Nonlinear Constrained Continuous Optimization with `OptimizationModel`

`OptimizationModel` is a class that can model and optimize a nonlinear constrained continuous programming problem. A general continuous programming problem in mathematical notation could be as follows:

$$\min z = f(\vec{\mathbf{x}})$$

subject to:

$$G(\vec{\mathbf{x}}) \leq \overrightarrow{\boldsymbol{b}_k} \in \mathbb{R}^k$$

$$E(\vec{\mathbf{x}}) = \overrightarrow{\boldsymbol{d}_m} \in \mathbb{R}^m$$

where $G: \mathbb{R}^n \to \mathbb{R}^k$ and $E: \mathbb{R}^n \to \mathbb{R}^m$ are two functions that represent the constraints, $f: \mathbb{R}^n \to \mathbb{R}$ is the objective function and $\vec{\mathbf{x}} \in \mathbb{R}^n$.

Since any maximization problem can be converted to a minimization problem and any greater than inequalities can be converted to less than inequalities, this model is assumed for generality. Nevertheless, the `OptimizationModel` supports adding greater-than inequalities too.

`OptimizationModel`:

The objects of this class, called an optimization model, can be used to set an objective function and constraints to optimize. Optimization could also be maximization instead of the default minimization. When an optimization model is created, it expects only the dimension of the domain, also called the search space, n to set up the model. After the creation of the model, the model will have the following attributes:

- `objective_function`: The objective function, f
- `E`: The function that represents the equality constraints, $E(\vec{\mathbf{x}})$
- `E_RHS`: The right-hand side of the equality constraints, $\overrightarrow{\boldsymbol{d}_m}$
- `G`: The function that represents the less-than constraints, not present in the general model, added for support
- `G_RHS`: The right-hand side of the less-than constraints, not present in the general model, added for support
- `H`: The function that represents the greater-than constraints, not present in the general model, added for support

- `H_RHS`: The right-hand side of the greater-than constraints, not present in the general model, added for support
- `current_results`: This is a dictionary that will be set after calling the optimization method
- `dim`: The dimension of the search space, n
- Other 2 attributes at the end are only for auxiliary purposes

The differentiation between greater-than and less-than constraints in the model are for general convenience.

`AddConstraint` method:

This method takes a function defined on $\mathbb{R}^n$, a real number for right-hand side and a string as type to create a constraint, and then adds it to the model. For example, $g_i: \mathbb{R}^n \to \mathbb{R}$ with $b_i \in \mathbb{R}$ and type "$\leq$" would add the following constraint to the model:

$$g_i(\vec{\mathbf{x}}) \leq b_i$$

So, it is easy to see that $G(\vec{\mathbf{x}})$ is composed of such many functions as follows:

$$G(\vec{\mathbf{x}}) = [g_1(\vec{\mathbf{x}}), \dots g_i(\vec{\mathbf{x}}), \dots, g_k(\vec{\mathbf{x}})]^T$$

and

$$\overrightarrow{\boldsymbol{b}_k} = [b_1, \dots, b_i, \dots, b_k]^T.$$

The same logic also directly applies to the other types of constraints too. Hence the following strings as type are expected: "$=$", "$\leq$" and "$\geq$".

`BoundVariables` method:

This method of the class takes a tuple (a, b) and an index value where a and b are in $\mathbb{R} \cup \{-\infty, \infty\}$ and $a < b$. The index value can be a valid integer ranging from 0 to $n - 1$, or it can be a `slice` object. This method can be used to add constraints to bound individual variables in $\vec{\mathbf{x}}$ so that each i-th component of $\vec{\mathbf{x}}$ given in the index object is bounded between a and b. A `slice` object can be given as index to bound many variables between a and b at the same time.

`SetObjective` method:

As the name suggests, this method takes a function and sets it as the objective of the model. `objective` keyword argument is default to "min" but can be changed to "max" to maximize the given function.

`SetConstraints` method:

This method just sets all previously given constraints to the model, ready to be fed into the optimizer method. It does not take any input.

