \end{cases}. \tag{30}$$

In other words, the previous individual is replaced by the new individual if and only if the new individual is better.

3.2.5 Summary of the algorithm and further notes

In summary, a pseudocode of differential evolution is shown in Algorithm 3. It is easy to see, heuristically, why and how differential evolution works [45]. The concept of mutating the individuals with vector differences automatically makes the mutation a self-adjusting phenomenon. Initially the vector differences remain large in the initial population, so the algorithm searches widely the search space for promising areas. Gradually the individuals start to get settled into local minima, so the vector differences start to contain information about the landscape of the search space. The scale of the differences then slowly decays out while more and more individuals start to get near the global minimum.

Algorithm 3: A pseudocode of the original differential evolution (DE) algorithm. The termination criterion is usually a maximum number of generations or fitness function evaluations.

```
Data: Rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, fitness function
      $f : D \to \mathbb{R}$, population size $N_{\text{pop}}$, mutation strength parameter $F \in [0, 2]$,
      crossover rate $C \in [0, 1]$
Initialize a random population $P^0 = \{\mathbf{x}_i^0\}_{i=1}^{N_{\text{pop}}}$: generate random individual
$\mathbf{x}_i^0 \in D$ for each $i$, calculate their fitness values, and save the best individual;
Initialize the generation $G = 0$;
while termination criterion is not met do
    for each individual (target vector) do
        Mutation: form the mutant vector $\tilde{\mathbf{v}}_i^G$ using Eq. (26);
        Saturation: form $\mathbf{v}_i^G$ by transforming the mutant vector $\tilde{\mathbf{v}}_i^G$ back to the
        feasible area $D$ using Eq. (27);
        Crossover: form the trial vector $\mathbf{u}_i^G$ using Eqs. (28) and (29);
        Calculate the fitness $f(\mathbf{u}_i^G)$ of the new trial vector, select it to the next
        generation according to Eq. (30) and update the best known individual;
    $G$++;
return the best known individual
```

The obvious advantage of differential evolution is its simplicity: it is very easy to implement and it has only three control parameters. Despite of its simplicity it seems to be a very robust and consistent optimizer [46].

3.2.6 Common variants

Looking at the definition of the mutation operator, it is easy to invent new variants of differential evolution by adding small modifications to Eq. (26). Indeed, a few variants of the basic DE have shown to be useful. In order to keep track of the different variants, Storn and Price introduced [1] the notation

$$\mathrm{DE}/x/y/z, \tag{31}$$

where

- DE stands for differential evolution
- x specifies the vector to be mutated, and it is usually either "rand" (random vector from the population) or "best" (the best vector from the population)
- y defines the number of difference vectors to be used in the mutation

- z specifies the crossover scheme, which is usually either "bin" (binomial) or "exp" (exponential).

The name "binomial" for the crossover operator comes from the fact that the number of new components in the mutant vector approximately follows the binomial distribution. Another common solution is to use an operator where the number of new components follows the exponential distribution.

Using this notation the standard differential evolution scheme may be written as DE/rand/1/bin. Other common variants listed in Ref. [47] include

$$\begin{aligned}
\text{DE/best/1/}z\text{:}\quad & \tilde{\mathbf{v}}_i^{G+1} = \mathbf{x}_{\text{best}}^G + F(\mathbf{x}_{r_2}^G - \mathbf{x}_{r_3}^G) \\
\text{DE/cur-to-best/1/}z\text{:}\quad & \tilde{\mathbf{v}}_i^{G+1} = \mathbf{x}_i^G + F(\mathbf{x}_{\text{best}}^G - \mathbf{x}_i^G) + F(\mathbf{x}_{r_2}^G - \mathbf{x}_{r_3}^G) \\
\text{DE/best/2/}z\text{:}\quad & \tilde{\mathbf{v}}_i^{G+1} = \mathbf{x}_{\text{best}}^G + F(\mathbf{x}_{r_2}^G - \mathbf{x}_{r_3}^G) + F(\mathbf{x}_{r_4}^G - \mathbf{x}_{r_5}^G) \\
\text{DE/rand/2/}z\text{:}\quad & \tilde{\mathbf{v}}_i^{G+1} = \mathbf{x}_{r_1}^G + F(\mathbf{x}_{r_2}^G - \mathbf{x}_{r_3}^G) + F(\mathbf{x}_{r_4}^G - \mathbf{x}_{r_5}^G).
\end{aligned}$$

3.3 Particle swarm optimization

Darwinian evolution is not the only possible phenomenon to learn from Nature. The idea for *particle swarm optimization* [4] (PSO) can be seen to be picked up from the behaviour of bird flocks: in order to find food, the birds move in large flocks and the information of promising areas is exchanged between the individuals. This is why the algorithm belongs to a class of *swarm intelligence*, even though it is very similar to the other evolutionary algorithms described above. In PSO the individuals moving in the search space are called *particles* and each particle's movement is influenced by its own best known position as well as by the best globally known position.

A simple overview of particle swarm optimization can be found from the article by van der Bergh and Engelbrecht [48]. As in evolutionary algorithms, having a rectangular search space $D = \prod_{i=1}^n [a_i, b_i] \subset \mathbb{R}^n$, the population at the G'th generation consists of N_{pop} individuals (now particles) as

$$P^G := \left\{\mathbf{p}_i^G\right\}_{i=1}^{N_{\text{pop}}} = \left\{(\mathbf{x}_i^G, \mathbf{v}_i^G, \mathbf{y}_i^G)\right\}_{i=1}^{N_{\text{pop}}}. \tag{32}$$

Again as in DE we use real-vector representations of the particles. The difference is that in addition to the particle's current position $\mathbf{x}_i^G \in D$ the particle also stores its current *velocity* $\mathbf{v}_i^G \in \mathbb{R}^n$ as well as its personal best known position $\mathbf{y}_i^G \in D$. Moreover, the best position $\hat{\mathbf{y}}^G$ found so far by any individual is stored by the population.

3.3.1 Position update

In contrast to genetic algorithms, differential evolution, and most evolutionary algorithms, PSO has only one update rule or "genetic operator". At the generation G, for each particle $\mathbf{p}_i^G \in P^G$ its velocity is updated as

$$\mathbf{v}_i^{G+1} = \mathbf{v}_i^G + cr_i(\mathbf{y}_i^G - \mathbf{x}_i^G) + ds_i(\hat{\mathbf{y}}^G - \mathbf{x}_i^G), \tag{33}$$

where $c, d \in \mathbb{R}$ are the *acceleration coefficients* that determine how much the particle will move towards its personal best known position and the best globally known position, respectively. Furthermore, $r_i, s_i \in [0, 1[$ are random numbers from two uniform random number generators and they are generated again for each i.

The position of the particle is then updated straightforwardly by

$$\mathbf{x}_i^{G+1} = \mathbf{x}_i^G + \mathbf{v}_i^{G+1}. \tag{34}$$

Here it can be seen that even though $\mathbf{v}_i^G$ is called the velocity, it is not the real physical velocity but it only describes how much to move and in which direction instead. The original PSO does not have the saturation component so the updated particles might wander off the feasible area D. However, if D is large enough and because the initial population is inside of it, this should not become a problem.

3.3.2 Updating the personal and global best vectors

After the position has been updated, the best personal position is updated greedily by

$$\mathbf{y}_i^{G+1} = \begin{cases} \mathbf{x}_i^{G+1} & \text{if } f(\mathbf{x}_i^{G+1}) < f(\mathbf{y}_i^G) \\ \mathbf{y}_i^G & \text{otherwise} \end{cases} \tag{35}$$

and the best globally known position also greedily by

$$\hat{\mathbf{y}}^{G+1} = \begin{cases} \mathbf{x}_i^{G+1} & \text{if } f(\mathbf{x}_i^{G+1}) < f(\hat{\mathbf{y}}^G) \\ \hat{\mathbf{y}}^G & \text{otherwise} \end{cases}. \tag{36}$$

3.3.3 Summary of the algorithm and further notes

To further illustrate particle swarm optimization, a pseudocode of the algorithm is shown in Algorithm 4. In summary, even though the original idea of the PSO algorithm was very different from differential evolution, the update rule is very similar to the mutation operator in DE and its variants. This again is one good reason why the different algorithms really belong to the same family, even though they have different terminology because of historical reasons.

Algorithm 4: A pseudocode of particle swarm optimization (PSO). The termination criterion is usually a maximum number of fitness function evaluations or a maximum number of generations.

```
Data: Rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, fitness function
      $f : D \to \mathbb{R}$, population size $N_{\text{pop}}$, acceleration coefficients $c, d \in \mathbb{R}$
Initialize a random population $P^0 = \{\mathbf{p}_i^0\}_{i=1}^{N_{\text{pop}}} = \{(\mathbf{x}_i^0, \mathbf{v}_i^0, \mathbf{y}_i^0)\}_{i=1}^{N_{\text{pop}}}$: generate
random positions $\mathbf{x}_i^0 \in D$ and velocities $\mathbf{v}_i^0$ for each particle $i$, calculate the
fitness values, and update the personal $\mathbf{y}_i^0$ and global best $\hat{\mathbf{y}}^0$ positions;
Initialize the generation $G = 0$;
while termination criterion is not met do
    for each particle $\mathbf{p}_i^G = (\mathbf{x}_i^G, \mathbf{v}_i^G, \mathbf{y}_i^G)$ do
        Calculate the new velocity $\mathbf{v}_i^{G+1}$ using Eq. (33);
        Calculate the new position $\mathbf{x}_i^{G+1}$ using Eq. (34);
        Calculate the fitness $f(\mathbf{x}_i^{G+1})$ of the new position and update the
        personal best $\mathbf{y}_i^{G+1}$ and global best positions $\hat{\mathbf{y}}^{G+1}$ using Eqs. (35) and
        (36);
    $G$++;
return the best globally known position $\hat{\mathbf{y}}^G$
```

As differential evolution, a good side of particle swarm optimization is its simplicity and its ease to be implemented. Furthermore, it usually seems to converge very fast, but unfortunately often only to a local minimum [46].

4 State-of-the-art metaheuristics

The few population-based metaheuristics described in the previous section have been modified and developed a lot further by many people. In this section I will describe two state-of-the-art algorithms that provide faster and more reliable convergence than the basic versions in many optimization problems. The first algorithm, jDE, is a very simple modification of differential evolution that adaptively chooses the important F and C parameters, leaving the user with only one free parameter. The second algorithm, CCPSO2, is a little more complex version of particle swarm optimization that tries to be efficient also in high dimensions by dividing the problem into smaller subproblems.

