Computational Intelligence Optimisation

Task Three: Designing a Novel Optimiser Algorithm

Adam Leonard Hubble
P17175774

Abstract – Memetic Algorithms (MA's) are recognised combinatorically, for their Evolutionary Algorithm (EA) and Local Search (LS) operator composites, which as extensions to Genetic Algorithms (GA), purpose to nullify the likelihood of both premature convergence and stagnation, during a solutions navigation and settlement to the global optimum of a given problem function or domain. Thereby within this extract, the design and implementation of a Meme Algorithm (MA) that integrates the operations of the elected Self-Adaptive Differential Evolution (jDE) and Short Distance Exploration (S) algorithms is presented, as well, is the accompanying experimentation led for its performance evaluation and concluding discussions.

Keywords – Memetic Algorithms, Evolutionary Algorithm, Local Search, operator, Genetic Algorithms, premature convergence, stagnation, navigation, settlement, optimum, Meme Algorithm, Self-Adaptive Differential Evolution, Short Distance Exploration, experimentation, performance evaluation

I. Introduction

In the proceeding passages, presents the motivation, design, and implementation of a novel optimiser algorithm, that combines operators of a population-based algorithm and a single Local Search (LS) algorithm; a Memetic Algorithm (MA), or Meme. Alongside said MA, the implementation of both the elected population-based and single-solution (local search) algorithms are addressed, namely, Differential Evolution (DE) [1] and Short Distance Exploration (S) [2]; these algorithms are purposed for comparative sake, as later acknowledged as benchmark algorithms for the novel optimiser designed (the reference algorithm).

In continued mention of benchmarking, each of the three algorithms abovesaid are selected to address the CEC 2014 benchmark suite of optimisation problems [3], in the determination of the novel algorithm's advancement, from its subsequent operators, standalone. Notably, each algorithm and the corresponding benchmark suite configured for the optimisation tasking, share occupancy within the Stochastic Optimisation Software (SOS) platform [4] provided.

Conveniently, the paper is orchestrated into six conventional sections, one of which comprises the introduction. Featured in the proceeding sections, contains discussions regarding the problem definition of the optimisation tasking undertaken, the motivation for the solutions design, the implementation of the solution design motivated, the benchmarking procedure conducted as well with the numerical results obtained, and concluding observations concerning the performance of the novel optimiser and its resultant capability.

II. Problem Definition

With regards to the definition of the problem tasked, the problem can be characterised simply, as the search for real-valued vectors $x \in D$ representing global optimality (global minimum or minima) within multivariate domains $D \subset \mathbb{R}^n$, that can be populated by algorithmically minimising the cost or fitness $f(x)$ of mathematical problem functions $f : D \to C \in \mathbb{R}$ [5].

Facilitating the search, requires the implementation of a Memetic Algorithm that is composited of a single population-based optimiser algorithm and one or more Local Search operators or algorithms, to address the design specification of a proposed novel optimiser. As mentioned previously, the novel optimiser proposed is then expected to be performatively compared to one elected population-based algorithm and one single-solution algorithm, which also require separate implementations, for determining the novel algorithms capability as a minimisation tool for optimisation. Supporting the comparative analysis, the provided test-bed suite for optimisation, CEC 2014, is applicated to benchmark the algorithms over the series of problem functions that it encompasses, for the series of dimensionalities given (*10D*, *50D* and *100D*). Resultantly, from benchmarking the algorithms, series of statistical tests and numerical results are expected to be produced and presented, in the format of a mini-paper.

III. Solution Design

As the No Free Lunch Theorem (NFLT) [6] addresses, the "average performance of any two algorithms on all possible fitness functions is equivalent" [5], which formulates the perceived "need of tailoring an optimiser to the specific problem, where possible, in order to increase the performance". Given this relation, it can be acknowledged that by optimising an algorithm for a specific or collective of applications, it will inevitably perform better than a general-purpose optimiser, however, will opposingly perform worse than said optimiser when subjected to problems that are not performatively optimal. Thereby, in accordance with the problem function diversity sponsored by the CEC 2014 test-bed suite for optimisation [3]: unimodal, simple-multimodal, hybrid and composition function types, the algorithmic design established a prerequisite to cater for a "balance between the capability of exploring the search space, and exploiting potential optima" [5], for yielding an adequate performance for all the problems (increased general focus).

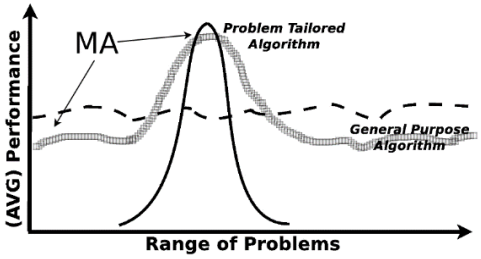


Figure 1: No Free Lunch Theorem visualisation, Memetic Algorithm (MA) focus, increasingly tailored to the range of existing problems, as inferred by performance [5].

Given the expectation of the optimisation tasking, the underlining principle of designing a Memetic Algorithm purposes to fulfil the imbalance by combining an "explorative component", an Evolutionary Algorithm, with an "exploitative local search" algorithm or operator. With the balance adhered to, such that an EA and LS coexist within the same framework and generation loop, the search process can be redirected from "premature convergence (i.e., too fast convergence towards a suboptimal region of the search space) and support the algorithm to continually discover new promising search areas" (overcome stagnation) [7]. This is feasible due to Evolutionary Algorithms operating with populations of individuals or solutions, as population-based optimisers, for which "a diverse population offers benefits at some stages of the evolutionary process" [5], given that a "degree of exploration is kept to avoid premature convergence and to promptly leave local minima". Whereas Local Search algorithms and operators can be utilised "to exploit the most promising basins of attraction, in order to converge within a reasonably small neighbourhood of the global optimum", thus eluding stagnation, as an LS can potentially continue to work "on the same solution until stagnation is overcome".

Proposed as the novel optimisers composition (jDES), features the hybridisation of an Evolutionary Algorithm, namely Self-Adaptive Differential Evolution (jDE) [8] and a single-solution optimiser, namely Short Distance Exploration (S) [2]. Providing the operational contrast mentioned, jDE as a population-based metaheuristic and further DE variant, demonstrates prominent "explorative and exploitative tendencies during the course of search" [9], dependant on the mutation and crossover operators configured for determining its perturbation logic. As an extension of classical DE, the self-adaptation scheme proposed for DE's control parameters "lead to better individuals that in turn, are more likely to survive and produce offspring" when exploring the landscape of a fitness function "as much as possible during an initial phase" [5], which ultimately can then be mutated further via the perturbation logic of the Local Search algorithm, S, to then focus the search "around potential optima"; this scheme is motivated to generate more "fitter individuals" in the population, that provides a "boost in the performance of the algorithm, and in case of DE, also helps prevent stagnation" (see ***Appendix A***).

Memetic Algorithm (MA) Pseudocode

Initialise *Generate an initial population*
while *Stopping conditions are not satisfied* **do**
  Evaluate *all individuals in the population*
  Select *the subset of individuals, that should undergo the individual improvement procedure*
  Variation *mutate and then crossover the target and mutant individuals to create offspring*
  Local Search
  for *each of f srping individual* **do**
    Perform *individual learning using meme(s) with frequency or probability of* f_{il}*, for a period of* t_{il}
    Proceed *with Lamarckian or Baldwinian lifetime learning*
  end for
end while

Respectively, jitter-DE was elected as the evolutionary component for the proposed MA, given its renowned performance advancements from the classical DE framework, for which it achieved the "first rank in the Competition on Evolutionary Computation in Dynamic and Uncertain Environments in CEC2009", whilst being a DE variant, that when "compared to most other

EAs, is much more simple and straightforward to implement" [9]. jDE as a self-adaptive variant, differentiates from the classical DE framework by embedding a self-adaptive control mechanism, that is used to simply "change the control parameters *F* and *CR* during the run" [10], for discovering combinations of the parameters that lead to the population of fitter individuals; this is understood as "each individual in the population is associated with its personal parameters" ($< x, F, CR >$) [2]. Self-adaptive parameter control is formally considered an "evolution of the evolution" [8], in which the parameters to be adapted are "encoded into the chromosome (individuals) and undergo the actions of genetic operators". Given that the parameters "*F* and *CR* are transmitted into the new generation" of individuals, the parameters are regarded as evolving, which is recognised as "tuning on the fly" [5], as both the scaling factor (*F*) and crossover rate (*CR*) parameters of its operators share a small probability ($\tau_1 = \tau_2 = 0.1$) [10] of being resampled (controlled randomisation), before "the mutation operation is performed" on each individual in the population, throughout the evolutionary process. Notably, jDE "does not increase significantly the complexity or the computational effort in comparison to the standard DE" [11], and as previous studies have acknowledged, "outperforms the classic DE/rand/1/bin scheme, among other schemes also", of the classical framework. Despite the simplicity of jDE, it has been proven to be "competitive with modern and complex optimisers, both on benchmark problems and machine learning applications" [2], for which is why its nomination for the EA of the proposed MA, was determined suitable (see ***Appendix B***).

Self-Adaptive Differential Evolution (jDE) Algorithm Pseudocode

$\boldsymbol{g} \leftarrow 1$ (first generation)
$\boldsymbol{Pop^g} \leftarrow$ *randomly sample M, n-dimensional individuals within D*
$\boldsymbol{x_{best}} \leftarrow$ *fittest individual* $\in \boldsymbol{Pop^g}$
while *Condition on budget* **do**
  for each $\boldsymbol{x_j} \in \boldsymbol{Pop^g}$ **do** ($j = 0, 1, 2, ..., M$)
    Resample scaling factor $\boldsymbol{F}$ of the mutation operator
    $\boldsymbol{F_{i,\ g+1}} \leftarrow \begin{cases} F_l + rand * F_u & if\ rand < probability\ \tau_1 \\ F_{i,\ g} & else \end{cases}$
    $\boldsymbol{x_m} \leftarrow$ *Mutation* (create the mutant vector)
    Resample crossover rate $\boldsymbol{CR}$ of the crossover operator
    $\boldsymbol{CR_{i,\ g+1}} \leftarrow \begin{cases} rand & if\ rand < probability\ \tau_2 \\ CR_{i,\ g} & else \end{cases}$
    $\boldsymbol{x_{off}} \leftarrow$ CrossOver ($\boldsymbol{x_j}, \boldsymbol{x_m}$) (generate an offspring)
    if $f\ (\boldsymbol{x_{off}}) \leq f\ (\boldsymbol{x_j})$ **then** (survivor selection)
      $\boldsymbol{Pop^{g+1}}[j] \leftarrow \boldsymbol{x_{off}}$
    else
      $\boldsymbol{Pop^{g+1}}[j] \leftarrow \boldsymbol{x_j}$
    end if
  end for
  $\boldsymbol{g} \leftarrow \boldsymbol{g} + 1$ (update previous population)
  $\boldsymbol{x_{best}} \leftarrow fittest\ individual \in \boldsymbol{Pop^g}$ (update best indiv.)
end while
Output *Best Individual* $\boldsymbol{x_{best}}$

Elected as the LS algorithm of the proposed MA, S, a single-solution deterministic metaheuristic algorithm, was nominated in response to its prior study and perturbation logic that embraces an increasingly exploitative or "greedy-descent" [2] search method in a given problem domain $D \subset \mathbb{R}^n$, overtime; this was assumed sensible for reducing the likelihood of stagnation, given that "the DE framework seems to be particularly prone to stagnation rather than premature convergence" [5]. Operationally, S "executes an asymmetric sequence" of design variable perturbations or gene mutations across all function axes, orthogonally, for the dimensionality of the problem domain n; this "exploration move attempts to fully exploit promising search directions" [13] in the domain, for seeking the basin of attraction that is "globally optimal", to exploit potential optima [5] that yield fitter individuals. As the search or perturbation logic of S is recognised as being "particularly efficient in dealing with separable and LSOP" [2], it was anticipated that the fitness of the individuals in the population would evolve further, "since the performance of basic DE schemes also degrade with massive increase in problem dimensions" [9]; thus, determining DE impractical for handling large-scale optimisation problems (LSOP), as "important attempts have been made by the researchers to make DE suitable for handling such" (see ***Appendix C***).

Short Distance Exploration (S) Algorithm Pseudocode

$\boldsymbol{x_s} = \boldsymbol{x_{best}} \leftarrow initial\ guess \in \mathbb{R}^n$
$\boldsymbol{\delta} \leftarrow \alpha(x^U - x^L)$ (x^U/x^L: upper and lower bounds of $\mathbb{R}^n$)
while *Condition on budget* **do**
  for $i = 1 : n$ **do**
    $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i] - \delta[i]$
    if $f\ (\boldsymbol{x_s}) \leq f\ (\boldsymbol{x_{best}})$ **then**
      $\boldsymbol{x_{best}}[i] \leftarrow \boldsymbol{x_s}[i]$
    else
      $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i]$
      $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i] + \frac{\delta}{2}$
      if $f\ (\boldsymbol{x_s}) \leq f\ (\boldsymbol{x_{best}})$ **then**
        $\boldsymbol{x_{best}}[i] \leftarrow \boldsymbol{x_s}[i]$
      else
        $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i]$
      end if
    end if
  end for
  if $f\ (\boldsymbol{x_{best}})$ *has never improved* **then**
    $\boldsymbol{\delta} \leftarrow \frac{\delta}{2}$
  end if
end while
Output $\boldsymbol{x_{best}}$

With correspondence to the activation of the LS "within the evolutionary cycle" [5], it was sought for S to mutate the genes of the elitist offspring of the DE population of individuals, to further evolve the fitness of the elite and guarantee fitter individuals in the "generational replacement" [1] scheme, via exploitation. This was motivated by the limited, computational budget $5000 * n$, in which executing the LS for the worst or every offspring would exhaust quickly, given the existence of the individual(s) within a sub-optimal basin of attraction, that requires an explorative capacity to escape before exploiting a new promising area found. Also, as the algorithms share the computational budget, a localised termination condition for S was considered relative to the balance of exploration and exploitation, for yielding adequate performance for all problems in the test-bed suite; this criterion takes upon the formality of an iterative budget, representing the iterative capacity of the operator's procedure.

For comparative measures, the classical DE framework is implemented as a separate population-based optimiser, providing its acknowledged simplicity to implement and tune, given the "number of control parameters in DE is very few" [9], as well as being recognised to be "largely better than the PSO variants over a wide variety of problems". As previously mentioned, DE does not support a self-adaptive parametric mechanism unlike jDE but equivalently "optimizes a problem by maintaining a population of candidate solutions and creating new candidate solutions (individuals) by combining existing ones" [11] using mutative and combinative strategies, and "keeping whichever candidate solution has the best objective value" in the corresponding generation, the fitter individual(s) (elitism). Given this relation, an improved or fitter individual is accepted into the population via the "1-to-1 spawning scheme" [5] adopted from Swarm Intelligence (SI), as the survivor selection mechanism of DE selects "which vectors from the parent and offspring populations will survive to the next generation" [9], otherwise the individual is discarded; this recurs "until a termination criterion is satisfied", which is when the computational budget expires. As an EA, the fitness of the population of individuals evermore evolves in its future generations (see ***Appendix B***).

Differential Evolution (DE) Algorithm Pseudocode

$\boldsymbol{g} \leftarrow 1$ (first generation)
$\boldsymbol{Pop^g} \leftarrow$ *randomly sample M, n-dimensional individuals within D*
$\boldsymbol{x_{best}} \leftarrow$ *fittest individual* $\in \boldsymbol{Pop^g}$
while *Condition on budget* **do**
  for each $\boldsymbol{x_j} \in \boldsymbol{Pop^g}$ **do** ($j = 0, 1, 2, \ldots, M$)
    $\boldsymbol{x_m} \leftarrow$ *Mutation* (create the mutant vector)
    $\boldsymbol{x_{off}} \leftarrow$ CrossOver $(\boldsymbol{x_j}, \boldsymbol{x_m})$ (generate an offspring)
    if $f(\boldsymbol{x_{off}}) \leq f(\boldsymbol{x_j})$ **then** (survivor selection)
      $\boldsymbol{Pop^{g+1}}[j] \leftarrow \boldsymbol{x_{off}}$
    else
      $\boldsymbol{Pop^{g+1}}[j] \leftarrow \boldsymbol{x_j}$
    end if
  end for
  $\boldsymbol{g} \leftarrow \boldsymbol{g} + 1$ (update previous population)
  $\boldsymbol{x_{best}} \leftarrow fittest\ individual \in \boldsymbol{Pop^g}$ (update best indiv.)
end while
Output *Best Individual* $\boldsymbol{x_{best}}$

Moreover, as the elected single-solution optimiser, the S operator utilised by the proposed MA is also implemented as an independent algorithm, providing its readily available state, as well, acknowledging its ability to compete with DE in LSOP's, where DE is recognised as being worse, performatively [9]; this is problematically "two-fold", given that the "complexity of the problem usually increases with the size of problem" and the "solution space of the problem increases exponentially with the problem size", whilst the mutation and crossover schemes of DE degrade with increasing problem dimensions also.

III.I. Initialisation and Variation

Traditionally, is it known by EA's to "resort to simply generating $|pop|$ solutions at random" [7] via pseudo-random functions, meanwhile, it is "typical for MAs to attempt to use high-quality solutions as a starting point", from using a more intelligent mechanism, like that of a "constructive heuristic" that can "inject good solutions into the initial population". This purposes to increase the average fitness of the individuals in the population, although their diversity depreciates, which poses constraints on an MA's explorative capabilities, initially. Given this relation and providing the function diversity advertised by the CEC 2014 test-bed suite, it was time-optimal and suitable to apply jDE's pseudo-random population initialisation technique, given its availability and encouragement for initial diversity and thus a resultant, explorative search. Which entails sampling the individuals or parents of the initial population, "randomly with uniform distribution" [8] from the search space of the given problem domain $D \subset \mathbb{R}^n$, for the number of members configured for the population *NP* (population size); simply, a pseudo-random function that promotes the diversity of individuals in the initial population, to "ensure a good coverage of the search space" is achieved (see ***Appendix A***).

Fundamental to the variation operators of the MA, jDE's mutation and crossover schemes are used to "generate new vectors (trial vectors),

where the selection then determines which of the vectors will survive into the next generation" [8]. In focus of mutative operator proposed for the explorative intent of jDE within the MA, mutation can be regarded as the "linear combination of individuals" [2] amongst the population, in which three "random individuals" x_{r1}, x_{r2} and x_{r3} are sampled from the population, such that each is unique $r_3 \neq r_2 \neq r_1 \neq j (j = 1, 2, \ldots, M)$, that when scaled by a factor *F,* can produce the mutant vector x_m representing the "summation of two components: x_{r1} and a difference vector $(x_{r2} - x_{r3})$, whose aim is to move x_{r1} towards the optimum" position in the decision space. Derived from the "differential" [9] operation of the difference vector, the optimiser attains the name of Differential Evolution; given the linear combination of individuals mentioned, the approach "guarantees large moves in the early stages of the optimisation, becoming smaller and smaller, and so more precise around the optimum, as soon as the population converge" [2]. This constitutes to the explorative focus of the MA, which prepares the individuals of the population for exploitation (LS).

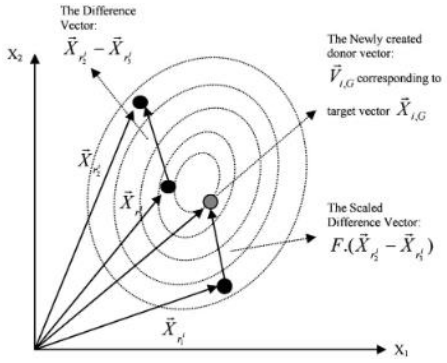


Figure 2: Visualisation of a simple Differential Evolution (DE) mutation scheme, within a two-dimensional (2D) parametric space [9].

In consideration of the mutation scheme nominated for both jDE and DE, the explorative behaviour intended for the initial search of the individuals encouraged the "classic DE (DE/rand/1)" [9] mutation scheme, given that it is regarded "as simple as it is efficient" [2], whilst it "provides with an exploratory search". Despite the "several alternative mutation formulas that have been proposed by researchers in the field" [5], the "standard and robust" [1] operator enables individuals to be mutated at no additional computational expense. Also, when referring to the range of minimisation problems featured within the CEC 2014 test-bed suite, any single mutation operator cannot handle all problem types providing their "set of different moves" [2], therefore, for the simplicity and explorative intent of the EA, the classical operator was deemed suitable for application (see ***Appendix A and B***).

DE/rand/1 Mutation Operator Pseudocode

procedure mutation($\boldsymbol{Pop}, \boldsymbol{F}$)
  for *i = 1 : NP* **do**
    $\boldsymbol{r_p} \leftarrow$ *i* (random permutation)
  end for
  $\boldsymbol{r_p} \leftarrow$ *randomPermutation*($\boldsymbol{r_p}$)
  $\boldsymbol{r_1} \leftarrow \boldsymbol{r_p}[j]$ (first random point, *j = 0*)
  $\boldsymbol{r_2} \leftarrow \boldsymbol{r_p}[j]$ (second random point, *j = 1*)
  $\boldsymbol{r_3} \leftarrow \boldsymbol{r_p}[j]$ (third random point, *j = 2*)
  for *i = 1 : n* **do**
    $\boldsymbol{x_m}[i] = \boldsymbol{x_{r1}}[i] + \boldsymbol{F}(\boldsymbol{x_{r2}}[i] - \boldsymbol{x_{r3}}[i])$ (mutant vector)
  end for
  Output $\boldsymbol{x_m}$
end procedure

In correspondence to the combinative operator proposed for the utilisation of jDE within the MA, the crossover operation aims to "create new promising candidate solutions by blending existing solutions" [7] and thus enhancing the "potential diversity of the population" [9]; this potentializes leading the search of the optimisation process to "areas where better solutions may be found" [7], for ultimately increasing the average fitness of the individuals contained within the population. Given this relation, crossover can be acknowledged as a gene adoption method, in which the trial solution $U_{i,G} = [u_{1,i,G}, u_{2,i,G}, u_{3,i,G}, \ldots, u_{D,i,G}]$ is produced from the mutant vector x_m exchanging "its components with the target vector $x_{i,G}$" [9]; simply, the original and mutant individuals (vectors) exchange "design variables according to a given probability *CR*" (crossover rate) [5]. In case of the solution developing an existence outside of the problem domain, it can be corrected back into the space, toroidally, where it can reclaim its feasibility as a valid solution for a given problem. Like in Evolution Strategy (ES), there is no selection pressure for the choice of the parents undergoing variation; "DE's weak selective pressure" [9] derives from its unbiased, acting parent selection scheme, in which the parents are the original and mutant individuals.

