MAMSO (Multi-agents multi-strategies optimiser)

Maurice Clerc

To cite this version:

Maurice Clerc. MAMSO (Multi-agents multi-strategies optimiser). 2021. ⟨hal-03150719⟩

HAL Id: hal-03150719
https://hal.science/hal-03150719v1

Preprint submitted on 24 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

MAMSO (Multi-agents multi-strategies optimiser)

Maurice Clerc (Maurice.Clerc@WriteMe.com)

23rd February 2021

1 What is it, what is it for?

MAMSO is a tool to easily compare population-based stochastic black-box optimisers, and, more important, to combine strategies coming from them. By using it you can generate many other algorithms, and answer to some questions like 'What happens if I add an estimation of distribution algorithm to PSO?', or ' What happens if I alternate the strategies of Differential Evolution and the ones of the Adaptive Population-based Simplex?'.

2 Coming from which algorithms?

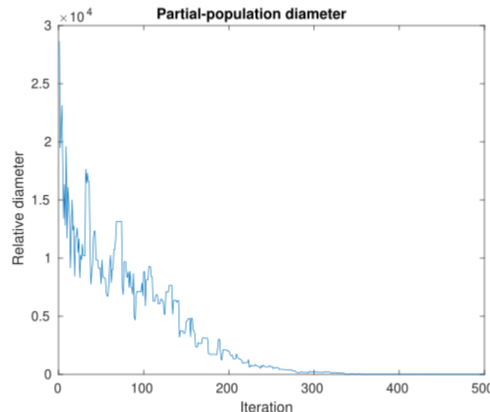
Today the Octave/Matlab MAMSO code contains fifteen strategies or mechanisms, mainly inspired by the following methods (alphabetical order):

- AOA - Archimedes Optimization Algorithm [6]
- APS - Adaptive Population-based Simplex [1]
- DE - Differential Evolution [7]
- EDA - Estimation of Distribution Algorithm (loosely borrowed from CMA-ES [5])
- MPA - Marine Predators Algorithm [4]
- PSO - Particle Swarm Optimisation [9]

However I modified most of these strategies, for example because they were not mathematically consistent or unnecessarily complicated, or using too many user-defined parameters. Also I added a few mechanisms never used in any published methods (to the best of my knowledge). The most important one is an estimation of the progress of the run by computing a partial diameter.

Partial diameter

The population size is N. After each iteration, we consider the set of the $N/2$ agents nearest to the current best, and we compute its diameter. See below a typical evolution of this measure during a run. It can be used it as a criterion to select the next strategy.



The classical method that just compares the number of evaluations to the budget (the maximum allowed) is sometimes inconsistent (see below the comment about MPA) but nevertheless included.

3 Comments on some algorithms

AOA

The original version is mathematically inconsistent. It also has several parameters that can be safely removed. So the version coded here should be called in fact AOA-like. Inconsistency is due to terms like

$$r \times position_{best} - position$$

The moves and the final solution are therefore depending on the coordinate system. The algorithm is not translation independent. Let us see an example.

Problem 1

$$f(x) = x^2,\ x \in [-100, 100]$$

Problem 2

$$f(x) = (x+100)^2,\ x \in [-200, 0]$$

The two landscapes are exactly the same. Now let us run AOA (with Matlab 2018), under the following conditions:

- population size= 25
- number of runs= 15
- number of iterations/run= 5

On the problem 1 the best solution found is 3.327178×10^{-8}. On the problem 2, only 1.55388×10^{-3}.

The reason can be easily seen on the first signature [2] of the figure 1, generated after ten run. It is highly center-biased, and too large (because the code of the confinement method `fun_checkpositions` is not correct).

Sampling outside the definition space

Many algorithms sometimes sample outside the search space, particularly when they use a normal distribution or a Lévy flight. Basically there are three ways to cope with this situation:

1. Do not evaluate the position (keep the previous value). Some PSO variants indeed apply this 'Let it fly' mechanism for, sooner or later, the particle comes back inside the search space.
2. Assign an arbitrary high value. A refinement is even to assign a high value increasing function of the distance to the center of the search space.
3. Force the position to be inside the search space (confinement). There are many variants. For example: a) the updated position can be then on the border, or b) randomly selected, or c) the result of a bounce on the border.