Why did I choose exactly these two methods to explain here? In Section 5.1 I will show that these methods, jDE and CCPSO2, are actually very efficient all-round optimizers even in the difficult Lennard-Jones problem. On the other hand, in Section 5.2, I will use CCPSO2 to study the separability properties of the Lennard-Jones problem to show that it is possible to solve the difficult, nonseparable problem by dividing it into very small pieces.

The reason why the algorithms are discussed so thoroughly is that the original articles describing them contain a few errors and lack certain details. So I try to provide here a detailed and mathematically exact explanation of both the algorithms, so detailed that it should be possible to implement them after a careful reading of this section.

4.1 jDE

A very simple but efficient *self-adaptive* algorithm, based on differential evolution, was introduced by J. Brest *et al.* in 2006 [49]. This algorithm was later named jDE [50] probably after the first author. jDE is very similar to the original DE, having only a small modification. While in differential evolution there were the constant user-defined mutation parameter F and the crossover rate C, in jDE each individual keeps track of its own F and C parameters and they are adaptively updated. To be more precise, having a rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, the population at the generation G is

$$P^G := \left\{\mathbf{z}_i^G\right\}_{i=1}^{N_\text{pop}} = \left\{\left(\mathbf{x}_i^G, F_i^G, C_i^G\right)\right\}_{i=1}^{N_\text{pop}} \tag{37}$$

where $\mathbf{x}_i^G \in D$ is the point in the search space and F_i^G and C_i^G are the DE's F and C parameters, now evolving through generations and being unique to each individual.

4.1.1 Temporary mutation parameter and crossover rate formation

Before the mutation and crossover operators for the i'th individual, temporary mutation parameter $F_i^{G,\text{tmp}}$ and crossover rate $C_i^{G,\text{tmp}}$ are formed as†

$$F_i^{G,\text{tmp}} = \begin{cases} F_\text{l} + r_1 F_\text{u} & \text{if } r_2 < \tau_1 \\ F_i^G & \text{otherwise} \end{cases} \tag{38}$$

and

$$C_i^{G,\text{tmp}} = \begin{cases} r_3 & \text{if } r_4 < \tau_2 \\ C_i^G & \text{otherwise} \end{cases}, \tag{39}$$

where $r_1, r_2, r_3, r_4 \in [0, 1]$ are random numbers from a uniform random number generator that are generated again for each i. Here also the new "parameters" $F_\text{l} = 0.1$, $F_\text{u} = 0.9$, and $\tau_1 = 0.1 = \tau_2$ were introduced, but, according to the authors, their choice does not significantly alter the behaviour of the algorithm.

These newly formed temporary parameters are then used in the standard DE mutation, saturation, and crossover operations to form $\mathbf{u}_i^G$ from the vectors $\mathbf{x}$. It has to be pointed out that only the position components $\mathbf{x}$ are used in these genetic operators instead of the full individuals $\mathbf{z}$. The adaptivity becomes part of the algorithm only in the slightly modified selection operator, which is described next.

4.1.2 Selection

The decision whether the new trial vector survives to the next generation is done greedily as in DE [Eq. (30)], but now also the individual-dependent parameters are updated according to

$$\begin{aligned} \mathbf{z}_i^{G+1} &= (\mathbf{x}_i^{G+1}, F_i^{G+1}, C_i^{G+1}) \\ &= \begin{cases} (\mathbf{u}_i^G, F_i^{G,\text{tmp}}, C_i^{G,\text{tmp}}) & \text{if } f(\mathbf{u}_i^G) < f(\mathbf{x}_i^G) \\ (\mathbf{x}_i^G, F_i^G, C_i^G) & \text{otherwise} \end{cases}. \end{aligned} \tag{40}$$

This means that the new perturbed parameters survive to the next generation if and only if they formed a better individual. In this way only the good parameter values survive, providing the adaptivity to the algorithm.

†. There is a flaw in the logic explaining this part in the original article, so the correct version I present here looks a little different.

4.1.3 Summary of the algorithm and further notes

As a summary, a pseudocode of the jDE algorithm is presented in Algorithm 5. It must be noted that the term *self-adaptive* the authors used is a little questionable in describing the algorithm. According to their definition [49] self-adaptivity would mean that the usual DE parameters F and C are encoded in the chromosomes and are also going through the evolution operators. As can be seen from the explanation above, the extra parameters are indeed encoded in the chromosomes and they go through the selection operator, but they *do not* go through the mutation and crossover operators.

Algorithm 5: A pseudocode of the jDE algorithm. The termination criterion is usually a maximum number of fitness function evaluations or a maximum number of generations.

```
Data: Rectangular search space $D = \prod_{i=1}^{n}[a_i, b_i] \subset \mathbb{R}^n$, fitness function
      $f : D \to \mathbb{R}$, population size $N_{\text{pop}}$
Initialize a random population $P^0 = \{\mathbf{z}_i^0\}_{i=1}^{N_{\text{pop}}} = \{(\mathbf{x}_i^0, F_i^0, C_i^0)\}_{i=1}^{N_{\text{pop}}}$: for each
individual $i$, generate a random position $\mathbf{x}_i^0 \in D$, a random mutation coefficient
$F_i^0 \in [0.1, 1]$, and a random crossover rate $C_i^0 \in [0, 1]$, calculate the fitness value
$f(\mathbf{x}_i^0)$, and update the best known position;
Initialize the generation $G = 0$;
while termination criterion is not met do
    for each individual $\mathbf{z}_i^G = (\mathbf{x}_i^G, F_i^G, C_i^G)$ do
        Generate $F_i^{G,\text{tmp}}$ and $C_i^{G,\text{tmp}}$ according to Eqs. (38) and (39);
        Mutation: form the mutant vector $\tilde{\mathbf{v}}_i^G$ using $F_i^{G,\text{tmp}}$ as the mutation
        strength parameter in Eq. (26);
        Saturation: form $\mathbf{v}_i^G$ by transforming the mutant vector $\tilde{\mathbf{v}}_i^G$ back to the
        feasible area D using Eq. (27);
        Crossover: form the trial vector $\mathbf{u}_i^G$ using $C_i^{G,\text{tmp}}$ as the crossover rate in
        Eq. (29);
        Calculate the fitness $f(\mathbf{u}_i^G)$ of the new trial vector, update the individual
        $\mathbf{z}_i^{G+1}$ according to Eq. (40), and update the best known position;
    G++;
return the best known position
```

Even though jDE is a little more complex than the basic differential evolution, it is still a very simple algorithm. It has also the big advantage of having only one control parameter, the population size, making it very practical in actual optimization applications. There is thus only a minimal need for time-consuming parameter tuning. Moreover, jDE seems to be generally a very good optimizer in many numerical benchmark problems [50].

4.2 CCPSO2

In 2012 Li *et al.* presented [18] a new PSO-based algorithm that can handle very high-dimensional problems. This algorithm, called cooperatively coevolving particle swarms for large scale optimization (CCPSO2), has a little longer historical story than the simple jDE described above.

4.2.1 The historical way from CCGA to CCPSO2

The history of CCPSO2 started in 1994 from Potter's and De Jong's *cooperative coevolutionary genetic algorithm* (CCGA) [17], which presented the first idea of a divide-and-conquer strategy to tackle high-dimensional problems. The idea in this method was simply to optimize each component of the problem separately, thus assuming full separability of the fitness function. They first divided the n-dimensional search space into n one-dimensional subspaces by generating a subpopulation for each of these components. The standard genetic algorithm was then used to optimize each of these subproblems separately. However, this division created a new problem: how to measure the fitness of an individual in a subproblem? The fitness function needs an n-dimensional vector to calculate a value, while the individuals in the subproblems are real numbers. Potter and De Jong solved this problem by assembling this component along with the best members from the other subpopulations and calculating the fitness for the resulting n-dimensional vector. Mathematically, when calculating the fitness of the j'th individual $x_i^j \in \mathbb{R}$ in the i'th subpopulation, they actually calculated

$$f(x_1^{\text{best}}, \ldots, x_{i-1}^{\text{best}}, x_i^j, x_{i+1}^{\text{best}}, \ldots, x_n^{\text{best}}), \tag{41}$$

where x_1^{best} is the best individual so far in the first subpopulation, x_2^{best} is the best individual so far in the second subpopulation, and so on.

Potter and De Jong also tried a version where they chose either random or best members from the other subpopulations in the fitness calculation. As expected, both of these versions worked well in separable functions but performed poorly in nonseparable functions, since the division into n subproblems is simply not possible in the latter case.

In CCGA the divide-and-conquer strategy was applied to a simple genetic algorithm, but nothing prevents one applying it to different methods also, such as differential evolution or particle swarm optimization. In addition to first applying this idea to PSO, van den Bergh and Engelbrecht allowed for a more flexible division of the problem [48]. Instead of dividing the n-dimensional problem into n parts, the problem is divided into K parts, each having a dimension of $s = n/K$, as specified by the user. This way a higher group size s can be chosen for nonseparable problems and a lower group size s for separable problems.

Yang *et al.* developed this idea even further by making their framework, *multilevel cooperative coevolution* (MLCC) [51], to adaptively choose a suitable group size s for a given problem. The user only has to provide a set of possible group sizes, *e.g.* $S := \{2, 5, 10, 100, 250\}$ for a 500-dimensional problem. Between generations the algorithm then chooses a new group size s from the set S depending on its performance record such that better-performing group sizes have higher probability to be chosen.

4.2.2 Details of CCPSO2

CCPSO2 followed in the route of MLCC. The user has to provide a set of possible group sizes and the algorithm adapts the group size to the problem. This time, however, there are no performance records and the new group size is chosen *uniformly at random* if and only if the best fitness didn't improve during the last generation. In addition to this, the dimension indices are permuted at each generation to allow for more flexible grouping. Without the permutation the successive components are likely to be always at the same group. The subpopulations formed this way are called *swarms* in CCPSO2.

The basic idea of the algorithm can be understood well from the original article [18]. However, because the notation is a little sloppy, a few of the ideas are explained wrong, and many corners are cut, I will present a corrected version here, as reverse-engineered from the original source code that can be found from Ref. [52]. Because of this the presentation will look quite different in some parts. Furthermore, because I am writing everything explicitly out, the mathematical presentation becomes quite technical. So while reading the details below, I encourage the reader to simultaneously keep eye on the pseudocode of CCPSO2, presented in Algorithm 6, in order to keep track of the main outline of the algorithm.

