ORNL/TM-2019/1443

Metaheuristic Optimization Tool

Ilham Variansyah
Jin Whan Bae
Benjamin R. Betzler
Germina Ilas

March 2020

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2019/1443

Reactor and Nuclear Systems Division

Metaheuristic Optimization Tool

Ilham Variansyah

Jin Whan Bae

Benjamin R. Betzler

Germina Ilas

March 2020

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

APSO	adaptive particle swarm optimization
ELS	Elitist Learning System
ESE	evolutionary state estimation
HFIR	High Flux Isotope Reactor
LEU	low enriched uranium
MARS	multivariate adaptive regression splines
MDEM	multidimensional extrapolation method
MOPSO	multiobjective particle swarm optimization
MOAPSO	multiobjective adaptive particle swarm optimization
MOT	metaheuristic optimization tool
ORNL	Oak Ridge National Laboratory
PSO	particle swarm optimization
SA	simulated annealing

ACKNOWLEDGEMENTS

This work is supported by the National Nuclear Security Administration Office of Material Management and Minimization, U.S. Department of Energy. The authors also thank the reviewers of the manuscript William Wieselquist and Shane Henderson at Oak Ridge National Laboratory.

1 INTRODUCTION

The metaheuristic optimization tool (MOT) is a generic python module for solving a real value, multiobjective, multidimensional optimization problem. Users design their optimization problem by defining search variables, simulations, evaluation metric functions, and constraints. Search variables and constraints are straightforwardly added via python object methods, while simulations and metric functions used for search variable set evaluation must be implemented as a python class. Final optimum result(s) are provided in a tabular text file, as well as search history if needed.

As a problem-independent method, metaheuristic optimization is reliable for solving challenging, high dimensional, nonconvex, noncontinuous problems. MOT currently supports two metaheuristic optimization methods: simulated annealing (SA) and adaptive particle swarm optimization (APSO). Except for SA, methods that are currently and will continue to be supported by the MOT must be (1) robust or converging to the same optimum result with any initialization, (2) problem independent or not sensitive to any parameter tuning, and (3) simple.

Currently, the multiobjectivity of an optimization problem is solved via the weighted sum aggregation approach and the Pareto nondominance sorting approach. The weighted sum aggregation approach yields a single optimum aggregated response value corresponding to the specified importance weight configuration, while the Pareto nondominance sorting approach yields a set of Pareto optimal values.

Constraints are categorized into 2 types: penalizing constraints and hard constraints. A penalizing constraint penalizes the evaluation score according to the violation degree. A hard constraint becomes the first priority in optimization; whenever a hard constraint is violated, the optimization problem is switched to minimize the hard constraint's degree of violation.

The remainder of this report is organized as follows. In Section 2, related methodologies and procedures are discussed, including basic concepts of multiobjective optimization, the selected metaheuristic methods, and some specific treatments. In Section 3, a description is provided of MOT's building block and its pythonic usage are presented. Some examples are provided in Section 4. Finally, plans for future development are discussed in Section 5.

2 METHODOLOGY AND PROCEDURE

A basic introduction to metaheuristic method is presented below; and simple, single objective optimization is considered first. An analogy of an optimization scheme as particles moving in a search space is discussed to illustrate how the metaheuristic method works. Then the more advanced metaheuristic methods, and constraint treatments are discussed.

2.1 SINGLE OBJECTIVE OPTIMIZATION

In single objective optimization, the goal is to find a set of search variables (or design parameters),

$$\underline{x} = [x_1, x_2, \ldots, x_I], \tag{1}$$

within a certain search space

$$x_i \in \left[x_i^{min}, x_i^{max}\right], \quad i = 1, 2, \ldots, I, \tag{2}$$

that gives maximum* value of a fitness function:

$$F(\underline{x}). \tag{3}$$

The fitness function $F(\underline{x})$ may be noncontinuous and may also be nonconvex, in which case there exist several local maxima. Given the generality of the encountered fitness function, the problem-independent metaheuristic methods are preferred over the more specialized methods such as gradient descent–based methods.

It is interesting to analogize an optimization problem as a moving particle in a search space. To better visualize this, consider the following single objective, two-dimensional optimization problem. With two search variables, $\underline{x} = [x, y]$ the Beale function is taken as the fitness function:

$$F(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2. \tag{4}$$

The search space is bounded as

$$x, y \in [-4.5, 4.5], \tag{5}$$

and here a minimization problem is considered with the following solution:

$$F(3, 0.5) = 0. \tag{6}$$

The search space is shown in Figure 1. The initial conjecture (x_0, y_0) will be the particle's first location. Guided by an optimization procedure, the particle moves (or jumps), generating new, hopefully better solutions, and it converges into the actual global optimum solution. Figure 1 shows the particle track (blue line) generated by the Newton gradient descent method as the optimizer, which essentially forces the particle to roll down the hill. The considered fitness function here is very well defined and is continuous, with a known gradient, which makes the gradient descent–based method the method of choice. Even so, such a specialized method could still converge into a local minimum, as it depends on the initial conjecture: for example, if the particle were initialized at the upper left corner instead (see Figure 1). Metaheuristic optimization methods are discussed below.

*Minimization problem could be treated similarly by putting minus to the fitness/metric function.

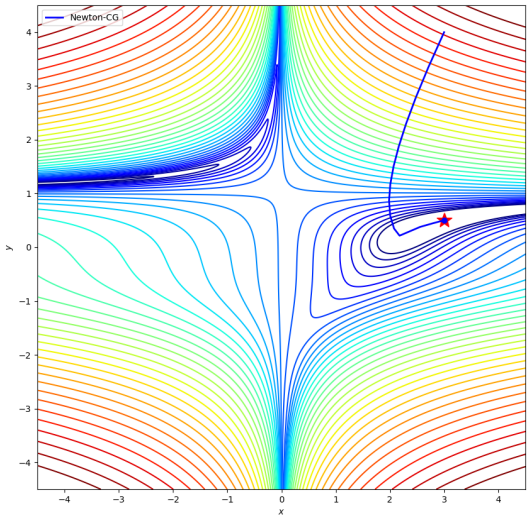

Figure 1. Minimization test problem Beale function solved with conjugate gradient method. The blue contour indicates lower fitness or a better solution. The red star denotes the global minimum. The blue line connected to the star represents the optimization track

2.2 SIMULATED ANNEALING

Simulated annealing (SA) does not include all three criteria mentioned in Section 1, as it relies mostly on parameter tuning. However, due to its simplicity, it serves as a good tool for introducing the metaheuristic method's basic features.

[Step 0] Beginning with a single initial conjecture or particle location,

$$\underline{x}^{(0)} = \left[x_1^{(0)}, x_2^{(0)}, \ldots, x_I^{(0)}\right]. \tag{7}$$

This could be a random point.

[Step 1] At each iteration n, the particle location is randomly perturbed, with a perturbation fraction r:

$$x_i^{(n+1)} = x_i^{(n)} + r\,(\xi - 0.5)\left(x_i^{max} - x_i^{min}\right), \quad i = 1, 2, \ldots, I \tag{8}$$

$$\xi = random[0, 1]. \tag{9}$$

Then the perturbed location is clamped to guarantee that it does not exceed the search space:

$$x_i^{(n+1)} = \min\left[x_i^{max}, x_i^{(n+1)}\right] \tag{10}$$

$$x_{l,i}^{(n+1)} = \max\left[x_i^{min}, x_{l,i}^{(n+1)}\right]. \tag{11}$$

[Step 2] The perturbed location is accepted if it yields better fitness:

$$F(\underline{x}^{(n+1)}) \geq F(\underline{x}^{(n)}). \tag{12}$$

If the result is worse than the previous fitness, then the perturbed location is accepted with a probability p, which depends on the current and previous fitness difference ΔE and the annealing temperature T:

$$\Delta E = F(\underline{x}^{(n+1)}) - F(\underline{x}^{(n)}) \tag{13}$$

$$p = \exp\left(\frac{\Delta E}{T}\right). \tag{14}$$

[Step 3] Then the annealing temperature T is allowed to decay in each iteration:

$$T = T_0 \alpha^n, \tag{15}$$

where $T_0 \geq 0$ and $\alpha \in [0, 1]$.