The algorithms on which MAMSO is based use the variant a) of the method 3. Unfortunately, for two original versions (AOA and MPA) this mechanism is wrongly coded, so, sometimes, the updated position is still outside the search space. The bad news is that it does not generate any error with Octave/Matlab, even if the position is outside the *definition* space. Let us consider for example the function $f(x) = \sqrt{x}$. For a negative x the two languages just return a complex number whose real part is null. Not only no error is generated but the comparisons are misleading or, at least, disturbing. For example $f(-9)$ is said to be greater than $f(-4)$.

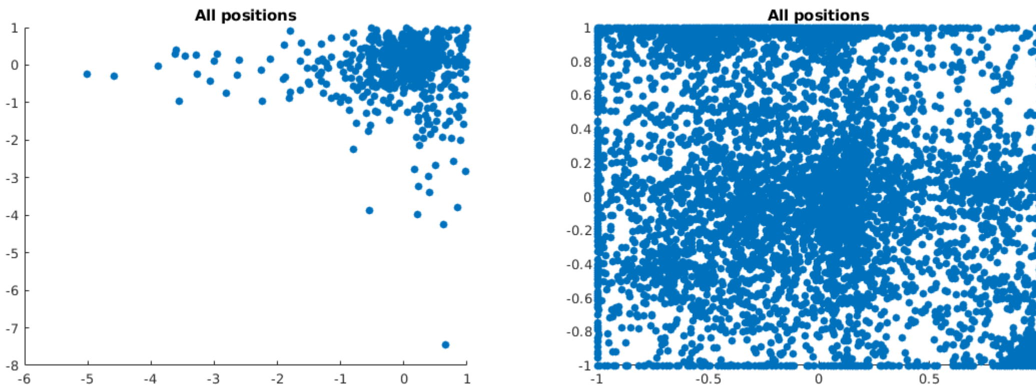


Figure 1: AOA signatures. For the original version, it is clearly highly biased. Moreover the confinement method does not work well, and some points are outside the search space $[-1, 1]^2$. The signature of the AOA-like is far better, although there is now too many positions on the borders, because of large jumps, and the confinement method, which is now correct. Also each run generates a small cluster around a random position, as for any intrinsically convergent stochastic algorithm.

So I modified the formulae so that the algorithm is now consistent (exactly the same result on the two problems), and the confinement mechanism is correct. The second signature is the one of the updated AOA-like version that we can run in MAMSO by selecting the right strategy (code 15). It is now border-biased (because of the jumps and the confinements), and there are ten concentration points (one for each run), because this is a general trivial property of any stochastic convergent optimiser. But their positions are at random inside the search space, which is a good point.

APS

Very similar to the original one (version 12). It uses expansion, contraction, local search, stagnation detection and partial restart.

DE

The formula used in MAMSO is just a DE-like one, combining differences between N pairs of agents randomly chosen (see the section 9), but the whole DE algorithm is not entirely emulated here.

EDA

All stochastic optimisers do use an estimation of distribution to select where to sample, but most of the time *implicitly*. However it is sometimes explicit, like say in CMA-ES. So I added such a strategy, but only thanks to a very simple formula (see the Pieces of code section 9)

MPA

The original version is mathematically inconsistent, like for AOA. And like for AOA the confinement method is wrong, some points are evaluated outside the search space.

On the problem 1 the best solution found is 4.027×10^{-11}. On the problem 2, only 1.117×10^{-6}. And, again, there is a clear difference between the signatures of the original MPA and of the MPA-like version defined in MAMSO.

There is another feature that can be seen as a drawback, but it depends on the point of view, so I let it as is: the choice of a given strategy is only depending on the number of iterations compared to the maximum allowed one (budget). It implies that increasing the budget, the search effort, may sometimes lead to a *worse* final result.

