RBFOpt user manual

Giacomo Nannicini

IBM T. J. Watson, Yorktown Heights, NY. E-mail: nannicini@us.ibm.com

November 17, 2022

Contents

1 Getting started

RBFOpt is an open-source library for black-box optimization. It is available on GitHub at:

`https://github.com/coin-or/rbfopt`

The code is mainly developed for Python 3, but it also runs on Python 2.7. We recommend using Python 3 if possible. The recommended way to install the package is via the Python module manager `pip`. The code is on PyPI, therefore it can be installed using:

```
pip install rbfopt
```

The user can install from source, downloading an archive or cloning from GitHub (for example to use a development version that is not released on PyPI yet), running the following command from the directory containing `setup.py`:

```
pip install -e .
```

The `-e` switch is typically necessary if the user is employing a virtual environment to ensure the scripts use the correct Python interpreter; the library should be installed correctly even without `-e`. To build the documentation, the user also needs numpydoc:

```
pip install numpydoc
```

On Windows systems, we recommend WinPython `http://winpython.sourceforge.net/`, which comes with NumPy, SciPy and pip already installed. After installing WinPython, it is typically necessary to update the PATH environment variable. The above command using `pip` to install missing libraries has been successfully tested on a fresh WinPython installation.

RBFOpt requires the solution of convex and nonconvex nonlinear programs (NLPs), as well as nonconvex mixed-integer nonlinear programs (MINLPs) if some of the decision variables (design parameters) are constrained to be integer. Solution of these subproblems is performed through Pyomo, which in principle supports any solver with an AMPL interface (.nl file format). The code is setup to employ Bonmin and Ipopt, that are open-source, with a permissive license, and available through the COIN-OR repository. The end-users are responsible for checking that they have the right to use these solvers. To use different solvers, a few lines of the source code have to be modified: ask for help on GitHub.

To obtain pre-compiled binaries for Bonmin and Ipopt for several platforms, we suggest having a look at the AMPL opensource solvers at `http://ampl.com/products/solvers/open-source/` or `http://ampl.com/dl/open/`, for static binaries. Note that these binaries might be outdated: better performance can sometimes be obtained compiling Bonmin from scratch (Bonmin contains Ipopt as well), especially if compiling with a different solver for linear systems rather than the default Mumps, e.g., ma27. Bonmin and Ipopt must be compiled with ASL support.

In case any of the packages indicated above is missing, some features may be disabled, not function properly, or the software may not run at all.

1.1 Verifying the installation

Install the package with `pip` as indicated above. This will install the two executable Python scripts called `rbfopt_cl_interface.py` and `rbfopt_test_interface.py` in the user's `bin/` directory (whatever is used by `pip` for this purpose), as well as the module files in the user's `site-packages` directory.

The user must make sure Bonmin and Ipopt are in their path; otherwise, the options `minlp_solver_path` and `nlp_solver_path` in RbfoptSettings have to be set to indicate the full path to the solvers. If the user employs RBFOpt as a library and creates their own RbfoptSettings object, these options can be given as:

```
import rbfopt
settings = rbfopt.RbfoptSettings(minlp_solver_path='full/path/to/bonmin',
                                 nlp_solver_path='full/path/to/ipopt')
```

If the user employs the command-line tools, these options can be simply provided preceded by double hyphen, as in:

```
rbfopt_test_interface.py --minlp_solver_path='full/path/to/bonmin' branin
```

The user can test the installation by running:

```
rbfopt_test_interface.py branin
```

See:

```
rbfopt_test_interface.py --help
```

for more details on command-line options for the testing tool. Many more test functions, with different characteristics, are implemented in the file `rbfopt_test_functions.py`. They can all be used for testing.

1.2 Performing unit tests

Unit tests for the library can be executed by running:

```
python setup.py test
```

or:

```
python setup.py nosetests
```

or:

```
nosetests
```

from the main directory, i.e., the one containing `setup.py`. If some of the tests fail, the library may or may not work correctly. Some of the test failures are relatively harmless. Users are advised to contact the mailing list (see Section 1.11) if they are unsure about some test failure.

Additional slow tests, that check if various parametrizations of the optimization algorithm can solve some global optimization problems, are found in the file `test_rbfopt_algorithm_slow.py`, which is ignored by nosetests by default. These additional tests can be executed by running:

```
python -m nose tests/test_rbfopt_algorithm_slow.py
```

1.3 Minimal working example

After installation, the easiest way to optimize a function is to use the `RbfoptUserBlackBox` class to define a black box, and execute `RbfoptAlgorithm` on it. This is a minimal example to optimize the 3-dimensional function defined below:

```
import rbfopt
import numpy as np
def obj_funct(x):
  return x[0]*x[1] - x[2]

bb = rbfopt.RbfoptUserBlackBox(3, np.array([0] * 3), np.array([10] * 3),
                               np.array(['R', 'I', 'R']), obj_funct)
settings = rbfopt.RbfoptSettings(max_evaluations=50)
alg = rbfopt.RbfoptAlgorithm(settings, bb)
objval, x, itercount, evalcount, fast_evalcount = alg.optimize()
```

As can be seen from this example, there are only four necessary steps:

- Instantiate an object of a class derived from `RbfoptBlackBox`, such as `RbfoptUserBlackBox` (see next subsection for details).
- Instantiate an object of class `RbfoptSettings`, containing all settings for the optimization algorithm.
- Instantiate an object of class `RbfoptAlgorithm`, which must be given the black box and the settings object.
- Run the optimization with the `optimize()` function of the object of class `RbfoptAlgorithm`.

Another possibility is to define a class derived from `RbfoptBlackBox` in a separate file, and execute the command-line interface instructing it to read the black box from such file. An example is provided under src/rbfopt/examples, in the file `rbfopt_black_box_example.py`. This can be executed with:

```
rbfopt_cl_interface.py --max_evaluations 50 src/rbfopt/examples/
 rbfopt_black_box_example.py
```

1.4 Structure of a user-defined black box

The class `RbfoptAlgorithm` requires an object derived from the abstract class `RbfoptBlackBox`, which describes the black box problem to be optimized. An `RbfoptBlackBox` must implement the following methods:

- `get_dimension(self)`: returns the dimension of the problem (i.e., number of decision variables), as an integer.
- `get_var_lower(self)`: returns the array of lower bounds on the decision variables, as a 1-dimensional NumPy array of floats with length equal to the dimension of the problem.
- `get_var_upper(self)`: returns the array of upper bounds on the decision variables, as a 1-dimensional NumPy array of floats with length equal to the dimension of the problem.
- `get_var_type(self)`: returns the type of each decision variable, as a 1-dimensional NumPy array of char with length equal to the dimension of the problem. Possible types are 'R' for real (continuous) variables, 'I' for integer (discrete, ordered) variables, 'C' for categorical (discrete, unordered) variables. Bounds for categorical variables are interpreted the same way as for integer variables, but categorical variables are handled differently by the optimization algorithm; e.g., a categorical variable with lower bound 2 and upper bound 4 can take the value 2, 3 or 4.
- `evaluate(self, x)`: evaluates the black-box function at the point x, and returns its value as a float. The point x is a 1-dimensional NumPy array of floats, with length equal to the dimension of the problem.
- `evaluate_noisy(self, x)`: evaluates a fast approximation of the black-box function at the point x, and returns an approximation of the value of `evaluate()`, hopefully much more quickly, as well as error bounds on the evaluation. If `has_evaluate_noisy()` returns False, this function will never be queried and therefore it does not have to return any value. The point x is a 1-dimensional NumPy array of floats, with length equal to the dimension of the problem. The return value of this function must be a NumPy array with three floats (value, lower, upper) containing the approximate

value of the function at x, the lower error bound, and the upper error bound, such that the true function value is contained between value + lower and value + upper. Hence, lower should be ≤ 0 while upper should be ≥ 0.