[Step 4] Return to **[Step 1]** for iteration.

Parameter tuning is important in SA. A large initial temperature T_0 and high-temperature cooling schedule α allow for more exploration in the search space, avoiding possible premature convergence into local optimum. However, making these factors too large results in the search slowly converging to any optimum.

It is also beneficial to anneal the perturbation fraction:

$$r = r_0 \beta^n. \tag{16}$$

This helps in fine-tuning the final solution, yet it gives another parameters to tune. Figure 2 illustrates how SA works, showing that SA failed to find the global optimum with the given initial location and annealing parameters.

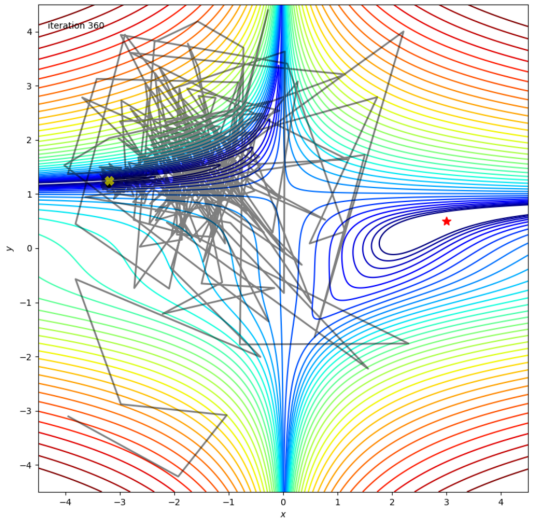

Figure 2. SA applied to test problem Beale function. The gray line represents the search path. The image is taken after 360 iterations. The red star denotes the global minimum. The yellow cross represents the recorded best solution at this stage. The annealing parameters are $T_0 = 1.0$, $\alpha = 0.95$, $r_0 = 1.0$, and $\beta = 0.99$.

Another undesirable feature of SA is that it does not take advantage of the search history. It only compares the last two search solutions. Thus, we consider SA as a trajectory approach. Another approach that takes advantage of more (yet not all) search history is the populational approach, which employs several search particles in parallel. It gathers information from the current particle population and uses that to improve the particle population in the next generation. One of the more recently developed populational methods is the particle swarm optimization (PSO).

2.3 PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO) [1] is inspired by bird flocking and fish schooling. Imagine several particles moving in the search space and communicating with each other, working together to search for the best location. Each particle l carries the following information:

- the current location representing the search variable set $\underline{x}_l^{(n)}$,
- the corresponding fitness value of the current location,
- a personal best location $\underline{x}_l^{(pBest)}$,
- the associated best fitness value, and
- velocity $\underline{v}_l$ that directs the particle to the point to which it will jump in the next generation.

Furthermore, each particle knows the location of the current best particle $\underline{x}^{(gBest)}$ which found the current global best location.

[Step 0] First, the particle locations and velocities (which could be random points and zeros respectively) are analyzed:

$$\underline{x}_l^{(0)} = \left[x_{l,1}^{(0)}, x_{l,2}^{(0)}, \ldots, x_{l,I}^{(0)}\right], \quad l = 1, 2, \ldots, L, \tag{17}$$

$$\underline{v}_l = \left[v_{l,1}, v_{l,2}, \ldots, v_{l,I}\right]. \tag{18}$$

[Step 1] At each iteration n, the velocity of each particle l is updated:

$$v_{l,i} = wv_{l,i} + \xi_{l,i,1}c_1\left[x_{l,i}^{(pBest)} - x_{l,i}\right] + \xi_{l,i,2}c_2\left[x_i^{(gBest)} - x_{l,i}\right], \quad i = 1, 2, \ldots, I. \tag{19}$$

[Step 2] Then the particle locations are updated accordingly:

$$x_{l,i}^{(n+1)} = x_{l,i}^{(n)} + v_{l,i}. \tag{20}$$

The new location is clamped to guarantee that it does not exceed the search space:

$$x_{l,i}^{(n+1)} = \min\left[x_i^{max}, x_{l,i}^{(n+1)}\right] \tag{21}$$

$$x_{l,i}^{(n+1)} = \max\left[x_i^{min}, x_{l,i}^{(n+1)}\right]. \tag{22}$$

[Step 3] Return to **[Step 1]** for iteration.

As with SA, PSO depends on tuning several parameters (w, c_1, c_2) emerging in the velocity update in Equation 19. Considering the parameter in the first term, the particle inertia weight w, the inertia weight determines how much the previous velocity impacts the current velocity. This also serves to suppress the particle movement, leading to a more fine-tuned search. Figure 3 illustrates how inertia weight affects the particle search. With the other parameters set to zero, all particles continue moving forward following their initial velocity while gradually being slowed down by inertia weight.

Now the second parameter, c_1, is added, which is called the *cognitive acceleration*. Figure 4 illustrates how c_1 affects the particle search path. Here the particles are taught to learn their own success. They keep track of their best location so far and then try to return to that location if they happen to move to a worse

location. However, they may not return to that exact location, as randomness has been included in the process. Here, a local search feature is added to each the particle. Given well-spread particles, this parameter introduces an element of effective exploration into the method.

Now an attempt is made to turn off the cognitive acceleration c_1 and activate the social acceleration c_2. Figure 5 illustrates how c_2 affects the particle search path. Social acceleration c_2 differs from cognitive

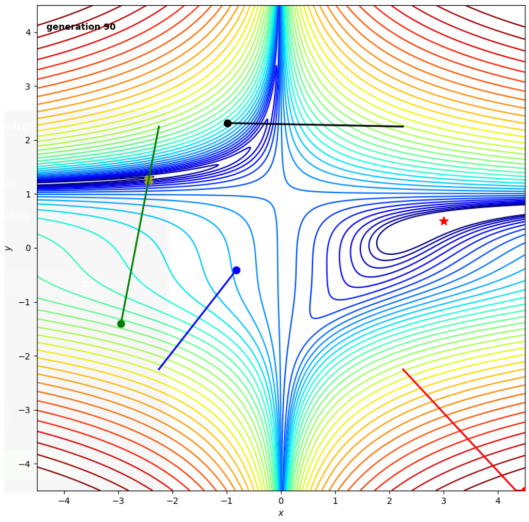

Figure 3. PSO applied to test problem Beale function - weight inertia w significance. The four colors of the circles and lines represent different particles and their search paths. The image is taken after 90 iterations. The red star denotes the global minimum. The yellow cross represents the recorded best solution so far. The search parameters are $w = 0.9$, $c_1 = 0$, and $c_2 = 0$.

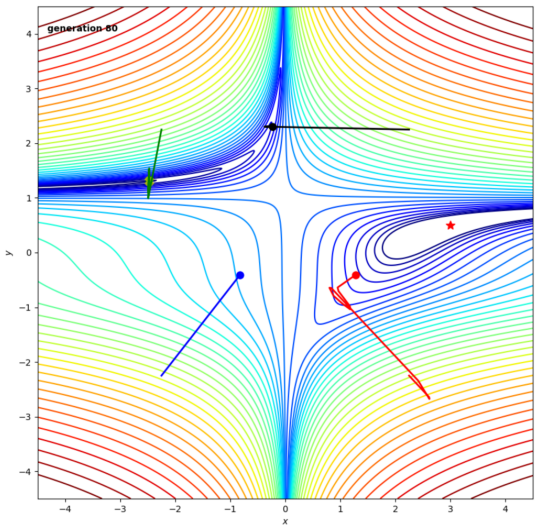

Figure 4. PSO applied to test problem Beale function - cognitive acceleration c_1 significance. The four colors of circles and lines represent different particles and their search paths. The image is taken after 90 iterations. The red star denotes the global minimum. The yellow cross represents the recorded best solution so far. The search parameters are $w = 0.9$, $c_1 = 1.0$, and $c_2 = 0$.

acceleration in that it drives the particles to move toward the global population success, the best particle location. This helps the population to jump and converge to what is hopefully the global optimum. This also adds an element of exploitation around the assumed best location. However, if c_2 is too large relative to c_1, it may lead to premature convergence into the local optimum.