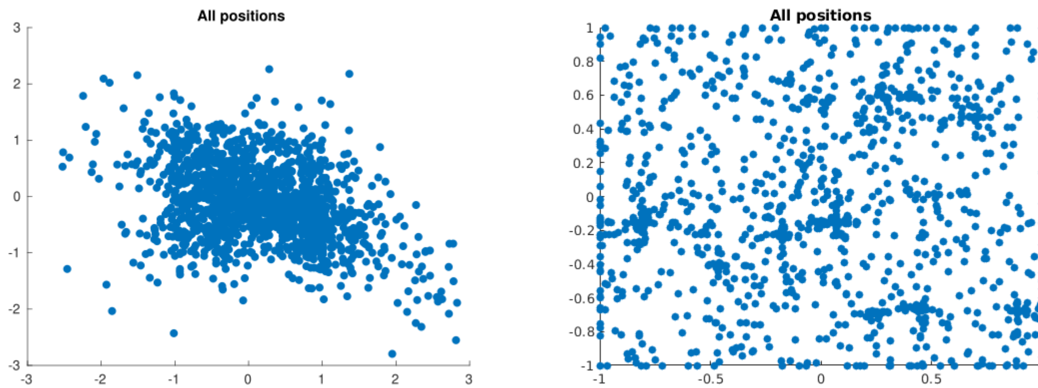


Figure 2: MPA signatures. For the original version, it is clearly center-biased, and many points are sampled outside the search space. Not anymore with the modified one, although there are now some positions on the borders, because of the Lévy flights and the confinement method. Also, as usually, each run generates a cluster around a random position.

Example

Problem CEC 2019 F07, population 25, 15 runs

Iterations	Min	Median	Mean
500	24.19316	65.12419	62.39378
600	128.614	478.7456	480.8439

If you carefully examine what happens it appears than in the second case the last phase of MPA, that can be vaguely seen as a kind of exploitation, is triggered too soon, and the best agents are in fact trapped in the attraction basin of a local optimum.

Fortunately it apparently does not happen very often.

3.1 PSO

The original version is asynchronous. This one is synchronous (positions are evaluated *after* all moves computed during an iteration. With the asynchronous version the convergence is usually quicker, but also sometimes *too* quick.

The topology is randomly chosen, as in SPSO 2007 (Particle Swarm Central [9], Programs section), but slightly different. See the details in the Pieces of codes section (9).

4 Structure and principles

The code of MAMSO is quite long, because it contains so many possibilities, but its structure, which can also be seen as a flowchart, is simple:

1. User-defined parameters, including the sets of strategies to use. Note there are two sets: one for the step 3 and one for the step 4 .
2. Prepare strategies (mainly some initialisations).
3. Use one strategy inside a loop on agents, to move them.
4. Use some strategies after the loop on agents, and evaluations.
5. Select a next strategy to use and back to 3, until the budget is exhausted (maximum number of evaluations).

It implies that the behaviour is synchronous. On the one hand it is not always the best approach (as said for PSO) but, on the other hand, the synchronous approach can easily be distributed.

Thanks to this structure, it is not difficult to add more strategies/mechanisms.

An obvious drawback, though, is that the algorithm is slow. Not only because it is Octave/Matlab coded, but because of the high number of 'if ... then' and 'switch ... case ...' in it. Moreover I sometimes do not use too specific Matlab instructions, so that the code could be more easily translated into a more efficient language.

Also, when the 'local search' strategy is triggered, we can use a method natively proposed by the language. That is why, in such a case, the results are not exactly the same with Octave and with Matlab. Moreover it means that the search is not always really local. And if you translate into another language, you have to replace the local search by another one. So, as proof of concept, I coded a rudimentary local search method (`localOption=1`).

Note that, though, as we will see, some set of strategies can be efficient without any explicit local search.

5 How to use MAMSO

To explain how to use this tool I consider here just one small problem: solving a Loaded Die, more precisely the Brandeis Dice one ([10]) by the maximum entropy method.

MaxEnt and the Brandeis Dice problem

On a fair die, the probability is 1/6 for each side. Therefore the expectation (mean value) is

$$E = \frac{1}{6}\sum_{i=1}^{6} i = 3.5$$

Now, let us suppose we know that it is 4.5. What are the 'most probable' probabilities p_i for the sides? Depending on what you mean by 'most probable' there are several approaches, in particular the Bayesian one and the MaxEnt (maximum entropy) one. I choosed here MaxEnt, and therefore, as MAMSO is looking for a minimum, the objective function is simply the opposite of the entropy:

$$\min\left(S = \sum_{i=1}^{6} p_i \ln(p_i)\right)$$

under the conditions

$$\sum_{i=1}^{6} p_i = 1$$

$$\sum_{i=1}^{6} i p_i = 4.5$$

The Jaynes' MaxEnt solution presented in [10] is $S = -1.61278$. Quite good, but we can do better.

