PlatEMO

Evolutionary Multi-Objective Optimization Platform

User Manual 3.0

BIMK Group

February 1, 2021

 All publications using the platform should acknowledge the use of "PlatEMO" and reference the following literature:

Ye Tian, Ran Cheng, Xingyi Zhang, and Yaochu Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization [educational forum]," IEEE Computational Intelligence Magazine, 2017, 12(4): 73-87.

If you have any comment or suggestion to PlatEMO, please send it to *field910921@gmail.com* (*Dr. Ye Tian*). If you want to add your code to PlatEMO, please send the ready-to-use code and the relevant literature to *field910921@gmail.com* as well. You can obtain the newest version of PlatEMO from https://github.com/BIMK/PlatEMO.

Contents

I. Quick Start

PlatEMO provides a variety of algorithms for solving optimization problems in a black-box manner. To this end, users should define the optimization problem, select an algorithm, and set the parameter values, by means of one of the following ways:

1) Calling the main function with parameters (on MATLAB R2012a or higher):

```
platemo('problem',@SOP_F1,'algorithm',@GA,'Name',Value,…);
```

Then the specified benchmark problem will be solved by the specified algorithm with specified parameter settings, where the result can be displayed, saved, or returned (see *Solving Benchmark Problems* for details).

2) Calling the main function with parameters (on MATLAB R2012a or higher):

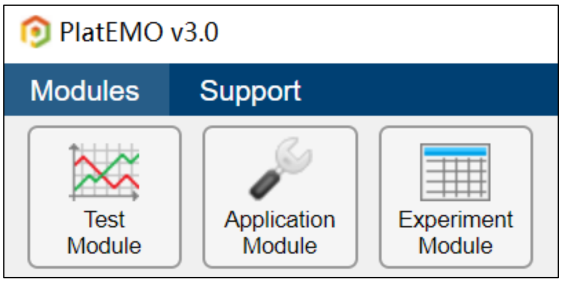
```
f1 = @(x,d)sum(x*d);
f2 = @(x,d)1-sum(x*d);
platemo('objFcn',f1,'conFcn',f2,'algorithm',@GA,…);
```

Then the user-defined problem will be solved by the specified algorithm with specified parameter settings (see *Solving User-Defined Problems* for details).

3) Calling the main function without parameter (on MATLAB R2020b or higher):

```
platemo();
```

Then a GUI with three modules will be displayed, where the test module is used to visually investigate the performance of an algorithm on a benchmark problem (see *Functions of Test Module* for details), the application module is used to solve user-defined problems (see *Functions of Application Module* for details), and the experiment module is used to statistically analyze the performance of multiple algorithms on multiple benchmark problems (see *Functions of Experiment Module* for details).



II. Using PlatEMO without GUI

A. Solving Benchmark Problems

Users can use PlatEMO without GUI by calling the main function `platemo()` with parameters like

```
platemo('Name1',Value1,'Name2',Value2,'Name3',Value3,…);
```

where all the acceptable names and values are

Name	Data type	Default value	Description
`'algorithm'`	Function handle or cell	`dependent`	Class of algorithm
`'problem'`	Function handle or cell	`dependent`	Class of benchmark problem
`'N'`	Positive integer	`100`	Population size
`'M'`	Positive integer	`dependent`	Number of objectives
`'D'`	Positive integer	`dependent`	Number of variables
`'maxFE'`	Positive integer	`10000`	Number of evaluations
`'save'`	Integer	`0`	Number of saved populations
`'outputFcn'`	Function handle	`@ALGORITHM.Output`	Function called before each iteration

- `'algorithm'` denotes the algorithm to be run, whose value should be the function handle of an algorithm, such as `@GA`. The value can also be a cell like `{@GA,p1,p2,…}`, where `p1,p2,…` specify the parameter values of the algorithm.
- `'problem'` denotes the benchmark problem to be solved, whose value should be the function handle of a benchmark problem, such as `@SOP_F1`. The value can also be a cell like `{@SOP_F1,p1,p2,…}`, where `p1,p2,…` specify the parameter values of the benchmark problem.
- `'N'` denotes the population size of the algorithm, which usually equals to the number of solutions in the final population.
- `'M'` denotes the number of objectives of the benchmark problem, which is valid for some multi-objective benchmark problems.
- `'D'` denotes the number of decision variables of the benchmark problem, which is valid for some benchmark problems.
- `'maxFE'` denotes the maximum number of function evaluations, where the algorithm terminates once the number of generated solutions exceeds this value.
- `'save'` denotes the number of saved populations, where the populations are saved to a file if the value is positive and displayed in a figure if the value is zero (see

Collecting the Results for details).

- `'outputFcn'` denotes the function called before each iteration of the algorithm. An output function has two inputs and no output, where the first input is the current `ALGORITHM` object and the second input is the current `PROBLEM` object.

