Scriptie _af_met_goede_titel.pdf

by Rens Anderson

Submission date: 24-Aug-2022 03:24PM (UTC+0200)
Submission ID: 1886391054
File name: Scriptie__af_met_goede_titel.pdf (1,015.9K)
Word count: 5715
Character count: 29778

Opleiding Informatica and Economie

Investigating the difference among implementations of Particle Swarm Optimisation and Differential Evolution and their impact on empirical performance

Rens Anderson

Supervisors:
Hao Wang
Diederick Vermetten

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

24/08/2022

Abstract

Differential evolution and particle swarm optimisation problems are examples of natural computing. These problems require an algorithm to solve them. This thesis analyses different implementations of these algorithms based on their performance regarding 24 commonly-used objective functions. The best- and worst-performing implementations are selected and analysed. The codes of these implementations are compared to each other as well to the pseudocode of a typical solution for these problems in order to ascertain whether differences exist between these implementations and identify the impact of such differences. In this research, several implementations have been found that differ from each other and after comparing these implementations, it can be concluded that the differences in the implementations are significant. This is because one implementation differs from another in how frequently it updates the algorithm's key variables.

Contents

1 Introduction 1
- 1.1 Research questions . . . 1

2 Related work 2
- 2.1 Iterative Optimisation Heuristic Software . . . 2
  - 2.1.1 Iterative Optimisation Heuristic Experiments . . . 2
  - 2.1.2 Iterative Optimisation Heuristic Analyser . . . 3
- 2.2 Particle Swarm Optimisation . . . 4
- 2.3 Differential Evolution . . . 5

3 Methods 7

4 Results 8
- 4.1 Particle swarm optimisation . . . 8
- 4.2 Differential evolution . . . 13

5 Conclusions 18

6 Further research 19

References 21

1 Introduction

Optimisation problems are common in a wide range of fields, and as such a variety of algorithm solutions exist. One class of algorithms is part of natural computing, which takes inspiration from biology. A commonly used example is the particle swarm optimisation (PSO) algorithm. This optimisation algorithm was originally inspired by observing shoals of fish and flocks of birds. The shoals and flocks represent a simple entity, referred to as the population. The population consists of a group of particles, which move through a parameter space of some problem or function [Pol07].

Differential evolution (DE) is another commonly used algorithm in natural computing. The search space is the same as for the PSO algorithm. The DE concept is to create random parameter vectors using three population members to generate a new member. The resulting vector is evaluated using an objective function test. The resulting objective function value is compared to the particle with the best objective function value. If the function value of the new parameter is lower than that of the best particle, the previous vector is replaced by the new generated vector [SP95].

Both problems are examples of optimisation problems. The problem is solved when a score is under a minimum or above a maximum benchmark and therefore considered optimal. These problems can be challenging to visualise because they can scale up in n dimensions. Another characteristic of these algorithms is that they are both examples of black-box solvers. In a black-box problem, an algorithm is given some input and therefore returns some output [NH10]. DE and PSO are algorithms for black-box optimisation. These problems form the basis of this research.

Although considerable research has been conducted on DE and PSO, with each study proposing an implementation that provides a solution to determine the optimum, the implementations are diverse. It has not been looked into why these implementations differ and if there is a performance difference.

1.1 Research questions

This research focuses on the DE and PSO algorithms because they are commonly used in natural computing. The goal is to determine whether the performance of DE and PSO implementations differ from each other and, if so, why this difference occurs. Therefore, this study addresses two research questions:

- **What are the most significant differences between implementations of the DE and PSO algorithms?**
- **To what extent do these differences affect the impact of the DE and PSO algorithm?**

The thesis begins by presenting related work 2 before detailing the methods 3 used for the PSO and DE problems. Subsequently, the results 4 of these problems are analysed and discussed. The final section draws and presents conclusions 5 regarding DE and PSO implementations as well as recommendations for further research 6.

2 Related work

The PSO and DE algorithms are examples of iterative optimisation heuristics (IOHs). An algorithm is classified as an IOH when it aims to find the optimum by repeating the same process [CDB18].

2.1 Iterative Optimisation Heuristic Software

IOH software is a program that can analyse the same type of algorithms. These algorithms are black-box optimisation algorithms. Algorithm 1 describes the structure of a typical IOH. For a complete explanation of the pseudocode, please see [CDB18].

Algorithm 1 Typical structure for a IOH algorithm [CDB18]

1: $t \leftarrow 0$
2: $H(1) = \emptyset$
3: **while** termination criterion not met **do**
4: $t \leftarrow t + 1$ Based upon the search history $H(t)$, choose a probability distribution on $\mathbf{N}$ and sample from it $\lambda(t)$
5: Based upon $H(t)$ and $\lambda(t)$, choose a probability distribution $D(t)$ on $S^{\lambda(t)}$
6: From $D(t)$ sample $x^{t,1}, ..., x^{(t,\lambda(t))} \in S$ and evaluate their function values $f(x^{t,1}), ..., f(x^{t,\lambda(t)})$;
7: Build $H(t+1)$ by selecting which of the samples $(x^{t,1}, f(x^{t,1})), ..., (x^{(t,\lambda(t))}, f(x^{t,\lambda(t)}))$ and which of the samples from $H(t)$ to keep in the search history;
8: **end while**

The IOH software consists of two parts: an experimental component and an IOH analyser. The experimental part is used in Python by installing the package IOH. The IOH analyser is written in R; thus, it has a clear user interface [CDB18].

