AutoOptLib Documentation

April 2025

1 Getting Started

1.1 Introduction

AutoOptLib is a Matlab/Octave library for automatically designing metaheuristic optimizers. It provides:

1. Rich library of design choices - Over 40 representative metaheuristic components for designing algorithms for continuous, discrete, and permutation problems with/without constraints and uncertainties (Table 1).
2. Flexibility to designing diverse algorithms - Design algorithms with diverse structures in a single run, enables great possibility to find novel and efficient algorithms.
3. Fair benchmark of various design objectives and techniques - Various design objectives, e.g., solution quality, runtime, and anytime performance (Table 2). Different design techniques, e.g., racing, intensification, and surrogate (Table 3).
4. Good accessibility - Graphical user interface (GUI) for users to input problems, manage the algorithm design process, make experimental comparisons, and visualize results with simple one-clicks.
5. Easy extensibility - Easily add new algorithm components, objectives, and techniques by a uniform interface.

AutoOptLib's benefits include:

1. Save labor resources and time - Human experts may cost days or weeks to conceive, build up, and verify the optimizers; AutoOptLib would saves such labor resources and time costs with today's increasing computational power.
2. Democratize metaheuristic optimizers - Through automated algorithm design techniques, AutoOptLib would democratize the efficient and effective use of metaheuristic optimizers. This is significant for researchers and practitioners with complicated optimization problem-solving demands but without the expertise to distinguish and manage suitable optimizers among various choices.
3. Surpass human algorithm design - By fully exploring potential design choices and discovering novelties with computing power, AutoOptLib would go beyond human experience and gain enhanced performance regarding human problem-solving.
4. Promote metaheuristic research - With a uniform collection of related techniques, AutoOptLib would promote research of the automated algorithm design and metaheuristic fields and be a tool in the pursuit of autonomous and general artificial intelligence systems.

1.2 Installation

AutoOptLib is downloadable at https://github.com/auto4opt/AutoOpt. The Matlab source code is recommended to be executed via Matlab R2018 or higher versions; the Octave source code is recommended to be executed via Octave 8.3 or higher versions. Matlab R2020a or higher versions are required for invoking the GUI. Users can use, redistribute, and modify it under the terms of the GNU General Public License v3.0.

1.3 Quick Start

Following the steps to use AutoOptLib:

1. Download AutoOptLib and add it to MATLAB/Octave path.
2. Implement the target optimization problem.
3. Define the space for designing algorithms.
4. Run AutoOptLib by command or GUI.

Steps 2, 3, and 4 will be detailed in Sections 2.3.1, 2.3.2, and 2.3.3, respectively.

1.4 Contact

AutoOptLib is developed and maintained by the Swarm Intelligence laboratory, Department of Computer Science and Engineering, Southern University of Science and Technology.

Users may ask question in the Issues block and upload contributions by Pulling request in AutoOptLib's Github repository (https://github.com/auto4opt/AutoOpt).

For any question, comment, or suggestion, please feel free to get in touch with Dr. Qi Zhao, Department of Computer Science and Engineering, Southern University of Science and Technology, email: zhaoq@sustech.edu.cn.

2 User Guide

2.1 What is automated algorithm design?

Given a target problem, through algorithm design, we would like to find an algorithm(s) with the best performance on the problem [BK09]:

$$\begin{aligned} &\arg\max_{A} \ \mathbb{E}[\mathbb{E}[P(A)|i]|\mathcal{I}], \\ &\text{s.t.} \quad A \in \mathcal{S}, \end{aligned} \tag{1}$$

where A is the designed algorithm; $\mathcal{S}$ is the design space, from where A can be instantiated; $i \in \mathcal{I}$ is an instance of the target problem domain $\mathcal{I}$; $P : \mathcal{S} \times \mathcal{I} \to \mathbb{R}$ is a performance metric that measures the performance of A in $\mathcal{I}$. The design aims to find algorithm(s) with the maximum expected performance in $\mathcal{I}$.

In reality, the distribution of problem instances in $\mathcal{I}$ is often unknown, and one cannot exhaust all the instances during the design process. The common practice of settling for the reality is to consider a finite set of instances from $\mathcal{I}$. Consequently, Eq. (1) can be reformulated as

$$\begin{aligned} &\arg\max_{A} \ \mathbb{E}[\mathbb{E}[P(A)|i]|I_t], t = 1, 2, \cdots, T \\ &\text{s.t.} \quad A \in \mathcal{S}, \\ &\qquad\quad I_t \subseteq \mathcal{I}, \forall t \in \{1, 2, \cdots, T\} \end{aligned} \tag{2}$$

where I_t is the finite set of instances that are target at time (i.e., iteration[1]) t of the design process. The target problem instances can either be fixed (i.e., $I_1 = I_2 = \cdots = I_T$) or dynamically changed during the design process. The output of solving Eq. (2) is algorithm(s) with the best performance on the considered instances. To avoid the designed algorithms overfitting, the design process can be followed by validation to investigate the generalization of the designed algorithms to instances from $\mathcal{I} \backslash \{I_1, I_2, \cdots, I_T\}$.

The general process of automated design of metaheuristic optimizers can be abstracted into four parts, as shown in Fig. 1. First, the design space collects of candidate primitives or components for instantiating metaheuristic algorithms. It regulates what algorithms can be found in principle. Second, the design strategy provides a principle way to design algorithms by selecting and combining the primitives or components from the design space. Third, the performance evaluation strategy defines how to measure the performance of the designed algorithms. The measured performance guides the design strategy to find desired algorithms. Finally, because the design aims to find algorithms with promising performance on solving a target problem, the target problem acts as external data to support the performance evaluation.

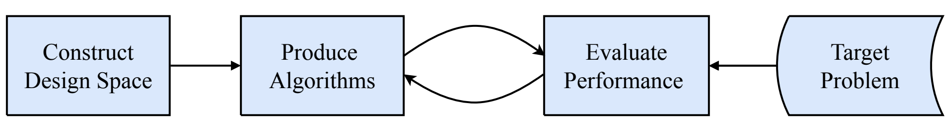

Figure 1: Process of automated design of metaheuristic optimizers.

2.2 AutoOptLib Architecture

2.2.1 File Structure

The file structure of AutoOptLib is given in Figure 2. As shown in Figure 2, source files of the library are organized in a clear and concise structure. One interface function *AutoOpt.m* and three folders are in the root directory. The folder */Utilities* contains public classes and functions. Specifically, the subfolder */@DESIGN* stores the class and functions for designing algorithms for a target problem, including functions for initializing, searching, and evaluating algorithms. */@SOLVE* contains the class and functions for solving the target problem by the designed algorithms, e.g., functions for

[1] Since Equation 2 is a black-box problem, it is often solved in an iterative manner.