For example, the following code runs the genetic algorithm on the sphere function with a population size of 50, where the populations are displayed in a figure:

```
platemo('algorithm',@GA,'problem',@SOP_F1,'N',50);
```

The following code runs NSGA-II on 5-objective 40-variable DTLZ2 for 20000 function evaluations, where the populations are saved to a file:

```
platemo('algorithm',@NSGAII,'problem',@DTLZ2,'M',5,'D',40,'
maxFE',20000,'save',10);
```

The following code runs MOEA/D with Tchebycheff approach on ZDT1 for ten times, where the populations obtained in each time are saved to a file:

```
for i = 1 : 10
    platemo('algorithm',{@MOEAD,2},'problem',@ZDT1,'save',5);
end
```

Note that users need not specify all the parameters as each of them has a default value.

B. Solving User-Defined Problems

When the parameter `'problem'` is not specified, users can define their own problem by specifying the following parameters:

Name	Data type	Default value	Description
`'encoding'`	char	`'real'`	Encoding scheme
`'objFcn'`	Function handle or cell	`@(x,d)sum(x)`	Objective functions
`'conFcn'`	Function handle or cell	`@(x,d)0`	Constraint functions
`'lower'`	Row vector	`0`	Lower bounds of variables
`'upper'`	Row vector	`1`	Upper bounds of variables
`'initFcn'`	Function handle	`[]`	Function for initializing a population
`'decFcn'`	Function handle	`[]`	Function for repairing invalid solution
`'parameter'`	Cell	`{}`	Dataset

- `'encoding'` denotes the encoding scheme of the problem, whose value can be `'real'` (variables are real or integer numbers), `'binary'` (variables are binary numbers), or `'permutation'` (variables constitute a permutation). Algorithms may use different reproduction operators for different encoding schemes.

- `'objFcn'` denotes the objective functions of the problem, whose value can be a function handle (a single objective) or cell (multiple objectives). An objective function has two inputs and an output, where the first input is a decision vector, the second input is the dataset specified by `'parameter'`, and the output is the objective value. All the objectives are to be minimized.
- `'conFcn'` denotes the constraint functions of the problem, whose value can be a function handle (a single constraint) or cell (multiple constraints). A constraint function has two inputs and an output, where the first input is a decision vector, the second input is the dataset specified by `'parameter'`, and the output is the constraint violation. A constraint is satisfied if and only if the constraint violation is not positive.
- `'lower'` denotes the lower bounds of variables, which is valid when the value of `'encoding'` is `'real'`.
- `'upper'` denotes the upper bounds of variables, which is valid when the value of `'encoding'` is `'real'`.
- `'initFcn'` denotes the function for initializing a population, whose value should be a function handle having two inputs and an output, where the first input is the number of solutions in the population, the second input is the dataset specified by `'parameter'`, and the output is a matrix consisting of the decision vectors in the initial population. This function is called at the beginning of most algorithms.
- `'decFcn'` denotes the function for repairing invalid solution, whose value should be a function handle having two inputs and an output, where the first input is a decision vector, the second input is the dataset specified by `'parameter'`, and the output is the repaired decision vector. This function is called before the objective calculation of each solution.
- `'parameter'` denotes the dataset of the problem, which is used as the second input of the functions specified by `'objFcn'`, `'conFcn'`, `'initFcn'`, and `'decFcn'`.

For example, the following code solves a unimodal problem with 10 variables by differential evolution:

```
platemo('objFcn',@(x,d)sum(x.^2),'lower',zeros(1,10)-10,
'upper',zeros(1,10)+10,'algorithm',@DE);
```

The following code solves a rotated unimodal problem with 10 variables by the default algorithm:

```
platemo('objFcn',@(x,d)sum((x*d).^2),'lower',zeros(1,10)-
10,'upper',zeros(1,10)+10,'parameter',rand(10));
```

The following code solves a constrained bi-objective problem with 20 variables by NSGA-II with a population size of 50:

```
f1 = @(x,d)x(1)*sum(x(2:end));
f2 = @(x,d)sqrt(1-x(1)^2)*sum(x(2:end));
g1 = @(x,d)1-sum(x(2:end));
platemo('objFcn',{f1,f2},'conFcn',g1,'lower',zeros(1,20),'upper',ones(1,20),'algorithm',@NSGAII,'N',50);
```

C. Collecting the Results

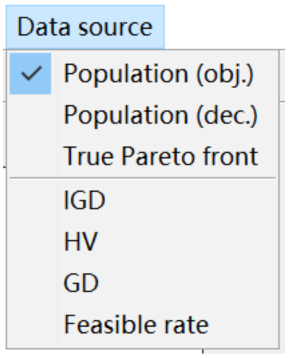
The generated populations can be displayed, saved, or returned after the algorithm terminates. If the main function is called like

```
[Dec,Obj,Con] = platemo(…);
```

Then the final population will be returned, where `Dec` is a matrix consisting of the decision vectors in the final population, `Obj` is a matrix consisting of the objective values in the final population, and `Con` is a matrix consisting of the constraint violations in the final population. If the main function is called like

```
platemo('save',Value,…);
```

Then the generated populations will be displayed in a figure if `Value` is zero (default), where various plots can be displayed by switching the `Data source` menu on the figure.



