Project Assignment - Fuzzy Systems and Evolutionary Computing (a.y. 2024/2025)

Instructions

The project is optional and can be realized individually or in small groups (max 2 people). The project must be realized in Python (v. 3+) and must be consistent with the softpy library interface that we've seen in the laboratory lesson. You are free (and encouraged) to use additional libraries or code in your implementation. The code must be submitted by sending an email to **both** Prof. Ciucci (davide.ciucci@unimib.it) and Prof. Campagner (andrea.campagner@unimib.it) with subject "Project Assignment - Fuzzy Systems and Evolutionary Computing (a.y. 2024/2025)". The text of the email should include the names and matriculation numbers of the students who participated in implementing the project. The project must be attached as a .zip file containing Python (.py files) or Jupyter Notebook (.ipynb files) files containing the implementation of the project itself.

Grading

The project will be evaluated in terms of three criteria:

- Correctness (does it realize the desired specification without errors?)
- Completeness (does it address all the given requirements?)
- Consistency (does the implementation have good code quality and conforms with the softpy interface?). By good quality we mean: the code should be duly documented, it should not feature unnecessary repetitions (i.e., encapsulate re-usable functionalities in support functions or methods), it should have clear naming conventions for methods and variables, etc.

Depending on the degree of satisfaction of these three criteria the project can contribute up to 4 points to the final degree (such points will be directly added to the vote resulting from the written/oral examination, provided this latter is at least equal to 18).

Task Description

The objective is to produce a working implementation of **Particle Swarm Optimization** (PSO), a population-based meta-heuristic algorithm that you have seen in class. The general idea of PSO is that we maintain a population of candidate solutions (called *particles*), represented in terms of their *position* (the actual representation of the solution) and *velocity* (which is used to modify the solutions). At every iteration, each particle p_i is assigned a set of neighbors $(p_1, \ldots, p_k)$, and its velocity is updated based on its own velocity v_i, the best known position b_i found by particle i, the best known position found by one of its neighbors b_N, and the best known position found by any particle b. Then, the position of the particle is updated based on the velocity.

The implementation should be compatible with the softpy library. More specifically, you will have to address the following requirements:

1. Particles (i.e., candidate solutions) must be implemented as a class `ParticleCandidate` which must inherit from the softpy class `FloatVectorCandidate`. This class must have the following attributes:

   - *size*: an integer, the number of components in the position
   - *lower*: a numpy array of *size* cells and of type float, the smallest possible value for each position
   - *upper*: a numpy array of *size* cells and of type float, the largest possible value for each position
   - *candidate*: a numpy array of *size* cells and of type float. Cell i of this array must contain numbers between $lower[i]$ and $upper[i]$
   - *velocity*: a numpy array of *size* cells and of type float
   - *inertia*: a float
   - w_l: a float
   - w_n: a float
   - w_g: a float

   with the constraint that $w_l + w_n + w_g = 1$. The class should also implement the following methods:

   - *__init__*: the constructor. It must accept *size*, *lower*, *upper*, *candidate* and *velocity* as input parameters
   - *generate*: it must accept *size*, *lower*, *upper* as input parameters. It must generate a *candidate* (numpy array of *size* cells) by drawing values uniformly at random between *lower* and *upper*. It then must generate a *velocity* (numpy array of *size* cells) by drawing values uniformly at random between $-|upper - lower|$ and $|upper - lower|$. It must then call the constructor with the given input parameters and return its result.

- *mutate*: it has *self* as its only input parameter. It must change the value of *candidate* to $candidate + velocity$.
- *recombine*: it must accept *self* as well as three other `ParticleCandidate` instances (*local_best*, *neighborhood_best*, *best*) as its input parameters. It must update *velocity* according to the following formula:

$$\begin{aligned} velocity \leftarrow\ & inertia \cdot velocity \\ & + r_l \cdot w_l \cdot (candidate - local\_best) \\ & + r_n \cdot w_n \cdot (candidate - neighborhood\_best) \\ & + r_g \cdot w_g \cdot (candidate - best) \end{aligned}$$

  where r_l, r_n, r_g are numbers selected uniformly at random between 0 and 1.

2. The PSO algorithm must be implemented as a class `ParticleSwarmOptimizer`, which must inherit from the softpy class `MetaHeuristicsAlgorithm`. This class must have the following attributes:

   - *pop_size*: an integer, the size of the population
   - *population*: a list (or numpy array) of *pop_size* `ParticleCandidate` instances
   - $fitness\_func$: a fitness function that takes as input a `ParticleCandidate` and returns a number
   - *n_neighbors*: an integer, the number of neighbors for each particle
   - *best*: an array of `ParticleCandidate` instances of size *pop_size*. For each particle i, it must contain the *position* which gave the **largest** fitness for that particle
   - $fitness\_best$: a numpy array of floats, of size *pop_size*. For each particle i, it must contain the **largest** fitness value found so far for that particle
   - *global_best*: a `ParticleCandidate`. It must contain the *position* that gave the **largest** fitness value found so far
   - $global\_fitness\_best$: a float. It must contain the **largest** fitness value found so far

   The class must also implement the following methods:

   - *__init__*: the constructor. It must accept $fitness\_func$, *pop_size*, *n_neighbors* and $**kwargs$ as its input parameters.
   - *fit*. It must accept *n_iters* as its only input parameter. As a first step, the method should create a *population* (of size *pop_size*) of `ParticleCandidate` instances, by calling the respective *generate* method with the content of $**kwargs$ as input. For *n_iters* times, it must then perform the following operations:

(a) For each candidate in the *population*, it computes its fitness by calling $fitness\_func$.

(b) It updates the *best* and $fitness\_best$ arrays, by updating each cell if the new value of the fitness is **larger** than the current best. It also updates *global_best* and $global\_fitness\_best$ in the same way.

(c) For each candidate i in the *population*, it randomly selects $n\_neighbors$ other candidates in the population (with indices $i_1, \ldots, i_{n\_neighbors}$). It then identifies the neighbor, $best\_neighbor[i]$, of i for which $fitness\_best$ is **largest** (among the neighbors of i, i excluded).

(d) For each candidate i in the population, it calls the *recombine* method passing $best[i]$, $best\_neighbor[i]$ and *global_best* as input parameters. Then it calls the *mutate* method

Aside from the attributes and methods explicitly required above, you can also define other attributes, methods (or also global functions) if needed.