- `has_evaluate_noisy(self)`: indicate whether `evaluate_noisy` is available. If True, such function will be used to try to accelerate convergence of the optimization algorithm. If False, the function `evaluate_noisy` will never be queried.

Rather than defining a new class derived from `RbfoptBlackBox`, the user can also instantiate an object of class `RbfoptUserBlackBox` and pass as arguments the required information. This will return an appropriate black box. All the details are provided in the documentation of the API, within the classes `RbfoptBlackBox` and `RbfoptUserBlackBox`.

1.5 Checkpointing the optimization process

RBFOpt provides a checkpointing mechanism to save all intermediate results of the optimization. Using a checkpoint, the optimization process can be resumed at a subsequent time and no information is lost, for example in case of crashes during the evaluation of the objective function. The checkpointing mechanism saves the state of the optimization in a file, that can be loaded to recreate exactly the same state and continue the optimization. Automatic checkpointing can be performed uusing the options `save_state_interval` and `save_state_file`: after every `save_state_interval` iterations, the state will be written to `save_state_file`. By default, `save_state_interval` is equal to a large number (10000) so that checkpointing does not happen; see also Section 3.6.

In the API, checkpointing can be achieved using the functions `save_to_file(filename)` and `load_from_file(filename)` of the class `RbfoptAlgorithm`. The following code snippet performs 50 iterations of the optimization, saves the state, then loads the state and resumes.

```
settings = rbfopt.RbfoptSettings(max_evaluations=500)
# Assume bb is the RbfoptBlackBox, as in the previous example
alg = rbfopt.RbfoptAlgorithm(settings, bb)
# Perform only 50 iterations
alg.optimize(pause_after_iters=50)
alg.save_to_file('state.dat')
# Now load the state back: this creates a new RbfoptAlgorithm object
alg_loaded = rbfopt.RbfoptAlgorithm.load_from_file('state.dat')
# We can continue to optimize
alg_loaded.optimize()
```

When using the command-line interface, the relevant options are:

- `--pause num_iterations`: pause the optimization after the given number of iterations.
- `--save filename`: save the state in a file with the given name, at the end of the optimization.
- `--load filename`: load the state from the file with the given name, before resuming the optimization.

1.6 Providing initial points

The optimization algorithm generates a suitable set of points to build an initial model of the unknown objective function. The user can provide additional points, or an alternative set of points, using the arguments `init_node_pos` and `init_node_val` in the constructor of `RbfoptAlgorithm`, together with the boolean argument `do_init_strategy`. This feature works as follows:

- If `do_init_strategy` is True (which is the default value), the algorithm will generate a set of initial points. All points given in `init_node_pos`, if any, will be added to the initial set, provided that they are sufficiently far from other points. Function values will be taken from `init_node_val`, if it is of the same size of `init_node_pos`, otherwise they will be computed using the `evaluate()` function of the `RbfoptBlackBox`.
- If `do_init_strategy` is False, the algorithm will use the points given in `init_node_pos` as initial points. Note that if not enough points are given, or if not enough points satisfy the minimum distance criterion, the algorithm will raise an exception. Function values will be taken from `init_node_val`, if it is of the same size of `init_node_pos`, otherwise they will be computed using the `evaluate()` function of the `RbfoptBlackBox`.

1.7 Interpreting and manipulating the output

The output printed on screen during the optimization, when using the command-line interface, looks like this:

```
RbfoptSettings:
algorithm: MSRSM, debug: False, discarded_window_size: 30, do_infstep: False,
    domain_scaling: auto, dynamism_clipping: auto, dynamism_threshold: 1000.0,
     eps_impr: 0.0001, eps_linear_dependence: 1e-06, eps_opt: 0.01, eps_zero:
    1e-15, function_scaling: auto, ga_base_population_size: 400,
    ga_num_generations: 20, global_search_method: genetic, init_strategy:
    lhd_maximin, local_search_box_scaling: 0.5, local_search_threshold: 0.25,
    log_scaling_threshold: 1000000.0, max_clock_time: 1e+30,
    max_consecutive_refinement: 5, max_consecutive_restoration: 15,
    max_cross_validations: 50, max_cycles: 1000, max_evaluations: 300,
    max_fraction_discarded: 0.5, max_iterations: 1000, max_noisy_evaluations:
    200, max_noisy_iterations: 200, max_noisy_restarts: 2, max_random_init:
    50, max_stalled_iterations: 100, min_dist: 1e-05, minlp_solver_path:
    bonmin, modified_msrsm_score: True, nlp_solver_path: ipopt, num_cpus: 1,
    num_global_searches: 5, num_samples_aux_problems: 1000,
    parallel_wakeup_time: 0.1, print_solver_output: False, rand_seed:
    71321312, rbf: auto, rbf_shape_parameter: 0.1, refinement_frequency: 3,
    save_state_file: rbfopt_algorithm_state.dat, save_state_interval: 100000,
    target_objval: 0.397887357729739, targetval_clipping: True,
    thresh_unlimited_refinement: 0.9, thresh_unlimited_refinement_stalled:
    0.9, tr_acceptable_decrease_enlarge: 0.6, tr_acceptable_decrease_move:
    0.1, tr_acceptable_decrease_shrink: 0.2, tr_init_radius_multiplier: 2.0,
    tr_min_grad_norm: 0.01, tr_min_radius: 0.001, tr_num_integer_candidates:
    10

Iter  Cycle  Action            Objective value      Time      Gap
----  -----  ------            ---------------      ----      ---
   0      0  Initialization          92.080159      0.00  2122.85
   0      0  Initialization           8.844441      0.00  2122.85 *
   0      0  Initialization         121.904879      0.00  2122.85
   0      0  GlobalStep             145.655356      0.03  2122.85
   1      0  GlobalStep               9.641585      0.06  2122.85
   2      0  GlobalStep               1.243340      0.10   212.49 *
   3      0  GlobalStep               2.417437      0.14   212.49
   4      0  GlobalStep               1.971173      0.17   212.49
   5      0  LocalStep                0.640711      0.26    61.03 *
```

```
   6      1  GlobalStep               2.530805      0.30    61.03
   7      1  GlobalStep              19.054517      0.34    61.03
   8      1  GlobalStep              31.567234      0.37    61.03
   9      1  GlobalStep               1.206690      0.41    61.03
  10      1  GlobalStep               0.735354      0.45    61.03
  11      1  LocalStep                5.377267      0.54    61.03
  12      2  GlobalStep               2.700249      0.59    61.03
  13      2  GlobalStep              13.339276      0.63    61.03
  14      2  GlobalStep               0.872641      0.66    61.03
  15      2  GlobalStep               0.570051      0.70    43.27 *
  16      2  GlobalStep               0.436091      0.74     9.60 *
  17      2  LocalStep                0.411601      0.81     3.45 *
  18      2  RefinementStep           0.453924      0.81     3.45
  19      2  RefinementStep           0.432610      0.81     3.45
  20      2  RefinementStep           0.407989      0.81     2.54 *
  21      2  RefinementStep           0.412121      0.81     2.54
  22      2  RefinementStep           0.406428      0.81     2.15 *
  23      3  GlobalStep              22.357065      0.89     2.15
  24      3  GlobalStep               4.239370      0.92     2.15
  25      3  GlobalStep               0.820612      0.97     2.15
  26      3  GlobalStep               0.939742      1.01     2.15
  27      3  GlobalStep               0.452616      1.06     2.15
  28      3  LocalStep              113.537180      1.20     2.15
  29      4  GlobalStep             106.104713      1.29     2.15
  30      4  GlobalStep              28.446172      1.33     2.15
  31      4  GlobalStep               8.011501      1.36     2.15
  32      4  GlobalStep               1.207318      1.40     2.15
  33      4  GlobalStep               0.403351      1.46     1.37 *
  34      4  LocalStep                0.397904      1.56     0.00 *
Summary: iters  35 evals  38 noisy_evals   0 cycles   5 opt_time    1.56
   tot_time    1.56 obj         0.397904 gap   0.00
RbfoptAlgorithm.optimize() returned function value 0.397904139152688
x0   :         9.425439
x1   :         2.471726
```