`StartOptimization` method:

This method starts the optimization procedure for the model. It uses a modified version of multi-swarm optimization algorithm. Input arguments for this method are as follows:

- `search_space_size`: This is the edge length of the hypercube $[-L, L]^n$ where the particles are initialized to start the search
- `team_size`: The number of particles in each team, default value is 30
- `team_number`: The number of teams to initialize, default value is 10
- `social_coefficient`: The so-called social coefficient of the particles, default value is 0.95
- `cognitive_coefficient`: The so-called cognitive coefficient of the particles, default value is 0.95
- `inertia`: The inertia of the particles, default value is 0.75 – Note that a decreasing inertia during optimization is utilized where the final inertia is 0.4
- `use_merge_and_exploit`: Whether to use a sub-procedure called Merge&Exploit or not, default value is `True`
- `keep_exploration_short`: Whether to keep feasible region search short or not, default value is `True`

All these arguments are related to the details of the algorithm; hence, they will be explained in the next section where the `SpecializedTeams` algorithm is introduced and documented.

The `StartOptimization` method supports documentation during and after the optimization procedure. It uses `time` class to measure the time passed during the optimization, it can print out the finished stage of the optimization, and in the end, it prints out the results of the optimization.

To summarize, this class can be instantiated to use model and optimize a nonlinear constrained continuous programming model in almost the most general case. For mixed problems where some variables must be integer, some approaches are in research; however, no built-in support exists yet.

Introduction to PSO

What is PSO?

Particle swarm optimization is heuristic algorithm in which a swarm of particles are initialized in a search space $\Omega \subset \mathbb{R}^n$ and set to move throughout the space, searching for some optima. Generally, the initialized search space is a hypercube of the form $\Omega = [-L, L]^n$. In PSO, each particle has its own position and velocity which are updated each iteration. Let there be N particles and let $i \in \{1,2,3, \dots, N\}$. Then, position and velocity of the particles are set and then updated throughout the iterations as follows:

$$p_{i,0} \sim U(\Omega)$$

$$v_{i,0} \sim U([-2L, 2L]^n)$$

$$v_{i,k} = \omega \cdot v_{i,k-1} + r_1 \phi_{cog}\left(p_{i,best} - p_{i,k-1}\right) + r_2 \phi_{soc}\left(g_k - p_{i,k-1}\right)$$

$$p_{i,k} = p_{i,k-1} + \chi \cdot v_{i,k}$$

Here, $p_{i,k}$ and $v_{i,k}$ are the position and velocity of the i-th particle at iteration-k, respectively. $U(\mathcal{A})$ is a random variable uniformly distributed over a set $\mathcal{A}$. ω is the so-called inertia of the particles. χ is the constriction factor. r_1 and r_2 are two independently drawn uniform numbers from the interval $[0,1]$. ϕ_{soc} and ϕ_{cog} are the social and cognitive coefficients, respectively. Given f is the objective to optimize, $p_{i,best}$ is the most optimum position of f found by the particle-i until the iteration-k. Similarly, g_k is the most optimum position found by the entire swarm until the iteration-k.

This so-called velocity update formula immediately tells us how each particle moves: after initialization, each particle moves in the direction that is a random combination of its previous destination, the “best” position itself found and the “best” position found by the entire swarm.

This is called the g-best PSO since the best position found by the entire swarm is used in the velocity update formula. Another approach is l-best PSO where a local best position, where locality is determined between the particles according to some metric, is substituted instead of the global best found by the entire swarm. For further information on PSO, and comparison between g-best PSO and l-best PSO; please check the first three entries of the references.

What are the problems with g-best and l-best PSOs?

