RBFOpt Documentation

Release 4.0.2

Giacomo Nannicini

Oct 06, 2018

Contents

Contents:

CHAPTER 1

rbfopt_algorithm module

Main optimization algorithm.

This module contains a class that implements the main optimization algorithm. The class has a state so that optimization can be stopped and resumed at any time.

class rbfopt_algorithm.**RbfoptAlgorithm**(*settings*, *black_box*, *init_node_pos=None*, *init_node_val=None*, *do_init_strategy=True*)

Optimization algorithm.

Implements the main optimization algorithm, and contains all its state variables so that the algorithm can be warm-started from a given state. Some of the logic of the algorithm is not inside this class, because it can be run in parallel and therefore should not modify the state.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

black_box [rbfopt_black_box.BlackBox] An object derived from class BlackBox, that describes the problem.

init_node_pos [2D numpy.ndarray[float] or None] Coordinates of points at which the function value is known. These points will be added to the points generated by the algorithm. If None, all the initial points will be generated by the algorithm.

init_node_val [1D numpy.ndarray[float] or None] Function values corresponding to the points given in init_node_pos. Should be None if the previous argument is None. If init_node_pos is not None but init_node_val is None, the points in init_node_pos will be evaluated in the initialization phase of the algorithm.

do_init_strategy [bool] Perform initialization strategy. If True, the algorithm will generate an initial set of points on top of any user-provided points. If False, we will rely on user-provided points only. Notice that the algorithm may fail if not enough points are provided, and do_init_strategy is False. Default True.

Attributes

elapsed_time [float] Elapsed CPU time up to the point where state was last saved.

best_local_rbf [(string, float)] Best RBF type to construct model for local search, and the corresponding shape parameter.

best_global_rbf [(string, float)] Best RBF type to construct model for global search, and the corresponding shape parameter.

n [int] Dimension of the problem.

itercount [int] Iteration number.

evalcount [int] Total number of function evaluations in accurate mode.

noisy_evalcount [int] Total number of function evaluations in noisy mode.

current_step [int] Identifier of the current step within the cyclic optimization strategy counter.

cyclecount [int] Number of cycles.

num_cons_refinement [int] Current number of consecutive refinement steps.

num_stalled_iter [int] Number of consecutive iterations without improvement.

num_noisy_restarts [int] Number of restarts in noisy mode.

discarded_iters [collections.deque] Rolling window to keep track of discarded iterations

do_init_strategy [bool] If True, perform initialization strategy on first cycle.

inf_step [int] Identifier of the InfStep.

local_search_step [int] Identifier of the LocalSearchStep.

refinement_step [int] Identifier of the Step.

cycle_length [int] Length of an optimization cycle.

restoration_step [int] Identifier of the RestorationStep.

first_step [int] Identifier of the first step of an optimization cycle.

two_phase_optimization [bool] Is the fast but noisy objective function is available?

eval_mode [string] Evaluation mode for the objective function at a given stage. Can be either 'noisy' or 'accurate'.

node_pos [2D numpy.ndarray[float]] Coordinates of the interpolation nodes (i.e. the points where the objective function has already been evaluated). The coordinates may be in the transformed space. This matrix only includes points since the last restart.

node_val [1D numpy.ndarray[float]] Objective function value at the points in node_pos. This array only includes points since the last restart.

node_is_noisy [1D numpy.ndarray[bool]] For each interpolation node in node_pos, was it evaluated in 'noisy' mode?

node_err_bounds [2D numpy.ndarray[float]] The lower and upper variation of the function value for the nodes in node_pos. The variation is assumed 0 for nodes evaluated in accurate mode.

all_node_pos [2D numpy.ndarray[float]] Coordinates of the interpolation nodes. This matrix contains all evaluated points in the original space, and is persistent across restarts.

all_node_val [1D numpy.ndarray[float]] Objective function value at the points in all_node_pos.

all_node_is_noisy [2D numpy.ndarray[bool]] For each interpolation node in all_node_pos, was it evaluated in 'noisy' mode?

all_node_err_bounds [2D numpy.ndarray[float]] The lower and upper variation of the function value for the nodes in all_node_pos. The variation is assumed 0 for nodes evaluated in accurate mode.

num_nodes_at_restart [int] Index of the first new node in all_node_pos after the latest restart.

l_lower [1D numpy.ndarray[float]] Variable lower bounds in the transformed space.

l_upper [1D numpy.ndarray[float]] Variable upper bounds in the transformed space.

fmin_index [int] Index of the minimum value among the nodes since last restart.

fmin [float] Minimum value among the nodes since last restart.

fmax [float] Maximum value among the nodes since last restart.

fmin_stall_check [float] Best function value at the beginning of the most recent optimization cycle.

fmin_last_refine [float] Best function value at the most recent refinement step.

iter_last_refine [int] Iteration number of the most recent refinement step.

fbest_index [int] Index in all_node_pos of the minimum value among all nodes.

fbest [float] Minimum value among all nodes.

best_gap_shown [float] Best gap shown on the log.

is_fmin_noisy [bool] Was the best known objective function since restart evaluated in noisy mode?

is_fbest_noisy [bool] Was the best known objective function evaluated in noisy mode?

fixed_vars [List[(int, float)]] Indices and values of fixed variables. The indices are with respect to the vector of all variables.

tr_model_set [1D numpy.ndarray[int]] Indices of nodes (in node_pos) that are used to build the linear model for the trust region refinement phase.

tr_radius [float] Radius of the trust region.

tr_iterate_index [int] Index of the node (in node_pos) that is the current iterate for the trust region refinement method.

add_node(*point*, *orig_point*, *value*)

Add a node to the all relevant data structures.

Given the data corresponding to a node, add it to all relevant places: the list of current nodes, the list of all nodes. Also, update function minimum and maximum.

Parameters

point [1D numpy.ndarray[float]] Coordinates of the node.

orig_point [1D numpy.ndarray[float]] The point coordinates in the original space.

value [float] Objective function value of the node

add_noisy_node(*point*, *orig_point*, *value*, *err_l*, *err_u*)

Add a noisy node to the all relevant data structures.

Given the data corresponding to a node, add it to all relevant places: the list of current nodes, the list of all nodes. Also, update function minimum and maximum.

Parameters

point [1D numpy.ndarray[float]] Coordinates of the node.

orig_point [1D numpy.ndarray[float]] The point coordinates in the original space.

value [float] Objective function value of the node

err_l [float] Lower variation for the error interval of the node.

err_u [float] Upper variation for the error interval of the node

advance_step_counter()
Advance the step counter of the optimization algorithm.

Advance the step counter of the optimization algorithm, and update cycle number.

classmethod load_from_file(*filename*)
Load object from file, with its state.