The command-line interface starts by printing out the full settings for the algorithm. The remaining part is printed by the `optimize()` function of `RbfoptAlgorithm`. This function prints one line per iteration, reporting (as indicated by the header):

- Iteration number.
- Cycle number. Optimization is performed in cycles, as described in Section 2; this reports the current optimization cycle.
- The type of iteration, which is one of the following: Initialization, GlobalStep, LocalStep, AdjLocalStep, RefinementStep, Restart, Restoration. These steps are explained in Section 2 during the detailed description of the algorithm; the "AdjLocalStep" is an adjusted local step performed when the initial attempt at a LocalStep fails.
- Objective function value of the function evaluation performed at the current iteration (multiple evaluations per iteration can only happen during Initialization). This will be followed by a "~" if the noisy objective function was evaluated instead.
- Wall-clock CPU time since the beginning of the optimization.

- Gap with respect to the given target objective function value (if any).
- A $*$ if an improved solution is found at the current iteration.

At the end of the optimization process, the summary line, starting with the keyword `Summary`, prints:

- `iters`: Total number of iterations.
- `evals`: Total number of function evaluations, i.e., evaluations of the black box function `evaluate()` (excluding noisy evaluations).
- `noisy_evals`: Total number of noisy function evaluations, i.e., evaluations of the black box function `evaluate_noisy()`.
- `cycles`: Total number of optimization cycles.
- `opt_time`: Time spent in the optimization algorithms (this excludes the time to evaluate the objective function).
- `tot_time`: Total elapsed time.
- `obj`: Best objective function value.
- `gap`: Final gap with respect to the given target objective function value.

To redirect the output to file one can use the `--log` option of the command-line scripts, for example:

```
rbfopt_cl_interface.py --max_evaluations 50 --log log_file_name.txt src/
  rbfopt/examples/rbfopt_black_box_example.py
```

Alternatively, through the API, the function `set_output_stream(file)` of `RbfoptAlgorithm` accepts any file for output redirection. The function must be called before `optimize()` to have any effect. Notice that output can be suppressed by passing an appropriate null file, such as `os.devnull`.

1.8 Parallel optimization

RBFOpt supports asynchronous parallel optimization using Python's multiprocessing library. This mode is enabled whenever the parameter `num_cpus` is set to a value greater than 1. Black-box function evaluations as well as some of the heaviest computatations carried out by the algorithm will then be executed in parallel. Since the parallel computations are asynchronous, determinism cannot be guaranteed: in other words, if the parallel optimizer is executed twice in a row, it may (and often will) yield different results, even if the same random seed was provided. This is because the order in which the computations will be completed may change, and this may impact the course of the algorithm.

The default parameters of the algorithm are optimized for the serial optimization mode. For recommendations on what parameters to use with the parallel optimizer, it may be a good idea to ask for suggestions on the mailing list or GitHub, see Section 1.11.

Note that the parallel optimizer is oblivious of the system-wide settings for executing linear algebra routines (BLAS) in parallel. We recommend setting the number of threads for BLAS to 1 when using the parallel optimizer, see the next section.

1.9 Known issues with OpenBLAS

We are aware of an issue when launching multiple distinct processes that use RBFOpt and the NumPy implementation is configured to use OpenBLAS in parallel: in this case, on rare occasions we have observed that some processes may get stuck forever when computing matrix-vector multiplications. The problem can be fixed by setting the number of threads for OpenBLAS to 1. We do not know if the same issue occurs with other parallel implementations of BLAS.

For this reason, and because parallel BLAS uses resources suboptimally when used in conjunction with the parallel optimizer of RBFOpt (if BLAS runs in parallel, each thread of the parallel optimizer would spawn multiple threads to run BLAS, therefore disregarding the option `num_cpus`), RBFOpt attempts to set the number of BLAS threads to 1 at run time.

All scripts (`rbfopt_cl_interface.py` and `rbfopt_test_interface.py`) set the environment variables `OMP_NUM_THREADS` to 1. Furthermore, the rbfopt module does the same when imported for the first time.

Note that these settings are only effective if the environment variable is set before NumPy is imported; otherwise, they are ignored. Users facing the same issue should try to set the environment variable `OMP_NUM_THREADS` to 1 before NumPy is imported. In Python, this can be done with:

```
import os
os.environ['OMP_NUM_THREADS'] = '1'
```

1.10 Documentation of the API

The documentation for the code can be built using Sphinx with the `numpydoc` extension. `numpydoc` can be installed with pip:

```
pip install numpydoc
```

After that, the directory `src/rbfopt/doc/` contains a Makefile (on Windows, the file `make.bat` should be used instead) and the Sphinx configuration file `conf.py`.

The user can build the HTML documentation (recommended) with:

```
make html
```

The output will be located in `_build/html/` and the index can be found in `_build/html/index.html`.

A PDF version of the documentation (much less readable than the HTML version) can be built using the command:

```
make latexpdf
```

An online version of the documentation for the latest master branch of the code, and for the latest stable release, are available on ReadTheDocs for the latest branch `http://rbfopt.readthedocs.org/en/latest/`, and stable branch `http://rbfopt.readthedocs.org/en/stable/`.

1.11 Support

The best place to ask question is to start a thread in GitHub's discussion page: `https://github.com/coin-or/rbfopt/discussions`. Issues opened on GitHub are regularly checked.

2 The optimization algorithm

In this section we provide a detailed description of the algorithms implemented in RBFOpt. Further details can be found in two papers: [2] describes the mathematical foundations for RBFOpt v1.0, many of which

$\phi(r)$		d	value param. `rbf`
r	(linear)	0	`linear`
r^3	(cubic)	1	`cubic`
$\sqrt{r^2+\gamma^2}$	(multiquadric)	0	`multiquadric`
$r^2 \log r$	(thin plate spline)	1	`thin_plate_spline`
$e^{-\gamma r^2}$	(Gaussian)	-1	`gaussian`

Table 1: RBF functions available in RBFOpt, and corresponding value of the parameter `rbf` to select a type of RBF function.

are relevant for the current version; [5] described the algorithm of version 4.2, in a very detailed manner — including handling of categorical variables.