In vanilla g-best PSO, we have two main problems: getting stuck at a local optimum and premature convergence. These two problems are deeply interrelated. For constrained optimization, this algorithm also frequently fails to find the feasible region or to find multiple disjoint feasible regions that arise from nonlinear constraints. The other variant l-best PSO was proposed to solve such problems to some extent. This variant is expected to do better since particles explore and exploit the space more locally. However, in the article [2], it is argued that choosing g-best or l-best variant is another hyperparameter open for tuning, and there is not even a generalizable rule of thumb for which to choose. Furthermore, another problem with l-best is that it introduces complexity to the algorithm and adds computational cost. Because l-best PSO introduces the notion of locality that must be checked using a pre-determined metric every iteration. This especially endangers a potential parallelization of the algorithm.

For these reasons, I developed a variant of multi-swarm algorithm where many swarms are used instead of only one, so-called Specialized Teams algorithm.

Specialized Teams Algorithm:

In the Specialized Teams (ST) algorithm, a multi-swarm approach is employed. In this approach, instead of using one big swarm of particles, many swarms are used and ran in the algorithm independently. Furthermore, this algorithm also employs three other procedures to enhance exploration and exploitation: feasible region search, chaotic exploration and Merge&Exploit session. Before introducing these procedures, let us introduce how the particles move and how the constraints are enforced in the ST algorithm.

Particle Set-up and Update Formula of ST:

TS algorithm uses an initially given search space of the form $\Omega = \vec{\boldsymbol{\mu}} + [-L, L]^n \subset \mathbb{R}^n$ where $\vec{\boldsymbol{\mu}} \in \mathbb{R}^n$ is the center of the hypercube, generally the origin. Given the team size $N \geq 2$ and team number $K \geq 2$, a radius $r_c = \frac{L}{\sqrt[n]{K}}$ is determined. In ST, each team is initialized around a center called team center. The team centers are initialized in the search space so that two different team centers have at least $1.5r_c$ separation between them. This provides some separation between each team so that they explore and exploit different subsets

of the search space. Then, the particles of a team have a Gaussian distribution around the team center with each component having a standard deviation of $\frac{r_c}{2}$ and no intercomponent correlation. Finally, initial velocities are set by normalizing a random vector uniformly distributed on $[0,1]^n$ and then scaling by another random scalar uniformly distributed on $[0, r_c]$. Note that these details of the initialization are tentative.

Let $p_{i,j,k}$ and $v_{i,j,k}$ be the position and velocity of the particle-i in team-j at iteration-k. Then the following formulae are used to update the positions and velocities:

$$v_{i,j,k} = \omega \cdot v_{i,j,k-1} + r_1\phi_{soc}\left(g_{j,k-1} - p_{i,j,k-1}\right) + r_2\phi_{cog}(p_{i,best,k-1} - p_{i,j,k-1})$$

$$p_{i,j,k} = p_{i,j,k-1} + \chi \cdot v_{i,j,k}$$

Here, $g_{j,k-1}$ is the best position found by all the particles of team-j until the iteration-k and $p_{i,best,k-1}$ is the best position found by the particle itself, particle-i of the team-j until the iteration-k. Other coefficients and parameters are as explained above, same as they are in the g-best PSO.

Penalty Method: Penalty method is a method to convert constraints into another objective to optimize. So, let's say we have the following model to minimize:

$$\min z = f(\vec{\mathbf{x}})$$

subject to:

$$G(\vec{\mathbf{x}}) \leq \overrightarrow{\boldsymbol{b}_k} \in \mathbb{R}^k$$

$$E(\vec{\mathbf{x}}) = \overrightarrow{\boldsymbol{d}_m} \in \mathbb{R}^m$$

where $G: \mathbb{R}^n \to \mathbb{R}^k$ and $E: \mathbb{R}^n \to \mathbb{R}^m$ are two functions that represent the constraints, $f: \mathbb{R}^n \to \mathbb{R}$ is the objective function and $\vec{\mathbf{x}} \in \mathbb{R}^n$.

