Advances in Engineering Software 166 (2022) 103092

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

VecMetaPy: A vectorized framework for metaheuristic optimization in Python

AmirPouya Hemmasian [a], Kazem Meidani [a], Seyedali Mirjalili [b,c], Amir Barati Farimani [*,a,d,e,*]

[a] Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
[b] Centre for Artificial Intelligence Research and Optimization, Torrens University Australia, Australia
[c] Yonsei Frontier Lab, Yonsei University, Seoul, Republic of Korea
[d] Machine Learning Department, Carnegie Mellon University, Pittsburgh, PA, USA
[e] Department of Biomedical Engineering, Carnegie Mellon University, Pittsburgh, PA, USA

ARTICLE INFO

Keywords:
Vectorization
Metaheuristic
Optimization
Swarm intelligence
Algorithm

ABSTRACT

This work aims to accelerate metaheuristic optimization algorithms and their experimental environment using the vectorization feature in the NumPy library in Python. It is built upon EvoloPy, a framework previously introduced by Faris et al. (2016)[41] for metaheuristic optimization in Python. We have vectorized every aspect of this framework including function evaluation, random number sampling, strategy selection, and performing the update equations. We compare the wall-clock time of our vectorized framework for a few algorithms with the original implementation in EvoloPy. The results demonstrate the substantial improvement in the algorithm's execution time, even to three orders of magnitude in cases like a 180-dimensional search space. The codes can be found at https://github.com/BaratiLab/VecMetaPy.

1. Introduction

During the last decade, metaheuristic optimization algorithms have drawn the attention of many researchers and engineers in a variety of applications [1–7]. We can say that the Genetic Algorithm (GA) [8], Particle Swarm Optimization (PSO), [9] and Differential Evolution [10] were among the first algorithms to gain such popularity and inspire many following works in the last two decades. There are three main reasons behind the growing popularity of these algorithms. First, they are based on simple concepts from natural ecosystems and usually have straightforward implementations [11–17]. This makes them easy to modify and refashion for specific applications [18–20]. The second reason is their black-box property, meaning that the only information that these algorithms need about the optimization task is the output of the objective functions for a specific point in the search space [21,22]. Unlike gradient-based algorithms, they do not require information about the gradient or the Hessian of the objective function, which is sometimes very hard or impossible to obtain. The third and arguably the most important feature of metaheuristics is their randomness which makes them capable of escaping local optima stagnation [23,24]. Every metaheuristic optimization has its specific stochastic operators and update rules to explore the search space thoroughly and not to get stuck in one neighborhood. Despite these advantages, there is a drawback regarding the computational time for these algorithms when dealing with large dimensional search spaces and a large number of search agents, especially for Swarm Intelligence (SI) algorithms. The inspiration of these algorithms is the cooperative behavior of predator packs in nature and their hunting and foraging strategies. Ant Colony Optimization (ACO) [25], Artificial Bee Colony [26], Particle Swarm Optimization (PSO) [27], Cuckoo Search (CS) [28], and Bacterial Foraging Optimization (BFO) [29] are among the first popular examples of such algorithms. In the last few years, several novel SI algorithms have been developed which are still the center of attention for many researchers. Some well-known examples of these algorithms are Krill Herd [30],Grey Wolf Optimizer (GWO) [12], Whale Optimization Algorithm (WOA) [15], Moth-flame Optimization (MFO) [13], Dragonfly Algorithm (DA) [31], Grasshopper Optimization Algorithm (GOA) [32], and the Ant Lion Optimizer (ALO) [33]. In this work, we greatly reduce the execution time of such algorithms by optimizing their implementation using the concept of vectorization.

Vectorization is one of the most important features of almost every programming framework that deals with matrices and tensors. In fact,

* Corresponding author at: Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA, USA.
E-mail addresses: ahemmasi@andrew.cmu.edu (A. Hemmasian), mmeidani@andrew.cmu.edu (K. Meidani), ali.mirjalili@gmail.com (S. Mirjalili), barati@cmu.edu (A. Barati Farimani).

https://doi.org/10.1016/j.advengsoft.2022.103092
Received 9 November 2021; Received in revised form 4 January 2022; Accepted 22 January 2022
Available online 11 February 2022

vectorization is a methodology to make calculations with vectors, matrices, and tensors much faster by efficient allocation of the memory and performing many element-wise calculations simultaneously. In many cases for iterative algorithms like metaheuristics, the elements of a matrix need to be updated based on their value from the previous iterations, without any explicit dependence on the other elements of the matrix. Therefore, it is possible to perform the calculation of the elements in a parallel approach and not serial using `for` loops. This has the potential to substantially reduce the run time of metaheuristics as one of their main drawbacks, which is due to the use of a population of solutions and the need to calculate the objective values for each solution.

The original implementation of the metaheuristic algorithms usually has a serial approach. The position of the search agents is generally stored in an $n \times d$ matrix, where n is the number of search agents and d is the number of the dimensions in the search space. The elements of this matrix are updated one by one using `for` loops, based on their value in the previous iteration and some randomly sampled numbers. In this work, all aspects of the optimization are vectorized, which means all elements of a matrix are updated simultaneously, without the use of `for` loops for different search agents or dimensions. This implementation hugely decreases the execution time of these algorithms, especially in high-dimensional search spaces. To this end, we have leveraged upon the NumPy library [34,35] in Python and its powerful built-in functions and array operations. NumPy has been one of the most useful and popular array programming frameworks for scientific computations in the Python language.

This paper is structured as follows: Section 2 gives a brief overview of the literature and similar works from researchers trying to make metaheuristic algorithms faster and more efficient. Section 3 provides a general framework in order to make the vectorization process more understandable to the reader. The main methodology of vectorization is discussed in Section 4 along with a brief overview of the algorithms of interest in this paper. Section 5 provides the experimental results and discussions. Finally, the conclusion and future work is discussed in Section 6.

2. Related works

The majority of the novel metaheuristic methods aim to introduce better optimization capabilities that have outstanding properties of fast convergence rate and finding the global minima. In addition to these algorithmic improvements, the performance of the optimization processes can also be enhanced in terms of implementation and hardware utilization. While this kind of improvement is less studied in the metaheuristic field, it is of very high importance. In fact, since a major metric of the optimization algorithms is the convergence rate, the computation time is a critical aspect of every metaheuristic especially when solving computationally expensive problems. There have been some previous works that have tried to find methods for efficient implementation of the optimization methods for faster computations, which will be briefly discussed in the following paragraphs.

A group of proposed solutions for a more efficient implementation of the optimization algorithms are based on parallel computing in Crainic [36]. Tasks and calculations at different stages of the optimization can be performed simultaneously using parallel techniques which would reduce the amount of time required for the process compared to the serial calculations. Parallel metaheuristic and distributed methods have been specifically studied for evolutionary algorithms [37,38]. Computational resources and memory usage can be also exploited efficiently to perform a faster optimization [39,40]. However, these works focus on utilizing several processing cores simultaneously and efficiently. In our work, we are going to get the most out of a single processing unit through vectorization.