While if `Value` is positive, the generated populations will be saved to a MAT file named as `PlatEMO\Data\alg\alg_pro_M_D_run.mat`, where `alg` is the algorithm name, `pro` is the problem name, `M` is the number of objectives, `D` is the number of variables, and `run` automatically increases from 1 until the file name does not exist. A file saves a cell `result` consisting of the generated populations and a struct `metric` consisting of the metric values. The whole optimization process of the algorithm is divided into `Value` equal intervals, where the first column of `result` stores the number of consumed function evaluations at the last iteration of each interval, the second column of `result` stores the population at the last iteration of each interval, and `metric` stores the metric values of the stored populations. Note that the above are achieved by the default output function `@ALGORITHM.Output`, while users can collect the results in their own ways by specifying the value of `'outputFcn'` to the handle of a user-defined output function.

```
result =
  6×2 cell array
    {[ 1650]}    {1×50 SOLUTION}
    {[ 3300]}    {1×50 SOLUTION}
    {[ 5000]}    {1×50 SOLUTION}
    {[ 6650]}    {1×50 SOLUTION}
    {[ 8300]}    {1×50 SOLUTION}
    {[10000]}    {1×50 SOLUTION}
```

```
metric =
  struct with fields:

    runtime: 0.3317
        IGD: [6×1 double]
```

Besides, the metric values can be automatically calculated and saved in the experiment module of the GUI. To calculate the metric values manually, users should obtain the optimums of the problem and then call the metric functions, for example,

```
pro = DTLZ2();
IGD(result{end},pro.optimum);
```

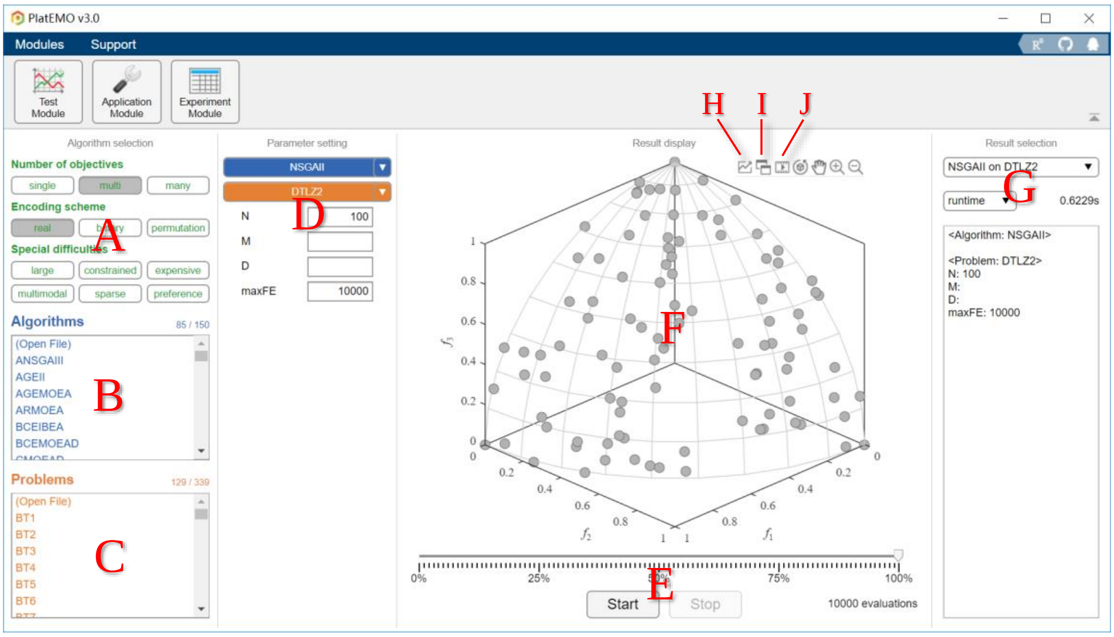
III. Using PlatEMO with GUI

A. Functions of Test Module

Users can use PlatEMO with GUI by calling the main function `platemo()` without parameter like

```
platemo();
```

Then the test module of the GUI will be displayed, which is used to visually investigate the performance of an algorithm on a benchmark problem.