If we have a rectangular search space $D = \prod_{i=1}^{n} [a_i, b_i] \subset \mathbb{R}^n$, a population size N_{pop}‡ and denote *row* vectors by boldface font, the population at the generation

‡. Note that in the original article N_{pop} is denoted by *swarmSize*. Because *swarmSize* can be

G is stored in the matrix

$$X^G := \begin{pmatrix} \mathbf{x}_1^G \\ \vdots \\ \mathbf{x}_{N_{\text{pop}}}^G \end{pmatrix} = \begin{pmatrix} x_{1,1}^G & \cdots & x_{1,n}^G \\ \vdots & \ddots & \vdots \\ x_{N_{\text{pop}},1}^G & \cdots & x_{N_{\text{pop}},n}^G \end{pmatrix}, \tag{42}$$

where each *row* $\mathbf{x}_i^G$ is a point from the search space D. Furthermore, the best personal positions are stored in the matrix

$$Y^G := \begin{pmatrix} \mathbf{y}_1^G \\ \vdots \\ \mathbf{y}_{N_{\text{pop}}}^G \end{pmatrix} = \begin{pmatrix} y_{1,1}^G & \cdots & y_{1,n}^G \\ \vdots & \ddots & \vdots \\ y_{N_{\text{pop}},1}^G & \cdots & y_{N_{\text{pop}},n}^G \end{pmatrix}, \tag{43}$$

where each *row* $\mathbf{y}_i^G \in D$ is the best personal position of the i'th particle. These two matrices and the best vector

$$\hat{\mathbf{y}}^G = (\hat{y}_1^G, \dots \hat{y}_n^G) \tag{44}$$

found so far by any particle are entities that are affected by the column-permutation, but not by the grouping process.

During the first generation the trivial permutation $\sigma^0 = (12\dots n)$ is used. In the end of each generation $G-1$, a new permutation of $(12\dots n)$, called σ^G, is formed randomly. This is then used at the next generation to permute the columns of X^G, Y^G, and $\hat{\mathbf{y}}^G$. To help with the notation, I define the function

$$\mathcal{P}_\sigma : \mathcal{M}_{m\times n} \to \mathcal{M}_{m\times n} \tag{45}$$

that reorders the columns of $m \times n$ matrices [in permutation (12...n)] into the permutation σ. I will also be using the inverse function of this,

$$\mathcal{P}_\sigma^{-1} : \mathcal{M}_{m\times n} \to \mathcal{M}_{m\times n}, \tag{46}$$

that reverses the permutation σ back to $(12\dots n)$.

4.2.3 Forming the new swarms

In the beginning of each generation, a new group size s is chosen randomly from the set S if the fitness of $\hat{\mathbf{y}}$ didn't improve during the previous generation. This

easily mixed up with the dimensionality of each swarm, I'm using the notation N_{pop} to denote the number of particles in each swarm.

means the number of swarms is now $K := n/s$. New swarms $\{P_i^G\}_{i=1}^K$ are then formed by denoting the columns $i, \ldots, i+s$ of the current permuted population $\mathcal{P}_{\sigma^G}(X^G)$ by P_i^G. Similarly, the personal bests $\{Q_i^G\}_{i=1}^K$ of the particles in these swarms are formed by denoting the columns $i, \ldots, i+s$ of $\mathcal{P}_{\sigma^G}(Y^G)$ by Q_i^G. The j'th particle (now s-dimensional) in the i'th swarm is denoted as $\mathbf{p}_{j,i}$ and its personal best as $\mathbf{q}_{j,i}$, so the particles in the swarms constitute the *permuted* population now as

$$\mathcal{P}_{\sigma^G}(X^G) = (P_1^G, \ldots, P_K^G) = \begin{pmatrix} \mathbf{p}_{1,1}^G & \cdots & \mathbf{p}_{1,K}^G \\ \vdots & \ddots & \vdots \\ \mathbf{p}_{N_{\text{pop}},1}^G & \cdots & \mathbf{p}_{N_{\text{pop}},K}^G \end{pmatrix} \tag{47}$$

and their personal bests constitute the *permuted* Y matrix as

$$\mathcal{P}_{\sigma^G}(Y^G) = (Q_1^G, \ldots, Q_K^G) = \begin{pmatrix} \mathbf{q}_{1,1}^G & \cdots & \mathbf{q}_{1,K}^G \\ \vdots & \ddots & \vdots \\ \mathbf{q}_{N_{\text{pop}},1}^G & \cdots & \mathbf{q}_{N_{\text{pop}},K}^G \end{pmatrix}. \tag{48}$$

Moreover, we choose the components $i, \ldots, i+s$ of the permuted global best vector $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^G)$ to be the *swarm best* vector $\mathbf{r}_i^{G,0}$, that is, the best vector of the i'th swarm. The meaning of the extra index 0 will be clarified later in the swarm best update process. In other words,

$$\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^G) = (\mathbf{r}_1^{G,0}, \ldots, \mathbf{r}_K^{G,0}). \tag{49}$$

Later on we will use the swarm bests to form the so-called *context vector* when calculating fitnesses of lower-dimensional vectors.

As an example, if the problem dimension is $n = 6$, population size is set to $N_{\text{pop}} = 2$, group size happens to be $s = 3$, and the permutation is $\sigma = (214365)$, the permuted population would look like (dropping the generation index G)

$$\mathcal{P}_\sigma(X) = \begin{pmatrix} \underbrace{\begin{matrix} x_{1,2} & x_{1,1} & x_{1,4} \\ x_{2,2} & x_{2,1} & x_{2,4} \end{matrix}}_{\text{swarm 1 } P_1} & \underbrace{\begin{matrix} x_{1,3} & x_{1,6} & x_{1,5} \\ x_{2,3} & x_{2,6} & x_{2,5} \end{matrix}}_{\text{swarm 2 } P_2} \end{pmatrix}, \tag{50}$$

the two swarms would be

$$P_1 = \begin{pmatrix} x_{1,2} & x_{1,1} & x_{1,4} \\ x_{2,2} & x_{2,1} & x_{2,4} \end{pmatrix} = \begin{pmatrix} \mathbf{p}_{1,1} \\ \mathbf{p}_{2,1} \end{pmatrix} \quad \text{and} \quad P_2 = \begin{pmatrix} x_{1,3} & x_{1,6} & x_{1,5} \\ x_{2,3} & x_{2,6} & x_{2,5} \end{pmatrix} = \begin{pmatrix} \mathbf{p}_{1,2} \\ \mathbf{p}_{2,2} \end{pmatrix} \tag{51}$$

and their personal bests

$$Q_1 = \begin{pmatrix} y_{1,2} & y_{1,1} & y_{1,4} \\ y_{2,2} & y_{2,1} & y_{2,4} \end{pmatrix} = \begin{pmatrix} \mathbf{q}_{1,1} \\ \mathbf{q}_{2,1} \end{pmatrix} \quad \text{and} \quad Q_2 = \begin{pmatrix} y_{1,3} & y_{1,6} & y_{1,5} \\ y_{2,3} & y_{2,6} & y_{2,5} \end{pmatrix} = \begin{pmatrix} \mathbf{q}_{1,2} \\ \mathbf{q}_{2,2} \end{pmatrix}. \tag{52}$$

Furthermore, the permuted global best vector is

$$\mathcal{P}_\sigma(\hat{\mathbf{y}}) = (\hat{y}_2, \hat{y}_1, \hat{y}_4, \hat{y}_3, \hat{y}_6, \hat{y}_5) \tag{53}$$

so the swarm bests would be picked from this as

$$\mathbf{r}_1^0 = (\hat{y}_2, \hat{y}_1, \hat{y}_4) \quad \text{and} \quad \mathbf{r}_2^0 = (\hat{y}_3, \hat{y}_6, \hat{y}_5). \tag{54}$$

4.2.4 Updating the personal and swarm best vectors

We first calculate the fitness of the particles in these new swarms because the index-permutation changes them, and update the personal best and the swarm best vectors accordingly. For each swarm P_i^G, first its j'th particle's personal best vector $\mathbf{q}_{j,i}$ is updated as

$$\mathbf{q}_{j,i}^{G+1} = \begin{cases} \mathbf{p}_{j,i}^G & \text{if } f(\mathcal{P}_{\sigma^G}^{-1}(\mathbf{b}(\mathbf{p}_{j,i}^G, \mathbf{u}^{G,i}, i))) < f(\mathcal{P}_{\sigma^G}^{-1}(\mathbf{b}(\mathbf{q}_{j,i}^G, \mathbf{u}^{G,i}, i))) \\ \mathbf{q}_{j,i}^G & \text{otherwise} \end{cases}, \tag{55}$$

its swarm best $\mathbf{r}_i$ is checked for update by

$$\mathbf{r}_i^{G,j} = \begin{cases} \mathbf{q}_{j,i}^{G+1} & \text{if } f(\mathcal{P}_{\sigma^G}^{-1}(\mathbf{b}(\mathbf{q}_{j,i}^{G+1}, \mathbf{u}^{G,i}, i))) < f(\mathcal{P}_{\sigma^G}^{-1}(\mathbf{b}(\mathbf{r}_i^{G,j-1}, \mathbf{u}^{G,i}, i))) \\ \mathbf{r}_i^{G,j-1} & \text{otherwise} \end{cases}, \tag{56}$$

and this is repeated for each particle. In other words, it is first checked whether the current particle is better than its personal best and then it is checked whether this new personal best is even better than the best particle in the whole swarm. Here the extra index j in the swarm best vector $\mathbf{r}_i^{G,j}$ just indexes the particles in the loop because it is updated by each particle, and this update process always changes the "context" vector $\mathbf{u}^{G,i}$ for the next round. Furthermore, remember that $\mathbf{q}_{j,i}^G$ is part of the matrix $\mathcal{P}_{\sigma^G}(Y^G)$, so the matrix $\mathcal{P}_{\sigma^G}(Y^{G+1})$ gets also formed here. Similarly, because the context vector $\mathbf{u}^{G,i}$, defined below, consists of the swarm bests, it gets also updated on-the-fly.

Here the fitness calculations become a little messy because of the permutations and the necessary context vector, but let us go through this one part at a time.