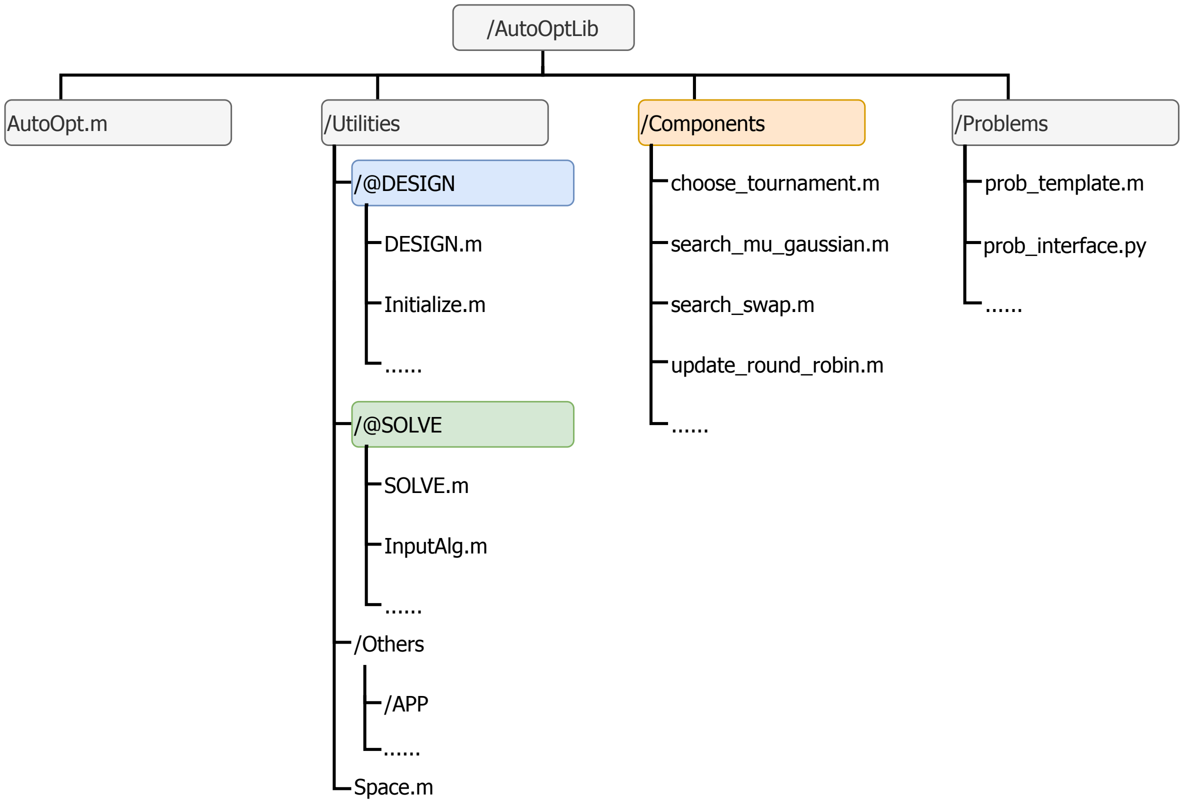

Figure 2: File structure of AutoOptLib.

inputting algorithms, executing the algorithms, and repairing solutions. */Others* involves miscellaneous functions, e.g., sources of the GUI, functions of experimental tools, etc. The *Space.m* function is for constructing the design space by algorithm components.

The */Components* folder contains the algorithm components for constructing the design space. We package each component with ranges of its endogenous parameter values in a single *.m* file. For example, in Figure 2, *choose_tournament.m* and *search_mu_gaussian.m* are the functions of the tournament selection [ES+03] and Gaussian mutation [Fog98], respectively. All the component functions are written in the same structure, so users can easily implement and add new components to the library according to existing ones.

Finally, the */Problems* folder is for the target problems. A problem template, i.e., the *prob_template.m* in Figure 2, is given to guide users to easily implement and interface their problems with the library. A python interface *prob_interface.py* is also provided.

2.2.2 Classes

We involve two main classes in AutoOptLib, namely `DESIGN` and `SOLVE`, which manage the process of designing algorithms for a target problem and solving the target problem by the designed algorithms, respectively. The class diagram is given in Figure 3. An object of the `DESIGN` class is a designed algorithm with several properties, e.g., `operator` (components that constitute the algorithm), `parameter` (endogenous parameters of the algorithm), and `performance` (performance of the algorithm). The class have some methods to be invoked by the objects. For example, the method `Initialize()` works on initializing the designed algorithms; `Evaluate()` is for evaluating the algorithms' performance according to a design objective.

An object of the `SOLVE` class is a solution to the target problem. It has several properties, including `dec` (decision variables), `obj` (objective value), `con` (constraint violation), etc. The class has several methods for achieving the solutions, such as `InputAlg()` (preprocessing and inputting the designed algorithm) and `RunAlg()` (running the algorithm on the target problem).

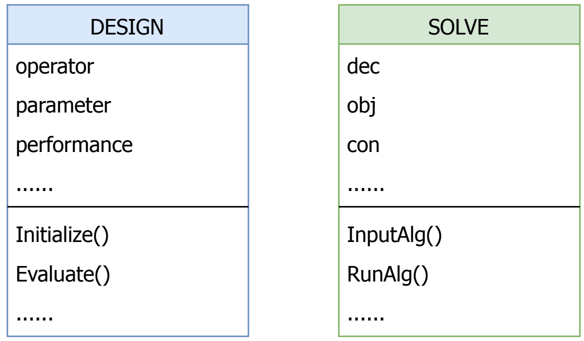

Figure 3: Class diagram of AutoOptLib.

2.2.3 Operating Sequence

AutoOptLib's sequence diagram is depicted in Figure 4. To begin with, the interface function *AutoOpt.m* invokes *DESIGN.m* to instantiate objects (the designed algorithms) of the `DESIGN` class. In detail, firstly, *DESIGN.m* uses the `Initialize()` method to initialize algorithms over the design space. Then, the algorithms' performance on solving the "training" instances [2] of the target problem is evaluated by the `Evaluate()` method. To get the performance, the `Evaluate()` method invokes the `SOLVE` class, and `SOLVE` further calls functions of the algorithms' components and function of the target problem. Finally, the initial algorithms are returned to *AutoOpt.m*.

After initialization, AutoOptLib goes into iterative design. In each iteration, firstly, *AutoOpt.m* invokes *DESIGN.m*. Then, *DESIGN.m* instantiate new objects (new algorithms) of the `DESIGN` class based on the current ones by the `Disturb()` method. Next, the new algorithms' performance is evaluated in the same scheme as in the initialization. After that, the new algorithms are returned to *AutoOpt.m*. Finally, the `Select()` method of the `DESIGN` class is invoked to select promising algorithms from the current and new ones.

After the iteration terminates, *AutoOpt.m* invokes the `Evaluate()` method of the `DESIGN` class to test the final algorithms' performance on the test instances of the target problem. Then, the final algorithms are returned in *AutoOpt.m*.

The above operating sequence has some significant advantages:

1. Metaheuristic component independence - Functions of algorithm components do not interact with each other but invoke independently by the `SOLVE` class. This independence provides great flexibility in designing various algorithms and extensibility to new components.

2. Design technique packaging - The design techniques are packaged in different methods (e.g., `Disturb()`, `Evaluate()`) of the `DESIGN` class. Such packaging brings good understandability and openness to new techniques without modifying the library's architecture.

3. Target problem separation - The targeted problem is enclosed separately and do not directly interact with algorithm components and design techniques. This separation allows users to easily interface their problems with the library and use the library without much knowledge of metaheuristics and design techniques, thereby ensuring the accessibility of the library to researchers and practitioners from different communities.

2.3 Use AutoOptLib

Following the three steps below to use AutoOptLib:

[2]Since the distribution of instances of a real problem is often unknown, one has to sample some of the problem instances and target these instances (training instances) during the algorithm design procedure. To avoid the designed algorithms overfit on the training instances, some other instances (test instances) of the target problem are then employed to test the final algorithms after the design procedure terminates.