This work is built upon Evolopy [41], an open-source Python framework for metaheuristic optimization developed by Faris et al.. Although the information about the positions of the search agents is stored using NumPy arrays, the full capacity and potential of the features of NumPy have not been utilized in Evolopy. In this work, we have implemented every aspect of our framework in a much more efficient way, taking advantage of the built-in functions of NumPy whenever possible, and with minimum use of explicit loops in our implementation.

3. General framework

In this section, we formulate a general framework mathematically in order to have clear definitions of the steps taken in each iteration of a metaheuristic algorithm. In all kinds of multi-agent algorithms like swarm intelligence algorithms, we are optimizing an objective function in a d dimensional search space. We initialize a population with n points (search agents) randomly in the search space and update them according to the positions of some elite agents and some randomly sampled numbers. We denote the positions matrix as X shown in Eq. 1 which stores the locations of the search agents. Each row represents a search agent, and each column represents one dimension of the search space. So the element $X[i,j]$ is the value of the jth dimension of the ith search agent.

$$X_{n\times d} = \begin{bmatrix} X[1,1] & X[1,2] & . & . & . & X[1,d] \\ X[2,1] & X[2,2] & . & . & . & X[2,d] \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ X[n,1] & X[n,2] & . & . & . & X[n,d] \end{bmatrix} \tag{1}$$

We use the notation $X[i,j]$ to represent individual elements of this matrix from now on. We will refer to the ith row of X as $X[i,:]$, and the jth column as $X[:,j]$. Simply put, the : symbol indicates the whole set, so if we use : instead of the index of the column, we mean all columns.

Next, the function evaluation step comes into the mix. This step simply applies the objective function to each search agent. We denote the objective function as f shown in Eq. (2). It takes the position vector of a search agent, i.e. one row of the matrix X, as the input and returns a real number which is the fitness value of that search agent.

$$f : \mathbb{R}^d \rightarrow \mathbb{R} \tag{2}$$

We store the fitness values of the search agents in the vector F as shown in Eq. (3).

$$F_{n\times 1} = [f(X[1,:]) \quad f(X[2,:]) \quad . \quad . \quad . \quad f(X[n,:])]^T \tag{3}$$

Next, some of the search agents should be selected as the leaders or elites in order to guide the search toward their location. The number of leaders usually ranges from one, i.e. only the best search agent, to n, i.e. the same number as the original population. The leader matrix X^* is shown in Eq. (4). We denote the number of elite agents with n^*.

$$X^*_{n^*\times d} = \begin{bmatrix} X^*[1,1] & X^*[1,2] & . & . & . & X^*[1,d] \\ X^*[2,1] & X^*[2,2] & . & . & . & X^*[2,d] \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ X^*[n^*,1] & X^*[n^*,2] & . & . & . & X^*[n^*,d] \end{bmatrix} \tag{4}$$

and we store their fitness values in the vector F^* as shown in Eq. (5).

$$F^*_{n^*\times 1} = [f(X^*[1,:]) \quad f(X^*[2,:]) \quad . \quad . \quad . \quad f(X^*[n^*,:])]^T \tag{5}$$

In every iteration, X^* and F^* are updated with the operator *UpdateElites* as shown in Eq. (6).

$$X^*_{t+1}, F^*_{t+1} = UpdateElites\left(X^*_t, F^*_t, X_{t+1}, F_{t+1}\right) \tag{6}$$

UpdateElites often concatenates the vector $F(t+1)$ and $F^*(t)$ and selects the top n^* indexes as the best agents in the concatenated vector. Then it outputs the corresponding agents and their fitness values in the new X^*

and F^* respectively. In most cases, X^* and F^* store the location and the fitness values of the best n^* locations in the search space observed so far from the beginning of the optimization process up until the current iteration.

Finally, we formulate the main step, which is updating the location of the search agents in the population. For the update equations, there are usually some parameters and random numbers involved. Also, algorithms may have different update equations associated with different movement strategies for exploration and exploitation. The common case is that the parameters are a function of the iteration number t. We define the parameters as the vector $\vec{a}$ and all the random numbers needed as the vectors $\vec{r}_1$ and $\vec{r}_2$. $\vec{a}$ is the same for all elements of X, but $\vec{r}_1$ is sampled for each search agent separately and mainly used to decide which update equation is going to be selected for its corresponding search agent, and the random vector $\vec{r}_2$ is used in the update equations. $\vec{r}_1$ might also be used in the update formula in some situations and algorithms.

Suppose there are n_s different strategies, each with its own update equation. Each strategy has its own detector operator, which decides if the current search agent is supposed to be updated with that update equation or not. We denote this operator with $UseStrategy_s$ for the strategy number s as shown in Eq. (7). Note that for each search agent, only one of the strategies can be selected, meaning that for only one value of s, $UseStrategy_s$ can return *True* and it returns *False* for all other strategies.

$$UseStrategy_s\left(\vec{a}, \vec{r}_1\right) \in \{True, False\}, s = 1, 2, \ldots, n_s \tag{7}$$

Each Strategy has its own update equation, and that update equation is applied to each element of the matrix one by one. The general form of this update is shown in Eq. (8).

$$X_{t+1}[i,j] = Update_s\left(X_t[i,j], \vec{a}, \vec{r}_1, \vec{r}_2, X_t^*, X_t\right) \tag{8}$$

This equation simply means that the new value of each element of X is calculated based on the locations of the elite agents, the parameters and random numbers, and obviously, it inherits values from the previous iteration. In some scenarios, the exploration is based on randomly selected agents among the population, and that is why the matrix X itself is one of the arguments for $Update_s$.

Every metaheuristic algorithm has the general scheme explained above, which is also shown in Algorithm 1. This is usually the vanilla implementation proposed by the authors of the algorithms in the main papers.

4. Vectorized implementation

We know that the general setting of most metaheuristic algorithms is that the same operators are applied to many search agents and their dimensions, using `for` loops that iterate through them one by one. This approach in implementing the algorithm is not efficient in utilizing the computational resources and hardware as many independent calculations are being conducted one by one. However, these can be performed simultaneously because they are independent of each other. In this section, we go through each aspect of these algorithms and explain the process of vectorization for each step.

4.1. Function evaluation

As explained in the preceding section in Eq. (2), the objective function takes one point in the search space and returns a real number as its fitness value. In a vectorized implementation, we define our objective functions in a way that they apply f on every row of the matrix X at the same time, and without using a `for` loop to go over the search agents one by one. With this, the vectorized objective function f_v is as follows:

$$f_v : \mathbb{R}^{n\times d} \rightarrow \mathbb{R}^n \tag{9a}$$

$$F = f_v(X) \tag{9b}$$

This function applies the same calculations to every row of the matrix X at the same time and returns a vector of length n containing the fitness values of each row. In the vanilla implementation, there are operations like summation and product over the dimensions that can be easily implemented in Python using NumPy's built-in functions like `numpy.sum()`, `numpy.prod()`, `numpy.mean()`, etc. We can use the exact same NumPy functions to apply them to whole matrices, with providing the argument `axis=1` to the function. This makes the function be applied only along the specified axis of the matrix, which is the dimension axis. For example, `numpy.sum(x**2)` is the implementation of the benchmark function F1 in Python, in which `x` is a vector of length d. We simply implement the vectorized version of this function as `numpy.sum(X**2, axis=1)` where `X` is the whole $n \times d$ positions matrix. The argument `axis=1` tells the function to take the summation along the dimension axis, meaning that it is performed for each row independently and simultaneously. This implementation reduces the wall-clock time that it takes to obtain the F by a huge factor, especially when we are dealing with very high dimensional objective functions that also requires a large number of search agents.