Read the current state from file, and return an object of this class. The optimization can be resumed immediately. This function will attempt to set the random number generators to the state they were in. Note that the output stream is set to stdout, regardless of the output stream when the state was saved, so the caller may have to set the desired output stream.

Parameters

filename [string] Name of the file from which the state will be read.

Returns

RbfoptAlgorithm An object of this class.

optimize(*pause_after_iters=9223372036854775807*)
Optimize a black-box function.

Optimize an unknown function over a box using an RBF-based algorithm. This function will select the serial or parallel version of the optimizer, depending on settings.

Parameters

pause_after_iters [int] Number of iterations after which the optimization process should pause. This allows the user to do other activities and resume optimization at a later time. Default sys.maxsize, which is larger than any practical integer.

Returns

(float, 1D numpy.ndarray[float], int, int, int) A quintuple (value, point, itercount, evalcount, noisy_evalcount) containing the objective function value of the best solution found, the corresponding value of the decision variables, the number of iterations of the algorithm, the total number of function evaluations, and the number of these evaluations that were performed in 'noisy' mode.

optimize_parallel(*pause_after_iters=9223372036854775807*)
Optimize a black-box function using parallel evaluations.

Optimize an unknown function over a box using an RBF-based algorithm, using as many CPUs as requested.

Parameters

pause_after_iters [int] Number of iterations after which the optimization process should pause. This allows the user to do other activities and resume optimization at a later time. Default sys.maxsize, which is larger than any practical integer.

optimize_serial(*pause_after_iters=9223372036854775807*)
Optimize a black-box function. Serial engine.

Optimize an unknown function over a box using an RBF-based algorithm. This is the serial version of the optimization routine.

Parameters

pause_after_iters [int] Number of iterations after which the optimization process should pause. Default sys.maxsize.

phase_update()
Check if we should switch phase in two-phase optimization.

Check if we should switch to the second phase of two-phase optimization. The conditions for switching are: 1) Optimization in noisy mode restarted too many times. 2) We reached the limit of noisy mode iterations. If both are met, the switch is performed.

print_init_line()
Print first line of the output, with headers.

print_summary_line(*node_is_noisy*, *gap*)
Print summary line of the algorithm.

Parameters

node_is_noisy [bool] Is the objective function value to be printed associated with a node evaluated in noisy mode?

gap [float] Relative distance from the optimum. This will be multiplied by 100 before printing.

refinement_update(*model_impr*, *real_impr*, *to_replace*)
Perform updates to refinement step and decide if continue.

Update the radius of the trust region and the iterate for the refinement step. Also, decide if the next step should be another refinement step, or if we should go back to global search.

Parameters

model_impr [float] Improvement in quadratic model value.

real_impr [float] Improvement in the real function value.

to_replace [int] Index in tr_model_set of the point to replace.

refinement_update_parallel(*model_impr*, *real_impr*, *to_replace*)
Perform updates to refinement step and decide if continue.

Update the radius of the trust region and the iterate for the refinement step. This is the version that should be used for parallel computation, because the update phase is different.

Parameters

model_impr [float] Improvement in quadratic model value.

real_impr [float] Improvement in the real function value.

to_replace [int] Index in the tr_model_set of the point to replace.

remove_node(*index*, *all_node_shift=0*)
Remove a node from the lists of interpolation nodes.

Given the index of a node, remove its references from all relevant places.

Parameters

index [int] Index of the node to be removed, in the list self.node_pos

all_node_shift [int] A shift that has to be applied to the index to find the corresponding node in self.all_node_pos. Typically, this is the size of self.all_node_pos at the latest restart.

require_accurate_evaluation(*noisy_val*)
Check if a given noisy value qualifies for accurate evaluation.

Verify if a point with the given objective function value in noisy mode qualifies for an immediate accurate re-evaluation.

Parameters

noisy_val [float] Value of the point to be tested, in noisy mode.

Returns

bool True if the point should be re-evaluated in accurate mode immediately.

restart(*pool=None*)
Perform a complete restart of the optimization.

Restart the optimization algorithm, i.e. discard the current RBF model, and select new sample points to start the algorithm from scratch. Previous point evaluations are ignored, but they are still recorded in the appropriate arrays.

Parameters

pool [multiprocessing.Pool()] A pool of workers to evaluate the initialization points in parallel. If None, parallel evaluation will not be performed.

Raises

RuntimeError If the routine does not find enough good initialization points.

restoration_search()
Perform restoration step to repair RBF matrix.

Try to repair an ill-conditioned RBF matrix by selecting points far enough from current interpolation nodes, until numerical stability is restored.

Returns

1D numpy.ndarray[float] or None The next point to be evaluated, or None if it cannot be found.

save_to_file(*filename*)
Save object on file, with its state.

Saves the current state of the algorithm on file. The optimization can be subsequently resumed reading the state from file. This function will also attempt to save the state of the random number generators so that if resumed on the same machine, the optimization process is identical to an uninterrupted process.

Parameters

filename [string] Name of the file that the state will be saved to.

set_output_stream(*output_stream*)
Set output stream for the log.

Parameters

output_stream [file] Stream to be used for output. Must have a 'write' and a 'flush' method.

stalling_update()
Check if the algorithm is stalling.

Check if the algorithm is stalling, and perform the corresponding updates.

unlimited_refinement_active()
Should the usual limits of refinement phase be ignored?

Verify if, based on the unlimited_refinement_thresh parameter, the usual limitations on the refinement phase should be ignored.

Returns

bool True if unlimited refinement is active, False otherwise.

update_log(*tag*, *node_is_noisy=None*, *obj_value=None*, *gap=None*)
Print a single line in the log.

Update the program's log, writing information about an iteration of the optimization algorithm, or a special message.

Parameters

tag [string] Iteration id tag, or unique message if at least one of the other arguments are None.

node_is_noisy [bool or None] Is the objective function value to be printed associated with a node evaluated in noisy mode?

obj_value [float or None] Objective function value to print.

gap [float or None] Relative distance from the optimum. This will be multiplied by 100 before printing.

rbfopt_algorithm.**global_step**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *tfv*, *Amatinv*, *fmin_index*, *current_step*)
Perform global search step.

Perform a global search step, with a different methodology depending on the algorithm chosen.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars: 1D numpy.ndarray[int] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

tfv [(List[float], float, float, List[(float, float)])] Transformed function values: scaled node values, scaled minimum, scaled maximum, and node error bounds.

Amatinv [numpy.matrix or None] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a square numpy.matrix of appropriate dimension. Can be None if algorithm is MSRSM.

fmin_index [int] Index of the minimum value among the nodes.

current_step [int] Identifier of the current step within the cyclic optimization strategy counter.