Now the cognitive acceleration c_1 is turned back on to see if both of the accelerations work together. Figure 6 illustrates how these parameters affect the search path together. Interestingly, the red particle was attracted to its own success while it was approaching the global success. It turns away from the current best location, which is a local optimum, and moves toward another best location, which in this case is the global optimum. However, the other particles are not following the better path of the red particle, as they are more attracted to their own success.

Clearly, just like SA, PSO also suffers from parameter tuning. A larger inertia weight w during exploration would be useful, but smaller w is needed during convergence. Cognitive and social accelerations, c_1 and c_2, are based on the tendency of the particle to follow its own success or the global success, respectively. During exploration, larger c_1 and smaller c_2 are desirable, while during convergence, smaller c_1 and larger c_2 are desirable.

Experiments revealed that the recommended values for these parameters are $w \in [0.4, 0.9]$, $c_1, c_2 \in [1.5, 2.5]$, and $c_1 + c_2 \in [3.0, 4.0]$. It is interesting to introduce annealing process to these parameters, but it is not always the case that idealized monotonic transition occurs from exploration into the convergence search mode. Therefore, it would be useful to adaptively change the search parameters according to the particles' current state: exploration, convergence, or something between and beyond. Adaptive particle swarm optimization is discussed in the following section.

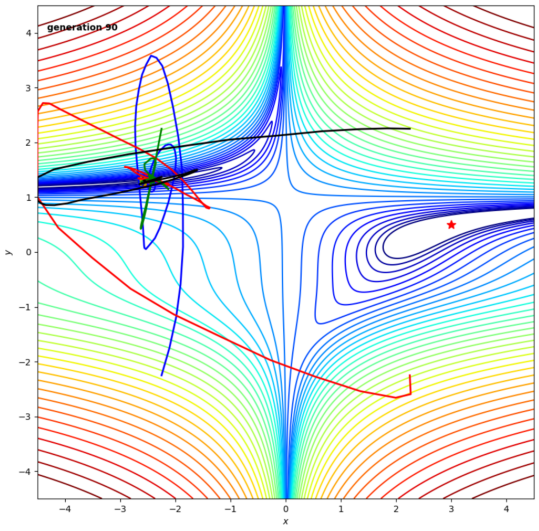

Figure 5. PSO applied to test problem Beale function - social acceleration c_2 significance. The four colors of circles and lines represent different particles and their search paths. The image is taken after 90 iterations. The red star denotes the global minimum. The yellow cross represents the recorded best solution so far. The search parameters are $w = 0.9$, $c_1 = 0.0$, and $c_2 = 0.1$.

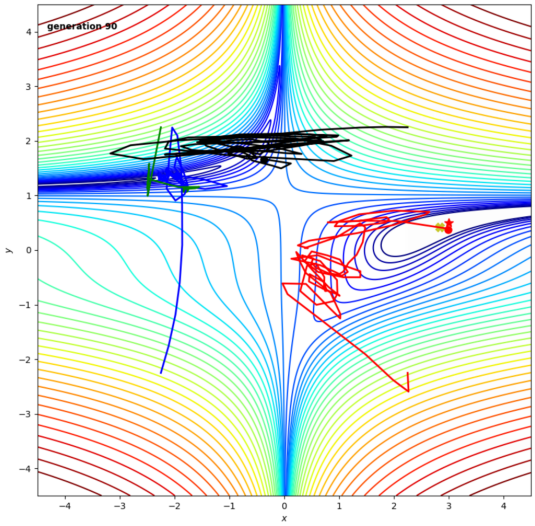

Figure 6. PSO applied to test problem Beale function - combination of cognitive and social acceleration significance. The four colors of circles and lines represent different particles and their search paths. The image is taken after 90 iterations. The red star denotes the global minimum. The yellow cross represents the recorded best solution so far. The search parameters are $w = 0.9$, $c_1 = 1.0$, and $c_2 = 0.1$.

2.4 ADAPTIVE PARTICLE SWARM OPTIMIZATION

In APSO [2], evolutionary state estimation (ESE) adaptively changes the parameters w, c_1, and c_2 leading to an effectively converging method, while the elitist learning system (ELS) carefully searches for other possible, better optima, avoiding premature convergence.

2.4.1 Evolutionary State Estimation

The procedure in this subsection is performed before **[Step 1]** of PSO. Here, evolutionary factor f is introduced. This evolutionary factor is determined from each particle's mean distance to all the other particles d:

$$d_l = \frac{1}{L-1} \sum_{q=1}^{L} \sqrt{\sum_{i=1}^{I} \left(x_{l,i} - x_{q,i}\right)^2} \tag{23}$$

$$f = \frac{d_{best} - d_{min}}{d_{max} - d_{min}} \in [0, 1]. \tag{24}$$

This evolutionary factor represents the current search mode or state of the particles. A larger f indicates that the best particle is remotely located away from the swarm, while a smaller f means that the best particle is surrounded in the swarm (See Figure 7).

A sigmoid mapping function is used to adaptively change the inertia weight according to the evolution factor:

$$w(f) = \frac{1}{1 + 1.5e^{-2.6f}} \in [0.4, 0.9]. \tag{25}$$

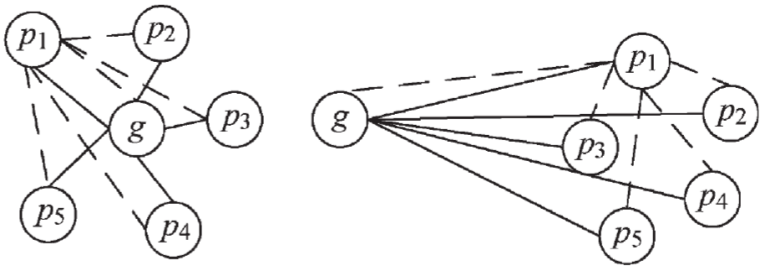

Figure 7. Typical particle locations for small (left) and large (right) evolutionary factor f [2].

Also, evolutionary factor f is used to classify the particle population state into one of the four defined states: jumping out (S_1), exploration (S_2) , exploitation (S_3), and convergence (S_4). The state classification is performed by employing fuzzy membership degree function. Figure 8 shows an experimentally optimized fuzzy membership degree function for ESE. At any given value f, each member group has a membership degree representing the probability that the current state is classified as its member. Those probabilities at any value of f are normalized to one before classification sampling is performed.

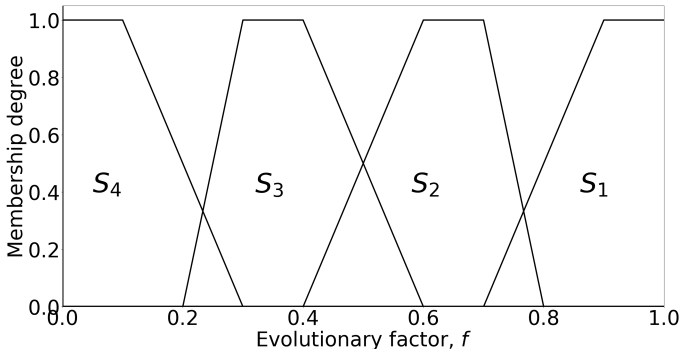

Figure 8. Fuzzy membership degree function employed in MOT. The member groups are jumping out (S_1), exploration (S_2), exploitation (S_3), and convergence (S_4).

