 doi:10.20944/preprints202003.0381.v1

Article

The SOS Platform: Designing, Tuning and Statistically Benchmarking Optimisation Algorithms

Fabio Caraffini

Institute of Artificial Intelligence, School of Computer Science and Informatics, De Montfort University, Leicester (UK); fabio.caraffini@dmu.ac.uk

Abstract: The Stochastic Optimisation Software (SOS) is a Java platform facilitating the algorithmic design process and the evaluation of metaheuristic optimisation algorithms. It reduces the burden of coding miscellaneous methods for dealing with several bothersome and time-demanding tasks such as parameter tuning, implementation of comparison algorithms and testbed problems, collecting and processing data to display results, measuring algorithmic overhead, etc. SOS provides numerous off-the-shelf methods including 1) customised implementations of statistical tests, such as the Wilcoxon Rank-Sum test and the Holm-Bonferroni procedure, for comparing performances of optimisation algorithms and automatically generate result tables in PDF and LaTeX formats; 2) the implementation of an original advanced statistical routine for accurately comparing couples of stochastic optimisation algorithms; 3) the implementation of a novel testbed suite for continuous optimisation, derived from the IEEE CEC 2014 benchmark, allowing for controlled activation of the rotation operator. each testbed function. Moreover, this article comments on the current state of the literature in stochastic optimisation and highlights similarities shared by modern metaheuristics inspired by nature. It is argued that the vast majority of these algorithms are simply a reformulation of the same methods and that metaheuristics for optimisation should be simply treated as stochastic processes with less emphasis on the inspiring metaphor behind them.

Keywords: algorithmic design; metaheuristic optimisation; evolutionary computation; swarm intelligence; memetic computing; parameter tuning; fitness trend; Wilcoxon Rank-Sum; Holm-Bonferroni; benchmark suite

1. Introduction

Experimentalism plays a major role in all fields of metaheuristic optimisation, as e.g. evolutionary computation (EC) and swarm intelligence (SI), and can be a time-demanding, repetitive and tedious process when required to fine-tune optimisers, compare them or to validate new algorithmic strategies on benchmark problems. Due to their stochastic nature, and the lack of theoretical knowledge on the internal dynamics of many of these nature-inspired approaches, their algorithmic design is partially blind and often empirically performed through a series of trial-and-error phases [1]. This involves using several artificially build benchmark problems, simulating characteristics present in real-world scenarios, but for which some knowledge is available. The use of such testbed problems is preferable since they usually have a more reasonable execution time than real-world applications, they can be run any machine, this being e.g. a modest personal computer or a high-performance computing system, and they are not subject to time and other limitations typical of real-world scenarios if not purposely imposed to reproduce a specific situation. Thus, these problems are suitable choices for performing the algorithmic design phase of novel algorithms and evaluating the performances of specific operators and algorithmic variants. By design, these problems display particular characteristics reported in the corresponding technical reports in term of (e.g. ill-conditioning, separability, m-separability, noisy functional landscapes, modality (i.e. multimodal or unimodal function) etc., allowing also for studying the algorithmic behaviour of certain operators [2–4]).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

In this light, the research community benefits from platforms featuring a large number of algorithms and benchmark testbed problems, as e.g. the Decision Tree for Optimization software [5], COCO [6], IOHprofiler [7] and jMetal [8], which facilitate this testing. However, the availability of code implementing complete of benchmark suites and popular optimisation algorithms is only one amongst the many "tools" required by an algorithm designer to work efficiently and productively focus 100% on algorithmic issues rather than having to deal with the implementation of ancillary software. Useful off-the-shelf methods that a research-oriented platform for metaheuristic optimisation should at least have are:

- performance evaluation methods, based on metrics as algorithmic overhead, scalability, best results, average best result, convergence (e.g. in terms of fitness and population diversity [9,10]), structural bias [1,11] etc.;
- statistical methods for comparing the performances of algorithms [12,13];
- results visualisation methods (in both table and graphical formats);

on top of providing

- a vast and heterogeneous number of benchmark functions and real-world testbed problems;
- several state-of-the-art algorithms to be used for comparisons with newly designed optimisation strategies;
- libraries with implementations of the most popular algorithmic operators such as mutation methods, crossover methods, parent and survivor selection mechanisms, heuristics seleection methods for memetic computing (MC) and hyperheuristic approaches, etc;
- utility methods for generating random numbers, solutions or population of solutions, keeping track of the best newly found solutions and save them to plot fitness trends, handling infeasible solutions [1,14,15];
- a system to spread the runs over multiple cores and threads, thus speeding up the data generation process, and automatically collect and process raw data into meaningful information;

and should be based on

- open source software so that researchers can freely use it and build up upon it;
- a simple and flexible structure so that new problems and algorithms can be easily added.

The Stochastic Optimisation Software, henceforth SOS, exhibits the aforementioned key features thus facilitating the design, the evaluation and the use of metaheuristic optimisation algorithms.

Thanks to its simple and extendable structure, SOS can be easily used by researchers to implement new algorithms and compare their performances with those of several other benchmark algorithms. The most recent SOS release, available in the Zenodo repository [16], contains over fifty ready-to-use algorithms amongst single-solution and population-based metaheuristics, as those described in [17], all belonging to established optimisation paradigms as e.g. evolutionary algorithms (EA) [18], swarm intelligence (SI) optimisation [19] and other hybrid schemes. Hence, it provides users with a variety of techniques to explore multiple sectors of metaheuristic/stochastic optimisation. More algorithms will be added in future releases. Researchers can benefit in several ways from having such a large availability of algorithms. First, this means that comparison algorithms do not need to be implemented as they are already present and ready to be executed. Second, their source code is accessible and can be modified to create new variants or simply visioned as a source of inspiration for implementing other algorithms or algorithmic operators. Third, they can coordinate or embedded into other algorithms thus creating novel hybrid methods as done e.g. in the memetic computing (MC) and hyperheuristics fields [3,20–23]. It must be highlighted that implementing algorithms in SOS is quite simple since most algorithmic operators are provided by the platform. Moreover, the "template" class `Algorithm`, which is equipped with several ancillary methods, can simply be extended to implement any metaheuristic, regardless of the optimisation family they belong to. In this light, SOS

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

also represents an excellent pedagogical tool for teaching purposes[1] and can be used to give guidance to students while implementing optimisation algorithms. Researchers willing to collaborate and add code (algorithms, real-world problems, etc.) to the platform are welcome and invited to get in touch to be added to the SOS GitHub repository[2]. Finally, it is worth mentioning that due to the simplicity in adding problems and algorithms, SOS represents a useful tool for practitioners who might have less interest in the algorithmic design but a high need for off-the-shelf implementations of optimisation algorithms.

2. The SOS platform

For portability reasons, SOS is coded in Java, thus being platform-independent. To speed up the execution of extensive experiments, some benchmark suites are also available in the form of C/C++ written native libraries (compiled for MS Windows, but mainly MAC OS and Linux machines) using the Java Native Interface (JNI). By default, SOS spreads the "runs" (i.e. the optimisation processes in which one algorithm optimises one problem) over the available threads and execute them in parallel. This can results in quite CPU consuming processes that can be avoided, e.g. if the experiment is executed with a laptop or personal computer, by switching to the single-thread mode by means of the setter method `setMT(boolean MT)` of the class `Experiment` (i.e. its boolean flag variable must be set to false). By looking at [16], one can notice that in its current release, a strong emphasis is given to real-valued box-constrained single-objective optimisation, aka "single-objective continuous optimisation" in the EC community. Nevertheless, SOS can be used for addressing other optimisation domains such as e.g. discrete or combinatorial ones.

2.1. Functional overview

SOS is research-oriented and is designed to have an intuitive structure making it straightforward to use by both expert algorithms' designers in the research community and, most importantly, by practitioners and doctoral students from different subject fields.

Indeed, to execute an experiment it is sufficient to add a class, which extends the abstract `Experiment` class, in the homonym folder (package). Such class will automatically inherit the `add` methods, that can be used to add algorithms and problems to the experiment, and other methods to set e.g. the allowed computational budget (by default this is assumed to be equal to $5000 \times n$ with n being the dimensionality of the problem) and the number of runs to be performed. Subsequently, a `main` method must be written to launch one or multiple experiments. To do this, the methods of the utility class `RunAndStore` can be imported to e.g.

- execute a list of experiments having different budgets, number of runs, problems and algorithms, or the same experiments but repeated for different dimensional values (if the problems are scalable);
- generate log files describing the performed experiments, (this contains the list of problems, problems and corresponding details such as dimensionality, parameter setting, number of runs, etc.);
- store raw data and process them into results tables and plottable formats as shown in section 5.

Figure 1 shows an example of a main method used for executing a list of experiments and display the final outputs, i.e. tables and graphs, generated by SOS after generating and processing the results. Many examples of experiment and main files are present in SOS and can be used as a template to follow for writing new ones. With reference the latest release in [16], a large number of experiment files can be found in the `experiments` package while several examples of main files (showing different

---

[1] Since 2015 its first releases have been used to teach the master module "Computational Intelligence Optimisation" led by the author of this article at De Montfort University (Leicester, UK).

[2] The GitHub repository link is https://github.com/facaraff/SOS. Useful links to repositories are all displayed in table 1

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

configurations and options for storing and processing results in different formats) are available in the `mains` package. However, the most convenient execution point is the class `RunExperiments`, shown in figure 1, which is located in the default package of the SOS platform. More indications on how to write an experiment class are given in section 2.2 and graphically shown in figure 2.

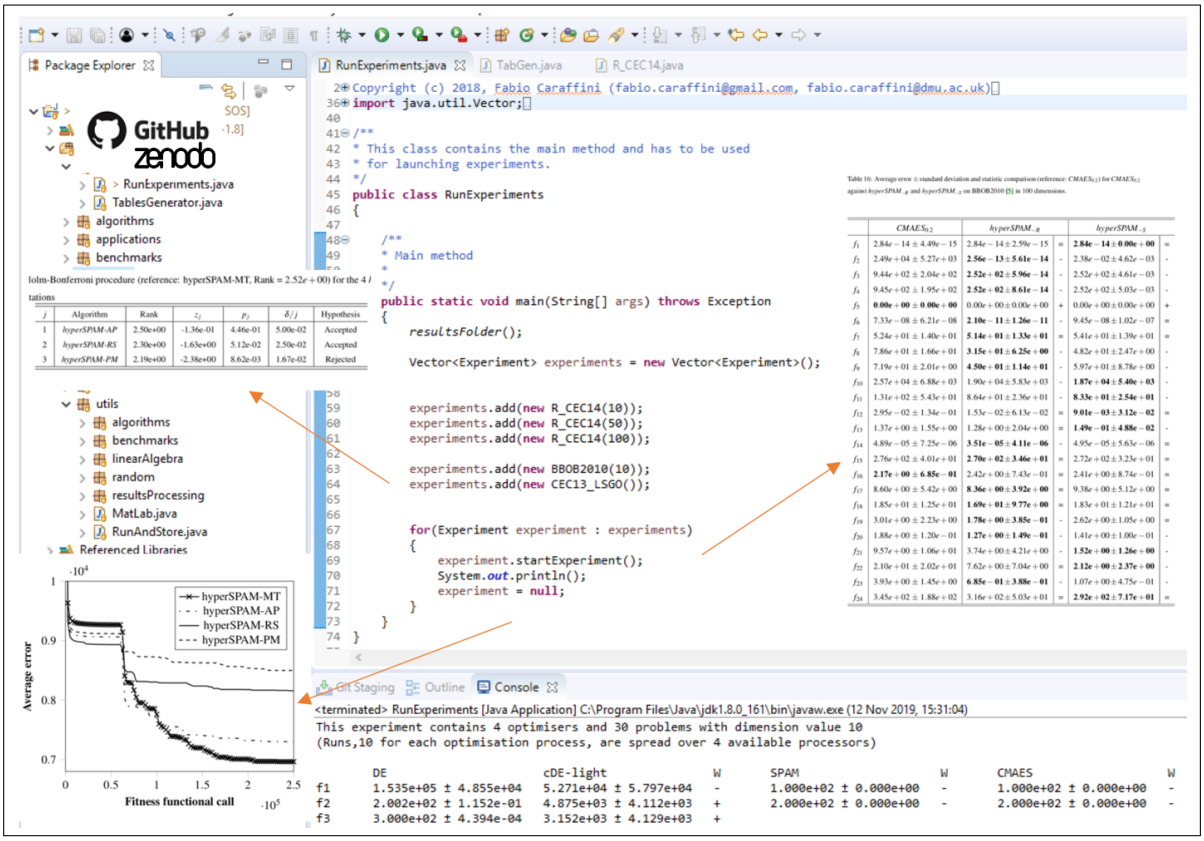


Figure 1. A graphical functional overview of the SOS platform.

To minimise the need for referring to a detailed document, SOS is equipped with a high number of self-explanatory example classes. Moreover, the clear nomenclature used for methods, variable types, class names, and the user-friendly structure of the proposed platform, give guidance to the users, who may not even need further indications. Nonetheless, technical software documentation is made available online to provide users with further information on its packages, classes and methods. Table 1 displays relevant links to SOS webpages, from where SOS documentation can be accessed. These webpages are be kept up-to-date and the documentation itslef is constantly updated to reflect major changes and evolutions of this software platform. This documentation plays a key role for those who intend to modify SOS source code, extend it or customise it to a specific need.

Description	Content	Link
SOS webpage	software documentation	https://sites.google.com/site/facaraff/research/sos
GitHub repository	source code	https://github.com/facaraff/SOS
Zenodo repository [16]	SOS releases	https://doi.org/10.5281/ZENODO.3237023

Table 1. Relevant links to source code and software documentation.

SOS makes it very easy to add new algorithms and new problems which are fully compatible with the platform so to exploit its full capabilities. For the sake of good organisation, it is suggested to place their implementation in the three provided packages: `algorithms`, `benchmarks` and `applications`.

As previously mentioned, the folder `algorithms` currently contains a vast list of optimisation algorithms for continuous optimisation, i.e. over fifty algorithms, some of which can be run with

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

different combination of variation and/or selection operators thus having a higher number of choices. This results in a good balance between established optimisation methods, as e.g. classic differential evolution (DE) [1,24], simulated annealing [25], genetic algorithms, particle swarm optimisation [26], evolutionary strategies algorithms [27–29], and more modern optimisers as e.g the self-adaptive algorithms in [30–37] and the state-of-the-art memetic algorithm (MA), memetic computing (MC) amd hyperheuristic approaches [3,22,23,38,39]. For a complete list, one can download the `algorithms` folder from [16] or check the online documentation. More information on SOS algorithms are given in section 2.3.

The `benchmarks` folder is meant to contain the implementation of problems from established benchmark suites for optimisation. SOS implements several testbed problems and complete benchmarks suites to add in experiment files. A complete list of these optimisation problems and relevant documentation is provided in section 3. Furthermore, a novel benchmark suite is presented in section 3.1.