In other objective functions, there might not be straightforward operations like summation that can be vectorized by providing the `axis=1` argument. Suppose that a function f (not vectorized) consists of some calculations of the elements of a vector, which are the dimensions

```
Initialize X_0 randomly, obtain F_0
obtain X*_0 and F*_0 by sorting the initial population
for t = 0 : T − 1 do
    update a⃗
    sample R_1
    for s = 1 : n_s do
        idx_s = UseStrategy_s(a⃗, R1)
    for s = 1 : n_s do
        sample R_2
        X_{t+1}[idx_s] = Update_s(X_t[idx_s], a⃗, R_1[idx_s], R_2[idx_s], ...)
    F_{t+1} = f_v(X_{t+1})
    obtain X*_{t+1} and F*_{t+1} using eq. 6
return X*_T[1, :] and F*_T[1]
```

Algorithm 1. General scheme of a population-based metaheuristic algorithm.

of a search agent for our case. We can vectorize this function by replacing the elements of the vector with the columns of a matrix in the code. For example, if the search space is two dimensional, and the objective function is $x_1 x_2^2$, the function f is defined as `x[0]*x[1]**2` and takes one row of the positions matrix (a vector) as input. The vectorized version or f_v for this function is implemented as `X[:,0]*X[:,1]**2` and takes the whole positions matrix X and performs the same operations for all elements of each column which are the values of the same dimension for each search agent. The output of this function will be a vector of length n which is the fitness vector F.

In this work, we have fully vectorized the 23 popular benchmark objective functions commonly used to evaluate metaheuristic algorithms in the literature [12,42]. The detailed definition of these functions is provided in appendix A.

4.2. Random sampling and strategy selection

In the original implementation, the random numbers needed for updating each dimension of each search agent are sampled one at a time in the corresponding step of the `for` loops for that particular element. With the vectorized implementation, a vector or a matrix of random numbers is sampled from the same distribution, meaning that each element of that matrix is sampled from the same distribution. We call these random vectors or matrices R_1 and R_2 which have the same role in the algorithm as $\vec{r}_1$ and $\vec{r}_2$ respectively. They will be used in the vectorized implementation of strategy selection and update equations, and will lead to the same exact results but in a much shorter time.

In order to vectorize the strategy selection step of the algorithm, we apply the same $UseStrategy_i$ operation from Eq. (7) but on the whole vectors or matrices R_1 and R_2. The output will be a Boolean vector or matrix of the same size, meaning each element is either `True` or `False`. Now, instead of several `if` statements for each `i, j`, we will use the matrix output of *UseStrategy* operation as the index of the other matrices in the calculations in order to apply the update equations on the elements of X which have a *True* value in their corresponding element of the *UseStrategy* output, as shown in the following subsection.

4.3. Update equations

This step of the algorithms is vectorized by simply replacing the individual elements in the update equations with whole matrices or slices of whole matrices defined by the output of *UseStrategy*. As such, there will be no `for` loops needed in order to update each element of the matrix X one after another, because all of them will be updated with the same equation or a finite set of equations. The general schematic of the vectorized implementation of a metaheuristic algorithm is shown in Algorithm 2. Note that we have stored the Boolean matrices idx_s for each strategy and used them as indexes for the matrices involved in the update equations.

```
Initialize X_0 randomly, obtain F_0
obtain X*_0 and F*_0 by sorting the initial population
for t = 0 : T − 1 do
    update a⃗
    for i = 1 : n do
        sample r⃗_1
        for j = 1 : d do
            if UseStrategy_1 then
                sample r⃗_2
                calculate X_{t+1}[i, j] using eq. 8 with s = 1
            else if UseStrategy_2 then
                sample r⃗_2
                calculate X_{t+1}[i, j] using eq. 8 with s = 2
            .
            .
            .
            else if UseStrategy_{n_s} then
                sample r⃗_2
                calculate X_{t+1}[i, j] using eq. 8 with s = n_s
    for i = 1 : n do
        F_{t+1}[i] = f(X_{t+1}[i, :])
    obtain X*_{t+1} and F*_{t+1} using eq. 6
return X*_T[1, :] and F*_T[1]
```

Algorithm 2. General scheme of the vectorized implementation of a population-based metaheuristic algorithm.

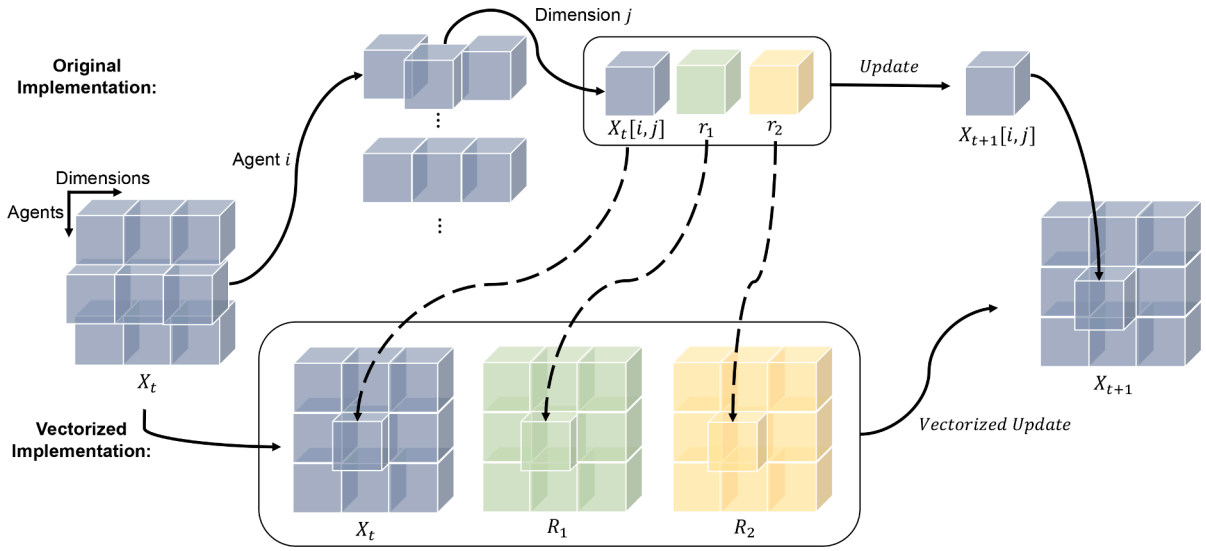

Fig. 1. A simple illustration of the mechanism of vectorization compared to the original implementation of the update equation. This is a simple case where all agents follow the same update equations, but the general concepts can be applied to subsets of the mentioned matrices.

In Fig. 1, we have provided a simple illustration of how vectorization works. In the vectorized implementation, each element in the output matrix X_{t+1} is calculated based on the corresponding element of the input matrices of the *Update* operator. The final output matrix is exactly the same as the one computed with the original implementation and for loops, with the only difference of performing the updates simultaneously and efficiently.