Assuming $n = sK$, the function

$$\mathbf{b} : \mathbb{R}^s \times \mathbb{R}^n \times \{1, \dots, K\} \to \mathbb{R}^n \tag{57}$$

helps in calculating the fitnesses of s-dimensional vectors in a provided context. If a context vector $\mathbf{u} = (\mathbf{u}_1, \dots, \mathbf{u}_K) \in \mathbb{R}^n$ is divided into K s-dimensional parts and we want to calculate the fitness of the vector $\mathbf{z} \in \mathbb{R}^s$ in the i'th swarm, it returns

$$\mathbf{b}(\mathbf{z}, \mathbf{u}, i) = (\mathbf{u}_1, \dots, \mathbf{u}_{i-1}, \mathbf{z}, \mathbf{u}_{i+1}, \dots, \mathbf{u}_K) \in \mathbb{R}^n, \tag{58}$$

thus providing a vector suitable for the fitness function. The idea is similar as in CCGA's expression (41). Now the context where we want to calculate the fitnesses is the *concatenation of the current swarm bests*. Now that the swarm best gets updated for each particle j, the context vectors get updated all the time, and thus the mathematical notation becomes a little difficult. Assume we are checking the conditions in Eqs. (55) and (56) for the j'th particle in the i'th swarm and that we go through the particles in the natural order. Now because only the components different from i matter, we may define the context vector at the generation G for the i'th swarm as

$$\mathbf{u}^{G,i} := (\mathbf{r}_1^{G,N_{\text{pop}}}, \dots, \mathbf{r}_{i-1}^{G,N_{\text{pop}}}, \mathbf{r}_i^{G,0}, \dots, \mathbf{r}_K^{G,0}), \tag{59}$$

and so

$$\mathbf{b}(\mathbf{z}, \mathbf{u}^{G,i}, i) = (\mathbf{r}_1^{G,N_{\text{pop}}}, \dots, \mathbf{r}_{i-1}^{G,N_{\text{pop}}}, \mathbf{z}, \mathbf{r}_{i+1}^{G,0}, \dots, \mathbf{r}_K^{G,0}). \tag{60}$$

Now when calculating the fitness of $\mathbf{z}$ in this context, what we really do is just assemble the vector $\mathbf{z}$ in side of the *current* swarm bests from the other swarms. The only reason why some of the swarm bests have the extra index N_{pop} is that they have already been updated by Eq. (56) and by all the particles, and thus the context vector is not anymore simply $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^G)$.[§]

In order to make the fitness calculation more sensible between the different generations and permutations, the vector returned by $\mathbf{b}$ is furthermore reversed back to the original order by the inverse permutation function $\mathcal{P}_{\sigma^G}^{-1}$.

To better understand the update equations (55) and (56), let us continue the

§. It would be an option not to update the context vector on-the-fly, but instead use the same permuted global best vector $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^G) = (\mathbf{r}_1^{G,0}, \dots, \mathbf{r}_K^{G,0})$ as the context vector for all the swarms. However, according to tests I made this deteriorates the performance of the algorithm. In fact the context vector was defined exactly like this in the original article but it was not implemented this way in the source code.

example above for clarification, where we had

$$\mathcal{P}_\sigma(\hat{\mathbf{y}}) = (\hat{y}_2, \hat{y}_1, \hat{y}_4, \hat{y}_3, \hat{y}_6, \hat{y}_5) = (\mathbf{r}_1^0, \mathbf{r}_2^0). \tag{61}$$

When calculating the fitnesses for the particles in the first swarm, the context vector is

$$\mathbf{u}^1 = (\mathbf{r}_1^0, \mathbf{r}_2^0), \tag{62}$$

so for the first particle $\mathbf{p}_{1,1} = (x_{1,2}, x_{1,1}, x_{1,4})$ in the first swarm we get simply

$$\mathbf{b}(\mathbf{p}_{1,1}, \mathbf{u}^1, 1) = (\mathbf{p}_{1,1}, \mathbf{r}_2^0) = (x_{1,2}, x_{1,1}, x_{1,4}, \hat{y}_3, \hat{y}_6, \hat{y}_5), \tag{63}$$

which is then reversed back to the original order as

$$\mathcal{P}_\sigma^{-1}(\mathbf{b}(\mathbf{p}_{1,1}, \mathbf{u}^1, 1)) = (x_{1,1}, x_{1,2}, \hat{y}_3, x_{1,4}, \hat{y}_5, \hat{y}_6). \tag{64}$$

The fitness of $\mathbf{p}_{1,1}$ is then defined to be the number we get by calculating the fitness for the vector above. On the other hand, currently the fitness of the first swarm best is simply

$$f(\mathcal{P}_\sigma^{-1}(\mathbf{b}(\mathbf{r}_1^0, \mathbf{u}^1, 1))) = f(\mathcal{P}_\sigma^{-1}(\mathbf{r}_1^0, \mathbf{r}_2^0))) = f(\mathcal{P}_\sigma^{-1}(\mathcal{P}_\sigma(\hat{\mathbf{y}}))) = f(\hat{\mathbf{y}}), \tag{65}$$

which is already known, and the swarm best gets updated to $\mathbf{r}_1^1$. When checking the same for the second particle, the fitness is

$$f(\mathcal{P}_\sigma^{-1}(\mathbf{b}(\mathbf{r}_1^1, \mathbf{u}^1, 1))) = f(\mathcal{P}_\sigma^{-1}(\mathbf{r}_1^1, \mathbf{r}_2^0)) \neq f(\hat{\mathbf{y}}), \tag{66}$$

which is also known from the previous update procedure. Now the swarm best gets updated to $\mathbf{r}_1^2$. Moving on to the first particle in the second swarm, the fitness of the swarm best is

$$f(\mathcal{P}_\sigma^{-1}(\mathbf{b}(\mathbf{r}_2^0, \mathbf{u}^2, 2))) = f(\mathcal{P}_\sigma^{-1}(\mathbf{r}_2^2, \mathbf{r}_2^0)), \tag{67}$$

which is also known from the previous update.

Now that we understand how the fitness of an s-dimensional vector is calculated, let us clarify the notation by defining a fitness function for s-dimensional vectors. For all permutations σ, for all context vectors $\mathbf{u} \in \mathbb{R}^n$, and for all swarm indices $i \in \{1, \ldots, K\}$, define the function

$$\begin{aligned} &\tilde{f}_{\sigma,\mathbf{u},i} : \mathbb{R}^s \to \mathbb{R}, \\ &\tilde{f}_{\sigma,\mathbf{u},i}(\mathbf{z}) = f(\mathcal{P}_\sigma^{-1}(\mathbf{b}(\mathbf{z}, \mathbf{u}, i))) \quad \forall \mathbf{z} \in \mathbb{R}^s. \end{aligned} \tag{68}$$

Now the update equations (55) and (56) can be written more clearly as

$$\mathbf{q}_{j,i}^{G+1} = \begin{cases} \mathbf{p}_{j,i}^{G} & \text{if } \tilde{f}_{\sigma^G,\mathbf{u}^{G,i},i}(\mathbf{p}_{j,i}^{G}) < \tilde{f}_{\sigma^G,\mathbf{u}^{G,i},i}(\mathbf{q}_{j,i}^{G}) \\ \mathbf{q}_{j,i}^{G} & \text{otherwise} \end{cases} \tag{69}$$

and

$$\mathbf{r}_i^{G,j} = \begin{cases} \mathbf{q}_{j,i}^{G+1} & \text{if } \tilde{f}_{\sigma^G,\mathbf{u}^{G,i},i}(\mathbf{q}_{j,i}^{G+1}) < \tilde{f}_{\sigma^G,\mathbf{u}^{G,i},i}(\mathbf{r}_i^{G,j-1}) \\ \mathbf{r}_i^{G,j-1} & \text{otherwise} \end{cases}. \tag{70}$$

4.2.5 Updating the global best vector

After the fitness calculations and personal and swarm best updates of all the particles in the i'th swarm, the global best vector $\hat{\mathbf{y}}^{G+1}$ is formed and updated. If the new fitness of the i'th swarm best $\mathbf{r}_i^{G,N_{\text{pop}}}$, as calculated above, got better than the best globally known fitness value $f(\hat{\mathbf{y}}^G)$, then the i'th part of the permuted global best vector $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^{G+1})$ is set to $\mathbf{r}_i^{G+1,N_{\text{pop}}}$. But because by definition $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^{G+1}) = (\mathbf{r}_1^{G+1,0}, \ldots, \mathbf{r}_K^{G+1,0})$, this can be written as

$$\mathbf{r}_i^{G+1,0} = \begin{cases} \mathbf{r}_i^{G,N_{\text{pop}}} & \text{if } \tilde{f}_{\sigma^G,\hat{\mathbf{y}}^G,i}(\mathbf{r}_i^{G+1}) < f(\hat{\mathbf{y}}^G) \\ \mathbf{r}_i^{G,0} & \text{otherwise} \end{cases}. \tag{71}$$

4.2.6 Finding the neighbourhood best

In its position update procedure, the original PSO explores the fitness landscape around the personal best and the global best position, as seen in Eq. (33). While CCPSO2 also uses the personal best position, it utilizes the best position of a local neighbourhood instead of the global best. Here the neighbourhood of the particle $\mathbf{p}_{j,i}$ includes the nearest neighbour particles $\mathbf{p}_{j-1,i}$, $\mathbf{p}_{j,i}$, and $\mathbf{p}_{j+1,i}$, which means the neighbourhood best is chosen from the vectors $\mathbf{q}_{j-1,i}^{G+1}$, $\mathbf{q}_{j,i}^{G+1}$, and $\mathbf{q}_{j+1,i}^{G+1}$. In order to define the neighbours of the first and the last particle, the basic ring-topology is used to connect the ends of this list together.

In Algorithm 6 the function "localBest" implements exactly this behaviour. Calling localBest(Q_i^{G+1}, j) for the j'th particle returns the best-fit vector of the set $\{\mathbf{q}_{j-1,i}^{G+1}, \mathbf{q}_{j,i}^{G+1}, \mathbf{q}_{j+1,i}^{G+1}\}$, where their fitnesses have already been calculated in the last step. It also defines $\mathbf{q}_{N_{\text{pop}}+1,i}^{G+1} = \mathbf{q}_{1,i}^{G+1}$ and $\mathbf{q}_{0,i}^{G+1} = \mathbf{q}_{N_{\text{pop}},i}^{G+1}$ to satisfy the boundary conditions discussed above. This function is used to select the neighbourhood-best vector $\mathbf{n}_{j,i}^{G+1}$ of each particle $j \in \{1, \ldots, N_{\text{pop}}\}$ in each swarm $i \in \{1, \ldots, K\}$, and these are used in the position update procedure described next.