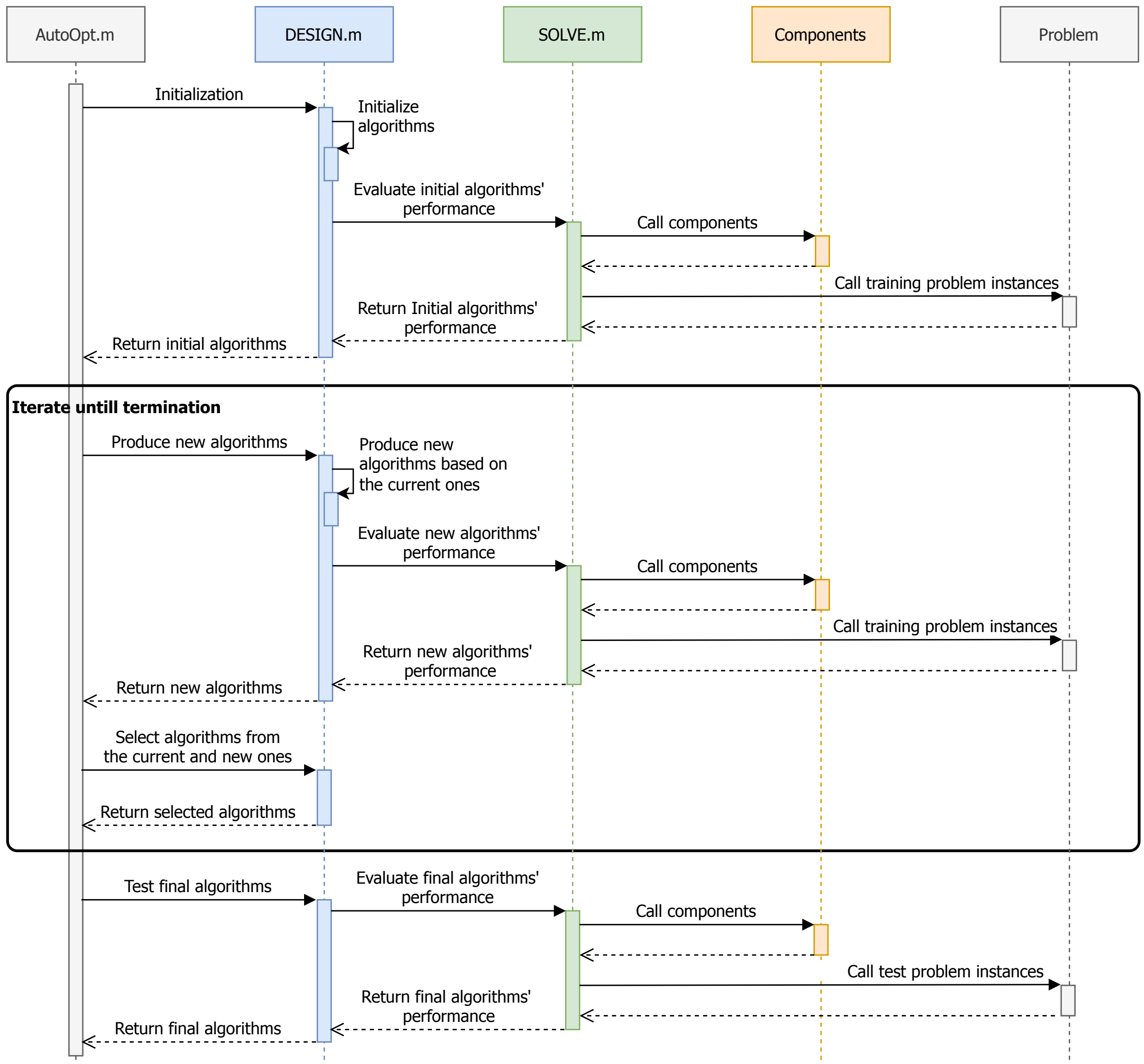

Figure 4: Sequence diagram of AutoOptLib.

2.3.1 Step 1: Implement Problem

AutoOptLib supports implementing the target problem in Matlab/Octave and Python. More formats will be supported in future versions.

Implement in Matlab/Octave:

Users can implement their target optimization problem according to the template *prob_template.m* in the */Problems* folder. *prob_template.m* has three cases. `Case 'construct'` is for setting problem properties and loading the input data. In particular, line 7 defines the problem type, e.g., `Problem.type = {'continuous','static','certain'}` refers to a continuous static problem without uncertainty in the objective function. Lines 10 and 11 define the lower and upper bounds of the solution space. Lines 18 and 21 offer specific settings as indicated in the comments of lines 14-17 and 20, respectively. Line 25 or 26 is for loading the input data. As a result, problem proprieties and data are saved in the `Problem` and `Data` structs, respectively.

```
1 case 'construct' % define problem properties
2     Problem = varargin{1};
3     % define problem type in the following three cells.
4     % first cell : 'continuous'\'discrete'\'permutation'
5     % second cell: 'static'\'sequential'
```

```
    % third cell : 'certain'\'uncertain'
    Problem.type = {'','',''};

    % define the bound of solution space
    lower = []; % 1*D, lower bound of the D-dimension decision space
    upper = []; % 1*D, upper bound of the D-dimension decision space
    Problem.bound = [lower;upper];

    % define specific settings (optional), options:
    % 'dec_diff'          : elements of the solution should be different w.r.t each other for discrete problems
    % 'uncertain_average': averaging the fitness over multiple fitness evaluations for uncertain problems
    % 'uncertain_worst'   : use the worse fitness among multiple fitness evaluations as the fitness for uncertain problems
    Problem.setting = {''}; % put choice(s) into the cell

    % set the number of samples for uncertain problems (optional)
    Problem.sampleN = [];
    output1 = Problem;

    % load/construct data file in the following
    Data = load(''); % for .mat format
    % Data = readmatrix(''); % for other formats
    output2 = Data;
```

`Case 'repair'` is for repairing solutions to keep them feasible, e.g., keeping the solutions within the box constraint. Lines 2 and 3 input the problem data and solutions (decision variables). Programs for repairing solutions should be written from line 5. Finally, the repaired solutions will be returned.

```
1 case 'repair' % repair solutions
2     Data = varargin{1};
3     Decs = varargin{2};
4     % define methods for repairing solutions in the following
5 
6     output1 = Decs;
```