Returns

1D numpy.ndarray[float] or None The point to be evaluated next, or None if errors occurred.

rbfopt_algorithm.**local_step**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *tfv*, *Amat*, *Amatinv*, *fmin_index*, *two_phase_optimization*, *eval_mode*, *node_is_noisy*)

Perform local search step, possibly adjusted.

Perform a local search step. This typically accepts the minimum of the RBF model as the next point if it is a viable option; if this is not viable, it will perform an adjusted local search and try to generate a different candidate. It also verifies if it is better to evaluate a brand new point, or re-evaluate a previously known point. The test is based on bumpiness of the resulting interpolant.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars: 1D numpy.ndarray[int] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

tfv [(1D numpy.ndarray[float], float, float, 2D numpy.ndarray[float])] Transformed function values: scaled node values, scaled minimum, scaled maximum, scaled node error bounds.

Amat [numpy.matrix] RBF matrix, i.e. [Phi P; P^T 0].

Amatinv [numpy.matrix or None] Inverse of the RBF matrix, i.e. [Phi P; P^T 0]^{-1}. Can be None if the algorithm is MSRSM.

fmin_index [int] Index of the minimum value among the nodes.

two_phase_optimization [bool] Is the noisy but fast objective function is available?

eval_mode [string] Evaluation mode for the objective function at a given stage. Can be either 'noisy' or 'accurate'.

node_is_noisy [List[bool]] For each interpolation node in node_pos, was it evaluated in 'noisy' mode?

Returns

(bool, 1D numpy.ndarray[float] or None, int or None) A triple (adjusted, point, index) where adjusted is True if the local search was adjusted rather than a pure local search, point is the point to be evaluated next (or None if errors occurred), and index is the index that this point should replace, or None if the point should be appended.

See also:

rbfopt_utils.transform_function_values

rbfopt_algorithm.**objfun**(*data*)
Call the evaluate() method of a RbfoptBlackBox object.

Apply the evaluate() method of the given RbfoptBlackBox object to the given point. This way of calling the method indirectly is necessary for parallelization.

Parameters

data [(rbfopt_black_box.RbfoptBlackBox, 1D numpy.ndarray[float],]

List[(int, float)])

A triple or list with three elements (black_box, point, fixed_vars) containing an object derived from class RbfoptBlackBox, that describes the problem, the point at which we want to apply the evaluate() method, and a list of fixed variables given as pairs (index, value).

Returns

float The value of the function evaluate() at the point.

rbfopt_algorithm.**objfun_noisy**(*data*)
Call the evaluate_noisy() method of a RbfoptBlackBox object.

Apply the evaluate_noisy() method of the given RbfoptBlackBox object to the given point. This way of calling the method indirectly is necessary for parallelization.

Parameters

data [(rbfopt_black_box.RbfoptBlackBox, 1D numpy.array[float],]

List[(int, float)])

A triple or list with three elements (black_box, point, fixed_vars) containing an object derived from class RbfoptBlackBox, that describes the problem, the point at which we want to apply the evaluate() method, and a list of fixed variables given as pairs (index, value).

Returns

(float, float, float) The value of the function evaluate_noisy() at the point, and the possible variation given as lower and upper variation.

rbfopt_algorithm.**pure_global_step**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *mat*)
Perform the pure global search step.

Parameters

mat [numpy.matrix] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0]. Must be a square numpy.matrix of appropriate dimension.

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [List[List[float]]] List of coordinates of the nodes

mat [numpy.matrix] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a square numpy.matrix of appropriate dimension. Can be None when using the MSRSM algorithm.

Returns

List[float] or None The point to be evaluated next, or None if errors occurred.

rbfopt_algorithm.**refinement_step**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *tfv*, *h*, *b*, *rank_deficient*, *model_set*, *start_point_index*, *tr_radius*)

Perform a refinement step.

Perform a refinement step to locally improve the solution.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars: 1D numpy.ndarray[int] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

h [1D numpy.ndarray[float]] Linear coefficients of the quadratic model.

b [float] Constant term of the quadratic model.

rank_deficient [bool] True if the points used to build the linear model do not span the space.

model_set [1D numpy.ndarray[int]] Indices of points in node_pos to be used to compute model.

start_point_index [int] Index in node_pos of the starting point for the descent.

tr_radius [float] Radius of the trust region.

Returns

(1D numpy.ndarray[float], float, int) Next candidate point for the search, the corresponding model value difference, and the index in model_set of the point to replace.

CHAPTER 2

rbfopt_aux_problems module

Auxiliary problems for the optimization process.

This module is responsible for constructing and solving all the auxiliary problems encountered during the optimization, such as the minimization of the surrogate model, of the bumpiness. The module acts as an interface between the high-level routines, the low-level PyOmo modules, and the search algorithms.

class rbfopt_aux_problems.**GutmannHkObj**(*settings*, *n*, *k*, *node_pos*, *rbf_lambda*, *rbf_h*, *Amatinv*, *target_val*)

Objective function h_k for the Gutmann method.

This class computes the value of the h_k objective function for the Gutmann method. Lower values are better.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one for each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k. Can be None if dist_weight is equal to 1, in which case RBF values are not used.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1. Can be None if dist_weight is equal to 1, in which case RBF values are not used.

Amatinv [numpy.matrix] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0]. Must be a square numpy.matrix of appropriate dimension.

target_val [float] Value f* that we want to find in the unknown objective function. Used by Gutmann's RBF method only.

evaluate(*points*)
Evaluate the objective for the Gutmann h_k objective.

Compute -1/(mu_k(x) [s_k(x) - f^st]^2)), where s_k is the value of the RBF interpolant, and f^st is the target value. This is because we want to maximize its negative.

Parameters

points [2D numpy.ndarray[float]] Points at which we want to evaluate the objective function (one for each row).

Returns

float The score for the h_k criterion (lower is better).

class rbfopt_aux_problems.**GutmannMukObj**(*settings*, *n*, *k*, *node_pos*, *Amatinv*)
Objective function mu_k for the Gutmann method.

This class computes the value of the mu_k objective function for the Gutmann method. Lower values are better.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one for each row).

Amatinv [numpy.matrix or None] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0]. Must be a square numpy.matrix of appropriate dimension.

evaluate(*points*)
Evaluate the objective for the Gutmann mu objective.

Compute -1/mu_k(x), which we want to minimize.

Parameters

points [2D numpy.ndarray[float]] Points at which we want to evaluate the objective function (one for each row).

Returns

float The score for the mu_k criterion (lower is better).

class rbfopt_aux_problems.**MaximinDistanceObj**(*settings*, *n*, *k*, *node_pos*)
Objective function for the Maximin Distance criterion.

This class facilitates the computation of the objective function for the Maximin Distance criterion. The objective function is the minimum distance from the closest point, multiplied by -1 so that lower values are better (we always minimize).