4.2.7 Position update and saturation

As the name suggests, CCPSO2 uses a modified version of particle swarm optimization as the actual optimizer. The outline of this optimizer is the same as in the basic PSO but the position update rule is very different from Eq. (34) and there are no velocities present. Instead, the permuted population is updated as follows. First, new components $k \in \{1, \ldots, s\}$ for the j'th particle $\mathbf{p}_{j,i}$ in the i'th swarm are calculated as[¶]

$$\tilde{p}_{j,i,k}^{G+1} = \begin{cases} \text{Cauchy}(q_{j,i,k}^{G+1}, |q_{j,i,k}^{G+1} - n_{j,i,k}^{G+1}|/2) & \text{if } r_{j,i,k} \leq p \\ \text{Gaussian}(n_{j,i,k}^{G+1}, |q_{j,i,k}^{G+1} - n_{j,i,k}^{G+1}|/2) & \text{otherwise} \end{cases}. \tag{72}$$

This might produce vectors outside the rectangle D, so the components are transformed back inside it by some sensible, user-defined function as

$$p_{j,i,k}^{G+1} = \text{saturate}(\tilde{p}_{j,i,k}^{G+1}). \tag{73}$$

The original version has (although not mentioned in the article) a saturation function that simply mirrors the solution from the boundary but this does not guarantee the saturated solution to be in the feasible area D. Instead, I use a different saturation function as described in Section 5.

Now that the particle $\mathbf{p}_{j,i}^{G+1}$, updated above, is part of the permuted population matrix $\mathcal{P}_{\sigma^G}(X^{G+1})$ of the next generation, so X^{G+1} gets also formed here. In Eq. (72) $p \in [0, 1]$ is a user-defined probability that chooses the ratio of exploration/exploitation and $r_{j,i,k} \in [0, 1]$ is a random number from a uniform distribution, and it is calculated again for each i, j, and k. Furthermore, Cauchy(a, b) is a random number following the Cauchy distribution with the location parameter (the "mean") a and the scale parameter (the "standard deviation") b and Gaussian(a, b) is a random number following the Gaussian distribution with the mean a and the standard deviation b. They are also called again for each i, j, and k.

Above, the short-tailed Gaussian distribution is used for local optimization around the local best vector $\mathbf{n}_{j,i}^{G+1}$ of the neighbourhood (exploitation) while the long-tailed Cauchy distribution is used for searching wider areas around the personal best vector $\mathbf{q}_{j,i}^{G+1}$ (exploration).

¶. Note the 1/2 difference to the original article.

4.2.8 Summary of the algorithm and further notes

Again as a summary, a pseudocode of CCPSO2 is shown in Algorithm 6. Here a small modification with respect to the original implementation is presented. In the original version, during the first generation, new random particles from each swarm were chosen to the context vector whenever needing it, instead of the usual swarm bests. In this new version, I simply initialize $\hat{\mathbf{y}}^0$ by picking a random row from X^0 in the beginning. This is much simpler, and according to tests I made doesn't change the behaviour of the algorithm.

The explanation of the algorithm presented here should help in understanding the algorithm mathematically. However, because the explanation is quite technical, its implementation might be nontrivial. This is why there is also another explanation of CCPSO2 presented in Appendix A that is written in pseudo-Java styled code. That version should help in implementing the algorithm in practice.

Algorithm 6: A pseudocode of the CCPSO2 algorithm. The termination criterion is usually a maximum number of generations or fitness function evaluations.

```
Data: Rectangular search space $D \subset \mathbb{R}^n$, fitness function $f : D \to \mathbb{R}$,
      population size $N_{\text{pop}}$, Cauchy/Gaussian search rate parameter $p \in [0, 1]$,
      set of possible group sizes $S$
Initialize a random population: for each individual $i$, generate a random
position $\mathbf{x}_i^0 \in D$, form the population matrix $X^0$ out of these, and set the
personal best matrix to $Y^0 = X^0$;
Initialize the global best vector: set a random row from $X^0$ to $\hat{\mathbf{y}}^0$;
Choose a group size $s \in S$ randomly and set number of swarms to $K = n/s$;
Initialize the permutation to $\sigma^0 = (12 \dots n)$;
Initialize the generation $G = 0$;
while termination criterion is not met do
    if $\hat{\mathbf{y}}$ didn't improve then
        Randomly choose a new group size $s \in S$ and set $K = n/s$;
    // Form $K$ swarms
    foreach $i \in \{1, \dots, K\}$ do
        Denote the columns $i, \dots, i + s$ of the permuted population $\mathcal{P}_{\sigma^G}(X^G)$ by
        the particle matrix $P_i^G = (\mathbf{p}_{1,i}^G, \dots, \mathbf{p}_{N_{\text{pop}},i}^G)^\top$ of the swarm;
        Denote the columns $i, \dots, i + s$ of the permuted matrix $\mathcal{P}_{\sigma^G}(Y^G)$ by the
        personal best matrix $Q_i^G = (\mathbf{q}_{1,i}^G, \dots, \mathbf{q}_{N_{\text{pop}},i}^G)^\top$ of the swarm;
        Denote the components $i, \dots, i + s$ of the permuted best vector $\mathcal{P}_{\sigma^G}(\hat{\mathbf{y}}^G)$
        by the swarm best vector $\mathbf{r}_i^{G,0}$;
    foreach swarm $i \in \{1, \dots, K\}$ do
        foreach particle $j \in \{1, \dots, N_{pop}\}$ do
            // Update $Y^{G+1}$
            Update the personal best $\mathbf{q}_{j,i}^{G+1}$ according to Eq. (69);
            // Update the context vector $\mathbf{u}^{G,i}$
            Update the swarm best $\mathbf{r}_i^{G,j}$ according to Eq. (70);
        Update the global best vector $\hat{\mathbf{y}}^{G+1}$ according to Eq. (71);
    foreach swarm $i \in \{1, \dots, K\}$ do
        foreach particle $j \in \{1, \dots, N_{pop}\}$ do
            Find the neighbourhood best $\mathbf{n}_{j,i}^{G+1} = \text{localBest}(Q_i^{G+1}, j)$;
            // Update $X^{G+1}$
            Update the position of the $j$'th particle $\mathbf{p}_{j,i}^{G+1}$ in the $i$'th swarm
            according to Eqs. (72) and (73);
    Randomly generate a new permutation $\sigma^{G+1}$ of $(12 \dots n)$;
    $G$++;
return the global best position $\hat{\mathbf{y}}^{G+1}$
```

5 Efficiency of population-based metaheuristics and practical separability of the Lennard-Jones problem

In this section I will present two main results of this thesis. Firstly, I will show how the population-based metaheuristics presented in the previous section, that are all general all-round optimizers, compare against problem-tailored methods and simulated annealing. Secondly, I will study the degree of separability of the Lennard-Jones cluster problem.

For fair comparison, I used $5000n$ function evaluations as the stopping criterion for all the algorithms, where n is the problem dimension. Furthermore, to get a little statistics, I repeated each of the optimization runs 8 times with new random initial configurations. So in this section whenever I am talking about an "average" or a "mean", I have taken the average of the results of 8 independent optimization runs. For all the algorithms except the two versions of simulated annealing, I defined the saturation function to transform the infeasible solutions back to the feasible area using periodic boundary conditions. Mathematically, if the number $x \in \mathbb{R}$ should be bounded by the inequality $a \leq x \leq b$, the saturation function returns

$$\text{saturate}_{a,b}(x) = \begin{cases} b - [(a-x) \mod (b-a)] & \text{if } x < a \\ a + [(x-b) \mod (b-a)] & \text{if } x > b \\ x & \text{otherwise} \end{cases} . \tag{74}$$

5.1 Comparison between population-based metaheuristics, simulated annealing, and problem-tailored methods

So far I have presented algorithms that are not that widely known among physicists and that can be applied to any problem without much tuning. How do they compare against more traditional methods such as simulated annealing and, on the other hand, against problem-tailored methods that are specifically developed for the Lennard-Jones problem?

5.1.1 Settings of the algorithms

First of all, in addition to algorithm parameters, let me specify the simulated annealing (Algorithm 1) versions I used. In the neighbour generation rule in the first version, which I call SA1, $\mathbf{r}^k$ is a uniformly distributed random vector from D such that $||\mathbf{r}^k|| = 1$, as suggested in Ref. [53]. In this version the step size θ^k is not adjusted, so we have $A(\theta^k) = \theta^k = \theta^0 = 0.002$ for all k. The temperature reduction factor and initial temperature were chosen as $\chi = 0.88$ and $T^0 = 0.9$, respectively. The number of steps at each temperature was kept at a constant value of $l^k = 100$.

In the second version, SA2, $\mathbf{r}^k$ is a uniformly distributed random vector in $[-1, 1]^n$. The step size is adjusted adaptively as

$$A(\theta^k) = \begin{cases} 2\theta^k & \text{if acceptance} > 50\ \% \\ \theta^k/2 & \text{if acceptance} < 25\ \%\ , \\ \theta^k & \text{otherwise} \end{cases} \tag{75}$$

where acceptance means the acceptance ratio of accepted moves during the previous temperature loop. This should keep the acceptance ratio at a desirable 25–50 % level as discussed in Ref. [54]. On the other hand, to allow for more trials at lower temperatures and at higher dimensions, I set the number of trials at the k'th step to

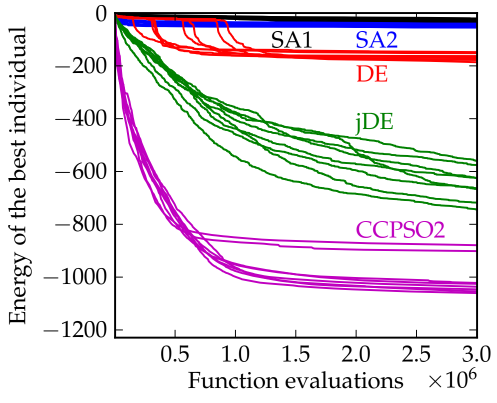
$$l^k = \frac{n}{3} \log\left(\frac{n}{3}\right)\left(1 - \log\left(T^k\right)\right)$$

as suggested in Ref. [23]. The other parameters were chosen as $\chi = 0.8$, $T^0 = 1.3$ and $\theta^0 = 0.001$.