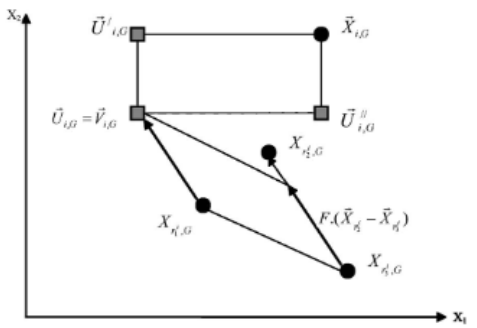


Figure 3: Visualisation of the different possible trail vectors formed due to the crossover operation between the target and mutant individuals (vectors), within a two-dimensional (2D) parametric space [9].

In recognition of the available crossover operators, the "DE family of algorithms can use two kinds of crossover methods", that are: exponential (two-point modulo) or binomial (uniform). As the "most common choices for DE" [5], it was deemed sensible to elect the operator with the highest probability of producing diverse individuals, for adhering to the explorative design intent of the EA. Given this relation, the binomial crossover strategy was selected, as per name, the "number of parameters inherited from the donor has a (nearly) binomial distribution" [9], compared to the "burst of consecutive design variables" [5] that the exponential operator offers, as the "number of components the donor vector actually contributes to the target vector" [9]. Arguably, the implementation of the binomial variant of the crossover operators is comparatively simpler and less computationally expensive to fulfil also (see ***Appendix A and B***).

Binary Crossover Operator Pseudocode

procedure crossover$(\boldsymbol{x_1}, \boldsymbol{x_2})$
  Index ← rand int $U[1, n]$ (uniformly distributed integer)
  for $i = 1 : n$ **do**
    if $U(0, 1) \leq$ ***CR*** || $i ==$ ***Index*** **then**
      $\boldsymbol{x_{off}}[i] \leftarrow \boldsymbol{x_1}[i]$ (mutant individual gene)
    else
      $\boldsymbol{x_{off}}[i] \leftarrow \boldsymbol{x_2}[i]$ (original individual gene)
    end if
  end for
  Output $\boldsymbol{x_{off}}$
end procedure

Collectively, the rand/1 mutation and binomial crossover operators formulate the "basic" [2] DE/rand/1/bin framework of jDE; this is given by the "DE/$x/y/z$ notation" [5], where x refers to the vector being mutated, y represents the number of employed difference vectors, and z signifies the crossover strategy. Particularly, the DE/rand/1/bin variant is regarded "successful in optimizing the multimodal and separable benchmarks" [9], which the CEC 2014 test-bed suite features [3].

III.II. Local Search

Local Search functions offer a localised improvement of offspring, where the "goal of local improvement is to improve the quality of an offspring as far as possible" [7], proceeding the crossover operation between the target and mutant individuals. This is achieved as the search becomes "focused around potential optima" [5], for which the computational budget is exhausted more to "exploit the most promising basins of attraction, in order to converge within a reasonably small neighbourhood of the global optimum"; in acquirement of said focus, a LS typically employs a unique perturbation logic that enables a solutions (individuals) design variables (genes) to be perturbed (mutated) "along the coordinate axes" of a given problem domain $D \subset \mathbb{R}^n$. The perturbation logic that an LS complies with can either be "deterministically worked out, or randomly generated" (stochastic), which poses implications on a given optimisers performance, situationally. Whereby, "stochastic mechanisms tend to outperform deterministic ones over noisy landscapes", as they are more explorative by nature, which presents a "better chance to get out of local minima" entrapment, while gradient-based mechanisms (deterministic) "get stuck" and are resultingly "less adequate for exploring multiple optima" but typically "converge faster" and are instead recognised for "exploiting and refining a point quite accurately". Thus, it can be concluded that a deterministic metaheuristic generally caters for exploitative, minimisation behaviours better, upon an individual being introduced to the LS readily within an optima basin of attraction, through its budget-friendly, refinement process.

Property	Description
Order	***Zero*** *– uses function directly (direct search)* ***One*** *– uses first-order derivative* ***Two*** *– uses second-order derivative*
Pivot Rule	***Steepest Descent*** *– explore all possible moves before selecting a new base point* ***Greedy Descent*** *– explore the first better found search direction*
Depth	***Stop criterion*** *– define the termination condition for the outer loop (procedure)*
Neighbourhood Generating Function	$\boldsymbol{\varphi(i)}$ *– defines a set of points that can be reached by the application of some move operator to the point* ***i*** *(within* ***i's*** *vicinity)*

Table 1: Local Search (LS) function characteristic overview.

Given this prospect, S, the LS operator, was deemed sensible as the LS operation of the MA, as it is a recognised "deterministic logic" and further "greedy-descent" [2] search method, which expresses its perturbation logic as the exploration of the search direction that detects an improvement in solution fitness first, in accordance with the Pivot Rule characteristic of LS functions (see ***Table 1***). This design consideration was sought to be budget-friendly, which proposed jDE to operate with a larger explorative capacity for the initial search and to overcome the noisy landscapes and there contained local minima, that a

deterministic metaheuristic renders challenging. As "derived" from Hooke-Jeeves Method proposed in 1961 [12], the operator's perturbation logic can be understood as an asymmetric sequence [5] of gene mutations (design variable perturbations) that undertake an orthogonal development, across all function axes for the dimensionality *D* of a given problem domain n. For each dimension in a problem domain presented to the operator, S perturbs the design variable (or gene) of the offspring solution provided by jDE's crossover operation, along its corresponding axis for a full step of its exploratory radius $\delta[i]$; if the offspring's fitness does not improve (not detected), then a "half step is performed on the opposite direction" [2] of the equivalent axis, in attempt to identify a more promising search direction. If no improvement is registered from any gene mutation (perturbation) performed, the offspring's genes are then restored to their pre-perturbation state and the exploratory radius that each gene is associated with is then minimized $\delta = \frac{\delta}{2}$, for the purpose of reducing the step size and the resultant explorative potential of the operators search; this enables the algorithm to progressively exploit more promising areas surrounding the offspring solution, overtime. In continued mention of this relation, as a metaheuristic, the exploratory radius of the operator's search is recommended to be relatively small for it to be considered a LS, given its global-search potential; as proposed for the operator's stochastic variant: Stochastic Short Distance Exploration, a sensible radius could conform to a "width equal to 20% of the width of *D*", the problem domain.

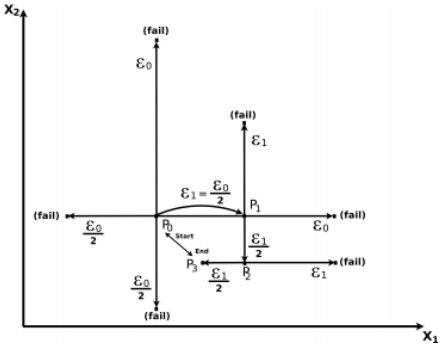


Figure 4: Visualisation of the Short Distance Exploration (S) operator perturbation cycling for a two-dimensional problem domain [2], and a given offspring solution when the initial exploratory radius ε, is set to ε_0. Where P_0 represents the offspring solution pre-perturbation (start), P_3 represents the offspring solution converged (end), P_1 represents the first successful perturbation operation or gene mutation (improvement), and $(fail)$ represents an unsuccessful perturbation or gene mutation operation (no improvement).

As previously detailed, the operation recurs until the procedures computational budget is met (expires); hereby, the solutions optimality is sensitive to the budget, which poses constraints on the search and resultant basin of attraction in the fitness landscape that the operator converges to. If insignificant in value, the likelihood of the operator converging to a local optimum heightens, whereas if more significant in value, the operator will more likely converge to an increasingly optimal basin in the search space but exhaust the budget that is also shared by the EA, quicker; this would be detrimental to the MA's explorative potential. Given the gene (individual design variable) focus of the operator's perturbation logic (mutation) as opposed to the offspring solution (all genes), independently, S can "be particularly promising on separable and large-scale problems" and where there is a "presence of stagnation" (DE); conveniently the CEC 2014 test-bed suite addresses such problem types, where it is expected to perform well.

III.III. Lifetime Learning

Lifetime learning concerns the appliance of the LS operator on the offspring of a given EA, at the time of its generation; this scheme purposes to feature more performing individuals in the next generation of the corresponding population, for enhancing the prospect of converging to the optimum in the domain $D \subset \mathbb{R}^n$. Given the prior definition of an MA, MA's can be modelled and "referred to as Lamarckian-EAs and Baldwinian-EAs" [5], which represent two succinct approaches to "how the corrected solutions should be used" [13] by their subsequent LS operator(s) in the evolutionary process. As traditional models of lifetime learning, Lamarckian-EAs apply local searchers to "improve upon the genotype of some individuals" [5] by "driving the evolution towards promising points" [2] in the search space, while in Baldwinian-EAs "only the phenotype gets modified", where the offspring receives the fitness of a fitter neighbour individual but not its genotype; in light of this relation, an LS can be recognised as an independent mutation operator, that finalises the search by navigating within the vicinity of near-optimum solutions discovered by the EA.

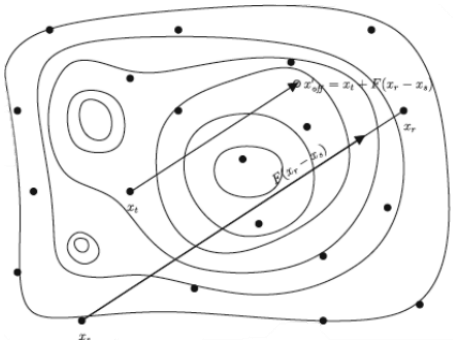


Figure 5: Visualization of Differential Evolution (DE) algorithms mutation operator, displaying high individual diversity in the initial search of the decision space.

Providing the one-to-one spawning scheme proposed as the survivor selection mechanism of jDE [5], traits acquired by an individual during its lifetime can be transmitted to its offspring in the next generation (end of generation loop), via its replacement with a fitter neighbour; this enables the population size to remain constant whilst the fitness of its contained individual's advances. When featuring the replacement scheme at the end of the generation loop, parallel implementations become compatible, which enables multiple LS operators to operate on the offspring individual before the population member is updated. Overtime, the diversity of the individuals in the population marginalises, as the search becomes increasingly exploitative from being focused around a found optima; hence the fitness of each individual gradually improves. Although both methods are advantageous and problematic, the Baldwinian approach to lifetime learning is neglected by the MA due to Lamarckian learning being "reported as a better approach to unimodal optimization problems and in problems in which the environment is static" [13], which situationally, accommodates for the problem types disclosed by the CEC 2014 benchmark suite [3].

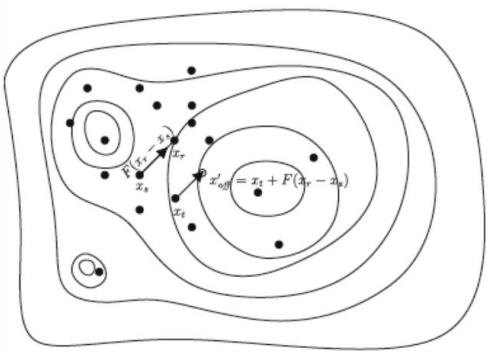


Figure 6: Visualization of Differential Evolution (DE) algorithms mutation operator, displaying low individual diversity in the final search of the decision space.

In exploiting the solutions passed by jDE, as previously mentioned, S is activated for the elitist offspring of the DE population of individuals, only. Providing the finite, $5000 * n$ capacity of the MA's computational budget (fitness evaluations), operating upon the fittest individuals guarantees increasingly fitter individuals in the proceeding generation; this is accepted as the DE scheme in presence of stagnation, "is not able to improve upon individuals" [5] and whilst configured with explorative operators, DE cannot exploit promising basins of attraction but rather near them. Moreover, operating on all offspring or selectively the least-performing individuals exhausts the computational budget whilst not guaranteeing improvements; as previously told, budget exhaustion then hinders the explorative capacity of DE, as a shared resource, which presents broader implications on the average fitness of the population. Where the likelihood of converging to a sub-optimal basin of attraction and optima, increases. In essence, the activation criteria of S is sought to govern and "guarantee a fair balance between exploration and exploitation" [5] behaviours of the MA (jDES), that enables jDES to yield a general-purposed performance for all problem types featured within the CEC 2014 test suite.

III.IV. Parametric Overview

Fundamental to the performative optimality of the MA and the resultant fitness achieved for the individuals comprising the population, "choosing suitable control parameter values is, frequently, a problem-dependent task" [8]. This is inevitable, as "for all evolutionary algorithms, the operational parameters control the balance between exploitation (using the existing material in the population to best effect) and exploration (searching for better genes)" [1], which affects their search locality and resultant capability to minimalize a problem. As well, as they "frequently interact with each other" a general-purpose combination can be "difficult to find". In selecting jDE as the EA of the MA, reduces the configurative pressure of the MA significantly, as the "effectiveness, efficiency, and robustness of the DE algorithm are very sensitive to the values of these control parameters" [11], and jDE as a DE variant, mitigates this pressure by employing "self-adaptive control parameter settings" that encourage good performance over a range of "numerical benchmark problems".

In jDE, parameter control focuses upon the self-adaptation of the "parameters *F* and *CR* of the DE evolutionary process", such that each individual in the population "has its own *F* and *CR*" ($< x, F, CR >$) [5]. This provides a flexible DE that self-adjusts to problems to "achieve the best optimisation outcome"; as DE's third of "three main control parameters" [9], population size *NP*, although a dynamic variable is not concerned in the self-adaptive scheme of jDE. However, it is regarded equally important, given that a large population "is preferable in highly multi-modal problems, but it requires more time to converge" [2], whereas a small population "tends to come together quicker" but increases the likelihood of premature convergence from occurring, although it "can also happen with a high number of individuals". Interactionally, jDE's behaviour is known to be "more sensitive to the choice of *F* than it is to the choice of *CR*" [11], which infers the significance of the EA's mutation operator on the search, when compared to its crossover counterpart. However, the "global optimum searching capability and the convergence speed" [9] of jDE, is "very sensitive" to all the control parameters configurations, respectively.

Alongside jDE, the LS operator S also seeks parametric optimisation, however, unlike jDE, S does not facilitate a self-adjustment mechanism of its parameters and requires manual refinement alternatively. Although simpler, S focuses upon the search radius α in a given decision space, as a neighbourhood generating function that narrows the search "to a small portion of *D*" [5], and its local computational budget, which defines the termination condition and resultant depth of the LS operation. Given that S is applied to an offspring solution to "quickly exploit the most promising search directions and refine the search" [5], its exploratory radius should be relatively small, "proportionate to the range of optimization space bounds"; as previously mentioned, this would enable S to qualify as a LS operator, as it capacitates an explorative potential of a Global Search (GS) operator also. Relating to the operator's computational budget and providing that only the fittest or best (elitist) individuals in the population are operated on within a small neighbourhood, it is deemed sensible for a larger budget to be allocated to the LS operator; this would purpose to maximise exploitation in the basin that the solution is most attracted to in the fitness functions landscape, in attempt to further the fitness of the individual. Simply, the computational budget must be "used to exploit the most promising basins of attraction, in order to converge within a reasonably small neighbourhood of the global optimum"; this is however, relative to the initial, "fixed" exploratory radius of the operator, as a larger, initial search radius would typically require more fitness evaluations before convergence to the optimum could be achieved. Alternatively, a smaller, initial search radius would typically require less fitness evaluations before convergence to the optimum could be achieved; if configured with a larger budget, the LS operator would then be exhausting the resource that it shares with the EA, without an "improvement being registered", as the procedure would continue "until the computational budget allocated expires". Inevitably, the computational budget of the LS operator is impactful on the exploratory capability of jDE.

Below, features the parametric configuration of jDE, the EA, and S, the LS operator of the MA, jDES:

jDES Parameter Configuration		
Algorithm	**Parameter**	**Value**
jDE	NP	10 * D
	F_l	0.1
	F_u	1.0
	τ_1	1.0
	τ_2	1.0
S	α	0.2
	Procedural iterations	100

Table 2: Parametric configuration of the proposed Memetic Algorithm (MA): Self-adaptive Differential Evolution (jDE) Short Distance Exploration (S), jDES.

NP – population size, the size of the pool of "candidate solutions that will survive in the following generations and be used to create new solutions" [7]. As recommended [1], jDE takes upon the "population size of 5–20 times the dimensionality of the problem", specifically, "ten times the dimensionality of the problem at hand" [5], 10 * D. Although a general rule for "choosing the population size does not exist", the dynamicity of the value proposed, maintains scalability with problem complexity; this aims to overcome premature convergence, when a "small population size" is apparent, as well as stagnation and bias in the search, in causation of a "large population" that encourages an exponentially increasing diversity between the populations members (individuals). Given said scalability, the number of generations produced for a given problem and dimensionality remains the same. In correspondence to the computational budget $5000 * n$ (*FE*), all problem functions are handled by five-hundred generations of individuals $\frac{FE}{10 \cdot D}$; this is irrespective of the dimensionally *D*.

F_l and F_u – "lower and upper limits of *F*" [9], representing the sample space $U[0,1]$ for randomising the scaling factor of jDE's rand/1 mutation operator, as part of the employed, self-adaptive parametric scheme. Per recommendation [2][8], the lower bound of the sample space adopts the value of '0.1' and the upper bound takes upon the value of '1.0'; this enables the scaling factor of said mutation operator to take a "value from [0.1, 1.0] in a random manner" (uniformly distributed), $F = F_l + U(0,1) \cdot F_u$. Notably, a "set of *F* and *CR* values are assigned to each individual in the population", which are sampled before the "mutation operation is performed" [10] and are further "refreshed after a certain amount of functional calls according to a given probability τ_1, for scale factor, and τ_2 for crossover rate" [2]. Acknowledged as a configuration that would "lead to good results", both threshold values adopt a ten-percent chance (probability) $\tau_1 = \tau_2 = 0.1$ to "adjust factors *F* and *CR*" [8] during optimisation, prior to gene mutation commencing for the "donor" or mutant individual. However, it should be known that jDE performances are "less sensitive to variations of τ_1 and τ_2", meaning that their refinement is less significant on the problem minimisation process; this accredits their absence from the self-adaptation scheme. Accordingly, improved values for the control parameters "lead to better individuals that in turn, are more likely to survive and produce offspring and, thus, propagate these better parameter values" [9], such that increasingly fitter individuals are produced, generationally.

α – alpha-cut (exploratory radius), the explorative potential of the LS operator, S, representing "a hypercube centred on the elite, whose edge has a width equal to the 20% of the width of *D*" [5]. As recommended for Global Search operations [2], the exploratory radius is typically "set to 40% of the search space size (basically it covers the entire search space)"; providing the exploitative, LS assignment of S in the MA, an exploratory radius of twenty-percent of the size of the search space is configured, for the initial search to be significantly more localised and exploitative, "to converge within a reasonably small neighbourhood of the global optimum" [5]. Moreover, in directing the search "around potential optima", preserves the computational budget for jDE's explorative operations; this is deemed sensible as jDE can then further explore and navigate the individuals towards the optimum basin of attraction, that S can then exploit to refine.

Procedural Iterations – the computational budget of the LS operator, S, representing the number of iterations that the procedure of its operations executes for; this defines the depth of the search, locally, and subsequently, the operator's termination condition. Per recommendation [2], "150 iterations of the procedure are sufficient to reach a good precision", when the exploratory radius of the operator α is configured as forty-percent of the width of the search space *D,* for a given problem domain. Providing that this configuration employs an eighty-percent coverage (diameter) of the search space, as a GS operation, more iterations are required to converge within a reasonably small neighbourhood of the global optimum [5]. Therefore, providing the proposed locality of the search that concerns a forty-percent coverage (diameter) of the search space, fewer iterations are required to converge within a reasonably small neighbourhood of the global optimum; by configuring the operator to execute for one-hundred iterations, enables S to "quickly exploit the most promising search directions and refine the search" of the offspring, whilst reducing unnecessary exhaustion of the global computational budget, that jDE shares.