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one for each row).

evaluate(*points*)
Evaluate the objective for Maximin Distance.

Evaluate the score of a set of points.

Parameters

points [2D numpy.ndarray[float]] Points at which we want to evaluate the objective function (one for each row).

Returns

float The score for Maximin Distance algorithm (lower is better).

class rbfopt_aux_problems.**MetricSRSMObj**(*settings*, *n*, *k*, *node_pos*, *rbf_lambda*, *rbf_h*, *dist_weight*)

Objective function for the Metric SRM method.

This class facilitates the computation of the objective function for the Metric SRSM. The objective function combines the distance from the closest point, and the response surface (i.e. RBF interpolant) value. Lower values are better.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k. Can be None if dist_weight is equal to 1, in which case RBF values are not used.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1. Can be None if dist_weight is equal to 1, in which case RBF values are not used.

dist_weight [float] Relative weight of the distance and objective function value. A weight of 1.0 corresponds to using solely distance, 0.0 to objective function.

evaluate(*points*)

Evaluate the objective for Metric SRSM.

Evaluate the score of a set of points.

Parameters

points [2D numpy.ndarray[float]] Points at which we want to evaluate the objective function (one for each row).

Returns

float The score for the Metric SRSM algorithm (lower is better).

rbfopt_aux_problems.**ga_mate**(*father*, *mother*)

Generate offspring for genetic algorithm.

The offspring will get genes uniformly at random from the mother and the father.

Parameters

father [2D numpy.ndarray[float]] First set of individuals for mating.

mother [2D numpy.ndarray[float]] Second set of individuals for mating.

Returns

2D numpy.ndarray(float) The offspring. Same size as mother and father.

rbfopt_aux_problems.**ga_mutate**(*n*, *var_lower*, *var_upper*, *is_integer*, *individual*, *max_size_pert*)
Mutate an individual (point) for the genetic algorithm.

The mutation is performed in place.

Parameters

n [int] The dimension of the problem, i.e. size of the space.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

is_integer [1D numpy.ndarray[bool]] List of size n, each element is True if the corresponding variable is integer.

individual [1D numpy.ndarray[float]] Point to be mutated.

max_size_pert [int] Maximum size of the perturbation for the mutation, i.e. maximum number of coordinates that can change.

rbfopt_aux_problems.**ga_optimize**(*settings*, *n*, *var_lower*, *var_upper*, *integer_vars*, *objfun*)
Compute and optimize a fitness function.

Use a simple genetic algorithm to quickly find a good solution for a minimization subproblem.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

objfun [Callable[2D numpy.ndarray[float]]] The objective function. This must be a callable function that can be applied to a list of points, and must return a list containing one fitness vale for each point, such that lower values are better.

Returns

1D numpy.ndarray[float] The best solution found.

rbfopt_aux_problems.**generate_sample_points**(*settings*, *n*, *var_lower*, *var_upper*, *integer_vars*, *num_samples*)
Generate sample points uniformly at random.

Generate a given number of points uniformly at random in the bounding box, ensuring that integer variables take on integer values.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

num_samples [int] Number of samples to generate

Returns

2D numpy.ndarray[float] A list of sample points (one for each row).

rbfopt_aux_problems.**get_bump_new_node**(*settings*, *n*, *k*, *node_pos*, *node_val*, *new_node*, *node_err_bounds*, *target_val*)

Compute the bumpiness with a new interpolation point.

Computes the bumpiness of the interpolant obtained by setting a new node in a specified location, at value target_val.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the space where the point lives.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] Location of current interpolation nodes.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

new_node [1D numpy.ndarray[float]] Location of new interpolation node.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

target_val [float] Target function value at which we want to move the node.

Returns

float The bumpiness of the interpolant having a new node at the specified location, with value target_val.

rbfopt_aux_problems.**get_min_bump_node**(*settings*, *n*, *k*, *Amat*, *node_val*, *node_err_bounds*, *target_val*)

Compute the bumpiness obtained by moving an interpolation point.

Compute the bumpiness of the interpolant obtained by moving a single node (the one that yields minimum bumpiness, which is determined by this function) within target_val plus or minus error, to target_val.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the space where the point lives.

k [int] Number of nodes, i.e. interpolation points.

Amat [numpy.matrix] The matrix A = [Phi P; P^T 0] of equation (3) in the paper by Costa and Nannicini.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

target_val [float] Target function value at which we want to move the node.

Returns

(int, float) The index of the node and corresponding bumpiness value indicating the sought node in the list node_pos.

rbfopt_aux_problems.**get_noisy_rbf_coefficients**(*settings*, *n*, *k*, *Phimat*, *Pmat*, *node_val*, *node_err_bounds*, *init_rbf_lambda=None*, *init_rbf_h=None*)

Obtain coefficients for the noisy RBF interpolant.

Solve a quadratic problem to compute the coefficients of the RBF interpolant that minimizes bumpiness and lets all points with deviate by a given amount from their value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

Phimat [numpy.matrix] Matrix Phi, i.e. top left part of the standard RBF matrix.

Pmat [numpy.matrix] Matrix P, i.e. top right part of the standard RBF matrix.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

init_rbf_lambda [1D numpy.ndarray[float] or None] Initial values that should be used for the lambda coefficients of the RBF. Can be None.

init_rbf_h [1D numpy.ndarray[float] or None] Initial values that should be used for the h coefficients of the RBF. Can be None.

Returns

—

(1D numpy.ndarray[float], 1D numpy.ndarray[float]) Two vectors: lambda coefficients (for the radial basis functions), and h coefficients (for the polynomial). If initialization information was provided and was valid, then some values will always be returned. Otherwise, it will be None.

Raises

ValueError If some parameters are not supported.

RuntimeError If the solver cannot be found.

rbfopt_aux_problems.**global_search**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *mat*, *target_val*, *dist_weight*, *fmin*, *fmax*)

Global search that tries to balance exploration/exploitation.

If using Gutmann's RBF method, compute the maximum of the h_k function, see equation (8) in the paper by Costa and Nannicini. If using the Metric SRSM, select a point based on a combination of distance and objective function value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

mat [numpy.matrix or None] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a square numpy.matrix of appropriate dimension, or None if using the MSRSM algorithm.

target_val [float] Value f* that we want to find in the unknown objective function. Used by Gutmann's RBF method only.

dist_weight [float] Relative weight of the distance and objective function value, when selecting the next point with a sampling strategy. A weight of 1.0 corresponds to using solely distance, 0.0 to objective function. Used by Metric SRSM only.

fmin [float] Minimum value among the interpolation nodes.

fmax [float] Maximum value among the interpolation nodes.

Returns

1D numpy.ndarray[float] A local optimum. It is difficult to do global optimization so typically this method returns a local optimum.