2.1.1 Iterative Optimisation Heuristic Experiments

The IOH software contains several experiments that test how the algorithm reacts. Therefore, multiple dimensions, which are explained later, are used. The goal of these experiments is to gather as much information as possible about the empirical performance of the algorithm. These experiments consist of testing the algorithm on the 24 commonly-used benchmark functions. Table 1 presents the information gained for each algorithm. By testing each algorithm on 24 benchmark functions, a clear conclusion can be drawn regarding how an algorithm performs [SFA13].

Function id	Information gain
1	What is the optimal convergence rate of an algorithm?
2	Is symmetry or separability exploited?
3	What is the effect of multi-modality?
4	What is the effect of asymmetry?
5	Can the search go from the convex hull into the boundary?
6	What is the effect of highly asymmetric landscape?
7	Does the search get stuck under the plateaus?
8	Can the search follow a long path with D-1 changes in the direction?
9	Can the search follow a long path without exploiting partial separability?
10	What is the effect of rotation (non-separability)?
11	What is the effect of constraints?
12	Can the search continuously change its search direction?
13	What is the effect of non-smoothness, non-differentiable ridge?
14	What is the effect of missing self-similarity?
15	What is the effect of non-separability for a highly multi-modal function?
16	Does ruggedness or a repetitive landscape deter the search behaviour?
17	What is the effect of multi-modality on a less regular function?
18	What is the effect of ill-conditioning?
19	What is the effect of high signal-to-noise ratio?
20	What is the effect of a weak global structure
21	Is the search effective without any global structure?
22	What is the effect of higher condition?
23	What is the effect of regular local structure on the global search?
24	Can the search behaviour be local on the global scale but global on a local scale?

Table 1: The explanation of the different objective functions [SFA13]

2.1.2 Iterative Optimisation Heuristic Analyser

The IOH analyser uses a different interface, which is a free software located on the website https://iohanalyzer.liacs.nl/. The website asks the user to load zip file data, which is the same zip file produced during the experimental portion. Afterwards, the IOH analyser scans the data and makes it immediately visible in graphs and tables [HWB22].

2.2 Particle Swarm Optimisation

The PSO algorithm optimises the particles in a parameter space by evaluating the fitness scores of these particles. The particles move individually through velocity (v_i), which is determined by three factors. The first factor is its current velocity. The second factor is based on its fittest location so far. The last factor is based on the particle's social neighbourhood, which describes its communication on the best position with other particles. Therefore, the second factor is the best fitness score of a particle's social neighbours. The velocity of the particle can be described as follows:

Velocity update: $v_i(t+1) = \omega v_i(t) + \psi_1 R_1(x_{S_i} - x_i(t)) + \psi_2 R_2(x_{p_i} - x_i(t))$ [Pol07]

Thus, the velocity depends on its current velocity ($\omega v_i(t)$), its personal best fitness ($x_{p_i} - x_i(t)$) and the best fitness of the social neighbourhood ($x_{f_i} - x_i(t)$). R_1 and R_2 are random variables, and ψ_1 and ψ_2 are constants. This formula is used for each dimension of the problem and for all the particles. Subsequently, the position in the parameter space is updated using the following formula:

Position update: $x_i(t+1) = x_i(t) + v_i(t+1)$ [Pol07]

These formulae form the basis of the PSO algorithm. Besides these formulas, the PSO algorithm consists of updating the personal best and social neighbourhood best parameters, based on their fitness scores. Each iteration of the PSO algorithm uses the variables p_i, g_i, and v_i to calculate the velocity of the subsequent iteration. After that, the current position and the velocity of the subsequent iteration are combined to determine the position for the following round. By contrasting the position and velocity of the individual's personal best with those of the social neighbourhood, the algorithm is able to determine the optimal setting. Therefore, to find this ideal, evaluation scores from each iteration are compared until they are identical to the optimum. The pseudocode of the PSO algorithm describes how this process works:

Algorithm 2 PSO [TB22b]

Randomly initialise particles (x_i) and their velocities (v_i) in the search space
while termination criterion are not met **do**
    for each particle i **do**
        Evaluate the fitness y_i at current postition x_i
        if y_i is better than its person best $pbest_i$ **then**
            update p_i and $pbest_i$
        end if
        if y_i is better than its global best $gbest_i$ **then**
            update g_i and $gbest_i$
        end if
    end for
    for each particle i **do**
        Update the velocity v_i based on p_i, g_i and v_i
        Update the position: $x_i \leftarrow x_i + v_i$
    end for
end while

2.3 Differential Evolution

The DE algorithm improves itself by making new parameter vectors and evaluating these vectors based on their fitness score. A new parameter vector is created by adding the weighted difference vector between two population members to a third vector from the population. The mutation can be described mathematically as:

Mutation: $v_{i,g} = x_{r_{0,g}} + F * (x_{r_{1,g}} - x_{r_{2,g}})$ [TB22a]

The mutant ($v_{i,g}$) is the new parameter, and it depends on the three population vectors ($x_{r_{0,g}}, x_{r_{1,g}}, x_{r_{2,g}}$). The new vector then undergoes the crossover process. The crossover process determines whether the new vector $u_{i,g}$ receives the vector value of the mutation or the parent value and is described in the following formula:

Crossover:

$$u_{i,g} = u_{j,i,g} = \begin{cases} v_{j,i,g}, & \text{if } (rand_j(0,1) \leq Crorj = j_{rand}) \\ x_{j,i,g}, & \text{otherwise} \end{cases} \text{ [TB22a]}$$

Finally, the new vector parameter is evaluated using a fitness function. This fitness score is compared with the parents' score, and the best score is used to determine which parameter vector has the lowest fitness score.

Selection:

$$x_{i,g+1} = \begin{cases} u_{i,g}, & \text{if } (f(u_{i,g}) \leq f(x_{i,g})) \\ x_{i,g}, & \text{otherwise} \end{cases} \text{ [TB22a]}$$

These formulas are used to optimise the parameters in an iterative process. Algorithm 3 describes this process.