RBFOpt addresses a problem cast in the following form:

$$\left.\begin{array}{rrcl} \min & f(x,w) & & \\ & x & \in & [x^L, x^U] \\ & x & \in & \mathbb{R}^{n_r} \times \mathbb{Z}^{n_d} \\ & w & \in & \bigtimes_{h=1}^{n_c} S_h, \end{array}\right\} \tag{1}$$

where for $h = 1, \ldots, n_c$, S_h is an (unordered) finite set, and $x^L, x^U \in \mathbb{R}^{n_r+n_d}$ are vectors of finite lower and upper bounds. We assume that the analytical expression for f is unknown and function values are only available through an oracle that is expensive to evaluate, e.g., a time-consuming computer simulation. In the literature, this is typically called a black-box optimization problem with costly evaluation. The sets S_h represent the possible values of the categorical variables.

For $i = 1, \ldots, n_c$, we define $m_h = |S_h|$ and $\hat{m}_h = \sum_{k=1}^{h} m_k$, with $\hat{m}_0 = 0$ for convenience. Extend x^L by appending the all-zero vector of length $\hat{m}_{n_c}$, and extend x^U by appending the all-one vector of length $\hat{m}_{n_c}$. The optimization is performed using the following reformulation:

$$\left.\begin{array}{rrcl} \min & f(x_1, \ldots, x_{n_r+n_d}, \hat{C}(x_{n_r+n_d+1}, \ldots, x_{n_r+n_d+\hat{m}_{n_c}})) & & \\ & x & \in & [x^L, x^U] \\ \forall h = 1, \ldots, n_c & \sum_{j=\hat{m}_{h-1}+1}^{\hat{m}_h} x_{n_r+n_d+j} & = & 1 \\ & x & \in & \mathbb{R}^{n_r} \times \mathbb{Z}^{n_d} \times \{0,1\}^{\hat{m}_{n_c}} \end{array}\right\} \tag{2}$$

where $\hat{C} : \{0,1\}^{\hat{m}_{n_c}} \to \bigtimes_{h=1}^{n_c} S_h$ maps a binary vector $\in \{0,1\}^{\hat{m}_{n_c}}$ to a choice of elements from the sets S_h, by viewing its argument as the juxtaposition of the characteristic vectors of the sets S_h. Let Ω be the feasible region of this problem. We assume that the box constraints on the integer and categorical have integer endpoints.

Given k distinct points $x^1, \ldots, x^k \in \Omega$, a radial basis function interpolant s_k is defined as:

$$s_k(x) := \sum_{i=1}^{k} \lambda_i \phi(\|x - x^i\|) + p(x), \tag{3}$$

where $\phi : \mathbb{R}_+ \to \mathbb{R}$, $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ and p is a polynomial of degree d. The degree d is chosen according to Table 1, depending on the type of radial basis functions $\phi(r)$. The type of RBF is chosen with the parameter `rbf` and the value of $\gamma > 0$ can be changed with the parameter `rbf_shape_parameter`, set to 0.1 by default.

If $\phi(r)$ is cubic or thin plate spline, we obtain an interpolant of the form:

$$s_k(x) := \sum_{i=1}^{k} \lambda_i \phi(\|x - x^i\|) + h^T \begin{pmatrix} x \\ 1 \end{pmatrix}, \tag{4}$$

where $h \in \mathbb{R}^{n+1}$. The values of λ_i, h can be determined by solving the following linear system:

$$\begin{pmatrix} \Phi & P \\ P^T & 0_{(n+1)\times(n+1)} \end{pmatrix} \begin{pmatrix} \lambda \\ h \end{pmatrix} = \begin{pmatrix} F \\ 0_{n+1} \end{pmatrix}, \tag{5}$$

with:

$$\Phi = \left(\phi(\|x^i - x^j\|)\right)_{i,j=1,\ldots,k}, \quad P = \begin{pmatrix} (x^1)^T & 1 \\ \vdots & \vdots \\ (x^k)^T & 1 \end{pmatrix}, \quad \lambda = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_k \end{pmatrix}, \quad F = \begin{pmatrix} f(x^1) \\ \vdots \\ f(x^k) \end{pmatrix}.$$

The algorithm presented later in this section ensures that the points $x^1, \ldots, x^k$ are pairwise distinct and if $k \geq n+1$, then rank$(P) = n+1$, guaranteeing that the system (5) is nonsingular. We denote by A_k the matrix of (5) with points $x^1, \ldots, x^k$.

If $\phi(r)$ is linear or multiquadric, $d = 0$ and P is the all-one column vector of dimension k. In the Gaussian case, $d = -1$ and P is removed from system (5). The dimensions of the zero matrix and vector in (5) are adjusted accordingly.

The general algorithmic scheme employed by the algorithm is the following:

- **Initial step:** Set k equal to the size of the initial sample set. Choose k affinely independent points $x^1, \ldots, x^k \in \Omega_I$ using an initialization strategy.
- **Iteration step:** Repeat the following steps until k exceeds the prescribed number of function evaluations.
  - (i) Compute the RBF interpolant s_k to the points $x^1, \ldots, x^k$, solving (5). If the system is underdetermined, because $k < n+1$, find the least squares solution.
  - (ii) Choose a trade-off between *exploration* and *exploitation*.
  - (iii) Determine the next point x^{k+1} based on the choice at step (ii).
  - (iv) Evaluate f at x^{k+1}.
  - (v) Set $k \leftarrow k+1$. If the last Refinement step was performed sufficiently many iterations ago, go to the Refinement step. Otherwise, repeat the Iteration step.
- **Refinement step:**
  - (i) Select $n+1$ points out of $x^1, \ldots, x^k$ to initialize a local model.
  - (ii) Apply a trust region method for a specified number k' of iterations, obtaining points $x^{k+1}, \ldots, x^{k+k'}$.
  - (iii) Set $k \leftarrow k+k'$ and go back to the Iteration step.

At Iteration step (ii), *exploration* implies trying to improve the surrogate model in unknown parts of the domain, whereas *exploitation* implies trying to find the best objective function value based on the current surrogate model.

Whenever numerical errors in the computation of the surrogate model are detected, the algorithm will switch to a Restoration step that tries to improve the set of interpolation points. The Restoration step is described in [5].

2.1 Choice of the initial sample points

In serial mode, the initial number of sample points is given by `init_sample_fraction`$(n+1)$, where the value of `init_sample_fraction` is chosen automatically if the user provided a negative value for it. In parallel mode, the initial number of sample points is given by `init_sample_fraction`$(n+1)(1+$ `init_sample_increase_parallel` $*$ `num_cpus`$)$. The choice of the initial points is determined by the parameter `init_strategy`. The possible values are:

- `all_corners`: select all 2^n corners of the box $[x^L, x^U]$.
- `lower_corners`: select x^L and the n vectors $x^L + (x^U - x^L)e^i$, $i = 1, \ldots, n$.
- `rand_corners`: select $n+1$ corners of the box $[x^L, x^U]$ uniformly at random.
- `lhd_maximin`: select a Latin Hypercube Design with $n+1$ points. We first generate 50 LHDs at random, and then choose the one that attains the maximum minimum distance between the points. This is the default strategy.
- `lhd_corr`: select a Latin Hypercube Design with $n+1$ points. We first generate 50 LHDs at random, and then choose the one that attains the minimum maximum correlation between the points.