Self-Adaptive Differential Evolution Short Distance Exploration (jDES) Algorithm Pseudocode

$\boldsymbol{g} \leftarrow 1$ (first generation)
$\boldsymbol{Pop^g} \leftarrow$ *randomly sample M, n-dimensional individuals within D*
$\boldsymbol{x_{best}} \leftarrow$ *fittest individual* $\in \boldsymbol{Pop^g}$
while *Condition on budget* **do**
  for each $\boldsymbol{x_j} \in \boldsymbol{Pop^g}$ **do** ($j = 0, 1, 2, \ldots, M$)
    Resample scaling factor $\boldsymbol{F}$ of the mutation operator
    $\boldsymbol{F_{i,\ g+1}} \leftarrow \begin{cases} F_l + rand * F_u & if\ rand < probability\ \tau_1 \\ F_{i,\ g} & else \end{cases}$

    $\boldsymbol{x_m} \leftarrow$ *Mutation* (create the mutant vector)
    Resample crossover rate $\boldsymbol{CR}$ of the crossover operator
    $\boldsymbol{CR_{i,\ g+1}} \leftarrow \begin{cases} rand & if\ rand < probability\ \tau_2 \\ CR_{i,\ g} & else \end{cases}$

    $\boldsymbol{x_{off}} \leftarrow$ CrossOver $(\boldsymbol{x_j}, \boldsymbol{x_m})$ (generate an offspring)
    if $f(\boldsymbol{x_{off}}) \leq f(\boldsymbol{x_j})$ **then** (survivor selection)
      if $f(\boldsymbol{x_{off}}) \leq f(\boldsymbol{x_{best}})$ **then** (LS activation)
        $\boldsymbol{x_s} = \boldsymbol{x_{best}} \leftarrow initial\ guess \in \mathbb{R}^n$
        $\boldsymbol{\delta} \leftarrow \alpha(x^U - x^L)$ (x^U/x^L: upper and lower bounds of $\mathbb{R}^n$)
        for $l = 1 : procedural\ iterations$ **do**
          for $i = 1 : n$ **do**
            $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i] - \delta[i]$
            if $f(\boldsymbol{x_s}) \leq f(\boldsymbol{x_{best}})$ **then**
              $\boldsymbol{x_{best}}[i] \leftarrow \boldsymbol{x_s}[i]$
            else
              $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i]$
              $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i] + \frac{\delta}{2}$
              if $f(\boldsymbol{x_s}) \leq f(\boldsymbol{x_{best}})$ **then**
                $\boldsymbol{x_{best}}[i] \leftarrow \boldsymbol{x_s}[i]$
              else
                $\boldsymbol{x_s}[i] \leftarrow \boldsymbol{x_{best}}[i]$
              end if
            end if

```
            end for
            if f (x_best) has never improved then
              δ ← δ/2
            else
              Pop^{g+1}[j] ← x_best (update individual)
            end if
          end for
        end if
      else
        Pop^{g+1}[j] ← x_j
      end if
    end for
    g ← g + 1 (update previous population)
    x_best ← fittest individual ∈ Pop^g (update best indiv.)
  end while
Output Best Individual x_best
```

IV. Solution Implementation

IV.I. Algorithm

Given the Stochastic Optimisation Software (SOS) platform [4] provided, the framework of the jDES algorithm was implemented as a new Java class, 'jDES', within a new source code file, 'jDES.java', located within the 'algorithms' module of the software platform. Similarly, both the population-based (DE) and single-solution (S) optimisers are implemented in the corresponding format also. Specific to the jDES algorithm, three methods are implemented to abstract the algorithmic procedure from its affiliate, mutation, and crossover operations; this "helps encapsulate" [14] the behaviours of the algorithm, in support of potentially using "more self-contained modules" or operator strategies (see ***Appendix A***).

Self-Adaptive Differential Evolution Short Distance Exploration (jDES) Algorithm Java Class

Class jDES
Method 1: $execute()$ (procedural loop)
Function call → $originalMutation()$
Function call → $binomialCrossover()$
Method 2: $originalMutation()$ (mutation stratgey)
Method 3: $binomialCrossover()$ (crossover startegy)

Assisting the mathematical operations constituting to both the mutation and crossover operators of jDE and the MA, jDES, a series of prebuilt methods declared within the platforms random utility module, 'RandUtils', are utilised for the performance of random gene permutations and random numeric generation, of integral and floating-point types; this was convenient for conserving time and for also maintaining the robustness (readability) of the platforms code base.

Random Utility Method Classification	
Utility	**Description**
randomPermutation	**Return**: static int[] **Operation**: Permute the order of elements contained within an array variable
randomInteger	**Return**: static int **Operation**: Randomly sample an integer value from the range of $[0, x]$
random	**Return**: static double **Operation**: Randomly sample a uniform, floating-point value from the range of [0, 1]

Table 3: 'RandUtils' class, utility method classification. Methods adopted by the Self-adaptive Differential Evolution (jDE) and the Self-adaptive Differential Evolution Short Distance Exploration (jDES) algorithms.

IV.II. Experimental Setup

Purposed as the test-bed suite for optimisation within the SOS platform, the "minimisation" [3] problem set CEC 2014, as instructed, is utilised to exercise, and compare the performance of the memetic, population-based, and single-solution optimisers for the sponsored problem domains, in ten, fifty and one-hundred dimensions $D \subset \mathbb{R}^n$ (see ***Appendix D***), and for thirty experimental runs, each. Formulating the benchmark, are four classifications of problem: three unimodal, thirteen simple-multimodal, six hybrid-one and eight compositional problem types; all functions are "treated as black-box problems" that are "defined" as: $\mathrm{Min}\, f(x), x = [x_1, x_2, \dots, x_D]^{\mathrm{T}}$, and collectively accrue a thirty-problem series that can ascertain the MA's generality, in accordance with the NFLT [6] (see ***Appendix E***). Furthermore, each of the test functions contained in the benchmark are scalable and shifted to o, representing the shifted global optimum $o_{i1} = [o_{i1}, o_{i2}, \dots, o_{iD}]^{\mathrm{T}}$, which is "randomly distributed in $[-80, 80]^{\mathrm{D}}$", as opposed to the search space $[-100, 100]^{\mathrm{D}}$ configured for every problem.

CEC 2014 Problem Function Classification	
Problem Type	**Properties**
Unimodal	Unimodal Non-separable Rotated
Simple Multimodal	Multimodal Non-separable and separable Shifted Shifted and Rotated
Hybrid 1	Multimodal or unimodal Non-separable subcomponents
Composition	Multimodal Non-separable Rotated Asymmetrical

Figure 7: CEC 2014 test-bed suite, minimisation problem types and their properties, overview [3].

Unimodal – a function $f(x)$ that presents a distribution "with one clear peak" [15] only, within its fitness landscape; also renowned as the global optimum x^*. Given the interval $a \leq x \leq b$, a function is considered unimodal "if and only if it is monotonic on either side of the single optimal point x^* in the interval" [16]. Thus, $x^* \leq x_1 \leq x_2$ implies that $f(x^*) \leq f(x_1) \leq f(x_2)$ and $x^* \geq x_1 \geq x_2$ implies that $f(x^*) \leq f(x_1) \leq f(x_2)$.

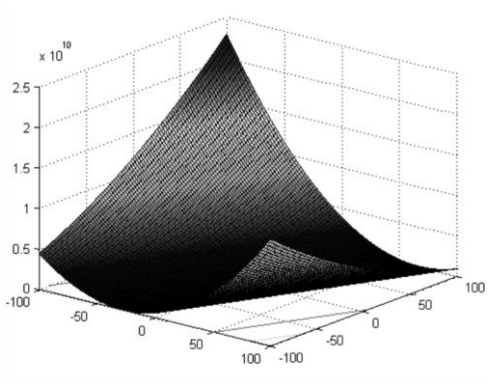


Figure 8: Unimodal problem function, fitness landscape visualisation. CEC 2014 test-bed suite $f_1(x)$, Rotated High Conditioned Elliptic function [3].

Simple-multimodal – a function $f(x)$ that poses a distribution with "two or more modes" [17] (or peaks), within its fitness landscape. In which there are "more than one optimum" [18] and resultingly two extrema points: a global minimum (global optimum) and global maximum (local optimum).

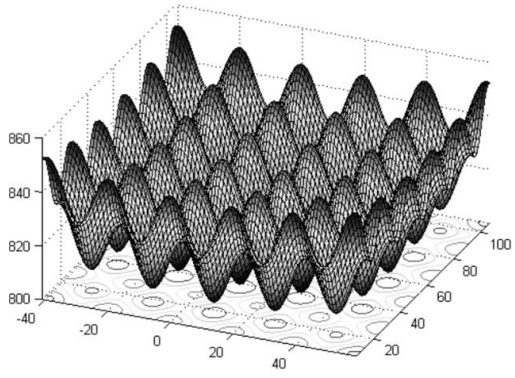


Figure 9: Simple-multimodal problem function, fitness landscape visualisation. CEC 2014 test-bed suite $f_8(x)$, Shifted Rastrigin's function [3].

Hybrid-one – a hybridised function $f(x)$ that proposes a shared, multimodal distribution with two or more peaks within its landscape, dependent on the modality of the number N of "different basic functions that are used for different subcomponents" [3] of the hybrid scheme. Each of the basic functions $g_i(x)$ comprising the scheme is allocated a proportion of landscape control p_i, representing each's contribution to the landscape derived.

Composition – a function $f(x)$ that also presents a shared, multimodal distribution with two or more peaks within its landscape, dependent on the modality of the number N of basic functions comprising the composition. In CEC 2014, the "hybrid functions are also used as the basic functions for composition functions" in which "merges the properties of the sub-functions better and maintains continuity around the global/local optima". As a composition function can be configured with a bias, the global optimum in its fitness landscape can be redefined; the global optimum can also be shifted in position, for each basic function $g_i(x)$ by the value of o_i.

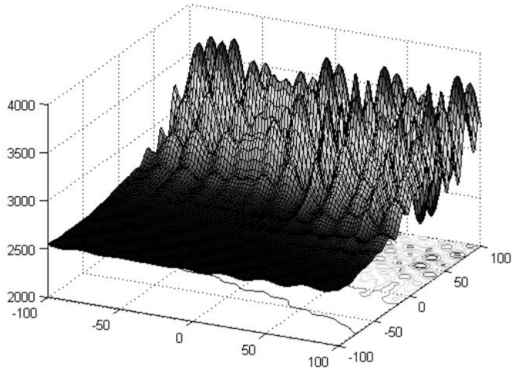


Figure 10: Composition problem function, fitness landscape visualisation. CEC 2014 test-bed suite $f_{24}(x)$, composition: Schwefel's function $f_{10}(x)$, Rotated Rastrigin's function $f_9(x)$, Rotated HGBat function $f_{14}(x)$ [3].

In use of the test-bed suite abovesaid, a class namely 'CEC14', that was readily provided in the source code file 'CEC14.java', located within the 'experiments' module of the SOS software platform, was moderated to facilitate the experimentation of the memetic, population-based, and single-solution optimisers, for the thirty-experimental runs already disclosed. Therein, passes the parametric configuration of jDES as seen within ***Table 2***, as well as the parametric configurations of both DE and S also; DE shares the same configurable state as jDE, whilst S takes upon the GS configuration mentioned prior [2], for comparative measures (see ***Appendix F***). Elements within the 'CEC14.java' file can be commented or uncommented and values altered, to meet the

requirements of either optimisers design specifications; using this interface was considered a time-friendly solution, when compared to statically initialising the values of each algorithm's parameters, separately by their class.

V. Numerical Findings

V.I. Wilcoxon Rank-Sum Test

Purposed as the primary approach to analysing the performance of the memetic algorithm, jDES, across all problems featured within the CEC 2014 test-bed suite for optimisation, the statistic test namely Wilcoxon rank-sum test [2] is elected for conducting "a single hypothesis test (a single optimisation problem is considered) between each pair of algorithms" concerned for comparison, in attempt to identify the outperforming optimiser for each problem and in every dimensionality of the problems domain. As the statistical test is non-parametric and therefore does not "require any specific form or assumption for the distribution of the numeric values", the method is deemed effective in the "presence of outliers or non-Gaussian distributions, thus making the proposed tests more reliable" for the nature of this very application and "suitable for evaluating the performance of algorithms", generically. Particularly, the null-hypothesis between the performative matter of the algorithms is "examined and then accepted if a relationship between data from two different algorithms exist", where the null-hypothesis $H_0: A = B$ is verified upon the elements of two "similar continuous distributions" *A* and *B,* sharing the "same means" (indicated by '$=$'); otherwise, the "two distributions differ" and thus the null-hypothesis $H_0: A = B$ is rejected against the "alternative-hypothesis", which presents to be either $H_1: A \geq B$ (indicated by '$+$') or $H_1: A \leq B$ (indicated by '$-$'), when the "lowest mean" of the two continuous data distributions cannot be patently determined. The alternative-hypothesis can simply be represented as $H_1: A \neq B$.

V.I.I. Unimodal Problems

In focus of the preliminary range of problem functions $f_1(x) - f_3(x)$ within the CEC 2014 test-bed suite, which identify as unimodal problem types, it was observed (see ***Figure 11***) that for an increasing dimensionality of each problem domain, the novel optimiser, jDES, progressively outperforms DE whilst becoming progressively outperformed by the single-solution optimiser, S; per the appointed figure, this correlation is best realised by the trendline features of the visualisation, that linearly depict the MA's pairwise win count (y axis), of the abovementioned statistical test. Given the relationship established between the algorithms, problem functions and an increasing dimensionality of their domains, the depreciating performances of both jDES and DE are assumed to be the cause of an exponentially increasing population size and thus resultant, hastened rate of computational budget exhaustion. This is believed to be due to the scalable adjustment of parameter *NP*, that is configured to the value of $10 * D$, alongside the initial, explorative, and high diversity of individuals comprising their populations, which is undoubtedly excessive for overcoming simple unimodal landscapes. This observation is feasible, as S, the single-solution optimiser, independently outperforms both jDES and DE in higher dimensionalities of each problem domain (see ***Appendix G***), as given by the increasing count of pairwise wins that S amasses. As S only perturbs one individual (no population), it can afford to exhaust its computational budget on an increasingly exploitative search operation amongst the landscape of each problem function, to converge to a more-optimal point in the domain. Whereas jDES and DE operate with vaster explorative perturbation cycles, for a series of individuals constituting a population, which poses computational expense; providing the scalability of both the computational budget and population size available to the optimisers, each individual has the potential to be improved for five-hundred iterations or generations when disregarding the LS operator of jDES, irrespective of the dimensionality of the problem domain. Thereby, for an increasing dimensionality, assumes a gradual performance degradation of population-based optimisers, where individuals become increasingly unfit. Specific to jDES, the number of generations could potentially be confined to a significantly smaller amount, dependant on the activation count of the LS operator, S, on the elitist individuals in the population; given that jDE and S share the same computational budget, means that the budget is exhausted faster than its DE counterpart, hence the significance of balancing exploration and exploitation behaviours of the MA.

Providing that a unimodal landscape nullifies the potentiality of premature convergence from occurring, S is independently expected to be the outperforming candidate for the given problem types, in all dimensionalities, considering its exploitative focus; however, in ten dimensions, this hypothesis is rejected as both jDES and DE on average, converge to more-optimal points in every

problem domain (see ***Appendix G***). Such result is assumed to reflect the smaller and less diverse (tightly distributed) population of individuals, that both optimisers operate with, in which jDES, in cooperation with its LS operator, can further refine to render more optimal than DE, standalone; fewer individuals that occupy a population implies that an optimiser possesses a vaster exploitative capability, given that less perturbation cycles and resultant fitness evaluations are consumed for exploration and improving the fitness of the population members, per generation. Moreover, acknowledging the initial, stochastic generation of individuals within the boundaries of a given problem domain, it is also plausible that jDES and DE were more regularly initialised with fitter individuals, compared to the fitness of the single solution, initialised for S. Given the perturbation logic of S, it is further sensible to insinuate the possibility of S continually overstepping the optimal point x^* in the domain, as its exploratory radius progressively shrinks, from the initial, global span of eighty-percent of the width α of the search space D. Providing that "the performance of basic DE schemes also degrade with massive increase in problem dimensions" [9], as well as S being recognised as "particularly efficient in dealing with separable and LSOP" [2], although the problems are non-separable, the performance of the algorithms presented, reinforces these expectations.

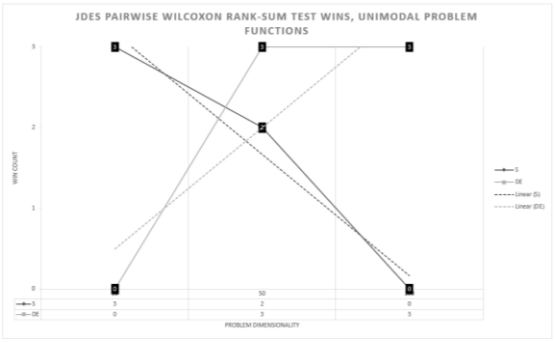


Figure 11: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across the range of unimodal problem functions featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 10D, 50D and 100D, for '30' experimental runs.

Statistically, it is evidenced (see ***Appendix G***) that jDES performs better in lower dimensionalities of non-separable, unimodal problem domains, given by the three (of six) wins it collects in ten dimensions, five (of six) wins that it achieves in fifty dimensions and three (of six) wins that it accumulates in one-hundred dimensions. Per the deviation in the results obtained, jDES is averagely more consistent in converging to more-optimal solutions when compared to both DE and S, as the spread between each experimental run is comparatively, less significant. Moreover, in correspondence with the collective optimality of the solutions produced, for the problem range given, the mean fitness of the solutions attained by jDES is significantly more-optimal in ten dimensions than S but is disputably less-optimal than DE, and is also marginally less-optimal in fifty and one-hundred dimensions than S only; this exemplifies that jDES can outperform and if not, compete with contemporary population-based and single-solution optimisers, for the problem type and domain dimensionality range disclosed.

V.I.II. Simple-Multimodal Problem Functions

Secondly, in focus of the range of problem functions $f_4(x) - f_{16}(x)$ within the CEC 2014 test-bed suite, which alternatively identify as simple-multimodal problem types, it was observed (see ***Figure 12***) that for an increasing dimensionality of each problem domain, the novel optimiser, jDES, once more progressively outperforms DE whilst becoming progressively outperformed by the single-solution optimiser, S; this can be realised from the trendline components of the appointed figure, that linearly depict the MA's pairwise win count (y axis). From this association, the performances of both jDES and DE can again be assumed to be the cause of an exponentially increasing population size, that exhausts the computational budget of their operations, quickly. Given the presence of noisy and some, highly multi-modal fitness landscapes of the focused problem domains, it is unexpected for S to outperform both jDES and DE, providing its deterministic perturbation logic, that is "less adequate for exploring multiple optima" [5] as it is renowned for becoming "stuck" within sub-optimal basins of attraction. Through being sensitive to "noisy landscapes", S is identifiably prone to converging prematurely, however, in accordance with the margin of error calculated between the mean solution fitness and the supposed, optimal fitness of each problem domain (see ***Appendix H***), S for the mentioned range of problems, converges near the optimal point x^* in most of the problem domains in the series, which peculiarly, disproves this behaviours intervention on its search.

Therefore, the performance degradation of jDES can more confidently be assumed a constraint of the computational budget $5000 * n$

on the operatable population size $10 * D$. As within a ten-dimensional configuration of each problem domain, both jDES and DE on average, outperform S, which is evidenced by the pairwise wins that jDES accumulates, in addition to the lower mean fitness and deviation values of jDES and DE, when compared to S (see ***Appendix G***). Moreover, as the basic mutation and crossover schemes of DE and its variants are known to be performatively degrading for an increase in problem dimensionality, it is sought to contribute to the outcome presented also. Otherwise, and although unlikely given the multi-modality of each problem functions fitness landscape, as "the DE framework seems to be particularly prone to stagnation rather than premature convergence", it could also be feasible that either optimiser does not register improvements amongst its search, periodically, in which the fitness of the individuals in the population does not advance but the computational budget continues to be exhausted.

Figure 12: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across the range of simple-multimodal problem functions featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 10D, 50D and 100D, for '30' experimental runs.

In correspondence to the statistical test (see ***Appendix G***), it is inevitable that jDES once again performs better in lower dimensionalities, of both separable and non-separable multimodal problem domains; this is acknowledged by the twenty-five (of twenty-six) wins it claims in ten dimensions, sixteen (of twenty-six) wins that it attains in fifty dimensions and fifteen (of twenty-six) wins that is amasses in one-hundred dimensions. Despite functions $f_8(x)$ and $f_{10}(x)$ being separable problems that S is known to be "particularly efficient in dealing with", across all dimensionalities of their domains, jDES either outperforms or produces solutions similar to it; this is assumed to be in consequence of the high multi-modality of both function's fitness landscapes, that can divert the search of S to a sub-optimal basin of attraction, given its greedy-driven, perturbation logic that relies upon "gradient information to guide the search straight to the closest optimum". When studying the deviation between the results obtained for all three optimisers, across all problem domains and for all dimensionalities, all optimisers are recognised to share a high consistency in their search, for which they all prove to converge to similar points (relative to fitness value) in the domain, for every experimental run. Also, it is worthy to note that for all problem domains and for the range of dimensionalities experimented, the difference in mean fitness achieved by all optimisers is mostly marginal, for which all optimisers are considered competitors for the series of simple-multimodal problems entertained by the benchmark.

V.I.III. Hybrid Problem Functions

Thirdly, in focus of the range of problem functions $f_{17}(x) - f_{22}(x)$ within the CEC 2014 benchmark suite, which otherwise identify as hybrid problem types, it was also observed (see ***Figure 13***) that for an increasing dimensionality of each problem domain, the novel optimiser, jDES, progressively outperforms DE whilst becoming progressively outperformed by the single-solution optimiser, S. As told for both unimodal and simple-multimodal problem types, this can simply be acknowledged by the trendline elements in the appointed figure, which linearly characterises the pairwise win count (y axis) of jDES. Considering the correlation detected, once more, the performances of jDES and DE are assumed to be the result of an exponentially increasing population size, which consequently exhausts the computational budget, quickly. As hybridised functions are aggregates of "basic functions" [3], they assume unimodal or multimodal landscape composites, with non-separable subcomponents, that consequently present asymmetrical distributions of modality in each problem domain. Providing the potentiality of noisy and highly-multimodal features in each function's fitness landscape, jDES and DE as population-based optimisers are expected to yield fitter solutions, on average, given their reserved explorative capabilities to "promptly leave local minima" [5] and thus prevent local entrapment within sub-optimal basins of attraction, unlike S, an increasingly exploitative optimiser.

