The optFUMOLA package

A Simulation-Based Black-Box Optimization Library and Interface

Version 1.0

Documentation

v2017-04-21-14-53

Contents

1. About

1.1 Introduction

Simulations offer valuable insights into mathematically modeled systems, especially by evaluating the effects of different values of model input parameters on the system behavior. A special case is the determination of optimal values for model input parameters with regards to a defined system performance measure, the so-called objective function. With the number of possible input parameter values being huge or even infinite, it becomes unfeasible to perform this search by hand or by brute-force. This is were mathematical optimization comes into play and offers advanced methods to perform this search more efficiently.

Within this context, simulation-based black-box optimization applies mathematical optimization techniques solely based on the analysis of simulation results, without further knowledge of the system under consideration. The corresponding problems belong to the class of non-linear non-convex derivative-free multi-modal objective functions. Typically, such an approach is necessary whenever there is no adequate closed analytical representation of the model available, for instance in the case of co-simulation environments or closed-source domain-specific simulators. To make matters more complicated, for many applications that require to resort to simulation-based black-box approaches, the evaluation of the objective function is computationally expansive (minutes to hours per evaluation), due to the complexity of the simulation model.

optFUMOLA[1] is an open-source optimization library and interface designed for simulation-based black-box optimization. It provides generic yet versatile interfaces between simulation tools and optimization algorithms, simplifies procedures related to handling simulation input and output, and facilitates parallel execution of optimization routines. Furthermore, the approach is general enough to enable both single- and multi-objective optimization.

1.2 Implementation

MATLAB is used to implement the interface and library since many suitable optimization algorithms are available in MATLAB.

[1] optFUMOLA including manual and code documentation, available at:
https://github.com/benpesen/optFUMOLA

1.3 Compatibility

To be compatible with your simulation tool and model, the following points have to be fulfilled:

- Model input parameters can be specified to the simulation tool with the help of files or input arguments.
- Simulations can be invoked through command line or an equivalent method.
- Simulation results are stored in files MATLAB that are readable (parseable) for MATLAB.

2. Setup

The setup of a simulation-based optimization problem with optFUMOLA can be divided into steps

Setup Start by implementing the abstract base classes for your specific problem.
Optimization Define the optimization problem and run the optimization.

2.1 Setup

2.1.1 Implementation of ObjFUNbase.m

To implement a subclass of `ObjFUNbase`, it is best to copy the template file `ObjFUNtemplate.m` and manipulate it according to the following instructions.

Constructor

The first part that has to be modified is the constructor of the class:

Listing 2.1: Definition of the constructor.

```
function obj = ObjFUN<your_tool>(objrun)
 output_dir_base = '<path>';
 retry_timeout = 2;
 n_retry_max = 3;
 dry_run = 0;
 obj@ObjFUNbase( objrun, output_dir_base, n_retry_max,
    retry_timeout, dry_run );
end
```

The only input of the function is the object of class `RunTASKSbase` as it is used withing this class. The object of class `ObjFUNbase` is the output of the constructor. To specify your problem, `<your_tool>` should be replaced by your tool name and has to match the file name. You can then set the timeout for errors in file reading and opening `retry_timeout` in seconds and the maximum number of such errors `n_retry_max`. The last setting is for the case were results for the optimization run are already available due to a previous run with the exact same settings.

When setting dry_run to a non-zero value, the simulations are not invoked but the result files are analyzed and the optimization algorithm is run with these values. Keep in mind that also the same seed for the random number generator must be used in this case.

Setup simulation task

In the next step, a task for the simulation runs has to be defined in a generic way and is done in the function setup_single_task.

Listing 2.2: Definition of function to setup a single tasks.

```
function sim_task = setup_single_task( obj , output_dir ,
   task_iter , x )
 % Set result file name for current task.
 task_result_file_name = [ 'result', num2str(task_iter), '.
    csv' ];

 el_heatpump_load_pon = x(1);
 el_store_c = x(2);

 % convert parameters to strings
 str_el_heatpump_load_pon = num2str( el_heatpump_load_pon );
 str_el_store_c = num2str( el_store_c );

 %define command to run simulation
 sim_task = [ 'python.exe E:\DeMat_HybridEnergySystem\
    runFumola.py ', working_dir , ' ', output_dir , ' ',
    resources_dir , ' ', batch_file , ' ', model_file , ' ',
    task_optional_fumola_args ];
 sim_task = cellstr(sim_task);

end
```

The first function argument is the object of the class ObjFUNbase. Also the output directory is passed to the function to allow to define an output directory for each iteration and save the result files there. The indented folder structure is outputdirbase/outputdir/resultfilename. Here output_dir_base is set in the constructor. This defined structure can the be used to access and find the result files later on. task_iter is the iteration number of the corresponding task in this algorithm iteration. It can be used to distinctly identify the task and enable unique result filename. The last and most important function argument is the array that contains the trial values for the design parameter provided by the optimization algorithm. The return value of this method has to be a cell array that contains the description of the simulation task. This task will later be executed by the method run_tasks of class RunTASKSbase. The most important thing that has to be specified here by the user is how the design variables are passed to the simulation tool. This might be possible through command line arguments or by writing them to files. It is also very important that the result files of the simulation tool can later be found and accessed. This is why it is encouraged to also pass the output directory and a unique filename withing this directory. The structure can be in the form of output_dirbase/output_dir/result_file_name.