We convert this programming problem into the following:

$$\min z = f(\vec{\mathbf{x}}) + P(\vec{\mathbf{x}})$$

where $\vec{\mathbf{x}} \in \mathbb{R}^n$ and $P(\vec{\mathbf{x}}) = C\left\|E(\vec{\mathbf{x}}) - \overrightarrow{\boldsymbol{d}_m}\right\|_1 - C \cdot \min(\min\left(\overrightarrow{\boldsymbol{b}_k} - G(\vec{\mathbf{x}})\right), 0)$, and $C > 0$ is a big constant. The penalty function here is 0 whenever $\vec{\mathbf{x}}$ is in the feasible region, a big positive number otherwise due to C. In a way, the penalty function here just represents the slacks of the constraints. Note that $\|\cdot\|_1$ is the

L_1 norm – sum of the absolute values of the vector. Basically, the idea is that since we have a minimization problem, minimizing the penalty function to 0 with the objective f gives us the feasible optimality of f, ideally.

However, notice that this is a multi-objective problem disguised as one objective problem. We have two *objectives*: minimize f and minimize P. So, if we naively apply a heuristic to this problem, we can observe unintended behavior such as minimizing P but not minimizing f adequately. Generally, it is really difficult to tune the balance between two objectives during the optimization. This is a big problem since it is often the case that neither local nor global optimality is guaranteed in heuristic algorithms. On top of this, if the algorithm fails to find a feasible point or to optimize f sufficiently, one cannot check if this is due to the model itself (e.g., feasible region is empty) or heuristic was not run for enough iterations. Hence, before optimizing the true objective f, we firstly run the algorithm to search for feasible region to minimize the penalty (finding the near/exact feasible region). Then we find the minimum of f in this region while keeping the penalty at its lowest.

Here, we can begin to introduce the aforementioned procedures.

1) Feasible Region Search

Put simply, feasible region search is the procedure where all the teams try to minimize the penalty function. True objective is not taken into account during this phase. By the design of the penalty function P, as given above, P decreases whenever the point is "more feasible". It is intentionally not a 0-1 function where it is 0 if the point is feasible and 1 if it is not. Because a function that decreases to 0 gradually when the point gets closer to the feasible region is far better to find the feasible region by this algorithm.

The team structure here also helps with finding feasible region search in two distinct ways:

a) Each team can locally search their own local subset of search space for feasible regions, letting the algorithm explore many disjoint feasible regions if there are any
b) Makes it easier and more efficient to search and explore the feasible regions formed by equality constraints

The first one here is already discussed. The second one is rather tricky. This is because the equality constraints reduce the "dimension" of the feasible region. Consider the following constraint:

$$\vec{\mathbf{x}} = [x_1, x_2]^T \in \mathbb{R}^2$$

subject to:

$$x_1 + x_2 = 1$$

This constraint represents a line the plane $\mathbb{R}^2$. If we assume this is the only constraint, then it follows that the feasible region has a dimension of 1, where the search space is a 2 dimensional plane. This introduces a great difficulty when finding the feasible region, especially when the heuristics deal with high dimensional spaces. In this example, if a g-best PSO is used, in the best scenario it converges to a random point on this line during the feasible region search. However, it can be expected that this point lies sufficiently far from the optimum point for the true objective which can diminish the performance and efficiency. This worsens in high dimensional cases and when f is not a gradually decreasing continuous function. This situation is improved greatly by ST algorithm where many teams are expected to converge to random but different points near or within the feasible region. Consider another example as follows:

$$\vec{\mathbf{x}} = [x_1, x_2]^T \in \mathbb{R}^2$$

subject to:

$$x_1 + x_2 = 0$$

$$x_1^2 + x_2^2 \geq 1$$

$$x_1^2 + x_2^2 \leq 2.$$

In this example, the feasible region is even more pathological as can be seen below, two blue line segments form the entire feasible region in this case:

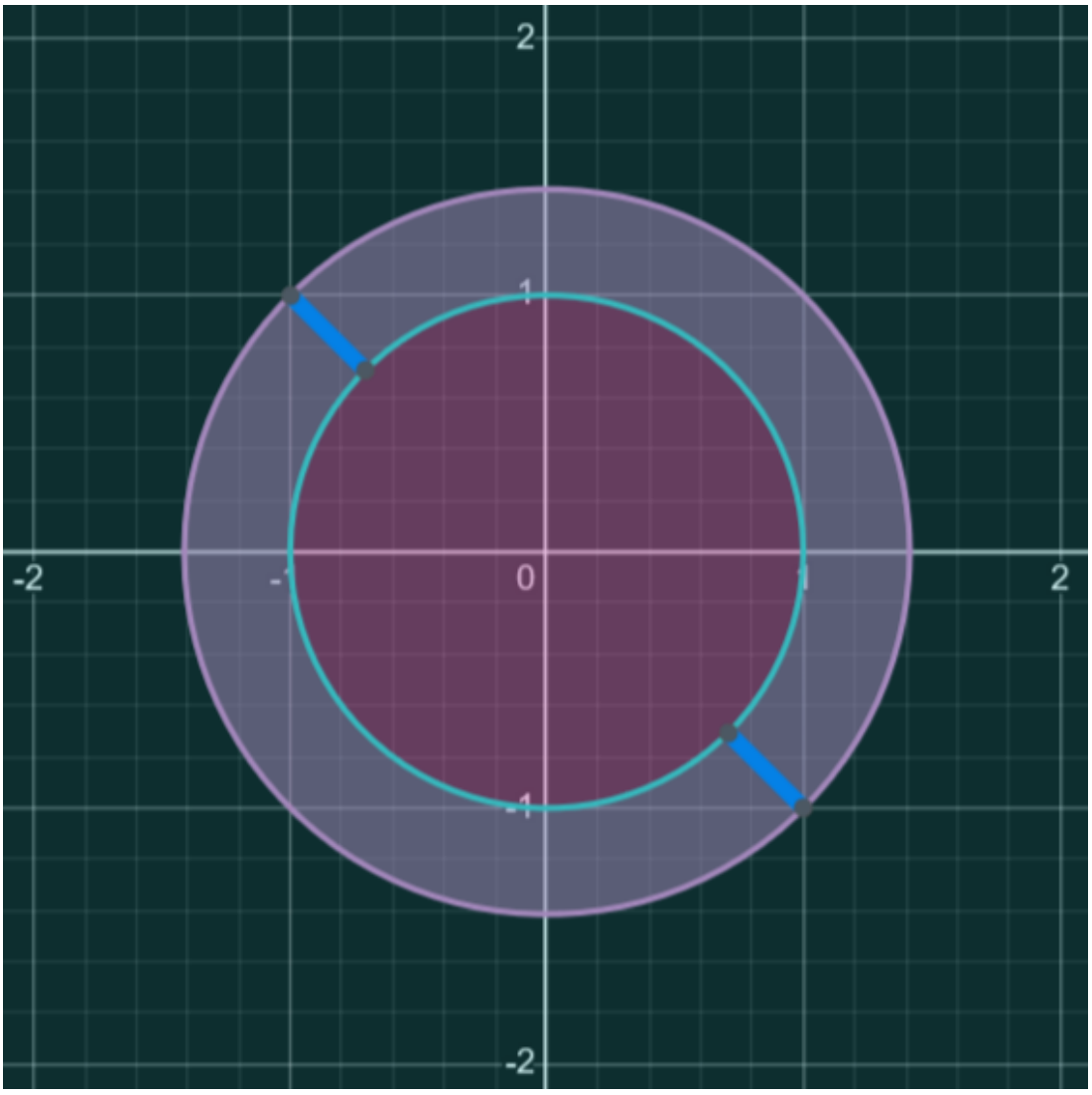

Figure: The constraints and the feasible region sketched with Desmos

Even in this very simple scenario, both g-best PSO and l-best PSO can fail. The failure of g-best PSO is rather obvious since all particles are expected to converge to one point (or to its close neighborhood) on one of the blue line segments shown in the figure, missing an entire other segment of the feasible region. The variant l-best PSO can fail because the metric used to evaluate locality might not let enough number of particles to converge to one of the two line segments of this feasible region. ST is expected to solve this problem by expecting at least two different teams to converge to different line segments, reaching successfully to both of the disconnected parts of the feasible region. This is expected because although the teams are fully independent of each other, the information flow between the particles of each team is instantaneous, like it is in g-best PSO. This lets each team discover different regions fully and independently.