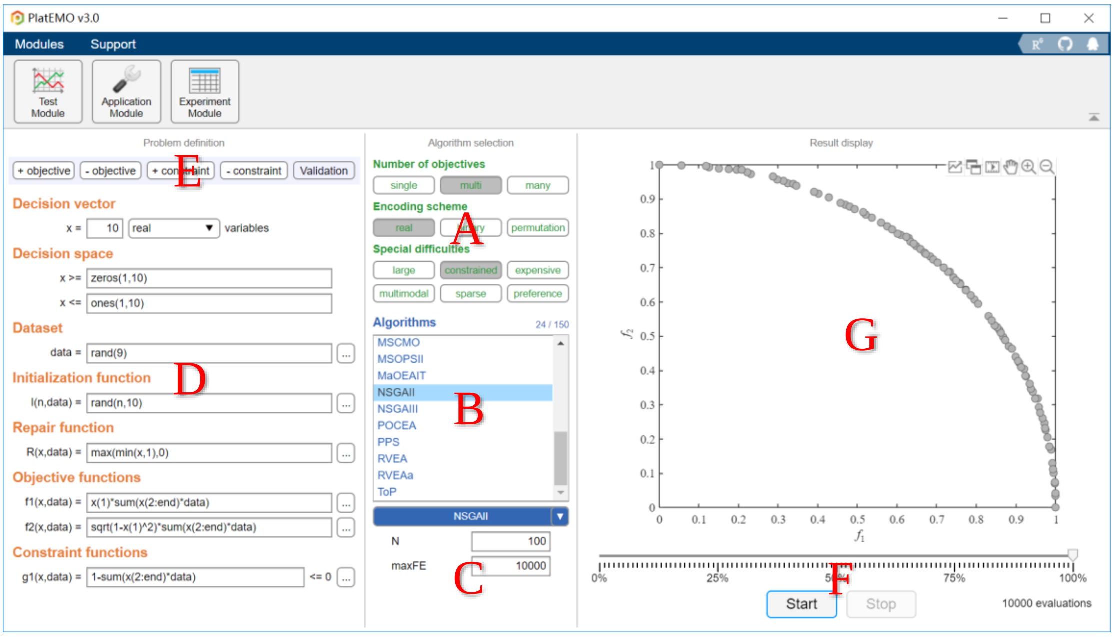


Users should first determine the type of problems in Region A (see *Labels of Algorithms and Problems* for details), select an algorithm in Region B, select a benchmark problem in Region C, and set the parameter values in Region D. Then, the optimization process can be started and controlled in Region E, where the real-time result is displayed in Region F and the historical results can be reviewed in Region G.

Pressing Button H can choose the plot to be displayed, pressing Button I can display the plot in a new figure and save the data in the plot to workspace, and pressing Button J can save the whole optimization process to a GIF file with 20 frames.

B. Functions of Application Module

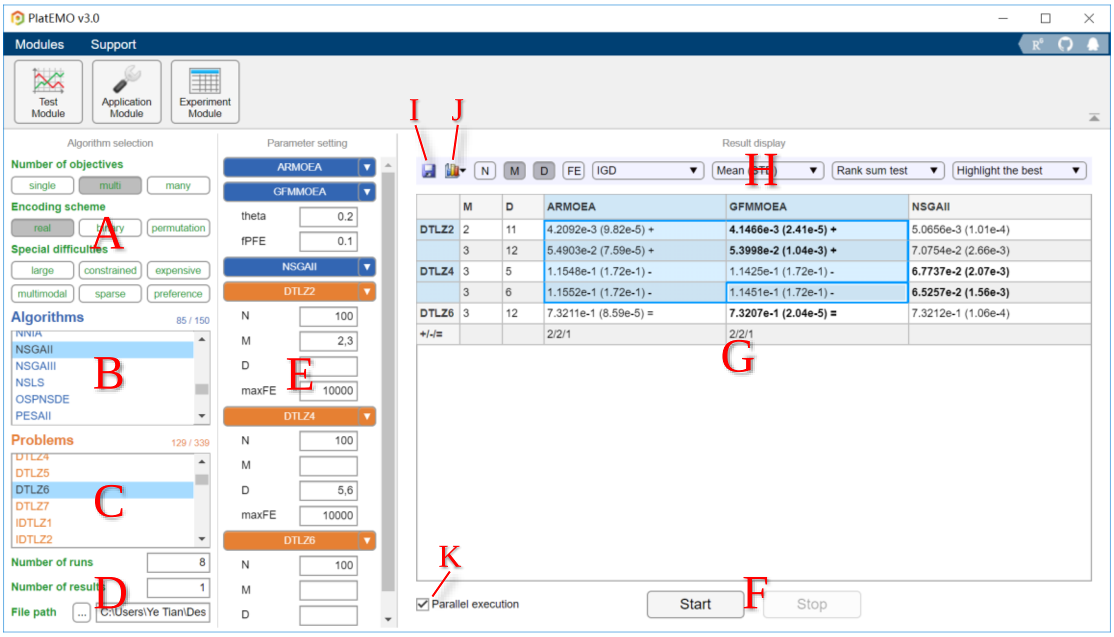
Users can press the menu button to switch to the application module, which is used to solve user-defined problems.



Users should first define the problem in Region D, whose details are the same to those in *Solving User-Defined Problems*. Meanwhile, users can increase or decrease the numbers of objectives and constraints, and check the validity of the problem in Region E. Then, the type of problems can be automatically determined in Region A, while users should select an algorithm in Region B and set the parameter values in Region C. The optimization process can be started and controlled in Region F, and the real-time result is displayed in Region G.

C. Functions of Experiment Module

Users can press the menu button to switch to the experiment module, which is used to statistically analyze the performance of multiple algorithms on multiple problems.



Users should first determine the type of problems in Region A (see *Labels of Algorithms and Problems* for details), select multiple algorithms in Region B, select multiple benchmark problems in Region C, configure the experimental settings in Region D, and set the parameter values in Region E, where the number of objectives `M` and the number of variables `D` can be vectors. Then, the optimization process can be started and controlled in Region F, where the statistical results are listed in Region G.