Remember we consider only stochastic methods, so we have to use a random number generator (RNG). For reproducible results the MAMSO code always uses the same seed. I could have added a generator like KISS ([8]) whose code is quite simple but I did not, and MAMSO calls either an Octave or Matlab RNG. Therefore, again, depending on what language you use, results are different, although they should be statistically equivalent.

Let us suppose we want to emulate PSO with local search.

The needed parameters are (Octave/Matlab notation):

```
c = 0.5 + log(2);
```

```
w=1/(2*log(2));
```

The code of the strategy to use in step 3 is 6:

```
sequenceList=[6];
```

The code for local search (step 4) is 9:

```
outLoop=[9]
```

We can specify that the local search is not performed at each iteration, but about every ten time:

```
localFrequency=0.1;
```

Then we launch MAMSO like this

```
mamsoloop(100,25,600,'Dice')
```

meaning 100 runs with a population of 25 agents, a budget of 6000 evaluations for each run, on the Brandeis Dice problem.

It finds a solution whose opposite is a maximum entropy

```
1.61257
```

with the six probabilities

```
[0.06101866  0.07642109  0.1098024  0.1553628  0.22490897  0.3483054].
```

So, we can emulate several 'classical' algorithms in order to compare them. But it is more interesting to create new algorithms by trying some other sets of strategies/mechanisms. With the fifteen strategies already implemented we could think that about 30,000 such new methods can be emulated ($2^{15}-1$). However, on the one hand, the real number is far higher, for you can specify something like that

```
sequenceList=[6, 7*ones(1,5), 8*ones(1,5), 9]
```

which means

- one iteration with strategy 6
- then five with strategy 7
- then five with strategy 8
- then one local search

And, on the other hand, as explained in the section 7, if there are some interesting synergies, there also are many incompatibilities, so, finally, the number of interesting methods is probably quite small (it is difficult to be more precise, though). The table 1 presents some of them. With the best combination of this table, the solution found is `1.612942` on `[0.05932825 0.07346170, 0.1116672, 0.1675561, 0.2393993, 0.3485876]`.

Note that with the same budget some strategies are designed to work well with many agents and few iterations, and even sometimes only on relatively high dimension problems, like APS, and for some others it is the contrary: small population and many iterations, mainly for low dimension problems.

So, using the same population size for all methods is not really fair, but here I just want to illustrate how to use MAMSO.

Table 1: Comparisons of MaxEnt solutions of the Brandeis Dice problem. For all set of strategies the number of runs is 100, the population 25, the budget 6000 evaluations/run. When there is no local search the number of iterations is $6000/25 = 240$. If the local search strategy is in the list, it is performed after each iteration, except if `localFrequency` is explicitly given. The instruction `sequenceOption=2` means that the next strategy is selected only if stagnation is detected. Runs with Matlab 2018, and the default local search is fmincom, if not defined as 'coded'.

Description	Strategies	MaxEnt
AOA-like	sequenceList=[15];	1.575393

APS-like	sequenceList=[7 8 12];
	outLoop=[9 10 11];
	sequenceOption=2;
		1.588200

EDA-like	sequenceList=[14];	0.620131
MPA-like	sequenceList=[1 2 3]; outLoop=[4 5 13];	1.578969
PSO-like	sequenceList=[6];	1.597379
AOA-like+local search	sequenceList=[15]; outLoop=[4 9];	1.610661
EDA+local search	sequenceList=[14]; outLoop=[9];	1.609296
MPA-like (simplified)+local search	sequenceList=[1 2 3]; outLoop=[9];	1.604304

PSO-like+local search	sequenceList=[6];
	outLoop=[9];
	localFrequency=0.1;
		1.612570

PSO-like+local search (coded)	sequenceList=[6];
	outLoop=[9];
	localFrequency=0.1;
	localOption=1;
		1.609316