Raises

ValueError If some parameters are not supported.

RuntimeError If the solver cannot be found.

rbfopt_aux_problems.**initialize_instance_variables**(*settings*, *instance*, *start_point=None*)

Initialize the variables of a problem instance.

Initialize the x variables of a problem instance, and set the corresponding values for the vectors (u, π). This helps the local search by starting at a feasible point.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

instance [pyomo.ConcreteModel] A concrete instance of mathematical optimization model.

start_point [1D numpy.ndarray[float] or None] The starting point for the local search, or None if it should be randomly generated.

rbfopt_aux_problems.**initialize_msrsm_aux_variables**(*settings*, *instance*)

Initialize auxiliary variables for the MSRSM model.

Initialize the rbfval and mindist variables of a problem instance, using the values for for x and u_pi already given. This helps the local search by starting at a feasible point.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

instance [pyomo.ConcreteModel] A concrete instance of mathematical optimization model.

rbfopt_aux_problems.**minimize_rbf**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *best_node_pos*)

Compute the minimum of the RBF interpolant.

Compute the minimum of the RBF interpolant with a PyOmo model.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

best_node_pos [1D numpy.ndarray[float]] Coordinates of the best interpolation point.

Returns

1D numpy.ndarray[float] A minimizer. It is difficult to do global optimization so typically this method returns a local minimum.

Raises

ValueError If some parameters are not supported.

RuntimeError If the solver cannot be found.

rbfopt_aux_problems.**pure_global_search**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *mat*)

Pure global search that disregards objective function.

If using Gutmann's RBF method, Construct a PyOmo model to maximize :math: *1/mu*. If using the Metric SRM, select a point purely based on distance.

See paper by Costa and Nannicini, equation (7) pag 4, and the references therein.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

mat [numpy.matrix or None] The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a square numpy.matrix of appropriate dimension if given. Can be None when using the MSRSM algorithm.

Returns

List[float] A maximizer. It is difficult to do global optimization so typically this method returns a local maximum.

Raises

ValueError If some parameters are not supported.

RuntimeError If the solver cannot be found.

rbfopt_aux_problems.**set_minlp_solver_options**(*solver*)

Set MINLP solver options.

Set the options of the MINLP solver.

Parameters

solver: pyomo.opt.SolverFactory The solver interface.

rbfopt_aux_problems.**set_nlp_solver_options**(*solver*)

Set NLP solver options.

Set the options of the NLP solver.

Parameters

solver: pyomo.opt.SolverFactory The solver interface.

CHAPTER 3

rbfopt_black_box module

Black-box function.

This module contains the definition of the black box function that is optimized by RBFOpt, when using the default command line interface. This is an abstract class: all methods *must* be reimplemented by the user.

class rbfopt_black_box.**RbfoptBlackBox**

Bases: object

Abstract class for a black-box function that can be optimized.

A class that declares (but does not implement) the necessary methods to describe a black-box function. The user can implement a derived class and use it to compute the function that must be optimized.

evaluate(*x*)

Evaluate the black-box function.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

float Value of the function at x.

evaluate_noisy(*x*)

Evaluate a fast approximation of the black-box function.

Returns an approximation of the value of evaluate(), hopefully much more quickly, and provides 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.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

1D numpy.ndarray[float] 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.

get_dimension()
Return the dimension of the problem.

Returns

int The dimension of the problem.

get_var_lower()
Return the array of lower bounds on the variables.

Returns

1D numpy.ndarray[float] Lower bounds of the decision variables.

get_var_type()
Return the type of each variable.

Returns

1D numpy.ndarray[char] An array of length equal to dimension, specifying the type of each variable. Possible types are 'R' for real (continuous) variables, and 'I' for integer (discrete) variables.

get_var_upper()
Return the array of upper bounds on the variables.

Returns

1D numpy.ndarray[float] Upper bounds of the decision variables.

has_evaluate_noisy()
Indicate whether evaluate_noisy is available.

Indicate if a fast but potentially noisy version of evaluate is available through the function evaluate_noisy. 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.

Returns

bool Is evaluate_noisy available?

CHAPTER 4

rbfopt_degree0_models module

Pyomo models with zero-degree polynomial for RBFOpt.

This module creates all the auxiliary problems that rely on zero-degree polynomials. The models are created and instantiated using Pyomo. This module does not solve the problems.

rbfopt_degree0_models.**add_integrality_constraints**(*model*, *integer_vars*)

Add integrality constraints to the model.

Add integrality constraints to the model by introducing extra variables.

Parameters

model [pyomo.ConcreteModel] The model to which we want to add integrality constraints.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

rbfopt_degree0_models.**create_max_h_k_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *mat*, *target_val*)

Create the abstract model to maximize h_k.

Create the abstract model to maximize h_k, also known as the Global Search Step of the RBF method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

target_val [float] Value f* that we want to find in the unknown objective function.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree0_models.**create_max_one_over_mu_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *mat*)

Create the concrete model to maximize 1/mu.

Create the concrete model to maximize :math: *1/mu*, also known as the InfStep of the RBF method.

See paper by Costa and Nannicini, equation (7) pag 4, and the references therein.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree0_models.**create_maximin_dist_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*)

Create the concrete model to maximize the minimum distance.

Create the concrete model to maximize the minimum distance to the interpolation nodes, which is the infstep of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree0_models.**create_min_bump_model**(*settings*, *n*, *k*, *Phimat*, *Pmat*, *node_val*, *node_err_bounds*)

Create a model to find RBF coefficients with min bumpiness.

Create a quadratic problem to compute the coefficients of the RBF interpolant that minimizes bumpiness and lets all points deviate by a specified amount from their value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

Phimat [numpy.matrix] Matrix Phi, i.e. top left part of the standard RBF matrix.

Pmat [numpy.matrix] Matrix P, i.e. top right part of the standard RBF matrix.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree0_models.**create_min_msrsm_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *dist_weight*, *dist_min*, *dist_max*, *fmin*, *fmax*)

Create the concrete model to optimize the MSRSM objective.

Create the concreate model to minimize a weighted combination of the value of the RBF interpolant and the (negative of the) distance from the closes interpolation node. This is the Global Search Step of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

dist_weight [float] The weight paramater for distance and RBF interpolant value. Must be between 0 and 1. A weight of 1.0 corresponds to using solely distance, 0.0 to objective function.

dist_min [float] The minimum distance between two interpolation nodes.

dist_max [float] The maximum distance between two interpolation nodes.

fmin [float] The minimum value of an interpolation node.

fmax [float] The maximum value of an interpolation node.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree0_models.**create_min_rbf_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*)

Create the concrete model to minimize the RBF.