However, it appears that S unexpectedly outperforms both jDES and DE yet again, for an increasing problem dimensionality (see ***Appendix G***), as is determined by S attaining more pairwise

wins in fifty and one-hundred dimension configurations of the domains, when compared to jDES and DE; this supports the prior assumption of computational budget exhaustion, as S is vulnerable to noisy landscapes, for which its search would most likely be diverted away from the optimum basin of attraction and therein point. Additionally, considering the previously mentioned performance defects of DE's basic mutation and crossover schemes in higher dimensions, performance depreciation is highly anticipated for increasingly complex fitness landscapes, when concerning the performances of jDES and DE for the simple-multimodal problem functions, aforementioned; this too, is assumed to be the resultant performance of computational budget insufficiency, especially given that DE alone, achieved "first rank in the Competition on Evolutionary Computation in Dynamic and Uncertain Environments in CEC2009". Moreover, upon recognising that JDES predominantly outperforms DE across all experiments conducted, it can further be assumed that for an increasing dimensionality, both jDES and DE were more prone to stagnation, in which jDES could offset from, in use of its LS operator; this observation comes to exist as the difference within the mean fitness values of their solutions across all problems and dimensionalities is marginal, in favour of JDES.

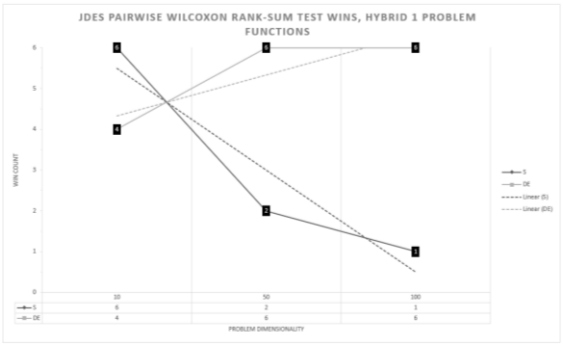


Figure 13: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across the range of hybrid-one problem functions featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 10D, 50D and 100D, for '30' experimental runs.

Aligned with the statistical analysis test (see ***Appendix G***), in lower dimensionalities of the hybridised problem domains, it is proven that jDES outperforms both DE and S; this is affirmed by the ten (of twelve) wins it achieves in ten dimensions, eight (of twelve) wins it accomplishes in fifty dimensions and seven (of twelve) wins it collects in one-hundred dimensions. When reviewing the deviation between the results obtained for all three optimisers, across all problem domains and for all dimensionalities, jDES and DE share a high regularity of converging to similar basins of attraction in either fitness function landscape; this is also transferred to solution optimality, where in ten dimensions, the solutions of jDES and DE share similar fitness values, as is determined by their mean across all thirty experimental runs. S on the contrary, is the least performing optimiser in ten dimensions, given its underperforming solutions that are also highly propagated in the domain, thus inferring an unreliable search operation that is more so dependent on the stochastic initialisation of the single solution. Generally, the difference within the optimality of the solutions produced by jDES and S across all domains and dimensionalities, is insignificant when comparing DE and S alternatively; this assumes that jDES can outperform and compete with S for the series of hybridised problems entertained by the benchmark, despite the established limitations of the computational budget. When considering said computational limitations, it further, indirectly suggests that S prematurely converges and thus to a local optimum, as it only maintains focus on one individual for the same number of fitness evaluations available; given the margin of error calculated between the mean solution fitness and the supposed, optimal fitness of each problem domain (see ***Appendix H***), this observation becomes feasible, as for all problem functions in the range, the margin of error is in fact sizeable. This determines that the solutions produced by S are sub-optimal and are thereby distant from their optimum points, which could also be factored by the non-separable subcomponents of each domain, that S is recognised to less capable in handling, as being "efficient for tackling separable problems".

V.I.IV. Composition Problem Functions

Lastly, in focus of the range of problem functions $f_{23}(x) - f_{30}(x)$ within the CEC 2014 benchmark suite, that classify as composition problem types, it was once more observed (see ***Figure 14***) that for an increasing dimensionality of each problem domain, the novel optimiser, jDES, progressively outperforms DE whilst becoming progressively outperformed by the single-solution optimiser, S. As already mentioned for all problem types discussed prior, this relationship can be understood by the trendline features in the appointed figure, that linearly characterises the pairwise win count (y axis) of jDES. Providing the observations given for the previous problem types, the correlative performances of jDES and DE can

also be resolved to the matter of an exponentially increasing population size, primarily. As composition functions are arguably more complex than their hybrid counterparts, given that the "hybrid functions are also used as the basic functions for composition functions" [3] themselves, composition functions employ highly-multimodal landscape complexes that are collectively asymmetrical, and aim to maintain "continuity around global/ local optima", to promote premature convergence in each of the problem domains. Considering each function's fitness landscape, jDES and DE as population-based optimisers are yet again expected to yield fitter solutions, on average, given their reserved explorative capabilities that are rendered useful for escaping local entrapment within sub-optimal basins of attraction, unlike S.

However, it is once more apparent that S unpredictably outperforms both jDES and DE, for an increasing problem dimensionality (see ***Appendix G***), as is determined by S attaining more pairwise wins in fifty and one-hundred-dimension configurations of the domains, when compared to jDES and DE. But as previously unseen with all other problem types, jDES is able to compete with S more in higher dimensionalities of the focused problem domains, given that jDES comparatively accumulates more pairwise wins for a similar range of problems. Disregarding the computational budget limitation of jDES and DE, a frailer, comparative performance of S can be assumed to be in consequence of increasing navigable complexities of each problem's fitness landscape, that exponentially augments the likelihood of search diversion to sub-optimal basins of attraction; this is feasible as the problem function's purpose continual features in each's landscape to stress the perturbation logic of either of the featured optimisers, and to test each optimisers "tendency to converge to the search centre, a local optimum is set to the origin as a trap for each composition function" also. As S is a greedy-search method that is prone to premature convergence, it is highly likely that S does in fact converge to sub-optimal basins of attraction in each of the problem domains; with reference to the margin of error calculated between the mean solution fitness and the supposed, optimal fitness of each problem domain (see ***Appendix H***), this hypothesis is validated, as for all problem functions in the range, the margin of error for S is significantly scaled.

Resultingly, as being recognisably better performing in noisier and highly-multimodal landscapes, disregarding fitness evaluation constraints, the explorative natures of the mutation and crossover schemes of jDES and DE should direct the search towards more promising basins of attraction, in each problem's domain when compared to S; if an unlimited computational budget were to be considered, both jDES and DE would likely be more competent within locating the optimum basin of attraction in each domain, given their capabilities to escape local entrapment and operate with diversity, whereas S would become indefinitely trapped in a local minima overtime, subject to the initial solutions starting location in the search space. This remark is viable, providing the increasingly exploitative search operation of S. However, the magnitude of optimality that jDES and DE could achieve in comparison to S, would be subjective to the performance defects of DE's basic mutation and crossover schemes in higher dimensions. Undoubtedly, this is yet again assumed to be a contributing factor to the depreciating performance of both jDES and DE, as the problem dimensionality heightens.

Meanwhile, although unlikely, given the general distribution of multimodal features in each problem functions fitness landscape, problem function $f_{28}(x)$ specifically, presents a stagnation prospect for jDES and DE, given that the majoritive area of the landscape appears featureless [3]. When evaluating the numerical results in fifty and one-hundred dimensions, jDES is observed to yield solutions with noticeably larger fitness values, than for the solutions of the preceding problem functions, despite the optimal fitness being incrementally shifted by the value of one-hundred, per problem in the benchmark. Thus, a more difficult and exhaustively wasteful search operation is recognised, that can be interpreted as fewer individuals improving per generation, periodically, as is similarly correlated by the performance of DE, which supports the potential presence of stagnation in their search. For the problem function in focus, jDES surprisingly proves to be the candidate optimiser, given its fitter and more consistently navigated-to solutions, across ten and fifty-dimension variants of the corresponding domain.

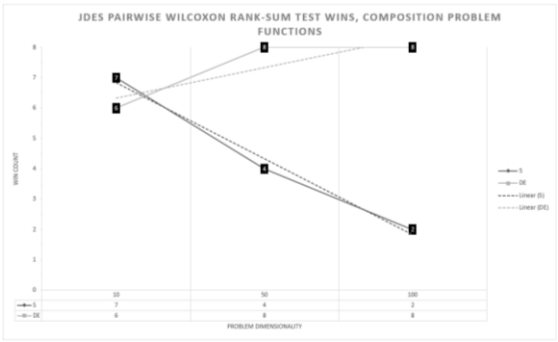


Figure 14: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across the range of composition problem functions featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 10D, 50D and 100D, for '30' experimental runs.

In accordance with statistical analysis (see ***Appendix G***), it is evidenced that jDES outperforms both DE and S in lower dimensional configurations of the composition function domains, featured; this is acknowledged by the seven (of eight) wins it achieves in ten dimensions, four (of eight) wins it accomplishes in fifty dimensions and two (of eight) wins it collects in one-hundred dimensions. When assessing the deviation between the results obtained for all three optimisers, across all problem domains and for all dimensionalities, all optimisers present the commonality of converging to similar, sub-optimal basins of attraction in either fitness function landscape, for every experimental run. Moreover, for all problem domains and for the range of dimensionalities experimented, the difference in mean fitness achieved by all optimisers is mostly negligible, meaning all optimisers are considered competitors for the series of composition problems entertained by the benchmark. Although on average, DE as in every problem domain with an increasing dimensionality, is least-performatively optimal, which is assumed to be the cause of the computational budget constraints on an ever-growing population size. In addition to the absence of a LS operator, that can refine the search and resultant optimality of the solutions that it produces, which is demonstrated by jDES throughout the experimentation led.

For the dimensional breakdown of each optimiser's performance, relative to the Wilcoxon rank-sum test experimentation presented, refer to ***Appendix I***.

V.II. Optimum Mean Error

Proceeding from the Wilcoxon rank-sum test, in order to gauge perspective upon the generality of each optimiser's performance across all thirty problem functions featured in the CEC 2014 test-bed suite [3], error ε, calculated as the difference between the mean fitness value of the solutions generated by each optimiser $f_i(\overline{x})$, and the optimal fitness value of each problem $f_i(x^*)$ in the benchmark, is compared0, for determining the outperforming optimiser relative to the NFLT [6].

$$\varepsilon = f_i(x^*) - f_i(\overline{x})$$

A substantial error infers the presence of an underperforming series of solutions, whereas an insignificant error infers the presence of a performing series of solutions, alternatively. Through the evaluation of error, can each optimisers performance be measured and acknowledged, relative to their convergence capability, in respect of converging to or as near to the optimum in all problem domains sponsored by the benchmark.

With reference to ***Figure 15*** depicted below, and in focus of its moving average (four-period) features representing the performance of each optimiser, across all problem domains with a ten-dimensional configuration, it is evidential that jDES, on average, converges within a closer vicinity of each problem domains optimum point; this is depicted by the smaller error margins exposed within the appointed figure, in addition to jDES achieving a 98.33% win-loss ratio against S and a 73.33% win-loss ratio against DE (see ***Appendix I***), which are both majoritive shares. Given the correlation established, S for all problem domains is oppositely the least-performing optimiser, in which amasses zero (of thirty) pairwise wins against jDES, compared to DE's eight (of thirty). Surprisingly, S performs better for the simple-multimodal problem types, as the comparatively smaller error margins accumulated across the range of problems suggests; given its increasingly exploitative search operation, S was expected to be optimal for unimodal landscapes providing their simplicity, however, it is assumed as previously said, to be the cause of the initial, global explorative capacity of its search radius, that could be the basis of S continually overstepping the optimum point in each of the domains, which subsequently exhausts the computational budget. Meanwhile, as expected, S on average, performs the worst out of all three optimisers for both hybrid-one and composition problem types, which is assumed to be in effect of highly-multimodal fitness landscapes, that can potentially trap and therefore cause S to

prematurely converge to a sub-optimal basin of attraction.

Whereas for DE, it can be acknowledged that a better performance is achieved for the unimodal range of problems in the series, on average, as for two of the three problems: $f_2(x)$ and $f_3(x)$, DE converges to the optimum point in each of the domains, similarly alike jDES. Meanwhile, DE is recognised to perform its worst for the range of simple-multimodal problems sponsored; this was not anticipated, as the hybrid-one and composition problem functions arguably present an increased navigable complexity for the optimiser, given the continual and featureless regions in their landscapes, that promote the presence of stagnation behaviours in its search. Although typically, the simple-multimodal problem functions feature more highly-dense multimodal features in their landscapes, across a vaster portion of the search space, which is assumed to cause an increase in the rate of computational budget consumption, as a population of individuals must be perturbed and accounted for. When aligning the margin or error with the frequency of multimodal features within the functions fitness landscapes, this observation can be deemed feasible, as is reflected by the results of problem functions $f_8(x) - f_{11}(x)$, which are considerably worse by comparison. Thus, the performance of DE for the range of problems discussed, can be reasoned as a constraint of the computational budget available.

Meanwhile, it is inevitable that jDES similarly performs better across the range of unimodal problem functions featured, on average, which is validated by the smaller margins of error generated, when compared to all other problem ranges. Evidencing this performance, jDES alike DE, converges to the optimum point in the domains of problem functions: $f_2(x)$ and $f_3(x)$, as is represented by no margin of error between the mean fitness of the solutions and the fitness of the optimum point. However, as expected unlike DE, jDES performs its worst across the range of composition problem functions in the benchmark, which is illustrated by the appropriate moving average feature in the focused figure, that underlines problem functions: $f_{23}(x)$, $f_{28}(x)$ and $f_{29}(x)$, as being the most problematic. With relevance to the fitness landscape descriptions provided previously regarding the composition problem functions, each of said problems present some featureless regions in their landscapes to the optimiser, which an optimiser that is subsequently prone to stagnation finds somewhat exhausting to overcome and offset from. Undoubtedly, within a ten-dimensional configuration of all problem's domains, jDES is averagely better performing than both DE and S; DE can compete with jDES for the unimodal and hybrid-one problem types mostly, however, this does not deter jDES away from being recognised as the outperforming optimiser. Therefore, aligned with the NFLT [6][5], the novel, memetic algorithm proves to be more general-purposed for the problems entertained by the CEC 2014 test-bed suite, in comparison to a single-solution and population-based algorithm independently.

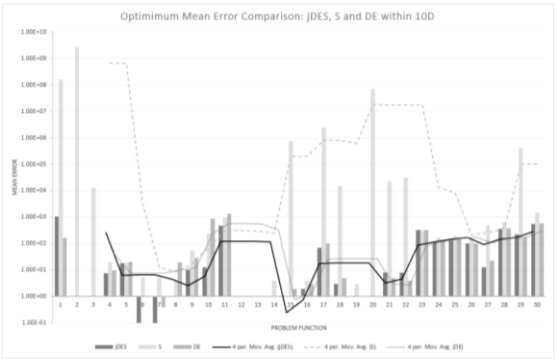


Figure 15: Graphical visualisation of the mean error of solutions generated by the jDES, DE and S optimisers, from the global optimum of each problem function featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 10D, for '30' experimental runs.

In focus of ***Figure 16*** displayed beneath this passage, and with emphasis upon its moving average (four-period) elements representing the performance of each optimiser, across all problem domains with a fifty-dimensional configuration, it can be understood that S, on average, converges within a closer neighbourhood of each problem domains optimum point. This is demonstrated by the margins of error displayed in the corresponding figure, as well as through S yielding a 51.67% win-loss ratio against jDES, that poses a 100% win-loss ratio against DE (see ***Appendix I***); although insignificantly outperforming, S is nonetheless acknowledged as the more general-purposed optimiser of the three. Oppositely, DE for all problem domains is seemingly the least-performing optimiser, in which accrues zero (of thirty) pairwise wins against jDES, compared to the twelve (of thirty) pairwise wins collected by S. Expectedly, S performs better for the unimodal problem types, as the comparatively smaller error margins accumulated across the range of problems alludes; given its increasingly exploitative search operation, the performance represented reinforces the expectation of S to navigate unimodal landscapes easily, due to their monotonic simplicity. Inversely,

as anticipated, S once again performs the worst out of all three optimisers for both hybrid-one and composition problem types; this is reassumed to be the cause of highly-multimodal fitness landscapes, that trap and reason S to prematurely converge to a sub-optimal basin of attraction.

While for DE, it can be admitted that a better performance is achieved for the simple-multimodal range of problems in the series, on average, as for two of the thirteen problems: $f_{12}(x)$ and $f_{13}(x)$, DE converges to and considerably near the optimum point in each of the domains, as observed for jDES and S also. Meanwhile, DE is acknowledged to perform its worst for the range of hybrid-one and composition problems sponsored, which was expected, given that the hybridised and composition problem functions arguably present an increased navigable complexity for the optimiser, via featureless regions in their landscapes, that resultingly promote the intervention of stagnation behaviours on its search.

Moreover, it is certain that jDES similarly performs better across the range of simple-multimodal problem functions featured, on average, which is endorsed by the smaller margins of error generated, when compared to all other problem ranges. Corroborating this performance, jDES alike DE, can converge to and around the optimum point in the domains of problem functions: $f_{12}(x)$ and $f_{13}(x)$, as is represented by the small and non-existent margins of error between the mean fitness of the solutions and the fitness of the optimum point. However, as expected alike DE, jDES performs its worst across the range of hybrid-one and composition problem functions in the benchmark; acknowledged by the appointed moving average features, problem functions: $f_{17}(x)$, $f_{21}(x)$ and $f_{29}(x)$, are recognised as being the most problematic. Once more, the performance of jDES described, is assumed to be in consequence of the complexity of each function's fitness landscape, providing the highly-multimodal and featureless regions that they facilitate, which are inevitably, computationally exhaustive for population-based algorithms as already established. Conclusively, within a fifty-dimensional configuration of all problem's domains, S is averagely better performing than both jDES and DE, however, jDES is mostly able to compete with S for the unimodal, simple-multimodal and composition problem types. Therefore, relevant to the NFLT, the memetic and single-solution optimisers prove to be more general-purposed for the problems entertained by the CEC 2014 test-bed suite, in comparison to an independent, population-based algorithm. But it must be acknowledged that the configuration submitted for jDES does not provide a performative adequacy for all the problem types and domains featured in the benchmark.

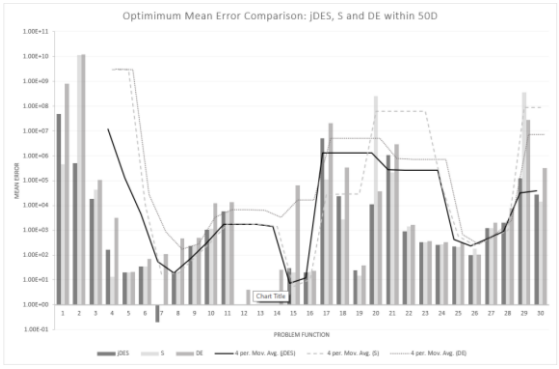


Figure 16: Graphical visualisation of the mean error of solutions generated by the jDES, DE and S optimisers, from the global optimum of each problem function featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 50D, for '30' experimental runs.

In discussion of ***Figure 17*** featured below, and in focus of its moving average (four-period) elements representing the performance of each optimiser, across all problem domains with a one-hundred-dimensional configuration, it can be supported that S, in regularity, converges within a closer proximity of each problem domains optimum point. This is depicted by the finer error margins revealed within the appointed figure, in addition to S achieving a 66.67% win-loss ratio against jDES, that attains a 100% win-loss ratio against DE (see ***Appendix I***); inevitably, S demonstrates a greater performance for the bulk of the problem functions, when compared to jDES and DE. Once again, DE for all problem domains proves to be the underperforming optimiser, for which does not accumulate a pairwise win against jDES, unlike S, that achieves fifteen (of thirty) pairwise wins. Unexpectedly, S performs better for the simple-multimodal problem types, as the comparatively smaller error margins accumulated across the range of problems indicates; given its increasingly exploitative search operation, S was expected to be optimal for unimodal landscapes providing their simplicity, however, it is assumed as previously told, to be the cause of the initial, global explorative capacity of its search radius, that could be the basis of S continually overstepping the optimum point in each of the domains, which consequently exhausts the computational budget. Meanwhile, as expected, S on average, performs its worst for the hybrid-one problem types, which is assumed to be in effect of highly-multimodal fitness landscapes,

that trap and cause S to prematurely converge to a sub-optimal basin of attraction.

Whereas for DE, it is once again realised that a better performance is attained for the simple-multimodal range of problems in the series, on average, specifically for the same two of the thirteen problems: $f_{12}(x)$ and $f_{13}(x)$, where DE demonstrates a convergence capability that facilitates near-optimum solutions to be produced in each of the domains; this is similarly achieved by jDES and S, however, both optimisers converge to the actual optimum instead. Meanwhile, DE can be observed to perform its worst for the range of hybrid-one and composition problems, which was previously achieved and expected, given their fitness landscape features that were described many times prior.

Nevertheless, it is proven that jDES, alike DE, performs better across the range of simple-multimodal problem functions entertained by the benchmark, on average, which is backed by the sharper margins of error generated, when compared to all other problem ranges. In support of this observation, jDES unlike DE, is able to converge to the optimum point in the domains of problem functions: $f_{12}(x)$ and $f_{13}(x)$, as can be interpreted by the insignificant margins of error between the mean fitness of the solutions and the fitness of the optimum point. However, as similarly expected for DE, jDES also performs its worst across the range of hybrid-one and composition problem functions in the test-bed. As acknowledged by the appointed moving average features, problem functions: $f_{17}(x)$, $f_{21}(x)$ and $f_{29}(x)$, are yet again recognised as being the most problematic. We can once more assume that the performance displayed and described for jDES, is proportionate to the complexity of each function's fitness landscape. Summarily, within a one-hundred-dimensional configuration of all problem's domains, S is recognisably better performing than jDES and DE, however, jDES is generally able to compete with S for the unimodal, simple-multimodal and composition problem types. Therefore, in relation to the NFLT, the memetic and single-solution optimisers prove to be more general-purposed for the problems employed by the CEC 2014 benchmark, in comparison to an independent, population-based algorithm. However, the configuration submitted for jDES also proves to not provide a performative capability that can cater for all the problem types and domains explored.