The statistical results to be listed can be customized in Region H. Pressing Button I can save the table to an Excel, TeX, TXT, or MAT file, and pressing Button J can display the results in the selected cells of the table in a new figure. Button K determines whether the experiment is performed on a single CPU (in sequence) or all the CPUs (in parallel).

All the results are saved to MAT files in the folder specified in Region D. If a result file already exists, the file will be loaded and the algorithm will not be run.

D. Labels of Algorithms and Problems

Each algorithm or benchmark problem is tagged with labels by the comment in the second line of its main function. For example, in the code of `PSO.m`:

```
classdef PSO < ALGORITHM
% <single> <real> <large/none> <constrained/none>
```

which indicates the types of problems that the algorithm can solve. All the labels are

Label	Description
`<single>`	The problem has a single objective
`<multi>`	The problem has two or three objectives
`<many>`	The problem has four or more objectives
`<real>`	The decision variables are real or integer numbers
`<binary>`	The decision variables are binary numbers
`<permutation>`	The decision variables constitute a permutation
`<large>`	The problem has more than 100 decision variables
`<constrained>`	The problem has at least one constraint
`<expensive>`	The objectives are computationally expensive, i.e., only a very limited number of function evaluations are available
`<multimodal>`	There exist multiple optimal solutions with similar objective values but considerably different decision vectors, all of which should be found
`<sparse>`	Most decision variables of the optimal solutions are zero
`<preference>`	Only the optimal solutions in the predefined regions of the Pareto front are expected to be found
`<none>`	Empty label

An algorithm may have multiple sets of labels, where the Cartesian product between all the label sets include all the types of problems that can be solved by the algorithm. If the label sets of an algorithm are `<single> <real> <constrained/none>`, it will be able to solve single-objective continuous optimization problems with or without constraints. On the other hand, the label sets `<single> <real>` mean that the algorithm can only solve unconstrained problems, the label sets `<single> <real> <constrained>` mean that the algorithm can only solve constrained problems, and the label sets `<single> <real/binary>` mean that the algorithm can solve problems with either real variables or binary variables.

Each algorithm or benchmark problem should be tagged with labels, otherwise it will not be appeared in the lists in the GUI. When determining the type of problems in Region A, the algorithms that can solve such type of problems will be appeared in the list in Region B, and the benchmark problems belonging to this type will be appeared in the list in Region C.

IV. Extending PlatEMO

A. ALGORITHM Class

An algorithm should be written as a subclass of `ALGORITHM` and put in the folder `PlatEMO\Algorithms`, which contains the following properties and methods:

Property	**Specified by**	**Description**
`parameter`	Users	Parameters of the algorithm
`save`	Users	Number of populations saved in an execution
`outputFcn`	Users	Function called in `NotTerminated()`
`pro`	`Solve()`	Problem solved in current execution
`result`	`NotTerminated()`	Populations saved in current execution
`metric`	`NotTerminated()`	Metric values of current populations
Method	**Be redefined**	**Description**
`ALGORITHM`	Cannot	Set the properties specified by users
`Solve`	Cannot	Call `alg.Solve(pro)` to solve problem `pro` by algorithm `alg`
`main`	Must	Main procedure of the algorithm
`NotTerminated`	Cannot	Function called before each iteration in `main()`
`ParameterSet`	Cannot	Set the parameter values according to `parameter`

Each algorithm should inherit `ALGORITHM` and redefine the method `main()`. For example, the code of `GA.m` is

```
classdef GA < ALGORITHM
% <single><real/binary/permutation><large/none><constrained/none>
% Genetic algorithm
% proC --- 1 --- Probability of crossover
% disC --- 20 --- Distribution index of crossover
% proM --- 1 --- Expectation of the number of mutated variables
% disM --- 20 --- Distribution index of mutation

%-------------------------- Reference --------------------------
% J. H. Holland, Adaptation in Natural and Artificial Systems,
% MIT Press, 1992.
%---------------------------------------------------------------

    methods
        function main(Alg,Pro)
```

```
16              [proC,disC,proM,disM] = Alg.ParameterSet(1,20,1,20);
17              P = Pro.Initialization();
18              while Alg.NotTerminated(P)
19                  P1 = TournamentSelection(2,Pro.N,FitnessSingle(P));
20                  O = OperatorGA(P(P1),{proC,disC,proM,disM});
21                  P = [P,O];
22                  [~,rank] = sort(FitnessSingle(P));
23                  P = P(rank(1:Pro.N));
24              end
25          end
26      end
```

The functions of each line are as follows:

Line 1:	Inheriting the `ALGORITHM` class;
Line 2:	Tagging the algorithm with labels (see *Labels of Algorithms and Problems* for details);
Line 3:	Full name of the algorithm;
Lines 4-7:	Parameter name --- default value --- description, which are shown in the parameter setting list in the GUI;
Lines 9-12:	Reference of the algorithm;
Line 15:	Redefining the method of main procedure;
Line 16:	Obtaining the parameter values specified by users, where `1,20,1,20` are default values of the four parameters `proC,disC,proM,disM`;
Line 17:	Obtaining an initial population by calling a method of the problem;
Line 18:	Storing the last population and checking whether the number of function evaluations exceeds; if so, the algorithm will terminate immediately;
Line 19:	Binary tournament based mating selection by calling a public function;
Line 20:	Using the mating pool to generate offsprings by calling a public function;
Line 21:	Combing the current population with the offsprings;
Line 22:	Sorting the solutions based on their fitness calculated by a public function;
Line 23:	Retaining the solutions with better fitness for next iteration.