4.4. Vectorized algorithms

In this work, we have vectorized the implementation of five recent and popular metaheuristic optimization algorithms: Grey Wolf Optimization (GWO) [12], Whale Optimization Algorithm (WOA) [15], Moth Flame Optimizer (MFO) [13], Sine Cosine Algorithm (SCA) [43], and Salp Swarm Algorithm (SSA) [16]. The details of these algorithms are not of interest in this work as we focus on the process of developing their vectorized versions, so we refer the readers to the papers cited. We will give a very brief overview of each algorithm and how it matches the framework defined in the previous subsection. If an algorithm has a specific step not included in the aforementioned template, more explanation will be provided.

If not the most popular, Grey Wolf Optimizer (GWO) has been one of the most popular metaheuristics introduced in the last decade. In this algorithm, there are three elite search agents named alpha, beta, and delta which are inspired by the hierarchy of the grey wolf packs. So in the standard framework, we have $n^* = 3$ for this algorithm. In GWO, all search agents are updated with the same update equation which makes them search around the estimated position of the prey, which is calculated based on the position of alpha, beta, and delta wolves. Therefore, the updating step can be implemented in a single line of code using `numpy` functions and operators. Since all agents are updated with one update equation, i.e. one movement strategy, there is no strategy selection operator, so this algorithm has one of the simplest architectures among the recent metaheuristic algorithms.

Whale Optimization Algorithm (WOA) is another popular metaheuristic algorithm inspired by the bubble-net hunting method of humpback whales. This algorithm has three different update equations for three movement strategies of whales: encircling another random whale, encircling the top whale, and performing the bubble-net method around the top whale. The overall structure of this algorithm and its vectorized version matches the explained framework in this section as well.

Sine Cosine Algorithm (SCA) has a very simple update equation as well and can be described using the standard framework perfectly. This algorithm has only one elite search agent, and the other agents search around it according to two simple update equations, one including a sine function and another with a cosine.

As suggested by its name, Moth-Flame Optimization Algorithm (MFO) is inspired by the spiral movement of moths around small lights when navigating at night. In this algorithm, there are n moths and $n^* = n$ flames which are the top n locations visited by the moths in the search space. Each moth is assigned to one flame index during the optimization process over the course of iterations, meaning that moth number i moves around flame number i in all iterations, until that flame dies out. For the sake of convergence, the flames are turned off one by one as the optimization goes on, so that all the moths exploit the neighborhood of the best solution in the final iteration. Each time a flame dies out, the moths flying around it will change their target flame to the last active flame. Therefore, there are two different update equations, one for the moths with their flame active, and another one for the moths which are not moving around their specific flame and instead of searching around the last active flame. The vectorization process for this algorithm is also the same as the previous ones, fully matching the standard framework.

The last algorithm that is vectorized in this work is the Salp Swarm algorithm (SSA). Unlike the previous algorithms, SSA has a distinct step different from the previously discussed algorithms, and that is the chain of salps. In this algorithm, the best solution is stored as the position of the food, and the other salps are updated according to one of two stra-

Table 1
The average execution time (in seconds) for a 100 iteration run of the algorithms for different benchmark functions, with $n = 180$ ($d = 180$ for F1–F13).

	GWO		WOA		MFO		SSA		SCA	
	original	vectorized	original	vectorized	original	vectorized	original	vectorized	original	vectorized
F1	38.1857	0.1593	31.1726	0.0293	33.3267	0.1608	29.9698	0.1964	32.7037	0.3061
F2	38.3905	0.1641	31.9027	0.0342	33.1297	0.1649	30.2930	0.2195	33.3058	0.3081
F3	46.6328	0.1661	40.7062	0.0370	40.8243	0.1689	38.8379	0.1989	42.9711	0.3127
F4	37.5439	0.1589	31.1526	0.0296	31.8156	0.1608	29.9480	0.1720	33.5649	0.3060
F5	38.0623	0.1664	31.0726	0.0395	32.3171	0.1707	29.9626	0.2292	33.3311	0.3154
F6	38.0467	0.1555	30.9250	0.0304	31.3223	0.1616	29.1280	0.1797	33.9834	0.3081
F7	38.4473	0.3278	31.2330	0.1908	31.7463	0.3305	29.2611	0.6724	33.7172	0.4769
F8	38.0577	0.2560	30.9886	0.0973	32.9042	0.2439	29.2177	0.4499	33.8236	0.3890
F9	37.8595	0.2378	31.1184	0.0670	31.3042	0.2508	28.6467	0.3701	32.2853	0.3789
F10	38.1960	0.2326	31.5084	0.0700	31.4862	0.2438	29.3979	0.4656	34.0958	0.3931
F11	38.6069	0.2176	31.5759	0.0709	32.8178	0.2591	29.9498	0.4483	34.2235	0.3977
F12	38.8496	0.6142	32.5861	0.4488	32.5259	0.5994	30.3496	1.3630	34.5541	0.7438
F13	38.8517	0.6071	32.2592	0.4723	32.5489	0.6466	30.4602	1.4576	34.7794	0.8803
F14	6.3610	0.0589	6.6653	0.0624	6.3039	0.0572	6.3732	0.0605	6.6353	0.0576
F15	1.1931	0.0135	1.1098	0.0163	1.1006	0.0132	1.0585	0.0092	1.1160	0.0145
F16	0.4886	0.0123	0.4251	0.0163	0.4668	0.0109	0.4388	0.0094	0.4450	0.0115
F17	0.5092	0.0095	0.4418	0.0136	0.4616	0.0082	0.4587	0.0065	0.4624	0.0086
F18	0.5256	0.0109	0.4587	0.0149	0.5005	0.0099	0.4900	0.0079	0.4812	0.0101
F19	1.2996	0.0138	1.2141	0.0174	1.2260	0.0131	1.2384	0.0104	1.2464	0.0143
F20	1.9469	0.0164	1.7560	0.0184	1.7724	0.0166	1.7605	0.0117	1.8520	0.0198
F21	3.5636	0.0139	3.6113	0.0170	3.3914	0.0136	3.5679	0.0102	3.5936	0.0154
F22	4.4080	0.0149	4.6252	0.0181	4.3294	0.0146	4.5121	0.0111	4.5568	0.0164
F23	5.7404	0.0164	5.7778	0.0196	5.7698	0.0163	5.7852	0.0127	5.8617	0.0180

tegies: half of the population move randomly around the food position, and the other half form a chain following the last salp from the first half. This strategy is inspired by the natural movement of salps in the oceans. The chain is modeled mathematically by the following update equation:

$$x_i = \frac{1}{2}(x_{i-1} + x_i) \tag{10}$$

where x_i is salp number i, or the ith row of the matrix X, and i ranges from $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n. This chain can be represented using a matrix multiplication in the vectorized version. If we break down this chain of equations, we will have:

$$x_i = \frac{1}{2}(x_{i-1} + x_i) \tag{11a}$$

$$x_{i+1} = \frac{1}{2}(x_i + x_{i+1}) = \frac{1}{2}\left(\frac{1}{2}(x_{i-1} + x_i) + x_{i+1}\right) = \frac{1}{4}x_{i-1} + \frac{1}{4}x_i + \frac{1}{2}x_{i+1} \tag{11b}$$

$$x_{i+2} = \frac{1}{8}x_{i-1} + \frac{1}{8}x_i + \frac{1}{4}x_{i+1} + \frac{1}{2}x_{i+2} \tag{11c}$$