`Case 'evaluate'` is for evaluating solutions' fitness (objective values penalized by constraint violations). In detail, lines 2 and 3 input the problem data and solutions. The target problem's objective function should be written from line 6. Constraint functions (if any) should be written from line 8. For the constrained problems, AutoOptLib follows the common practice of the metaheuristic community, i.e., using constraint violations as penalties to discount infeasible solutions. Constraint violation can be calculated in line 10 by [JD13]:

$$CV(\mathbf{x}) = \sum_{j=1}^{J} \langle \overline{g}_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} |\overline{h}_k(\mathbf{x})|,$$

where $CV(\mathbf{x})$ is the constraint violation of solution $\mathbf{x}$; $\overline{g}_j(\mathbf{x})$ and $\overline{h}_k(\mathbf{x})$ are the jth normalized inequality constraint and kth normalized equality constraint, respectively, in which the normalization can be done by dividing the constraint functions by the constant in this constraint present (i.e., for $g_j(\mathbf{x}) \geq b_j$, the normalized constraint function becomes $\overline{g}_j(\mathbf{x}) = g_j(\mathbf{x})/b_j \geq 0$, and similarly $\overline{h}_k(\mathbf{x})$ can be normalized equality constraint); the bracket operator $\langle \overline{g}_j(\mathbf{x}) \rangle$ returns the negative of $\overline{g}_j(\mathbf{x})$, if $\overline{g}_j(\mathbf{x}) < 0$ and returns zeros, otherwise. During solution evaluation, accessory (intermediate) data for understanding the solutions may be produced. This can be written from line 12. Finally, the objective values, constraint violations, and accessory data will be returned by lines 13-15.

```
1 case 'evaluate' % evaluate solution's fitness
2     Data = varargin{1}; % load problem data
3     Decs = varargin{2}; % load the current solution(s)
4 
5     % define the objective function in the following
```

```

    % define the inequal constraint(s) in the following, equal constraints should be transformed to inequal ones

    % calculate the constraint violation in the following

    % collect accessory data for understanding the solutions in the following (optional)

    output1 = ; % matrix for saving objective function values
    output2 = ; % matrix for saving constraint violation values (optional)
    output3 = ; % matrix or cells for saving accessory data (optional), a solution's accessory data should be saved in a row
```

Examples of problem implementation can be seen in the CEC 2005 benchmark problem files in the */Problems/CEC2005 Benchmarks* folder. The implementation of a real constrained problem *beamforming.m* is given in the */Problems/Real-World/Beanforming* folder.

Implement in Python:

AutoOptLib provides a Python interface *prob_interface.py* in the */Problems* folder to input the target problem from Python files. It contains three methods. The first method `get_type` is for users defining their problems' type (line 5). The second method `get_bound` is for users defining the solution space boundary of their target problem (lines 13 and 14). In the third method `evaluate`, the input `Decs` is the solutions fetched from Matlab/Octave. Users should define the function for evaluating the solutions as `your_evaluate_method` (line 23). The function should have three output variables `obj`, `con`, and `acc` that contain the solutions' objective values, constraint violation values, and accessory data, respectively. `con` and `acc` can be replaced with `_` if not applicable. Returns of the interface *prob_interface.py* will be fetched to the Matlab/Octave problem file *prob_from_py.m*, which will be invoked during algorithm design.

```
1  def get_type():
2      # get problem type
3      # return type = ['continuous'/'discrete'/'permutation', 'static'/'sequential', 'certain'/'uncertain']
4      # e.g.,
5      type = ['continuous', 'static', 'certain']  # a static, continuous problem without uncertainty
6      return type
7
8  def get_bound():
9      # get solution space boundary
10     # shape: [1, D], where D is the dimensionality of solution space, type: 'list'
11     # e.g.,
12     lower = [0, 0, 0, 0, 0]
13     upper = [1, 1, 1, 1, 1]  # a 5D solution space
14     return lower, upper
15
16 def evaluate(Decs, instanceInd):
17     # evaluate solutions
18     # 'Decs' is the solutions fetched from Matlab/Octave, shape: [N, D], where N and D are the number of solutions and the dimensionality of a solution, respectively.
19     # 'instanceInd' is the index of problem instance
20
21     obj, con, acc = your_evaluate_method(Decs, instanceInd)
22     # your_evaluate_method contains your code for evaluating solutions on the current problem instance
23     # 'obj': solutions' objective values, shape: [N, 1] for single-objective optimization, type: 'list'
24     # 'con': solutions' constraint violation values, shape: [N, 1], type: 'list'
25     # 'acc': accessory data, shape: [N, P], one row for one solution' accessory data, type: 'list'
26     # replace 'con' and 'acc' with '_' if not applicable
27     return obj, con, acc
```

2.3.2 Step 2: Define Design Space

AutoOptLib provides over 40 widely-used algorithm components for designing algorithms for continuous, discrete, and permutation problems. Each component is packaged in an independent *.m* file in the */Components* folder. The included components are listed in Table 1.

The default design space for each type of problems covers all the involved components for this type, as shown in Listing 1. Users can either employ the default space with the furthest potential to discover novelty or define a narrow space in *Space.m* in the */Utilities* folder according to interest. For example, when designing algorithms for continuous problems, the candidate Search components can be set by collecting the string of component file name in line 4. More components can be added, which will be detailed in Section 3.1.

Code Listing 1: Design space

```
1  switch Problem(1).type{1}
2      case 'continuous'
3          Choose  = {'choose_traverse';'choose_tournament';'choose_roulette_wheel
               ';'choose_brainstorm';'choose_nich'};
4          Search  = {'search_pso';'search_de_current';'search_de_current_best';'
               search_de_random';'cross_arithmetic';'cross_sim_binary';'
               cross_point_one';'cross_point_two';'cross_point_n';'
               cross_point_uniform';'search_mu_gaussian';'search_mu_cauchy';'
               search_mu_polynomial';'search_mu_uniform';'search_eda';'search_cma';
               reinit_continuous'};
5          Update  = {'update_greedy';'update_round_robin';'update_pairwise';'
               update_always';'update_simulated_annealing'};
6
7      case 'discrete'
8          Choose  = {'choose_traverse';'choose_tournament';'choose_roulette_wheel
               ';'choose_nich'};
9          Search  = {'cross_point_one';'cross_point_two';'cross_point_uniform';'
               cross_point_n';'search_reset_one';'search_reset_rand';'
               reinit_discrete'};
10         Update  = {'update_greedy';'update_round_robin';'update_pairwise';'
               update_always';'update_simulated_annealing'};
11
12     case 'permutation'
13         Choose  = {'choose_traverse';'choose_tournament';'choose_roulette_wheel
               ';'choose_nich'};
14         Search  ={'cross_order_two';'cross_order_n';'search_swap';'
               search_swap_multi';'search_scramble';'search_insert';'
               reinit_permutation'};
15         Update  = {'update_greedy';'update_round_robin';'update_pairwise';'
               update_always';'update_simulated_annealing'};
16 end
```

2.3.3 Step 3: Run AutoOptLib

Users can run AutoOptLib either by Matlab/Octave command or GUI.

Run by Command:

Users can run AutoOptLib by typing the following command in Matlab/Octave command window [3]:

`AutoOpt('name1',value1,'name2',value2,...)`,

where `name` and `value` refer to the input parameter's name and value, respectively. The parameters are introduced in Table 4. In particular, parameters `Metric` and `Evaluate` define the design objective and algorithm performance evaluation method, respectively. They are summized in Tables 2 and 3, respectively.

[3]For Octave, three packages should be loaded via `pkg load communications`, `pkg load statistics`, and `pkg load io` before executing the command.

Parameters `Problem`, `InstanceTrain`, `InstanceTest`, and `Mode` are mandatory to input into the command. For other parameters, users can either use their default values without input to the command or input by themselves for sophisticated functionality. The default parameter values can be seen in *AutoOpt.m*. As an example, `AutoOpt('Mode', 'design', 'Problem', 'beamforming', 'InstanceTrain', [1,2], 'InstanceTest', 3, 'Metric', 'quality'` is for designing algorithms with the best solution quality on the beamforming problem.

There are also conditional parameters when certain options of the main parameters are chosen. For example, setting `Metric` to `runtimeFE` incurs conditional parameter `Thres` to define the algorithm performance threshold for counting the runtime. All conditional parameters have default values and are unnecessary to set in the command.