In the above codes, the functions `ParameterSet()` and `NotTerminated()` are provided by the `ALGORITHM` class, and the function `Initialization()` is provided by the `PROBLEM` class. Besides, the functions `TournamentSelection()`, `FitnessSingle()` and `OperatorGA()` are public functions in the folder `PlatEMO\Algorithms\Utility functions`, which provides a number of operations commonly used in algorithms. The following table lists the functions that can be used in algorithms, where the details of them are referred to the comments in their codes:

Function Name	Description
ALGORITHM. NotTerminated	Function called before each iteration of the algorithm
ALGORITHM. ParameterSet	Set the parameter values specified by users
PROBLEM. Initialization	Initialize a population for the problem
CrowdingDistance	Crowding distance calculation for multi-objective optimization
FitnessSingle	Fitness calculation for single-objective optimization
NDSort	Non-dominated sorting
OperatorDE	The reproduction operator of differential evolution
OperatorFEP	The reproduction operator of fast evolutionary programming
OperatorGA	The reproduction operators of genetic algorithm
OperatorGAhalf	The reproduction operators of genetic algorithm, where only the first half of offsprings are generated
OperatorPSO	The reproduction operator of particle swarm optimization
RouletteWheel Selection	Roulette-wheel selection
Tournament Selection	Tournament selection
UniformPoint	Generate a set of uniformly distributed points

B. PROBLEM Class

A benchmark problem should be written as a subclass of `PROBLEM` and put in the folder `PlatEMO\Problems`, which contains the following properties and methods:

Property	Specified by	Description
N	Users	Population size of algorithms
M	Users and Setting()	Number of objectives of the problem
D	Users and Setting()	Number of decision variables of the problem
maxFE	Users	Maximum number of function evaluations
FE	SOLUTION()	Number of function evaluations consumed in current execution
encoding	Setting()	Encoding scheme of the problem
lower	Setting()	Lower bounds of the decision variables
upper	Setting()	Upper bounds of the decision variables
optimum	GetOptimum()	Optimal values of the problem, such as the minimum objective value of single-objective optimization problems and a set of points on the Pareto front of multi-objective optimization problems
PF	GetPF()	Pareto front of the problem, such as a 1-D curve of bi-objective optimization problems, a 2-D surface of tri-objective optimization problems, and feasible regions of constrained optimization problems
parameter	Users	Parameters of the problem

Method	Be redefined	Description
PROBLEM	Cannot	Set the properties specified by users
Setting	Must	Default settings of the problem
Initialization	Can	Initialize a population for the problem
CalDec	Can	Repair invalid solutions in a population
CalObj	Must	Calculate the objective values of solutions in a population. All objectives are to be minimized
CalCon	Can	Calculate the constraint violations of solutions in a population. A constraint is satisfied if and only if the constraint violation is not positive
GetOptimum	Can	Generate the optimal values and store in optimum
GetPF	Can	Generate the Pareto front and store in PF
DrawDec	Can	Display the decision variables of a population
DrawObj	Can	Display the objective values of a population
Current	Cannot	Static method for getting or setting the current PROBLEM object
ParameterSet	Cannot	Set the parameter values according to parameter

Each benchmark problem should inherit PROBLEM and redefine the methods Setting() and CalObj(). For example, the code of SOP_F1.m is

```
classdef SOP_F1 < PROBLEM
% <single><real><expensive/none>
% Sphere function

%------------------------------- Reference --------------------------------
% X. Yao, Y. Liu, and G. Lin, Evolutionary programming made
% faster, IEEE Transactions on Evolutionary Computation, 1999, 3
% (2): 82-102.
%--------------------------------------------------------------------------

    methods
        function Setting(obj)
            obj.M = 1;
            if isempty(obj.D); obj.D = 30; end
            obj.lower = zeros(1,obj.D) - 100;
            obj.upper = zeros(1,obj.D) + 100;
            obj.encoding = 'real';
        end
        function PopObj = CalObj(obj,PopDec)
            PopObj = sum(PopDec.^2,2);
        end
    end
```

The functions of each line are as follows:

Line 1: Inheriting the `PROBLEM` class;

Line 2: Tagging the problem with labels (see *Labels of Algorithms and Problems* for details);

Line 3: Full name of the problem;

Lines 5-9: Reference of the problem;

Line 12: Redefining the method of default parameter settings;

Line 13: Setting the number of objectives;

Line 14: Setting the number of decision variables if it is not specified by users;

Lines 15-16: Setting the lower bounds and upper bounds of decision variables;

Line 17: Setting the encoding scheme of the problem;

Line 19: Redefining the method of calculating objective values;

Line 20: Calculating the objective values of solutions in a population.