Each state has its own strategy for updating the acceleration parameters, as shown in Table 1. The procedure implemented in MOT is as follows:

$$c_i = c_i \pm 0.05(1.0 + \xi), \quad i = 1, 2. \tag{26}$$

A factor of 0.05 is added to the second term for the "slight" changes.

Table 1. Strategy for adaptive change of acceleration parameters c_1 and c_2

No.	Strategy	c_1	c_2
S_1	Jumping-out	Decrease	Increase
S_2	Exploration	Increase	Decrease
S_3	Exploitation	Slight increase	Slight decrease
S_4	Convergence	Slight decrease	Slight increase

2.4.2 Elitist Learning System

This following procedure is performed between **[Step 2]** and **[Step 3]** of PSO. In ELS, the best particle randomly mutates one dimension of its personal best location (the current global best). If it yields worse fitness, then the worst particle will take the mutated position instead. As in SA's annealing perturbation fraction, ELS mutation uses a normally distributed perturbation with the annealing standard deviation, as shown in the following:

$$x_d = x_d + \left(x_d^{max} - x_d^{min}\right) \mathcal{N}\left(0, \sigma^2\right) \tag{27}$$

$$\sigma = \sigma - \left(\sigma_{max} - \sigma_{min}\right) \frac{n}{N}, \tag{28}$$

where the recommended value of maximum and minimum standard deviation are 1.0 and 0.1, respectively.

2.5 MULTIOBJECTIVE OPTIMIZATION

Multiobjective optimization includes several objectives to be optimized:

$$f(\underline{x}) = [f_1, f_2, \ldots, f_K]. \tag{29}$$

However, these objectives usually contradict one another. This gives rise to the Pareto optimality concept, which argues that generally there is no single solution to a multiobjective optimization problem; instead there is a continuous number of solutions, called the *Pareto optimum solution*, in which one objective cannot be improved without reducing another (see Figure 9). The green solid line represents the Pareto front, which is a line of Pareto optimum solutions. This subsection discusses the two most commonly used approaches in addressing multiobjective optimization problem—the weighted sum aggregation approach and the Pareto nondominance approach.

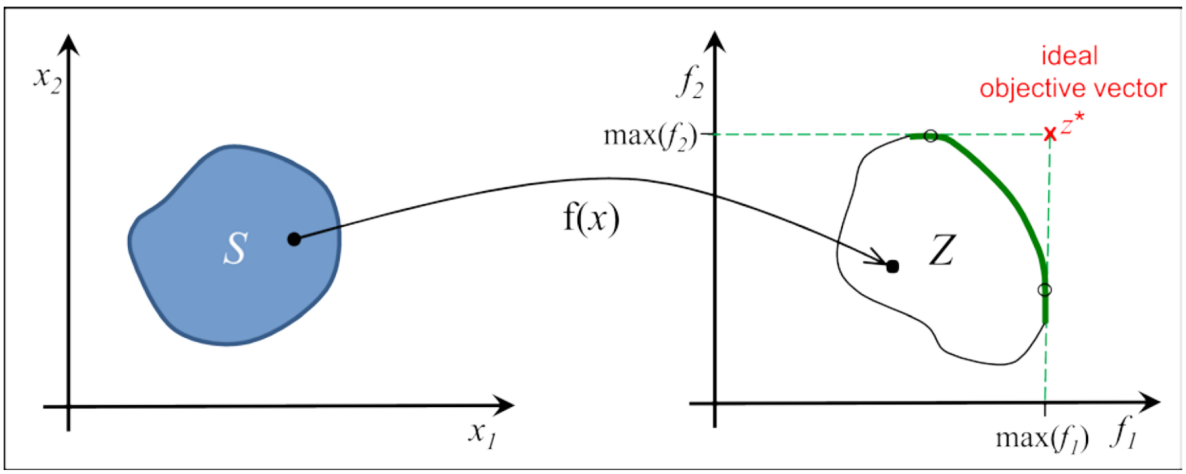

Figure 9. Example of two objective optimizations illustration of the Pareto front [4]. Blue area S represents the search variable. The space labeled with a Z on the right represents the acceptable region of the two evaluation functions. The green solid line represents the Pareto front.

2.5.1 Weighted Sum Aggregation

Weighted sum aggregation may be the simplest, most widely used approach. The objective metric functions $f_k(\underline{x})$ are aggregated into one single response or fitness function $F(\underline{x})$ by assigning an importance weight w_k to each objective:

$$F(\underline{x}) = \sum_{k=1}^{K} w_k f_k^{norm}(\underline{x}) \tag{30}$$

$$\sum_{k=1}^{K} w_k = 1, \tag{31}$$

where the normalized objective metric is

$$f_k^{norm}(\underline{x}) = \frac{f_k(\underline{x}) - f_k^{min}}{f_k^{max} - f_k^{min}}. \tag{32}$$

The user must specify the f_k^{min} and f_k^{max}. It is recommended that the maximum and minimum possible values be used within the search space by running unconstrained single objective maximization and minimization, respectively, for each objective metric. These could be rough searches, as in running the search with small number of generations, computational meshes, probabilistic samples, and/or using regression models to replace expensive evaluations.

This approach reduces the problem to a single objective optimization which can be solved by the SA or APSO methods previously discussed. Given a set of weights, the resulting global optimum of the fitness function $F(\underline{x})$ falls at a point in the Pareto front (See Figure 10). Therefore, with the summed weight aggregation approach, only one Pareto optimum solution can be obtained that is associated with the weight configuration. Other Pareto optimum solutions could be obtained by reconfiguring the importance weight and/or adding the optimization constraints, which will be discussed in Subsection 2.6.

2.5.2 Pareto Nondominance Sorting Approach

In a multiobjective optimization problem, a *nondominated solution set* is a set of all the solutions that are not dominated by any member of the solution set. Dominance in this context denotes that one sample is superior to another sample in every objective. Thus, a solution to a multiobjective optimization problem is not a single point, but is rather a set of solutions along the Pareto front. The nondominated sorting genetic algorithm (NSGA) - II is a multiobjective optimization genetic algorithm that uses an explicit diversity-preserving strategy, along with an elite preservation strategy, to find the Pareto front [5]. With every generation, it uses a nondominated sorting method to rank the solutions to a number of mutually exclusive nondominated sets, or *fronts*. It also attributes a crowding distance for every sample that is evaluated later to maintain diversity (Fig. 11). Then, through a selection method that prefers samples with higher rank and better crowding distance, the superior samples move on to the next generation.

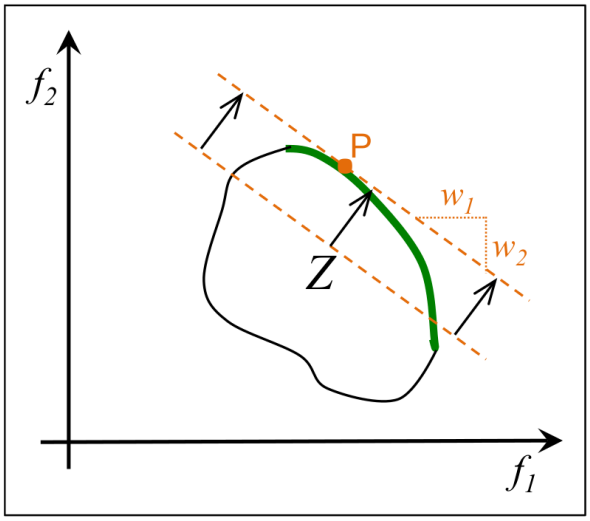

Figure 10. Weighted sum aggregation in multiobjective optimization [4]. The orange solid line illustrates the weight of importance configuration. The diagonal arrow represents the gravity of the optimization search. The orange circle represents the global optimum corresponding to the specified weight configuration.