Define objective function

The last and most crucial step is to retrieve the simulation results and compute a objective function out of it.

Listing 2.3: Definition of function to retrieve results from a single task and objective function definition.

```
% Retrieve results from simulation task and define objective function
function sim_result = result_single_task( obj , output_dir , task_iter)

 % Set result file name for current task (same as in setup_single_task)
 output_file_name = [ output_dir , '\result', num2str( task_iter), '.csv' ];

 % check if result file is ready
 error = check_result_file( obj , output_file_name );

 % read data from result file
 try
  data = dlmread( output_file_name );
 catch % reading the data did not work as expected, make a short pause and then try to re-read
  pause( obj.retry_timeout );
  data = dlmread( output_file_name );
 end

 %read data

 sim_result = %compute objective function

 % For multiobjective optimization:
 % sim_result(1) =
 % sim_result(2) =
end
```

The first input argument of this function is again the object of class `ObjFUNbase`. The output directory is also needed in this function to know where to look for the result files given by `output_dir`. The iteration number of the task within one iteration is passed to identify the filename of the respective task. It is very essential to use the same syntax used in `setup_single_task`. The return value of this function should be the numerical value(s) of the objective function(s). This is passed directly to the used optimization algorithm. For single objective optimization use `sim_result)` and for multi-objective optimization use `sim_result(i)` to define the value for objective `i`.

2.1.2 Implementation of RunTASKSbase.m

This superclass handle only consists of one abstract method called `run_tasks`. This method receives a list of tasks from `ObjFUNbase` and its purpose is to execute each task of this list according to the method supplied by the user.

Listing 2.4: Definition of function to execute a list of tasks.

```
function status=run_tasks(objrun, tasks)
```

```
 % run all tasks{i} where i=1:size(tasks,2)
end
```

The function input is the object of class `RunTASKSbase` called `objrun` and the actual list of tasks called `tasks` as defined in `setup_single_task`. The list has the data format of a cell array and single tasks can therefore be accessed by `tasksi`. The status of the function should return zero if no error occurred during the simulation execution `status=0` and should be none zero otherwise.

2.2 Optimization

Once the interface of input and output between simulation tool and optimization algorithm and objective function are defined, the optimization problem and choice of algorithm settings has to be specified. This is done in a separate MATLAB file as can be seen in the example file `example.m`:

Listing 2.5: Choice of algorithm and optimization specific settings.

```
options.LB = [1000 1000];
options.UB = [200000 200000];
options.A_eq = [];
options.b_eq = [];
options.A_ineq = [1 -1];
options.b_ineq = [0];
options.maxfunevals = 300;
options.nvars = 2;
options.npop = 20;
options.nobj = 1;
algorithm = 'NSGA-II';
runobj = RunTASKSvmcontrol();
obj = ObjFUNfumola(runobj);

result=optFUMOLA(obj, algorithm, options);
```

Before invoking the optimization process various parameters have to be set that define the optimization problem:

options all the following option and definitions of the optimization problem have to be passed to `optFUMOLA` as member of this struct which should contain the following fields:

- `LB` defining the lower bounds
- `UB` defining the upper bounds
- `A_eq` defining the left side of the equality constraint $A*x = b$ (only supported by PSO algorithm; leave empty if no constraints apply)
- `b_eq` defining the right side of the equality constraint $A*x = b$ (only supported by PSO algorithm; leave empty if no constraints apply)
- `A_ineq` defining the left side of the inequality constraint $A*x \leq b$ (leave empty if no constraints apply)
- `b_ineq` defining the right side of the inequality constraint $A*x \leq b$ (leave empty if no constraints apply)
- `maxfunevals` defines the maximum number of function evaluations, i.e., simulation runs
- `nvars` defines the number of design variables

- `npop` defines the number of population members (for DE, PSO, PSwarm and NSGA-II) and number of new points selected at each iteration (for MATSuMoTo)
- `nobj` defines the number of objective functions (only used for multi-objective optimization with NSGA-II)

algorithm your choice of optimization algorithm from the following list
- `'DE'` for Differential Evolution
- `'PSO'` for Particle Swarm
- `'MATSuMoTo'` for Efficient Global Optimization building and using a surrogate model
- `'PSwarm'` for a hybrid algorithm combining Pattern Search and Particle Swarm
- `'NSGA-II'` for a multi-objective optimization. Here more than one objective function in `ObjFUN<user>.m` has to be defined.

runobj instantiates the class `RunTASKSbase`

obj instantiates the class `ObjFUNbase`

To access additional options for the different algorithms you have to modify `optFUMOLA.m` directly.