The `applications` folder is where specific real-world applications should be implemented. Some examples from the literature, e.g. the code for the application described in [22], and from the suite for real-world problems are available in this folder and downloadable from the repositories indicated in table 1.

The `applications` folder is where specific real-world applications should be implemented. Some examples from the literature, e.g. the code for the application described in [22] and for some applications from the suite for real-world problems, are available in this folder.

Following the execution of an experiment, the produced raw data can be processed by means of the classes in the package `utils.resultsProcessing`. These provide several functionalities including

- the execution of statistical tests for comparing stochastic optimisation algorithms (described in section 4);
- the execution of the novel procedure to compare couples of optimisation algorithms presented in section 4.3;
- the generation of plottable files suitable for simple Tikz, Python Matplotlib, Gnuplot or Matlab scripts for displaying best/median/worse/average fitness trends as shown in section 5 and appendix A;
- the generations of files containing further plottable information such as histograms, relative to final bests distribution amongst several runs, and graphs describing infeasibility and structural bias in optimisation algorithms (see [1,15,40,41] for details);
- the generation of LaTeX tables (both source code and PDF files are produced) showing results in terms average fitness value (or average fitness error) with corresponding standard deviations and further statistical evidence as explained in section 4 and graphically shown in in section 5.

The file `TabGen.java`, located in the `Mains` folder, contains the methods for generating the previously mentioned tables and graphical outputs and can be modified to customise the production of tables with different layouts and informative content as discussed in section 5.

It is worth stressing the fact that having a fast way to visualise results in multiple formats is key to performing many research activities. As an example, when writing papers, it is very useful since raw data are automatically processed and the LaTeX source for the comparative tables, to be copied and pasted in the main draft, is generated in a few seconds.

Moreover, this also accelerates the algorithmic design phase, which is often performed empirically through several trial-and-error steps. At each step, different variants of the same algorithm need to be compared and tables have to be generated. This is quite common in the EC filed where using a different combination of variation operators [18] with different combinations of parents and survivors selection schemes [18] can lead to a variety of possible algorithmic behaviours. The same issue arises when designing MC and hyperheuristic algorithms for which several coordination logics have to be tested [3,20,21,23] to ensure high performances. This aspect is quite interesting as it suggests that empirical design approach can only produce algorithms tailored to the problem, or problems, considered during

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

the design phase. To some extents, this problem can be considered dual to the training process in machine learning.

In this light, the algorithmic design process is not very different to the process performed to choose the less disruptive correction method for handling infeasible solutions generated while addressing a problem [1,15,40] and to the fine-tuning process performed to find the most appropriate set of parameters for an algorithm A meant to address specific problem P.

2.2. Parameters fine-tuning

Metaheuristics for optimisation must be tuned on the problem at hand to make them achieve their full potential and return high-quality solutions. Against the original common belief that a "universal" algorithm could have been designed, theoretical achievements such as, first and foremost, the "no free lunch theorems" (NFLTs) for optimisation [42,43] highlighted the need for fine-tuning, self-adaptation and the use of problem-specific operators.

However, finding the optimal parameter configuration is an optimisation problem itself. Despite meta-optimisation strategies have been proposed [44,45], these do not completely solve the problem and are arguable since introducing further complexity while still needing to tune the meta-optimiser. Furthermore, the fact that most real-world optimisation problems are black-box systems, for which no theoretical information on the fitness function is available, increases the difficulty in finding an exact method for finding the optimal configuration of parameters.

Under these circumstances, the parameters tuning process is necessarily performed empirically, thus resulting in a rather time-consuming and tiresome activity. SOS can be used to significantly accelerate and facilitate this task as 1) the same algorithm can be included multiple times in the same experiment file with different parameters configurations; 2) runs can be spread over the available processors and threads to speed up the generation of results; 3) raw data are immediately statistically analysed and results automatically displayed in compact and still highly informative tables and graphs (examples are provided in section 5). Hence, the only portion of code that may be needed to tune an algorithm A on a specific real-world problem P with SOS, is actually the one implementing P.

It must be pointed out that also artificially built testbed problems as those already implemented in SOS, see section 3, can play a role in the parameter tuning process. Indeed, studying how the algorithmic behaviour of a metaheuristic algorithm changes under different configurations of its parameters can help shed light on how to perform the tuning process. By using testbed problems, whose mathematical properties (e.g. differentiability, separability, modality, ill-conditioning etc.) are known, we can understand what the effect of the parameters is when the fitness landscape displays particular mathematical features. As a result, specific regions of the parameters space could be detected to obtain a robust general-purpose behaviour rather then a very problem-specific one, or to strengthen exploitation capabilities rather than exploration capabilities, or to minimise the rise of algorithmic structural biases, as done in [1,11,40], the generation of a high number of infeasible solutions, as done in [1,15] or premature convergence [46] and stagnation [47], as done in [10].

Most of the aforementioned studies have been performed with SOS and their experiment files can be found in [16]. Portions of code from three experiment files are reported in figure 2, which helps understand how the parameters space of an algorithm can be explored in SOS.

In figure 2-(a) four DE variants are added into an experiment named *ExperimentDE* by using the constructor `super(probDim,5000,"ExperimentDE");`. Therefore, all the produced text files containing raw data will be saved in a homonym folder, located inside the SOS default `results` folder. From the constructor method, it can be seen that a maximum computational budget of $5000 \times n$ fitness functional calls, with n being the dimensionality of the problem is used. Clearly, this experient file contains scalable testbed problems and not real-world applications since the dimensionality of the problems is not fixed and passed as an argument with the variable `probDim`. The number of performed runs, 30 in this case, is specified with the command `setNrRuns(30);` (SOS performs 100 runs if not specified otherwise). It must be noted that parameters, apart from the population size, are fixed.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1


```
super(probDim,5000,"ExperimentDE");
setNrRuns(30);

Algorithm a;
Problem p;

a = new DE("ro",'b');
a.setID("DEr1bin");
a.setParameter("p0", (double)probDim);//Population size
a.setParameter("p1", 0.7);//F
a.setParameter("p2", -1.0); //CR
a.setParameter("p3", 0.3);//Alpha
add(a);

a = new DE("ro",'e');
a.setID("DEr1exp");
a.setParameter("p0", (double)probDim);//Population size
a.setParameter("p1", 0.7);//F
a.setParameter("p2", -1.0); //CR
a.setParameter("p3", 0.3);//Alpha
add(a);

a = new DE("ctro");
a.setID("DEctr1");
a.setParameter("p0", (double)probDim);//Population size
a.setParameter("p1", 0.7); //F
a.setParameter("p2", Double.NaN);//CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);

a = new DE("ro",'x');
a.setID("DEr1noxo");
a.setParameter("p0", (double)probDim);//Population size
a.setParameter("p1", 0.7);//F
a.setParameter("p2", Double.NaN); //CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);
```

(a)

```
a = new DE("ro",'e');
a.setID("rDEr1exp01");
a.setParameter("p0", 10.0); //Population size
a.setParameter("p1", 0.4); //F
a.setParameter("p2", 0.1); //CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);

a = new DE("ro",'e');
a.setID("rDEr1exp05");
a.setParameter("p0", 10.0); //Population size
a.setParameter("p1", 0.4); //F
a.setParameter("p2", 0.5); //CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);

a = new DE("ro",'e');
a.setID("rDEr1exp07");
a.setParameter("p0", 10.0); //Population size
a.setParameter("p1", 0.4);//F
a.setParameter("p2", 0.7);//CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);

a = new DE("ro",'e');
a.setID("rDEr1exp09");
a.setParameter("p0", 10.0); //Population size
a.setParameter("p1", 0.4);//F
a.setParameter("p2", 0.9);//CR
a.setParameter("p3", Double.NaN);//Alpha
add(a);

for(int i = 1; i<=30; i++)
        add(new RCEC2014(probDim, i));
```

(b)

```
char[] corrections = {'s','t','m','c'};
String[] DEMutations = {"ro","rt","bo","bt","ctbo","rtbt"};
char[] CrossOvers = {'b','e'};
int[] populationSizes = {5,20,100};
double[] FValues = new double[10];
double[] CRValues = new double[5];

for (int i=0; i<5; i++)
    FValues[i] = 0.05+i*(1.95/9.0);
for (int i=0; i<5; i++)
    CRValues[i] = 0.05 +i*((0.94/4.0));
```

→

```
for (char correction : corrections)
    for (String mutation : DEMutations)
        for(char xover : CrossOvers)
            for(int popSize: populationSizes)
                for (double F : FValues)
                    for (double CR : CRValues)
{
    a = new DE(mutation,xover);
    a.setCorrection(correction);
    a.setParameter("p0", (double)popSize); //Population size
    a.setParameter("p1", F); //F - scale factor
    a.setParameter("p2", CR); //CR - Crossover Ratio
    a.setParameter("p3", Double.NaN); //Alpha
    test.add(a);
    a = null;
}
```

(c)

Figure 2. Examples from experiment classes (original files are downloadable from [16]) for parameter tuning and algorithmic behaviour testing, i.e. the same algorithms are included in the same experiment under different parameters configurations. The fragment of code reported in (a) refers to an experiment performed for the studies in [4,48] located in `SOS→src→experiments→rotInvStudy→CEC11.java`. The file `RCEC14TuningDEroe.java`, from which the segment of code in (b) is taken, can be found in the same folder. In this case, the `DE/rand/1/exp` algorithm is executes with three different values for the crossover rate parameter C_r over the 30 rotated versions of the problems of the R-CEC14 benchmark suite proposed in this article in section 3.1. Finally, the fragment of code in (c) shows how to compactly define an experiment running 12 DE variants with several different parameters configurations, over one problem only, to find the most appropriate triplet $\langle$population size, scale factor (F), crossover ratio (C_r)$\rangle$. The complete class file for this example is located in `SOS→src→mains→ExampleTuning.java`.

This means that this experiment will only study the effect of varying population sizes, which are in this case proportionate to the problem dimension as required for the studies in [4,48] which can be

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

read for further details. To execute this experiment with increasing problem dimensionalities, and therefore, in turn, increasing population sizes, it is sufficient to instantiate objects with increasing `probDim` values by calling the class constructor in the file `RunExperiments.java` (as shown in the example of figure 1). It is interesting to note that, to avoid confusion, a name is assigned to each DE variant (e.g. `a.setID("DErn1bin")`). The assigned name will show up in the generated tables. If a name is not provided, as in the example of figure 2.2-(c), it will be automatically assigned equal to the class name of the launched algorithm. Therefore, in this case, the assinged names would start with DE, followed by "-i" where "i" represents a counter incremented every time an algorithm is added in an experiment. In a nutshell, if names were not specified in the example of figure 2-(a), tables would display DE-1, DE-2, DE-3 and DE-4, since all the algorithms in this experiment are instances of DE class.

Differently, in figure 2-(b) the same DE variant, namely DE/rand/1/exp, is added three times in the experiment with three different values for the so-called "crossover ratio" parameter C_r. As can be seen at the bottom of the figure, all the 30 rotated problems from the benchmark suite proposed in section 3.1 are added to this experiment. Hence, its purpose is to understand the impact of the C_r value on DE/rand/1/exp when facing different problems.

Finally, figure 2-(c) shows a very compact and fast way for writing a large experiment file in SOS, in which 12 DE variants are equipped with 4 different correction strategies for handling infeasible solutions, for a total of 48 different algorithms to tune. From the left-hand side of the figure it can be seen that the DE variants are obtained by combining the 6 mutations with the 2 crossover operators in the `DEMutations` and `CrosOvers` arrays respectively. The 4 correction strategies are in the `corrections` array. For details about the adopted notation for DE variants and correction strategies one can consult the articles [1,11] and the results in [41]. For each algorithm, the explored parameters space is defined as the Cartesian product of the three sets represented by the `populationSizes` array, containing 3 population sizes, the `FValues` array, containing 10 equally spaced values for the so-called "scale factor" F in the range $[0.05, 2]$, and the `CRValues` array, containing 5 equally spaced values for C_r in the range $[0.05, 0.99]$. Hence, each one of the 48 algorithms is tuned over a set of 150 possible combinations of the three parameters. In total, this means that $7200 \times N_r$ optimisation processes are executed, with N_r being the number of performed runs, for each problem added to this experiment. The full code for this examples, named `ExampleTuning`, is located in the `mains` package. For demonstration purposes, $N_r = 10$ and the computation budget is kept short, i.e. $1000 \times n$. In a real tuning experiment, this can be increased as required.

2.3. Adding new algorithms and problems

The literature is currently saturated by novel nature-inspired optimisation metaphors claiming to mimic the collaborative or individual behaviour animals [49–56], human activities [57–59] or other natural phenomena [60–62]. The contribution made by most of these new optimisation paradigms is arguable as they are very similar to established frameworks form the field of EA, such as the GA, the DE and the ES ones, and from the field of SI as e.g. the PSO algorithm. Furthermore, all of these new metaheuristics are subject to the NFLTs and cannot outperform classic ones over all possible black-box problems without fine-tuning. It must be remarked that also amongst the most established optimisation frameworks some similarities can be detected in support of the thesis that taxonomies based on metaphor inspiring their design could be indeed avoided. To provide some examples, it is sufficient to observe that DE mutations, being linear combinations of individuals from the population [1,24], operate the same as arithmetic crossovers in real-valued GAs and other EAs [18]. Similarly, it can be observed that the lack of parent selection in SI frameworks, as e.g. the PSO [26], is simply moved in the perturbation operator since requiring the definition of the neighbourhood of the candidate solution to b perturbed to find its *local best* solution.

In this light, a scientific approach to describe a metaheuristic for optimisation is by considering it as a stochastic process returning a near-optimal solution for an optimisation problem. Regardless

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

of the metaphor behind them, all metaheuristic methods are the expression of the same concept and aim at reaching a good balance between *exploration* of the fitness landscape, looking for promising basins of attraction, and their *exploitation*, to refine them and find local optimal points (i.e. maxima or minima according to the need). This dynamic process can alternate such phases as appropriate to keep looking for the global optimum, without prematurely converging to a locally optimal point or stagnating without being able to return any satisfactory near-optimal solution.

Hence, it makes sense to analyse the algorithmic structure of modern nature-inspired optimisation algorithms and extrapolate a common, high level, general skeleton to use as a template for their implementation. This is done in SOS by means of the abstract class `Algorithm`, from which all algorithms *inherit* ancillary and auxiliary methods (e.g. for algorithm execution, storing the fitness trend, setting or generating the initial guess, etc.) and *extend* only the parts to be customised to implement the specific optimisation framework (e.g. a GA rather than a DE, a hybrid method or a completely new optimisation paradigm). Therefore, any class file extending `Algorithm` is already equipped with the required methods to be added into an experiment file and be executed as described in the previous section, while it must implement one method, named `execute`, which returns an object of the kind `FTrend`). To give further details in this regards, other than referring to online software documentation, it is worth briefing about the classes of the package `utils` listed below.