After AutoOptLib running terminates, results will be saved as follows:

- If running the `design` mode,
  - The designed algorithms' graph representations, phenotypes, parameter values, and performance will be saved as *.mat* table in the root dictionary. Algorithms in the *.mat* table can later be called by the `solve` mode to apply to solve the target problem or make experimental comparisons with other algorithms.
  - A report of the designed algorithms' pseudocode and performance will be saved as *.csv* and *.xlsx* tables, respectively. Users can read, analyze, and compare the algorithms through the report.
  - The convergence curve of the design process (algorithms' performance versus the iteration of design) will be depicted and saved as *.fig* figure. Users can visually analyze the design process and compare different design techniques through the figure.
- If running the `solve` mode,
  - Solutions to the target problem will be saved as *.mat*, *.csv*, and *.xlsx* tables, respectively.
  - Convergence curves of the problem-solving process (solution quality versus algorithm execution) will be plotted in *.fig* figure.

Run by GUI:

The GUI can be invoked by the command `AutoOpt()` without inputting parameters. It is shown in Figure 5. The GUI has three panels, i.e., Design, Solve, and Results:

- The Design panel is for designing algorithms for a target problem. It has two subpanels, i.e., Input Problem and Set Parameters:
  - Users should load the function of their target problem (e.g., *prob_template.m* or *prob_from_py.m* mentioned in Section 2.3.1) and set the indexes of training and test instances in the Input Problem subpanel.
  - Users can set the main and conditional parameters related to the design in the Set Parameters subpanel. All parameters have default values for non-expert users' convenience. The objective of design, the method for comparing the designed algorithms, and the method for evaluating the algorithms can be chosen by the pop-up menus of the Metric, Compare, and Evaluate fields, respectively.

  After setting the problem and parameters, users can start the run by clicking the RUN bottom.
- When the running starts, warnings and corresponding solutions to incorrect uses (if any) will be displayed in the text area at the top of the Results panel. The real-time stage and progress of the run will also be shown in the area. After the run terminates, results will be saved in the same format as done by running by commands. Results will also be displayed on the GUI as follows:
  - The convergence curve of the design process will be plotted in the axes area of the Results panel.
  - The pseudocode of the best algorithm found during the run will be written in the text area below the axes, as shown in Figure 5.

- Users can use the pop-up menu at the bottom of the Results panel to export more results, e.g., other algorithms found during the run, and detailed performance of the algorithms on different problem instances.

- The Solve panel is for solving the target problem by an algorithm. It follows a similar scheme to the Design panel. In particular, users can load an algorithm designed by AutoOptLib in the Algorithm File field to solve the target problem. Alternatively, users can choose a classic algorithm as a comparison baseline through the pop-up menu of the Specify Algorithm field. AutoOptLib now provides 17 classic metaheuristic algorithms in the menu. After the problem-solving terminates, the convergence curve and best solutions will be displayed in the axes and table areas of the Results panel, respectively; detailed results can be exported by the pop-up menu at the bottom.

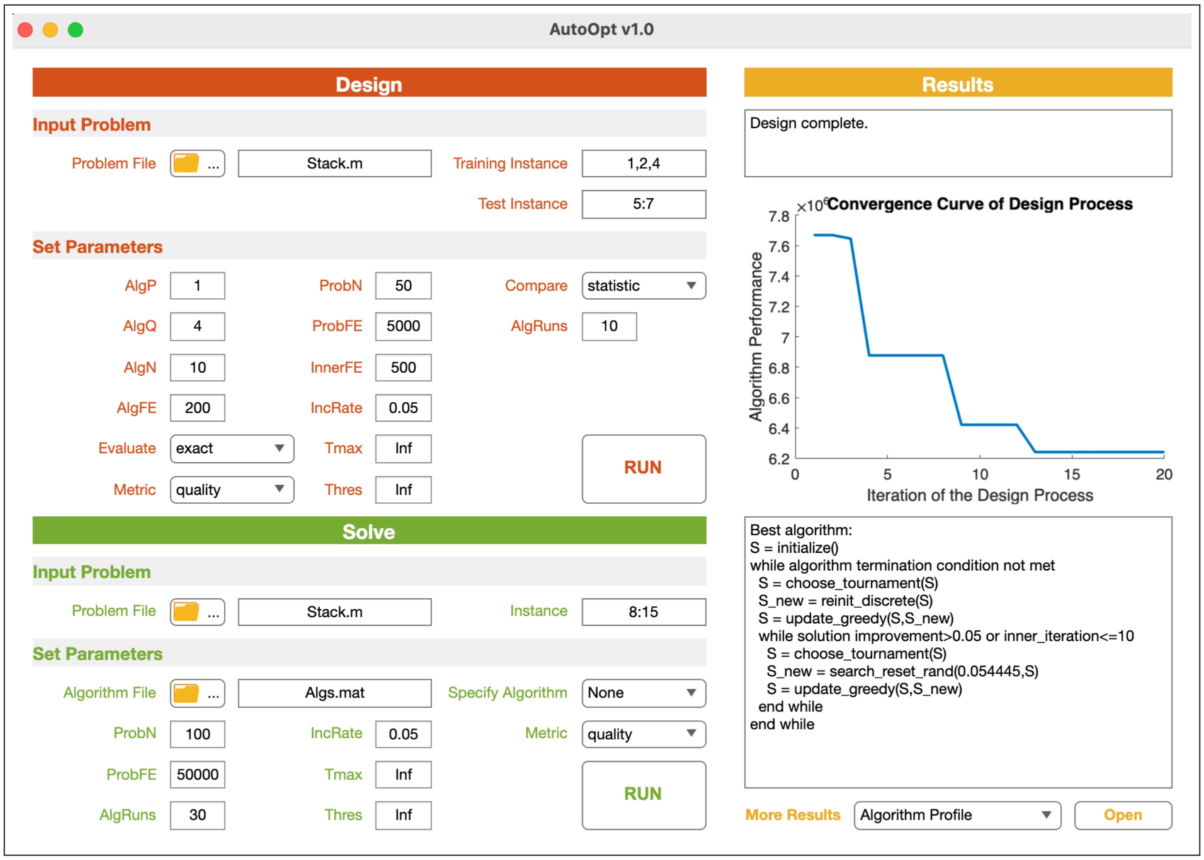

Figure 5: GUI of AutoOptLib.

2.3.4 Other Uses

Beyond the primary use of automatically designing algorithms, AutoOptLib provides functionalities of hyperparameter configuration, parameter importance analysis, and benchmark comparison.

Hyperparameter Configuration:

In many scenarios, users may have a preferred algorithm and only need to configure endogenous parameters (hyperparameters). In essence, hyperparameter configuration is equivalent to designing an algorithm with predefined component composition but unknown endogenous parameter values. Thus, it is easy to perform hyperparameter configuration in AutoOptLib by fixing the component composition and leaving the endogenous parameters tunable in the design space. Following the steps below to conduct hyperparameter configuration:

1. Implement the target problem as illustrated in Section 2.3.1.

2. Define the design space as illustrated in Section 2.3.1. In particular, the design space should only involve the components of the preferred algorithm. The components can be existing ones in the library or user implemented with the same interface as existing ones. Turn `Setting.TunePara` to `true` in *space.m*.

```
    Setting.TunePara = true; % true/false, true for hyperparameter
        configuration
```

3. Run AutoOptLib as illustrated in Section 2.3.3.

Parameter Importance Analysis:

Users can further conduct parameter importance analysis by leaving only one tunable parameter in the design space. That is, users can define the design space with one tunable parameter and fix the component composition and other parameter values. Then, AutoOptLib runs and returns the algorithms found during the design process. These algorithms only differ in the values of the tunable parameter. Users can compare the performance of different parameter values and analyze the impact of parameter changes on the algorithm performance.