2) Chaotic Exploration:

This procedure exists to help with getting the particles out of local optimum and let each particle explore their nearby neighborhood randomly. Hence the name, chaotic. This is a procedure related to particles individually, so it is not related to the team structure, not even related to the PSO itself. Its aim is to move the particles randomly around their current positions to find new and better regions by ignoring everything and moving them according to formulae:

$$v_{i,j,k} = v_{i,j,k-1} + \vec{\mathbf{X}}$$

$$p_{i,j,k} = p_{i,j,k-1} + v_{i,j,k}$$

where $\vec{\mathbf{X}} = [x'_1, \dots, x'_l, \dots, x'_n]^T$ such that $x'_l \sim \mathcal{N}(0, \sigma \cdot r_c)$ are identically and independently distributed. Best positions $g_{j,k}$ and $p_{i,best,k}$ are updated each iteration if better values obtained.

It is observed that chaotic exploration works well when the objective function has a pathological structure (e.g., discontinuous jumps) or many optima near each other.

3) Merge&Exploit:

This procedure is applied after the optimization with team structure is done, at the end of all the other procedures. In this procedure, all particles are treated as if they are in one swarm. Indeed, this procedure is the same as applying the g-best PSO algorithm to the existing particles. This procedure aims to explore the close neighborhood of the best position found by the teams and improve the accuracy of the optimum. Exploration happens when particles approach to the global best position, best position found by the teams, from all directions because teams are expected to have converged to different local optima. This is why it is called merge and exploit, all particles are merged into a swarm and then they are let converge to a final optimum.

How does ST Algorithm work in `OptimizationModel`?

`SpecializedTeams` class provides all the methods required to run the procedures mentioned and more. So, an instance of this class can be used to set the teams up and assign the parameters, constraints and the objective.

In `OptimizationModel` class, after setting up the model and calling the `StartOptimization` method of the model, an instance of `SpecializedTeams` class is made. Then, the parameters, constraints and objective of the model are

assigned to this instance. The following is the default behavior. After this setup, the actual optimization procedure starts with feasible region search procedure. After this, the optimization of $f + P$ starts. P is kept in the rest of the process to keep the feasibility. During the iterations of the ST algorithm, a given stopping criteria is checked. If the criteria is met, then it is checked whether to apply chaotic sessions. If not, then optimization process is done. Otherwise, one chaotic exploration session is applied and then the loop proceeds. The optimization process stops either when the criteria is met and there are no chaotic exploration sessions to apply or when the iteration number reaches to a maximum limit. After then, by default, a Merge&Exploit procedure is applied after applying one more chaotic exploration session. This session is applied to make sure that all particles are slightly spread instead of being clumped up on each other before the Merge&Exploit procedure.

The `Testing_main.py` file includes some example objective functions and will include more models with constraints.

**Note that both the document and the codes are still lacking many important details. Moreover, the algorithm needs many changes and improvements, some of which are already being prepared. Hence, some of the documentation is tentative. However, the main idea of ST algorithm will stay mostly the same. The expected changes are, for now, configurability of the model and teams and a self-tuning algorithm setting (See ref. [3]).*

References and Further Reading:

1. A summary of the vanilla g-best PSO algorithm with pseudocode: https://en.wikipedia.org/wiki/Particle_swarm_optimization#Algorithm
2. *Particle Swarm Optimization: Global Best or Local Best?* by AP Engelbrecht. DOI: https://doi.org/10.1109/BRICS-CCI-CBIC.2013.31
3. Planned-to-be-implemented: *Fuzzy Self-Tuning PSO: A settings-free algorithm for global optimization* by Nobile et. al. DOI: https://doi.org/10.1016/j.swevo.2017.09.001
4. A review article – *A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications* by Zhang et. al. DOI: https://doi.org/10.1155/2015/931256