- `RunAndStore`: this class contains the implementation of all methods involved in executing an algorithm in single or multi-thread modes, collecting the results, display them on a console (unless differently specified) and saving them into text files.
- `RunAndStore.FTrend`: this class instantiate auxiliary objects storing information collected during the optimisation process and that must be returned by all the algorithms implemented in SOS. As shown in figure 3-(b), the primary purpose of these objects is to store the fitness function evaluation counter and the corresponding fitness value in order to be able to retrieve the best (max or min according to the specific optimisation problem) and plot a so-called fitness trend graphs like those in figure 1 and in figure 10 of section 5. The class provides numerous auxiliary method to return the initial guess, the best fitness value, the median value, and so on. Obviously, the optioned near-optimal solution must be also stored and, if needed, several other extra pieces of information can be added in the form of strings, `integer` or `double` values. As an example, in [10], population and fitness diversity measures were collected during the optimisation process.
- `MatLab`: This class provides several methods to quickly initialise and manipulate matrices. These methods include operations between matrices but, most importantly, the auxiliary methods for generating matrices of indices, for making copies of other matrices and decomposing them.
- `Counter`: This is an auxiliary class to handle counters when several of them are required to e.g. count fitness evaluations, count successful and unsuccessfully steps, etc. In [1,15] this class is used to keep track of the number of generated infeasible solutions how many times the pseudo-random number generator is called in a single run. It must be remarked that for obtaining this information in a facilitated way it is necessary to extend the abstract class `algorithmsISB` rather than `algorithms` [16].

Moreover, the packages `utils.algorithms` and `utils.random` provides very useful methods helping users to implement algorithms. These packages contain:

- utility methods for cloning and generating individuals and population in the search space for population-based, single-solution and estimation of distribution algorithms (with a specific package for compact optimisation [63]);
- utility methods for measuring population and fitness diversity according multiple metrics [10] and for manipulating populations into covariance matrices [64,65];
- the class `Corrections` with the implementation of several correction strategies to handle infeasible solutions, as those from [1,15], which are selected by simply setting a `char` flag of the the class `Algorithm` as shown in figure 2-(c);

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

- the `random` utilities, which provides methods for generating random numbers from different distributions, initialising random (integer and real-valued) arrays, permuting the components of an array passed as input, changing the pseudo-random number generator, changing seed, etc.;
- a vast number of algorithmic operators (sub-package `operators`) implementing simple local search routines, restart mechanisms and most importantly
  - the class `GAOp`, which provides numerous mutation, parent selection, survivor selection and crossover operators from the GA literature;
  - the class `DEOp`, which provides all the established DE mutation and crossover strategies plus several others from the recent literature;
  - the classes in aos, which implements adaptive heuristic/operator selection mechanisms from the hyperheuristic field [23].

Given the facility in adding new algorithms and the availability of numerous off-the-shelf operators, SOS is a suitable workbench to design hybrid algorithms or variants tailored to the problem at hand. In particular, the possibility of declaring variables of the kind `Algorithm`, to initiate and execute optimisation processes inside another algorithm, lead to the implementation of complex algorithms by writing very neat code. This is evident from figure 3, where the code of the iterated local search algorithm proposed in [66] is shown. At first glance, one can observe that the source code is clear and resembles pseudocode. Indeed, instead of implementing the sophisticated (1+1)–CMA-ES algorithm in [67], which plays the local searcher role, an `Algorithm` variable is instantiated by using the `CMAES_11` class available from the `algorithms` package and implementing the (1+1)–CMA-ES algorithm. Parameters and computational budget for this algorithm, which can be seen as an operator in this context, are passed as described in the previous sections. It can be noticed that the initial point for each iterated search is passed to the local searcher before it is executed. This point is generated by applying the exponential crossover operator [1] to the current best solution and a feasible randomly sampled individual. Obviously, the crossover method as well as the method for generating an individual in the search space are already present in SOS and therefore are simply imported and used, without the need of writing further code. To monitor the internal dynamic of the resulting algorithm, a `FTrend` object can be used as commented in figure 3.

Based on similar considerations, also optimisation problems are added to SOS by following a template defined in the abstract class `Problem`. Indeed, regardless their nature, i.e. black-box, grey-box, real-world or synthetic testbed, all optimisation problems share similar attributes indicating their dimensional (or the number of design variables), the boundaries delimiting the search space in which the optimal solution must be found, an optional name to describe the problem, which is accessed and changed with *setters* and *getters* method already implemented in `Problem`. This way, no coding is required to deal with such aspects thus allowing SOS users to focus on writing the body of only one abstract method "`f`" which obviously represent the fitness function. When the `execute` method of an algorithm is called, a reference to a `Problem` variable `P` is passed to the algorithm which evaluate the fitness value `y` with the code line `y = P.f(x)` with `x` being a candidate solution. To ensure that the return value is meaningful, algorithms inherit from the superclass `Algorithm` the function `x = correct(x,bounds)` that can be used before performing the fitness function evaluation. The latter executes the correction strategy specified when the algorithm object is instantiated (the default strategy is the toroidal correction [1]). The `Problem` class comes with multiple constructors so that problems can be instantiated in different ways. Usually, real-world applications have a fixed number of design variables and fixed boundaries while benchmark functions are expected to be scalable and with adjustable search space boundaries. Thus, being able to select the most appropriate constructor is very convenient. Finally, it is worth reminding that the methods from the `MatLab` class can aid the implementation of a novel problem and that, if a benchmark function displaying particular features is needed, its implementation might already be available amongst those indicated in section 3

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1


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

int i;

double[] best;
double fBest;

FTrend FT = new FTrend();

i = 0;
if (initialSolution != null)
{
    best = initialSolution;
    fBest = initialFitness;
}
else
{
    best = generateRandomSolution(bounds, problemDimension);
    fBest = problem.f(best);
    i++;
}
FT.add(i, fBest);

// INITIALISE MEMES//

double globalCR = getParameter("p0").doubleValue();

CMAES_11 cma11 = new CMAES_11();
cma11.setParameter("p0",getParameter("p1").doubleValue());
cma11.setParameter("p1",getParameter("p2").doubleValue());
cma11.setParameter("p2",getParameter("p3").doubleValue());
cma11.setParameter("p3",getParameter("p4").doubleValue());


double maxB = getParameter("p5").doubleValue();

FTrend ft = null;
```

(a) Initialisation phase

```java
while (i < maxEvaluations)
{
    xTemp = generateRandomSolution(bounds, problemDimension);
    xTemp = crossOverExp(best, xTemp, globalCR);
    fTemp = problem.f(xTemp);
    i++;
    if(fTemp<fBest)
    {
        fBest = fTemp;
        for(int n=0;n<problemDimension; n++)
            best[n] = xTemp[n];
        FT.add(i, fBest);
    }


        cma11.setInitialSolution(xTemp);
        cma11.setInitialFitness(fTemp);

        int  budget = (int)(min(maxB*maxEvaluations, maxEvaluations-i));

        ft = cma11.execute(problem, budget);
        xTemp = cma11.getFinalBest();
        fTemp = ft.getLastF();

        FT.merge(ft, i);
        i+=budget;

        if(fTemp<fBest)
        {
            fBest = fTemp;
            for(int n=0;n<problemDimension; n++)
                best[n] = xTemp[n];
        }