Furthermore, by leaving different parameters tunable in different AutoOptLib runs, users can collect and compare the trends of each parameter's changes versus algorithm performance, subsequently getting insight into each parameter's importance to the algorithm performance.

Benchmark Comparison:

As illustrated in Section 2.2, different design objectives (Figure 2) and techniques (Figure 3) in AutoOptLib are implemented with the same architecture; more objectives and techniques can be added by the same interface. This ensures fair and reproducible comparisons among the objectives or techniques. The comparison can be conducted by assigning different objectives or techniques to different AutoOptLib runs on the same targeted problem instances.

AutoOptLib can also be used to benchmark comparisons among different algorithms. Users can set `AlgN`> 1 in the running command or GUI, such that AutoOptLib will design multiple algorithms in a single run. These algorithms are built and evaluated in a uniform manner during the design process. Such uniformity ensures a fair comparison among the algorithms.

3 Developer Guide

3.1 Extend AutoOptLib

AutoOptLib follows the open-closed principle [Mey97, Lar01]. As illustrated in Figure 2 and Section 2.2.3, metaheuristic algorithm components and design techniques are packaged and invoked independently. This allows users easily implement their algorithm components, design objectives, and algorithm performance evaluation techniques based on the current sources, and add the implementations to the library by a uniform interface. Taking Listing 2 as an example, new algorithm components can be added as follows.

Code Listing 2: Implementation of the uniform mutation operator.

```
1  function [output1,output2] = search_mu_uniform(varargin)
2  mode = varargin{end};
3  switch mode
4      case 'execute'
5          Parent  = varargin{1};
6          Problem = varargin{2};
7          Para    = varargin{3};
8          Aux     = varargin{4};
9          if ~isnumeric(Parent)
10             Offspring = Parent.decs;
11         else
12             Offspring = Parent;
13         end
14         Prob  = Para;
15         [N,D] = size(Offspring);
16         Lower = Problem.bound(1,:);
17         Upper = Problem.bound(2,:);
18         ind = rand(N,D) < Prob;
19         Temp = unifrnd(repmat(Lower,N,1),repmat(Upper,N,1));
20         Offspring(ind) = Temp(ind);
21         output1 = Offspring;
22         output2 = Aux;
23     case 'parameter'
24         output1 = [0,0.3]; % mutation probability
25     case 'behavior'
26         output1 = {'LS','small';'GS','large'}; % small probabilities perform
               local search
27 end
28 if ~exist('output1','var')
29     output1 = [];
30 end
31 if ~exist('output2','var')
32     output2 = [];
33 end
```

An algorithm component is implemented in an independent `function` with three main cases. Case `execute` refers to executing the component. There are seven optional inputs:

1. Current solutions, i.e., `Parent` in line 5.
2. The problem proprieties, i.e., `Problem` in line 6.
3. The component's inner parameters, i.e., `Para` in line 7.
4. An auxiliary structure array for saving the component's inner parameters that are changed during iteration (e.g., the velocity in particle swarm optimization (PSO)), i.e., `Aux` in line 8.
5. The algorithm's generation counter `G`.
6. The algorithm's inner local search iteration counter `innerG`.

7. The target problem's input data `Data`.

The component should be implemented from line 9. The outputs of lines 21 and 22 are mandatory, in which `output1` returns solutions processed by the component, and `output2` returns the `Aux` structure array. If the component has inner parameters that are changed during iteration, `Aux` is updated (e.g., update PSO's velocity and save it in `Aux`); otherwise, `Aux` will be the same as that in line 8.

`Case 'parameter'` defines the lower and upper bounds of the component's inner parameter values. For example, the mutation probability is bounded within $[0, 0.3]$ in line 24. For components with multiple inner parameters, each parameter's lower and upper bounds should be saved in an independent row of the matrix `output1`.

For search operators (components) with inner parameters controlling the search behavior, case `behavior` defines how the inner parameters control the search behavior. For example, line 26 indicates that the uniform mutation with smaller mutation probabilities performs local search and that with larger probabilities performs global search. For other operators, `output1` in case `behavior` is left empty, i.e., `output1={};`.

4 Use Case

This section introduces a use case from the communication research community, which illustrates how to use AutoOptLib and how AutoOptLib benefit researchers and practitioners who have complicated optimization problem-solving demands but without the expertise to distinguish and manage suitable optimizers among various choices.

4.1 Problem Description

Reconfigurable intelligent surface (RIS) is an emerging technology to achieve cost-effective communications [MJ19, YLJ+21]. It is a planar passive radio structure with a number of reconfigurable passive elements. Each element can independently adjust the phase shift on the incident signal. Consequently, these elements collaboratively yield a directional beam to enhance the quality of the received signal. The active beamforming at the base station (BS) and RIS should be jointly considered to customize the propagation environment.

The targeted problem considers a RIS-aided downlink multi-user multiple-input single-output (MU-MISO) system, in which a BS equipped with multiple antennas transmits signals to K single-antenna users, as shown in Figure 6. Decision variables of the problem are continual active beamforming of BS and discrete phase shifts of RIS. The objective is to maximize the sum rate of all users, subjecting to the transmit power constraint. The problem is formulated as [YZL+22]:

$$\max_{,} \quad \sum_{k=1}^{K} \log_2(1+\gamma_k), \tag{3a}$$

$$s.t. \quad \theta_n = \beta_n e^{j\phi_n}, \tag{3b}$$

$$\phi_n = \frac{\tau_n 2\pi}{2^b}, \tau_n \in \{0, ..., 2^b - 1\}, \tag{3c}$$

$$\sum_{k=1}^{K} \|\mathbf{w}_k\|^2 \leq P_T, \tag{3d}$$

where $= [_1, ..., _K] \in^{M\times K}$ is the active beamforming at BS; $= diag(\theta_1, ..., \theta_n, ..., \theta_N)$ is the diagonal phase-shift matrix of RIS; γ is the signal-to-interference-plus-noise ratio of user k; β_n and ϕ_n stand for the reflection coefficients and phase shift of element n, respectively; and (3d) indicates that the transmit power is not larger than P_T.

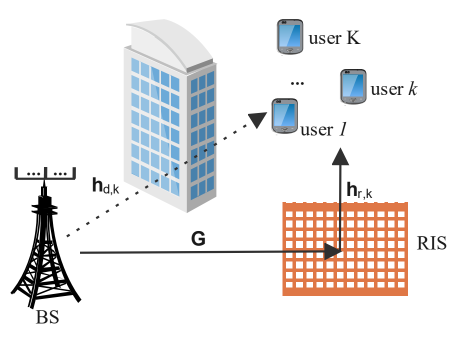

Figure 6: A downlink MU-MISO system with an RIS.