Algorithm 3 DE [TB22a]

Create an initial population $x_1, ..., x_{M-1}$ of M random real-valued vectors
Evaluate the fitness of each vector
while termination criterion are not met **do**
    for each vector $x_i \in x_0, ..., x_{M-1}$ **do**
        Select three other vectors randomly from the population
        Apply difference vector to base vector to create variant vector
        Combine vector x_i with variant vector to produce new trial vector
        Evaluate fitness of the new trial vector
        if trial vector has better fitness than x_i **then**
            Replace x_i with the trial vector
        end if
    end for
end while

3 Methods

Multiple DE and PSO algorithm implementations were required to determine whether the implementations were the same. In this experiment, five implementations were chosen per algorithm to compare their performance. These implementations, PSO_1 [Uzi21], PSO_2 [Roo16], PSO_3 [MM18], PSO_4 [Mac18] and PSO_5 [PSO21], were selected after doing a typical Google search using the terms "Particle Swarm Optimization Python Implementation". For the DE, a similar approach was taken, which resulted in the following implementations: DE_1 [Cri21], DE_2 [Mie17], DE_3 [McC21a], DE_4 [DE422] and DE_5 [McC21b]. For comparison purposes, the structure of each experiment remained the same for the individual algorithms. For instance, the parameter settings on the DE algorithm were the same for all five implementations. The different implementations can be found on the following GitHub website: https://github.com/Rens-byte/Natural-Computing-Thesis.git.

The DE and PSO implementations could be analysed using the same method because they are both black-box optimisation algorithms. Therefore, the same approach can be used and then examined to see if the same results emerge. The first step was to experiment with the implementations using the IOH software. To compare these implementations, the hyper-parameters were set with the same values, which are listed in Table 2.

Parameter	**Value**
Dimensions	5
Bounds	[-5, 5]
Population size	50
c1	0.8
c2	0.9
target error	0.2
W	0.5

Table 1: The PSO parameter values

Parameter	**Value**
Dimensions	5
Bounds	[-5, 5]
Population size	30
Differential scaling factor	0.8
Crossover probability	0.6

Table 2: The DE parameter values

When the parameter settings are equal, the evaluation budget must also be the same. The number of evaluation values depends on the number of iterations, and this number could therefore be different when comparing the implementations. The evaluation budget can be manually changed by changing the number of iterations. A crucial premise is that the evaluation budget should be identical if the population size and the number of iterations are both equal. If not, it can be assumed that the implementations differ fundamentally from one another.

The number of iterations is the only parameter that can be changed because it does not impact other variables in the algorithm. Due to the fact that increasing this parameter results in more evaluations, it is an effective method to set the number of evaluations equal to each other.

The rating of the implementations' performance on the 24 commonly-used objective functions determines how distinct they are from one another. The two implementations that were the most dissimilar from one another were compared by analysing the code of the implementation.

4 Results

4.1 Particle swarm optimisation

The evaluation budget must be the same, to conclude that the iterations are different and therefore, the implementations differ from one another. Table 3 describes the number of iterations required for the 5,000 evaluations. These PSO implementations can be divided into two groups. PSO_1 and PSO_4 used 50 iterations, whereas PSO_2, PSO_3 and PSO_5 used around 100 iterations due to the number of loops used inside each algorithm. In PSO_5, the fitness of each particle is evaluated once, as indicated in the pseudocode 2, while PSO_1 is evaluated twice in section 4.1 which confirms that PSO_1 does, in fact, differ from PSO_5.

Name implementations	Iterations	Evaluation budget
PSO_1	50	5050
PSO_2	100	5000
PSO_3	99	5000
PSO_4	50	5000
PSO_5	99	5000

Table 3: The number of iterations and evaluations for the PSO

Figure 1 depicts the performance of each implementation for the 24 noise-free real-parameter single-objective benchmark functions. The y-axis describes the number of evaluations, and the x-axis describes the best value. The implementations' performances were ranked and compared in a heatmap. The colour blue denotes an implementation that did not perform well in comparison with another implementation, whereas the colour red indicates an implementation that performed better in comparison with a different implementation.