Create the concrete model to minimize the RBF.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

CHAPTER 5

rbfopt_degree1_models module

Pyomo models with degree-one polynomial for RBFOpt.

This module creates all the auxiliary problems that rely on degree-one polynomials. The models are created and instantiated using Pyomo. This module does not solve the problems.

rbfopt_degree1_models.**add_integrality_constraints**(*model*, *integer_vars*)

Add integrality constraints to the model.

Add integrality constraints to the model by introducing extra variables.

Parameters

model [pyomo.ConcreteModel] The model to which we want to add integrality constraints.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

rbfopt_degree1_models.**create_max_h_k_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *mat*, *target_val*)

Create the concrete model to maximize h_k.

Create the concrete model to maximize h_k, also known as the Global Search Step of the RBF method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

target_val [float] Value f* that we want to find in the unknown objective function.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree1_models.**create_max_one_over_mu_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *mat*)

Create the concrete model to maximize 1/mu.

Create the concrete model to maximize :math: *1/mu*, also known as the InfStep of the RBF method.

See paper by Costa and Nannicini, equation (7) pag 4, and the references therein.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree1_models.**create_maximin_dist_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*)

Create the concrete model to maximize the minimum distance.

Create the concrete model to maximize the minimum distance to the interpolation nodes, which is the infstep of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree1_models.**create_min_bump_model**(*settings*, *n*, *k*, *Phimat*, *Pmat*, *node_val*, *node_err_bounds*)

Create a model to find RBF coefficients with min bumpiness.

Create a quadratic problem to compute the coefficients of the RBF interpolant that minimizes bumpiness and lets all points deviate by a specified amount from their value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

Phimat [numpy.matrix] Matrix Phi, i.e. top left part of the standard RBF matrix.

Pmat [numpy.matrix] Matrix P, i.e. top right part of the standard RBF matrix.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree1_models.**create_min_msrsm_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *dist_weight*, *dist_min*, *dist_max*, *fmin*, *fmax*)

Create the concrete model to optimize the MSRSM objective.

Create the concreate model to minimize a weighted combination of the value of the RBF interpolant and the (negative of the) distance from the closes interpolation node. This is the Global Search Step of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

dist_weight [float] The weight paramater for distance and RBF interpolant value. Must be between 0 and 1. A weight of 1.0 corresponds to using solely distance, 0.0 to objective function.

dist_min [float] The minimum distance between two interpolation nodes.

dist_max [float] The maximum distance between two interpolation nodes.

fmin [float] The minimum value of an interpolation node.

fmax [float] The maximum value of an interpolation node.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degree1_models.**create_min_rbf_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*)

Create the concrete model to minimize the RBF.

Create the concrete model to minimize the RBF.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension n+1.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

CHAPTER 6

rbfopt_degreem1_models module

Pyomo models with no polynomial (degree -1) for RBFOpt.

This module creates all the auxiliary problems that do not rely on polynomials. The models are created and instantiated using Pyomo. This module does not solve the problems.

rbfopt_degreem1_models.**add_integrality_constraints**(*model*, *integer_vars*)

Add integrality constraints to the model.

Add integrality constraints to the model by introducing extra variables.

Parameters

model [pyomo.ConcreteModel] The model to which we want to add integrality constraints.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

rbfopt_degreem1_models.**create_max_h_k_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *mat*, *target_val*)

Create the concrete model to maximize h_k.

Create the concrete model to maximize h_k, also known as the Global Search Step of the RBF method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension 0.

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

target_val [float] Value f* that we want to find in the unknown objective function.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degreem1_models.**create_max_one_over_mu_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *mat*)

Create the concrete model to maximize 1/mu.

Create the concrete model to maximize :math: *1/mu*, also known as the InfStep of the RBF method.

See paper by Costa and Nannicini, equation (7) pag 4, and the references therein.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

mat: numpy.matrix The matrix necessary for the computation. This is the inverse of the matrix [Phi P; P^T 0], see paper as cited above. Must be a numpy.matrix of dimension ((k+1) x (k+1))

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degreem1_models.**create_maximin_dist_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*)

Create the concrete model to maximize the minimum distance.

Create the concrete model to maximize the minimum distance to the interpolation nodes, which is the infstep of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degreem1_models.**create_min_bump_model**(*settings*, *n*, *k*, *Phimat*, *Pmat*, *node_val*, *node_err_bounds*)

Create a model to find RBF coefficients with min bumpiness.

Create a quadratic problem to compute the coefficients of the RBF interpolant that minimizes bumpiness and lets all points deviate by a specified amount from their value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

Phimat [numpy.matrix] Matrix Phi, i.e. top left part of the standard RBF matrix.

Pmat [numpy.matrix] Matrix P, i.e. top right part of the standard RBF matrix. Empty in this case.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

node_err_bounds [2D numpy.ndarray[float]] Allowed deviation from node values for nodes affected by error. This is a matrix with rows (lower_deviation, upper_deviation).

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degreem1_models.**create_min_msrsm_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*, *dist_weight*, *dist_min*, *dist_max*, *fmin*, *fmax*)

Create the concrete model to optimize the MSRSM objective.

Create the concreate model to minimize a weighted combination of the value of the RBF interpolant and the (negative of the) distance from the closes interpolation node. This is the Global Search Step of the MSRSM method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension 0.

dist_weight [float] The weight paramater for distance and RBF interpolant value. Must be between 0 and 1. A weight of 1.0 corresponds to using solely distance, 0.0 to objective function.

dist_min [float] The minimum distance between two interpolation nodes.

dist_max [float] The maximum distance between two interpolation nodes.

fmin [float] The minimum value of an interpolation node.

fmax [float] The maximum value of an interpolation node.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

rbfopt_degreem1_models.**create_min_rbf_model**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *integer_vars*, *node_pos*, *rbf_lambda*, *rbf_h*)

Create the concrete model to minimize the RBF.

Create the concrete model to minimize the RBF.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] The dimension of the problem, i.e. size of the space.

k [int] Number of nodes, i.e. interpolation points.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] List of indices of integer variables.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes (one on each row).

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to the polynomial. List of dimension 0.

Returns

pyomo.ConcreteModel The concrete model describing the problem.

CHAPTER 7

rbfopt_refinement module

Routines for trust-region based local search.

This module contains all functions that are necessary to implement a trust-region based local search to refine the solution quality. The local search exploits a linear model of the objective function.

rbfopt_refinement.**get_candidate_point**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *h*, *start_point*, *tr_radius*)

Compute the next candidate point of the trust region method.

Starting from a given point, compute a descent direction and move in that direction to find the point with lowest value of the linear model, within the radius of the trust region.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

h [1D numpy.ndarray[float]] Linear coefficients of the quadratic model.

start_point [1D numpy.ndarray[float]] Starting point for the descent.

tr_radius [float] Radius of the trust region.