2.6 HANDLING OPTIMIZATION CONSTRAINTS

This section introduces constraints into the optimization problem:

$$g_j\left(\underline{x}\right) \leq 0, \quad j = 1, 2, \ldots, J. \tag{33}$$

which is typically

$$g_j\left(\underline{x}\right) = A_j - \tilde{g}_j\left(\underline{x}\right) \tag{34}$$

or

$$g_j\left(\underline{x}\right) = \tilde{g}_j\left(\underline{x}\right) - B_j. \tag{35}$$

Note that $\tilde{g}_j\left(\underline{x}\right)$ could be one of the $f_k\left(\underline{x}\right)$.

There are several ways to treat constraint violation; the simplest is perhaps to reject the violating solution by returning the lowest or the worst possible fitness value. However, this treatment does not take advantage of the actual fitness information and the degree of violation, which may lead to a blind random search. MOT employs two approaches:

- the penalizing constraint, and
- the hard constraint.

2.6.1 Penalizing Constraint

The penalizing factor is defined as follows:

$$\delta(\underline{x}) = 1 - \frac{1}{J}\sum_{j=1}^{J} \max\left[0, g_j^{norm}(\underline{x})\right]. \tag{36}$$

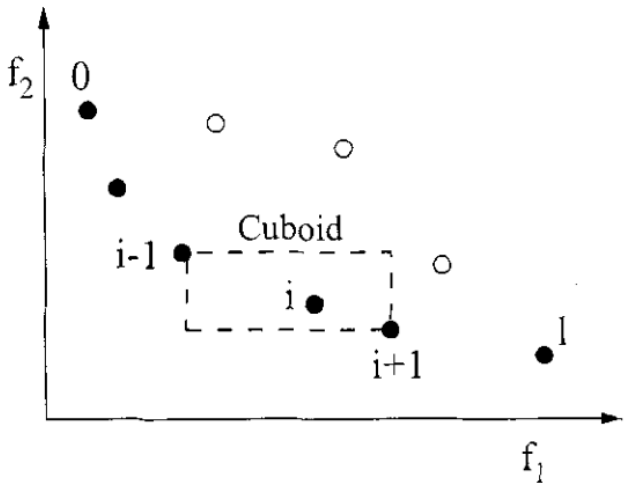

Figure 11. The crowding distance is determined by half of the perimeter of the enclosing cuboid with the nearest neighboring samples in the same front (rank) [5].

This penalizing factor should give 1 if there is no constraint violation and 0 for a complete violation. In weighted sum aggregation approaches (SA,APSO) the penalizing factor is multiplied by the appropriate fitness,

$$F(\underline{x}) = \delta(\underline{x})F(\underline{x}) \tag{37}$$

while in Pareto the nondominance sorting approach (MOAPSO), the penalizing factor is multiplied by each objective metric.

The normalized constraint function $g_j^{norm}(\underline{x})$ is defined as follows:

$$g_j^{norm}(\underline{x}) = \frac{A_j - \tilde{g}_j(\underline{x})}{A_j - \tilde{g}_j^{min}} \tag{38}$$

or

$$g_j^{norm}(\underline{x}) = \frac{\tilde{g}_j(\underline{x}) - B_j}{\tilde{g}_j^{max} - B_j}. \tag{39}$$

Specifying values for $\tilde{g}_j^{min}$ and $\tilde{g}_j^{max}$ is similar to that of f_k^{min} and f_k^{max}, as explained in previous section.

This approach may favor a greater fitness value with a small constraint violation over a lower fitness value without any constraint violation. This could present a problem if the constraint absolutely cannot be violated. Therefore, *hard constraint* is introduced in MOT.

2.6.2 Hard Constraint

Hard constraint becomes the first priority in optimization: whenever hard constraint is violated, the optimization problem is switched to minimize the hard constraint violation degree. The *hard constraint violation degree* is defined as a function, and with this function, a minimization problem is performed if

there is any violation. When there is no hard constraint violation, the usual fitness maximization is used. The violation degree function is defined as the following:0

$$D(\underline{x}) = \sum_{j=1}^{J} \max\left[0, g_j^{norm}(\underline{x})\right] \tag{40}$$

3 METAHEURISTIC OPTIMIZATION TOOL

MOT consists of 3 major python classes collected in the python `MOT` package: `Perturber`, `Fitter`, and `Method`*. The MOT can be obtained from the following git repository: `https://code.ornl.gov/ybb/MOT.git`. The repository is currently accessible to people with an invitation, but is planned to be openly accessible in the near future. This section provides a brief discussion of the object of each class. A more detailed discussion is provided in Section 4, including examples.

The manner in which the objects are connected when performing optimization is depicted in Figure 12. Users define the optimization problem by supplying descriptions of each search variable in `Perturber` and supplying user-defined <`Simulation`> and <`Function`> objects into `Fitter`. Then, users supply the set up `Perturber` and `Fitter` into the `Method`. The parameters of these objects may need to be set up before they are entered. Finally, `Method` solves the optimization problem described by the `Perturber` and `Fitter`, resulting in the optimum variable set.

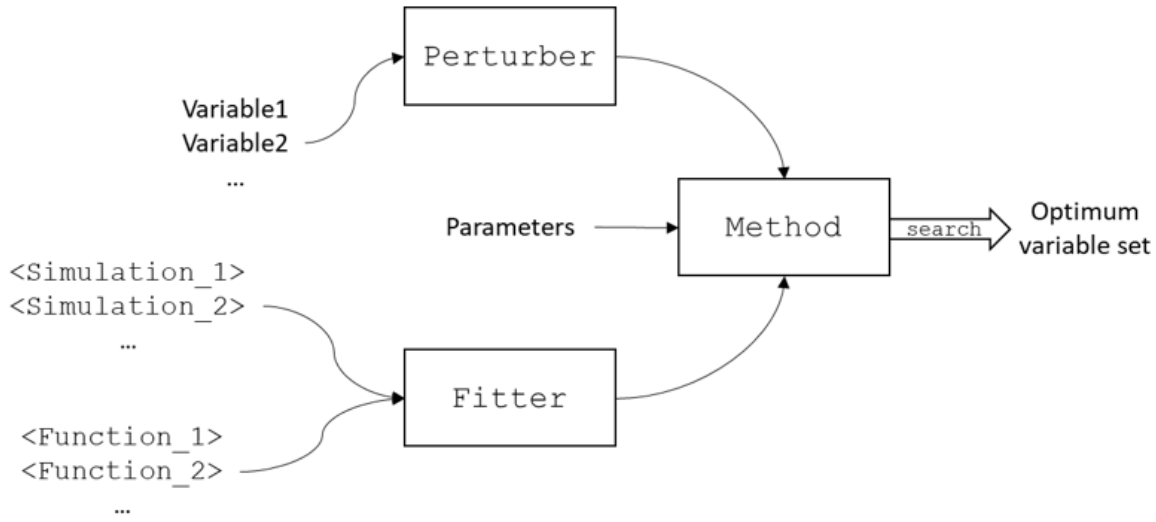

Figure 12. `MOT` objects.

3.1 `Perturber`

The `Perturber` object handles and performs perturbation of the search variables. Each variable supplied must include a label, a minimum value, and a maximum value; supplying the unit is optional. Given these inputs, `Perturber` carries the information of the search variable space. The integer variable is treated by rounding to the given real value.

Python usage - `Perturber`

```
MOT.Perturber.__init__()
```

```
MOT.Perturber.add_variable(label,min,max,unit="",integer=False)
```

*Whose derived class could be any of the supported methods.