PSO-AOA	sequenceList=[6 15];	1.607403
PSO-EDA	sequenceList=[6*ones(1,200)... 14*ones(1,40)];	1.600295
PSO-MPA	sequenceList=[6*ones(1,200)... 3*ones(1,40)];	1.606186

PSO-AOA+local search	sequenceList=[6 15];
	outLoop=[9];
	localFrequency=0.1;
		1.612942

PSO-AOA+local search (coded)	sequenceList=[6 15];
	outLoop=[9];
	localFrequency=0.1;
	localOption=1;
		1.612139

PSO-EDA + local search	sequenceList=[6*ones(1,200)...
	14*ones(1,40)];
	outLoop=[9];
	localFrequency=0.1;
		1.612493

PSO-AOA-EDA	sequenceList=[6*ones(1,190) ... 15*ones(1,30)... 14*ones(1,20)];	1.609904

PSO-AOA-EDA + local search	sequenceList=[6*ones(1,181)...
	15*ones(1,21)...
	14*ones(1,11)];
	outLoop=[9];
	localFrequency=0.1;
		1.612219

Table 2: CEC 2019. 15 runs of 25 000 evaluations. D is the dimension of the search space.

PSO-AOA +local search (Matlab)	D	Min	Median	Mean	Std. Dev.
Storn's Chebyshev Polynomial Fitting Problem	9	1.000000	1.000000	1.000000	5.934392e-17
Inverse Hilbert Matrix Problem	16	3.126630e+01	2.248882e+02	2.142521e+02	9.312909e+01
Lennard-Jones Minimum Energy Cluster	18	1.000000	1.409135	1.381859	1.056381e-01
Rastrigin's Function	10	3.984877	8.959667	9.627396	5.252601
Griewank's Function	10	1.000000	1.000000	1.005254	9.657571e-03
Weierstrass Function	10	1.007101	2.578166	2.545898	1.592436
Modified Schwefel's Function	10	2.451590e+02	5.989860e+02	6.130508e+02	2.436936e+02
Expanded Schaffer's F6 Function	10	2.392202	3.761145	3.689942	8.853950e-01
Happy Cat Function	10	1.026962	1.042938	1.064724	5.436263e-02
Ackley Function	10	3.579928	2.099713e+01	1.885662e+01	5.658389

Table 3: CEC 2019. 15 runs of 25 000 evaluations.

PSO +local search (Matlab)	D	Min	Median	Mean	Std. Dev.
Storn's Chebyshev Polynomial Fitting Problem	9	1.000000	1.000000	1.000000	0.000000
Inverse Hilbert Matrix Problem	16	1.405335e+01	1.218607e+02	1.792922e+02	1.252293e+02
Lennard-Jones Minimum Energy Cluster	18	1.000000	1.409135	1.354583	1.439605e-01
Rastrigin's Function	10	1.393446e+01	3.206686e+01	3.453766e+01	1.686761e+01
Griewank's Function	10	1.000000	1.039417	1.116483	1.992705e-01
Weierstrass Function	10	1.073839	3.193584	3.684661	1.808272
Modified Schwefel's Function	10	4.660135e+02	1.112152e+03	1.117533e+03	4.499380e+02
Expanded Schaffer's F6 Function	10	3.265620	4.485090	4.209082	5.296619e-01
Happy Cat Function	10	1.082152	1.195531	1.239123	1.293084e-01
Ackley Function	10	2.099181e+01	2.100000e+01	2.099943e+01	2.109053e-03

Table 4: Two methods on the Dice problem. One may think that PSO-AOA is better.