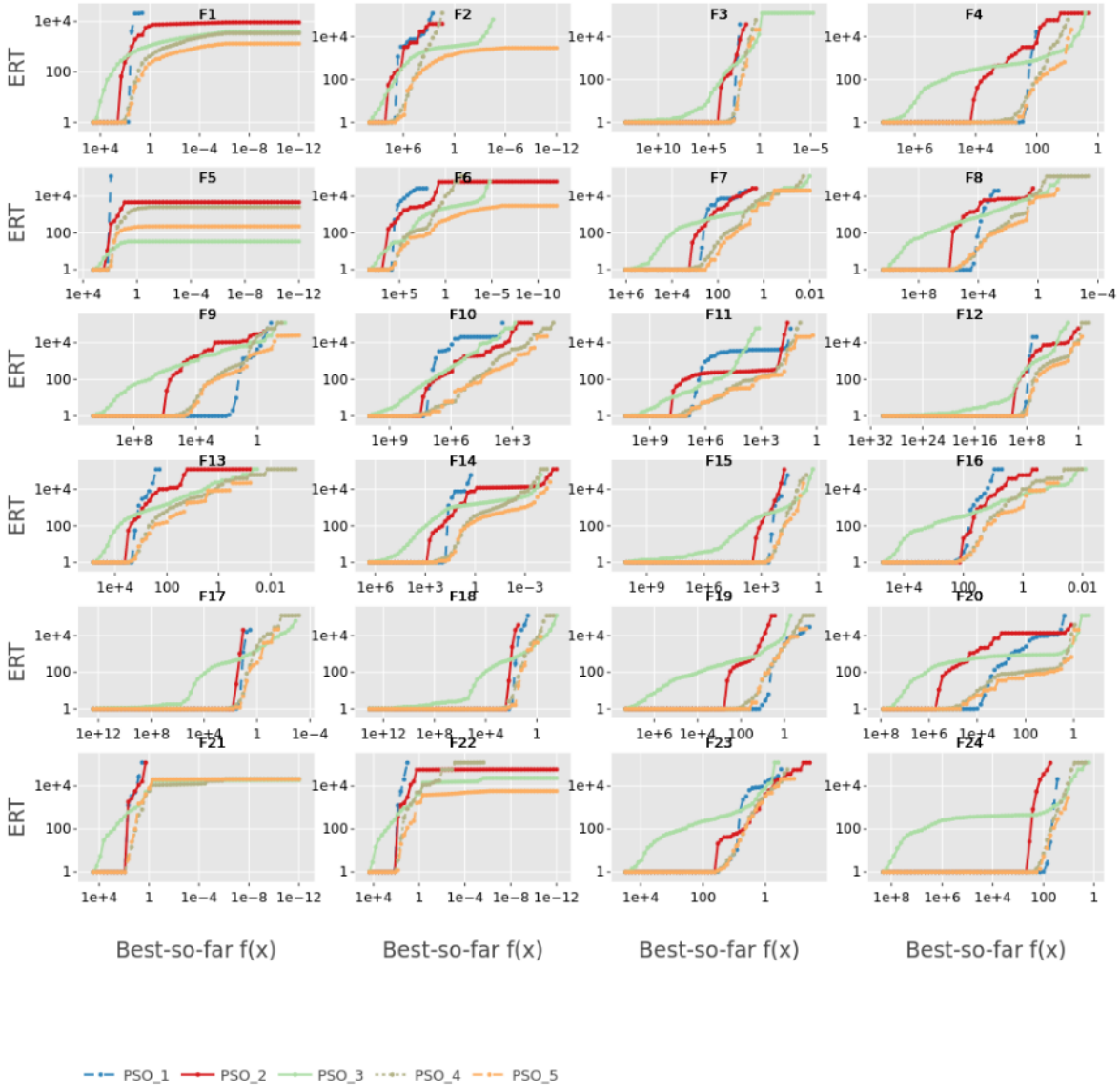

Figure 1: The performance of the PSO implementations on the 24 commonly-used objective functions

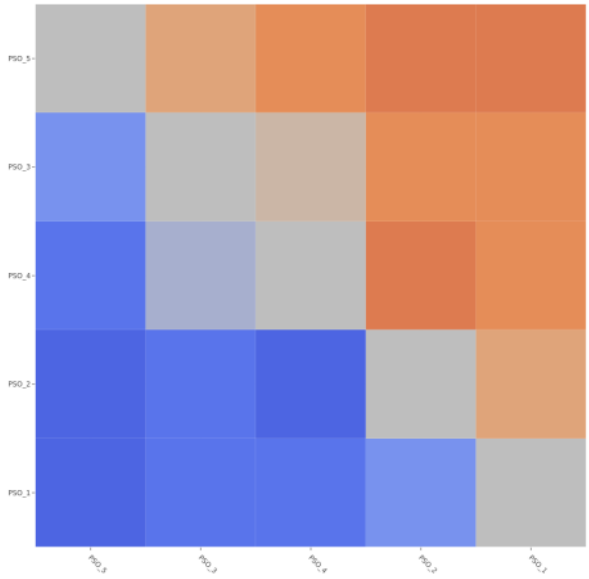

Figure 2: The overall ranking of the implementations on the 24 commonly-used objective functions

To explain why PSO_5 performed better than PSO_1, it is important to compare the algorithms with the pseudocode 2. The PSO pseudocode consists primarily of updating the personal best position and global best position; the velocity and position are subsequently updated. This cycle repeats until no iterations remain. When comparing PSO_5 to PSO_1, this aspect is an area in which most of the implementations are found on this part.

According to the velocity update and position update in 2.2, PSO_5 and PSO_1 use the correct formulae in the code. Each implementation must complete a specific number of iterations, which varies from implementation to implementation, in order to reach the evaluation budget. Therefore, the number of iterations is where the difference in evaluation budget may be detected. This results in the fact that whereas PSO_5 updates the p_i, g_i and v_i twice, PSO_1 only does so once. PSO_5 uses twice the updates to gather the optimal for each of these parameters during every iteration. This method is efficient and effective for obtaining the best possible outcome. The remainder of this section describes part of the PSO_5 and PSO_1 codes. These are the most crucial sections of the code since they describe how the implementation modifies the position update and the velocity update formulae.

```
while Iter < max_iter:
    for i in range(n):
        for k in range(dim):
            swarm[i].velocity[k] = (
                                    (w * swarm[i].velocity[k]) +
                                    (c1 * r1 * (swarm[i].best_part_pos[k] -
                                    swarm[i].position[k])) +
                                    (c2 * r2 * (best_swarm_pos[k]
                                    -swarm[i].position[k]))
                                   )
        ...

        for k in range(dim):
            swarm[i].position[k] += swarm[i].velocity[k]
```

The part of the code where PSO_5 updates its position and velocity

PSO_5 has a clear structure from the outset. It starts with two loops. The first loop is the **while** loop, which contains the same number of iterations as the parameter iterations. The second loop is a **for** loop. This loop analyses the performance of each particle in the swarm. The last step of this loop, and the most important part of the algorithm, is to compare the particle's new position with the particle with the best fitness score. The algorithm updates many variables during the double loop.