}

FT.add(i,fBest);
finalBest = best;
return FT;
```

(b) Optimisation phase

Figure 3. Algorithmic design and implementation in SOS. This example shows portions of code from the algorithm class `RI1p1CAMES`, located in the `algorithms` package, which implements the algorithm proposed in [66]. On the left-hand side (a), the initialisation phase, where the algorithm `cma11` of the king `CMAES_11` (which extends `Algortithms`) is initialised and ready to be executed inside another class extending `Algortithms`. On the right-hand side (b), the implementation of an iterated local search method using `cma11` as a local. The `FTrend` variable `ft` returned by `cames11` is automatically merged to the one of the main algorithm, i.e. `FT`, to obtain a unique global fitness trend.

3. Benchmarking with SOS

Due to the difficulties in dealing with real-world black-box problems, the metaheuristic optimisation research community started developing artificially built functions [68] to

- significantly reduce the optimisation time;
- investigate algorithmic behaviours over problems with known or partially known characteristics and properties;
- be able to empirically study the scalability of optimisation algorithms.

Since the publication of the first testbed problems [68], several and ever more challenging benchmark suites have been released on an annual basis. These usually consist of heterogeneous groups of function displaying similar features in terms of modality, separability and ill-conditioning. Despite the high number of suites in the literature, their technical reports show similarities as a set of established functions, as e.g. Michalewicz, Ackley, Rastrigin, De Jong and Schwefel functions (see technical reports listed in the remainder of this section for details) are always present. Hence, many of them were basically equivalent if it was not for the recent tendency to increase the degree of difficulty by also including hybrid functions, obtained by combining or composing the aforementioned basic ones or by shifting and rotating the basic cases. Even though the utility of having (often over-complicated) composition of functions, which are sometimes as cryptic as black-box real-word problems, these are constantly been employed to propose challenging competitions for stochastic optimisation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

To remove the burden of implementing such a high number of testbed problems, the SOS platform comes with full implementations of:

- several basic testbed problems as those in [20];
- several complete benchmark suites amongst those released over the years for competitions on real-parameter optimisation at the *IEEE congress on evolutionary computation* (CEC), namely
  - CEC 2005 [69];
  - CEC 2008 (for Large Scale Global Optimisation) [70];
  - CEC 2010 (for Large Scale Global Optimisation) [71];
  - CEC 2013 [72];
  - CEC 2013 LSGO (for Large Scale Global Optimisation) [73];
  - CEC 2014 [2];
  - CEC 2015 [74];
- the fully scalable test suite for the *special issue of soft computing* (SSCI) 2010 on *scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems* [75];
- a stand-alone implementation of the *black-box optimization benchmarking* (BBOB) suite [76] as well as all the BBOB suites included in the 2019 release of the COCO platform [6] (which is internally called by SOS);
- a novel variant of the CEC 2014 benchmark, named "R-CEC14"[3], presented in section 3.1.

It is worth indicating that some applications from the CEC 2011 benchmark suite for real world optimisation [77] are implemented in the `applications` package.

3.1. The R-CEC14 benchmark suite

The original IEEE CEC 2014 benchmark suite [2] consists of 30 challenging functions for single objective real-parameter numerical optimisation. These problems are obtained by shifting, rotating and ill-conditioning well established classical functions in the fields of computational optimisation. Only 2 of the 30 mathematical functions are not subject to rotation, i.e. function number 8 (Shifted Rastrigin's Function) and function number 10 (Shifted Schwefel's Function), but do have a rotated counterpart in the benchmark suite.

For this reason, one could argue that only these 4 functions, i.e. number 8, 10 and the 2 rotated versions, are insufficient to draw interesting conclusions on

- the efficacy of so-called rotation invariant operators in optimising rotated and unrotated landscapes, as done in [48] and [4] for DE, which unveiled no differences in the performances of such operators over the two classes of problems[4];
- separability and degrees of separability, measured e.g. with the separability index proposed in [3], as well as on the suitability of certain algorithms and algorithmic operators for addressing separable and non-separable problems.

Indeed, rotating a separable function does not alter its modality or ill-conditioning features, but does alter its separability. In this light, while working on the study in [48] I redesigned the original CEC 2014 and implemented a Java version in which the rotation operator can be activated and deactivated by simply setting the integer rotation (R) flag. If equal to 1, the rotation operator is active and the functions are identical to those of CEC 2014. Conversely, if null, no rotation takes place. To avoid duplicates, functions 8 and 10 are removed since obtainable from their rotated counterparts, i.e. functions 9 and 11, by setting the rotation flag equal to 0.

---

[3] The R indicates a rotation flag used to activate or deactivate the rotation operator.

[4] Indeed, from the point of view of the algorithm these are just different problems. Hence, the need for rotating benchmark functions, if not for transforming separable functions into non-separable ones, can be argued [4,48].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

Thus, the resulting R-CEC14 suite contains 56 functions:

- the first 28 are the functions listed in table 2, whose analytical expressions can be found in [2];
- the last 28 functions are their rotated versions.

These problems are optimised within an hypercubical search space $\mathcal{D}$ defined as $[-100, 100]^n$, with $n \in [10, 30, 50, 100]$ being the admissible dimensionality values. For each problem, the corresponding $n \times n$ rotation matrices are stored in the `benchmarks.problemsImplementation.CEC2014.files_cec2014` package [16] and loaded when the rotation operator is active. The minimum fitness function value $f_{min} = \min_{x \in \mathcal{D}} f(x) = f(x_{min})$, with $x_{min} = \operatorname{argmin}_{x \in \mathcal{D}} f(x)$, is shown in table 2 and more detailed information on the functions can be found in [2] or by controlling the source code [16] and SOS online documentation.

f class	f number	Function Name & Descirption	f_{min}
Unimodal	1	High Conditioned Elliptic Function	100
	2	Bent Cigar Function	200
	3	Discus function	300
Multimodal	4	Shifted Ackley's Function	400
	5	Shifted Rosenbrock's	500
	6	Shifted Griewank's Function	600
	7	Shifted Weierstrass Function	700
	8	Shifted Rastrigin's	900
	9	Shifted Schwefel's	1100
	10	Shifted Katsuura	1200
	11	Shifted HappyCat	1300
	12	Shifted HGBat	1400
	13	Shifted Expanded Griewank's plus Rosenbrock's	1500
	14	Shifted and Expanded Scaffer's F6 Function	1600
Hybrid	15	Hybrid Function 1	1700
	16	Hybrid Function 2	1800
	17	Hybrid Function 3	1900
	18	Hybrid Function 4	2000
	19	Hybrid Function 5	2100
	20	Hybrid Function 6	2200
Hybrid	21	Composition Function 1	2300
	22	Composition Function 2	2400
	23	Composition Function 3	2500
	24	Composition Function 4	2600
	25	Composition Function 5	2700
	26	Composition Function 6	2800
	27	Composition Function 7	2900
	28	Composition Function 8	3000

Table 2. R-CEC14 problems list. Each problem can be evaluated with and without the action of the rotation operator. Technical details on these problems and the rotation procedure are available at [2].

The easiest way to use the R-CEC14 benchmark suite is through SOS, but relevant code can also be taken from the `benchmarks` folder to be adapted to other platforms since SOS is an *open source* project and source code can be freely used, extended and modified as long as credit is given to this article, as well as other online resources (see table 1), and by keeping the information reported in the disclaimer of class files.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

4. Statistical analysis with SOS

Stochastic algorithms can be evaluated e.g. by measuring their overhead[5], by calculating their time and memory complexity, in terms of scalability, average performances (i.e. final fitness value returned averaged over multiple runs) and, qualitatively, by visual inspection of fitness trend graphs. However, to be able and claim that an algorithm is capable of outperforming one, ore more algorithms, when tested on a specific problem, or a set of multiple problems, statistical evidence must be sought.

A review of potential statistical tests to be used for analysing and comparing stochastic algorithms was published in [12]. Amongst the suggested methods, *non-parametric* tests such as the Holm test [78] and the Wilcoxon Signed-Ranks test [79] soon became the most employed since results collected over multiple runs are not necessarily normally distributed. Several recent studies adopted similar variants of these methods, namely the Wilcoxon Rank-Sum test and the Holm-Bonferroni test, which can now be considered quite established in the field [3,4,23,39].

To facilitate the use established and advanced statistical tests for evaluating performances of stochastic optimisation algorithms, SOS provides implementations of

- a comparison test based on the Wilcoxon Rank-Sum test as described in section 4.1
- a comparison test based on the Holm-Bonferroni test as described in section 4.2;
- the Advanced Statistical Analysis (ASA) test proposed in section 4.3.

4.1. The Wilcoxon Rank-Sum test for stochastic optimisation

The Wilcoxon Rank-Sum test [80], aka Mann–Whitney U test [81], is a non-parametric test used to understand whether two independent samples belong to populations having the same distribution. Unlike similar parametric counterparts, as e.g. the popular two-sample unpaired t-test [82], it does not operate on data but scores, referred to as *ranks*, associated to the acual values and for which assumptions on their distribution can be made. This makes it suitable for analysing results of stochastic algorithms, whose distributions may significantly vary over different problems and combinations of parameters, while their ranks distributions can be assumed to follow a normal model.

To describe the procedure implemented in SOS, let us consider two generic algorithms A and B run on a generic problem P for n_A and n_B times respectively[6]. The *null-Hypothesis* is then formulated as $H_0 : A = B$ which means that, regardless of their working logic, the two algorithms are statistically identical, i.e. they are instances of the same population as the solutions provided over multiple runs are equally distributed. This is graphically shown in figure 4. The Wilcoxon Rank-Sum test can then be used to produce evidence to reject H_0. Failing this task would instead support the so-called alternative hypothesis, which is formulated as $H_1 : A < B$ or $H_1 : A > B$, if a one-sided (aka one-tailed) test is performed, or $H_1 : A \neq B$, if a two-sided (aka two-tailed) test is performed.

By default, SOS performs the two-tailed variant. However, both the one-sided variants and the two-sided variant are implemented and usable. Indications on how to run this test in SOS are in the online documentation and an example is also displayed in figure 7. Regardless of the used variant, i.e. single or two sided, the null hypothesis has always the same formulation and can be either accepted or rejected as shown in figure 4. Conclusions can then be drawn from the outcome of the test in terms of performances for A and B. For example, if $H_0 : A = B$ is rejected in a two-sided execution of the test, indicators such as the average fitness value or the median fitness value over the performed runs can be used to understand which algorithm outperform the other on the problem P. Obviously, this is not needed the two one-sided variants are considered.

The steps in sections 4.1.1 to 4.1.5 describe the decision-making process implemented in the SOS platform.

---

[5] In SOS the class `mains.test.TestOverhead` can be used for this purpose.

[6] Usually, $n_A = n_B$ but having an unbalanced number of runs would not prevent one from using the Wilcoxon Rank-Sum test.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

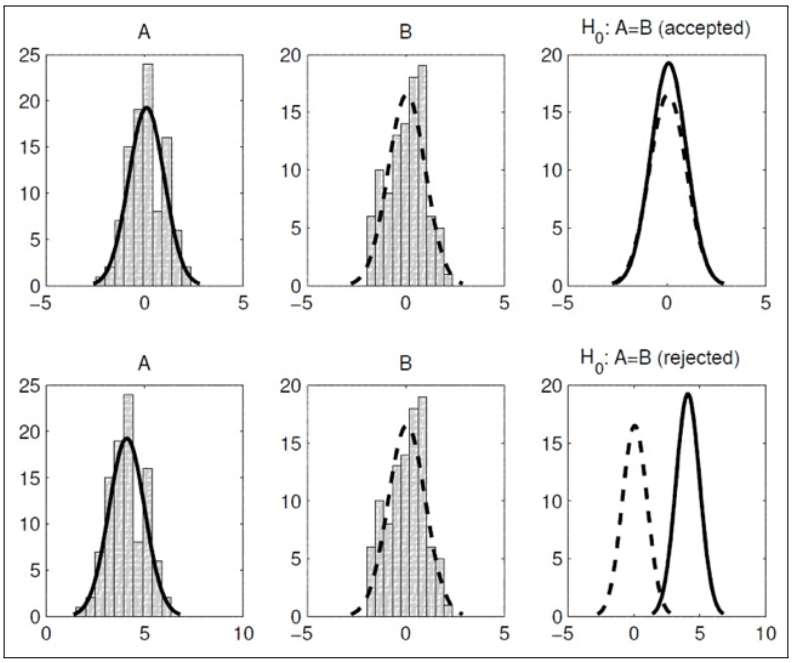


Figure 4. Hypothesis testing. On the top row, the the test fails at rejecting the null-hypothesis (i.e. H_0 is accepted) as the algorithms A and B have significantly similar distributions. On the bottom row, H_0 is rejected as A and B are two different stochastic processes with significantly different distributions.

4.1.1. Reference and comparison algorithms

To be perform a statistical test, a "reference" is needed. When asked to perform the Wilcoxon Rank-Sum test on the results generated with an experiment E, the list of algorithms executed in E is shown and the reference can be indicated. If not specified, SOS assumes that the reference algorithm is the first that was added in E. Without a loss of generality, let us conceptualise A as the reference algorithm. The remaining algorithms in E will form a set of "comparison" algorithms. If more than two comparison algorithms are present, SOS will iteratively schedule a Wilcoxon Rank-Sum test to compare the reference algorithm A with each comparison algorithm B, taken from such set, for each problem P in E. The selection order of each comparison algorithm can be specified. This will be the order of appearance for the algorithms on the automatically generated result table, as those shown in the graphical examples included on section 5. By pressing "c" during the selection process, a table will be created even though the comparison set is not empty. This way, multiple and smaller results tables can be generated from a single experiment. If the "a" (i.e. all) option is insted used, a single long table for the whole experiment E will be generated. This will containing as many columns as algorithms in E and as many rows as problems in E.

4.1.2. Assigning ranks

Let us consider the totality of the observations obtained from the $N = n_A + n_B$ runs. Without a loss of generality, let us refer to a minimisation problem, for which the lower the fitness value the better the algorithm's performance. Observations are then sorted with ascending order to assign ranks $r_i = i$ (with $i = 1, 2, 3, \ldots, N$) so that the smallest value has rank 1 (i.e.$r_1 = 1$), the second smallest value has rank 2 (i.e. $r_2 = 2$), and so on to get to the observation with the greatest value, having a rank $r_N = N$).

This process need to be slightly modified if dataset presents "ties", i.e. observations with identic numerical value, for which SOS will assign the same rank by computing the average of their position index i in the ordered sequence. This have a better representation of the distribution of the original results. To clarify this case with a numerical example, let us suppose that four observations have ordinal ranks 4, 5, 6, and 7 but equal fitness value. To make sure the rank distribution would be a

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

good representation of the real distribution, these four ties will be assigned with the same rank value $r_j = \frac{4+5+6+7}{4} = 5.5$ ($j \in \{4,5,6,7\}$).

4.1.3. The rank distributions

Let us indicate with W_A the random variable associated with A. Its distribution of probability is tabulated only for small sample sizes [83], i.e. $N < 20$, which is very small and unusual for an optimisation experiments. Hence, SOS must implement such distribution to deal with larger sample sizes and get more accurate evaluations results. To do this, see [80,83], it is sufficient to define the normal distribution $\mathcal{N}(\mu_A, \sigma_A)$ whose mean value μ_A and standard deviation σ_A are calculated with

$$\mu_A = \sqrt{n_A n_B \frac{N+1}{12}} \quad \text{and} \quad \sigma_A = n_A \frac{N+1}{2}$$

respectively.

4.1.4. Wilcoxon rank-sum statistic and p-value

Let us fill a set R_A with the n_A ranks r_i associated to the observations from the reference algorithm. The so-called Wilcoxon rank-sum statistic w_A is calculated by summing all the values as indicated below