- label [string]: label of the search variable
- min [float]: minimum value
- max [float]: maximum value
- unit [string,optional]
- integer [bool,optional]: if the variable is an integer

3.2 Fitter

The Fitter object handles the simulations and functions used to evaluate fitness of the search variable set. Users must define each simulation as python class <Simulation> and each function as python class <Function>. There are two typical methods to configure Fitter:

1. All of the objective and constraint metrics can be explicitly modeled as functions. The <Function> classes are defined, and their object into the Fitter is supplied. It is not necessary to run any simulation or set up any <Simulation> class.
2. While there are several objective and constraint metrics, they can only be obtained by extracting information from simulation results. The simulation is run first, which is implemented as Simulation class. The process typically includes generating an input for a simulation code according to the given variable set and then running the simulation code with the generated code input. Finally the Function is called to extract the metrics from the simulation results.

<Function> class

<Function> is a user-defined python class in which the __call__ method takes a list of variables and returns a scalar metric. This can be an objective and/or a constraint metric. Each <Function> class must have name and unit string attributes, and may have __init__ method. The template or minimum requirement for the Function class is provided below:

```
class <Function>():
    name = <name>
    unit = <unit>

    def __call__(self,<list_of_variables>):
        <procedures...>
        return <scalar_metric>
```

[IMPORTANT] The variables supplied to Perturber must be in the **same order** as the list of variables passed as argument in <Function> and <Simulation>.

<Simulation> class

Simulations are optional and run in order one time at each search iteration evaluation. Simulation is a user-defined python class, and its __call__ method takes a list of variables and performs the simulation procedure based on the given variable set. The <Simulation> class must have a name string attribute, and it may have __init__ method. The template or minimum requirement for the Simulation class is provided below:

```
class <Simulation>():
    name = <name>
```

```
    def __call__(self,<var>):
        <procedures...>
```

Python usage - `Fitter`

`MOT.Fitter.__init__()`

`MOT.Fitter.add_simulation(simulation)`

- `simulation` [<Simulation>]: simulation used for search variable set evaluation.

`MOT.Fitter.add_function(function, weight=1.0,`
`limit_min=−∞, limit_max=∞, limit_hard=True,`
`norm_min=0.0, norm_max=1.0)`

- `function` [<Function>]: objective and/or constraint metric function evaluating the search variable set
- `weight` [`float`,optional]: weight of importance; set to zero if the `function` is a pure constraint metric
- `limit_min` [`float`,optional]: minimum constraint; default means no constraint
- `limit_max` [`float`,optional]: maximum constraint; default means no constraint
- `limit_hard` [`bool`,optional]: the constraint type; `True` means hard constraint, penalizing otherwise
- `norm_min` [`float`, optional]: maximum possible value of the metric for normalization
- `norm_max` [`float`,optional]: maximum possible value of the metric for normalization; default values for `norm_min` and `norm_max` give an un-normalized function in the calculation of combined fitness and violation degree

3.3 `Method`

`Method` is a super-class of the supported optimization method's subclass, `SimulatedAnnealing`, `APSO`, and `MOAPSO`. New methods may be added by defining a new subclass of the `Method` class. The attributes inherited from the subclasses are the following:

- `self.name` [`string`]: name of the method
- `self.populational` [`bool`]: whether the method is populational or trajectory
- `self.var_best` [`list`]: the optimum variable set
- `self.value_best` [`list`]: metrics of the optimum variable set
- `self.fitness_best` [`float`]: fitness of the optimum variable set
- `self.var_stored` [`list`]
- `self.value_stored` [`list`]
- `self.fitness_stored` [`list`]

The first two attributes must be specified by each subclass. The remaining attributes are the outputs of the optimization. The `*_stored` attributes are the optionally stored search history. Several print-out and output handling methods are also inherited to the subclasses.

Sub-class of `Method` class

The template or minimum requirement for the subclass of the `Method` class is as follows:

```
class <TheMethod>(Method):
    name         = <name>
    populational = <True/False>

    def __init__(<args...>):
        <initializations...>

    def search(<Perturber>, <Fitter>, <args...>,
               record=False, file_name="<default_name>",
               report=True, store=False)
        <search_procedures...>
```

In `__init__`, the method parameters are passed as arguments. The `search` method solves the optimization problem described by the passed `Perturber` and `Fitter` objects. Note that several method setups are passed, as well, such as the number of iterations, the number of particles, etc. The remaining arguments are output handlers, which will be discussed in the section on python usage below.

Python usage - `SimulatedAnnealing`

```
MOT.SimulatedAnnealing.__init__(T0, alpha, frac0, beta)
```

- `T0` [`float`]: initial annealing temperature
- `alpha` [`float`]: annealing temperature decay factor
- `frac0` [`float`]: initial annealing perturbation fraction
- `beta` [`float`]: annealing perturbation fraction decay factor

```
MOT.SimulatedAnnealing.search(Perturber, Fitter, Niter, var_init=0,
                              record=False, file_name="SA.out",
                              report=True, store=False)
```

- `Perturber` [`Perturber`]: `Perturber` object
- `Fitter` [`Fitter`]: `Fitter` object
- `Niter` [`int`]: number of iterations
- `var_init` [`list`,optional]: initial variable set. Default means random initialization
- `record` [`bool`,optional]: record search description, history and final results into the specified `file_name`. This record file is being populated while the tool is running so the file can be used for monitoring the running tool
- `file_name` [`string`,optional]: file name for output records. If `record=False`, the file is not generated
- `report` [`bool`,optional]: reports search description and final results to the terminal. Regardless, final results are obtainable from `var_best`, `value_best`, and `fitness_best` attributes.
- `store` [`bool`,optional]: stores search history into `var_stored`, `value_stored`, and `fitness_stored` attributes

Python usage - `APSO`

```
MOT.APSO.__init__()
```

```
MOT.APSO.search(Perturber, Fitter, Np, Ngen,
                record=False, file_name="APSO.out",
                report=True, store=False,
```

```
                plot_parameters=True, Nproc=1)
```

- Perturber [Perturber]: Perturber object
- Fitter [Fitter]: Fitter object
- Np [int]: number of particles
- Ngen [int]: number of generations
- record [bool,optional]: records search description, history, and final results into the specified file_name. This record file is being populated at the end of each generation while the tool is running so that the file can be used for monitoring the running tool
- file_name [string,optional]: file name for output records. If record=False, the file is not generated
- report [bool,optional]: reports search description and final results to the terminal. Regardless, final results are obtainable from var_best, value_best, and fitness_best attributes
- store [bool,optional]: stores search history in var_stored, value_stored, and fitness_stored attributes
- plot_parameters [bool,optional]: generates <file_name>.png file showing APSO parameters (w, c_1, c_2) evolution in generation
- Nproc [int,optional]: number of processors used for parallel evaluation of particles in each generation

Python usage - inspyred_optimize This class leverages the inspyred python package to perform multiobjective optimization using genetic algorithms.

```
inspyred_optimize.search(Perturber, Fitter, algorithm='', workdir='./',
                n=100, maxiter=100, record=True,
                maximize=False)
```

- Perturber [Perturber]: Perturber object
- Fitter [Fitter]: Fitter object
- algorithm [str]: algorithm to use: ['PAES', 'NSGA2', 'GA', 'ES', 'SA', 'DEA', 'EDA', 'PSO']
- workdir [string,optional]: path for the output files
- n [int]: number of population
- maxiter [int]: number of generations
- record [bool,optional]: records search description, history, and final results into the specified file_name. This record file is being populated (at the end of each generation) while the tool is running so that the file can be used for monitoring the running tool
- maximize [bool, optional]: true to maximize the response, false to minimize the response

Python usage - scipy_optimize This class leverages the scipy python package to perform gradient-based optimization.

```
scipy_optimize.search(Perturber, Fitter, algorithm='differential_evolution',
                workdir='./', maxiter=100, record=True,
                maximize=False
```

- Perturber [Perturber]: Perturber object
- Fitter [Fitter]: Fitter object
- algorithm [str]: algorithm to use: ['fmin_slsqp', 'differential_evolution', 'basinhopping', 'brute', 'fmin_cobyla']

- `workdir` [`string`,optional]: path for the output files
- `maxiter` [`int`]: number of generations
- `record` [`bool`,optional]: records search description, history and final results into the specified `file_name`. This record file is being populated (at the end of each generation) while the tool is running so that the file can be used for monitoring the running tool
- `maximize` [`bool`, optional]: true to maximize the response, false to minimize the response

3.4 PYTHON REQUIREMENT

MOT works with either Python2 or Python3. The required packages are the following:

- `conda install numpy`
- `conda install tabulate`
- `conda install matplotlib`
- `conda install -c anaconda scipy`
- `conda install -c conda-forge inspyred`

These modules can be also installed with `pip install`.