PSO_1 has a different format in comparison with PSO_5. A part of the PSO_1 implementation is described in this next section.

```
while error.min() > eps and iter_num < max_iter and count <
early_stopping:
    v = w * v + c1 * r1 * (pbest - p) + c2 * r2 * (gbest - p)
    p = p + v
    ...
```

The part of the code where PSO_1 updates its position and velocity

PSO_1 begins by creating all the variables that will be useful in finding the optimum. Afterwards, the implementation starts with a single **while** loop. This **while** loop includes a few conditions. The first condition is that error.min() > eps. Eps is a parameter in PSO_1 and is small by default. When the error.min is larger than or equal to the last error.min, the result will be 1; otherwise, the result will be 0. The PSO_1 results are only 0, which means it improves at every iteration. The second condition is that $Iter_{num}$ is smaller than max_{iter}. max_{iter} is a parameter that can easily be changed. Every iteration $Iter_{num}$ increases by one, until the max_{iter} is reached. At this point, this condition changes to false. The final condition is that count < $early_{stopping}$. The variables begin updating in the loop, but unlike PSO_5, PSO_1 does not track which parameter has the best value. Instead, it only updates the global and personal best positions. The personal and global best positions are selected by taking arguments from the array error and *errorbest*. These arguments

are not updated repeatedly because the variables *pbest* and *gbest* are outside this loop.

The velocity update and the position update formulae, which were discussed in 2.2, are the two formulae that make up the PSO algorithm. When comparing the implementations, it can be seen that both of these formulae are applied correctly. In addition, the parameters are also set equal, as specified in 3. The only variable that varies depending on the implementation is how many iterations are necessary to attain the evaluation budget. Therefore, it is crucial to understand how many iterations are necessary to attain the evaluation budget in order to determine whether the implementations differ fundamentally from one another. Since PSO_5 has twice as many iterations as PSO_1, it can be assumed that PSO_5 is truly very different from PSO_1. The differences between PSO_5 and PSO_1 are the number of iterations and therefore, the number of p_i, g_i and v_i updates. Because PSO_5 may evaluate twice as much as PSO_1 as a result, PSO_5 outperforms PSO_1 for the 24 commonly-used objective functions.

4.2 Differential evolution

To draw the conclusion that the iterations in the implementations are distinct from one another, the evaluation budget must be the same. Table 4 details the iterations and evaluations for the DE implementations. The implementations were split into two groups. The first group consisted of DE_2, DE_3 and DE_5, and the second group consisted of DE_1, DE_4. The number of iterations for the first group was twice as large as for the first group. Consequently, the fitness of each particle in the first group is evaluated once, as specified in the pseudocode 3 while the second group is evaluated twice. This occurs at the section of the implementation where Mutation, Crossover, and Selection are used, which shows us how the first group and the second group are different.

Name implementation	**Iterations**	**Evaluation budget**
DE_1	83	5010
DE_2	166	5010
DE_3	166	5010
DE_4	82	5014
DE_5	166	5010

Table 4: The number of iterations and evaluations for the DE

The second group achieved better result than the first group because they attained a better f(x) value in fewer function evaluations. Figure 3 presents these results, based on the performance of the implementations for the 24 commonly-used objective functions. The various implementation performances were compared and ranked; Figure 4 contains the resulting heatmap.