	Min (i.e. -MaxEnt)	Mean
PSO-MPA	-1.606186	-1.359439
PSO-AOA	-1.607403	-1.405675

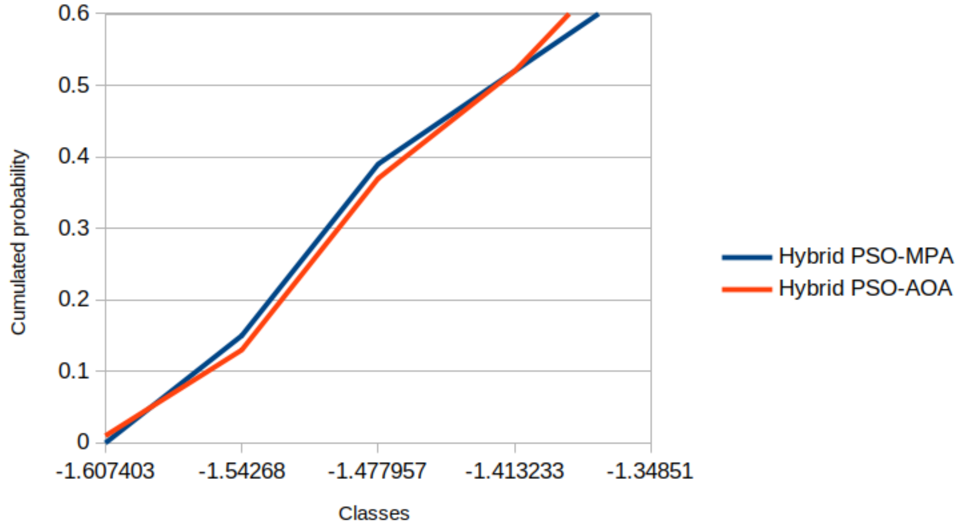


Figure 3: Two CDF for the Dice problem. As the curves cross, what is the 'best' algorithm is depending on the user's requirement.

6 More examples

For the Brandeis Dice problem we were only interested on the best solution (the maximum entropy). However, in practice, the user wants to know what kind of results can be found for a given budget. A classical approach is to consider as criteria not only the minimum found over many runs, but also the median, the mean, the standard deviations, like in the tables 2 and 3and to perform a statistical analysis (Wilcoxon, Friedman, etc.). You can compare these results with the one of some classical algorithms you know: they are not bad at all (remember that for this benchmark, the minimum is always 1).

However I explain in detail in [2] why this approach is not always pertinent and may even lead to wrong conclusions, and why a safer one is to compute the CDF (Cumulative distribution functions) of the best results over a big number of runs, at least 100. I give here just an example on the Dice problem solved by two methods. According to the table 4 it is tempting to claim that PSO-AOA is slightly better.

But the CDF curves show that this conclusion has to be qualified (figure 3). It depends on what is the requirement of the user, for the curves cross two times. If you want a result smaller than -1.606, even with a very small probability, then PSO-AOA is indeed the best. But if you are less demanding and happy with a result just smaller than -1.41 (which is of course far more probable, about half of the time), then PSO-MPA is now the best choice. And if you are even less demanding, PSO-AOA becomes again preferable.

7 A few remarks

If you play with MAMSO you will quickly see that many combinations are in fact very bad. Actually even applying a local search is not always a good idea. This is because it may be applied too often or too early, leading the algorithm towards a local minimum.

Another remark is that is seems difficult to successfully combine more than five or six strategies.

We can note on the Dice problem that the two algorithms that use six strategies (APS and MPA) are not very good (table 1).

On the other hand, this table suggests interesting synergies. For example AOA and PSO alone are not particularly good, but there combination is. And of course the challenge is to find an even better set of strategies.

We could use brute force, by coding a meta-MAMSO that would systematically try many combinations on a given benchmark.

But sometimes it is possible to predict that some combinations will induce a synergy, as for PSO-EDA. It does not call local search, all first iterations are PSO, the last ones are EDA. The reason is that EDA is a greedy strategy, so applying it only at the end of the run performs a kind of local search around the best position found by PSO. When an explicit local search is used, the result is indeed even better, but just a little.

So, before to run a meta-MAMSO it would be useful to define prediction rules in order to seriously decrease the number of combinations to try.

8 How to improve?

There are of course many ways to improve MAMSO. I just focus here on a few important ones.

8.1 Manual strategy selection

After having defined a set of strategies, the main difficulty is to define rules to select which one has to be used for the current iteration.

For example MPA has three strategies (coded here 1, 2 and 3) to apply inside the loop on agents, and three to apply after this loop, coded here 4, 5 and 13. These last three ones are always triggered. But 1 is applied as long as the effort (number of evaluations) is less than one third of the allowed budget, then 2 is applied as long as the effort is less than two-thirds of the budget, and, finally, 3. No adaptation at all and, as we have seen, increasing the search effort may decrease the efficiency.