4 EXAMPLES

Besides the `MOT` python package file `MOT/MOT.py`, the MOT tool directory also provides several examples collected in `MOT/examples/`. The first examples include optimization of test functions generally used for verification of optimization algorithms. The last two examples discuss two practical cases in which MOT could be useful: (1) high-end design stage with expensive, high-fidelity analysis tools for which some limited parametric studies have been performed, but the design should be improved, and (2) low-front design stage, in which there is still a large degree of freedom with fast, low-fidelity analysis tools.

4.1 THE TEST FUNCTIONS

This subsection provides examples for the following:

- setting up <`Function`> class
- setting up `Perturber` object with search variables
- setting up `Fitter` object with metric functions
- converting minimization into maximization problem
- setting up `SimulatedAnnealing` and `APSO` objects
- performing optimization search with the methods
- configuring output handling arguments: `report`, `record`, `store`
- directly accessing `Method` object attributes, (i.e. final results and stored search history)

`test_functions/` provides scripts for optimization of several unconstrained, single objective, 2-dimensional test functions with known global minimum/maximum:

- Bealef function: has two well-separated minima and sharp peaks at the corners of the search space
  - search space: $x, y \in [-5, 5]$
  - global minimum: $f(3.0, 5.0) = 0.0$
- Easom Function: has several local minima, and the global minimum has a small area relative to the search space
  - search space: $x, y \in [-100, 100]$
  - global minimum: $f(\pi, \pi) = -1.0$
- Rosenbrock Valley: the global minimum lies in a narrow, parabolic valley, making it difficult to converge
  - search space: $x, y \in [-2, 2]$
  - global minimum: $f(1.0, 1.0) = 0.0$
- Shekel Foxholes: has 25 local maxima, testing algorithm robustness
  - search space: $x, y \in [-60, 60]$
  - global maximum: $f(-32, -32) = 499.002$

The problems listed here could be used to verify methods implemented in `MOT`. Examples include running with different random number seeds, different methods, different method parameters, number of iterations, and/or switching the `store` argument ON so that the search history can be obtained and the search process can be animated for better illustration of the optimization process/algorithm.

Things to note:

- The metric functions of the problem are explicitly defined so that <`Simulation`> class set up is not

needed

- The <Function> classes are provided in test_functions/functions.py
- Fitter.add_function calls are at the simplest possible form, because only one function is considered, and there is no constraint subject in this example
- For all problems except for Shekel Foxholes, w=-1 is assigned while supplying the <Function> object to the Fitter, as MOT is designed to solve a maximization problem
- Both SA and APSO are used here for comparison
- report, record, and store are turned OFF (feel free to play with it); final results are directly accessed via Method object attributes
- Numpy pseudo random number generator seed is not fixed here, as the algorithms robustness may require testing. To make it fixed, simply add np.random.seed(<integer>).

4.2 HFIR LOW ENRICHED URANIUM (LEU) CORE DESIGN

This subsection provides examples for the following:

- optimizing multi-purpose nuclear reactor design
- metric function modeling: multivariate adaptive regression splines (MARS) and multidimensional extrapolation method (MDEM)
- setting up Fitter object with several constrained metric functions for multiobjective problem
- performing APSO search with multiple processors
- determining minimum and maximum possible metric values for normalization in calculation of fitness and constraint violation degree

HFIR_UMo_22/ provides scripts for solving High Flux Isotope Reactor (HFIR) LEU core design optimization, which includes a 7-dimensional optimization problem with 12 constrained objective metrics, including operation, neutronics, and thermal-hydraulics metrics. The search variables or the selected design parameters can be found in Table 2, while the metrics are provided in Table 3. The provided numbers are the configuration used in the example. Further discussion regarding HFIR LEU core design is found in the literature [8–17].

Table 2. HFIR LEU core design parameters or search variables

Design parameter	[min, max]
^{235}U loading (kg)	[18.5, 26.0]
Power rate (MW)	[95.0]
OFE mass ratio (%)	[70.755, 79.755]
IFE flatness	[0.3, 1.0]
OFE flatness	[0.34, 1.22]
IFE skewness	[-192.3, 274.0]
OFE skewness	[-363.2, 214.6]

A series of parametric studies dividing the problem into four 2-dimensional problems were conducted as part of the core optimization efforts. High-fidelity simulation tools were set up for the analysis. The results of the studies provided useful insights into how the system responds to variable changes. However, the

Table 3. HFIR LEU core design metrics

Metric	[min, max]	HEU core
^{252}Cf production rate (mg/day)	[1.46, -]	1.39
Cycle length (days)	[26.2, 34]	26.2
Cold source moderator vessel cold flux (10^{14}/cm^2-s)	[4.48, -]	4.48
Cold source moderator vessel cold flux ratio	[-, -]	0.736
Reflector fast flux (10^{14}/cm^2.s)	[2.89, -]	2.89
Reflector fast flux ratio	[0.192, -]	0.192
Flux trap fast flux (10^{15}/cm^2-s)	[1.07, -]	1.07
Flux trap fast flux ratio	[0.29, -]	0.29
Curium target thermal flux (10^{15}/cm^2-s)	[1.0, -]	1.0
Uranium utilization (days/kg)	[-, -]	2.78
Burnout margin (-)	[1.48, -]	1.61
Hotstreak equivalent gap (mils)	[35.5, -]	36.95

results were limited since there are still vast, unexplored design spaces that may yield better performance. The sensitivity of the system, along with the high dimensionality of both the search variable and the objective metric, not to mention the imposed constrains, require a different approach to further improve the design (further discussion on the previous study is provided in Betzler's work [9, 10]). MOT could be useful for improving high-end design.

It is prohibitively expensive to use MOT and have it run the HFIR LEU core high-fidelity Monte Carlo simulations over long periods of time. A rudimentary extrapolation technique and a regression model are employed here:

- multidimension extrapolation method (MDEM): a series of 2D interpolations designed to obtain approximate results of the unexplored spaces in the previous HFIR LEU core study
- multivariate adaptive regression splines (MARS) [3]: a nonparametric regression technique that can be seen as an extension of linear models and that automatically models nonlinearities and interactions between variables

Both approaches make use of the previous study data. MDEM uses the data for the interpolation references, while MARS uses the data to build the regression model. The use of MARS in nuclear reactor design was proposed by Kumar [6]. An advantage offered by MARS over MDEM is that it could be fed with new arbitrary information and thus would yield an improved model. `HFIR_UMo_22/HFIR_MARS` provides the information, data, and scripts used for generating the MARS model, while `HFIR_UMo_22/mdem_data` contains previous study data used by MDEM.

Once MOT provides a result, the high-fidelity simulation is performed for the proposed variable set for verification. If the proposed variable set does not yield the optimum performance as proposed, then the model is a poor approximation. MARS offers the opportunity to improve the model by supplying the new information and then running the MOT optimization again, possibly using an iterative scheme.

Things to note:

- Since the objective and constraint metrics are explicitly defined by MARS or MDEM model, `<Simulation>` class set up is not needed.
- The models are implemented as `<Function>` class and collected in `HFIR_UMo_22/functions.py`

- Separate scripts for running with MDEM and MARS are provided
- Hard constraints are used for the metric functions
- Maximum and minimum possible values of each metric are obtained as discussed following this list
- Parallel particles evaluation with multiple processors is performed by APSO
- Some procedures for finding the maximum and minimum possible metric values used for normalization of function and constraint violation degree are provided at the bottom of the scripts below. This is basically resetting weight of importance and constraints so that only the unconstrained maximization/minimization of each metric is considered separately.