There is an obvious pattern in the coefficients because this update equation is applied to the salps in order. We represent the slice of the matrix X which is going to be updated with the chaining strategy as X_{chain}. Then the chain movement can be represented by the following matrix multiplication in Eq. (12), in which $s = n - \left\lfloor \frac{n}{2} \right\rfloor$ is the number of agents updated according to the chain behavior, X_{chain} corresponds to the last s rows of the matrix X, and $x_{\left\lfloor \frac{n}{2} \right\rfloor}$ is the last salp moving randomly around the food position.

$$X_{chain} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & . & . & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} & 0 & 0 & . & . & 0 \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & 0 & . & . & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ \frac{1}{2^{s+1}} & \frac{1}{2^{s+1}} & . & . & . & \frac{1}{8} & \frac{1}{4} & \frac{1}{2} \end{bmatrix}_{s \times (s+1)} \times \begin{bmatrix} x_{\left\lfloor \frac{n}{2} \right\rfloor} \\ X_{chain} \end{bmatrix} \tag{12}$$

Although the vectorization process of most algorithms follows the pattern discussed in this section, there are algorithms with special operators like SSA that deviate from the general pattern. Depending on the computational complexity, number of different strategies and operators, and other features of an algorithm, the amount of improvement in execution time caused by vectorization varies for each one. Therefore, we investigate the results for the five algorithms above in the next section quantitatively and compare the execution time of the two implementations in order to show the power of vectorization and that it varies depending on the algorithm and the objective function being optimized.

5. Experiments and discussion

In order to quantitatively evaluate the improvement caused by vectorizing the benchmark objective functions and the mentioned metaheuristic algorithms, we used both the original implementation and the vectorized implementation of each algorithm and optimized every benchmark function with them, recording the wall-clock time of their

Table 2
The speed-up ratio for different algorithms and benchmark functions for two different values of $n = 30, 180$ (and $d = n$ for F1–F13).

	GWO		WOA		MFO		SSA		SCA	
	$n = 30$	$n = 180$	$n = 30$	$n = 180$	$n = 30$	$n = 180$	$n = 30$	$n = 180$	$n = 30$	$n = 180$
F1	112.43	239.76	84.26	1064.28	100.50	207.31	158.98	152.58	76.83	106.84
F2	103.52	233.91	76.24	932.27	95.31	200.88	146.34	138.04	72.51	108.10
F3	127.25	280.73	93.93	1100.85	116.10	241.72	149.28	195.26	88.77	137.43
F4	113.71	236.27	75.62	1052.20	99.66	197.85	161.97	174.07	74.17	109.68
F5	102.67	228.76	72.85	787.44	87.29	189.37	121.96	130.70	68.12	105.68
F6	114.27	244.71	78.45	1017.40	100.24	193.86	147.95	162.05	74.58	110.30
F7	74.92	117.30	56.98	163.73	62.64	96.06	82.93	43.52	53.05	70.70
F8	79.17	148.66	67.45	318.46	75.75	134.93	93.07	64.94	59.89	86.95
F9	85.85	159.18	69.86	464.51	73.09	124.81	99.84	77.39	58.15	85.21
F10	81.42	164.21	58.70	450.42	64.03	129.15	73.04	63.14	52.52	86.73
F11	87.64	177.39	64.31	445.31	70.32	126.67	89.95	66.81	57.34	86.06
F12	43.82	63.25	35.27	72.61	38.65	54.27	41.77	22.27	35.37	46.46
F13	45.86	63.99	35.42	68.30	38.06	50.34	41.08	20.90	32.36	39.51
F14	74.43	107.96	59.42	106.86	72.44	110.26	78.85	105.31	80.85	115.21
F15	25.97	88.38	15.63	68.00	28.12	83.54	26.71	114.64	26.99	76.76
F16	11.18	39.75	6.45	26.03	13.55	42.65	12.47	46.87	12.69	38.79
F17	13.00	53.39	6.86	32.43	16.54	56.41	15.22	70.07	15.36	53.53
F18	10.85	48.08	6.73	30.70	13.80	50.71	12.82	61.84	12.35	47.73
F19	27.89	94.03	16.80	69.97	32.83	93.34	32.65	118.54	31.78	87.41
F20	39.18	118.78	23.82	95.48	38.29	106.97	45.90	150.62	39.97	93.60
F21	81.21	256.08	52.08	211.99	89.73	248.60	101.60	351.34	90.08	232.90
F22	103.61	296.54	66.17	255.80	110.77	296.09	124.08	407.13	112.14	277.72
F23	134.15	349.18	81.79	295.10	141.87	354.67	159.73	456.72	142.58	326.15

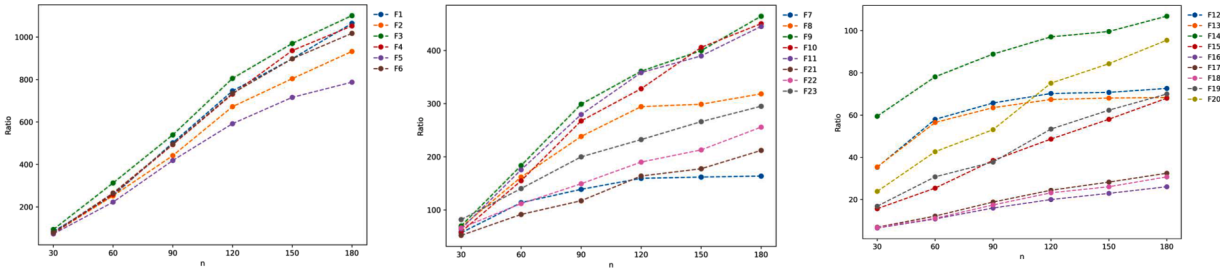

Fig. 2. The speed-up ratio for different values of n ranging from 30 to 180 for WOA. The speed-up ratio depends on the search space dimension and the mathematical complexity of the function definition.