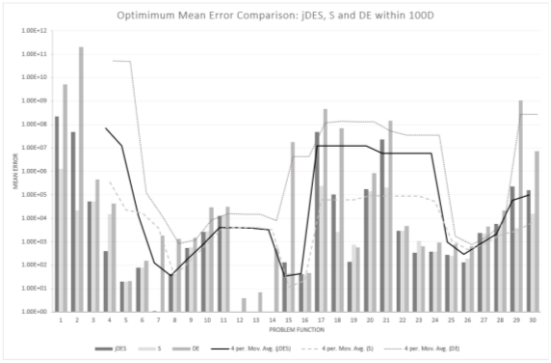


Figure 17: Graphical visualisation of the mean error of solutions generated by the jDES, DE and S optimisers, from the global optimum of each problem function featured in the CEC 2014 test-bed suite for optimisation. Experiments led in 100D, for '30' experimental runs.

VI. Conclusion

Summarily, in response to the performance evaluations of the novel optimiser proposed, throughout the experimentation led it is obvious that jDES across ten-dimensional problem domains is the outperforming candidate, when compared to contemporary single-solution and population-based optimisers, in singularity. Inarguably, the performance of jDES for the range of unimodal, simple-multimodal, hybrid-one and composition problem function types, proves consistent for the abovesaid dimensionality, thus validating its capability in achieving an adequate performance for a range of problem domains, as opposed to a finite set [6]. However, when subjected to higher dimensional configurations of each problem domain featured, it becomes evident that jDES for an increasing dimensionality, depreciates in produced solution optimality and consistency; this is more obviously noticed within a one-hundred-dimension configuration of each problem domain. Where in fact, S, proves to perform the most-optimal within, which demeans the general-purpose capabilities of the novel optimiser pledged. Generally, jDES is performatively optimal for the series of simple-multimodal problem functions and is least-performatively optimal for the range of hybrid-one and composition functions, across the series of domain dimensionalities trialled, which can be liable for the increasing complexity of each's fitness landscape. Relevant to the NFLT, jDES also demonstrates a sub-optimal performance for a series of hybrid-one and composition problem functions, in conjunction with the array of simple-multimodal problems, which underlines its more versatile and general-purposed operation, in comparison to what can be achieved using a

problem-tailored optimiser. Given the performance fluctuation known, jDES is nonetheless considered to comply with the NFLT.

In resolve of the degrading performance of jDES and subsequently DE, in higher dimensions, it is feasible that the population size could be reduced to preserve the computational budget, from using fewer individuals in their search; this is sensible when knowing that the population size "does not need to be overly high, and smaller populations can be considerably more efficient" [1]. In use of fewer individuals, assumes that a less diverse but more exploitative operation can be achieved, to produce more performing individuals than the $10 * D$ configuration supplied. Furthermore, it would also be sensible to consider the application of multiple mutation and crossover schemes; given a set of predefined rules, it may be possible to elect a combination of operators to perform well for specific problem types, in which the multitude of operator variants exist to address. This would undoubtedly incur a vaster computational expense however, to execute. Alternatively, other adaptive DE variants could be trailed, that feature novel mutation and crossover strategies which are performatively recognised to be better than the schemes proposed by jDE; this would inevitably alter the architecture of the novel optimiser but potentially enable it to harness a more-improved framework for future MA proposals, if successful. Aside from the EA component of the MA, it would also be plausible to introduce a self-adaptive scheme for controlling the depth of the local search operator employed, for preserving the available computational budget to better balance the exploration and exploitation operations of the optimisers search; this would propose to adjust the local, procedural iterations budget and exploratory radius of S, overtime, that could be addressed by a series of rules if suited. Additionally, the activation condition for the LS operator could also be made adaptive, in attempt to minimalize wasteful exhaustions of the shared computational budget; this would provide an improved explorative capacity of the optimiser, that would aim to guarantee more performing individuals in the population.

In future developments of the work presented, it would be beneficial to incorporate the performance proposals abovementioned, as well as to fine-tune the parameters that govern the self-adaptive scheme(s) of the novel optimiser proposed, in hope of advancing its performance, generally. Moreover, given more time was available, the relationships established between the optimisers and the range of problem domains, for a varied dimensional configuration could be better realised by formulating fitness-trend graphics. However, this is non-essential.

References

[1] Mayer, D.G. and Kinghorn, B.P. and Archer, A.A. (2005) Differential evolution – an easy and efficient evolutionary algorithm for model optimisation. *Agricultural Systems*. [Online] 83 (3). Available from: https://www.sciencedirect.com/science/article/pii/S0308521X04000745 [Accessed: 05/05/21].

[2] Caraffini, F. (2014) *Novel Memetic Computing Structures for Continuous Optimisation*. [Online] De Montfort University. Available from: https://dora.dmu.ac.uk/handle/2086/10629 [Accessed: 05/05/21].

[3] Liang, J. and Qu, B. and Ponnuthurai, S. (2013) Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization. [Online]. Available from: https://www.researchgate.net/publication/271646935_Problem_definitions_and_evaluation_criteria_for_the_CEC_2014_special_session_and_competition_on_single_objective_real-parameter_numerical_optimization [Accessed: 05/05/21].

[4] Zenodo (2020) *The Stochastic Optimisation Software (SOS) platform*. [Online] Zenodo. Available from: https://zenodo.org/record/3237024#.YFc4LtynyUn [Accessed: 05/05/21].

[5] Caraffini, F. (2016) Algorithmic Issues in Computational Intelligence Optimization. From Design to Implementation, from Implementation to Design. [Online]. Available from: https://www.researchgate.net/publication/308398389_Algorithmic_Issues_in_Computational_Intelligence_Optimization_From_Design_to_Implementation_from_Implementation_to_Design [Accessed: 05/05/21].

[6] Wolpert, D. and Macready, W. (1996) No Free Lunch Theorems for Search. [Online]. Available from: https://www.researchgate.net/publication/221997149_No_Free_Lunch_Theorems_for_Search [Accessed: 05/05/21].

[7] Neri, F. and Cotta, C. (2012) Memetic algorithms and memetic computing optimization: A literature review. Swarm and Evolutionary Computation. [Online] 2. Available from: https://www.researchgate.net/publication/271880689_Memetic_algorithms_and_memetic_computing_optimization_A_literature_review [Accessed: 05/05/21].

[8] Brest, J, and Greiner, S. and Boskovic, B. and Mernik, M. and Zumer, V. (2006) Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation*. [Online] 10. Available from: https://ieeexplore.ieee.org/document/4016057 [Accessed: 05/05/21].

[9] Das, S. and Ponnuthurai, S. (2011) Differential Evolution: A Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation*. [Online] 15. Available from: https://www.researchgate.net/publication/220380793_Differential_Evolution_A_Survey_of_the_State-of-the-Art [Accessed: 05/05/21].

[10] Brest, J. and Zamuda, A. and Boskovic, B. and Maucec, M.S. and Zumer, V. (2009) Dynamic optimization using Self-Adaptive Differential Evolution. In: *IEEE Congress on Evolutionary Computation*, Trondheim, Norway, May 2009. New You: IEEE, pp. 415-422.

[11] Georgioudakis, M. and Plevris, V. (2020) On the Performance of Differential Evolution Variants in Constrained Structural Optimization. *Procedia Manufacturing*. [Online] 44. Available from: https://www.sciencedirect.com/science/article/pii/S2351978920308684 [Accessed:05/05/21].

[12] Moser, I. (2009) Hooke-Jeeves Revisited. In: *2009 IEEE Congress on Evolutionary Computation, CEC 2009. Trondheim, Norway, May 2009*. IEEE: New York, pp. 2670-2676.

[13] Bereta, M. (2019) Baldwin effect and Lamarckian evolution in a memetic algorithm for Euclidean Steiner tree problem. *Memetic Computing*. [Online] 11. Available from: https://link.springer.com/article/10.1007/s12293-018-0256-7 [Accessed: 05/05/21].

[14] Golan, E. (2014) Why Abstraction is Really Important. [Weblog] *DZone*. 3rd April. Available from: https://dzone.com/articles/why-abstraction-really [Accessed: 05/05/21].

[15] Stat Trek (2021) *Statistics Dictionary*. [Online] Stat Trek. Available from: https://stattrek.com/statistics/dictionary.aspx?definition=unimodal%20distribution [Accessed: 05/05/21].

[16] Cheric (2021) *Function of a Single Variable*. [Online] Cheric. Available from: https://www.cheric.org/files/education/cyberlecture/d200103/d200103-201.pdf [Accessed: 05/05/21].

[17] Statology (2021) What is a Multimodal Distribution? [Weblog] Statology. 9th February. Available from: https://www.statology.org/multimodal-distribution/ [Accessed: 05/05/21].

[18] Farshi, R.T. and Behjat-jamal, S. and Derakhshi, M.R.F. (2014) An Improved Multimodal PSO Method Based on Electrostatic Interaction using N-Nearest-Neighbor Local Search. *International journal of artificial intelligence & applications*. [Online] 5. Available from: https://www.researchgate.net/publication/266259145_An_Improved_Multimodal_PSO_Method_Based_on_Electrostatic_Interaction_using_N-Nearest-Neighbor_Local_Search [Accessed: 05/05/21].

Appendices

Appendix A

Self-adaptive Differential Evolution Short Distance Exploration (jDES) Memetic Algorithm Implementation

```
package algorithms;

import static utils.algorithms.Misc.generateRandomSolution;
import static utils.MatLab.max;
import static utils.MatLab.min;

import utils.random.RandUtils;
```

```
import interfaces.Algorithm;
import interfaces.Problem;
import utils.RunAndStore.FTrend;
import static utils.algorithms.Misc.toro;

/**
 * Self-Adaptive Differential Evolution Short Distance Exploration
 */
public class jDES extends Algorithm // This class implements the algorithm. Every algorithm will have to contain its specific implementation within the method "execute". The latter will contain a main loop performing the iterations, and will have to return the fitness trend (including the final best) solution. Look at this examples before implementing your first algorithm.
{
	@Override
	public FTrend execute(Problem problem, int maxEvaluations) throws Exception
	{
		// We always need an object of the kind FTrend (for storing the fitness trend), and variables for storing the dimensionality value and the bounds of the problem as showed below
		FTrend FT = new FTrend(); // Create a fitness trend instance
		int problemDimension = problem.getDimension(); // Store the dimensionality of the problem domain
		double[][] bounds = problem.getBounds(); // Store the bounds of each variable in the problem domain

		double[] best = new double[problemDimension]; // Initialise the best known solution variable
		double fBest = Double.NaN; // Initialise the fitness value, i.e. "f(x)", of the best solution known
		int k = 0; // Initialise the incremental counter variable

		// Self-Adaptive Differential Evolution (jDE) variables
		int populationSize = getParameter("p0").intValue(); // Initialise the population size
		double scalingFactorLowerBound = getParameter("p3").doubleValue(); // Initialise the upper bound value of the scaling factor of the mutation operator
		double scalingFactorUpperBound = getParameter("p4").doubleValue(); // Initialise the lower bound value of the scaling factor of the mutation operator
		double tauOne = getParameter("p5").doubleValue(); // Initialise the first random probability controlling the scaling factor of the mutation operator
		double tauTwo = getParameter("p6").doubleValue(); // Initialise the second random probability controlling the binomial crossover rate of the crossover operator

		double[] scalingFactor = new double[populationSize]; // Initialise the scaling factor of the mutation operator
		double[] crossoverRate = new double[populationSize]; // Initialise the crossover rate of the binomial crossover operator

		double[][] population = new double[populationSize][problemDimension]; // Initialise the population of individuals (solutions) variable
		double[] solutionFitness = new double[populationSize]; // Initialise the fitness of a solution (individual) variable

		// Short Distance Exploration (SDE) variables
		double alpha = getParameter("p7"); // Initialise the alpha cut value of the length of the problems decision space
		double proceduralIterations = getParameter("p8").intValue(); // Initialise the procedural iteration budget variable

		double fShort = fBest; // Initialise the fitness value, i.e. "f(x)", of the trial solution
		double[] xShort = best; // Initialise the trial solution variable
		double[] exploratoryRadius = new double[problemDimension]; // Initialise the exploratory radius variable for the Short Distance Exploration (S) algorithm

		// Evaluate initial population of individuals
		for (int i = 0; i < populationSize; i++) // For the size of the population of individuals (solutions), do the following
		{
			double[] initialSolution = generateRandomSolution(bounds, problemDimension); // Generate the initial solution (guess)

			for (int j = 0; j < problemDimension; j++) // For the dimensionality of the problem, do the following
			{
				population[i][j] = initialSolution[j]; // Initialise the iterated solution (individual) that comprises the population, to the initial solution randomly generated
			}

			solutionFitness[i] = problem.f(population[i]); // Calculate and store the fitness (value) of the iterated solution (individual)
			k++; // Increment the counter to near the computational budget

			if (i == 0 || solutionFitness[i] < fBest) // If the solution generated is the first solution in the population or the fitness of the solution is better than the fitness of the best known solution, do the following
```

```
		{
			fBest = solutionFitness[i]; // Store the iterated solutions (individuals) fitness (value) as the best known solution
			//FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)

			for (int j = 0; j < problemDimension; j++) // For the dimensionality of the problem, do the following
			{
				best[j] = population[i][j]; // Update the best solution to the points comprising the iterated solution (individual) in the population
			}
		}

		if (i == 0 || k % 100 == 0) // If the iterated individual in the population is first individual in the population or the current iteration is divisible by '100' and has no remainder, do the following
		{
			FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
		}
	}

	// Main loop
	while (k < maxEvaluations) // While the computational budget has not been met, do the following
	{
		double[][] nextGeneration = new double[populationSize][problemDimension]; // Reinitialise the next generation of individuals representing the population variable

		for (int j = 0; j < populationSize && k < maxEvaluations; j++) // For the size of the population of individuals (solutions) and whilst within the computational budget, do the following
		{
			int fitnessTrendPopulated = 0; // Initialise the fitness trend populated variable

			double[] mutantIndividual = new double[problemDimension]; // Reinitialise the mutated individuals variable
			double[] crossoverIndividual = new double[problemDimension]; // Reinitialise the offspring variable
			double[] populationIndividual = new double[problemDimension]; // Reinitialise the individual (solution) comprising the population variable

			double currentFitness = Double.NaN; // Reinitialise the fitness of the current solution (individual)
			double crossoverFitness = Double.NaN; // Reinitialise the fitness of the offspring solution (individual)

			for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
			{
				populationIndividual[l] = population[j][l]; // Initialise the individual (solution) in the population to the iterated individual in the population
			}

			currentFitness = solutionFitness[j]; // Set the current fitness (value) to the fitness of the current individual (solution)

			// Update the scaling factor (F)
			if (RandUtils.random() < tauOne) // If the randomly generated number is smaller than the probability of controlling the scaling factor of the mutation operator, do the following
			{
				scalingFactor[j] = scalingFactorLowerBound + RandUtils.random() * scalingFactorUpperBound; // Resample the scaling factor for the iterated individual of the population
			}

			// DE/rand/1 mutation operator
			mutantIndividual = originalMutation(population, scalingFactor[j], problemDimension); // Function call, mutate the population of individuals to obtain a new mutated individual
			//mutantIndividual = toro(mutantIndividual, bounds); // Correct the mutated individual (solution) that may exist out of the bounds of the search space (problem domain)

			// Update the crossover rate (CR)
```

```
if (RandUtils.random() < tauTwo) // If the randomly generated number is smaller than the probability of controlling the crossover rate of the binomial crossover operator, do the following
{
	crossoverRate[j] = RandUtils.random(); // Resample the crossover rate for the iterated individual of the population
}

// Binomial crossover operator
crossoverIndividual = binomialCrossover(populationIndividual, mutantIndividual, crossoverRate[j], problemDimension); // Function call, crossover the current individual and the mutated individual, binomially (recombination)
crossoverIndividual = toro(crossoverIndividual, bounds); // Correct the offspring individual (solution) that may exist out of the bounds of the search space (problem domain)

crossoverFitness = problem.f(crossoverIndividual); // Calculate and store the fitness (value) of the offspring solution (individual)
k++; // Increment the counter to near the computational budget

// Replace the original individual in the population
if (crossoverFitness < currentFitness) // If the offspring individual is fitter than the original individual in the population, do the following
{
	for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
	{
		// Survivor selection (1-to-1 spawning)
		nextGeneration[j][l] = crossoverIndividual[l]; // Replace the original individual (solution) in the population with the offspring individual (solution)
	}

	solutionFitness[j] = crossoverFitness; // Update the fitness of the individual (solution) to the fitness of the offspring individual (solution)

	// Update the best known solution
	if (crossoverFitness < fBest) // If the offspring individual (solution) is fitter than the best known individual (solution), do the following
	{
		fBest = crossoverFitness; // Update the fitness (value) of the best known solution to the fitness (value) of the offspring individual (solution)
		//FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)

		if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100' and has no remainder whilst the fitness trend has not been populated for the current iteration, do the following
		{
			FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
			fitnessTrendPopulated = k; // The fitness trend has been populated for the current iteration
		}

		for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
		{
			best[l] = crossoverIndividual[l]; // Update the best known individual (solution) to the offspring individual (solution)
		}

		// Reset exploratory radius
		for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
		{
			exploratoryRadius[i] = alpha * (bounds[i][1] - bounds[i][0]); // Calculate and the exploratory radius for each variable in the problem domain
		}

		// Main loop
```

```
for (int l = 0; l < proceduralIterations && k < maxEvaluations; l++) // For the size of the procedural iteration budget and whilst within the computational budget, do the following
{
	boolean improved = false; // Initialise the improved variable (reinitialise every iteration)

	for (int i = 0; i < problemDimension && k < maxEvaluations; i++) // For the dimensionality of the problem and whilst within the computational budget, do the following
	{
		xShort[i] = best[i] - exploratoryRadius[i]; // Perturb the currently iterated variable in the problem domain, negatively, along the corresponding axis (exclusively)
		xShort = toro(xShort, bounds); // Correct the trial solution that may exist out of the bounds of the search space (problem domain)

		fShort = problem.f(xShort); // Calculate the new fitness value of the trial solution
		k++; // Increment the counter to near the computational budget

		if (fShort <= fBest) // If the trial solution is an improvement or equivalent to the best known solution, do the following
		{
			best[i] = xShort[i]; // Update the best known solution to the current trial solution (one variable perturbed at each iteration)

			fBest = fShort; // Store the fitness value of the improved trial solution
			//FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)

			if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100' and has no remainder whilst the fitness trend has not been populated for the current iteration, do the following
			{
				FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
				fitnessTrendPopulated = k; // The fitness trend has been populated for the current iteration
			}

			improved = true; // The trial solution is an improvement or equivalent to the best known solution
		}
		else if (k < maxEvaluations) // Else if the trial solution is not an improvement to the best solution found and its within the computational budget, do the following
		{
			if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100' and has no remainder whilst the fitness trend has not been populated
```

```
for the current iteration, do the
following
{
        FT.add(k, fBest); // Add the
        best solution found in this
        iteration to the fitness
        trend (saved to a .txt file)
        fitnessTrendPopulated = k;
        // The fitness trend has
        been populated for the
        current iteration
}

xShort[i] = best[i] +
(exploratoryRadius[i] / 2); // Perturb
the currently iterated variable in the
problem domain, positively (half-step),
along the corresponding axis
(exclusively)
xShort = toro(xShort, bounds); //
Correct the trial solution that may
exist out of the bounds of the search
space (problem domain)

fShort = problem.f(xShort); //
Calculate the new fitness value of the
trial solution
k++; // Increment the counter to near
the computational budget

if (fShort <= fBest) // If the trial
solution is an improvement or
equivalent to the best known solution,
do the following
{
        best[i] = xShort[i]; //
        Update the best known
        solution to the current trial
        solution (one variable
        perturbed at each iteration)


        fBest = fShort; // Store the
        fitness value of the
        improved trial solution
        //FT.add(k, fBest); // Add
        the best solution found in
        this iteration to the fitness
        trend (saved to a .txt file)

        improved = true; // The trial
        solution is an improvement
        or equivalent to the best
        known solution
}
else // Else if the trial solution is not an
improvement to the best solution
found, do the following
{
        xShort[i] = best[i]; // Return
        to the original point after
        the perturbations, as an
        improvement could not be
        recognised
}

if (k % 100 == 0 &&
fitnessTrendPopulated != k) // If the
current iteration is divisible by '100'
and has no remainder whilst the
fitness trend has not been populated
```

```
                                        for the current iteration, do the
                                        following
                                        {
                                                FT.add(k, fBest); // Add the
                                                best solution found in this
                                                iteration to the fitness
                                                trend (saved to a .txt file)
                                                fitnessTrendPopulated = k;
                                                // The fitness trend has
                                                been populated for the
                                                current iteration
                                        }
                                }

                                if (k % 100 == 0 && fitnessTrendPopulated != k) //
                                If the current iteration is divisible by '100' and has
                                no remainder whilst the fitness trend has not
                                been populated for the current iteration, do the
                                following
                                {
                                        FT.add(k, fBest); // Add the best
                                        solution found in this iteration to the
                                        fitness trend (saved to a .txt file)
                                        fitnessTrendPopulated = k; // The
                                        fitness trend has been populated for
                                        the current iteration
                                }
                        }

                        if (improved == false) // If the current best solution has never
                        improved from the initial best solution, do the following
                        {
                                for (int i = 0; i < exploratoryRadius.length; i++) //
                                For the dimensionality of the problem, do the
                                following
                                {
                                        exploratoryRadius[i] =
                                        exploratoryRadius[i] / 2; // Store the
                                        exploratory radius for each variable in
                                        the problem domain to itself, halved
                                        (closer search from initial solution in
                                        case the initial guess is the optimum)
                                }
                        }
                        else // Else if the current best solution has improved from
                        the initial best solution, do the following
                        {
                                for (int i = 0; i < problemDimension; i++) // For the
                                dimensionality of the problem, do the following
                                {
                                        nextGeneration[j][i] = best[i]; //
                                        Replace the original individual
                                        (solution) in the population with the
                                        best known solution (individual)
                                }

                                solutionFitness[j] = fBest; // Update the fitness of
                                the individual (solution) to the fitness of the best
                                known solution (individual)
                        }
                }
        }
}
else // Else if the offspring individual is not fitter than the original individual in the population,
do the following
{
        for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem,
        do the following
        {
                nextGeneration[j][l] = populationIndividual[l]; // Restore the design
                variables (genes) of the original individual (solution) as the updated
```

```
                                        individual (solution)
                                }

                                solutionFitness[j] = currentFitness; // Restore the fitness (value) of the solution to
                                the fitness (value) of the iterated individual (solution)
                        }

                        if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100'
                        and has no remainder whilst the fitness trend has not been populated for the current
                        iteration, do the following
                        {
                                FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend
                                (saved to a .txt file)
                                fitnessTrendPopulated = k; // The fitness trend has been populated for the current
                                iteration
                        }
                }

                // Survivor selection (1-to-1 spawning)
                population = nextGeneration; // Update the current population of individuals to the next generation of
                fitter individuals
        }

        finalBest = best; // Store the final iteration of the best known solution
        FT.add(k, fBest); // Add the final iteration of the best known solution to the fitness trend (saved to a .txt file)

        return FT; // Return the fitness trend
}

/**
* DE/rand/1 mutation operator
*
* @param population pool of solutions (individuals)
* @param scalingFactor scale factor of mutation
* @param problemDimensionality dimensionality of the problem domain
* @return mutantIndividuals mutated individual
*/
public static double[] originalMutation(double[][] population, double scalingFactor, int problemDimensionality)
{
        int problemDimension = problemDimensionality; // Store the dimensionality of the problem domain
        int populationSize = population.length; // Store the size of the population of individuals (solutions)

        int[] randomPermutation = new int[populationSize]; // Initialise the random permutation variable

        for (int i = 0; i < populationSize; i++) // For the size of the population of individuals (solutions), do the following
        {
                randomPermutation[i] = i; // Store the increment counter at each index of the random permutation
                variable (array: 0, 1, 2, 3...)
        }

        randomPermutation = RandUtils.randomPermutation(randomPermutation); // Permute the order of the elements
        contained within the random permutation variable

        int randomPointOne   = randomPermutation[0]; // Store the first element of the randomly permuted sequence of
        elements
        int randomPointTwo   = randomPermutation[1]; // Store the second element of the randomly permuted sequence of
        elements
        int randomPointThree = randomPermutation[2]; // Store the third element of the randomly permuted sequence of
        elements

        double[] mutantIndividual = new double[problemDimension]; // Initialise the mutated individual (solution) variable

        for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
        {
                mutantIndividual[i] = population[randomPointOne][i] + scalingFactor * (population[randomPointTwo][i] –
                population[randomPointThree][i]); // Linearly combine the randomly selected points in the population,
                to move the individual (solution) to a new promising area in the problem domain
        }

        return mutantIndividual; // Return the mutated individual
}
```

```
	/**
	* Binomial crossover operator
	*
	* @param populationIndividual original parent solution
	* @param mutantIndividual mutated parent solution
	* @param crossoverRate crossover rate
	* @return offspringSolution offspring solution
	*/
	public static double[] binomialCrossover(double[] populationIndividual, double[] mutantIndividual, double crossoverRate, int
	problemDimensionality)
	{
		int problemDimension = problemDimensionality; // Store the dimensionality of the problem domain
		int randomIndex = RandUtils.randomInteger(problemDimension - 1); // Generate a random index sampled from the
		range of [0, 1]

		double[] offspringSolution = new double[problemDimension]; // Initialise the offspring solution variable

		for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
		{
			if (RandUtils.random() < crossoverRate || i == randomIndex) // If the randomly generated number
			(within the range of [0, 1]) is smaller than the crossover rate or the increment counter is equal to the
			randomly generated index, do the following
			{
				offspringSolution[i] = mutantIndividual[i]; // Set the iterated design variable of the offspring
				solution (individual) to the iterated design variable of the mutated individual (solution)
			}
			else // Else if the randomly generated number is not smaller than the crossover rate or the increment
			counter is not equal to the randomly generated index, do the following
			{
				offspringSolution[i] = populationIndividual[i]; // Set the iterated design variable of the
				offspring solution to the iterated design variable of the original individual (solution)
			}
		}

		return offspringSolution; // Return the offspring solution
	}
}
```