4.3 SIMPLIFIED HFIR LEU CORE DESIGN - RING CONFIGURATION

This subsection provides examples for the following:

- setting up integer variables
- setting up <Simulation> class to perform simulation for search variable set evaluation

HFIR_UMo_22_Ring/ addresses the simplified ring configuration HFIR LEU core optimization, which is a 5-dimensional optimization problem with a constrained objective metric. This simplified version converts the HFIR LEU core into a 1D radial core. The outer and inner fuel element regions are blended into one region, comprising alternating water and fuel ring plates. The inner experimental region is assumed to be homogenized Zirc-4, and the outer experimental region is assumed to be homogenized natural beryllium. The design parameters (search variables) include the number of fuel rings (integer), fuel thickness, cladding thickness, and flatness and skewness of water channel thickness relative distribution. The metrics include k-eigenvalue (maximize with minimum constraints of 1.3), flux trap region flux (maximize), reflector region flux (maximize), and power peaking factor (minimize)

Things to note:

- A simulation, SCALE/TXSDRN, is performed for variable set evaluation. The simulation class, HFIR_TXSDRN, is collected in HFIR_UMo_22_Ring/simulations.py. Several python functions used to generate TXSDRN input from a template txsdrn.tmp are also included based on the given variable set
- <Function> classes only extract the information from TXSDRN results to obtain the metrics
- Parallel particles are evaluated with multiple processors using APSO. Note that processor IDs are passed on for generating different TXSDRN working directories to avoid any race condition

5 FUTURE DEVELOPMENT

There are several plans to further improve APSO:

- The implemented ELS hinders APSO parallel performance: ELS requires an additional evaluation after parallel evaluation of the particle population is performed. This ELS procedure needs to be improved or replaced. ELS adds a mutation procedure into APSO, which helps to avoid premature convergence to a local optima. An idea is to introduce decaying mutation (by a normal perturbation) to the velocity parameters (w, c_1, c_2).
- Parallel performance of APSO can be improved further: currently, job distribution is performed at each iteration. This could be improved, especially for large numbers of particles, by calculating each particle's mean distance to all the other particles d in calculation of evolutionary factor f performed at each iteration.
- A variable is clamped if it exceeds the limit. This may yield a problem: if the best particle reaches that limit, then it attracts all the other particle to hit and fix their positions at that limit, as well, stopping any further search on that variable. This may not be a problem if the optimum solution lies in that variable limit, but otherwise, the only feature that could remedy this issue is the ELS. One idea is to reflect the particle if it reaches a variable limit. This would not only avoid the fixed variable issue, but it would also add more mutation element into APSO.
- Another way to add more mutation element and avoid premature convergence to local optima is to introduce population topology: instead of finding and following the best particle of the whole population, each particle could only find and follow the best particle among the nearest N particles. This topology size N could then be grown linearly from 1 to the total number of particle population along the iteration.

Another area of research for MOT development is to continue working on implementing multiobjective adaptive particle swarm optimization (MOAPSO). MOAPSO uses the Pareto nondominance sorting approach to treat the multiobjectivity of the problem. Without employing any aggregation approach, MOAPSO yields several Pareto solutions given a sufficient number of iterations/generations [7].

6 REFERENCES

1. J. Kennedy and R. Eberhart, “Particle Swarm Optimization,” *Proceedings of IEEE International Conference on Neural Networks*, IV. pp. 1942–1948 (1995).

2. Z-H. Zhan, J. Zhang, Y. Li, and H.S-H. Chung, “Adaptive particle swarm optimization,” *IEEE Transactions on Systems Man, and Cybernetics - Part B: Cybernetics*, 39 (6), pp. 1362–1381. ISSN 0018-9472 (2009).

3. J. H. Friedman, “Multivariate Adaptive Regression Splines,” *The Annals of Statistics*, 19: 1 (1991).

4. W. Jakob and C. Blume, “Pareto Optimization or Cascaded Weighted Sum: A Comparison of Concepts,” ISSN 1999–4893, www.mdpi.com/journal/algorithms (2014).

5. Deb, Kalyanmoy, et al. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II." *IEEE Transactions on Evolutionary Computation 6.2* 182–197 (2002).

6. A. Kumar and P. V. Tsvetkov, “A New Approach to Nuclear Reactor Design Optimization Using Genetic Algorithms and Regression Analysis,” *Annals of Nuclear Energy*, 85: 27–35 (2015).

7. CAC Coello, ”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* 191: 1245–1287, (2002).

8. B. R. Betzler, D. Chandler, E. E. Davidson (née Sunny), and G. Ilas, “High Fidelity Modeling and Simulation for a High Flux Isotope Reactor Low-Enriched Uranium Core Design,” *Nuclear Science and Engineering*, 187(1), pp. 81–99 (2017).

9. B. R. Betzler, D. Chandler, D. H. Cook, E. E. Davidson (née Sunny), and G. Ilas,“High Flux Isotope Reactor Low-Enriched Uranium Core Design Optimization Studies,” *PHYSOR 2018 - Reactor Physics: Paving the Way Towards More Efficient Systems*, Cancun, Mexico, Apr. 22–26 (2018).

10. B. R. Betzler, D. Chandler, D. H. Cook, E. E. Davidson, and G. Ilas,“Design Optimization Methods for High-Performance Research Reactor Core Design,” *Nuclear Engineering and Design*, 352 (2019).

11. D. Chandler, B. R. Betzler, D. H. Cook, G. Ilas, and D. G. Renfro, “Neutronic and Thermal-Hydraulic Feasibility Studies for High Flux Isotope Reactor Conversion to Low-Enriched Uranium U_3Si_2-Al Fuel,” *PHYSOR 2018 - Reactor Physics Paving the Way Towards More Efficient Systems*, Cancun, Mexico, Apr. 22–26 (2018).

12. G. Ilas, B. R. Betzler, D. Chandler, and E. E. Sunny, “High Flux Isotope Reactor Core Analysis - Challenges and Recent Enhancements in Modeling and Simulation,” *Proc. PHYSOR 2016 - Unifying Theory and Experiments in the 21st Century*, Sun Valley, ID, USA, May 1–5 (2016).

13. B. R. Betzler, B. J. Ade, D. Chandler, G. Ilas, and E. E. Sunny, “Optimization of Depletion Modeling and Simulation for the High Flux Isotope Reactor,” *Proc. ANS Mathematics & Computation Topical Meeting*, Nashville, TN, USA, Apr. 19–23 (2015).

14. G. Ilas, B. R. Betzler, D. Chandler, E. E. Davidson (née Sunny), and D. G. Renfro, *Key Metrics for HFIR HEU and LEU Models*, Oak Ridge National Laboratory Report ORNL/TM-2016/581, Oct. (2016).

15. D. Chandler, B. R. Betzler, G. Hertz, G. Ilas, and E. E. Sunny, *Modeling and Depletion Simulations for a High Flux Isotope Reactor Cycle with a Representative Experiment Loading*, Oak Ridge National Laboratory Report ORNL/TM-2016/23, Sep. (2016).

16. G. Ilas, D. Chandler, B. J. Ade, E. E. Sunny, B. R. Betzler, and D. L. Pinkston, *Modeling and Simulations for the High Flux Isotope Reactor Cycle 400*, Oak Ridge National Laboratory Report ORNL/TM-2015/36, Mar. (2015).

17. G. Ilas and B. Patton, *Core Neutronics Reference Analysis for HFIR Zr-Clad U-Mo LEU Fuel*, Oak Ridge National Laboratory, ORNL/HFIRUP-2011/003, Sep. (2011).