$$w_A = \sum_{r \in R_A} r$$

and use it to define the p-value. For the default case, i.e. two-sided analysis with $H_0 : A = B$ versus $H_1 : A \neq B$, this is formulated as follows

$$\text{p-value} = \begin{cases} 2 \cdot \text{Prob}\{W_A \geq w_A\} & \text{if } w_A > \mu_A \quad \text{(i.e. } w_A \text{ is in the upper tail)} \\ 2 \cdot \text{Prob}\{W_A \leq w_A\} & \text{otherwise} \quad \text{(i.e. } w_A \text{ is in the lower tail)} \end{cases}$$

where the probability of falling into the tail of the distribution closest to w_A is doubled in order to consider both the cases in which the two algorithms differ because $A > B$ and $A < B$ simultaneously. This is not needed for the one-sided, for which the null hypothesis $H_0 : A = B$ is either tested against the alternative hypothesis $H_1 : A < B$, with a corresponding p-value $= \text{Prob}\{W_A \leq w_A\}$, against the alternative hypothesis $H_1 : A > B$, with a correspond p-value $= \text{Prob}\{W_A \geq w_A\}$. A graphical example is given in figure 5

The p-value gives us an indication the truthfulness of H_0 by calculating the probability of obtaining test results similar to those those observed experimentally, assuming that H_0 is correct. This probability can be used to decide weather or not the null hypothesis can be trusted to be true, or it has to be rejected. In this light, its numerical value must be calculated. This can be done by numerically integrating the test statistic for the specific test, in this case $\mathcal{N}(\mu_A, \sigma_A)$ with μ_A and σ_A as indicated in section 4.1.3, by following the appropriate definitions of p-value for the specific test, i.e. one or two-sided, as explained in this section.

4.1.5. Decision-making

To understand whether or not H_0 has to be rejected, the calculated p-value is compared to a threshold α commonly referred to as *significance level*. This value represents the probability of making a so-called "Type I" error, which occurs with the incorrect decision of rejecting a true null-hypothesis, as in indicated in table 3.

The α value is arbitrarily chosen. Usually, this is a small number amongst 0.10 (1 Type I error chance in 10 decisions is tolerated), 0.05 (1 Type I error chance in 20 decisions is tolerated), and 0.01 (1 Type I error chance in 100 decisions is tolerated). It is worth mentioning that if a decision is made with

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

a probability α of making a Type I error, its correctness can be trusted with a probability of $1 - \alpha$. This figure is referred to as *confidence level*, and it is sometimes provided instead of α in the literature.

	H_0 is true	**H_0 is false**
Test rejects H_0	Type I error (False Positive)	Correct inference (True Positive)
Test fails to rejects H_0	Correct inference (False Negative)	Type II error (True Negative)

Table 3. Table of truth, aka confusion matrix.

By default, SOS performs the Wilcoxon Rank-Sum test with $\alpha = 0.05$, which is the most used value, unless differently specified before running the test. This results in a confidence level of 95%. To employ a different α value, a better method is provided, see the online documentation, and other are also available to switch from two-sided to one-sided and decide whether or not p-values have to be shown on screen.

To conclude, once the p-value is computed and the significance level α chosen, a final decision is made by mean of the following logic:

- if p-value $\geq \alpha$, the test fails at rejecting $H_0 : A = B$ and does unveil any significant difference between A and B – the two algorithm are equivalent on P;
- if p-value $< \alpha$, the test reject $H_0 : A = B$ and one can be $(1 - \alpha)$ % confident that a significant difference does exist between A and B.

A graphical example summarising this process is provided in figure 5.

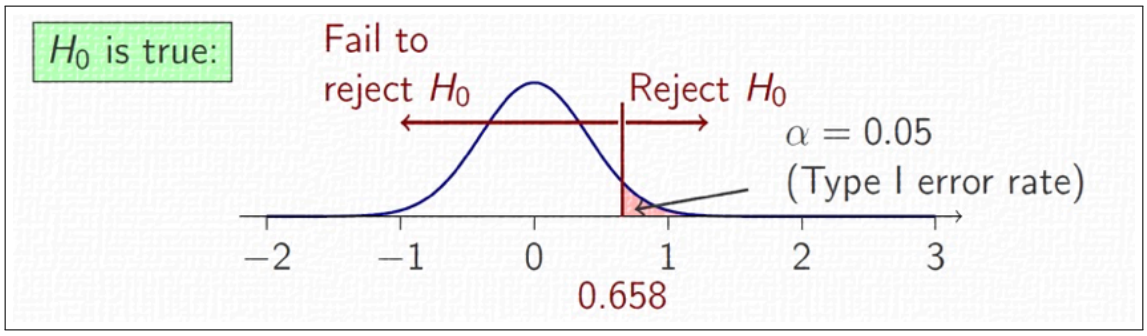


Figure 5. A graphical explanation of the hypothesis testing process for a generic one-sided test with alternative hypothesis $H_1 : A > B$ and $\alpha = 0.05$. The area under the normal distribution, highlighted in red, is equal to α and indicates the *rejection zone*. Indeed, any $x > 0.658$ would return a p-value (i.e. area under the distribution) lower than α, thus rejecting H_0. Conversely, for all $x < 0.658$ the corresponding p-value would be greater than α, thus failing to reject H_0. In a two-sided equivalent scenario, the red area on the right-hand side should have a symmetric counterpart on the left-hand side. The two resulting tails, each one casting an area of 0.025 (i.e. $\frac{\alpha}{2} = \frac{0.05}{2}$, so that the total significance level is 0.05 and the corresponding confidence is $1 - \alpha = 95\%$) would form the rejection zone.

4.2. The Holm-Bonferroni test for metaheuristic optimisation

The Holm test is a non-parametric and sequentially-rejective procedure for multiple-hypothesis testing which aims at rejecting a hypothesis at a time until no further rejection is possible [78].

In the optimisation field, this test can be used to compare the reference algorithm with more than one comparison algorithm over multiple problems simultaneously. This provides a different and more global view than the one provided by the Wilncoxon Runk-Sum test. In this light, the two tests complement each other and it is suggested to always use both, or two similar equivalent tests, to statistically analyse results obtained with empirical experimentation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

The original Holm test was designed with the intent of guaranteeing a reasonably low family-wise error rate (FWER) [12,78] – an error plaguing multiple-hypothesis testing and known to increase when the number of hypotheses increases. To further improve upon this aspect and keep the FWER low even when the number of hypotheses is high, several "corrections" for obtaining more informed p-values have been proposed. The Bonferroni's corrections are among the most popular [84].

SOS implements a simple Holm-Bonferroni test for comparing stochastic algorithms consisting of the steps in sections 4.2.1 to 4.2.4.

4.2.1. Choosing the reference algorithm

Let us consider a generic experiment E containing NA algorithms and NP problems. When the test is run on E, SOS will display the list of available algorithms and ask the user to select the reference algorithm.

Unlike the case described in section 4.1.1 for the Wilcoxon Rank-Sum test, this step can change the output of the test. Indeed, in the one-to-one comparison performed with a Wilcoxon Rank-Sum test, exchanging A with B would not alter the rank distributions obtained on P. Conversely, $NA - 1$ test statistics are computed in a Holm-Bonferroni test and these values do depend on the rank of the reference algorithm. In this light, if the goal is to test the performance of an algorithm A in E against the other algorithms, A should play the role of the reference algorithm. On the contrary, if the goal is to understand which algorithm has the best overall performance in E, it could be necessary to run the test twice. The first time, a random reference is chosen. The second time, the algorithm with the highest rank must be individuated and selected to be the reference for a second round of the test.

Once a reference algorithm is selected, value SOS proceeds with the test.

4.2.2. Assigning ranks

First, the average final fitness value must be computed with values returned by the NA algorithms after the execution of multiple runs. SOS automatically collects the text files containing the information stored in `FTrend` objects and provides the NP final average values for each algorithm[7].

Then,

- for each problem in E a score equal to NA is assigned to the algorithm displaying the best average performance in terms of its objective function value, i.e. the greatest if it is a maximisation problem or the smallest if it is a minimisation one, while a score equal to $NA - 1$ is assigned to the second-best algorithm, a score equal to $NA - 2$ to the third-best algorithm, and so on... the algorithm with the worst final average fitness value gets a score equal to 1;
- for each algorithm in E the scores assigned over the NP problems are collected and averaged:
  - the average score of the reference algorithm is referred to as rank R_0;
  - the remaining $NA - 1$ average scores are used to sort the corresponding algorithms in descending order and constitute their ranks, which are indicated with R_i ($i = 1, 2, 3, \ldots, NA - 1$).

Ranks give us a first indication of the global performance of the algorithms on E, and their order will be automatically displayed in the form of a "league" table.

4.2.3. Zed Statistic and P-values

This test requires the use of the so-called "zed value" statistic[12], calculated with the formula

[7] To avoid further delays, SOS process this information unless previously requested for another test or graphical procedure (in which case the values are retrieved and immediately returned). The platform is optimised minimise overheads and guarantee a prompt response.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

$$z_i = \frac{R_i - R_0}{\sqrt{\mathrm{NA}\frac{\mathrm{NA}+1}{6\mathrm{NP}}}}$$

for each i^{th} comparison algorithm ($i = 1, 2, 3, \ldots, \mathrm{NA} - 1$).

This allows for the determination of $\mathrm{NA} - 1$ probability values p_i-value ($i = 1, 2, 3, \ldots, \mathrm{NA} - 1$) through the *normalised cumulative normal* distributions z_i for each comparison algorithm.

4.2.4. Sequential decision-making

When all the previous steps are completed, SOS can proceeds with a sequential decision-making process in which A is tested against each comparison algorithm, following the order obtained in section 4.2.2, individually. A similar procedure to that one presented in section 4.1.5 for the Wilcoxon Rank-Sum test is therefore iterated $\mathrm{NA} - 1$ times in this step of the Hom-Bonferroni test. However, the rejection threshold requires some adjustments. In details, For each i^{th} comparison algorithm ($i = 1, 2, 3, \ldots, \mathrm{NA} - 1$), taken in the order established in section 4.2.2, the corresponding p_i-value and threshold α / i are compared:

- if p_i-value $\geq \frac{\alpha}{i}$ the test fails at rejecting the null-hypothesis $\mathrm{H}_0 : \mathrm{A} = \mathrm{B}$;
- if p_i-value $< \frac{\alpha}{i}$ the null-hypothesis is rejected and the rank can be used as an indicator to establish which algorithm displayed a better performence on E.

A table displaying the outcome of the test is automatically generated. Some examples are available in section 5. It is worth remarking that SOS employs a default $\alpha = 0.05$ also for this test. However, this can be changed how previously pointed out in section 4.1.

4.3. Advanced Statistical Analysis

The SOS platform provides a novel advanced statistical analysis procedure, referred to as ASA procedure, for comparing couples of algorithms on a single problem. This procedure makes use of several statistical tests and it is based on simple logic, displayed on figure 6, based on the fact that non-parametric tests are preferable if the assumption of normality cannot be made on the results distributions, whether parametric ones should be used if statistical evidence is found in support of such assumption. Therefore, SOS checks the results distributions of two selected algorithms, A and B, looking for such evidence and apply the most appropriate tests accordingly.

To launch this routine, the a `TableAvgStdStat` object must be initiated with the `UseAdvancedStastic` Boolean flag set equal to `true`, see figure 7. It must be remarked that if such flag is not activated, the Wilcoxon Rank-Sum Test is performed. It has been noted that the two tests differ when the number of runs is inferior to 100. In such a case, it is strongly recommended to use ASA for a more accurate caparison. Conversely, when a very high number of runs is available, ASA and Wilcoxon Rank-Sum are comparable. For this reason, the default number of runs performed by SOS is 100. However, this number of runs might be unpractical when facing time-consuming real-world optimisation problems. Hence, as explained in the previous sections, see figure 2, SOS provides a setter method for specifing the number of runs to be performed in a given experiment. The most common values used in the literature range from 30 to 60 runs.

When the ASA procedure is activated, SOS performs the following steps:

- the Shapiro-Wilk test [85] is run on the reference algorithm A and subsequently on the comparison algorithm B with null-hypothesis H_0: “results are normally distributed”;
- if both A and B are normally distributed (i.e. the Shapiro-Wilk test fails at rejecting H_0 both the cases) the homoscedasticity of the two distributions (i.e. homogeneity of variances) is tested employing the F-test [86] to check whether the normal distributions have identical variances and

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

- if the variances are equivalent it is concluded that both A and B have normal distributions, which suggests the use of a two-sided T-test [87,88] with null-hypothesis $H_0 : A \neq B$ to make a decision according to the following logic
  - if H_0 is rejected, A and B are two equivalent stochastic processes. The test terminates here.
  - if the test fails at rejecting if H_0 a one-sided T-test [87,88] is run to understand if A outperforms B, i.e. $H_0 : A < B$ is rejected, or B outperform A, i.e. $H_0 : A > B$ is rejected. The test terminates here.
- if variances are not equivalent, the Welch T-test [89,90] has to be used to with null-hypothesis $H_0 : A \neq B$;
  - if H_0 is rejected A and B are two equivalent stochastic processes. The test terminates here.
  - if the test fails at rejecting if H_0 a one-sided Welch T-test [89,90] is run to understand if A outperforms B, i.e. $H_0 : A < B$ is rejected, or B outperform A, i.e. $H_0 : A > B$ is rejected. The test terminates here.

- if at least one between A and B is not normally distributed (i.e. the Shapiro-Wilk test rejects H_0 in at least one case) a non-parametric test is necessary and a two-sided Wilcoxon Rank-Sum test is performed to test $H_0 : A \neq B$[8] and
  - if the null-hypothesis is rejected A and B are two equivalent stochastic processes. The test terminates here.
  - if the test fails at rejecting H_0, a one-sided Wilcoxon Rank-Sum test is run to understand if A outperforms B, i.e. $H_0 : A < B$ is rejected, or B outperform A, i.e. $H_0 : A > B$ is rejected. The test terminates here.

A graphical scheme is in figure 6.

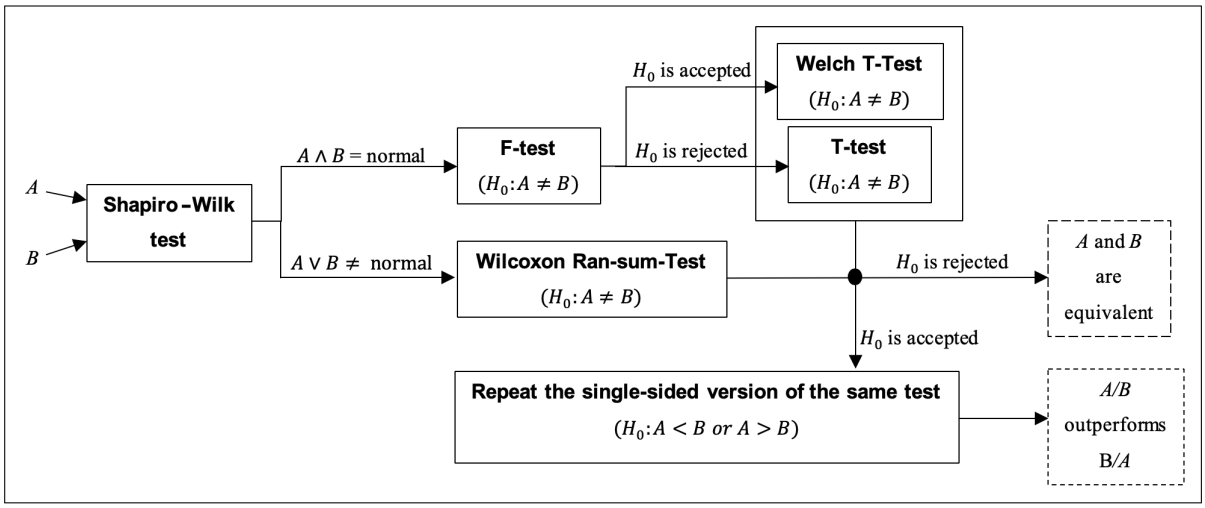


Figure 6. ASA flow diagram.

5. Visual representation of results

With SOS, results can be displayed in several formats. After performing an experiment, raw data are located in the SOS `results` folder, unless differently indicated, and can be processed straight away

[8] For logistic reasons, it is more convenient to formulate the null-hypothesis as $H_0 : A \neq B$ in the tests forming the ASA procedure, even though the SOS stand-alone version of the Wilcoxon Rank-Sum test of section 4.1 is implemented by considering $H_0 : A = B$. It must be remarked that the null-hypothesis can be arbitrarily chosen and should be formulated to facilitate the implementation of the test.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

or merged with those from other experiments before being processed. In the `results` folder, data are arranged in sub-folders with self-explanatory names to be easily individuated. The main folder comes with the name of the experiment and contains a text file describing it, as explained in section 2.1, as well as sub-folders whose names refer to the benchmark suite used, the specific function identifier and the dimensionality of the problem. Inside each folder, a fitness trend file is stored for each performed run. On top of the fitness values trend, these files also report full coordinates of the found near-optimal solution.

From the raw data, SOS can extrapolate information and create graphs and tables thanks to several auxiliary methods available in the platform. The class `Experiments` provides some of these routines for collecting data and transform them into more useful formats. Some of these can be seen in figure 7, which depicts the main method of the `TablesGenerator` class located in the default package.