Appendix B

Differential Evolution (DE) Algorithm Implementation

```
package algorithms;

import static utils.algorithms.Misc.generateRandomSolution;
import static utils.MatLab.max;
import static utils.MatLab.min;

import utils.random.RandUtils;
import interfaces.Algorithm;
import interfaces.Problem;
import utils.RunAndStore.FTrend;
import static utils.algorithms.Misc.toro;

/**
 * Differential Evolution (original variant - rand/1/bin)
 */
public class DE extends Algorithm // This class implements the algorithm. Every algorithm will have to contain its specific implementation
within the method "execute". The latter will contain a main loop performing the iterations, and will have to return the fitness trend
(including the final best) solution. Look at this examples before implementing your first algorithm.
{
	@Override
	public FTrend execute(Problem problem, int maxEvaluations) throws Exception
	{
		// We always need an object of the kind FTrend (for storing the fitness trend), and variables for storing the
		dimensionality value and the bounds of the problem as showed below
		FTrend FT = new FTrend(); // Create a fitness trend instance
		int problemDimension = problem.getDimension(); // Store the dimensionality of the problem domain
		double[][] bounds = problem.getBounds(); // Store the bounds of each variable in the problem domain
```

```
double[] best = new double[problemDimension]; // Initialise the best known solution variable
double fBest = Double.NaN; // Initialise the fitness value, i.e. "f(x)", of the best solution known
int k = 0; // Initialise the incremental counter variable

int populationSize       = getParameter("p0").intValue(); // Initialise the population size
double scalingFactor = getParameter("p1").doubleValue(); // Initialise the scaling factor of the mutation operator
double crossoverRate = getParameter("p2").doubleValue(); // Initialise the crossover rate of the binomial crossover
operator

double[][] population = new double[populationSize][problemDimension]; // Initialise the population of individuals
(solutions) variable
double[] solutionFitness = new double[populationSize]; // Initialise the fitness of a solution (individual) variable

// Evaluate initial population of individuals
for (int i = 0; i < populationSize; i++) // For the size of the population of individuals (solutions), do the following
{
        double[] initialSolution = generateRandomSolution(bounds, problemDimension); // Generate the initial
        solution (guess)

        for (int j = 0; j < problemDimension; j++) // For the dimensionality of the problem, do the following
        {
                population[i][j] = initialSolution[j]; // Initialise the iterated solution (individual) that comprises
                the population, to the initial solution randomly generated
        }

        solutionFitness[i] = problem.f(population[i]); // Calculate and store the fitness (value) of the iterated
        solution (individual)
        k++; // Increment the counter to near the computational budget

        if (i == 0 || solutionFitness[i] < fBest) // If the solution generated is the first solution in the population or
        the fitness of the solution is better than the fitness of the best known solution, do the following
        {
                fBest = solutionFitness[i]; // Store the iterated solutions (individuals) fitness (value) as the best
                known solution
                //FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to
                a .txt file)

                for (int j = 0; j < problemDimension; j++) // For the dimensionality of the problem, do the
                following
                {
                        best[j] = population[i][j]; // Update the best solution to the points comprising the
                        iterated solution (individual) in the population
                }
        }

        if (i == 0 || k % 100 == 0) // If the iterated individual in the population is first individual in the population
        or the current iteration is divisible by '100' and has no remainder, do the following
        {
                FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a
                .txt file)
        }
}

// Main loop
while (k < maxEvaluations) // While the computational budget has not been met, do the following
{
        double[][] nextGeneration = new double[populationSize][problemDimension]; // Reinitialise the next
        generation of individuals representing the population variable

        for (int j = 0; j < populationSize && k < maxEvaluations; j++) // For the size of the population of individuals
        (solutions) and whilst within the computational budget, do the following
        {
                int fitnessTrendPopulated = 0; // Initialise the fitness trend populated variable

                double[] mutantIndividual          = new double[problemDimension]; // Reinitialise the
                mutated individuals variable
                double[] crossoverIndividual  = new double[problemDimension]; // Reinitialise the offspring
                variable
                double[] populationIndividual = new double[problemDimension]; // Reinitialise the individual
                (solution) comprising the population variable
```

```
double currentFitness                = Double.NaN; // Reinitialise the fitness of the current solution (individual)
double crossoverFitness = Double.NaN; // Reinitialise the fitness of the offspring solution (individual)

for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
{
	populationIndividual[l] = population[j][l]; // Initialise the individual (solution) in the population to the iterated individual in the population
}

currentFitness = solutionFitness[j]; // Set the current fitness (value) to the fitness of the current individual (solution)

// DE/rand/1 mutation operator
mutantIndividual = originalMutation(population, scalingFactor, problemDimension); // Function call, mutate the population of individuals to obtain a new mutated individual
//mutantIndividual = toro(mutantIndividual, bounds); // Correct the mutated individual (solution) that may exist out of the bounds of the search space (problem domain)

// Binomial crossover operator
crossoverIndividual = binomialCrossover(populationIndividual, mutantIndividual, crossoverRate, problemDimension); // Function call, crossover the current individual and the mutated individual, binomially (recombination)
crossoverIndividual = toro(crossoverIndividual, bounds); // Correct the offspring individual (solution) that may exist out of the bounds of the search space (problem domain)

crossoverFitness = problem.f(crossoverIndividual); // Calculate and store the fitness (value) of the offspring solution (individual)
k++; // Increment the counter to near the computational budget

// Replace the original individual in the population
if (crossoverFitness < currentFitness) // If the offspring individual is fitter than the original individual in the population, do the following
{
	for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
	{
		// Survivor selection (1-to-1 spawning)
		nextGeneration[j][l] = crossoverIndividual[l]; // Replace the original individual (solution) in the population with the offspring individual (solution)
	}

	solutionFitness[j] = crossoverFitness; // Update the fitness of the individual (solution) to the fitness of the offspring individual (solution)

	// Update the best known solution
	if (crossoverFitness < fBest) // If the offspring individual (solution) is fitter than the best known individual (solution), do the following
	{
		fBest = crossoverFitness; // Update the fitness (value) of the best known solution to the fitness (value) of the offspring individual (solution)
		//FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)

		if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100' and has no remainder whilst the fitness trend has not been populated for the current iteration, do the following
		{
			FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
			fitnessTrendPopulated = k; // The fitness trend has been populated for the current iteration
		}

		for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem, do the following
		{
```

```
                                        best[l] = crossoverIndividual[l]; // Update the best known
                                        individual (solution) to the offspring individual (solution)
                                }
                        }
                }
                else // Else if the crossover individual is not fitter than the original individual in the
                population, do the following
                {
                        for (int l = 0; l < problemDimension; l++) // For the dimensionality of the problem,
                        do the following
                        {
                                nextGeneration[j][l] = populationIndividual[l]; // Restore the design
                                variables (genes) of the original individual (solution) as the updated
                                individual (solution)
                        }

                        solutionFitness[j] = currentFitness; // Restore the fitness (value) of the solution to
                        the fitness (value) of the iterated individual (solution)
                }

                if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100'
                and has no remainder whilst the fitness trend has not been populated for the current
                iteration, do the following
                {
                        FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend
                        (saved to a .txt file)
                        fitnessTrendPopulated = k; // The fitness trend has been populated for the current
                        iteration
                }
        }

        // Survivor selection (1-to-1 spawning)
        population = nextGeneration; // Update the current population of individuals to the next generation of
        fitter individuals
    }

    finalBest = best; // Store the final iteration of the best known solution
    FT.add(k, fBest); // Add the final iteration of the best known solution to the fitness trend (saved to a .txt file)

    return FT; // Return the fitness trend
}

/**
* DE/rand/1 mutation operator
*
* @param population pool of solutions (individuals)
* @param scalingFactor scale factor of mutation
* @param problemDimensionality dimensionality of the problem domain
* @return mutantIndividuals mutated individual
*/
public static double[] originalMutation(double[][] population, double scalingFactor, int problemDimensionality)
{
    int problemDimension = problemDimensionality; // Store the dimensionality of the problem domain
    int populationSize = population.length; // Store the size of the population of individuals (solutions)

    int[] randomPermutation = new int[populationSize]; // Initialise the random permutation variable

    for (int i = 0; i < populationSize; i++) // For the size of the population of individuals (solutions), do the following
    {
        randomPermutation[i] = i; // Store the increment counter at each index of the random permutation
        variable (array: 0, 1, 2, 3...)
    }

    randomPermutation = RandUtils.randomPermutation(randomPermutation); // Permute the order of the elements
    contained within the random permutation variable

    int randomPointOne   = randomPermutation[0]; // Store the first element of the randomly permuted sequence of
    elements
    int randomPointTwo   = randomPermutation[1]; // Store the second element of the randomly permuted sequence of
    elements
    int randomPointThree = randomPermutation[2]; // Store the third element of the randomly permuted sequence of
```

```
                elements

                double[] mutantIndividual = new double[problemDimension]; // Initialise the mutated individual (solution) variable

                for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
                {
                        mutantIndividual[i] = population[randomPointOne][i] + scalingFactor * (population[randomPointTwo][i] –
                        population[randomPointThree][i]); // Linearly combine the randomly selected points in the population,
                        to move the individual (solution) to a new promising area in the problem domain
                }

                return mutantIndividual; // Return the mutated individual
        }

        /**
        * Binomial crossover operator
        *
        * @param populationIndividual original parent solution
        * @param mutantIndividual mutated parent solution
        * @param crossoverRate crossover rate
        * @return offspringSolution offspring solution
        */
        public static double[] binomialCrossover(double[] populationIndividual, double[] mutantIndividual, double crossoverRate, int
        problemDimensionality)
        {
                int problemDimension = problemDimensionality; // Store the dimensionality of the problem domain
                int randomIndex = RandUtils.randomInteger(problemDimension - 1); // Generate a random index sampled from the
                range of [0, 1]

                double[] offspringSolution = new double[problemDimension]; // Initialise the offspring solution variable

                for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
                {
                        if (RandUtils.random() < crossoverRate || i == randomIndex) // If the randomly generated number
                        (within the range of [0, 1]) is smaller than the crossover rate or the increment counter is equal to the
                        randomly generated index, do the following
                        {
                                offspringSolution[i] = mutantIndividual[i]; // Set the iterated design variable of the offspring
                                solution (individual) to the iterated design variable of the mutated individual (solution)
                        }
                        else // Else if the randomly generated number is not smaller than the crossover rate or the increment
                        counter is not equal to the randomly generated index, do the following
                        {
                                offspringSolution[i] = populationIndividual[i]; // Set the iterated design variable of the
                                offspring solution to the iterated design variable of the original individual (solution)
                        }
                }

                return offspringSolution; // Return the offspring solution
        }
}
```

Appendix C

Short Distance Exploration (S) Algorithm Implementation

```
package algorithms;

import static utils.algorithms.Misc.generateRandomSolution;
import static utils.MatLab.max;
import static utils.MatLab.min;

import utils.random.RandUtils;
import interfaces.Algorithm;
import interfaces.Problem;
import utils.RunAndStore.FTrend;
import static utils.algorithms.Misc.toro;

/**
 * Short Distance Exploration
```

```
*/
public class S extends Algorithm // This class implements the algorithm. Every algorithm will have to contain its specific implementation within the method "execute". The latter will contain a main loop performing the iterations, and will have to return the fitness trend (including the final best) solution. Look at this examples before implementing your first algorithm.
{
	@Override
	public FTrend execute(Problem problem, int maxEvaluations) throws Exception
	{
		// We always need an object of the kind FTrend (for storing the fitness trend), and variables for storing the dimensionality value and the bounds of the problem as showed below
		FTrend FT = new FTrend(); // Create a fitness trend instance
		int problemDimension = problem.getDimension(); // Store the dimensionality of the problem domain
		double[][] bounds = problem.getBounds(); // Store the bounds of each variable in the problem domain

		double[] xBest = new double[problemDimension]; // Initialise the best known solution variable
		double fBest = Double.NaN; // Initialise the fitness value, i.e. "f(x)", of the best solution known
		int k = 0; // Initialise the incremental counter variable

		double alpha = getParameter("p0"); // Initialise the alpha cut value of the length of the problems decision space

		// Initial solution
		if (initialSolution != null) // If the initial solution has been assigned, do the following
		{
			xBest = initialSolution; // Set the best known solution to the already computed initial solution (guess)
			fBest = initialFitness; // Set the best known solution fitness value to the already computed initial fitness value
		}
		else // Else if the initial solution has not been assigned, do the following
		{
			xBest = generateRandomSolution(bounds, problemDimension); // Generate the initial solution (guess)
			fBest = problem.f(xBest); // Calculate the fitness value of the best known solution
			k++; // Increment the counter to near the computational budget
			FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
			//FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend (saved to a .txt file)
		}

		double fShort = fBest; // Initialise the fitness value, i.e. "f(x)", of the trial solution

		double[] xShort = xBest; // Initialise the trial solution variable
		double[] exploratoryRadius = new double[problemDimension]; // Initialise the exploratory radius variable for the Short Distance Exploration (S) algorithm

		for (int i = 0; i < problemDimension; i++) // For the dimensionality of the problem, do the following
		{
			exploratoryRadius[i] = alpha * (bounds[i][1] - bounds[i][0]); // Calculate and the exploratory radius for each variable in the problem domain
		}

		// Main loop
		while (k < maxEvaluations) // While the computational budget has not been met, do the following
		{
			boolean improved = false; // Initialise the improved variable (reinitialise every iteration)

			for (int i = 0; i < problemDimension && k < maxEvaluations; i++) // For the dimensionality of the problem and whilst within the computational budget, do the following
			{
				int fitnessTrendPopulated = 0; // Initialise the fitness trend populated variable

				xShort[i] = xBest[i] - exploratoryRadius[i]; // Perturb the currently iterated variable in the problem domain, negatively, along the corresponding axis (exclusively)
				xShort = toro(xShort, bounds); // Correct the trial solution that may exist out of the bounds of the search space (problem domain)

				fShort = problem.f(xShort); // Calculate the new fitness value of the trial solution
				k++; // Increment the counter to near the computational budget

				if (fShort <= fBest) // If the trial solution is an improvement or equivalent to the best known solution, do the following
				{
					xBest[i] = xShort[i]; // Update the best known solution to the current trial solution (one variable perturbed at each iteration)
```

```
                fBest = fShort; // Store the fitness value of the improved trial solution
                //FT.add(k, fBest); // Add the best solution found in this iteration to the fitness
                trend (saved to a .txt file)

                if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is
                divisible by '100' and has no remainder whilst the fitness trend has not been
                populated for the current iteration, do the following
                {
                        FT.add(k, fBest); // Add the best solution found in this iteration to the
                        fitness trend (saved to a .txt file)
                        fitnessTrendPopulated = k; // The fitness trend has been populated for
                        the current iteration
                }

                improved = true; // The trial solution is an improvement or equivalent to the best
                known solution
        }
        else if (k < maxEvaluations) // Else if the trial solution is not an improvement to the best
        solution found and its within the computational budget, do the following
        {
                if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is
                divisible by '100' and has no remainder whilst the fitness trend has not been
                populated for the current iteration, do the following
                {
                        FT.add(k, fBest); // Add the best solution found in this iteration to the
                        fitness trend (saved to a .txt file)
                        fitnessTrendPopulated = k; // The fitness trend has been populated for
                        the current iteration
                }

                xShort[i] = xBest[i] + (exploratoryRadius[i] / 2); // Perturb the currently iterated
                variable in the problem domain, positively (half-step), along the corresponding axis
                (exclusively)
                xShort = toro(xShort, bounds); // Correct the trial solution that may exist out of the
                bounds of the search space (problem domain)

                fShort = problem.f(xShort); // Calculate the new fitness value of the trial solution
                k++; // Increment the counter to near the computational budget

                if (fShort <= fBest) // If the trial solution is an improvement or equivalent to the
                best known solution, do the following
                {
                        xBest[i] = xShort[i]; // Update the best known solution to the current
                        trial solution (one variable perturbed at each iteration)

                        fBest = fShort; // Store the fitness value of the improved trial solution
                        //FT.add(k, fBest); // Add the best solution found in this iteration to the
                        fitness trend (saved to a .txt file)

                        if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current
                        iteration is divisible by '100' and has no remainder whilst the fitness
                        trend has not been populated for the current iteration, do the following
                        {
                                FT.add(k, fBest); // Add the best solution found in this
                                iteration to the fitness trend (saved to a .txt file)
                                fitnessTrendPopulated = k; // The fitness trend has been
                                populated for the current iteration
                        }

                        improved = true; // The trial solution is an improvement or equivalent
                        to the best known solution
                }
                else // Else if the trial solution is not an improvement to the best solution found,
                do the following
                {
                        xShort[i] = xBest[i]; // Return to the original point after the
                        perturbations, as an improvement could not be recognised
                }
        }
```

```
                                        if (k % 100 == 0 && fitnessTrendPopulated != k) // If the current iteration is divisible by '100'
                                        and has no remainder whilst the fitness trend has not been populated for the current
                                        iteration, do the following
                                        {
                                                FT.add(k, fBest); // Add the best solution found in this iteration to the fitness trend
                                                (saved to a .txt file)
                                                fitnessTrendPopulated = k; // The fitness trend has been populated for the current
                                                iteration
                                        }
                                }

                                if (improved == false) // If the current best solution has never improved from the initial best solution, do
                                the following
                                {
                                        for (int i = 0; i < exploratoryRadius.length; i++) // For the dimensionality of the problem, do
                                        the following
                                        {
                                                exploratoryRadius[i] = exploratoryRadius[i] / 2; // Store the exploratory radius for
                                                each variable in the problem domain to itself, halved (closer search from initial
                                                solution in case the initial guess is the optimum)
                                        }
                                }
                        }

                        finalBest = xBest; // Store the final iteration of the best known solution
                        FT.add(k, fBest); // Add the final iteration of the best known solution to the fitness trend (saved to a .txt file)

                        return FT; // Return the fitness trend
                }
}
```

Appendix D

Experimental Run Implementation (RunExperiments.java)

```
/** @file RunExperiments.java
 *
 *  @author Fabio Caraffini
*/
import java.util.Vector;
import interfaces.Experiment;
import static utils.RunAndStore.resultsFolder;
import experiments.*;

/**
* This class contains the main method and has to be used for launching experiments.
*/
public class RunExperiments
{

        /**
        * Main method.
        * This method has to be modified in order to launch a new experiment.
        */
        public static void main(String[] args) throws Exception
        {
                // Make sure that "results" folder exists
                resultsFolder();

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

                experiments.add(new CEC14(10)); // Add an experiment scenario for the CEC 2014 benchmark, of problem
                dimensionality '10' (design variables)
                experiments.add(new CEC14(50)); // Add an experiment scenario for the CEC 2014 benchmark, of problem
                dimensionality '50' (design variables)
                experiments.add(new CEC14(100)); // Add an experiment scenario for the CEC 2014 benchmark, of problem
                dimensionality '100' (design variables)

                for (Experiment experiment : experiments)
                {
```

```
                    //experiment.setShowPValue(true);
                    experiment.startExperiment();
                    System.out.println();
                    experiment = null;
                }
            }
}
```

Appendix E

	No.	Functions	$F_i^*=F_i(x^*)$
Unimodal Functions	1	Rotated High Conditioned Elliptic Function	100
	2	Rotated Bent Cigar Function	200
	3	Rotated Discus Function	300
Simple Multimodal Functions	4	Shifted and Rotated Rosenbrock's Function	400
	5	Shifted and Rotated Ackley's Function	500
	6	Shifted and Rotated Weierstrass Function	600
	7	Shifted and Rotated Griewank's Function	700
	8	Shifted Rastrigin's Function	800
	9	Shifted and Rotated Rastrigin's Function	900
	10	Shifted Schwefel's Function	1000
	11	Shifted and Rotated Schwefel's Function	1100
	12	Shifted and Rotated Katsuura Function	1200
	13	Shifted and Rotated HappyCat Function	1300
	14	Shifted and Rotated HGBat Function	1400
	15	Shifted and Rotated Expanded Griewank's plus Rosenbrock's Function	1500
	16	Shifted and Rotated Expanded Scaffer's F6 Function	1600
Hybrid Function 1	17	Hybrid Function 1 (N=3)	1700
	18	Hybrid Function 2 (N=3)	1800
	19	Hybrid Function 3 (N=4)	1900
	20	Hybrid Function 4 (N=4)	2000
	21	Hybrid Function 5 (N=5)	2100
	22	Hybrid Function 6 (N=5)	2200
Composition Functions	23	Composition Function 1 (N=5)	2300
	24	Composition Function 2 (N=3)	2400
	25	Composition Function 3 (N=3)	2500
	26	Composition Function 4 (N=5)	2600
	27	Composition Function 5 (N=5)	2700
	28	Composition Function 6 (N=5)	2800
	29	Composition Function 7 (N=3)	2900
	30	Composition Function 8 (N=3)	3000
Search Range: $[-100,100]^D$			

Figure 18: CEC 2014 test-bed suite, problem function overview [3].