In APS, there is a kind of adaptation, for the next strategy in the 'in loop' list is triggered only if a stagnation is detected. However this detection is not very satisfying (see the code in 9).

A better approach could be like this:

- For each effort define what a successful iteration is. Indeed, from the user point of view, a 'success' has probably not the same meaning at the beginning of the run and at its end.
- Progressively build a 'profile' for each strategy used, depending on the rank of the iteration and of the (evolutive) definition of 'successful'.
- According to these profiles, select the strategy for the next iteration, in a probabilistic way.

8.2 Automatic strategy selection

A really great improvement would be an automatic adaptive selection in the port-folio of the whole list. There is already something like that, in fact, thanks to the rule 'if stagnation then try the next strategy', but quite rudimentary, for the user has nevertheless to predefine the sequence of the strategies.

Again, to implement such a meta-strategy, some strategy profiles has to be built. Note that it could be partly done *before* the run. Usually we already know that *this* strategy is good to search around a good position, that *this* other one is good to escape a local attraction basin, etc.

8.3 Local search

As said, the 'local search' of Octave or Matlab is not always really local, although it is used here by setting the current best position as starting point. Another method could be used. A rudimentary one is coded in MAMSO (see in the section 9).

Also, defining when the local search has to be triggered should be more flexible. An easy way would be to modify its probability by adapting the localFrequency parameter. Maybe increasing it during the run? This is related to the trade off between exploration and exploitation. For rigorous definitions of these concepts, how to measure them, and how to use them, see [3].

9 Pieces of code

Notations

N=number of agents

D= number of dimensions of the search space. Note that some dimensions can be discrete.

x(n,d)=coordinate of the agent n on dimension d. It means x is the matrix of all positions.

xBest=position of the current best agent (a vector of D values)

FEmax= budget (maximum number of evaluations allowed)

FEs=number of evaluations already done

DE

Actually I call it DE for simplicity, but this strategy is not exactly the one used in Differential evolution.

Moving strategy (step 4 of MAMSO).

Typical parameter values:

aroundCoeff=0.2

progressCoeff1=2

```
u=FEs/FEmax;
prog=u-(N-1)/FEmax;
progress=(1-prog)^(progressCoeff1*prog);
if rand>aroundCoeff
    dx=progress*rand*(x(randperm(N),:) -x(randperm(N),:));
    x=x+dx;
end
```

EDA

Moving strategy (step 3 of MAMSO). Just looking around the best current position, according to a bell-like distribution. If used alone this strategy is of course very greedy, but it can be combined with some others in order to avoid premature convergence to a local minimum.

```
sigm=0.5*diam(iter)*ones(1,D); % Here the partial diameter is used
...
for n=1:N
    for d=1:D
        dxb=sigm(d);
        a=mu(d)-dxb; b=mu(d)+dxb;
        x(n,d)=bellLike(a,b,1,5);
    end
end
function rnd=bellLike(a,b, nRnd, nSum)
% Sample nRnd random numbers from a bell-like distribution
% The higher nSum, the more similar to the Normal distribution
% The support is [a, b]
% The mean is (a+b)/2
```

```
        rnd=zeros(1,nRnd);
        for k=1:nSum
            rnd=rnd+rand(1,nRnd);
        end
        rnd=rnd/nSum; % in [0,1]
        rnd=a+(b-a)*rnd;
    end
```

Random topology in PSO

A typical value for K is 2.

```
    function informers=inform(N,K)
        % Note that each particle informs itself
        thresh=K/N;
        informers=zeros(N,N);
        for i=1:N
            for j=1:N
                if rand<thresh
                  informers(i,j)=1;
                end
            end
            informers(i,i)=1;
        end
    end
```