Returns

(1D numpy.ndarray[float], float, float) Next candidate point for the search, the corresponding model value difference, and the norm of the gradient at the current point.

rbfopt_refinement.**get_integer_candidate**(*settings*, *n*, *k*, *h*, *start_point*, *tr_radius*, *candidate*, *integer_vars*)

Get integer candidate point from a fractional point.

Look for integer points around the given fractional point, trying to find one with a good value of the quadratic model.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

h [1D numpy.ndarray[float]] Linear coefficients of the model.

start_point [1D numpy.ndarray[float]] Starting point for the descent.

tr_radius [float] Radius of the trust region.

candidate [1D numpy.ndarray[float]] Fractional point to being the search.

integer_vars [1D numpy.ndarray[int]] Indices of the integer variables.

Returns

(1D numpy.ndarray[float], float) Next candidate point for the search, and the corresponding change in model value compared to the given point.

rbfopt_refinement.**get_linear_model**(*settings*, *n*, *k*, *node_pos*, *node_val*, *model_set*)

Compute a linear model of the function.

Determine a linear model h^T x + b of the objective function in an area that is centered on the given node. The model is computed by solving a (not necessarily square) linear system, inducing sparsity.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

model_set [1D numpy.ndarray[int]] Indices of points in node_pos to be used to compute model.

Returns

1D numpy.ndarray[float], float, bool Coefficients of the linear model h, b, and a boolean indicating if the linear model is underdetermined.

Raises

numpy.linalg.LinAlgError If the matrix cannot be computed for numerical reasons.

rbfopt_refinement.**get_model_improving_point**(*settings*, *n*, *k*, *var_lower*, *var_upper*, *node_pos*, *model_set*, *start_point_index*, *tr_radius*, *integer_vars*)

Compute a point to improve the model used in the trust region.

Determine a point that improves the geometry of the set of points used to build the trust region model. This point may not have a good objective function value, but it ensures that the model is well behaved.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

model_set [1D numpy.ndarray[int]] Indices of points in node_pos to be used to compute model.

start_point_index [int] Index in node_pos of the starting point for the descent.

tr_radius [float] Radius of the trust region.

integer_vars [1D numpy.ndarray[int]] Indices of the integer variables.

Returns

(1D numpy.ndarray[float], bool, int) Next candidate point to improve the model, a boolean indicating success, and the index of the point to replace if successful.

rbfopt_refinement.**init_trust_region**(*settings*, *n*, *k*, *node_pos*, *center*)

Initialize the trust region.

Determine which nodes should be used to create a linear model of the objective function, and determine the initial radius of the trust region.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

center [1D numpy.ndarray[float]] Node that acts as a center for the quadratic model.

Returns

(1D numpy.ndarray[int], float) Indices in node_pos of points to build the model, and initial radius of the trust region.

Raises

numpy.linalg.LinAlgError If the matrix cannot be computed for numerical reasons.

rbfopt_refinement.**update_trust_region_radius**(*settings*, *tr_radius*, *model_obj_diff*, *real_obj_diff*)

Update the radius of the trust region.

Compute the updated trust region radius based on the true objective function difference between the old point and the new point, and that of the quadratic model. Also, determine if the new iterate should be accepted.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

tr_radius [float] Current radius of the trust region.

model_obj_diff [float] Difference in objective function value of the new point and the previous point, according to the linear model.

real_obj_diff [float] Difference in the real objective function value of the new point and the previous point.

Returns

(float, bool) Updated radius of the trust region, and whether the point should be accepted.

CHAPTER 8

rbfopt_settings module

Settings for RBFOpt.

This module contains the settings of the main optimization algorithm.

class rbfopt_settings.**RbfoptSettings**(*max_iterations=1000*, *max_evaluations=300*, *max_noisy_evaluations=200*, *max_cycles=1000*, *max_clock_time=1e+30*, *num_cpus=1*, *parallel_wakeup_time=0.1*, *algorithm='MSRSM'*, *rbf='auto'*, *rbf_shape_parameter=0.1*, *target_objval=-10000000000.0*, *eps_opt=0.01*, *eps_zero=1e-15*, *eps_impr=0.0001*, *eps_linear_dependence=1e-06*, *min_dist=1e-05*, *do_infstep=False*, *num_global_searches=5*, *init_strategy='lhd_maximin'*, *max_random_init=50*, *function_scaling='auto'*, *log_scaling_threshold=1000000.0*, *domain_scaling='auto'*, *dynamism_clipping='auto'*, *dynamism_threshold=1000.0*, *local_search_threshold=0.25*, *local_search_box_scaling=0.5*, *max_stalled_iterations=100*, *discarded_window_size=30*, *max_fraction_discarded=0.5*, *max_consecutive_restoration=15*, *max_cross_validations=50*, *max_noisy_restarts=2*, *max_noisy_iterations=200*, *targetval_clipping=True*, *global_search_method='genetic'*, *ga_base_population_size=400*, *ga_num_generations=20*, *num_samples_aux_problems=1000*, *modified_msrsm_score=True*, *max_consecutive_refinement=5*, *thresh_unlimited_refinement=0.9*, *thresh_unlimited_refinement_stalled=0.9*, *refinement_frequency=3*, *tr_num_integer_candidates=10*, *tr_acceptable_decrease_shrink=0.2*, *tr_acceptable_decrease_enlarge=0.6*, *tr_acceptable_decrease_move=0.1*, *tr_min_radius=0.001*, *tr_init_radius_multiplier=2.0*, *tr_min_grad_norm=0.01*, *print_solver_output=False*, *save_state_interval=100000*, *save_state_file='rbfopt_algorithm_state.dat'*, *minlp_solver_path='bonmin'*, *nlp_solver_path='ipopt'*, *debug=False*, *rand_seed=937627691*)

Global and algorithmic settings for RBF method.

Class containing algorithmic settings for the enhanced RBF method, as well as global settings such as tolerances, limits and so on.

NOTICE: The docstring for the parameters below is used to build the help in the command-line interface. It is therefore very important that it is kept tidy in the correct numpy docstring format.

Parameters

max_iterations [int] Maximum number of iterations. Default 1000.

max_evaluations [int] Maximum number of function evaluations in accurate mode. Default 300.

max_noisy_evaluations [int] Maximum number of function evaluations in noisy mode. Default 200.

max_cycles [int] Maximum number of optimization cycles. Default 1000.

max_clock_time [float] Maximum wall clock time in seconds. Default 1.0e30.

algorithm [string] Optimization algorithm used. Choice of 'Gutmann' and 'MSRSM', see References Gutmann (2001) and Regis and Shoemaker (2007). Default 'MSRSM'.

num_cpus [int] Number of CPUs used. Default 1.