Appendix F

CEC 2014 Test Harness Implementation (CEC14.java)

```
package experiments;
```

```
import interfaces.Experiment;
import interfaces.Algorithm;
import benchmarks.CEC2014;
import algorithms.*;

public class CEC14 extends Experiment
{
	public CEC14(int probDim) throws Exception
	{
		super(probDim, 5000, "testCEC14"); // Instantiate an instance of an experiment

		setNrRuns(30); // Set the number of runs or number of times each optimiser is executed, for every experiment
		(default is 100 iterations)

		Algorithm algorithm; // A generic optimiser

		algorithm = new jDES(); // Create a Self-Adaptive Differential Evolution Short Distance Exploration (jDES) optimiser
		instance
		algorithm.setParameter("p0", 10.0 * probDim); // Initialise the first parameter for the optimiser instance
		(population size)
		//algorithm.setParameter("p1", 0.5); // Initialise the second parameter for the optimiser instance (scaling factor - F)
		//algorithm.setParameter("p2", 0.8); // Initialise the third parameter for the optimiser instance (crossover rate - CR)
		algorithm.setParameter("p3", 0.1); // Initialise the fourth parameter for the optimiser instance (scale factor lower
		bound - Fl)
		algorithm.setParameter("p4", 1.0); // Initialise the fifth parameter for the optimiser instance (scale factor upper
		bound - Fu)
		algorithm.setParameter("p5", 1.0); // Initialise the sixth parameter for the optimiser instance (tau one)
		algorithm.setParameter("p6", 1.0); // Initialise the seventh parameter for the optimiser instance (tau two)
		algorithm.setParameter("p7", 0.2); // Initialise the eighth parameter for the optimiser instance (alpha)
		algorithm.setParameter("p8", 100.0); // Initialise the ninth parameter for the optimiser instance (procedural
		iterations)
		add(algorithm); // Add the optimiser to the containing object (list) of algorithms

		algorithm = new S(); // Create a Short Distance Exploration (S) optimiser instance
		algorithm.setParameter("p0", 0.4); // Initialise the first parameter for the optimiser instance (alpha)
		add(algorithm); // Add the optimiser to the containing object (list) of algorithms

		algorithm = new DE(); // Create a Differential Evolution (DE) optimiser instance
		algorithm.setParameter("p0", 10.0 * probDim); // Initialise the first parameter for the optimiser instance
		(population size)
		algorithm.setParameter("p1", 0.5); // Initialise the second parameter for the optimiser instance (scaling factor - F)
		algorithm.setParameter("p2", 0.8); // Initialise the third parameter for the optimiser instance (crossover rate - CR)
		add(algorithm); // Add the optimiser to the containing object (list) of algorithms

		for(int i = 1; i <= 30; i++) // For all the problems contained within the benchmark, do the following
			add(new CEC2014(probDim, i)); // Create a problem instance (current index) and add it to the containing
			object (list) of problems
	}
}
```

Appendix G

Dimensionality 10D (30 executions per experiment)						
Problem Type	**Problem Function**	**jDES**	**S**	**W**	**DE**	**W**
Unimodal	f_1	1.156e+03 ± 8.476e+02	1.553e+08 ± 6.232e+08	+	2.593e+02 ± 8.106e+01	-
	f_2	2.000e+02 ± 2.434e-02	2.614e+09 ± 1.013e+10	+	2.000e+02 ± 3.490e-05	-
	f_3	3.000e+02 ± 4.458e-05	1.347e+04 ± 1.773e+04	+	3.000e+02 ± 8.395e-09	-
Simple Multimodal	f_4	4.078e+02 ± 1.357e+01	4.206e+02 ± 1.630e+01	+	4.099e+02 ± 1.510e+01	+
	f_5	5.181e+02 ± 5.840e+00	5.200e+02 ± 4.228e-04	+	5.203e+02 ± 9.786e-02	+
	f_6	6.001e+02 ± 2.279e-01	6.053e+02 ± 2.152e+00	+	6.000e+02 ± 2.186e-02	-
	f_7	7.001e+02 ± 5.767e-02	7.052e+02 ± 2.765e+01	+	7.004e+02 ± 8.405e-02	+
	f_8	8.000e+02 ± 2.090e-05	8.042e+02 ± 1.393e+00	+	8.193e+02 ± 2.869e+00	+
	f_9	9.101e+02 ± 4.029e+00	9.528e+02 ± 5.233e+01	+	9.292e+02 ± 3.960e+00	+
	f_{10}	1.013e+03 ± 9.620e+00	1.236e+03 ± 1.087e+02	+	1.900e+03 ± 1.288e+02	+
	f_{11}	1.577e+03 ± 2.199e+02	2.079e+03 ± 3.480e+02	+	2.448e+03 ± 1.361e+02	+
	f_{12}	1.200e+03 ± 1.546e-01	1.201e+03 ± 3.078e-01	+	1.201e+03 ± 1.802e-01	+
	f_{13}	1.300e+03 ± 4.569e-02	1.300e+03 ± 1.870e-01	+	1.300e+03 ± 3.055e-02	+
	f_{14}	1.400e+03 ± 4.714e-02	1.404e+03 ± 1.526e+01	+	1.400e+03 ± 4.094e-02	+

	f_{15}	1.501e+03 ± 4.778e-01	7.888e+05 ± 2.414e+06	+	1.502e+03 ± 3.146e-01	+
	f_{16}	1.602e+03 ± 4.299e-01	1.604e+03 ± 3.488e-01	+	1.603e+03 ± 1.466e-01	+
Hybrid 1	f_{17}	1.770e+03 ± 4.001e+01	2.414e+06 ± 9.160e+06	+	1.801e+03 ± 2.600e+01	+
	f_{18}	1.803e+03 ± 1.262e+00	1.647e+04 ± 1.083e+04	+	1.805e+03 ± 1.345e+00	+
	f_{19}	1.901e+03 ± 3.055e-01	1.903e+03 ± 1.325e+00	+	1.901e+03 ± 2.465e-01	+
	f_{20}	2.001e+03 ± 5.201e-01	7.060e+07 ± 1.671e+08	+	2.001e+03 ± 4.649e-01	+
	f_{21}	2.108e+03 ± 5.958e+00	2.451e+04 ± 1.842e+04	+	2.105e+03 ± 3.829e+00	-
	f_{22}	2.208e+03 ± 5.559e+00	3.394e+04 ± 1.436e+05	+	2.204e+03 ± 4.013e+00	-
Composition	f_{23}	2.629e+03 ± 7.050e-12	2.618e+03 ± 5.910e+01	=	2.629e+03 ± 9.095e-13	-
	f_{24}	2.517e+03 ± 4.860e+00	2.570e+03 ± 3.578e+01	+	2.536e+03 ± 3.336e+00	+
	f_{25}	2.635e+03 ± 9.440e+00	2.700e+03 ± 1.764e+01	+	2.662e+03 ± 2.338e+01	+
	f_{26}	2.700e+03 ± 4.304e-02	2.756e+03 ± 7.814e+01	+	2.700e+03 ± 3.850e-02	+
	f_{27}	2.713e+03 ± 5.355e+01	3.195e+03 ± 5.708e+02	+	2.723e+03 ± 7.407e+01	+
	f_{28}	3.158e+03 ± 3.012e+00	3.437e+03 ± 4.461e+02	+	3.173e+03 ± 4.052e+01	+
	f_{29}	3.124e+03 ± 1.150e+01	4.182e+05 ± 7.546e+05	+	3.072e+03 ± 2.912e+01	-
	f_{30}	3.545e+03 ± 2.904e+01	4.510e+03 ± 3.576e+02	+	3.583e+03 ± 3.174e+01	+

Figure 19: Numerical and statistical results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 10D and for '30' experimental runs.

Dimensionality 50D (30 executions per experiment)						
Problem Type	**Problem Function**	**jDES**	**S**	**W**	**DE**	**W**
Unimodal	f_{1}	4.854e+07 ± 2.448e+07	4.543e+05 ± 1.108e+05	-	8.278e+08 ± 1.183e+08	+
	f_{2}	5.021e+05 ± 1.155e+06	1.170e+10 ± 6.300e+10	+	1.194e+10 ± 1.367e+09	+
	f_{3}	1.856e+04 ± 1.019e+04	4.373e+04 ± 1.525e+04	+	1.066e+05 ± 8.127e+03	+
Simple Multimodal	f_{4}	5.677e+02 ± 4.580e+01	4.132e+02 ± 2.577e+01	-	3.602e+03 ± 2.937e+02	+
	f_{5}	5.200e+02 ± 1.746e-02	5.200e+02 ± 8.505e-03	-	5.212e+02 ± 3.849e-02	+
	f_{6}	6.358e+02 ± 5.014e+00	6.347e+02 ± 4.184e+00	=	6.699e+02 ± 1.184e+00	+
	f_{7}	7.002e+02 ± 2.823e-01	7.000e+02 ± 7.929e-03	-	8.129e+02 ± 1.468e+01	+
	f_{8}	8.201e+02 ± 6.621e+00	8.212e+02 ± 3.496e+00	=	1.279e+03 ± 1.374e+01	+
	f_{9}	1.136e+03 ± 5.498e+01	1.152e+03 ± 6.882e+01	=	1.408e+03 ± 2.076e+01	+
	f_{10}	2.049e+03 ± 3.718e+02	2.145e+03 ± 3.120e+02	=	1.343e+04 ± 4.204e+02	+
	f_{11}	6.892e+03 ± 8.988e+02	6.893e+03 ± 8.377e+02	=	1.473e+04 ± 3.670e+02	+
	f_{12}	1.200e+03 ± 1.403e-01	1.200e+03 ± 7.623e-02	=	1.204e+03 ± 2.190e-01	+
	f_{13}	1.300e+03 ± 6.645e-02	1.301e+03 ± 1.203e-01	+	1.301e+03 ± 2.200e-01	+
	f_{14}	1.400e+03 ± 2.005e-01	1.401e+03 ± 3.406e-01	+	1.426e+03 ± 3.726e+00	+
	f_{15}	1.530e+03 ± 9.014e+00	1.520e+03 ± 7.526e+00	-	6.683e+04 ± 2.579e+04	+
	f_{16}	1.620e+03 ± 6.624e-01	1.621e+03 ± 7.679e-01	+	1.623e+03 ± 1.890e-01	+
Hybrid 1	f_{17}	5.176e+06 ± 3.368e+06	1.133e+05 ± 6.058e+04	-	2.045e+07 ± 3.430e+06	+
	f_{18}	2.546e+04 ± 2.869e+04	4.499e+03 ± 1.129e+03	-	3.432e+05 ± 1.068e+05	+
	f_{19}	1.925e+03 ± 1.058e+01	1.915e+03 ± 2.705e+00	-	1.939e+03 ± 1.905e+00	+
	f_{20}	1.313e+04 ± 4.212e+03	2.562e+08 ± 6.129e+08	+	3.932e+04 ± 7.048e+03	+
	f_{21}	1.057e+06 ± 4.055e+05	2.220e+05 ± 1.257e+05	-	2.893e+06 ± 7.348e+05	+
	f_{22}	3.128e+03 ± 2.817e+02	3.648e+03 ± 3.237e+02	+	3.890e+03 ± 1.682e+02	+
Composition	f_{23}	2.641e+03 ± 2.158e-01	2.641e+03 ± 1.076e-06	-	2.662e+03 ± 2.195e+00	+
	f_{24}	2.667e+03 ± 4.447e+00	2.677e+03 ± 3.938e+00	+	2.734e+03 ± 3.436e+00	+
	f_{25}	2.725e+03 ± 5.602e+00	2.716e+03 ± 6.239e+00	-	2.812e+03 ± 8.236e+00	+
	f_{26}	2.700e+03 ± 7.391e-02	2.782e+03 ± 1.227e+02	+	2.707e+03 ± 5.415e-01	+
	f_{27}	3.938e+03 ± 1.178e+02	3.922e+03 ± 1.210e+02	=	4.745e+03 ± 3.459e+01	+
	f_{28}	4.875e+03 ± 5.202e+02	5.306e+03 ± 7.488e+02	+	1.063e+04 ± 4.048e+02	+
	f_{29}	1.279e+05 ± 1.101e+05	3.585e+08 ± 1.338e+09	+	2.868e+07 ± 6.845e+06	+
	f_{30}	2.999e+04 ± 1.107e+04	1.739e+04 ± 3.308e+03	-	3.260e+05 ± 5.866e+04	+

Figure 20: Numerical and statistical results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 50D and for '30' experimental runs.

Dimensionality 100D (30 executions per experiment)						
Problem Type	**Problem Function**	**jDES**	**S**	**W**	**DE**	**W**
Unimodal	f_{1}	2.244e+08 ± 8.593e+07	1.330e+06 ± 3.824e+05	-	5.228e+09 ± 5.272e+08	+
	f_{2}	4.912e+07 ± 4.826e+07	2.192e+04 ± 2.130e+04	-	2.028e+11 ± 1.356e+10	+
	f_{3}	5.404e+04 ± 1.866e+04	5.255e+04 ± 1.858e+04	=	4.461e+05 ± 2.299e+04	+
	f_{4}	8.089e+02 ± 1.138e+02	1.569e+04 ± 8.187e+04	+	4.419e+04 ± 3.882e+03	+

Simple Multimodal	f_5	5.201e+02 ± 4.254e-02	5.200e+02 ± 1.208e-03	-	5.213e+02 ± 2.439e-02	+
	f_6	6.795e+02 ± 7.490e+00	6.777e+02 ± 5.135e+00	=	7.575e+02 ± 1.782e+00	+
	f_7	7.011e+02 ± 4.603e-01	7.000e+02 ± 6.662e-03	-	2.566e+03 ± 1.027e+02	+
	f_8	8.415e+02 ± 1.196e+01	8.417e+02 ± 5.220e+00	=	2.139e+03 ± 3.521e+01	+
	f_9	1.459e+03 ± 9.019e+01	1.485e+03 ± 8.019e+01	=	2.394e+03 ± 4.569e+01	+
	f_{10}	3.647e+03 ± 5.075e+02	3.733e+03 ± 5.077e+02	=	3.020e+04 ± 4.426e+02	+
	f_{11}	1.401e+04 ± 1.443e+03	1.436e+04 ± 1.245e+03	=	3.186e+04 ± 3.468e+02	+
	f_{12}	1.200e+03 ± 7.846e-02	1.200e+03 ± 9.972e-02	-	1.204e+03 ± 2.611e-01	+
	f_{13}	1.301e+03 ± 7.641e-02	1.301e+03 ± 7.641e-02	+	1.307e+03 ± 2.691e-01	+
	f_{14}	1.400e+03 ± 2.471e-01	1.400e+03 ± 1.649e-01	=	1.933e+03 ± 3.309e+01	+
	f_{15}	1.635e+03 ± 6.527e+01	1.546e+03 ± 1.243e+01	-	1.752e+07 ± 2.871e+06	+
	f_{16}	1.643e+03 ± 9.489e-01	1.643e+03 ± 1.023e+00	=	1.647e+03 ± 2.343e-01	+
Hybrid 1	f_{17}	4.907e+07 ± 2.674e+07	2.435e+05 ± 8.385e+04	-	4.609e+08 ± 7.390e+07	+
	f_{18}	1.044e+05 ± 8.730e+04	4.473e+03 ± 2.175e+03	-	7.062e+07 ± 1.248e+07	+
	f_{19}	2.039e+03 ± 2.323e+01	2.654e+03 ± 3.603e+03	+	2.484e+03 ± 3.567e+01	+
	f_{20}	1.812e+05 ± 6.113e+04	1.431e+05 ± 5.149e+04	-	8.336e+05 ± 1.736e+05	+
	f_{21}	2.365e+07 ± 1.102e+07	2.100e+05 ± 1.005e+05	-	1.434e+08 ± 3.129e+07	+
	f_{22}	5.132e+03 ± 5.905e+02	5.121e+03 ± 6.360e+02	=	6.902e+03 ± 2.238e+02	+
Composition	f_{23}	2.645e+03 ± 4.979e+00	3.399e+03 ± 4.087e+03	+	2.930e+03 ± 2.015e+01	+
	f_{24}	2.784e+03 ± 1.158e+01	2.783e+03 ± 4.894e+00	=	3.325e+03 ± 3.056e+01	+
	f_{25}	2.781e+03 ± 1.049e+01	2.751e+03 ± 1.356e+01	-	3.308e+03 ± 3.999e+01	+
	f_{26}	2.734e+03 ± 5.146e+01	2.788e+03 ± 3.229e+01	+	3.238e+03 ± 3.880e+01	+
	f_{27}	5.112e+03 ± 1.569e+02	4.976e+03 ± 1.870e+02	-	7.156e+03 ± 5.305e+01	+
	f_{28}	8.726e+03 ± 1.153e+03	8.031e+03 ± 1.135e+03	-	2.503e+04 ± 9.988e+02	+
	f_{29}	2.318e+05 ± 1.242e+05	6.705e+03 ± 4.108e+02	-	1.090e+09 ± 1.373e+08	+
	f_{30}	1.637e+05 ± 1.071e+05	1.862e+04 ± 2.406e+03	-	7.226e+06 ± 1.293e+06	+

Figure 21: Numerical and statistical results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 100D and for '30' experimental runs.