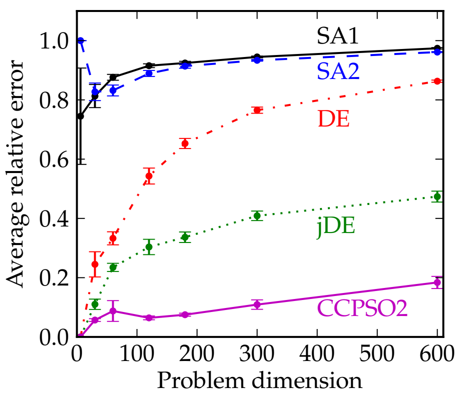
In this study the parameters for each algorithm were roughly tuned for the 120-dimensional problem for fair comparison. While the results provided by SA1, SA2, DE, and jDE were highly dependent on their parameters, CCPSO2 seemed to be very robust with respect to its parameters N_{pop} and p: as long as the population size was $\gtrsim 10$, neither of the parameters produced any noticeable differences in the results. So for CCPSO2 I simply used the same values for these parameters as suggested in the original article [18]: $N_{\text{pop}} = 30$ and $p = 0.5$. It became also immediately clear that the group size of 1 does not work at all for this problem (more discussion in Section 5.2), so for the set S I chose the one in Eq. (76), with 1 removed. As a summary, all the parameters and settings used for the algorithms are shown in Tables 1 and 2.

5.1.2 Comparison results

In Fig. 7 I have done the comparison between different algorithms in the Lennard-Jones problem in 6, 30, 120, 180, 300, and 600 dimensions. In Fig. 7a the improvement of the best individual in the population is shown for each algorithm and for each of the 8 independent optimization runs in the 600-dimensional problem. The bottom energy level is the *target energy*, *i.e.* the lowest energy that any (problem-tailored) algorithm has ever discovered, as listed in Ref. [19]. While every algorithm is quite robust between the different runs,



(a) $n = 600$



(b)

Figure 7. How the all-round optimizers differential evolution (DE), jDE, and CCPSO2 compare against the problem-tailored methods and, on the other hand, against two versions of simulated annealing (SA) in the Lennard-Jones problem in 6, 30, 60, 120, 180, 300, and 600 dimensions. **(a)** A closer look at the 600-dimensional problem showing the improvement of the best individual with increasing number of fitness function evaluations. Each of the 8 independent optimization runs are shown for each algorithm. The bottom energy level is the target energy, *i.e.* the lowest energy that any (problem-tailored) algorithm has ever found [19]. **(b)** The average relative error $|E_{\text{avg}} - E_{\text{target}}|/|E_{\text{target}}|$, where E_{avg} is the average final minimum energy found by the algorithm and E_{target} is the target energy [19], in different dimensions.

Table 1. Settings and parameters for the two simulated annealing algorithms in the comparison study.

SA1	SA2
$\mathbf{r}^k = \text{rand}() \in \mathbb{R}^n$ s.t. $\lvert\lvert \mathbf{r}^k \rvert\rvert = 1$	$\mathbf{r}^k = \text{rand}() \in [-1,1]^n$
$A(\theta^k) = \theta^k$	$A(\theta^k)$ as in Eq. (75)
$l^k = 100$	$l^k = \frac{n}{3}\log\left(\frac{n}{3}\right)\left(1-\log\left(T^k\right)\right)$
$\chi = 0.88$	$\chi = 0.8$
$T^0 = 0.9$	$T^0 = 1.3$
$\theta^0 = 0.002$	$\theta^0 = 0.001$

Table 2. Parameters for the algorithms DE, jDE, and CCPSO2 in the comparison study.

DE	jDE	CCPSO2
$N_{\text{pop}} = 35$	$N_{\text{pop}} = 20$	$N_{\text{pop}} = 30$
$F = 0.4$		$p = 0.5$
$C = 0.9$		$S_n = \{s \in \mathbb{N} : s \mid n\} \setminus \{1\}$

there are large differences between the algorithms. Both versions of simulated annealing, as well as differential evolution, get stuck very early and are thus poor optimizers for this problem. jDE is already much better, but CCPSO2 is able to get quite close to the target energy already after a million function evaluations. A million may sound a lot but imagine having a (unrealistically small) grid consisting of only 2 points in each of the 600 coordinate axes: the total grid would have an enormous amount of $2^{600} \approx 10^{180}$ points.

I then took the final minima the algorithms found, calculated their average E_{avg} and standard error of the mean for each algorithm, and repeated the study for the dimensions 6, 30, 60, 120, 180, 300, and 600. The results are presented in Fig. 7b where, for better comparison between different dimensions, I have plotted the *relative* error $|E_{\text{avg}} - E_{\text{target}}|/|E_{\text{target}}|$, where E_{target} is the target energy [19]. Both versions of simulated annealing are poor optimizers for this problem, having a large ~ 90 % error in all dimensions. Differential evolution works well in low dimensions but its performance deteriorates quickly when going into higher dimensions. jDE, on the other hand, is a little more robust between different dimensions, reaching a ~ 30–40 % accuracy. However, CCPSO2 works surprisingly well in all the studied dimensions, having a quite good and robust ~ 10–20 % accuracy.

It must be noted that the stopping criterion of $5000n$ (n problem dimension) function evaluations is quite arbitrary. Looking at the graphs in Fig. 7a one can see that both jDE and CCPSO2 are still decreasing in energy in the end, although CCPSO2 pretty slowly. This would allow one to easily achieve better results simply by running the optimization process longer. A single optimization run of the 600-dimensional problem of 3 million evaluations took around 6–8 hours on a 2.67 GHz Intel Xeon in series, so longer runs would be affordable.

5.2 Practical separability of the Lennard-Jones cluster problem

Let us now analyze the degree of separability of the Lennard-Jones cluster problem defined in Eq. (12). By looking at the definition of the objective function it is clear that the problem is fully nonseparable; every variable interacts with every other one. But let us study the separability from a practical perspective: are there algorithms that can solve the problem efficiently by dividing it into smaller pieces? CCPSO2 seems to be an optimal algorithm for this study since it both performs well (Fig. 7) and it has the property of dividing the problem adaptively into smaller pieces, where the possible group sizes (dimension of the subproblems) are determined by the user. It uses the same group size as long as it works, but whenever the algorithm gets stuck it chooses a new one. If we then allowed CCPSO2 to use every possible group size available we could easily see which group sizes are favoured and which avoided.

To do the study I first performed optimization runs of CCPSO2 in the Lennard-Jones cluster problem in dimensions 6, 30, 60, 120, 180, 300, and 600, where I allowed the algorithm to use every possible group size available. To achieve this I simply chose the set S in the n-dimensional case to include all the natural numbers that divide n, *i.e.*

$$S_n = \{s \in \mathbb{N} : s \mid n\}. \tag{76}$$

As an example, for 30-dimensional problem I used

$$S_{30} = \{1, 2, 3, 5, 6, 10, 15, 30\}.$$

Because the Lennard-Jones problem is fully nonseparable it could be expected that CCPSO2 uses a pretty high group size throughout the runs. But when looking at the group size behaviour of typical runs in 600D, shown in Fig. 8, a complete opposite happens. In some of the runs the group size saturates to a very low value of 1–6 after a short transitional phase (Fig. 8a), while in the rest of the runs a small group size of 1–6 is used throughout the run (Fig. 8b). I call this phenomenon *practical separability*, meaning that even though the problem is

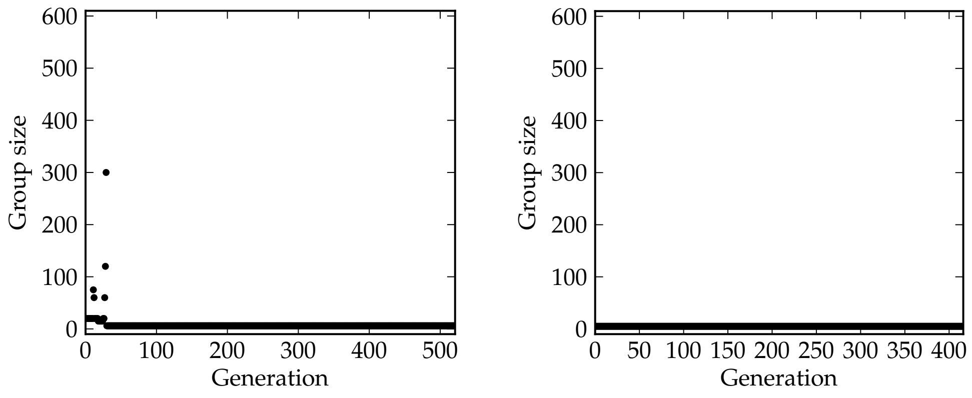


Figure 8. Two examples of typical adaptive group size behaviour of CCPSO2 in the 600-dimensional Lennard-Jones problem when allowing the algorithm to use every possible group size. **(a)** In some of the runs the group size saturates to a very low value (6 in this case) after a short transitional phase, **(b)** while in the rest of the runs a small group size (5 in this case) is used throughout the run. Exactly the same behaviour happens in all the studied dimensions, except in 6.

strictly nonseparable, in practice it can be solved effectively (by some algorithm) by dividing it into smaller subproblems.

There is, however, an interesting behaviour regarding the group size 1. In Fig. 9 improvement of the best individual in the 600-dimensional problem is shown for all the 8 runs. The algorithm is otherwise very robust between the different runs but there is one run being very ineffective. A closer look reveals that the algorithm happened to favour a constant group size of 1 in this run, which seems to be ineffective for this problem. The algorithm did not nevertheless discard it because it constantly produced lower energies, although very slowly.

The behaviour discussed above for 600-dimensional problem happened also in all the other dimensions[‖] studied: CCPSO2 favours small group sizes over large ones, but whenever the group size 1 gets chosen, it usually quickly degrades the performance of the algorithm.

But exactly *how* separable is the Lennard-Jones problem in the practical sense? Inspired by the results above, I tried also versions where I kept the group size at a constant value. Results for constant group sizes 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60 are shown in Fig. 10 where, for meaningful comparison between

‖. The dimension 6 is an exception since its minimum is found so quickly that it does not provide a sensible testbed for this study.