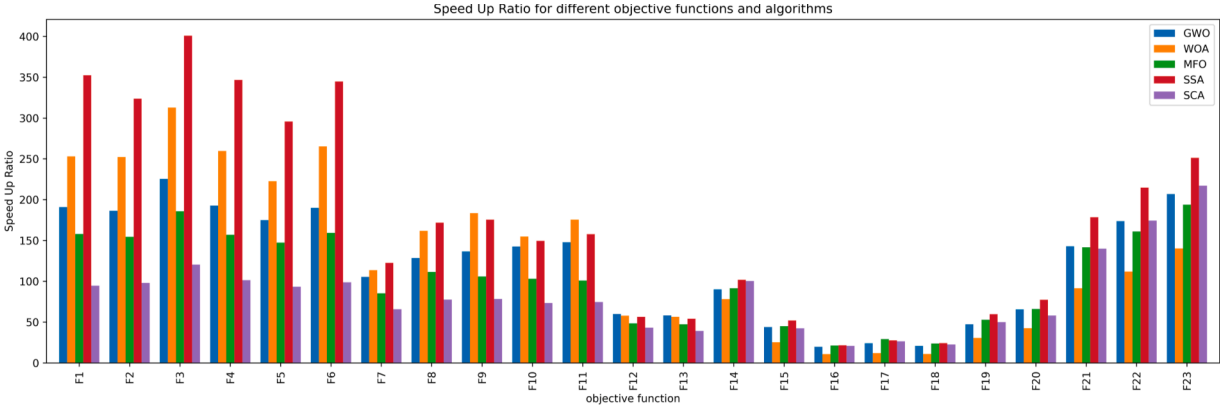

Fig. 3. The speed-up ratio for the 23 benchmark functions and the vectorized algorithms with $n = 60$. Similar to Fig. 2, F1–F6 have a bigger speed-up ratio compared to the other functions. The low dimensional functions (F14–F20) show a smaller speed-up ratio because of the low dimension of their search space. F21, F22 and F23 differ from other low dimensional functions because they have a higher computational complexity and number of mathematical operators.

execution. Since the execution time does not show much variation, we performed the experiments for 10 independent runs and reported the average time. We ran each algorithm for 100 iterations and tried several values for the number of search agents and the number of dimensions of the search space, ranging from 30 to 180. For the high dimensional functions, i.e. F1 through F13, the number of dimensions and the number of search agents are set to be the same. The processor used in our experiment is Intel(R) Core (TM) i9-9900 K CPU @ 3.6 GHz.

Some of the results for the values of $n = 30$ and $n = 180$ have been provided in Tables 1 and 2. By looking at Table 1, we can easily notice the substantial difference between the original implementation and the vectorized one in the order of magnitude of their execution time. For example, in some cases like F3 in WOA, the vectorized version is faster than the original implementation up to 1000 times. These numbers vary for different algorithms and benchmark functions based on the complexity of the algorithm mechanisms or function definitions. However, the general trend is that in higher dimensional search spaces and with large number of search agents, the speed-up ratio and the benefits of vectorization become more and more noticeable, as easily seen by comparing the rations between $n = 30$ and $n = 180$ in Table 2. To show the general trend, the speed-up ratio of WOA for several values of n is shown in Fig. 2. The first set of functions which are F1 through F6, show the biggest amount of improvement compared to the other functions. This is due to the fact that they only consist of simple operations like products and summations, and NumPy has very efficient built-in functions for these operations. Other functions like the ones in the second plot in Fig. 2 include more complex mathematical calculations and therefore need more time for their output to be computed. The fixed dimensional functions (F14–F23) show less improvement compared to the high dimensional functions because of their low dimensional search space. As mentioned before, vectorization is much more important when dealing with bigger and bigger matrices and vectors. To further illustrate the difference in the behavior of different algorithms and benchmark functions, Fig. 3 includes the speed-up ratios for the case $n = 60$. Similar to the trend visible in Fig. 2, we can see that the first few functions are the ones with the biggest speed-up ratios, and the low-dimensional functions show the least improvement among all the functions.

Although vectorization significantly reduces the execution time of these algorithms, the vectorized implementation might be less readable and interpretable than the loop-based implementation, and is also harder to be debugged and modified. In the loop-based implementation, the update equation for each dimension of each search agent can be accessed and modified inside the for loop of the corresponding element of the position matrix X. On the other hand, in the vectorized implementation, the operations are applied to many elements of the matrix X as a whole. This makes it more tricky to add new movement strategies and conditions for a particular set of agents to follow. In the loop-based approach, this just requires adding an else-if block inside the for loops, and because of the else-if structure, each agent will only follow one of the movement strategies. On the contrary, vectorized implementation needs extra caution to make sure this condition holds with the logical arrays that are going to be used as indexes. Another challenge is implementing the objective function using vectorization. Some operations need to introduce a dummy dimension into the matrix X and then taking a sum or average across that dimension which also needs extra caution while coding. To summarize, implementing objective functions and algorithms using vectorization might be challenging sometimes, but the trade-off is truly worth it, especially if you are working with high-dimensional objective functions and a high number of search agents.

6. Conclusion and future work

This work introduces an efficient approach in implementing metaheuristic, and in general population-based optimization algorithms by utilizing vectorization which is included in almost every programming framework that deals with matrices and tensors, like Matlab, Python's NumPy package, and so on. Every `for` loop in the original implementation of the 23 benchmark functions and the optimization algorithms have been replaced with an equivalent vectorized implementation and the built-in functions of NumPy. Based on the experiments, the vectorized implementation can be up to 1000 times faster than the original implementation in some cases for high dimensional search spaces with around 200 dimensions. This is due to the optimized and efficient allocation of the memory and the processing unit.

With this efficient implementation, there are now a lot more opportunities in adding new features and operations to these metaheuristic algorithms that were not possible or practical before because of the large computational time. It is important to note that the future improvements on these algorithms also have to be implemented in a vectorized approach and with the minimal use of `for` loops in order for the overall experiments to spend the least amount of time needed for their execution. In addition, the design cycle of improved versions and variants of these algorithms are now much less time-consuming, letting researchers evaluate their new ideas and improve them in a much faster framework.

Although there are many algorithms in the literature that can be vectorized in the same way as the ones in this paper, we only chose some of the most popular, simple, and powerful algorithms in recent years as an example to demonstrate the power of vectorization and the general procedure of vectorization for such algorithms.

CRediT author statement

AmirPouya Hemmasian: Conceptualization, Methodology, Software, Visualization, Writing - original draft. Kazem Meidani: Conceptualization, Visualization, Writing - original draft, Writing - review and editing. Seyedali Mirjalili: Supervision, Writing - review and editing. Amir Barati Farimani: Funding acquisition, Resources, Supervision.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the start-up fund provided by CMU Mechanical Engineering, United States and funding from National Science Foundation (CBET-1953222), United States.

Appendix A. Benchmark Functions

Fig. A.1, and Table A.1 show the form and landscape of the unimodal functions, and the multimodal functions are depicted and listed in Fig. A.2 and Table A.2, respectively.