Additionally, the midpoint of the box (i.e., the point $\frac{x^L+x^U}{2}$) will be included among the initial sample points if the option `init_include_midpoint` is set to True.

2.2 Determining the next point: Iteration step

RBFOpt implements a variation of two algorithms for global optimization using RBFs: Gutmann's RBF algorithm [4] and the Metric Stochastic Response Surface Method (MSRSM) [6]. The two algorithms can be chosen using the `algorithm` option, using the values `Gutmann` or `MSRSM` (default). Both algorithms use a parameter κ that is set by the option `num_global_searches`.

2.2.1 Gutmann's RBF algorithm

A detailed description is given in [2]. Here we report the main steps only. Let ℓ_k be the RBF interpolant to the points $(x^i, 0)$, $\forall i \in \{1, \ldots, k\}$ and $(y, 1)$. Let $\mu_k(y)$ be the coefficient of ℓ_k corresponding to the RBF centered at y. Define

$$g_k(y) = (-1)^{d+1}\mu_k(y)[s_k(y) - f_k^*]^2, \qquad y \in \Omega \setminus \{x^1, \ldots, x^k\},$$

where f_k^* is a given value. Furthermore, define:

$$h_k(x) = \begin{cases} \frac{1}{g_k(x)} & \text{if } x \notin \{x_1, \ldots, x_k\} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Gutmann's RBF method then implements the following Iteration step:

- **Iteration step** (for Gutmann's RBF algorithm):

  (ii) Choose a target value $f_k^* \in \mathbb{R} \cup \{-\infty\} : f_k^* \leq \min_{x\in\Omega} s_k(x)$.

  (iii) Compute

  $$x_{k+1} = \arg\max_{x\in\Omega} h_k(x), \tag{7}$$

  where $h(x)$ is defined as in (6).

Let $y^* := \arg\min_{x\in\Omega} s_k(x)$, $f_{\min} := \min_{i=1,\ldots,k} f(x^i)$, and $f_{\max} := \max_{i=1,\ldots,k} f(x^i)$. We employ a cyclic strategy that picks target values $f_k^* \in \mathbb{R}\cup\{-\infty\}$ according to the following sequence of length $\kappa+2$:

- Step -1 (*InfStep*): Choose $f_k^* \leftarrow -\infty$. In this case the problem of finding x^{k+1} can be rewritten as:

$$x_{k+1} = \arg\max_{x\in\Omega} \frac{1}{(-1)^{d+1}\mu_k(x)}.$$

  This is a pure exploration phase, yielding a point far from $x_1, \ldots, x_k$. This step is only performed if `do_infstep` is equal to True (default False).

- Step $h \in \{0, \ldots, \kappa-1\}$ (*Global search*): Choose

$$f_k^* \leftarrow s_k(y^*) - (1 - h/\kappa)^2(f_{\max} - s_k(y^*)). \tag{8}$$

  In this case, we try to strike a balance between improving model quality and finding the minimum.

- Step κ (*Local search*): Choose $f_k^* \leftarrow s_k(y^*)$. Notice that in this case (6) is maximized at y^*. Hence, if $s_k(y^*) < f_{\min} - 10^{-10}|f_{\min}|$ we accept y^* as the new sample point x_{k+1} without solving (7). Otherwise we choose $f_k^* \leftarrow f_{\min} - 10^{-2}|f_{\min}|$ (this is reported as AdjLocalStep in the optimization log, see Section 1.7). This is an exploitation phase, trying to find the best objective function value based on current information. This step is only performed if `do_local_search` is equal to True (default True).

2.2.2 MSRSM algorithm

Define $\text{dist}(x) := \min_{i=1,\ldots,k} \|x - x^i\|$. The MSRSM algorithm implements the following Iteration step:

- **Iteration step** (for the MSRSM algorithm):

  (ii) Choose a target value $\alpha \in [0,1]\cup\{\infty\}$.

  (iii) Choose a finite set of reference points $P \subset \Omega \setminus \{x^1, \ldots, x^k\}$, and compute

$$x_{k+1} = \arg\min_{x\in\Omega} \alpha\frac{\max_{y\in P}\text{dist}(y) - \text{dist}(x)}{\max_{y\in P}\text{dist}(y) - \min_{y\in P}\text{dist}(y)} + \frac{s_k(x) - \min_{y\in P} s_k(y)}{\max_{y\in P} s_k(y) - \min_{y\in P} s_k(y)}. \tag{9}$$

Essentially, (9) tries to solve a bi-objective optimization problem in which the two objective functions are the (negative of the) maximin distance from the points $x^1, \ldots, x^k$, and the value of the surrogate model. The paper [6] uses a variation of (9), in which the second fraction in the expression has weight $(1-\alpha)$ rather than 1. RBFOpt supports this variation: in order to use the $1-\alpha$ weight, rather than the recommended 1, it suffices to set the option `modified_msrsm_score` to False (the default is True). The default version of formula (9) was introduced in [1].

The value of α is chosen according to a cyclic strategy of length $\kappa+2$ in which each step has similar goals to the corresponding step discussed in Gutmann's RBF method. The cyclic strategy is as follows:

- Step -1 (*InfStep*): Choose $\alpha \leftarrow \infty$. In this case the problem of finding x^{k+1} can be rewritten as:

$$x_{k+1} = \arg\max_{x\in\Omega} \min_{i=1,\ldots,k} \|x - x^i\|.$$

  This is a pure exploration phase. This step is only performed if `do_infstep` is equal to True (default False).

- Step $h \in \{0, \ldots, \kappa - 1\}$ (*Global search*): Choose $\alpha \leftarrow \max\{1 - (h+1)/\kappa, 0.05\}$. This aims for balance between exploration and exploitation.

- Step κ (*Local search*): Choose $\alpha \leftarrow 0$. In this case, the solution to (9) is the point that minimizes the surrogate model, i.e., $y^* = \arg\min_{y \in \Omega} s_k y$. If y^* is such that $s_k(y^*) < f_{\min} - 10^{-10}|f_{\min}|$, we accept y^* as the new point x_{k+1}. Otherwise, choose $\alpha \leftarrow 0.05$ (this is reported as AdjLocalStep in the optimization log, see Section 1.7). This is an exploitation phase, trying to find the best objective function value based on current information. This step is only performed if `do_local_search` is equal to True (default True).

2.2.3 Solution of the search problems

RBFOpt implements three different approaches for the solution of the optimization problems (7) and (9). The choice is determined by the option `global_search_method`, which can take the following values:

- `genetic`: problems (7) and (9) are solved with a simple and fast genetic algorithm. The base population size is determined by the parameter `ga_base_population_size` and the number of generations is determined by `ga_num_generations`.

- `sampling`: rather than solving (7) and (9) directly, we sample a large number of points in Ω and choose the best point in the sample. The number of sample points is equal to: $n \times$ `num_samples_aux_problems`.

- `solver`: problems (7) and (9) are solved by means of the mathematical programming solvers Ipopt and Bonmin.

The default setting is `genetic`. Note that the MSRSM scoring function requires a set of reference points P, see (9): the set of reference points is taken to be the current population for `genetic`, the whole sample for `sampling`, and $x^1, \ldots, x^k$ for `solver`. For MSRSM, we recommend `genetic` or `sampling` (`solver` has poor performance with MSRSM). For Gutmann's RBF method we recommend `solver`, unless iteration speed is a concern – in which case `genetic` and `sampling` are to be preferred.