This problem is a non-convex mixed integer problem, which is generally NP-hard. Furthermore, the fitness landscape analysis in [YZL+22] revealed that the problem has a severe unstructured and rugged landscape, especially in cases with a large-sized RIS. While water-filling solutions [YRBC04] can settle BS beamforming, the discrete phase shifts of RIS are non-trivial because the reconfigurable passive elements couple with each other. Existing solvers that decouple the elements and estimate each phase shift separately have been demonstrated to be ineligible [YZL+22]. Metaheuristics' global search ability is promising in handling such unstructured, rugged and highly-coupled problem. AutoOptLib could benefit the problem researchers to quickly get access to an efficient metaheuristic solver from the variety of choices.

4.2 Use AutoOptLib to the Problem

AutoOptLib is used according to the three steps detailed in Sections 2.3.1, 2.3.2, and 2.3.3, respectively:

1. Implement problem: The target problem is implemented according to the template *prob_template.m*. In particular, in line 35, `get_sum_rate` is the user's method for calculating solutions' objective values (constraint violations have been involved in the objective values).

```
1  switch varargin{end}
2      case 'construct' % define problem properties
3          Problem  = varargin{1};
4          instance = varargin{2};
5  
6          orgData  = load('Beanforming.mat','Data');
7          Data = orgData.Data((instance));
8          for i = 1:length(instance)
9              D = size(Data(i).G,1);
10             phases_cnt = 2^Data(i).b-1;
11             lower = zeros(1,D); % 1*D, lower bound of the D-dimension decision space
12             upper = repmat(phases_cnt,1,D); % 1*D, upper bound of the D-dimension decision space
13             Problem(i).type = {'discrete','static','certain'};
14             Problem(i).bound = [lower;upper];
15         end
16 
17         output1 = Problem;
18         output2 = Data;
19 
20     case 'repair' % repair solutions
21         Decs = varargin{2};
22         output1 = Decs;
23 
24     case 'evaluate' % evaluate solution's fitness
25         Data = varargin{1}; % load problem data
26         m    = varargin{2}; % load the current solution(s)
27 
28         b     = Data.b;
29         PT    = Data.PT;
30         G     = Data.G;
31         Hd    = Data.Hd;
32         Hr    = Data.Hr;
33         omega = Data.omega;
34 
35         sR = get_sum_rate(m,b,Hd, Hr,G,PT,omega); % calculate objective value
36         sR = sR'; % N*1
37 
38         output1 = sR; % matrix for saving objective function values
39 end
40 
41 if ~exist('output2','var')
42     output2 = [];
43 end
44 if ~exist('output3','var')
45     output3 = [];
46 end
47 end
```

2. Define design space: AutoOptLib's default design space (with all components for discrete problems, as shown in Figure 1 and Listing 1, respectively) is utilized.

Algorithm 1 Pseudocode of Alg*

S = `initialize`() // initialize solution set S
while stopping criterion not met **do**
  S = `choose_nich`(S)
  S_{new} = `cross_point_uniform`$(0.1229, S)$
  S_{new} = `search_reset_one`(S_{new})
  S = `update_round_robin`(S, S_{new})
end while

3. Run AutoOptLib: AutoOptLib is executed by the command `AutoOpt('Mode', 'design', 'Problem', 'beamforming', 'InstanceTrain', [1:5], 'InstanceTest', [6:10], 'Metric', 'quality')`, where 10 problem instances with different numbers of RIS elements are considered; five of the instances are chosen as training instances; another five are for test; solution quality is set as the design objective (metric); other settings are kept default.

4.3 Results and Analysis

After AutoOptLib running terminates, the best algorithm (termed Alg*) found during design is verified by the test instances. Alg*'s pseudocode is shown in Algorithm 1. Interestingly, the niching mechanism `choose_nich` is involved, which restricts the following uniform crossover (`cross_point_uniform` with crossover rate of 0.1229) to be performed between solutions within a niching area. The reset operation (`search_reset_one` that resets one entity of the solution) further exploits the niching area. Finally, the round-robin selection (`update_round_robin`) maintains diversity by probably selecting inferior solutions. All these designs indicate that maintaining solution diversity may be necessary for escaping local optima and exploring the unstructured and rugged landscape.

To investigate the designed algorithm's efficiency, the algorithm is compared with baselines, i.e., random beamforming, sequential beamforming [DZS+20][4], and three classic metaheuristic solvers, i.e., discrete genetic algorithm (GA), iterative local search (ILS), and simulated annealing (SA)[5]. The algorithms were executed through the Solve mode of AutoOptLib on the five test instances for experimental comparison. All the metaheuristic algorithms conducted population-based search with a population size of 50 for a fair comparison. All algorithms terminated after 50000 function evaluations.

The algorithms' performance is summarized in Table 5. The performance is measured by final solutions' fitness (reciprocal of the quality of service of all users). From Table 5, sequential beamforming is inferior to most of the metaheuristic solvers. This result confirms the ineligibility of decoupling RIS elements and the need for global metaheuristic search. Among the metaheuristic solvers, Alg* outperforms others, especially in instances with large numbers of RIS elements (induce high-dimensional rugged landscape). This performance can be attributed to its diversity maintenance ability. All the above demonstrates the efficiency of AutoOptLib's automated design techniques on the problem.

[4] Sequential beamforming refers to exhaustively enumerating the phase shift of each element one-by-one on the basis of random initial RIS phase shifts.

[5] The discrete GA is consisted by tournament mating selection, uniform crossover, random mutation, and round-robin environmental selection; the crossover and mutation rates are both predefined to 0.2. The ILS and SA perform neighborhood search by randomly resetting one entity of the solution at each iteration.

References

[BK09] Mauro Birattari and Janusz Kacprzyk. *Tuning metaheuristics: A machine learning perspective*. Springer, New York, NY, USA, 2009.

[DZS+20] Boya Di, Hongliang Zhang, Lingyang Song, Yonghui Li, Zhu Han, and H Vincent Poor. Hybrid beamforming for reconfigurable intelligent surface based multi-user communications: Achievable rates with limited discrete phase shifts. *IEEE Journal on Selected Areas in Communications*, 38(8):1809–1822, Aug. 2020.

[ES+03] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, New York, NY, USA, 2003.

[Fog98] David B Fogel. *Artificial intelligence through simulated evolution.* Wiley-IEEE Press, 1998.

[HHLBS09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, Oct. 2009.

[JD13] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, Aug. 2013.

[Lar01] Craig Larman. Protected variation: The importance of being closed. *IEEE Software*, 18(3):89–91, 2001.

[LIDLC+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, Sept. 2016.

[Mey97] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[MJ19] Deepak Mishra and Håkan Johansson. Channel estimation and low-complexity beamforming design for passive intelligent surface assisted miso wireless energy transfer. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4659–4663, Brighton, UK, 2019. IEEE.

[YDWB22] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation*, 26(6):1526–1538, Dec. 2022.

[YLJ+21] Jie Yuan, Ying-Chang Liang, Jingon Joung, Gang Feng, and Erik G Larsson. Intelligent reflecting surface-assisted cognitive radio system. *IEEE Transactions on Communications*, 69(1):675–687, Jan. 2021.

[YRBC04] Wei Yu, Wonjong Rhee, Stephen Boyd, and John M Cioffi. Iterative water-filling for gaussian vector multiple-access channels. *IEEE Transactions on Information Theory*, 50(1):145–152, Jan. 2004.

[YZL+22] Bai Yan, Qi Zhao, Mengke Li, Jin Zhang, J Andrew Zhang, and Xin Yao. Fitness landscape analysis and niching genetic approach for hybrid beamforming in ris-aided communications. *Applied Soft Computing*, 131:109725, Dec. 2022.