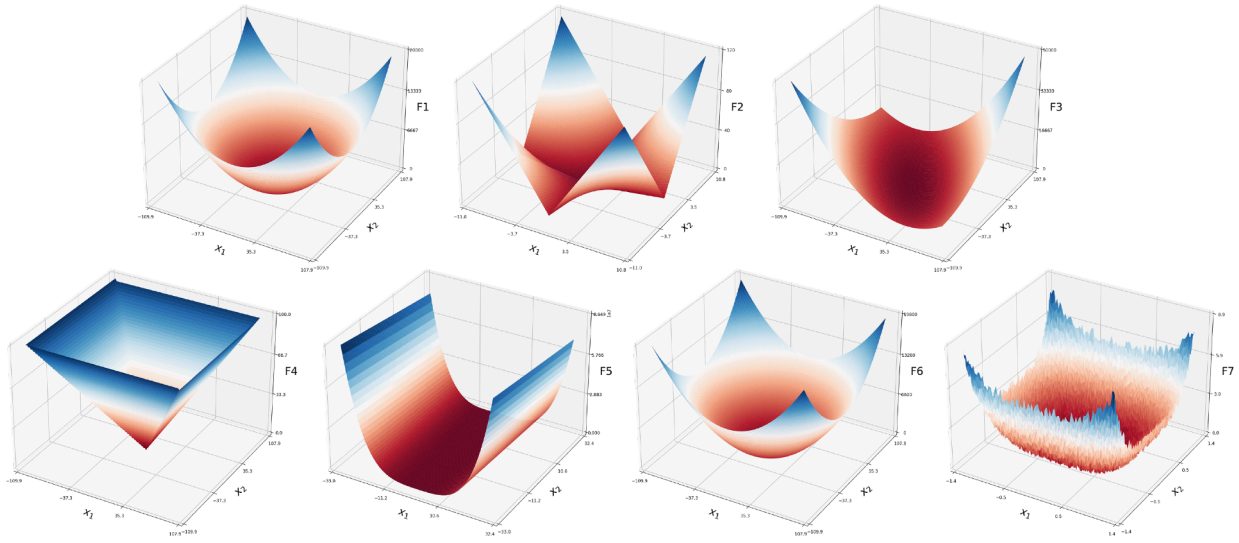

Fig. A.1. 2D version of unimodal test functions (30 dimensional versions are used for the experiments).

Table A.1
Unimodal benchmark functions used for the evaluation.

Function	Range	Dim
$F_1(x) = \sum_{i=1}^{N} x_i^2$	[− 100,100]	30
$F_2(x) = \sum_{i=1}^{N} \lvert x_i \rvert + \Pi_{i=1}^{N} \lvert x_i \rvert$	[− 10,10]	30
$F_3(x) = \sum_{i=1}^{N} (\sum_{j-1}^{i} x_j)^2$	[− 100,100]	30
$F_4(x) = \max_i \{\lvert x_i \rvert, 1 \leq i \leq N\}$	[− 100,100]	30
$F_5(x) = \sum_{i=1}^{N-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	[− 30,30]	30
$F_6(x) = \sum_{i=1}^{N} (\lvert x_i + 0.5 \rvert)^2$	[− 100,100]	30
$F_7(x) = \sum_{i=1}^{N} i x_i^4 + random[0,1)$	[− 1.28,1.28]	30

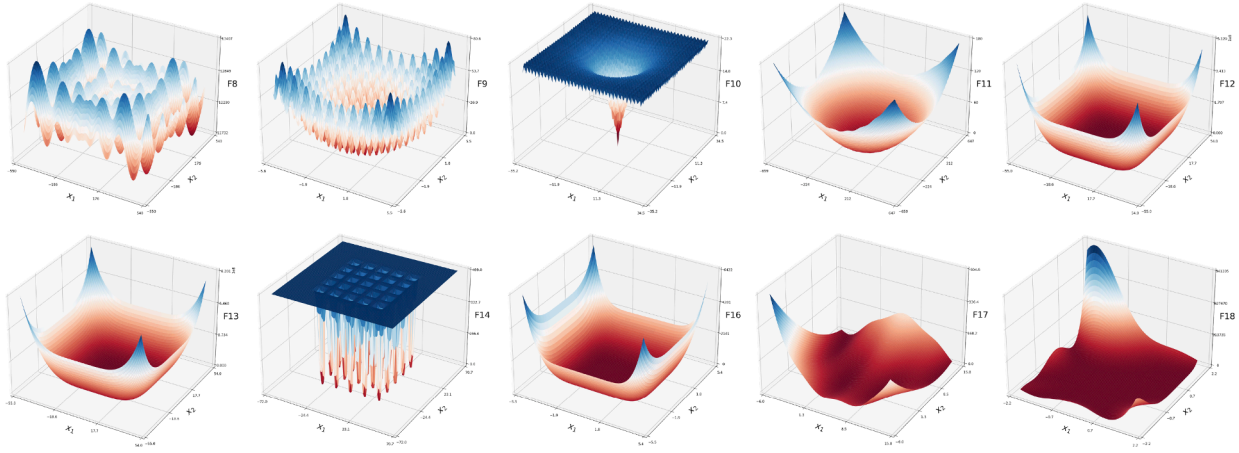

Fig. A.2. 2D version of some multimodal test functions ($F_8 - F_{13}$) and fixed dimension multimodal functions ($F_{14} - F_{18}$).

Table A.2
Multimodal benchmark functions used for the evaluation.

Function	Range	Dim
$F_8(x) = \sum_{i=1}^{N} -x_i \sin\sqrt{\lvert x_i \rvert}$	[− 500,500]	30
$F_9(x) = \sum_{i=1}^{N} [x_i^2 - 10\cos 2\pi x_i + 10]$	[− 5.12,5.12]	30
$F_{10}(x) = -20\exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}\right) - \exp\left(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)\right) + 20 + e$	[− 32,32]	30
$F_{11}(x) = \frac{1}{4000}\sum_{i=1}^{N} x_i^2 - \Pi_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	[− 600,600]	30
$F_{12}(x) = \frac{\pi}{N}\{10\sin(\pi y_1) + \sum_{i=1}^{N-1}(y_i - 1)^2\left[1 + 10\sin^2(\pi y_{i+1})\right] + (y_N - 1)^2\} + \sum_{i=1}^{N} u(x_i, 10, 100, 4), \quad y_i = 1 + \frac{x_i + 1}{4}$ $u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m & x_i > a \\ 0 & -a < x_i < a \\ k(-x_i - a)^m & x_i < -a \end{cases}$	[− 50,50]	30
$F_{13}(x) = 0.1\{\sin^2(3\pi x_1) + \sum_{i=1}^{N}(x_i - 1)^2[1 + \sin^2(3\pi x_i + 1)] + (x_N - 1)^2[1 + \sin^2(2\pi x_n)]\} + \sum_{i=1}^{N} u(x_i, 5, 100, 4)$	[− 50,50]	30
$F_{14}(x) = \left(\frac{1}{500} + \sum_{j=1}^{25}\frac{1}{j + \sum_{i=1}^{2}(x_i - a_{ij})^6}\right)^{-1}$	[− 65,65]	2
$F_{15}(x) = \sum_{i=1}^{11}\left[a_i - \frac{x_1(b_i^2 + b_i x_2)}{b_i^2 + b_i x_3 + x_4}\right]^2$	[− 5,5]	4
$F_{16}(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$	[− 5,5]	2
$F_{17}(x) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1)$	[− 5,5]	2
$F_{18}(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2 \times (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$	[− 2,2]	2
$F_{19}(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{3} a_{ij}(x_j - p_{-ij})^2\right)$	[1,3]	3
$F_{20}(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{6} a_{ij}(x_j - p_{ij})^2\right)$	[0,1]	6
$F_{21}(x) = -\sum_{i=1}^{5}[(X - a_i)(X - a_i)^T + c_i]^{-1}$	[0,10]	4
$F_{22}(x) = -\sum_{i=1}^{7}[(X - a_i)(X - a_i)^T + c_i]^{-1}$	[0,10]	4
$F_{23}(x) = -\sum_{i=1}^{10}[(X - a_i)(X - a_i)^T + c_i]^{-1}$	[0,10]	4