2.3 Determining the next point: Refinement step

RBFOpt executes a Refinement step periodically during the search. The purpose of the Refinement step is to improve the best solution available by performing a local search around it, using a variation of a trust region method. The Refinement step is triggered after `refinement_frequency` full cycles of the global search strategy in the Iteration step (i.e., the strategy to select f_k^* in Gutmann's RBF method, or α in MSRSM), but only if one of the following two conditions apply: (i) a new incumbent was discovered since the last execution of the Refinement step, or (ii) the last Refinement step was stopped because of its iteration limit, rather than for lack of improvement.

The Refinement step works as follows:

- **Model initialization:** Let $j \leftarrow \arg\min_{i=1,\ldots,k} f(x^i)$. Sort the points $x^1, \ldots, x^k$ by increasing distance from x^j, and select the first $n+1$ (this includes x^j itself). Let S be these points. Set $\bar{x} \leftarrow x^j$.

- Let $\hat{x}$ be the point in S with the $\lceil \frac{n+1}{4} \rceil$ smallest distance to $\bar{x}$. Compute the initial radius of the trust region ρ as:

$$\rho = \max\{\|\bar{x} - \hat{x}\|, \texttt{ref\_init\_radius} \times 2^{\texttt{ref\_init\_radius\_multiplier}}\}.$$

- **Refinement:** repeat `max_consecutive_refinement` times, or until a stopping criterion is met.

  (i) Let M be the matrix obtained using the points $x^i \in S$ as columns.

  (ii) If M is rank-deficient, use a QR factorization of M to replace one point in S with a new point (taken from the columns of Q after rescaling) that increases the rank of M, and go back to (i).

  (iii) Otherwise, build a linear model $c^\top x + b$ of the objective function using points $(x^i, f(x^i)), x^i \in S$.

  (iv) Move from the current iterate $\bar{x}$ in the direction of improvement $-c$ with step length:

$$t = \max_{0 \le t \le \rho} \{t : \bar{x} - tc \in [x^L, x^U]\}.$$

  Let $\bar{x}' = \bar{x} - tc$ be the new candidate point.

  (v) Evaluate $f(\bar{x}')$. Update the trust region radius based on the expected decrease $c^\top(\bar{x} - \bar{x}')$ and the actual decrease $f(\bar{x}) - f(\bar{x}')$: if $\frac{f(\bar{x})-f(\bar{x}')}{c^\top(\bar{x}-\bar{x}')} \le$ `ref_acceptable_decrease_shrink`, set $\rho \leftarrow \rho/2$, if $\frac{f(\bar{x})-f(\bar{x}')}{c^\top(\bar{x}-\bar{x}')} \ge$ `ref_acceptable_decrease_enlarge` set $\rho \leftarrow 2\rho$.

  (vi) If $\frac{f(\bar{x})-f(\bar{x}')}{c^\top(\bar{x}-\bar{x}')} \ge$ `ref_acceptable_decrease_move`, set $\bar{x} \leftarrow \bar{x}'$. Otherwise, reject $\bar{x}'$.

  (vii) Replace the point in S with largest objective function value with the new point $\bar{x}'$, and go back to (i).

When the Refinement step ends, all points evaluated with f are added to $x^1, \ldots, x^k$, and we go back to the Iteration step (with Gutmann's RBF method or MSRSM). There are several stopping criteria for the Refinement step:

- `max_consecutive_refinement`: the Refinement step stops after this many consecutive iterations, unless the total number of evaluations of the objective f exceeds `max_evaluations` $\times$ `thresh_unlimited_refinement` points, or the wall clock time since the beginning of the search exceeds `max_clock_time` $\times$ `thresh_unlimited_refinement`.

- `ref_min_radius`: if the trust region radius ρ drops below this value, the Refinement step stops.

- `ref_min_grad_norm`: if the norm of the gradient of the linear model used in the trust region drops below this value, the Refinement step stops.

The scheme described above is based on trust region methods, which implies that it is devised for continuous variables. RBFOpt applies the Refinement step even in the presence of discrete variables, with no guarantee of convergence. When the problem has discrete variables the Refinement step proceeds as described above, but every candidate point is rounded to an integer point before being evaluated with f. In particular, every integer variable that takes on a fractional value in the candidate point, say $\bar{x}_j$ is rounded down with probability $1 - \lfloor \bar{x}_j \rfloor$, and rounded up with probability $\lfloor \bar{x}_j \rfloor$. The rounding process is repeated `ref_num_integer_candidates` times, and the point with the best linear model score is chosen as the next candidate. Furthermore, in step (ii) the column of Q that is about to replace one column in M is also rounded to the closest integer in its fractional component.

2.4 Automatic model selection

RBFOpt assesses model quality using a cross validation scheme, in order to dynamically choose the surrogate model that appears to be the most accurate for the problem at hand. Given a data set, cross validation

consists in using part of the data set to fit a model, and testing its quality on the remaining part of the data set. The process is then iterated, rotating the parts of the data set used for model fitting and for testing.

Let s_k be the surrogate model for f based on k evaluation points $x^1, \ldots, x^k$. We assume that the points are sorted by increasing function value: $f(x^1) \leq f(x^2) \leq \cdots \leq f(x^k)$; this is without loss of generality as we can always rearrange the points. We perform cross validation as follows. For $j \in \{1, \ldots, k\}$, we can fit a surrogate model $\tilde{s}_{k,j}$ to the points $(x^i, f(x^i))$ for $i = 1, \ldots, k, i \neq j$ and evaluate the performance of $\tilde{s}_{k,j}$ at $(x^j, f(x^j))$. We use an order-based measure to evaluate performance of the surrogate model. For a given scalar y, let $\text{order}_{k,j}(y)$ be the position at which y should be inserted in the ordered list $f(x^1) \leq \cdots \leq f(x^{j-1}) \leq f(x^{j+1}) \leq \cdots \leq f(x^k)$ to keep it sorted. Since $\text{order}_{k,j}(f(x^j)) = j$, we use the value $q_{k,j} = |\text{order}_{k,j}(\tilde{s}_{k,j}(x^j)) - j|$ to assess the predictive power of the model. We then average $q_{k,j}$ with j ranging over some subset of $\{1, \ldots, k\}$ to compute a model quality score. This approach is a variation of leave-one-out cross validation in which we look at how the surrogate model ranks the left-out point compared to the other points, rather than evaluate the accuracy of the prediction in absolute terms. This is motivated by the observation that for the purpose of optimization, a surrogate model that ranks all points correctly is arguably more useful than a surrogate model that attains small absolute errors, but is not able to predict how points compare to each other.

We perform model selection at the beginning of every cycle of the search strategy to select f_k^* or α (depending on the choice of algorithm: Gutmann's RBF or MSRSM). Our aim is to select the RBF model with the best predictive power. We choose two different models: one for local search, one for global search, corresponding to different Iteration steps of the algorithm. We do this by computing the average value $\bar{q}_{10\%}$ of $q_{k,j}$ for $j = 1, \ldots, \lfloor 0.1k \rfloor$, and the average value $\bar{q}_{70\%}$ of $q_{k,j}$ for $j = 1, \ldots, \lfloor 0.7k \rfloor$.