Table 1: Metaheuristic algorithm components provided in AutoOptLib.

Component	Description
Choose where to search from:	
choose_roulette_wheel	Roulette wheel selection
choose_tournament	K-tournament selection
choose_traverse	Choose each of the current solutions to search from
choose_cluster	Brain storm optimization's idea picking up for choosing solutions to search from
choose_nich	Adaptive niching based on the nearest-better clustering
Discrete search:	
search_reset_one	Reset a randomly selected entity to a random value
search_reset_rand	Reset each entity to a random value with a probability
search_reset_creep	Add a small positive or negative value to each entity with a probability, for ordinal problems
search_cross_point_one	One-point crossover
search_cross_point_two	Two-point crossover
search_cross_point_n	n-point crossover
search_cross_uniform	Uniform crossover
reinit_discrete	Random reinitialization for discrete problems
Permutation search:	
search_swap	Swap two randomly selected entities
search_swap_multi	Swap each pair of entities between two randomly selected indices
search_scramble	Scramble all the entities between two randomly selected indices
search_insert	Randomly select two entities, insert the second entity to the position following the first one
search_cross_order_two	Two-order crossover
search_cross_order_n	n-order crossover
reinit_permutation	Random reinitialization for permutation problems
Continuous search:	
search_cross_arithmetic	Whole arithmetic crossover
search_cross_sim_binary	Simulated binary crossover
search_cross_point_one	One-point crossover
search_cross_point_two	Two-point crossover
search_cross_point_n	n-point crossover
search_cross_uniform	Uniform crossover
search_cma	The evolution strategy with covariance matrix adaption
search_eda	The estimation of distribution
search_mu_cauchy	Cauchy mutation
search_mu_gaussian	Gaussian mutation
search_mu_polynomial	Polynomial mutation
search_mu_uniform	Uniform mutation
search_pso	Particle swarm optimization's particle fly and update
search_de_random	The "random/1" differential mutation
search_de_current	The "current/1" differential mutation
search_de_current_best	The "current-to-best/1" differential mutation
reinit_continuous	Random reinitialization for continuous problems
Select promising solutions:	
update_always	Always select new solutions
update_greedy	Select the best solutions
update_pairwise	Select the better solution from each pair of old and new solutions
update_round_robin	Select solutions by round-robin tournament
update_simulated_annealing	Simulated annealing's update mechanism, i.e., accept worse solution with a probability
Archive:	
archive_best	Collect the best solutions found so far
archive_diversity	Collect most diversified solutions found so far
archive_tabu	The tabu list

Table 2: Design objectives involved in AutoOptLib.

Objective	Description
`quality`	The designed algorithm's solution quality on the target problem within a fixed computational budget.
`runtimeFE`	The designed algorithm's running time (number of function evaluations) till reaching a performance threshold on solving the target problem.
`runtimeSec`	The designed algorithm's running time (wall clock time, in second) till reaching a performance threshold on solving the target problem.
`auc`	The area under the curve (AUC) of empirical cumulative distribution function of running time, measuring the anytime performance [YDWB22].

Table 3: Algorithm performance evaluation methods provided in AutoOptLib.

Method	Description
`exact`	Exactly run all the designed algorithms on all test problem instances.
`approximate`	Use low complexity surrogate to approximate the designed algorithms' performance without full evaluation.
`racing` [LIDLC+16]	Save algorithm evaluations by stopping evaluating on the next instance if performance is statistically worse than at least another algorithm.
`intensification` [HHLBS09]	Save algorithm evaluations by stopping evaluating on the next instance if performance is worse than the incumbent.

Table 4: Parameters in the commands for running AutoOptLib.

Parameter	Type	Description
Parameters related to the target problem:		
`Problem`	character string	Function of the target problem
`InstanceTrain`	positive integer	Indexes of training instances of the target problem
`InstanceTest`	positive integer	Indexes of test instances of the target problem
Parameters related to the designed algorithms:		
`Mode`	character string	Run mode. Options: `design` - design algorithms for the target problem, `solve` - solve the target problem by a designed algorithm or an existing algorithm.
`AlgP`	positive integer	Number of search pathways in a designed algorithm
`AlgQ`	positive integer	Maximum number of search operators in a search pathway
`Archive`	character string	Name of the archive(s) that will be used in the designed algorithms
`LSRange`	$[0, 1]$ real number	Range of inner parameter values that make the component perform local search*.
`IncRate`	$[0, 1]$ real number	Minimum rate of solutions' fitness improvement during 3 consecutive iterations
`InnerFE`	positive integer	Maximum number of function evaluations for each call of local search
Parameters controlling the design process:		
`AlgN`	positive integer	Number of algorithms to be designed
`AlgRuns`	positive integer	Number of algorithm runs on each problem instance
`ProbN`	positive integer	Population size of the designed algorithms on the target problem instances
`ProbFE`	positive integer	Number of fitness evaluations of the designed algorithms on the target problem instances
`Metric`	character string	Metric for evaluating algorithms' performance (the objective of design). Options: `quality`, `runtimeFE`, `runtimeSec`, `auc`.
`Evaluate`	character string	Method for evaluating algorithms' performance. Options: `exact`, `intensification`, `racing`, `surrogate`.
`Compare`	character string	Method for comparing the performance of algorithms. Options: `average`, `statistic`
`AlgFE`	positive integer	Maximum number of algorithm evaluations during the design process (termination condition of the design process)
`Tmax`	positive integer	Maximum running time measured by the number of function evaluations or wall clock time
`Thres`	real number	The lowest acceptable performance of the designed algorithms. The performance can be solution quality.
`RacingK`	positive integer	Number of instances evaluated before the first round of racing
`Surro`	real number	Number of exact performance evaluations when using surrogate
Parameters related to solving the target problem:		
`Alg`	character string	Algorithm file name, e.g., Algs

$*$: Some search operators have inner parameters to control performing global or local search. For example, a large mutation probability of the uniform mutation operator indicates a global search, while a small probability indicates a local search over neighborhood region. As an example, in cases with `LSRange`$= 0.2$, the uniform mutation with probability lower than 0.2 is regarded as performing local search, and the probability equals or higher than 0.2 performs global search.

Table 5: Average and standard deviation of performance on the beamforming problem. Best results are in bold.

Algorithm	Number of RIS elements in the problem instances				
	120	160	280	320	400
Alg*	**0.0332**±5.05E-04	**0.0312**±4.84E-04	**0.0281**±1.57E-04	**0.0272**±6.76E-04	**0.0260**±1.11E-04
Random	0.0442±7.94E-04	0.0425±6.56E-04	0.0402±8.30E-04	0.0390±6.67E-04	0.0375±1.88E-04
Sequential	0.0382±6.19E-04	0.0387±6.75E-04	0.0374±4.17E-04	0.0369±4.38E-04	0.0354±8.27E-04
GA	0.0369±3.30E-04	0.0356±1.00E-04	0.0337±4.26E-04	0.0333±1.04E-04	0.0322±6.96E-04
ILS	0.0333±3.74E-04	0.0314±2.49E-04	0.0285±1.19E-04	0.0279±1.82E-04	0.0278±1.15E-04
SA	0.0398±5.59E-04	0.0388±7.75E-04	0.0369±3.27E-04	0.0360±4.18E-04	0.0355±9.50E-04