References

[1] Yang X-S, Deb S. Engineering optimisation by cuckoo search. Int J Math Model Numer Optim 2010;1(4):330–43.
[2] Gandomi AH, Yang X-S, Talatahari S, Alavi AH. Metaheuristic algorithms in modeling and optimization. Metaheuristic applications in structures and infrastructures. 1. Newnes; 2013. p. 1–24.
[3] Yang B, Wang J, Zhang X, Yu T, Yao W, Shu H, et al. Comprehensive overview of meta-heuristic algorithm applications on PV cell parameter identification. Energy Convers Manag 2020;208:112595.
[4] Soerensen JS, Johannesen L, Grove U, Lundhus K, Couderc J-P, Graff C. A comparison of IIR and wavelet filtering for noise reduction of the ECG. 2010 computing in cardiology. IEEE; 2010. p. 489–92.
[5] Abdel-Basset M, Abdel-Fatah L, Sangaiah AK. Chapter 10 - metaheuristic algorithms: acomprehensive review. In: Sangaiah AK, Sheng M, Zhang Z, editors. Computational intelligence for multimedia big data on the cloud with engineering applications. Intelligent data-centric systems. Academic Press; 2018, ISBN 978-0-12-813314-9. p. 185–231. https://doi.org/10.1016/B978-0-12-813314-9.00010-4.
[6] De León-Aldaco SE, Calleja H, Alquicira JA. Metaheuristic optimization methods applied to power converters: a review. IEEE Trans Power Electron 2015;30(12):6791–803.
[7] Memmah M-M, Lescourret F, Yao X, Lavigne C. Metaheuristics for agricultural land use optimization. A review. Agron Sustain Dev 2015;35(3):975–98.
[8] Holland JH. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT Press; 1992.
[9] Kennedy J, Eberhart R. Particle swarm optimization. Proceedings of ICNN'95-international conference on neural networks. vol. 4. IEEE; 1995. p. 1942–8.
[10] Storn R, Price K. Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces. J Global Optim 1997;11(4):341–59. https://doi.org/10.1023/A:1008202821328.
[11] Kaveh A, Khayatazad M. A new meta-heuristic method: ray optimization. Comput Struct 2012;112:283–94.
[12] Mirjalili S, Mirjalili SM, Lewis A. Grey wolf optimizer. Adv Eng Softw 2014;69:46–61.
[13] Mirjalili S. Moth-flame optimization algorithm: a novel nature-inspired heuristic paradigm. Knowl Based Syst 2015;89:228–49.
[14] Lee KS, Geem ZW. A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. Comput Methods Appl Mech Eng 2005;194(36):3902–33.
[15] Mirjalili S, Lewis A. The whale optimization algorithm. Adv Eng Softw 2016;95:51–67.
[16] Mirjalili S, Gandomi AH, Mirjalili SZ, Saremi S, Faris H, Mirjalili SM. Salp swarm algorithm: a bio-inspired optimizer for engineering design problems. Adv Eng Softw 2017;114:163–91.
[17] Desale S, Rasool A, Andhale S, Rane P. Heuristic and meta-heuristic algorithms and their relevance to the real world: a survey. Int J Comput Eng Res Trends 2015;351(5):2349–7084.
[18] Kannan S, Slochanal SMR, Padhy NP. Application and comparison of metaheuristic techniques to generation expansion planning problem. IEEE Trans Power Syst 2005;20(1):466–75.
[19] Hammouche K, Diaf M, Siarry P. A comparative study of various meta-heuristic techniques applied to the multilevel thresholding problem. Eng Appl Artif Intell 2010;23(5):676–88.
[20] Kaveh A. Applications of metaheuristic optimization algorithms in civil engineering. Springer; 2017.
[21] Sala R, Müller R. Benchmarking for metaheuristic black-box optimization: perspectives and open challenges. 2020 IEEE congress on evolutionary computation (CEC). IEEE; 2020. p. 1–8.
[22] Hussain K, Salleh MNM, Cheng S, Shi Y. Metaheuristic research: a comprehensive survey. Artif Intell Rev 2019;52(4):2191–233.
[23] Yang X-S, Ting T, Karamanoglu M. Random walks, Lévy flights, Markov chains and metaheuristic optimization. Future information communication technology and applications. Springer; 2013. p. 1055–64.
[24] Salimi H. Stochastic fractal search: a powerful metaheuristic algorithm. Knowl Based Syst 2015;75:1–18.
[25] Dorigo M, Birattari M, Stützle T. Ant colony optimization. IEEE, Comput Intell Mag 2006;1(4):28–39.
[26] Karaboga D, Basturk B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. J Global Optim 2007;39(3):459–71.
[27] Eberhart R, Kennedy J. A new optimizer using particle swarm theory. Proceedings of the sixth international symposium on micro machine and human science. 1995. p. 39–43.
[28] Yang X-S, Deb S. Cuckoo search via Lévy flights. Nature & biologically inspired computing, 2009. NaBIC 2009. World congress on. IEEE; 2009. p. 210–4.
[29] Passino K. Biomimicry of bacterial foraging for distributed optimization and control. IEEE Control Syst Mag 2002;22(3):52–67. https://doi.org/10.1109/MCS.2002.1004010.
[30] Gandomi AH, Alavi AH. Krill herd: a new bio-inspired optimization algorithm. Commun Nonlinear Sci Numer Simul 2012;17(12):4831–45.

[31] Mirjalili S. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems. Neural Comput Appl 2016;27(4):1053–73. https://doi.org/10.1007/s00521-015-1920-1.
[32] Saremi S, Mirjalili S, Lewis A. Grasshopper optimisation algorithm: theory and application. Adv Eng Softw 2017;105:30–47. https://doi.org/10.1016/j.advengsoft.2017.01.004.
[33] Mirjalili S. The ant lion optimizer. Adv Eng Softw 2015;83:80–98. https://doi.org/10.1016/j.advengsoft.2015.01.010.
[34] Van Der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22–30.
[35] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62.
[36] Crainic T. Parallel metaheuristics and cooperative search. Cham: Springer International Publishing; 2019, ISBN 978-3-319-91086-4. p. 419–51.
[37] Alba E, Tomassini M. Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 2002;6(5):443–62. https://doi.org/10.1109/TEVC.2002.800880.
[38] Hassani A., Treijs J.. An overview of standard and parallel genetic algorithms. 2009,.
[39] Alba E, Vidal P. Systolic optimization on GPU platforms. In: Moreno-Díaz R, Pichler F, Quesada-Arencibia A, editors. Computer aided systems theory – EUROCAST 2011. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 375–83.
[40] Talbi E-G, Bachelet V. Cosearch: a parallel cooperative metaheuristic. J Math Model Algorithms 2006;5(1):5–22. https://doi.org/10.1007/s10852-005-9029-7.
[41] Faris H, Aljarah I, Mirjalili S, Castillo PA, Guervós JJM. Evolopy: an open-source nature-inspired optimization framework in Python. IJCCI (ECTA). 2016. p. 171–7.
[42] Digalakis J, Margaritis K. On benchmarking functions for genetic algorithms. Int J Comput Math 2001;77(4):481–506. https://doi.org/10.1080/00207160108805080.
[43] Mirjalili S. SCA: a sine cosine algorithm for solving optimization problems. Knowl Based Syst 2016;96:120–33.