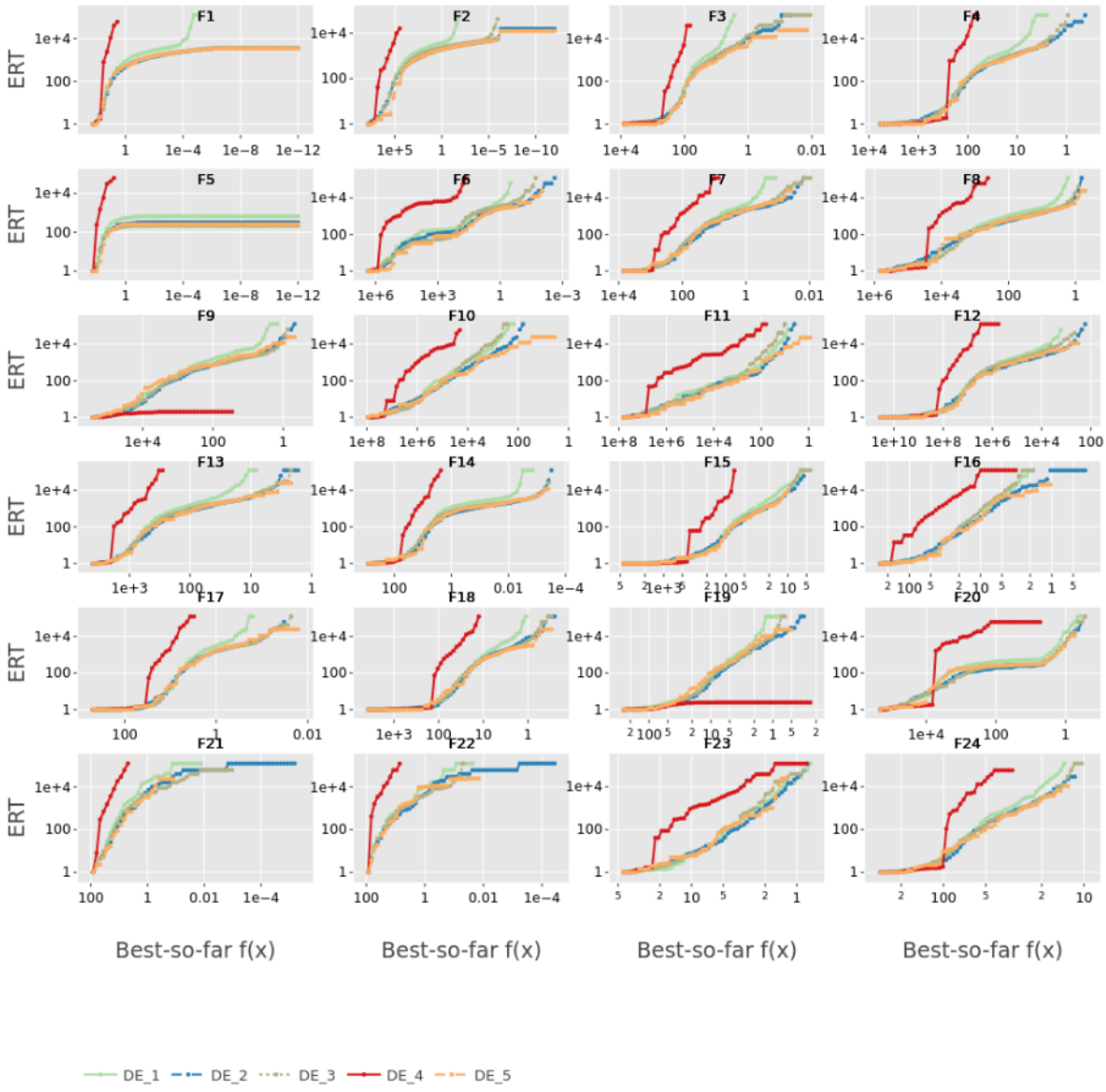

Figure 3: The performance of the DE implementations on the 24 commonly-used objective functions

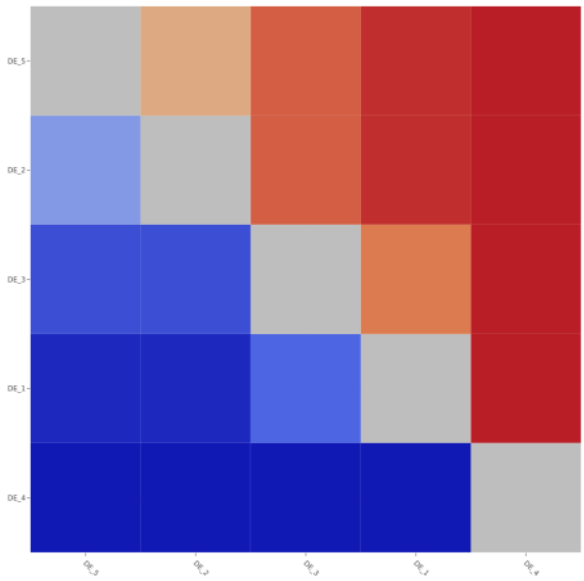

Figure 4: The overall ranking of the DE implementations on the 24 commonly-used objective functions

The implementations were compared to determine whether they differed and, if so, how this difference impacted the algorithm. According to Figure 4, DE_5 was the best-performing implementation, while DE_4 was the worst-performing implementation.

DE_5 is analysed based on how this implementation uses its Mutation, Crossover and Selection. This part is described in the next section.

```
for g in range(max_gen):
    for i in range(pop_size):
        ...
        a = indices[0]; b = indices[1]; c = indices[2]
        mutation = population[a] + F * \
            (population[b] - population[c])
        ...
        new_soln = np.zeros(dim)
        for k in range(dim):
            p = np.random.random()
            if p < cr:
                new_soln[k] = mutation[k]
            else:
                new_soln[k] = population[i][k]
```

The Mutation, Crossover and Selection sections in the DE_5 implementation

An analysis of DE_5 based on part of its code revealed that it begins with two loops to iterate over every vector in the population size. In a subsequent third loop, three random vectors are selected to create a mutation. Afterwards, the mutation and the new candidate vector are checked to ascertain whether the evaluation value of the new candidate has a better fitness score than the best vector so far. If so, the current best vector is replaced by the new candidate. This process continues until no vectors are left in the population. This code is similar to the pseudocode 3 and is therefore highly effective.

DE_5 is very similar to the Pseudocode 3, while the creation of the sections Mutation, Crossover and Selection of DE_4 has a different structure. DE_4 looks as follows:

```
for i in range(POPULATION):
...
    while i < MAX_ITER:
        i += 1
        k = np.random.randint(0,POPULATION,3)
        v = s + F * (f(s[k[0]]) - f(s[k[1]]))
...
is_cross = np.random.rand(POPULATION * len(lb)).reshape((POPULATION,len(lb)
v = s * (1 - is_cross) + v * is_cross
...
for p in range(POPULATION):
    f_new = f(v[p])
    if(f_new <= fitness[p]):
        s[p] = f(v[p])
        fitness[p] = f_new
```

The Mutation, Crossover and Selection section in the DE_4 implementation

Unlike DE_5, DE_4 consists of two main loops. The first loop goes over the number of vectors in the population, and the second loop goes over the number of iterations. In each iteration, three variables are selected for mutation. Afterwards, the mutation is checked with constraints and crossed to create a new gene. In the second part of the code, the new candidate vector is evaluated. This is where it goes wrong. The fitness function of f_{opt} is compared to the evaluation value of f_{new}. If the fitness score of f_{new} is better than the value of the fitness of the evaluation and f_{opt}, then f_{opt} is replaced by the new candidate vector f_{new}. However, this event rarely transpires because the evaluation process occurs in a different for loop. The **for** loop iterates over the population of vectors and thus repeats the optimisation only 30 times.