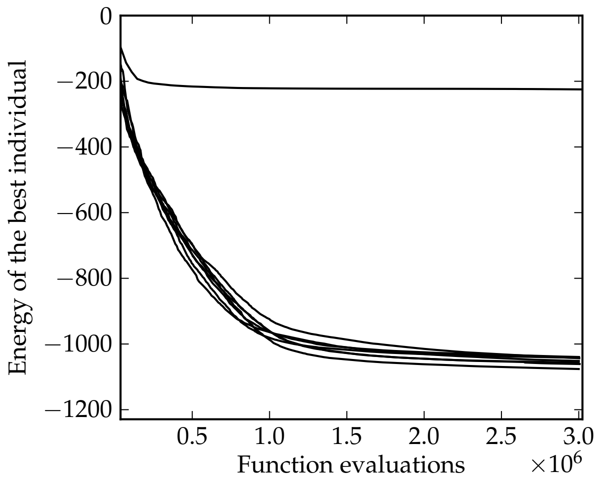


Figure 9. Improvement of the energy of the best individual of CCPSO2 in the 600-dimensional Lennard-Jones problem in each of the 8 runs. The algorithm seems to be very robust between the different runs with the exception of one run being very ineffective.

different dimensions, average relative error between the final energy and the target energy (the lowest energy currently known, as listed in Ref. [19]) is plotted for the dimensions (6, 30,) 60, 120, 180, 300, and 600. The dimensions 6 and 30 are in parentheses because they do not include all of the group sizes. For example, the group sizes 60 and 20 cannot be used for the 30-dimensional problem simply because 30 is not divisible by them.

The results are in good agreement with what was found by the earlier analysis: small group sizes work well while large group sizes and the group sizes 1 and 2 do not. More quantitatively, group sizes 1 and 2 are poor, 4, 5, 6, 10, and 12 are the best, and all the other group sizes work only in low dimensions. This is remarkable because this tells us that there exists an algorithm (CCPSO2) that can solve the difficult Lennard-Jones problem by dividing it into very low-dimensional subproblems, dramatically lowering the difficulty of the problem. Preliminary results on other benchmark problems also show that this behaviour is not a universal property of CCPSO2, but instead a property of the Lennard-Jones problem and the algorithm together. However, it is not known yet whether this is a universal property of the problem, *i.e.* whether it is always better to divide it into very small pieces regardless of the algorithm.

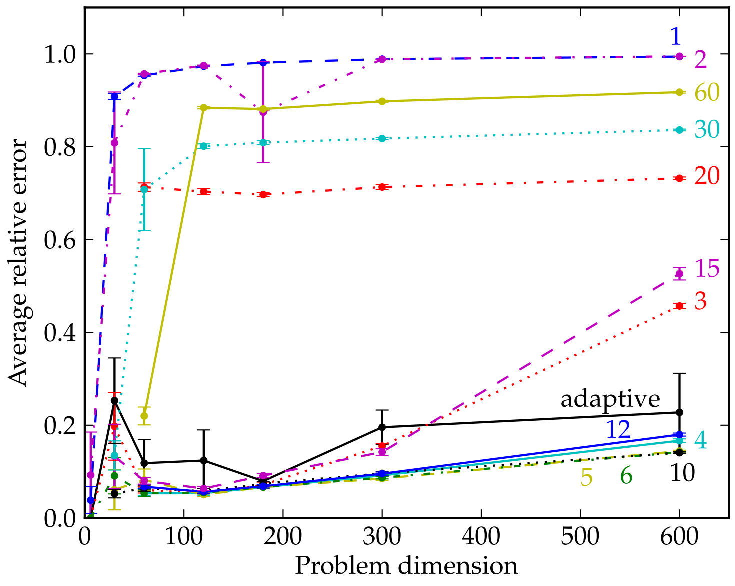


Figure 10. How the different constant group size versions of CCPSO2 perform in the Lennard-Jones problem in the dimensions (6, 30,) 60, 120, 180, 300, and 600. The coloured number in side of each graph denotes the constant group size of the corresponding CCPSO2 version, while the continuous, black curve is the "original" adaptive version where the algorithm can use every possible group size as in Eq. (76). For sensible comparison between different dimensions, the vertical axis is the average relative error $|E_{\text{avg}} - E_{\text{target}}|/|E_{\text{target}}|$, where E_{avg} is the average of the final best energies of the 8 runs and E_{target} is the target energy, *i.e.* the lowest energy currently known, as listed in Ref. [19]. Roughly, small group sizes work well while large group sizes do not, with 1 and 2 as exceptions.

6 Conclusions

In this work I did a survey on population-based metaheuristic global optimization algorithms, namely evolutionary algorithms and swarm intelligence, that try to be general all-round optimizers. As an example of a "real life" optimization problem in computational nanoscience, I used the Lennard-Jones cluster problem to show that these methods are, on the other hand comparable to problem-tailored methods, but also much better than two versions of simulated annealing, a popular method among physicists.

While both versions of simulated annealing were particularly poor optimizers in the Lennard-Jones problem in general, simple differential evolution worked already much better in low dimensions. The self-adaptive version of differential evolution, jDE, performed better also in high dimensions but left still room for improvement. CCPSO2, on the other hand, was really in its own class, although not finding the correct minima. It provided a surprisingly good and robust $\sim$ 10–20 % accuracy up to 600 dimensions (200 atoms), and it even reached it with a very small amount of function evaluations.

Moreover, CCPSO2 seemed to be very robust with respect to its parameters, providing a truly attractive alternative to problem-tailored methods. The only important parameter is the set S of possible group sizes available to the algorithm, but this is only important if trying to enhance the efficiency even further. Very good results can be obtained by simply setting S to include all the numbers s that divide the problem dimension, thus providing a general efficient algorithm that can be used with very little tuning. Although it could not find the actual minimum, it certainly can be used as a robust basis for further development of new, efficient problem-tailored algorithms. A few of the first modification ideas would be to tune the position update rule and to couple the algorithm with some local optimizers.

It was also shown that there exists an algorithm, namely CCPSO2, that is able to solve efficiently the Lennard-Jones cluster problem by dividing it into very small subproblems with a dimension of only 4–12, independent of the problem dimension. This was surprising since the problem is actually highly nonseparable, and thus this division should not have been possible. This is remarkable because the problem difficulty grows exponentially with dimension, so transforming one high-dimensional problem into many lower-dimensional problems is a dramatic relief. This gives a hint that when designing new algorithms for the Lennard-Jones problem one should probably stick to methods that perturb only a small subset of the variables at a time. Preliminary studies on other benchmark problems showed that this indeed is a property of the Lennard-

Jones problem and the algorithm together rather than only a property of the algorithm. However, it is still a matter of future studies whether this practical separability is a universal property of the problem among all algorithms.

Appendix A Pseudo-Java styled CCPSO2

The explanation of CCPSO2 in Section 4.2 is mathematically exact, which means the equality sign "$=$" always denotes the normal equality equivalence relation. That presentation is, however, quite technical, so I will present an alternative explanation here that should help in implementing the algorithm in practice.

The following pseudocodes are written in a style resembling Java, which is why I call them pseudo-Java styled. Variables, procedures, and programming operators are written in `teletype` font while everything else, mainly verbal descriptions, are written in the normal roman font. The equality sign "`=`" now denotes the *substitution* sign familiar from programming languages.

The pseudo-Java styled version of CCPSO2 is presented in Algorithm 7, but because it is quite lengthy, it is splitted in smaller procedures. Note that the matrix `pBests` is stored in a transposed form with respect to Section 4.2 for computational efficiency. Furthermore, the following small utility procedures are needed:

- `reverse(x, perm)`:
  form a new vector that is `x` reversed back to its original permutation (when `x` is permuted by `perm`)
- `b(x, context, i)`:
  form a new vector where the `i`'th part of `context` is replaced by the vector `x` (when `context` is divided into `K` $=$ `n/s` parts, where `s` is the length of `x`)
- `getVector(X, perm, i, j, s)`:
  get a copy of the `j`'th part of the `i`'th swarm of the permuted `X` matrix, when each row of `X` is divided into `K` $=$ `n/s` parts
- `setVector(X, perm, x, i, j)`:
  copy the contents of `x` into the `j`'th part of the `i`'th swarm of the permuted `X` matrix (when each row of `X` is divided into `K` $=$ `n/s` parts, where `s` is the length of `x`)
- `getSwarmthPart(yhat, perm, i, s)`:
  get the `i`'th part of the permuted `yhat` vector, when each part has the dimension `s`
- `setSwarmthPart(yhat, perm, x, i)`:
  copy the contents of `x` into the `i`'th part of the permuted `yhat` vector (when `yhat` is divided into `K` $=$ `n/s` parts, where `s` is the length of `x`)
- `saturate(x, bounds)`:
  transform `x` back to the feasible area according to `bounds` and some user-defined rule

- `localBest(pBests, pBestsFits, i, j)`:
  get the best-fit particle in the neighbourhood of the `j`'th particle in the `i`'th swarm, when the neighbourhood includes the vectors `pBests[i][j]`, `pBests[i][j-1]`, and `pBests[i][j+1]` s.t. periodic boundary conditions are used
- `cauchy(a, b)`:
  get a random number following the Cauchy distribution, where `a` is its location parameter and `b` is its scale parameter
- `gaussian(a, b)`:
  get a random number following the Gaussian distribution, where `a` is its mean and `b` is its standard deviation

Algorithm 7: Main outline of CCPSO2 in a pseudo-Java styled code.

```
Data: problem dimension n, fitness function f, bounds: n×2 matrix
      including the lower and upper bounds for each variable, population size
      Npop, Cauchy/Gaussian search rate parameter p, set of possible group
      sizes S
// Initialization
perm = [0,1,...,n-1];
s = random element from S;                              // group size
K = n / s;                                               // number of swarms
X = random Npop×n matrix s.t. each row is
    limited by bounds;                                   // population matrix
Y = X.copy();                                            // personal best matrix
yhat = random row from X;                                // global best
fhat = f(yhat);                                          // fitness of yhat
G = 0;                                                   // generation
while termination criterion is not met do
    if yhat did not improve then
        s = random element from S;
        K = n / s;
    // Form K swarms
    pBests,pBestsFits,sBests,sBestsFits =
        formSwarms(Y,perm,s,K,fhat);
    foreach swarm i = 0,...,K-1 do
        foreach particle j = 0,...,Npop-1 do
            // Form the n-dimensional context vector
            context = [sBests[0],...,sBests[K-1]];
            updatePersonalAndSwarmBests(X,perm,pBests,pBestsFits,
                sBests,sBestsFits,i,j,s,context));
        updateGlobalBest(sBests,sBestsFits,yhat,perm,fhat);
    foreach swarm i = 0,...,K-1 do
        foreach particle j = 0,...,Npop-1 do
            // Calculate a new position and update X and Y
            evolvePopulation(X,Y,perm,pBests,pBestsFits,i,j);
    G++;
    perm = random permutation of [0,1,...,n-1];
return the global best position yhat
```