Stagnation detection

The following method is coming from APS but is not very satisfying, for it depends only on the results of the previous iteration.

```
    function stagn=stagnation(fPrev,f, FEprev, FEs,expBeta)
        % Normalised pseudo-gradient of the fitness evolution
        N=length(f);
        stagna= true*ones(1,N); % Just to speed up a bit
        for n=1:N
            u=abs(fPrev(n)+f(n));
        if u>0
            delta=2*(fPrev(n)-f(n))/u;
            delta=delta/(FEs-FEprev);
            stagProba=expBeta*exp(-delta);
            % Probabilistic decision
            stagna(n)= rand<stagProba;
        end
        s=sum(stagna)/N; % We assume true=1, false=0
        stagn=rand<s;
    end
```

Local search, a rudimentary method

```
function [x,fitness,FEs]=localSearch(x,fitness,FEs,fobj,LB,UB,FEmax,...
quantis,quantOnce, diam, option)
% Here x is the real position, not the normalised one
    [N,D]=size(x);
    [fCenter,Ind]=min(fitness);
    xCenter=x(Ind,:); % Best known position
    if FEmax-FEs<=N return; end
    TolFun=1.e-6; TolX=1.e-6;
    % Pseudo-gradient local search.
    xTry=xCenter;
    stop=false;
    FEsLocal=0;
    dfBest=0;
    dBest=1;
    stepSign=0;
    step=diam/2; % Rule of thumb, using the partial diameter
    while ~stop
       % Find the best dimension to move along
       for d=1:D
          xTry(d)= xTry(d)+TolX;
          fTry=fobj(xTry); FEsLocal=FEsLocal+1;
          df=fCenter-fTry;
          if df>dfBest % Improvement
            dfBest=df;
            dBest=d;
            stepSign=1;
          end
       xTry(d)= xTry(d)-2*TolX; % Try the opposite direction
       fTry=fobj(xTry); FEsLocal=FEsLocal+1;
       df=fCenter-fTry;
       if df>dfBest
          dfBest=df;
          dBest=d;
          stepSign=-1;
       end
       xTry(d)=xCenter(d); % Back
    end
    % Move
    step=step/2; % Decrease the step size
    if step<TolX
       stop=true;
    else
       xNew=xCenter;
       xNew(dBest)=xNew(dBest)+stepSign*step;
       % Evaluate
       fNew=fobj(xNew); FEsLocal=FEsLocal+1;
       dF=abs(fNew-fCenter);
       fitness(Ind)=fNew;
       x(Ind,:)=xNew;
       stop=FEsLocal>=N || dF<=TolFun;
    end
    FEs=FEs+FEsLocal;
```

```
end
```

References

[1] APS. Adaptive Population-based Simplex. http://aps-optim.info/.

[2] Maurice Clerc. *Guided Randomness in Optimization*. ISTE (International Scientific and Technical Encyclopedia), Wiley, 2015.

[3] Maurice Clerc. Iterative Optimisation : The Questionable Balance Mantra. https://hal.archives-ouvertes.fr/hal-01930529, November 2018.

[4] Afshin Faramarzi, Mohammad Heidarinejad, Seyedali Mirjalili, and Amir Gandomi. Marine Predators Algorithm: A Nature-inspired Metaheuristic. *Expert Systems with Applications*, 152:113377, March 2020.

[5] Nikolaus Hansen. The CMA Evolution Strategy. https://www.lri.fr/ hansen/cmaesintro.html.

[6] Fatma A. Hashim, Kashif Hussain, Essam H. Houssein, Mai S. Mabrouk, and Walid Al-Atabany. Archimedes optimization algorithm: a new metaheuristic algorithm for solving optimization problems. *Applied Intelligence*, 51(3):1531–1551, March 2021. https://doi.org/10.1007/s10489-020-01893-z.

[7] Jouni Lampinen and Rainer Storn. Differential Evolution. In *New Optimization Techniques in Engineering*, pages 124–166. Springer, Heidelberg, Germany, 2004. pressure vessel, gear train, coil compression spring.

[8] George Marsaglia. KISS PRNG. http://zuttobenkyou.wordpress.com/2012/05/01/kiss-2011-version-in-c-and-haskell/, 2011.

[9] PSC. Particle Swarm Central. http://particleswarm.info.

[10] Steven J. van Enk. The Brandeis Dice Problem and Statistical Mechanics. *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 48:1–6, November 2014. https://www.sciencedirect.com/science/article/pii/S1355219814000914.