The difference between the DE_4 and DE_5 algorithms is their structure. DE_5 starts with a first loop over the max_{gen}, which is equal to the variable Max_{iter} in DE_4. The next loop goes over the population, which equals 30. The DE_4 process is an exact reverse; it first loops over the population and afterwards over the Max_{iter}. What occurs inside these loops is consequently highly important. DE_5 uses these two loops along with a third loop to create mutations and uses these mutations to create new candidate vectors. These candidate vectors are evaluated, and when the evaluation

score is better than the prior best fitness score, the new vector replaces the previous best vector. This method is highly efficient and enables an optimum to be found. In contrast, DE_4 uses the two loops to create a mutation, and afterwards the loop ends. DE_4 is therefore inferior as it fails to evaluate the mutations inside these two loops. The algorithm should create candidates and afterwards evaluate these candidates to determine whether the fitness score of the parameter is better than the current best vector. DE_4 thus has a third for loop, which evaluates the fitness scores of the new candidate vectors in the population. The lack of improvements is logical because the new candidate score is compared to the current best score in a new for loop. The new for-loop iterates over the population from 30, and this is where the function begins to evaluate.

The three acts to be carried out by the DE algorithm are Mutation, Crossover, and Selection, as explained in 2.3. Both implementations make use of the same three formulae. Additionally, both implementations use the same parameter settings, as explained in 3. As a result, the only parameter for DE that can differ between implementations is the number of iterations. To determine whether the differences between the implementations are significant, it is crucial to compare the number of iterations. Because the number of iterations for DE_5 is twice as large as for DE_4, DE_5 can update its Mutation, Crossover, and Selection two times as frequently. The difference in iterations can be explained by examining how frequently the implementation updates the Mutation, Crossover, and Selection formulae. Since DE_5 updates these formulae every iteration, whereas DE_4 primarily does this once every two iterations, DE_5 executes the most crucial part of the algorithm more frequently and thus performs better. Because of this, DE_5 outperforms DE_4 on the 24 most-commonly used objective functions.

5 Conclusions

Algorithm implementations can be compared in many ways. This study opted to compare their performance for 24 commonly-used objective functions. Using this metric, it was possible to rank the implementations and identify the greatest differences. To conclude that they differ in fact, the experiment on both implementations must be the same. As a result, the implementations' parameter settings, evaluation score, and formulae must all be the same. Since the only parameter that may change in each implementation is the number of iterations, it can be inferred that there is a fundamental difference between the implementations if the number of iterations is not equivalent.

The next step was to investigate the cause of these differences. For the PSO algorithm, PSO_5 ranked as the best-performing implementation, while PSO_1 was the worst-performing implementation. The high performance of PSO_5 was due to the number of total iterations combined with the number of velocity and position updates. PSO_5 had 99 iterations, and both the velocity and position were updated each iteration, while PSO_1 only consisted out of 50 iterations. The parameters for position and velocity must accurately update during each iteration. Each iteration involves adjusting p_i, g_i, and v_i to achieve this. Therefore, the parameter velocity should be updated every iteration and used to update the parameter position. The parameter position should be used to update the best particle and global positions when the value of the new position is better than the existing best particle or global position. Since PSO_5 has twice as many iterations as PSO_1, PSO_5 may update the position and velocity formulae two times as frequently. As a result, PSO_5 outperforms PSO_1 on the 24 commonly-used objective functions.

The DE implementations were also ranked based on performance. Accordingly, the best-performing implementation was DE_5, and the worst was DE_4. The reason why DE_5 performed better relates to what occurred inside the iterations. Three genes were initially chosen after the second loop and used for mutation and crossover to perhaps produce a solution. The possible solution was evaluated and compared with the best value achieved. The best value was replaced with the possible solution when this possible solution had a better evaluation score. The next iteration was identical by taking the next gene in the population. The three steps of the DE algorithm—Mutation, Crossover, and Selection—are crucial. To create a more successful algorithm, these steps should be repeated as frequently as possible. Currently, two times as many iterations with DE_5 as with DE_4 are required to achieve the same evaluation budget. As a result, there are twice as many updates in the Mutation, Crossover, and Selection for DE_5 as for DE_4. Therefore, DE_5 outperforms DE_4 on the 24 commonly-used objective functions.

6 Further research

This research has shown that there are significant differences between the frequently searched implementations of the DE and PSO algorithms. As a result, some implementations have greatly outperformed others. The PSO and DE algorithms were the main subject of this study; the other black-box optimisation benchmark problems were not. This could serve as the starting point for further research into whether this holds true for the other black-box optimisation benchmark problems. The second section of the study concentrated on how the implementations varied from one another. This study has demonstrated how important it is to repeat the algorithm's most crucial steps as frequently as feasible. The most important formulae, for instance, need to be modified after each iteration. It was discovered that an implementation performs noticeably worse when given half as many iterations as opposed to an implementation given twice as many iterations. How much an algorithm is impacted by each iteration that it receives might be the subject of future research.

References

[CDB18] F. Ye S. van Rijn C. Doerr, H. Wang and T. Bäck. Iohprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *Cornell university*, 1, 11 October 2018.

[Cri21] S. Cristina. Differential evolution from scratch in python. *Machine Learning Mastery*, 16 June 2021.

[DE422] Introduction of differential evolution algorithm (de) and its implementation in python. *Pythonmana*, 14 May 2022.

[HWB22] F. Ye C. Doerr H. Wang, D. Vermetten and T. Bäck. Iohanalyzer: Detailed performance analyses for iterative optimization heuristics. *Cornell University*, 4, 3 January 2022.

[Mac18] I. Macedo. Implementing the particle swarm optimization (pso) algorithm in python. *Medium*, 24 December 2018.

[McC21a] J. McCaffrey. Differential evolution optimization example using python. *WordPress*, 19 July 2021.

[McC21b] J. McCaffrey. Differential evolution optimization. *Visual Studio Magazine*, 2021.