3. Solvers

3.1 Single objective optimization

$$
\begin{array}{rll}
\min & f(x_1,\ldots,,x_n) & \\
\text{subject to} & g_i(x_1,\ldots,,x_n) \leq 0 & i = 1,\ldots,,m \\
& h_i(x_1,\ldots,,x_n) = 0 & i = 1,\ldots,,p \\
& l_i \leq x_i \leq u_i & i = 1,\ldots,,n
\end{array}
$$

Here f represents the objective function dependent on $x_1,\ldots,x_n$ that is to be minimized. The lower and upper bounds of design variable x_i are given by l_i and u_i, respectively. $g_i(x_1,\ldots,x_n)$ are called inequality constraints and $h_i(x_1,\ldots,x_n)$ equality constraints.
By convention, the standard form defines a minimization problem. A maximization problem can be treated by negating the objective function.

A number of algorithms with quite different approaches for solving optimization problems have been included into optFUMOLA. So far only continuous problems are considered, however, optFUMOLA is in this regard not restricted. For instance, mixed integer problems could be tackled once appropriate optimization algorithms are included.
The following is a short description of the solvers that are already part of optFUMOLA:

3.1.1 Particle Swarm

This population-based algorithm maintains at each iteration a swarm of particles with a position and velocity vector associated with each particle. A new set of particles is produced from the previous swarm using rules that take into account particle swarm parameters (inertia, cognition and social) and randomly generated weights. The implementation[1] used for optFUMOLA handles bound constraints as well as equality and inequality constraints.

[1] Constrained Particle Swarm Optimization, available at:
http://www.mathworks.com/matlabcentral/fileexchange/

3.1.2 Differential Evolution

The basic idea behind this population-based algorithm is that new trial members are produced by adding the weighted difference between two population members to a third member. An openly available implementation[2] of this algorithm has been adapted for the purpose of optFUMOLA. It has been modified to use an extreme barrier function, setting objective function values at unfeasible points to infinity, to handle inequality constraints.

3.1.3 PSwarm

This algorithm is an implementation of the particle swarm pattern search method. Its poll step relies on a coordinate search method that is responsible for local convergence, whereas its search step performs a global search based on the particle swarm algorithm. The latter enables the exploration of the whole design variable space. The implementation[3] adapted for optFUMOLA is able to deal with bound and inequality constraints.

3.1.4 MATSuMoTo

This algorithm belongs to the class of Efficient Global Optimization and uses so-called surrogate models to approximate expensive objective functions. It starts by performing a space-filling experimental design and evaluates the objective function at the design points. The surrogate model is then build with this information by using radial basis functions to interpolate the already evaluated points of the objective function. Then, during the optimization phase, information from the surrogate model is used to carefully select new points, where the computationally expensive objective function will be evaluated. The surrogate model is then updated and new points are selected. This process is repeated until a stopping criterion is met. The implementation[4] adapted for optFUMOLA handled only bound constraints. In order to deal with inequality constraints, a penalty function has been introduced, adding a penalty term to the objective function value at unfeasible points.

3.2 Multi objective optimization

$$\begin{array}{rlr} \min & f_i(x_1,\ldots,,x_n) & i=1,\ldots,,m \\ \text{subject to} & g_i(x_1,\ldots,,x_n) \leq 0 & i=1,\ldots,,l \\ & h_i(x_1,\ldots,,x_n) = 0 & i=1,\ldots,,k \\ & l_i \leq x_i \leq u_i & i=1,\ldots,,n \end{array}$$

Here f_i represents the ith objective function dependent on $x_1,\ldots,x_n$ that is to be minimized. The lower and upper bounds of design variable x_i are given by l_i and u_i, respectively. g_i are called inequality constraints and h_i equality constraints.
By convention, the standard form defines a minimization problem for each objective function. A maximization problem can be treated by negating the respective objective function.

[2] Differential Evolution (DE), available at:
http://www1.icsi.berkeley.edu/~storn/code.html

[3] PSwarm, available at:
http://www.norg.uminho.pt/aivaz/pswarm/

[4] MATSuMoTo, available at:
https://github.com/Piiloblondie/MATSuMoTo

3.2.1 NSGA-II

In contrast to the aforementioned single-objective optimization algorithms, NSGA-II allows for multiple objectives. This means that the result does not consist of a single point in the search space, but is represented by a so-called Pareto front. The version of this genetic algorithm used here[5] allows for different types of constraints.

3.3 Extensibility

Extending optFUMOLA by adding new algorithms involves only a manageable amount of effort. It typically involves the following adaptations to existing algorithm implementations:

- Instead of directly calling a simple function, optimization algorithms need to use method `objective_function` provided by class `ObjFUNbase` to compute values of the objective function.
- To enable parallel execution of simulations, it is necessary that all model input parameter sets are passed to `objective_function` at once as a single list.
- Explicit handling of options via function `optFUMOLA`, as done for already implemented algorithms, simplifies usability.

[5] NGPM – A NSGA-II Program in Matlab v1.4, available at: http://www.mathworks.com/matlabcentral/fileexchange/

4. License