Appendix H

Dimensionality 10D (30 executions per experiment)								
Problem Type	Problem Function	$F_i^* = F_i(x^*)$	jDES		S		DE	
			Mean	Error	Mean	Error	Mean	Error
Unimodal	f_1	100	1.16E+03	1.06E+03	1.55E+08	1.55E+08	2.59E+02	1.59E+02
	f_2	200	2.00E+02	0.00E+00	2.61E+09	2.61E+09	2.00E+02	0.00E+00
	f_3	300	3.00E+02	0.00E+00	1.35E+04	1.32E+04	3.00E+02	0.00E+00
Simple Multimodal	f_4	400	4.08E+02	7.80E+00	4.21E+02	2.06E+01	4.10E+02	9.90E+00
	f_5	500	5.18E+02	1.81E+01	5.20E+02	2.00E+01	5.20E+02	2.03E+01
	f_6	600	6.00E+02	1.00E-01	6.05E+02	5.30E+00	6.00E+02	0.00E+00
	f_7	700	7.00E+02	1.00E-01	7.05E+02	5.20E+00	7.00E+02	4.00E-01
	f_8	800	8.00E+02	0.00E+00	8.04E+02	4.20E+00	8.19E+02	1.93E+01
	f_9	900	9.10E+02	1.01E+01	9.53E+02	5.28E+01	9.29E+02	2.92E+01
	f_{10}	1000	1.01E+03	1.30E+01	1.24E+03	2.36E+02	1.90E+03	9.00E+02
	f_{11}	1100	1.58E+03	4.77E+02	2.08E+03	9.79E+02	2.45E+03	1.35E+03
	f_{12}	1200	1.20E+03	0.00E+00	1.20E+03	1.00E+00	1.20E+03	1.00E+00
	f_{13}	1300	1.30E+03	0.00E+00	1.30E+03	0.00E+00	1.30E+03	0.00E+00
	f_{14}	1400	1.40E+03	0.00E+00	1.40E+03	4.00E+00	1.40E+03	0.00E+00
	f_{15}	1500	1.50E+03	1.00E+00	7.89E+05	7.87E+05	1.50E+03	2.00E+00
	f_{16}	1600	1.60E+03	2.00E+00	1.60E+03	4.00E+00	1.60E+03	3.00E+00
Hybrid 1	f_{17}	1700	1.77E+03	7.00E+01	2.41E+06	2.41E+06	1.80E+03	1.01E+02
	f_{18}	1800	1.80E+03	3.00E+00	1.65E+04	1.47E+04	1.81E+03	5.00E+00
	f_{19}	1900	1.90E+03	1.00E+00	1.90E+03	3.00E+00	1.90E+03	1.00E+00
	f_{20}	2000	2.00E+03	1.00E+00	7.06E+07	7.06E+07	2.00E+03	1.00E+00
	f_{21}	2100	2.11E+03	8.00E+00	2.45E+04	2.24E+04	2.11E+03	5.00E+00
	f_{22}	2200	2.21E+03	8.00E+00	3.39E+04	3.17E+04	2.20E+03	4.00E+00
Composition	f_{23}	2300	2.63E+03	3.29E+02	2.62E+03	3.18E+02	2.63E+03	3.29E+02
	f_{24}	2400	2.52E+03	1.17E+02	2.57E+03	1.70E+02	2.54E+03	1.36E+02
	f_{25}	2500	2.64E+03	1.35E+02	2.70E+03	2.00E+02	2.66E+03	1.62E+02
	f_{26}	2600	2.70E+03	1.00E+02	2.76E+03	1.56E+02	2.70E+03	1.00E+02
	f_{27}	2700	2.71E+03	1.30E+01	3.20E+03	4.95E+02	2.72E+03	2.30E+01
	f_{28}	2800	3.16E+03	3.58E+02	3.44E+03	6.37E+02	3.17E+03	3.73E+02
	f_{29}	2900	3.12E+03	2.24E+02	4.18E+05	4.15E+05	3.07E+03	1.72E+02
	f_{30}	3000	3.55E+03	5.45E+02	4.51E+03	1.51E+03	3.58E+03	5.83E+02

Figure 22: Numerical, optimum mean error results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 10D and for '30' experimental runs.

Dimensionality 50D (30 executions per experiment)								
Problem Type	Problem Function	$F_i^* = F_i(x^*)$	jDES		S		DE	
			Mean	Error	Mean	Error	Mean	Error
Unimodal	f_1	100	4.85E+07	4.85E+07	4.54E+05	4.54E+05	8.28E+08	8.28E+08
	f_2	200	5.02E+05	5.02E+05	1.17E+10	1.17E+10	1.19E+10	1.19E+10
	f_3	300	1.86E+04	1.83E+04	4.37E+04	4.34E+04	1.07E+05	1.06E+05
Simple Multimodal	f_4	400	5.68E+02	1.68E+02	4.13E+02	1.32E+01	3.60E+03	3.20E+03
	f_5	500	5.20E+02	2.00E+01	5.20E+02	2.00E+01	5.21E+02	2.12E+01
	f_6	600	6.36E+02	3.58E+01	6.35E+02	3.47E+01	6.70E+02	6.99E+01
	f_7	700	7.00E+02	2.00E-01	7.00E+02	0.00E+00	8.13E+02	1.13E+02
	f_8	800	8.20E+02	2.01E+01	8.21E+02	2.12E+01	1.28E+03	4.79E+02
	f_9	900	1.14E+03	2.36E+02	1.15E+03	2.52E+02	1.41E+03	5.08E+02
	f_{10}	1000	2.05E+03	1.05E+03	2.15E+03	1.15E+03	1.34E+04	1.24E+04
	f_{11}	1100	6.89E+03	5.79E+03	6.89E+03	5.79E+03	1.47E+04	1.36E+04
	f_{12}	1200	1.20E+03	0.00E+00	1.20E+03	0.00E+00	1.20E+03	4.00E+00
	f_{13}	1300	1.30E+03	0.00E+00	1.30E+03	1.00E+00	1.30E+03	1.00E+00
	f_{14}	1400	1.40E+03	0.00E+00	1.40E+03	1.00E+00	1.43E+03	2.60E+01
	f_{15}	1500	1.53E+03	3.00E+01	1.52E+03	2.00E+01	6.68E+04	6.53E+04
	f_{16}	1600	1.62E+03	2.00E+01	1.62E+03	2.10E+01	1.62E+03	2.30E+01
Hybrid 1	f_{17}	1700	5.18E+06	5.17E+06	1.13E+05	1.12E+05	2.05E+07	2.04E+07
	f_{18}	1800	2.55E+04	2.37E+04	4.50E+03	2.70E+03	3.43E+05	3.41E+05
	f_{19}	1900	1.93E+03	2.50E+01	1.92E+03	1.50E+01	1.94E+03	3.90E+01
	f_{20}	2000	1.31E+04	1.11E+04	2.56E+08	2.56E+08	3.93E+04	3.73E+04
	f_{21}	2100	1.06E+06	1.05E+06	2.22E+05	2.20E+05	2.89E+06	2.89E+06
	f_{22}	2200	3.13E+03	9.28E+02	3.65E+03	1.45E+03	3.89E+03	1.69E+03
Composition	f_{23}	2300	2.64E+03	3.41E+02	2.64E+03	3.41E+02	2.66E+03	3.62E+02
	f_{24}	2400	2.67E+03	2.67E+02	2.68E+03	2.77E+02	2.73E+03	3.34E+02
	f_{25}	2500	2.73E+03	2.25E+02	2.72E+03	2.16E+02	2.81E+03	3.12E+02
	f_{26}	2600	2.70E+03	1.00E+02	2.78E+03	1.82E+02	2.71E+03	1.07E+02
	f_{27}	2700	3.94E+03	1.24E+03	3.92E+03	1.22E+03	4.75E+03	2.05E+03
	f_{28}	2800	4.88E+03	2.08E+03	5.31E+03	2.51E+03	1.06E+04	7.83E+03
	f_{29}	2900	1.28E+05	1.25E+05	3.59E+08	3.58E+08	2.87E+07	2.87E+07
	f_{30}	3000	3.00E+04	2.70E+04	1.74E+04	1.44E+04	3.26E+05	3.23E+05

Figure 23: Numerical, optimum mean error results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 50D and for '30' experimental runs.

Dimensionality 100D (30 executions per experiment)								
Problem Type	Problem Function	$F_i^* = F_i(x^*)$	jDES		S		DE	
			Mean	Error	Mean	Error	Mean	Error
Unimodal	f_1	100	2.24E+08	2.24E+08	1.33E+06	1.33E+06	5.23E+09	5.23E+09
	f_2	200	4.91E+07	4.91E+07	2.19E+04	2.17E+04	2.03E+11	2.03E+11
	f_3	300	5.40E+04	5.37E+04	5.26E+04	5.23E+04	4.46E+05	4.46E+05
Simple Multimodal	f_4	400	8.09E+02	4.09E+02	1.57E+04	1.53E+04	4.42E+04	4.38E+04
	f_5	500	5.20E+02	2.01E+01	5.20E+02	2.00E+01	5.21E+02	2.13E+01
	f_6	600	6.80E+02	7.95E+01	6.78E+02	7.77E+01	7.58E+02	1.58E+02
	f_7	700	7.01E+02	1.10E+00	7.00E+02	0.00E+00	2.57E+03	1.87E+03
	f_8	800	8.42E+02	4.15E+01	8.42E+02	4.17E+01	2.14E+03	1.34E+03
	f_9	900	1.46E+03	5.59E+02	1.49E+03	5.85E+02	2.39E+03	1.49E+03
	f_{10}	1000	3.65E+03	2.65E+03	3.73E+03	2.73E+03	3.02E+04	2.92E+04
	f_{11}	1100	1.40E+04	1.29E+04	1.44E+04	1.33E+04	3.19E+04	3.08E+04
	f_{12}	1200	1.20E+03	0.00E+00	1.20E+03	0.00E+00	1.20E+03	4.00E+00
	f_{13}	1300	1.30E+03	1.00E+00	1.30E+03	1.00E+00	1.31E+03	7.00E+00
	f_{14}	1400	1.40E+03	0.00E+00	1.40E+03	0.00E+00	1.93E+03	5.33E+02
	f_{15}	1500	1.64E+03	1.35E+02	1.55E+03	4.60E+01	1.75E+07	1.75E+07
	f_{16}	1600	1.64E+03	4.30E+01	1.64E+03	4.30E+01	1.65E+03	4.70E+01
Hybrid 1	f_{17}	1700	4.91E+07	4.91E+07	2.44E+05	2.42E+05	4.61E+08	4.61E+08
	f_{18}	1800	1.04E+05	1.03E+05	4.47E+03	2.67E+03	7.06E+07	7.06E+07
	f_{19}	1900	2.04E+03	1.39E+02	2.65E+03	7.54E+02	2.48E+03	5.84E+02
	f_{20}	2000	1.81E+05	1.79E+05	1.43E+05	1.41E+05	8.34E+05	8.32E+05
	f_{21}	2100	2.37E+07	2.36E+07	2.10E+05	2.08E+05	1.43E+08	1.43E+08

	f_{22}	2200	5.13E+03	2.93E+03	5.12E+03	2.92E+03	6.90E+03	4.70E+03
Composition	f_{23}	2300	2.65E+03	3.45E+02	3.40E+03	1.10E+03	2.93E+03	6.30E+02
	f_{24}	2400	2.78E+03	3.84E+02	2.78E+03	3.83E+02	3.33E+03	9.25E+02
	f_{25}	2500	2.78E+03	2.81E+02	2.75E+03	2.51E+02	3.31E+03	8.08E+02
	f_{26}	2600	2.73E+03	1.34E+02	2.79E+03	1.88E+02	3.24E+03	6.38E+02
	f_{27}	2700	5.11E+03	2.41E+03	4.98E+03	2.28E+03	7.16E+03	4.46E+03
	f_{28}	2800	8.73E+03	5.93E+03	8.03E+03	5.23E+03	2.50E+04	2.22E+04
	f_{29}	2900	2.32E+05	2.29E+05	6.71E+03	3.81E+03	1.09E+09	1.09E+09
	f_{30}	3000	1.64E+05	1.61E+05	1.86E+04	1.56E+04	7.23E+06	7.22E+06

Figure 24: Numerical, optimum mean error results table for the three featured optimisers, benchmarked over the problems sponsored by the CEC 2014 test-bed suite for optimisation, in 100D and for '30' experimental runs.

Appendix I

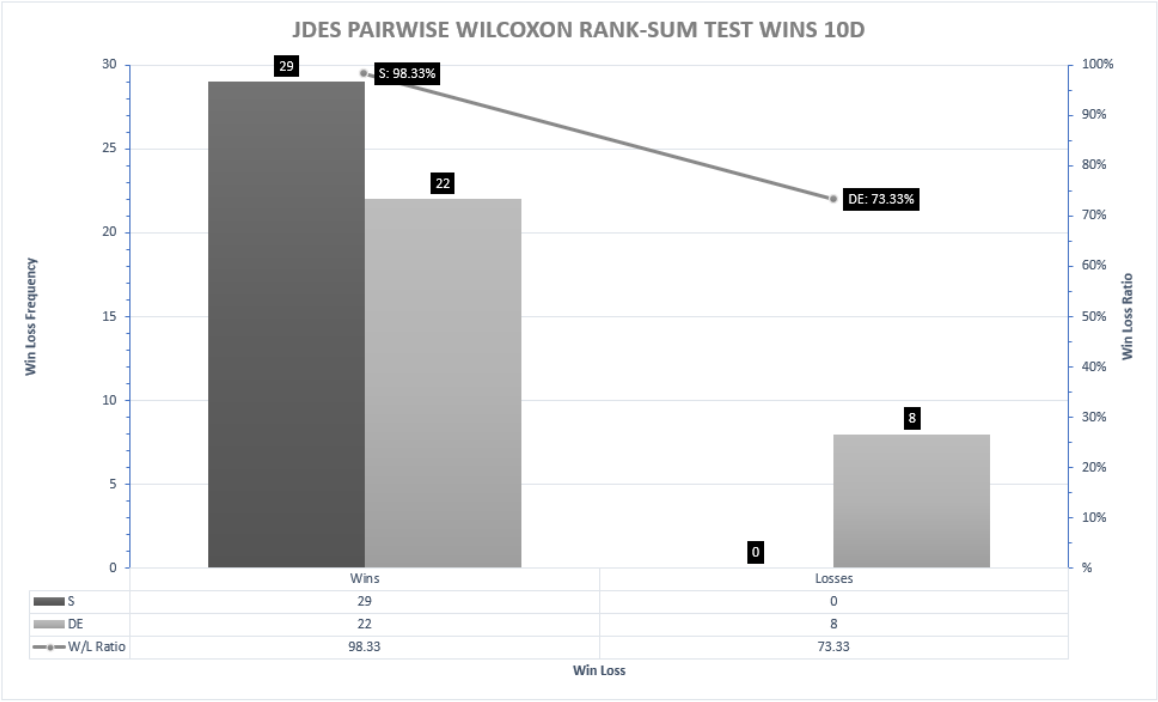


Figure 25: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across all problem functions featured in the CEC 2014 test-bed suite for optimisation, in 10D.

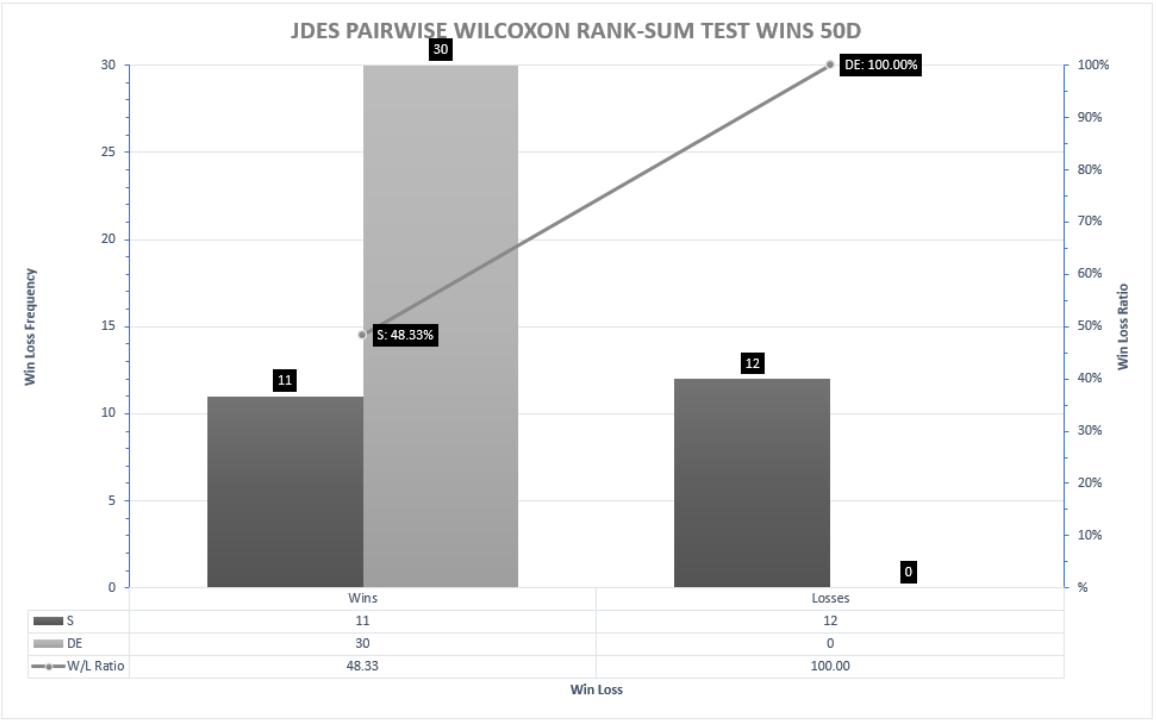


Figure 26: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across all problem functions featured in the CEC 2014 test-bed suite for optimisation, in 50D.

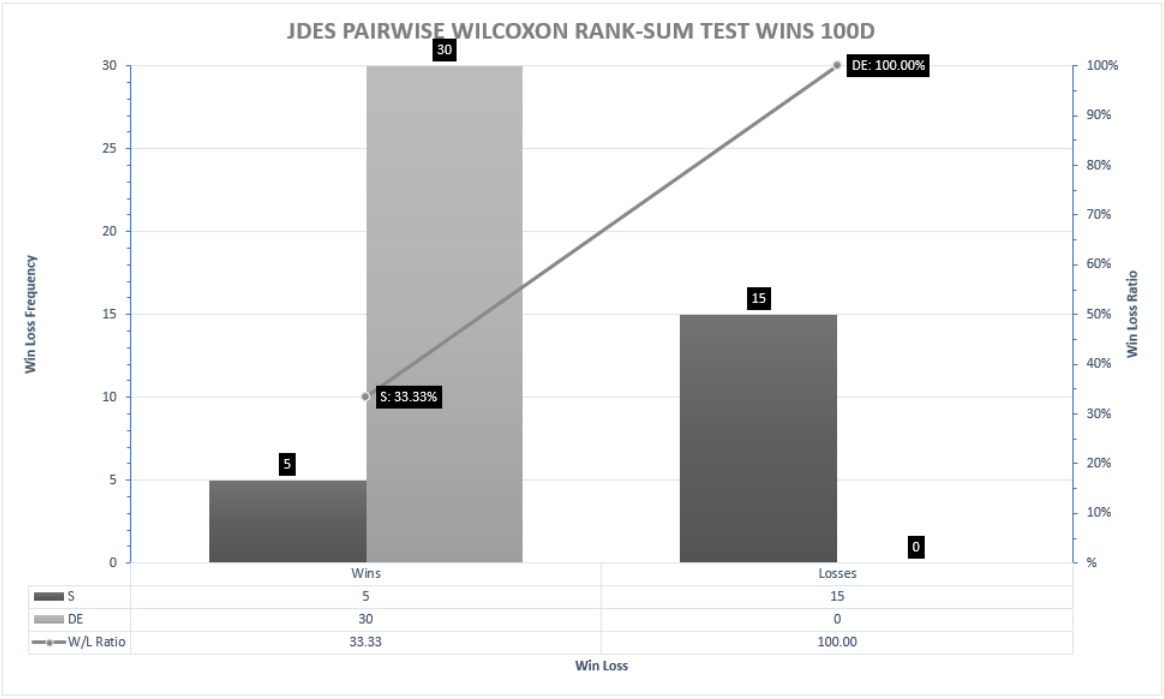
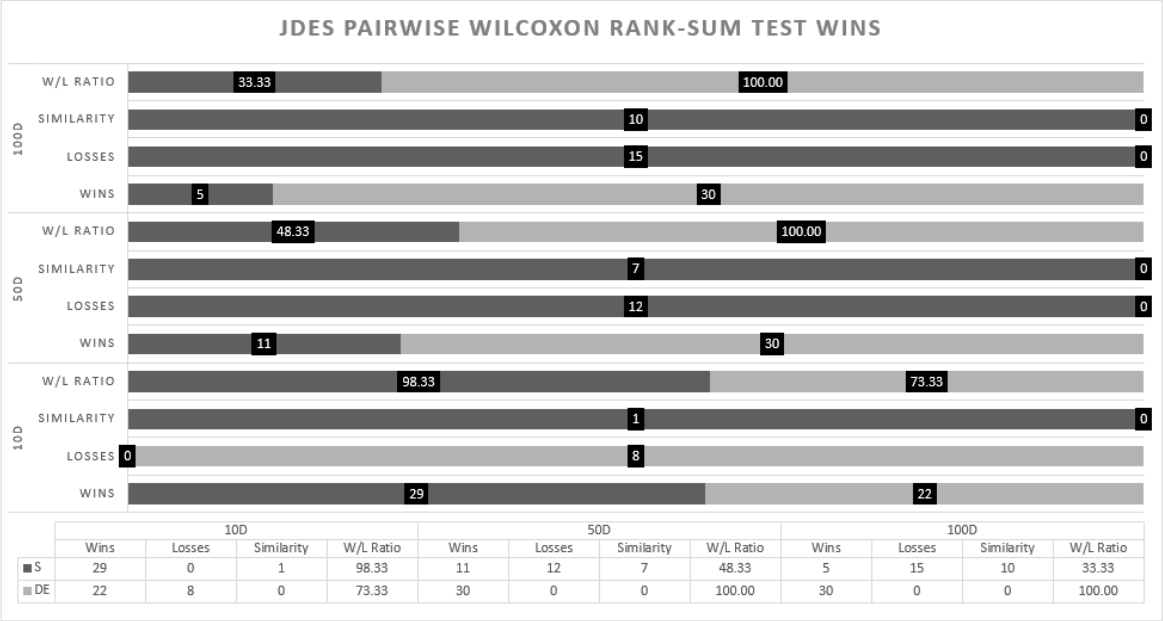


Figure 27: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across all problem functions featured in the CEC 2014 test-bed suite for optimisation, in 100D.



	10D				50D				100D			
	Wins	Losses	Similarity	W/L Ratio	Wins	Losses	Similarity	W/L Ratio	Wins	Losses	Similarity	W/L Ratio
S	29	0	1	98.33	11	12	7	48.33	5	15	10	33.33
DE	22	8	0	73.33	30	0	0	100.00	30	0	0	100.00

Figure 28: Graphical visualisation of the Wilcoxon rank-sum test pairwise win count, for jDES against DE and S, across all problem functions featured in the CEC 2014 test-bed suite for optimisation, in 10D, 50D and 100D.