Procedure `formSwarms(Y, perm, s, K, fhat)`

```
sBests = new double[K][s];                    // swarm bests
sBestsFits = new double[K];                   // their fitnesses
pBests = new double[K][Npop][s];              // personal bests
pBestsFits = new double[K][Npop];             // their fitnesses
for i = 0, ..., K-1 do
    for j = 0, ..., Npop-1 do
        // Pick personal bests from the permuted Y matrix
        pBests[i][j] = getVector(Y, perm, i, j, s);
    // Pick swarm bests from the permuted yhat vector
    sBests[i] = getSwarmthPart(yhat, perm, i, s);
    sBestsFits[i] = fhat;
return pBests, pBestsFits, sBests, sBestsFits
```

Procedure `updatePersonalAndSwarmBests(X, perm, pBests, pBestsFits, sBests, sBestsFits, i, j, s, context)`

```
// Pick the current vector from the permuted X matrix
currentVector = getVector(X, perm, i, j, s);
// Calculate its fitness in the current context
currentFullVector = b(currentVector, context, i);
currentFit = f(reverse(currentFullVector, perm));
// Calculate fitness of the personal best in the current context
pBestFull = b(pBests[i][j], context, i);
pBestsFits[i][j] = f(reverse(pBestFull, perm));
// Update the personal best
if currentFit < pBestsFits[i][j] then
    pBests[i][j] = currentVector.copy();
    pBestsFits[i][j] = currentFit;
// Update the swarm best
if pBestsFits[i][j] < sBestsFits[i] then
    sBests[i] = pBests[i][j].copy();
    sBestsFits[i] = pBestsFits[i][j];
```

```
Procedure updateGlobalBest(sBests, sBestsFits, yhat, perm, fhat)
// Update yhat
if sBestsFits[i] < fhat then
    fhat = sBestsFits[i];
    for k = 0, ..., K-1 do
        // Replace the k'th part of yhat with the corresponding
        // swarm best
        setSwarmthPart(yhat, perm, sBests[k], k)
```

```
Procedure evolvePopulation(X, Y, perm, pBests, pBestsFits, i, j)
// Find the neighbourhood best
nBest = localBest(pBests, pBestsFits, i, j);
// Cauchy/Gaussian position update
newPos = calculateNewPosition(pBests[i][j], nBest);
// Transform newPos back to the feasible area
newPos = saturate(newPos, bounds);
// Update X and Y
setVector(X, perm, newPos, i, j);
setVector(Y, perm, pBests[i][j], i, j);
```

```
Procedure calculateNewPosition(pBest, nBest)
newPos = new double[s];
for k = 0, ..., s-1 do
    if rand(0, 1) <= p then
        newPos[k] = cauchy(pBest[k], abs(pBest[k]-nBest[k])/2);
    else
        newPos[k] = gaussian(nBest[k], abs(pBest[k]-nBest[k])/2);
return newPos
```

References

[1] R. Storn and K. Price. Differential evolution — a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[2] K. Ohkura, Y. Matsumura, K. Ueda, and X. Yao. Robust evolution strategies. *Applied Intelligence*, 1585:10–17, 1999.

[3] X. Li and X. Yao. Tackling high dimensional nonseparable optimization problems by cooperatively coevolving particle swarms. In *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation*, CEC'09, pages 1546–1553, Piscataway, NJ, USA, 2009. IEEE Press.

[4] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, 1995*, pages 39–43, Oct 1995.

[5] K. P Wong and Z. Dong. Differential evolution, an alternative approach to evolutionary algorithm. In *Proceedings of the 13th International Conference on Intelligent Systems: Application to Power Systems*, pages 73–83, November 2005.

[6] S. Kannan, S. M. R. Slochanal, and N. P. Padhy. Application and comparison of metaheuristic techniques to generation expansion planning problem. *IEEE Transactions on Power Systems*, 20(1):466–475, February 2005.

[7] J. Chiou, C. Chang, and C. Su. Ant direction hybrid differential evolution for solving large capacitor placement problems. *IEEE Transactions on Power Systems*, 19(4):1794–1800, 2004.

[8] J. Chiou, C. Chang, and C. Su. Variable scaling hybrid differential evolution for solving network reconfiguration of distribution systems. *IEEE Transactions on Power Systems*, 20(2):668–674, May 2005.

[9] R. K. Ursem and P. Vadstrup. Parameter identification of induction motors using differential evolution. In *The 2003 Congress on Evolutionary Computation*, volume 2, pages 790–796, December 2003.

[10] R. Storn. Differential evolution design of an IIR-filter. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 268–273, May 1996.

[11] A. D. Brown and H. C. Card. Evolutionary artificial neural networks. In *IEEE 1997 Canadian Conference on Electrical and Computer Engineering*, volume 1, 1997.

[12] N. Chakraborti, K. Misra, P. Bhatt, N. Barman, and R. Prasad. Tight-binding calculations of Si-H clusters using genetic algorithms and related techniques: Studies using differential evolution. 22(5):525–530, 2001.

[13] V. Purmonen. *Differentiaalilaskentaa 1*. Department of mathematics and statistics, lecture notes 52. University of Jyväskylä, 4th edition, 2010.

[14] R. E. Bellman. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[15] R. E. Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 2003.

[16] R. E. Bellman. *Adaptive Control Processes: A Guided Tour*. Rand Corporation. Research studies. Princeton University Press, 1961.

[17] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, PPSN III, pages 249–257, London, UK, 1994. Springer-Verlag.

[18] X. Li and X. Yao. Cooperatively coevolving particle swarms for large scale optimization. *IEEE Transactions on Evolutionary Computation*, 16(2):210–224, April 2012.

[19] J. Doye. Lennard-Jones clusters. `http://doye.chem.ox.ac.uk/jon/structures/LJ.html`, 1997.

[20] V. Talanquer. A new phenomenological approach to gas–liquid nucleation based on the scaling properties of the critical nucleus. *The Journal of Chemical Physics*, 106(23):9957–9960, 1997.

[21] J. A. White. Lennard-Jones as a model for argon and test of extended renormalization group calculations. *The Journal of Chemical Physics*, 111(20):9352–9356, 1999.

[22] L. T. Wille. Simulated annealing and the topology of the potential energy surface of Lennard-Jones clusters. *Computational Materials Science*, 17:551–554, 2000.

[23] T. Coleman, D. Shalloway, and Z. Wu. Isotropic effective energy simulated annealing searches for low energy molecular cluster states. *Computational Optimization and Applications*, 2:145–180, 1993.

[24] T. Coleman, D. Shalloway, and Z. Wu. A parallel build-up algorithm for global energy minimizations of molecular clusters using effective energy simulated annealing. *Journal of Global Optimization*, 4:171–185, 1994.

[25] J. Zhang. A brief review on results and computational algorithms for minimizing the Lennard-Jones potential. *ArXiv e-prints*, December 2010.

[26] D. M. Deaven, N. Tit, J. R. Morris, and K. M. Ho. Structural optimization of Lennard-Jones clusters by a genetic algorithm. *Chemical Physics Letters*, 256(195):195–200, June 1996.

[27] W. Cai, Y. Feng, X. Shao, and Z. Pan. Optimization of Lennard-Jones atomic clusters. *Journal of Molecular Structure: THEOCHEM*, 579(1–3):229–234, March 2002.

[28] M. R. Hoare and J. A. McInnes. Morphology and statistical statics of simple microclusters. *Advances in Physics*, 32(5):791–821, January 1983.

[29] R. Elber and M. Karplus. A method for determining reaction paths in large molecules: application to myoglobin. *Chemical Physics Letters*, 139(5):375–380, September 1987.

[30] I. Grigorenko and M. E. Garcia. An evolutionary algorithm to calculate the ground state of a quantum system. *Physica A*, (284):131–139, 2000.

[31] J. C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control.* Wiley, 2003.

[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[33] M. Locatelli. Simulated annealing algorithms for continuous global optimization. In P. M. Pardalos and H. E. Romeijn, editors, *Nonconvex Optimization and Its Applications*, volume 62, pages 179–229. Springer US, 2002.

[34] D. J. Wales and J. P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. *J. Phys. Chem. A*, 101(28):5111–5116, July 1997.

[35] D. J. Wales. *Energy Landscapes: Applications to Clusters, Biomolecules and Glasses*. Cambridge Molecular Science, 2004.

[36] G. G. Rondina and L. F. Da Silva. Revised basin-hopping monte carlo algorithm for structure optimization of clusters and nanoparticles. *Journal of Chemical Information and Modeling*, 53:2282–2298, 2013.

[37] A. Prügel-Bennett. Benefits of a population: Five mechanisms that advantage population-based algorithms. *IEEE Transactions on Evolutionary Computation*, 14(4):500–517, Aug 2010.

[38] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. 8(2):239–287, 2009.

[39] A. M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, October 1950.

[40] N. A. Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.

[41] N. A. Barricelli. Symbiogenetic evolution processes realised by artificial methods. *Methodos*, pages 143–182, 1957.

[42] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[43] J. H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–72, 1992.

[44] G. J. E. Rawlins. *Foundations of Genetic Algorithms 1991 (FOGA 1).* Foundations of Genetic Algorithms. Morgan Kauffmann, 1st edition, 1991.

[45] K. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution: A Practical*

Approach to Global Optimization. Natural Computing Series. Springer Science and Business Media, 2006.

[46] J. Vesterstrøm and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Congress on Evolutionary Computation, 2004. CEC2004*, volume 2, pages 1980–1987, June 2004.

[47] F. Neri and V. Tirronen. Recent advances in differential evolution: a survey and experimental analysis. *Artificial Intelligence Review*, 33(1–2):61–106, 2010.

[48] F. van der Bergh and A. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):225–239, 2004.

[49] J. Brest, S. Greiner, B. Bošković, M. Mernik, and V. Žumer. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE Transactions on Evolutionary Computation*, 10(6):646–657, December 2006.

[50] J. Brest, B. Bošković, S. Greiner, V. Žumer, and M. S. Maučec. Performance comparison of self-adaptive and adaptive differential evolution algorithms. *Soft Computing*, 11(7):617–629, 2007.

[51] Z. Yang, K Tang, and X. Yao. Multilevel cooperative coevolution for large scale optimization. In *IEEE Congress on Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence)*, pages 1663–1670, June 2008.

[52] X. Li. Publications. http://goanna.cs.rmit.edu.au/~xiaodong/publications/publications.html. Accessed: 13-May-2015.

[53] I. O. Bohachevsky, Johnson M. E., and M. L. Stein. Generalized simulated annealing for function optimization. *Technometrics*, 28:209–217, 1986.

[54] L. T. Wille. Minimum-energy configuration of atomic clusters: new results obtained by simulated annealing. *Chemical Physics Letters*, 133:405–410, 1987.