The RBF model with the lowest value of $\bar{q}_{10\%}$ is employed in the subsequent optimization cycle for the *Local search* step and the *Global search* step with $h = \kappa - 1$, while the RBF model with lowest value of $\bar{q}_{70\%}$ is employed for all the remaining steps. RBFOpt considers the following RBF models: linear, cubic, Gaussian, multiquadric or thin plate spline basis functions. This implies that the type of RBF can dynamically change during the course of the optimization.

In [2], we show that the values $\bar{q}_{10\%}, \bar{q}_{70\%}$ can be computed in time $O(m^3)$, where m is the number of rows of (5) (i.e., $m = k + n + 1$ for cubic and thin plate spline RBF, $m = k + 1$ for linear and multiquadric, $m = k$ for Gaussian). This is achieved by reusing the same LU factorization of the system (5) for each iteration of the cross validation routine. Details of this approach are given in [2].

Automatic model selection is enabled by setting `rbf` to the value `auto`, which is the default choice. When `rbf` = `auto`, RBFOpt will build the surrogate model using thin plate splines until there are enough points to start the automatic model selection procedure. Furthermore, after `max_cross_validations` executions of the automatic model selection procedure, RBFOpt will trust the results obtained up to that point and use the type of RBFs that gave the smallest error the largest number of times. Results for the local search model and global search model are kept separate. In other words, the quantities $\bar{q}_{10\%}$ and $\bar{q}_{70\%}$ are computed at most `max_cross_validations` times; after that, RBFOpt always uses for local search the RBF type that gave the smallest value of $\bar{q}_{10\%}$ the largest number of times out of `max_cross_validations`, and similarly with $\bar{q}_{70\%}$ for global search.

2.5 Parallel optimizer

RBFOpt supports asynchronous parallel evaluation of the objective function f, which is generally the most time-consuming part of the optimization process. To use the parallel optimizer, the parameter `num_cpus` must be set to a number greater than its default value 1. The parallel optimizer is nondeterministic due to its asynchronous nature, hence different results should be expected in different executions, even under the same starting conditions.

The parallel optimizer works by creating a set of worker threads, coordinated by a master. The worker

threads perform tasks of two types: Type 1 is the evaluation of the objective function at a given point (which may take a very long time), Type 2 is the computation of a point at which the objective function should be evaluated (which usually takes only a fraction of a second, but may take a few seconds in rare cases, especially with Gutmann's RBF algorithm). RBFOpt always dedicates one worker to perform tasks of Type 1 or of Type 2 related to the Refinement step (but there is a limit to the fraction of Refinement steps that can be performed in total, see the parameter `refinement_frequency` in Section 3.5); the remaining workers are utilized for the Iteration step (with Gutmann's RBF method or MSRSM). As long as there are available processors, a task is removed from the queue of active tasks, and assigned to the worker. Tasks of Type 1 have priority over Type 2, due to their longer execution times. Within tasks of the same type, a first come, first served policy is used.

We remark that the parallel optimizer may not be as effective as the serial optimizer in terms of performing as few function evaluations as possible, however it may yield large savings in CPU time if the black-box function calls can be run in parallel. A computational evaluation of the parallel optimizer is given in [3]. A brief summary of the results reported in [3] is that when each function evaluation takes between 5 and 10 seconds, the speedup of the parallel optimizer compared to the serial one is roughly the square root of the number of CPUs used.

3 Parameters of RBFOpt

We provide here a list of all the options that can be set in the class `RbfoptSettings`. These options can be provided as arguments to the constructor of the class, for example:

```
settings = RbfoptSettings(max_clock_time=3600, max_evaluations=50)
```

or passed to the command-line interface with the syntax `--option_name=value`, for example:

```
rbfopt_cl_interface.py --max_clock_time 3600 --max_evaluations 50 src/
 examples/rbfopt_black_box_example.py
```

Default values can be checked in the constructor of the class `RbfoptSettings` or by running:

```
rbfopt_cl_interface.py --help
```

3.1 Limits and tolerances

- `max_iterations` : (int) Maximum number of iterations.
- `max_evaluations` : (int) Maximum number of function evaluations in accurate mode, i.e., calls of the black-box function `evaluate()`.
- `max_noisy_evaluations` : (int) Maximum number of function evaluations in noisy mode, i.e., calls of the black-box function `evaluate_noisy()`.
- `max_cycles` : (int) Maximum number of optimization cycles.
- `max_clock_time` : (float) Maximum wall clock time in seconds.
- `target_objval` : (float) The objective function value we want to reach, i.e. the value of the unknown optimum. It can be set to any acceptable value, if the optimum is unknown.
- `eps_opt` : (float) Optimality threshold. Any solution within this relative distance from the `target_objval` is considered optimal.

- `eps_zero` : (float) Tolerance for zeroing out small coefficients in the calculations. Any value smaller than this will be considered zero.
- `eps_impr` : (float) Tolerance for improvement of the objective function. Any improvement in the objective function by less than this amount in absolute and relative terms will be not be considered as finding an improved solution.
- `eps_linear_dependence` : (float) Tolerance to determine if a set of columns/rows is linearly dependent.
- `min_dist` : (float) Minimum Euclidean distance between interpolation points. A new point will be discarded if it is closer than this value from existing nodes. This prevents the RBF matrix, which is based on pairwise distances, from becoming singular.

3.2 Parallel optimization

- `num_cpus` : (int) Number of CPUs used. Any value greater than 1 uses the parallel optimization engine, rather than the default serial engine.
- `parallel_wakeup_time` : (float) Time (in seconds) after which the main optimization engine checks the arrival of results from workers busy with function evaluations or other computations. This parameter is only used by the parallel optimizer.

3.3 Surrogate model and scaling

- `rbf` : (string) Radial basis function used by the method. Choice of 'cubic', 'thin_plate_spline', 'linear', 'multiquadric', 'gaussian', 'auto'. In case of 'auto', the type of rbf and the shape parameter will be dynamically selected by the algorithm.
- `rbf_shape_parameter` : (float) Shape parameter for the radial basis function. Used only by the gaussian and multiquadric RBF, this is also known as the γ parameter. If the rbf is 'auto', this will be automatically selected from a finite set.
- `function_scaling` : (string) Rescaling method for the function values. Choice of 'off', 'affine', 'log', 'auto'.
- `log_scaling_threshold` : (float) Minimum value for the difference between median and minimum function value before a log scaling of the function values is applied in the 'auto' setting.
- `domain_scaling` : (string) Rescaling method for the domain. Choice of 'off', 'affine', 'auto'.
- `dynamism_clipping` : (string) Dynamism clipping strategy. Choice of 'off', 'median', 'clip_at_dyn', 'auto'.
- `dynamism_threshold` : (float) Minimum value of the ratio between the largest and the smallest absolute function values before the dynamism clipping strategy is applied.
- `max_cross_validations` : (int) Maximum number of cross validations before we trust our previous results and stop performing cross- validation.
- `targetval_clipping` : (bool) Clip target value selection based on periodically eliminating some of the largest function values. Used by Gutmann RBF method only.