```
public class TablesGenerator
{
    public static void main (String args[]) throws Exception
    {
        String workingDir = "";
        if (args.length < 1)
            workingDir ="/home/workingDirectory";
        else
            workingDir = args[0];

        Experiment experiment = new Experiment();
        experiment.setDirectory(workingDir);
        experiment.setTrendsFlag(true, true);

        experiment.importData();
        experiment.describeExperiment();


        experiment.computeAVG();
        experiment.computeSTD();
        experiment.computeMedian();
        experiment.deleteFinalValues();


        TableCEC2013Competition T1 = new TableCEC2013Competition(experiment);
        T1.setErrorFlag(true);
        T1.execute();

        TableBestWorstMedAvgStd T2 = new TableBestWorstMedAvgStd(experiment);
        T2.setErrorFlag(true);
        T2.execute();


        TableHolmBonferroni T3 = new TableHolmBonferroni(experiment);
        T3.setReferenceAlgorithm();
        T3.execute();

        TableStatistics T4 = new TableAvgStdStat(experiment, true, true);
        T4.setErrorFlag(true);
        T4.setReferenceAlgorithm();
        T4.execute();
    }
}
```

Figure 7. An example of the `TablesGenerator` class containing methods for visually display results in several different formats.

For starter, let us focus on the `experiment.setTrendsFlag(true,true);` method. Since the first Boolean value is `true`, while scanning the raw data for producing tables, SOS will simultaneously save a further text file with the vectors to plot for generating an average fitness trend graph. Since the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

second Boolean value is also `true`, the graph will show the fitness trend in terms of average error[9], as shown in figure 8, rather than average fitness. Outputs will look like the examples reported in figure 8. More details regarding the preparation of fitness trends diagrams are given in appendix A. The following `experiment.importData();` and `experiment.describeExperiment();` methods will start this process and describe it (in the generated log file) repetitively. If the fitness trends flag is not active, the `importData()` method will collect data from the corresponding folders, process them, but will store in memory only the information required for the generation of tables without saving the average trends information into the plottable files. Once raw data are loaded in memory, SOS can be asked to extrapolate information to be displayed in tables. For example, the commands `experiment.computeAVG();` and `experiment.computeSTD();` will lead to a table where the average fitness value (or average fitness error value if the error flag is active) $\pm$ the corresponding standard deviation are shown. The command `experiment.computeSTD();` will instead return the median fitness value (or median fitness error depending on the error flag) amongst the available runs. In the example of figure 7 the three methods are used but this is not compulsory. It is indeed possible to call only some of or none of them. Moreover, other methods not included in this example are also available, e.g. those for displaying the best of the wort run. If the methods are not called at all, when running a specific test, as shown in the reminder of figure 7, the required one will be automatically called. Conversely, if they are called, it is suggested to use `experiment.deleteFinalValues();` to free some memory by discarding the loaded final results value and keep only their average (plus standard deviation) and mean value. Before freeing the memory one may want to save the final values to plot histograms and distributions. Even if this is not common in the literature, it might be useful to do this in some cases and can be done in SOS by simply activating flags. For more technical details on SOS full capabilities, one may want to inspect the online software documentation.

At this point, the statistical tests described in section 4 can be launched to produce tables in PDF and LaTeXsource code formats. To structure compact but highly informative tables, SOS provides specific classes. Let us keep following the example in figure 7. One can see that four classes are used to produce tables for the `experiment` object passed as an argument to the constructor. These classes are:

- `TableCEC2013Competition`, whose objects can produce tables displaying results according to guidelines of IEEE CEC competitions (i.e. the indications for CEC 2013 [72] competition are considered);
- `TableBestWorstMedAvgStd`, whose objects can produce tables displaying the worst, the best, the median and the average fitness value over the performed runs;
- `TableHolmBonferroni`, whose objects can produce tables displaying the "experiment's league table", as shown in the examples of figure 9, obtained with the Hold-Bonferroni test as explained in section 4.2;
- TableAvgStdStat, whose objects can produce tables displaying fitness average value $\pm$ standard deviation and statistical analysis in terms of Wilcoxon Rank-Sum test, as described in section 4.1, or ASA test, as described in section 4.3. Examples are given in figure 10.

This four classes are all extensions of the abstract class `TableStatistics` from which they inherit several miscellaneous methods, as e.g. the `setErrorFlag()` method (used to indicate the reference algorithm) if tables should display average fitness or errors values), the `setRefenceAlgorithm()` method (used to indicate the reference algorithm) and the `execute()` method (which implements the specific statistical test), as well as attributes, as e.g. variables to store the significance levels α, confidence levels $\delta = 1 - \alpha$, error and other flags, etc.

[9] Unless the minimum fitness value is unknown, as it happens in most real-world applications. In this light, the error flag is mainly activated when benchmark functions are used.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

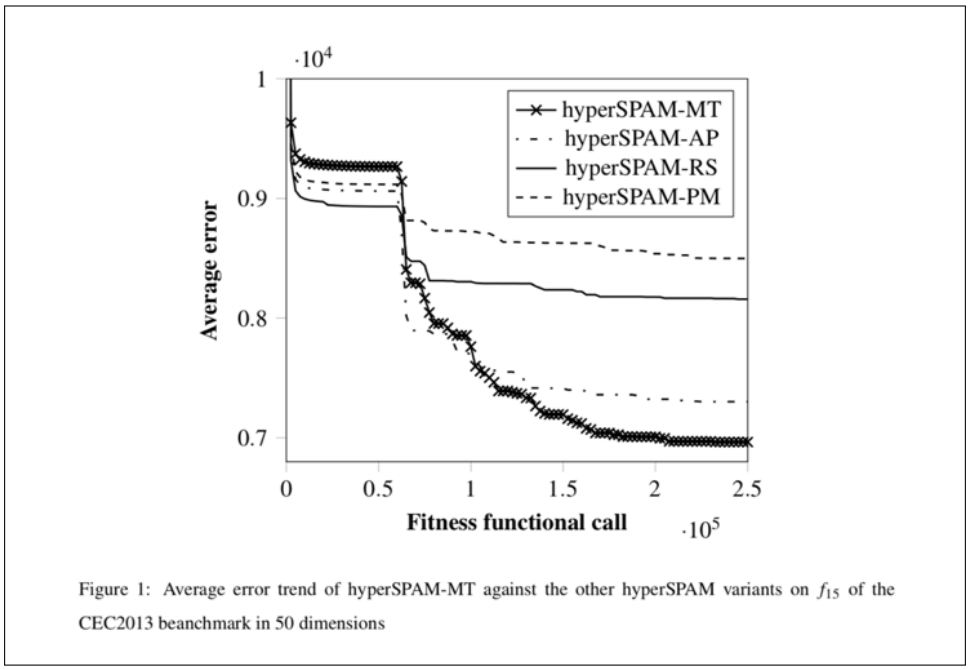
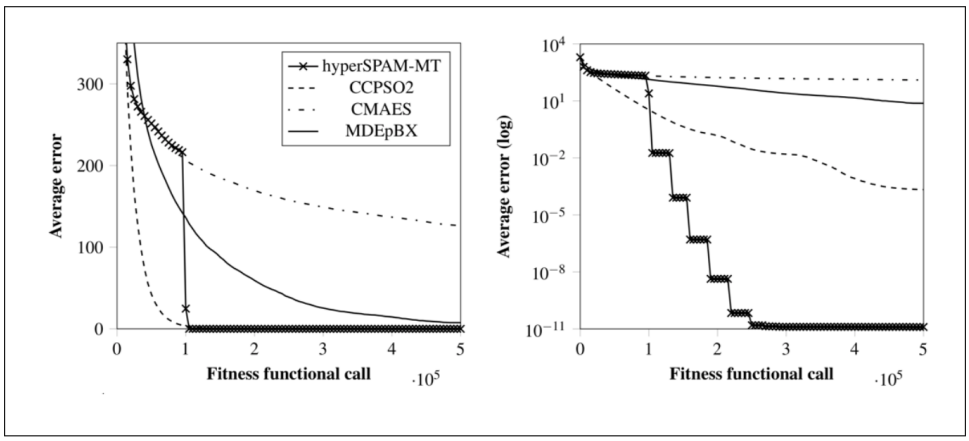


Figure 1: Average error trend of hyperSPAM-MT against the other hyperSPAM variants on f_{15} of the CEC2013 beanchmark in 50 dimensions

(a)



(b)

Figure 8. Example of average fitness (error) trends produced with SOS for the study in [23]. In (a), the full image with caption. In (b), a further example of an average error trend from the same study plotted in both linear and logarithmic scale. More examples from several other studies are gathered in [91].

Hence, if a new customised table must be added, this can be simply done by expending the `TableStatistics` superclass and make use of the auxiliary methods provided in such class to implement the `execute()` where new tests can be used and new layouts for the tables can be proposed.

However, the tables produced with the `TableHolmBonferroni` and `TableAvgStdStat` classes are the most requested for publications within the metaheuristic optimisation field when algorithms have to be compared in terms of the quality of the returned fitness values. Hence, in the vast majority of cases, these are the only two necessary kinds of tables to insert in a publication providing a comparative analysis. To further describe the informative content of these tables, let us refer to the examples reported in figures 9 and 10.

With reference to figure 9, which depicts the outcome of the Holm-Bonferssoni test, it can be noted that SOS reports the rank of the reference algorithm in the caption (which is automatically generated). Regardless of the number testbed or real-world problems added in the experiment file, SOS performs the test as described in section 4.2 and displays the comparison algorithms in descending

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

Table 25: Holm-Bonferroni procedure on non-rotated CEC2014 at 10, 50 and 100 dimension values (reference: DE/rand/1/exp, Rank = $8.35e+00$)

j	Optimizer	Rank	z_j	p_j	δ/j	Hypothesis
1	DE/rand/1/bin	7.68e+00	-1.76e+00	3.89e-02	5.00e-02	Rejected
2	eigen-DE/rand/1/exp	6.08e+00	-5.98e+00	1.09e-09	2.50e-02	Rejected
3	RIDE/rand/1/exp	5.11e+00	-8.57e+00	5.30e-18	1.67e-02	Rejected
4	MMCDE	4.76e+00	-9.48e+00	1.26e-21	1.25e-02	Rejected
5	RIDE/rand/1/bin	3.86e+00	-1.19e+01	8.04e-33	1.00e-02	Rejected
6	DE/rand/1/no-xo	3.79e+00	-1.21e+01	8.25e-34	8.33e-03	Rejected
7	eigen-DE/rand/1/bin	3.08e+00	-1.39e+01	2.34e-44	7.14e-03	Rejected
8	DE/current-to-rand/1	2.14e+00	-1.64e+01	8.12e-61	6.25e-03	Rejected

(a)

TABLE 10. Holm-Bonferroni procedure (reference: RI-(1 + 1)-CMA-ES, Rank = 7.16e+00)

j	Optimizer	Rank	z_j	p_j	δ/j	Hypothesis
1	μDEA	7.00e+00	-3.81e-01	3.52e-01	5.00e-02	Accepted
2	MDE-pBX	6.83e+00	-7.89e-01	2.15e-01	2.50e-02	Accepted
3	PMS	6.68e+00	-1.17e+00	1.21e-01	1.67e-02	Accepted
4	cDE-light	6.63e+00	-1.28e+00	1.00e-01	1.25e-02	Accepted
5	(1 + 1)-CMA-ES	6.12e+00	-2.53e+00	5.68e-03	1.00e-02	Rejected
6	Rosenbrock	5.19e+00	-4.82e+00	7.27e-07	8.33e-03	Rejected
7	ISPO	4.17e+00	-7.32e+00	1.23e-13	7.14e-03	Rejected
8	JADE	3.93e+00	-7.89e+00	1.48e-15	6.25e-03	Rejected
9	SPSA	1.27e+00	-1.44e+01	1.81e-47	5.56e-03	Rejected

(b)

Figure 9. Two examples of `TableHolmBonferroni` tables. In (a), some results from the study in [4] obtained with the non-rotated functions of the R-CEC14 benchmark suite presented in section 3.1. More examples are available in the extended results files stored in the repository [92]. In (b), some results from the study in [66] obtained over the functions of the SOS implementation of the CEC 2014 [2] benchmark suite. Extended results tables for this study are available in the repository [91].

order according to their rank. As can be seen by comparing the Rank and p_j columns, this corresponds to a descending order also for the p-values. This way, algorithms in occupying top positions are quite likely to behave similarly to the reference algorithm (i.e. the null-hypothesis is accepted) while those occupying lower positions behave worse than the reference algorithm (i.e. the null-hypothesis is rejected). Relevant information as the z-statistic values and normalised significance/confidence levels can be also added.

Conversely, figure 10 shows two examples of classic comparative tables obtained with the class TableAvgStdStat.At first glance, it can be noticed that the best performance (in terms of average fitness error) is highlighted in boldface. This is obtained by setting the `useBold` flag equal to `true`, as in the example of figure 7. Generally, the boldface option is useful as it allows for spotting the "winner" algorithm in a facilitated way. However, it can be turned off by simply setting `useBold = false`.

To strengthen the validity of the displayed results, the outcome of a statistic test is also added to the table. With reference to figure 7, if the `useAdvancedStatistic` flag is activated (i.e. it is equal to `true`) the ASA test presented in section 4.3 is performed. The Wilcoxon Rank-Sum test, described in section 4.1, is run otherwise.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

Regardless of the emplyed statistical test, the same compact notation is adopted to report its outcome in the table:

- if the reference algorithm A (i.e. the first one in the table) is statistically equivalent to a comparison algorithm B (i.e. $H_0 : A = B$ cannot be rejected), the symbol $=$ is placed next to B;
- if the reference algorithm A statistically outperforms a comparison algorithm B, the symbol $+$ is placed next to B;
- if the reference algorithm A is statistically outperformed by a comparison algorithm B, the symbol $-$ is placed next to B.

TABLE 1. Average error ± standard deviation and Wilcoxon Rank-Sum Test (reference: RI-(1 + 1)-CMA-ES) for RI-(1 + 1)-CMA-ES against (1 + 1)-CMA-ES, SPSA and Rosenbrock on CEC2014[4] in 10 dimensions.

	RI-(1 + 1)-CMA-ES	(1 + 1)-CMA-ES		SPSA		Rosenbrock	
f_1	1.80e + 00 ± 5.70e + 00	**0.00e + 00 ± 0.00e + 00**	-	5.40e + 07 ± 8.85e + 07	+	2.57e + 05 ± 6.33e + 05	+
f_2	0.00e + 00 ± 1.80e − 14	**0.00e + 00 ± 0.00e + 00**	=	8.02e + 06 ± 9.32e + 06	+	5.03e + 03 ± 3.80e + 03	+
f_3	**0.00e + 00 ± 0.00e + 00**	0.00e + 00 ± 0.00e + 00	=	6.33e + 04 ± 2.54e + 04	+	5.73e + 03 ± 6.03e + 03	+
f_4	**9.72e + 00 ± 1.52e + 01**	2.49e + 01 ± 1.51e + 01	+	2.87e + 02 ± 2.37e + 02	+	1.66e + 01 ± 1.92e + 01	+
f_5	2.00e + 01 ± 3.72e − 04	**2.00e + 01 ± 2.87e − 03**	-	2.06e + 01 ± 2.48e − 01	+	2.00e + 01 ± 5.72e − 03	+
f_6	**1.26e + 01 ± 2.26e + 00**	1.55e + 01 ± 2.29e + 00	+	1.89e + 01 ± 1.69e + 00	+	1.84e + 01 ± 2.65e + 00	+
f_7	**6.03e − 02 ± 3.45e − 02**	1.52e − 01 ± 1.29e − 01	+	6.21e + 02 ± 2.28e + 02	+	9.70e − 01 ± 3.09e + 00	+
f_8	**7.99e + 01 ± 2.67e + 01**	1.36e + 02 ± 4.46e + 01	+	1.47e + 02 ± 5.35e + 01	+	1.43e + 02 ± 3.68e + 01	+
f_9	**9.28e + 01 ± 3.43e + 01**	1.54e + 02 ± 5.14e + 01	+	1.55e + 02 ± 5.06e + 01	+	1.63e + 02 ± 6.68e + 01	+
f_{10}	**1.09e + 03 ± 3.13e + 02**	1.66e + 03 ± 3.20e + 02	+	1.56e + 03 ± 6.75e + 02	+	1.66e + 03 ± 4.85e + 02	+
f_{11}	**1.18e + 03 ± 2.08e + 02**	1.77e + 03 ± 3.88e + 02	+	1.62e + 03 ± 6.24e + 02	+	1.82e + 03 ± 4.56e + 02	+
f_{12}	**4.34e − 01 ± 2.70e − 01**	1.19e + 00 ± 1.19e + 00	+	3.95e + 00 ± 2.14e + 00	+	3.58e + 00 ± 2.56e + 00	+
f_{13}	**2.57e − 01 ± 8.39e − 02**	5.34e − 01 ± 1.85e − 01	+	1.34e + 01 ± 3.58e + 00	+	3.36e − 01 ± 1.22e − 01	+
f_{14}	**2.27e − 01 ± 6.07e − 02**	4.66e − 01 ± 3.08e − 01	+	1.80e + 02 ± 6.73e + 01	+	4.01e − 01 ± 2.27e − 01	+