[Mie17] P. Mier. A tutorial on differential evolution with python. *Pablo Rodriguez-Mier blog*, 5 September 2017.

[MM18] S. Chang M. Mofrad. A bi-population particle swarm optimizer for learning automata based slow intelligent system. *University of Pittsburgh*, 3 April 2018.

[NH10] R. Ros S. Finck P. Pošík N. Hansen, A. Auger. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. *Association for Computing Machinery*, 1:1689–1696, 7 July 2010.

[Pol07] R. Poli. Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, 2008, 30 November 2007.

[PSO21] Implementation of particle swarm optimization. *Geeks for Geeks*, 31 August 2021.

[Roo16] N. Rooy. Particle swarm optimization from scratch with python. 17 August 2016.

[SFA13] R. Ros S. Finck, N. Hansen and A. Auger. Real-parameter black-box optimization benchmarking 2010: Presentation of the noiseless functions. *Working Paper 2009/20*, 13 April 2013.

[SP95] R. Storn and K. Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. *Technical Report TR-95-012. ICSI*, 1995.

[TB22a] D. Vermetten T. Bäck, A. Kononova. Differential evolution week 7 powerpoint. *Universiteit Leiden*, 2022.

[TB22b] D. Vermetten T. Bäck, A. Kononova. Swarm-based intelligence week 5 powerpoint. *Universiteit Leiden*, 2022.

[Uzi21] A. Uzila. Complete step-by-step particle swarm optimization algorithm from scratch. *Towards Data Science*, 4 April 2021.

Scriptie _af_met_goede_titel.pdf

ORIGINALITY REPORT

13%	10%	10%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Anne Auger, Nikolaus Hansen. "A (biased) introduction to benchmarking", Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2022 Publication	2%
2	Natural Computing Series, 2015. Publication	1%
3	jamesmccaffrey.wordpress.com Internet Source	1%
4	www.geeksforgeeks.org Internet Source	1%
5	Submitted to Leiden University Student Paper	1%
6	thesis.univ-biskra.dz Internet Source	1%
7	citeseerx.ist.psu.edu Internet Source	<1%
8	Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, Thomas Bäck. "IOHanalyzer:	<1%

	Detailed Performance Analyses for Iterative Optimization Heuristics", ACM Transactions on Evolutionary Learning and Optimization, 2022 Publication	
9	www.e3s-conferences.org Internet Source	<1 %
10	Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M. Shir, Thomas Bäck. "Benchmarking discrete optimization heuristics with IOHprofiler", Applied Soft Computing, 2020 Publication	<1 %
11	dl.acm.org Internet Source	<1 %
12	Luis Fernando de Mingo López, Nuria Gómez Blas, Clemencio Morales Lucas. "Ant colony systems optimization applied to BNF grammars rule derivation (ACORD algorithm)", Soft Computing, 2020 Publication	<1 %
13	epdf.pub Internet Source	<1 %
14	www.waset.org Internet Source	<1 %
15	Maopeng Ran, Haibin Duan, Xinge Gao, Zhili Mao. "Improved particle swarm optimization	<1 %

	approach to path planning of amphibious mouse robot", 2011 6th IEEE Conference on Industrial Electronics and Applications, 2011 Publication	
16	www.xueshupaper.com Internet Source	<1 %
17	dblp.uni-trier.de Internet Source	<1 %
18	journals.scholarpublishing.org Internet Source	<1 %
19	Submitted to Gulf University for Science & Technology Student Paper	<1 %
20	Imen Jarraya, Laid Degaa, Nassim Rizoug, Mohamed Hedi Chabchoub, Hafedh Trabelsi. "Comparison study between hybrid Nelder-Mead particle swarm optimization and open circuit voltage—Recursive least square for the battery parameters estimation", Journal of Energy Storage, 2022 Publication	<1 %
21	mafiadoc.com Internet Source	<1 %
22	"Parallel Problem Solving from Nature – PPSN XV", Springer Science and Business Media LLC, 2018 Publication	<1 %

23	catalog.uttyler.edu Internet Source	<1%
24	Kuo, Gia-Hao, Jason Sheng-Hong Tsai, Shu-Mei Guo, and Chih-Yuan Hsu. "Fast large-scale image enlargement method with a novel evaluation approach: benchmark function-based peak signal-to-noise ratio", IET Image Processing, 2015. Publication	<1%
25	Rongda Zeng, Zihao Wu, Shengbang Deng, Jian Zhu, Xiaoyu Chi. "Adaptive smoothing length method based on weighted average of neighboring particle density for SPH fluid simulation", Virtual Reality & Intelligent Hardware, 2021 Publication	<1%
26	medium.com Internet Source	<1%
27	pdffox.com Internet Source	<1%
28	Adenilton J. Silva. "Evolving Artificial Neural Networks Using Adaptive Differential Evolution", Lecture Notes in Computer Science, 2010 Publication	<1%
29	Furong Ye, Carola Doerr, Hao Wang, Thomas Back. "Automated Configuration of Genetic	<1%

Algorithms by Tuning for Anytime Performance", IEEE Transactions on Evolutionary Computation, 2022
Publication

30	Tang, H.. "Differential evolution strategy for structural system identification", Computers and Structures, 200811 Publication	<1 %
31	coco.lri.fr Internet Source	<1 %
32	M. Fatih Tasgetiren, Yun-Chia Liang, Mehmet Sevkli, Gunes Gencyilmaz. "Particle swarm optimization and differential evolution for the single machine total weighted tardiness problem", International Journal of Production Research, 2006 Publication	<1 %
33	asco.sourceforge.net Internet Source	<1 %
34	Pratyusha Rakshit, Amit Konar. "Principles in Noisy Optimization", Springer Science and Business Media LLC, 2018 Publication	<1 %

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		