3.4 Search strategy

- `algorithm` : (string) Optimization algorithm used. Choice of 'Gutmann' and 'MSRSM'.
- `do_infstep` : (bool) If True, perform a pure global search in every cycle of the Iteration step.
- `do_local_search` : (bool) If True, perform a pure local search in every optimization cycle.
- `num_global_searches` : (int) Number of steps in the global search phase.
- `init_strategy` : (string) Strategy to select initial points. Choice of 'all_corners', 'lower_corners', 'rand_corners', 'lhd_maximin', 'lhd_corr'.
- `init_include_midpoint` : (bool) Include midpoint of the box among the initialization points.
- `init_sample_fraction` : (float) The initial sample size is set to $n + 1$ times this number, with some adjustment for parallel optimization based on `init_sample_increase_parallel`. If set to -1 (or any negative number), the size of the initial sample set will be determined automatically. Default -1.
- `init_sample_increase_parallel` : (float) Fraction of increase of the number of initial sample points in order to reduce synchronization efforst in asynchronous parallel evaluation. The number of total initialization points is increase by a factor $(1+$`num_cpus`$*$`init_sample_increase_parallel`$)$, and optimization starts when the originally targetd number of samples is reached. Default 0.05.
- `max_random_init` : (int) Maximum number of trials for the random initialization strategies, in case they generate a linearly dependent set of samples. After this number of trials, the initialization algorithm will bail out.
- `local_search_threshold` : (float) Threshold used to determines what is a local search. If the scaling factor used in the computation of f_k^* is less than this value (this applies to both Gutmann and MSRSM), it is assumed that the search is a local search.
- `local_search_box_scaling` : (float) Rescaling factor for the hyperbox used for local search. If an Iteration step is determined to be a local search, then the search for the next point will be constrained to be within a hyperbox centered on the incumbent, where the size of the hyperbox is determined by the initial hyperbox and this scaling parameters.
- `max_stalled_iterations` : (int) Maximum number of iterations without improvement before we perform a full restart.
- `discarded_window_size` : (int) Number of consecutive iterations that are considered to determine if a restart should be triggered, based on too many discarded points. This number is multiplied by the number of cpus to determine the actual rolling window size.
- `max_fraction_discarded` : (float) Maximum fraction of discarded points within the last `discarded_window_size`*`num_cpus` iterations before a restart is triggered.
- `max_consecutive_restoration` : (int) Maximum number of consecutive nonsingularity restoration phases before the algorithm fails.
- `max_noisy_restarts` : (int) Maximum number of restarts in noisy mode before we switch to accurate mode.

- max_noisy_iterations : (int) Maximum number of iterations in noisy mode before switching to accurate mode.
- global_search_method : (string) The methodology to be used in the solution of global search problems, i.e. the infstep and the global step. The options are 'genetic', 'sampling' and 'solver'. If 'genetic', a heuristic based on a genetic algorithm is used. If 'sampling', random sampling is used. If 'solver', the available solvers are used to try to solve mathematical programming models.
- ga_base_population_size : (int) Minimum population size for the genetic algorithm used to optimize the global search step or infstep, when the genetic global search method is chosen. The final population is computed as the minimum population $+n/5$, where n is the number of decision variables.
- ga_num_generations : (int) Number of generations for the genetic algorithm used to optimize the global search step or infstep, when the genetic global search method is chosen.
- num_samples_aux_problems : (int) Multiplier for the dimension of the problem to determine the number of samples used by the MSRSM algorithm at every iteration.
- modified_msrsm_score : (bool) Use the modified MSRSM score function in which the objective function value contribution always has a weight of 1, instead of $1 - \alpha$.

3.5 Refinement step

- max_consecutive_refinement : (int) Maximum number of consecutive refinement steps.
- thresh_unlimited_refinement : (float) Lower threshold for the amount of search budget depleted, after which the maximum limit on consecutive refinement is ignored. The search budget here is in terms of number of iterations, number of evaluations, wall clock time.
- thresh_unlimited_refinement_stalled : (float) Lower threshold for the percentage of stalled iterations, relative to the maximum number of stalled iterations that will trigger a restart, after which unlimited consecutive refinement are allowed.
- refinement_frequency : (int) In serial search mode, this indicates the number of full global search cycles after which the refinement step can be performed (in case a better solution has been found in the meantime). In parallel mode, this determines the maximum acceptable ratio between other search steps and refinement steps.
- ref_num_integer_candidates : (int) Number of integer candidates per dimension of the problem that are considered when rounding the (fractional) point computed during the refinement step.
- ref_acceptable_decrease_shrink : (float) Maximum ratio between real decrease and refinement model decrease for which the radius of the refinement region gets shrunk.
- ref_acceptable_decrease_enlarge : (float) Minimum ratio between real decrease and refinement model decrease for which the radius of the trust region gets enlarged.
- ref_acceptable_decrease_move : (float) Minimum ratio between real decrease and refinement model decrease for which the new candidate point is accepted as the new iterate.
- ref_min_radius : (float) Minimum radius of the refinement step.

- `ref_init_radius_multiplier` : (float) Exponent (with base 2) of the multiplier used to determine the minimum initial radius of the refinement step.

- `ref_min_grad_norm` : (float) Minimum norm of the gradient for the refinement step, before we assume that we converged to a stationary point.

3.6 Checkpointing

- `save_state_interval` : (int) Number of iterations after which the state of the algorithm should be dumped to file. The algorithm can be resumed from a saved state.

- `save_state_file` : (string) Name of the file in which the state of the algorithm will be saved at regular intervals, see the option above. Default '`rbfopt_algorithm_state.dat`'.

3.7 System-related options

- `print_solver_output` : (bool) If True, print the output of the solvers to screen. Note that this cannot be redirected to file (it is controlled by Pyomo, not by RBFOpt) so it will go to stdout.

- `minlp_solver_path` : (string) Full path to the MINLP solver executable, i.e., bonmin. If only the name solver is specified, it is assumed that the solver is part of your system path and can be called from anywhere.

- `nlp_solver_path` : (string) Full path to the NLP solver executable, i.e., ipopt. If only the name solver is specified, it is assumed that the solver is part of your system path and can be called from anywhere.

- `debug` : (bool) Print debug output. Internal error messages are typically printed to stderr, Pyomo error messages are determined by its logger. If False, all warnings and error messages from RBFOpt are suppressed, and Pyomo loggers are set to print "critical" messages only.

- `rand_seed` : (int) Seed for the random number generator. The maximum random seed supported by NumPy on all platforms is 2^{32}.

References

[1] A. Costa, E. Di Buccio, M. Melucci, and G. Nannicini. Efficient parameter estimation for information retrieval using black-box optimization. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1240–1253, 2018.

[2] A. Costa and G. Nannicini. RBFOpt: an open-source library for black-box optimization with costly function evaluations. *Mathematical Programming Computation*, 10(4):597–629, 2018. `doi:10.1007/s12532-018-0144-7`.

[3] G. I. Diaz, A. Fokoue, G. Nannicini, and H. Samulowitz. An effective algorithm for hyperparameter optimization of neural networks. *IBM Journal of Research and Development*, 61(4/5), 2017.

[4] H.-M. Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19(3):201–227, 2001. `doi:10.1023/A:1011255519438`.

[5] G. Nannicini. On the implementation of a global optimization method for mixed-variable problems. *Open Journal of Mathematical Optimization*, 2, 2021. URL: https://ojmo.centre-mersenne.org/item/OJMO_2021__2__A1_0/, doi:10.5802/ojmo.3.

[6] R. G. Regis and C. A. Shoemaker. A stochastic radial basis function method for the global optimization of expensive functions. *INFORMS Journal on Computing*, 19(4):497–509, 2007. doi:10.1287/ijoc.1060.0182.