The method `Initialization()` randomly initializes a population for the problem. This method can be redefined to specify a novel initialization strategy. For example, `Sparse_NN.m` initializes a population in which half the decision variables are zero:

```
function Population = Initialization(obj,N)
    if nargin < 2; N = obj.N; end
    PopDec = (rand(N,obj.D)-0.5)*2.*randi([0 1],N,obj.D);
    Population = SOLUTION(PopDec);
end
```

The method `CalDec()` repairs invalid solutions in a population, where each decision variable will be set to the boundary values if it is larger than the upper bound or smaller than the lower bound. This method can be redefined to specify a novel repair strategy. For example, `MOKP.m` repairs solutions that exceed the capacity:

```
function PopDec = CalDec(obj,PopDec)
    C = sum(obj.W,2)/2;
    [~,rank] = sort(max(obj.P./obj.W));
    for i = 1 : size(PopDec,1)
        while any(obj.W*PopDec(i,:)'>C)
            k = find(PopDec(i,rank),1);
            PopDec(i,rank(k)) = 0;
        end
    end
end
```

The method `CalCon()` returns zero as the constraint violation of the solutions in a population, i.e., all the solutions are feasible. This method can be redefined to specify constraint functions for the problem. For example, `MW1.m` calculates a constraint for each solution:

```
function PopCon = CalCon(obj,X)
    PopObj = obj.CalObj(X);
    l = sqrt(2)*PopObj(:,2) - sqrt(2)*PopObj(:,1);
    PopCon = sum(PopObj,2) - 1 - 0.5*sin(2*pi*l).^8;
end
```

The method `GetOptimum()` can be redefined to specify the optimal values of the problem. For example, `SOP_F8.m` returns the optimal value of the objective function:

```
function R = GetOptimum(obj,N)
    R = -418.9829*obj.D;
end
```

and `DTLZ2.m` returns a set of uniformly distributed points on the Pareto front:

```
function R = GetOptimum(obj,N)
    R = UniformPoint(N,obj.M);
    R = R./repmat(sqrt(sum(R.^2,2)),1,obj.M);
end
```

The method `GetPF()` can be redefined to specify the Pareto front or feasible regions of the problem for the visualization achieved in `DrawObj()`. For example, `DTLZ2.m` returns the data for plotting the 2-D or 3-D Pareto front:

```
function R = GetPF(obj)
    if obj.M == 2
        R = obj.GetOptimum(100);
    elseif obj.M == 3
        a = linspace(0,pi/2,10)';
        R = {sin(a)*cos(a'),sin(a)*sin(a'),cos(a)*ones(size(a'))};
    else
        R = [];
    end
end
```

and `MW1.m` returns the data for plotting the feasible regions:

```
function R = GetPF(obj)
    [x,y] = meshgrid(linspace(0,1,400),linspace(0,1.5,400));
    z = nan(size(x));
    fes = x+y-1-0.5*sin(2*pi*(sqrt(2)*y-sqrt(2)*x)).^8 <= 0;
    z(fes&0.85*x+y>=1) = 0;
    R = {x,y,z};
end
```

The method `DrawDec()` displays the decision variables of a population, which is used for the visualization of results in the GUI. This method can be redefined to specify a novel visualization method. For example, `TSP.m` displays the route of the best solution:

```
function DrawDec(obj,P)
    [~,best] = min(P.objs);
    Draw(obj.R(P(best).dec([1:end,1]),:),'-k','LineWidth',1.5);
    Draw(obj.R);
end
```

The method `DrawObj()` displays the objective values of a population, which is used for the visualization of results in the GUI. This method can be redefined to specify a novel visualization method. For example, `Sparse_CD.m` adds labels to the axes:

```
function DrawObj(obj,P)
    Draw(P.objs,{'Kernel k-means','Ratio cut',[]});
end
```

where `Draw()` is a function in the folder `PlatEMO\GUI` for displaying data. The details of the above functions are referred to the comments in their codes.

C. SOLUTION Class

A `SOLUTION` object denotes an individual, and an array of `SOLUTION` objects denote a population. The `SOLUTION` class contains the following properties and methods:

Property	**Specified by**	**Description**
`dec`	Users	Decision variables of the solution
`obj`	`SOLUTION()`	Objective values of the solution
`con`	`SOLUTION()`	Constraint violations of the solution
`add`	`adds()`	Additional properties (e.g., velocity) of the solution

Method	**Description**
`SOLUTION`	Receive the decision variables and calculate the objective values and constraint violations of one or more solutions. `PROBLEM.FE` will be automatically increased by the number of `SOLUTION` objects returned
`decs`	Get the matrix of decision variables of multiple solutions
`objs`	Get the matrix of objective values of multiple solutions
`cons`	Get the matrix of constraint violations of multiple solutions
`adds`	Get the matrix of additional properties of multiple solutions
`best`	Get the feasible and best solution for single-objective optimization, or the feasible and non-dominated solutions for multi-objective optimization

For example, the following code generates a population with ten solutions, then gets the objective matrix of the best solutions in the population:

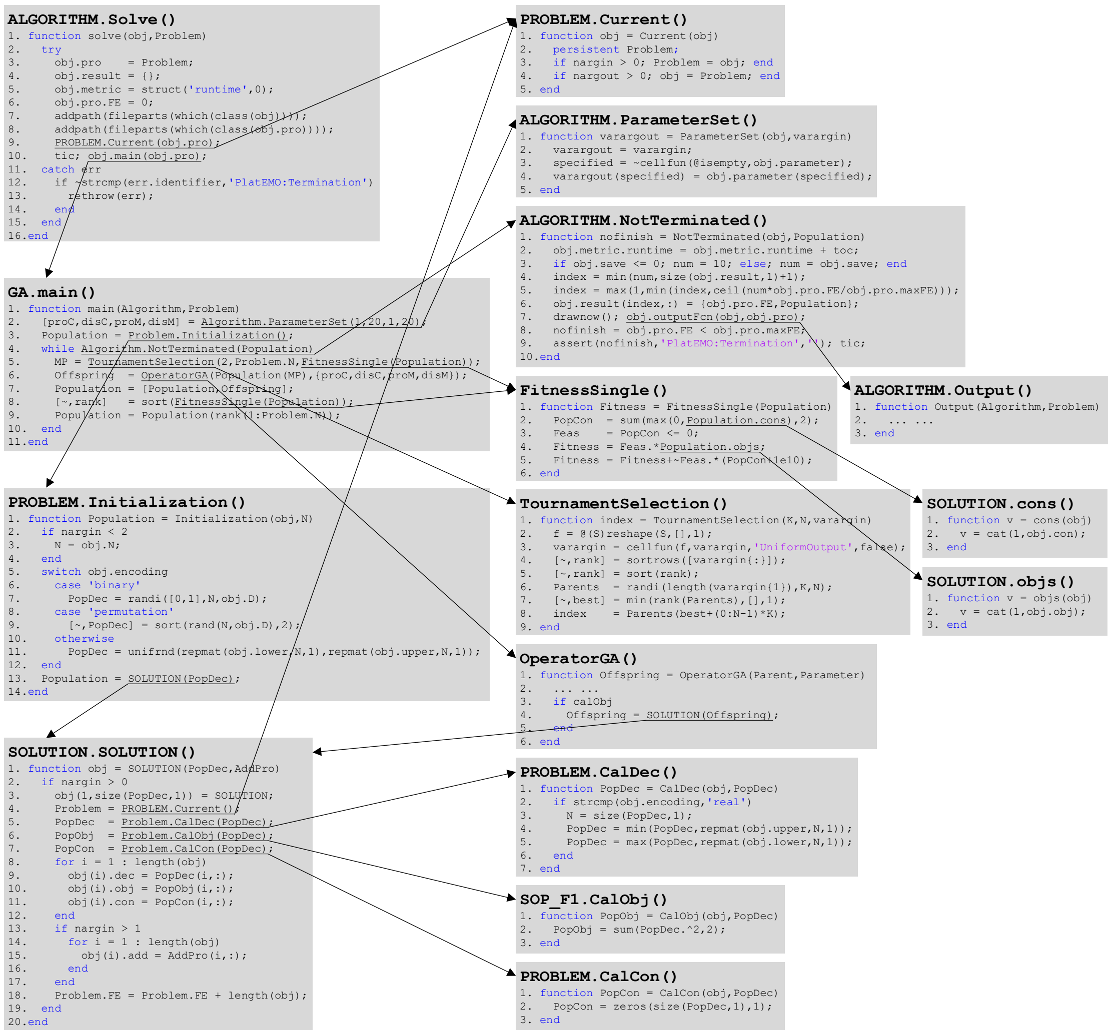
```
Population = SOLUTION(rand(10,5));
BestObjs = Population.best.objs
```

D. Whole Procedure of One Run

The following code uses the genetic algorithm to solve the sphere function:

```
Alg = GA();
Pro = SOP_F1();
Alg.Solve(Pro);
```

where the functions called in the execution of `Alg.Solve(Pro)` are as follows.



E. Metric Function

A metric should be written as a function and put in the folder `PlatEMO\Metrics`. For example, the code of `IGD.m` is

```
1  function score = IGD(Population,PF)
2  % <min>
3  % Inverted generational distance
4
5  %------------------------------- Reference --------------------------------
6  % C. A. Coello Coello and N. C. Cortes, Solving multiobjective
7  % optimization problem using an artificial immune system, Genetic
8  % Programming and Evolvable Machines, 2005, 6(2): 163-190.
9  %------------------------------------------------------------------------------
10
11     PopObj = Population.best.objs;
12     if size(PopObj,2) ~= size(PF,2)
13       score = nan;
14     else
15       score = mean(min(pdist2(PF,PopObj),[],2));
16     end
17 end
```

The functions of each line are as follows:

Line 1: Function declaration, where the first input is a population, the second input is the optimums of a problem, and the output is the metric value;

Line 2: Tagging the metric with `<min>` (the smaller metric value the better) or `<max>` (the larger metric value the better);

Line 3: Full name of the metric;

Lines 5-9: Reference of the metric;

Line 11: Obtaining the feasible and non-dominated solutions in the population;

Lines 12-13: Returns `nan` if there is no feasible and non-dominated solution;

Lines 14-15: Returns the IGD value of the feasible and non-dominated solutions.