f_{15}	**1.76e + 00 ± 7.83e − 01**	3.53e + 00 ± 2.31e + 00	+	2.87e + 02 ± 8.62e + 02	+	6.52e + 00 ± 6.38e + 00	+
f_{16}	**4.35e + 00 ± 3.27e − 01**	4.63e + 00 ± 3.04e − 01	+	4.85e + 00 ± 2.05e − 01	+	4.65e + 00 ± 2.78e − 01	+
f_{17}	**2.71e + 02 ± 1.28e + 02**	5.43e + 02 ± 2.98e + 02	+	2.08e + 06 ± 1.75e + 06	+	3.37e + 04 ± 8.08e + 04	+
f_{18}	**3.28e + 01 ± 1.60e + 01**	3.88e + 01 ± 2.47e + 01	=	2.34e + 05 ± 1.05e + 06	+	1.66e + 04 ± 1.15e + 04	+
f_{19}	**4.31e + 00 ± 9.56e − 01**	7.40e + 00 ± 2.60e + 00	+	1.27e + 02 ± 1.13e + 02	+	5.00e + 01 ± 4.24e + 01	+
f_{20}	**5.94e + 01 ± 2.61e + 01**	1.16e + 02 ± 7.45e + 01	+	1.64e + 06 ± 2.84e + 06	+	8.30e + 03 ± 8.82e + 03	+
f_{21}	**1.70e + 02 ± 1.35e + 02**	3.59e + 02 ± 2.10e + 02	+	1.48e + 06 ± 1.44e + 06	+	1.19e + 04 ± 1.13e + 04	+
f_{22}	**1.06e + 02 ± 8.99e + 01**	2.65e + 02 ± 1.48e + 02	+	4.92e + 02 ± 1.97e + 02	+	4.99e + 02 ± 1.88e + 02	+
f_{23}	3.22e + 02 ± 4.11e + 01	3.18e + 02 ± 5.91e + 01	=	5.67e + 02 ± 1.39e + 02	+	**3.07e + 02 ± 8.21e + 01**	-
f_{24}	**1.92e + 02 ± 2.11e + 01**	3.26e + 02 ± 1.78e + 02	+	3.38e + 02 ± 1.52e + 02	+	2.94e + 02 ± 1.55e + 02	+
f_{25}	**1.91e + 02 ± 1.39e + 01**	1.99e + 02 ± 1.14e + 01	+	3.32e + 02 ± 9.62e + 01	+	2.00e + 02 ± 1.09e + 01	+
f_{26}	**1.04e + 02 ± 1.79e + 01**	1.56e + 02 ± 9.08e + 01	+	3.00e + 02 ± 1.08e + 02	+	1.76e + 02 ± 1.05e + 02	+
f_{27}	**3.62e + 02 ± 1.43e + 02**	5.01e + 02 ± 1.43e + 02	+	7.17e + 02 ± 1.59e + 02	+	5.38e + 02 ± 1.57e + 02	+
f_{28}	**1.09e + 03 ± 3.69e + 02**	2.83e + 03 ± 1.65e + 03	+	1.78e + 03 ± 6.40e + 02	+	2.78e + 03 ± 1.77e + 03	+
f_{29}	**2.69e + 02 ± 2.84e + 01**	1.19e + 05 ± 4.43e + 05	+	3.48e + 06 ± 7.26e + 06	+	4.75e + 05 ± 7.87e + 05	+
f_{30}	**1.10e + 03 ± 2.31e + 02**	1.38e + 03 ± 4.85e + 02	+	9.40e + 04 ± 1.37e + 05	+	1.68e + 03 ± 4.82e + 02	+

(a) Wilcoxon Rank-Sum test

Table 21: Average fitness ± standard deviation and statistic comparison (reference: *hyperSPAM-MT*) for *hyperSPAM-MT* against *CMAES MDE-pBX*, and *CCPSO2* on CEC2011[1] in 6, 30, and 20 dimensions.

	hyperSPAM-MT	*CMAES*		*MDE-pBX*		*CCPSO2*	
$Prob_1(6D)$	1.87e+01 ± 1.21e+01	3.37e+01±1.28e+01	+	8.16e+00±6.77e+00	-	**6.77e+00±3.79e+00**	-
$Prob_2(30D)$	−1.86e+01 ± 4.68e+00	−2.53e+01±2.74e+00	-	−2.20e+01±4.32e+00	-	**−2.67e+01±1.74e+00**	-
$Prob_7(20D)$	7.29e−01 ± 1.47e−01	**5.82e−01±8.31e−02**	-	1.08e+00±1.77e−01	+	1.16e+00±1.25e−01	+

(b) ASA test

Figure 10. Two examples of `TableAvgStdStat` tables with Wilcoxon Rank-Sum test (a) and ASA test (b) respectively. In (a), some results from the study in [66] obtained over the functions of the SOS implementation of the CEC 2014 [2] benchmark suite. In (b), some from the study in [23] obtained over three real-world problems of the CEC 2011 [77] bechmakr suite implemented in SOS. Extended results tables for the studies in (a) and (b) are available in the repository [91].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 March 2020 doi:10.20944/preprints202003.0381.v1

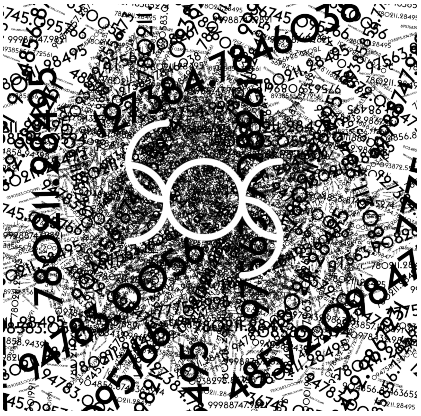
It is important to highlight the importance of having statistical evidence next to the average fitness error value. For example, with reference to figure 10-(a), it can be noticed that despite (1+1)-CMA-ES displays the best average error value for f_2, this is quite likely to be an isolated event as the $=$ symbol next to it suggests that its general behaviour is actually equivalent to the one of the reference algorithm, i.e. RI-(1+1)-CMA-ES, from the statical point of view. Indeed, there is a very small difference between the two average error values, i.e. of about 10^{-14}, which is mathematically in favour of the comparison algorithm but not practically insufficient to be able to state that the comparison algorithm outperforms the reference algorithm.

Supplementary Materials: Further material can be obtained from the following open access repositories

- Raw data and extended galleries of tables and fitness trend graphs generated with SOS for some of the studies mentioned in this article are available in [91,92];
- extensive galleries of images obtained with the joint use of SOS, for generating data, and the IOHprofiler in [7], for plotting them, are available in [41].

Funding: This research received no external funding.

Acknowledgments: I would like to thank Dr Olga Olga Zlydareva for making the SOS logo.



Terminology

The following terminology is used in this manuscript:

Computational budget	Maximum allowed number of fitness evaluations for an Optimisation process
Fitness function	The objective function (from the EAs jargon, usually refers to a scalar function)
Fitness	The value returned by the fitness function
Fitness landscape	Refers to the topology of the fitness function co-domain
Basin of attraction	A set of points from which a dynamical system moves to a particular attractor
Run	An optimisation process during which one algorithm optimises one problem
Initial guess	Initial (random/passed) solution of an algorithm
Population-based	A metaheuristic algorithm requiring a set of candidate solutions to function
Individuals	In the EA jargon this is equivalent to say candidate solution
Population	The set containing the individuals
Population size	Number of solutions processed by a population-based algorithm
Variation operator	Perturbation orator typical of EAs, such as e.g. crossover and mutation
Recombination op.	Operator (from EAs) producing a new solution from 1 or more available individuals
Mutation op.	Operator (from EAs) perturbing a single solution
parent selection	Methods for selecting candidate solutions in EAs to perform Recombination
Survivor selection	Method to form a new population from available individuals
Local searcher	Operator suitable for refining a solution rather than exploring the search space
Non-parametric test	A statistical test that does not require any assumption on how data are distributed
Null-hypothesis	In statistics it is the hypothesis of lack of significant difference between 2 distributions

 doi:10.20944/preprints202003.0381.v1

Abbreviations

The following abbreviations are used in this manuscript:

CEC	Congress on evolutionary computation
CMA	Covariance matrix adaptation
DE	Differential evolution
EA	Evolutionary algorithm
EC	Evolutionary computation
ES	Evolution strategy
FWER	family-wise error rate
GA	Genetic algorithm
IEEE	Institute of electrical and electronics engineers
NFLT	No free lunch theorem
PSO	Particle swarm optimisation
SI	Swarm Intelligence
SOS	Stochastic optimisation software

Appendix A. Producing the fitness trend

To save a plottable file containing the average fitness trend of an algorithm over a specific problem, SOS first fills an array of size equal to the computational budget with the values from the `FTrend` object returned by the algorithm. This is done for each performed run. The initial and final fitness values are always available. However, if some intermediary fitness values are not available from the returned `FTrend` object, SOS fills these gaps with the previous fitness value. This way, a monotone and complete sequence of fitness values, reproducing the optimisation trend of a single run, is available and ready to averaged with those obtained from the other runs. The resulting average fitness trend is stored in another array of equal dimension. Subsequently, the content of these arrays is downsampled to have 500 equispaced (in terms of fitness evaluation counter) elements. This results in better quality graphs as those in figure 10 – sometimes, having too many values does not lead to easier-to-interpret plots and it is always better to have equispaced points to avoid dense clusters followed by empty spaces in the fitness trend. However, it must be pointed out that the number of points plotted in the fitness trend can be changed, thus allowing for a higher or lower number of fitness values to be included in the graphs to adapt to specific needs.

References

1. Caraffini, F.; Kononova, A.V.; Corne, D. Infeasibility and structural bias in differential evolution. *Information Sciences* **2019**, *496*, 161–179. doi:10.1016/j.ins.2019.05.019.
2. Liang, J.J.; Qu, B.Y.; Suganthan, P.N. Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization. Technical report, Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore, 2013.
3. Caraffini, F.; Neri, F.; Picinali, L. An analysis on separability for Memetic Computing automatic design. *Information Sciences* **2014**, *265*, 1–22. doi:10.1016/j.ins.2013.12.044.
4. Caraffini, F.; Neri, F. A study on rotation invariance in differential evolution. *Swarm and Evolutionary Computation* **2019**, *50*, 100436. doi:10.1016/j.swevo.2018.08.013.
5. Mittelmann, H.D.; Spellucci, P. Decision tree for optimization software. http://plato.asu.edu/guide.html, 2005.
6. Hansen, N.; Auger, A.; Mersmann, O.; Tusar, T.; Brockhoff, D. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. *ArXiv* **2016**, [1603.08785].
7. Doerr, C.; Ye, F.; Horesh, N.; Wang, H.; Shir, O.M.; Bäck, T. Benchmarking discrete optimization heuristics with IOHprofiler. *Applied Soft Computing* **2020**, *88*, 106027. doi:10.1016/j.asoc.2019.106027.
8. Durillo, J.J.; Nebro, A.J. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software* **2011**, *42*, 760 – 771. doi:10.1016/j.advengsoft.2011.05.014.

9. Reed, D.H.; Frankham, R. Correlation between fitness and genetic diversity. *Conservation biology* **2003**, *17*, 230–237.
10. Yaman, A.; Iacca, G.; Caraffini, F. A comparison of three differential evolution strategies in terms of early convergence with different population sizes. AIP Conference Proceedings, 2019, Vol. 2070, p. 020002. doi:10.1063/1.5089969.
11. Kononova, A.; Corne, D.; De Wilde, P.; Shneer, V.; Caraffini, F. Structural bias in population-based algorithms. *Information Sciences* **2015**, *298*. doi:10.1016/j.ins.2014.11.035.
12. García, S.; Fernández, A.; Luengo, J.; Herrera, F. A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability. *Soft Computing* **2009**, *13*, 959–977. doi:10.1007/s00500-008-0392-y.
13. Del Ser, J.; Osaba, E.; Molina, D.; Yang, X.S.; Salcedo-Sanz, S.; Camacho, D.; Das, S.; Suganthan, P.N.; Coello Coello, C.A.; Herrera, F. Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation* **2019**, *48*, 220–250. doi:10.1016/j.swevo.2019.04.008.
14. Coello, C.A.C. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering* **2002**, *191*, 1245 – 1287. doi:10.1016/S0045-7825(01)00323-1.
15. Kononova, A.V.; Caraffini, F.; Wang, H.; Bäck, T. Can Single Solution Optimisation Methods Be Structurally Biased? *Preprints* **2020**. doi:10.20944/preprints202002.0277.v1.
16. Caraffini, F. The Stochastic Optimisation Software (SOS) platform. doi:10.5281/ZENODO.3237023.
17. Caraffini, F. *Algorithmic issues in computational intelligence optimization: from design to implementation, from implementation to design*; Number 243 in Jyväskylä studies in computing, University of Jyväskylä, 2016.
18. Eiben, A.; Smith, J. *Introduction to Evolutionary Computing*; Vol. 53, *Natural Computing Series*, Springer Berlin Heidelberg: Berlin, Heidelberg, 2015. doi:10.1007/978-3-662-44874-8.
19. Kennedy, J.; Shi, Y.; Eberhart, R.C. *Swarm Intelligence*; Elsevier, 2001; p. 512.
20. Burke, E.K.; Hyde, M.; Kendall, G.; Ochoa, G.; Özcan, E.; Woodward, J.R., A Classification of Hyper-heuristic Approaches. In *Handbook of Metaheuristics*; Springer US: Boston, MA, 2010; pp. 449–468. doi:10.1007/978-1-4419-1665-5_15.
21. Burke, E.K.; Gendreau, M.; Hyde, M.; Kendall, G.; Ochoa, G.; Özcan, E.; Qu, R. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* **2013**, *64*, 1695–1724. doi:10.1057/jors.2013.71.
22. Caraffini, F.; Neri, F.; Iacca, G.; Mol, A. Parallel memetic structures. *Information Sciences* **2013**, *227*, 60–82. doi:10.1016/j.ins.2012.11.017.
23. Caraffini, F.; Neri, F.; Epitropakis, M. HyperSPAM: A study on hyper-heuristic coordination strategies in the continuous domain. *Information Sciences* **2019**, *477*, 186–202. doi:10.1016/j.ins.2018.10.033.
24. V. Price, Kenneth Storn, Rainer M. Lampinen, J.A. *Differential Evolution*; Natural Computing Series, Springer-Verlag: Berlin/Heidelberg, 2005. doi:10.1007/3-540-31306-0.
25. Xinchao, Z. Simulated annealing algorithm with adaptive neighborhood. *Applied Soft Compututing* **2011**, *11*, 1827–1836.
26. Kennedy, J.; Eberhart, R. Particle swarm optimization. Proceedings of ICNN'95 - International Conference on Neural Networks, 1995, Vol. 4, pp. 1942–1948 vol.4. doi:10.1109/ICNN.1995.488968.
27. Auger, A. Benchmarking the (1+1) evolution strategy with one-fifth success rule on the BBOB-2009 function testbed. Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference - GECCO '09; ACM Press: New York, New York, USA, 2009; p. 2447. doi:10.1145/1570256.1570342.
28. Hansen, N.; Ostermeier, A. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* **2001**, *9*, 159–195. doi:10.1162/106365601750190398.
29. Hansen, N., The CMA Evolution Strategy: A Comparing Review. In *Towards a New Evolutionary Computation: Advances in the Estimation of Distribution Algorithms*; Springer Berlin Heidelberg: Berlin, Heidelberg, 2006; pp. 75–102. doi:10.1007/3-540-32494-1_4.
30. Liang, J.J.; Qin, A.K.; Suganthan, P.N.; Baskar, S. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE Transactions on Evolutionary Computation* **2006**, *10*, 281–295. doi:10.1109/TEVC.2005.857610.
31. Li, X.; Yao, X. Cooperatively Coevolving Particle Swarms for Large Scale Optimization. *Evolutionary Computation, IEEE Transactions on* **2012**, *16*, 210–224. doi:10.1109/TEVC.2011.2112662.

32. Brest, J.; Greiner, S.; Boskovic, B.; Mernik, M.; Zumer, V. Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation* **2006**, *10*, 646–657. doi:10.1109/TEVC.2006.872133.
33. Brest, J.; Maucec, M.S. Self-adaptive differential evolution algorithm using population size reduction and three strategies. *Soft Comput.* **2011**, *15*, 2157–2174. doi:10.1007/s00500-010-0644-5.
34. Brest, J.; Maucec, M.S. Self-adaptive differential evolution algorithm using population size reduction and three strategies. *Soft Comput.* **2011**, *15*, 2157–2174. doi:10.1007/s00500-010-0644-5.
35. Alic, A.; Berkovic, K.; Boskovic, B.; Brest, J. Population Size in Differential Evolution. Swarm, Evolutionary, and Memetic Computing and Fuzzy and Neural Computing - 7th International Conference, SEMCCO 2019, and 5th International Conference, FANCCO 2019, Maribor, Slovenia, July 10-12, 2019, Revised Selected Papers; Zamuda, A.; Das, S.; Suganthan, P.N.; Panigrahi, B.K., Eds. Springer, 2019, Vol. 1092, *Communications in Computer and Information Science*, pp. 21–30. doi:10.1007/978-3-030-37838-7_3.
36. Zhang, J.; Sanderson, A. JADE: Adaptive Differential Evolution With Optional External Archive. *Evolutionary Computation, IEEE Transactions on* **2009**, *13*, 945–958. doi:10.1109/TEVC.2009.2014613.
37. Islam, S.M.; Das, S.; Ghosh, S.; Roy, S.; Suganthan, P.N. An Adaptive Differential Evolution Algorithm With Novel Mutation and Crossover Strategies for Global Numerical Optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **2012**, *42*, 482–500. doi:10.1109/TSMCB.2011.2167966.
38. Molina, D.; Lozano, M.; Herrera, F. MA-SW-Chains: Memetic algorithm based on local search chains for large scale continuous global optimization. IEEE Congress on Evolutionary Computation, 2010, pp. 1–8. doi:10.1109/CEC.2010.5586034.
39. Epitropakis, M.G.; Caraffini, F.; Neri, F.; Burke, E.K. A Separability Prototype for Automatic Memes with Adaptive Operator Selection. IEEE SSCI 2014 - 2014 IEEE Symposium Series on Computational Intelligence - FOCI 2014: 2014 IEEE Symposium on Foundations of Computational Intelligence, Proceedings, 2015, pp. 70–77. doi:10.1109/FOCI.2014.7007809.
40. Caraffini, F.; Kononova, A.V. Structural bias in differential evolution: A preliminary study. AIP Conference Proceedings, 2019, Vol. 2070, p. 020005. doi:10.1063/1.5089972.
41. Structural Bias in Optimisation Algorithms: Extended Results, 2020. Mendeley Data, doi:10.17632/zdh2phb3b4.2.
42. Wolpert, D.; Macready, W. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* **1997**, *1*, 67–82. doi:10.1109/4235.585893.
43. Ho, Y.C.; Pepyne, D.L. Simple Explanation of the No Free Lunch Theorem of Optimization. *Cybernetics and Systems Analysis* **2002**, *38*, 292–298. doi:10.1023/A:1021251113462.
44. Mason, K.; Duggan, J.; Howley, E. A meta optimisation analysis of particle swarm optimisation velocity update equations for watershed management learning. *Applied Soft Computing* **2018**, *62*, 148 – 161. doi:10.1016/j.asoc.2017.10.018.
45. Neumüller, C.; Wagner, S.; Kronberger, G.; Affenzeller, M. Parameter Meta-optimization of Metaheuristic Optimization Algorithms. Computer Aided Systems Theory – EUROCAST 2011; Moreno-Díaz, R.; Pichler, F.; Quesada-Arencibia, A., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2012; pp. 367–374. doi:10.1007/978-3-642-27549-4_47.
46. Andre, J.; Siarry, P.; Dognon, T. An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization. *Advances in engineering software* **2001**, *32*, 49–60. doi:10.1016/S0965-9978(00)00070-3.
47. Lampinen, J.; Zelinka, I.; others. On stagnation of the differential evolution algorithm. Proceedings of MENDEL, 2000, pp. 76–83.
48. Caraffini, F.; Neri, F. Rotation Invariance and Rotated Problems: An Experimental Study on Differential Evolution. Applications of Evolutionary Computation; Sim, K.; Kaufmann, P., Eds.; Springer International Publishing: Cham, 2018; pp. 597–614. doi:10.1007/978-3-319-77538-8_41.
49. Sang, H.Y.; Pan, Q.K.; Duan, P.y. Self-adaptive fruit fly optimizer for global optimization. *Natural Computing* **2019**, *18*, 785–813. doi:10.1007/s11047-016-9604-z.
50. Mirjalili, S.; Mirjalili, S.M.; Lewis, A. Grey Wolf Optimizer. *Advances in Engineering Software* **2014**, *69*, 46 – 61. doi:10.1016/j.advengsoft.2013.12.007.

51. Chu, S.C.; Tsai, P.w.; Pan, J.S. Cat Swarm Optimization. PRICAI 2006: Trends in Artificial Intelligence; Yang, Q.; Webb, G., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2006; pp. 854–858. doi:10.1007/978-3-540-36668-3_94.
52. Meng, X.B.; Gao, X.; Lu, L.; Liu, Y.; Zhang, H. A new bio-inspired optimisation algorithm: Bird Swarm Algorithm. *Journal of Experimental & Theoretical Artificial Intelligence* **2016**, *28*, 673–687. doi:10.1080/0952813X.2015.1042530.
53. Chu, S.C.; Tsai, P.w.; Pan, J.S. Cat Swarm Optimization. PRICAI 2006: Trends in Artificial Intelligence; Yang, Q.; Webb, G., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2006; pp. 854–858. doi:10.1007/978-3-540-36668-3_94.
54. Mirjalili, S. Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm. *Knowledge-Based Systems* **2015**, *89*, 228 – 249. doi:10.1016/j.knosys.2015.07.006.
55. Wang, Y.; Du, T. An Improved Squirrel Search Algorithm for Global Function Optimization. *Algorithms* **2019**, *12*, 80. doi:10.3390/a12040080.
56. Sulaiman, M.H.; Mustaffa, Z.; Saari, M.M.; Daniyal, H. Barnacles Mating Optimizer: A new bio-inspired algorithm for solving engineering optimization problems. *Engineering Applications of Artificial Intelligence* **2020**, *87*, 103330. doi:10.1016/j.engappai.2019.103330.
57. Geem, Z.W.; Kim, J.H.; Loganathan, G.V. A new heuristic optimization algorithm: harmony search. *simulation* **2001**, *76*, 60–68. doi:10.1177/003754970107600201.
58. Atashpaz-Gargari, E.; Lucas, C. Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition. 2007 IEEE Congress on Evolutionary Computation, 2007, pp. 4661–4667. doi:10.1109/CEC.2007.4425083.
59. Rao, R.; Savsani, V.; Vakharia, D. Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems. *Computer-Aided Design* **2011**, *43*, 303 – 315. doi:10.1016/j.cad.2010.12.015.
60. Bidar, M.; Kanan, H.R.; Mouhoub, M.; Sadaoui, S. Mushroom Reproduction Optimization (MRO): A Novel Nature-Inspired Evolutionary Algorithm. 2018 IEEE Congress on Evolutionary Computation (CEC), 2018, pp. 1–10. doi:10.1109/CEC.2018.8477837.
61. Hatamlou, A. Black hole: A new heuristic optimization approach for data clustering. *Information Sciences* **2013**, *222*, 175 – 184. Including Special Section on New Trends in Ambient Intelligence and Bio-inspired Systems, doi:10.1016/j.ins.2012.08.023.
62. Taradeh, M.; Mafarja, M.; Heidari, A.A.; Faris, H.; Aljarah, I.; Mirjalili, S.; Fujita, H. An evolutionary gravitational search-based feature selection. *Information Sciences* **2019**, *497*, 219 – 239. doi:10.1016/j.ins.2019.05.038.
63. Iacca, G.; Caraffini, F. Compact Optimization Algorithms with Re-Sampled Inheritance. Applications of Evolutionary Computation; Kaufmann, P.; Castillo, P.A., Eds.; Springer International Publishing: Cham, 2019; pp. 523–534. doi:10.1007/978-3-030-16692-2_35.
64. Takahama, T.; Sakai, S. Solving nonlinear optimization problems by Differential Evolution with a rotation-invariant crossover operation using Gram-Schmidt process. 2010 Second World Congress on Nature and Biologically Inspired Computing (NaBIC), 2010, pp. 526–533. doi:10.1109/NABIC.2010.5716327.
65. Guo, S.; Yang, C. Enhancing Differential Evolution Utilizing Eigenvector-Based Crossover Operator. *IEEE Transactions on Evolutionary Computation* **2015**, *19*, 31–49. doi:10.1109/TEVC.2013.2297160.
66. Caraffini, F.; Iacca, G.; Yaman, A. Improving (1+1) covariance matrix adaptation evolution strategy: A simple yet efficient approach. *AIP Conference Proceedings* **2019**, *2070*, 020004. doi:10.1063/1.5089971.
67. Igel, C.; Suttorp, T.; Hansen, N. A Computational Efficient Covariance Matrix Update and a (1+1)-CMA for Evolution Strategies. Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation; Association for Computing Machinery: New York, NY, USA, 2006; GECCO '06, p. 453–460. doi:10.1145/1143997.1144082.
68. Xin Yao.; Yong Liu.; Guangming Lin. Evolutionary programming made faster. *IEEE Transactions on Evolutionary Computation* **1999**, *3*, 82–102. doi:10.1109/4235.771163.
69. Suganthan, P.N.; Hansen, N.; Liang, J.J.; Deb, K.; Chen, Y.P.; Auger, A.; Tiwari, S. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Technical Report

201212, Zhengzhou University and Nanyang Technological University, Zhengzhou China and Singapore, 2005.

70. Tang, K.; Yao, X.; Suganthan, P.N.; MacNish, C.; Chen, Y.P.; Chen, C.M.; Yang, Z. Benchmark Functions for the CEC 2008 Special Session and Competition on Large Scale Global Optimization. Technical report, Nature Inspired Computation and Applications Laboratory, {USTC}, China, 2007.
71. Tang, K.; Li, X.; Suganthan, P.N.; Yang, Z.; Weise, T. Benchmark functions for the CEC 2010 special session and competition on large-scale global optimization. Technical Report 1, Technical report, Univ. of Science and Technology of China, 2010.
72. Liang, J.J.; Qu, B.Y.; Suganthan, P.N.; Hernández-Díaz, A.G. Problem Definitions and Evaluation Criteria for the CEC 2013 Special Session on Real-Parameter Optimization. Technical Report 201212, Zhengzhou University and Nanyang Technological University, Zhengzhou China and Singapore, 2013.
73. Li, X.; Tang, K.; Omidvar, M.N.; Yang, Z.; Qin, K. Benchmark Functions for the CEC'2013 Special Session and Competition on Large-Scale Global Optimization. Technical report, RMIT University, Melbourne, Australia; University of Science andTechnology of China, Hefei, Anhui, China; National University of Defense Technology, Changsha 410073, China, 2013.
74. Li, Xiaodong; Tang, K.Y.Z.M.D. Benchmark Functions for the CEC'2015 Special Session and Competition on Large Scale Global Optimization. Technical report, Technical report, Univ. of Science and Technology of China, 2015.
75. Herrera, F.; Lozano, M.; Molina, D. Test Suite for the Special Issue of Soft Computing on Scalability of Evolutionary Algorithms and other Metaheuristics for Large Scale Continuous Optimization Problems. Technical report, University of Granada, Spain, 2010.
76. Hansen, N.; Auger, A.; Finck, S.; Ros, R.; Hansen, N.; Auger, A.; Finck, S.; Optimiza, R.R.R.p.B.b. Real-Parameter Black-Box Optimization Benchmarking 2010 : Experimental Setup. Technical report, INRIA, 2010.
77. Das, S.; Suganthan, P.N. Problem definitions and evaluation criteria for CEC 2011 competition on testing evolutionary algorithms on real world optimization problems. Technical report, Jadavpur University, Nanyang Technological University, Kolkata, 2010.
78. Holm, S. A simple sequentially rejective multiple test approach. *Scand J Statistics* **1979**, *6*, 65–70.
79. Rey, D.; Neuhäuser, M. Wilcoxon-Signed-Rank Test. In *International Encyclopedia of Statistical Science*; Springer Berlin Heidelberg, 2011; pp. 1658–1659. doi:10.1007/978-3-642-04898-2_616.
80. WILCOXON, F. Individual comparisons of grouped data by ranking methods. *Journal of economic entomology* **1946**, *39*, 269. doi:10.1093/jee/39.2.269.
81. Mann, H.B.; Whitney, D.R. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* **1947**, *18*, 50–60. doi:10.1214/aoms/1177730491.
82. Middleton, D.; Rice, J.A. Mathematical Statistics and Data Analysis. *The Mathematical Gazette* **1988**, *72*, 330. doi:10.2307/3619963.
83. Tango, T. 100 Statistical Tests. *Statistics in Medicine* **2000**, *19*, 3018–3018. doi:10.1002/1097-0258(20001115)19:21<3018::AID-SIM594>3.0.CO;2-U.
84. Frey, B.B. Holm's Sequential Bonferroni Procedure. In *The SAGE Encyclopedia of Educational Research, Measurement, and Evaluation*; SAGE Publications, Inc.: 2455 Teller Road, Thousand Oaks, California 91320, 2018; pp. 1–8. doi:10.4135/9781506326139.n311.
85. Mudholkar, G.S.; Srivastava, D.K.; Thomas Lin, C. Some p-variate adaptations of the shapiro-wilk test of normality. *Communications in Statistics - Theory and Methods* **1995**, *24*, 953–985. doi:10.1080/03610929508831533.
86. Cacoullos, T. The F-test of homoscedasticity for correlated normal variables. *Statistics & Probability Letters* **2001**, *54*, 1–3. doi:10.1016/S0167-7152(00)00189-9.
87. Kim, T.K. T test as a parametric statistic. *Korean Journal of Anesthesiology* **2015**, *68*, 540. doi:10.4097/kjae.2015.68.6.540.
88. Cressie, N.A.C.; Whitford, H.J. How to Use the Two Samplet-Test. *Biometrical Journal* **1986**, *28*, 131–148. doi:10.1002/bimj.4710280202.
89. Zimmerman, D.W.; Zumbo, B.D. Rank transformations and the power of the Student t test and Welch t' test for non-normal populations with unequal variances. *Canadian Journal of Experimental Psychology/Revue canadienne de psychologie expérimentale* **1993**, *47*, 523–539. doi:10.1037/h0078850.

90. Zimmerman, D.W. A note on preliminary tests of equality of variances. *British Journal of Mathematical and Statistical Psychology* **2004**, *57*, 173–181. doi:10.1348/000711004849222.
91. Caraffini, F. Novel Memetic Structures (raw data & extended results), 2020. Mendeley Data, doi:10.17632/6st7grtxfr.2.
92. Caraffini, F.; Neri, F. Raw data & extended results for: a study on rotation invariance in differential evolution, 2019. doi:10.17632/psp65d2nbc.1.