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. Default 0.1.

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. Default 'auto'.

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 gamma parameter. If the rbf is 'auto', this will be automatically selected from a finite set. Default 0.1.

target_objval [float] The objective function value we want to reach, i.e. the value of the unknown optimum. It can be set to an acceptable value, if the optimum is unknown. Default -1.0e10.

eps_opt [float] Optimality threshold. Any solution within this relative distance from the target_objval is considered optimal. Default 1.0e-2.

eps_zero [float] Tolerance for zeroing out small coefficients in the calculations. Any value smaller than this will be considered zero. Default 1.0e-15.

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 ignored. Default 1.0e-4.

eps_linear_dependence [float] Tolerance to determine if a set of columns/rows is linearly dependent. Default 1.0e-6.

min_dist [float] Minimum Euclidean distance between nodes. A new point will be discarded if it is closer than this value from existing nodes. This prevents the RBF matrix, which depends on pairwise distances, from becoming singular. Default 1.0e-5.

do_infstep [bool] Perform a pure global search in every optimization cycle. Default False.

num_global_searches [int] Number of steps in the global search phase. Default 5.

init_strategy [string] Strategy to select initial points. Choice of 'all_corners', 'lower_corners', 'rand_corners', 'lhd_maximin', 'lhd_corr'. Default 'lhd_maximin'.

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. Default 50.

function_scaling [string] Rescaling method for the function values. Choice of 'off', 'affine', 'log', 'auto'. Default '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. Default 1.0e6.

domain_scaling [string] Rescaling method for the domain. Choice of 'off', 'affine', 'auto'. Default 'auto'.

dynamism_clipping [string] Dynamism clipping strategy. Choice of 'off', 'median', 'clip_at_dyn', 'auto'. Default '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. Default 1.0e3.

local_search_threshold [float] Threshold used to determines what is a local search. If the scaling factor used in the computation of f_n^* is less than this value, it is assumed that the search is a local search. Default 0.25.

local_search_box_scaling [float] Rescaling factor for the hyperbox used for local search. See parameter nu of Regis and Shoemaker (2007). Default 0.5.

max_stalled_iterations [int] Maximum number of iterations without improvement before we perform a full restart. Default 100.

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. Default 30.

max_fraction_discarded [float] Maximum fraction of discarded points within the last discarded_window_size*num_cpus iterations before a restart is triggered. Default 0.5.

max_consecutive_restoration [int] Maximum number of consecutive nonsingularity restoration phases before the algorithm fails. Default 15.

max_cross_validations [int] Maximum number of cross validations before we trust our previous results. Default 50.

max_noisy_restarts [int] Maximum number of restarts in noisy mode before we switch to accurate mode. Default 2.

max_noisy_iterations [int] Maximum number of iterations in noisy mode before switching to accurate mode. Default 200.

targetval_clipping [bool] Clip target value selection based on periodically eliminating some of the largest function values, as proposed by Gutmann (2001) and later Regis and Shoemaker (2007). Used by Gutmann RBF method only. Default True.

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. Default 'genetic'.

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. Default 400.

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. Default 20.

num_samples_aux_problems [int] Multiplier for the dimension of the problem to determine the number of samples used by the Metric SRSM traditional algorithm at every iteration. Default 1000.

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 - distance_weight. This setting is more aggressive in improving the objective function value, compared to the original MSRSM score function. Default True.

max_consecutive_refinement [int] Maximum number of consecutive refinement steps. Default 5.

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. Default 0.9.

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. Default 0.9.

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. Default 3.

tr_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. Default 10.

tr_acceptable_decrease_shrink [float] Maximum ratio between real decrease and trust region model decrease for which the radius of the trust region gets shrunk. Default 0.2.

tr_acceptable_decrease_enlarge [float] Minimum ratio between real decrease and trust region model decrease for which the radius of the trust region gets enlarged. Default 0.6.

tr_acceptable_decrease_move [float] Minimum ratio between real decrease and trust region model decrease for which the new candidate point is accepted as the new iterate. Default 0.1.

tr_min_radius [float] Minimum radius of the trust region for the refinement step. Default 1.0e-3.

tr_init_radius_multiplier [float] Exponent (with base 2) of the multiplier used to determine the minimum initial radius of the trust region for the refinement step. Default 2.0.

tr_min_grad_norm [float] Minimum norm of the gradient for the trust region method in the refinement step, before we assume that we converged to a stationary point. Default 1.0e-2.

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. It can be useful in case something goes wrong. Default 100000.

save_state_file [string] Name of the file in which the state of the algorithm will be saved at regular intervals, see save_state_interval. Default 'rbfopt_algorithm_state.dat'.

print_solver_output [bool] Print the output of the solvers to screen? Note that this cannot be redirected to file so it will go to stdout. Default False.

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. Default 'bonmin'.

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. Default 'ipopt'.

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 are suppressed. Default False.

rand_seed [int] Seed for the random number generator. The maximum number supported by numpy on all platforms is 2^32. Default 937627691.

Attributes

_allowed_rbf [Dict[string]] Allowed types of RBF functions.

_allowed_init_strategy [Dict[string]] Allowed initialization strategies.

_allowed_function_scaling [Dict[string]] Allowed function scaling strategies.

_allowed_domain_scaling [Dict[string]] Allowed domain scaling strategies.

_allowed_dynamism_clipping [Dict[string]] Allowed dynamism clipping strategies.

_allowed_algorithm [Dict[string]] Allowed algorithms.

_allowed_global_search_method [Dict[string]] Allowed global search methods.

classmethod from_dictionary(*args*)

Construct settings from dictionary containing parameter values.

Construct an instance of RbfoptSettings by looking up the value of the parameters from a given dictionary. The dictionary must contain only parameter values in the form args['name'] = value. Anything else present in the dictionary will raise an exception.

Parameters

args [Dict[string]] A dictionary containing the values of the parameters in a format args['name'] = value.

Returns

RbfoptSettings An instance of the object of the class.

Raises

ValueError If the dictionary contains invalid parameters.

print(*output_stream*)

Print the value of all settings.

Prints the settings to the output stream, on a very long line.

Parameters

output_stream [file] The stream on which messages are printed.

set_auto_parameters(*dimension*, *var_lower*, *var_upper*, *integer_vars*)

Determine the value for 'auto' parameters.

Create a copy of the settings and assign 'auto' parameters. The original settings are untouched.

Parameters

dimension [int] The dimension of the problem, i.e. size of the space.

var_lower [1D numpy.ndarray[float]] Vector of variable lower bounds.

var_upper [1D numpy.ndarray[float]] Vector of variable upper bounds.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty list, all variables are assumed to be continuous.

Returns

RbfoptSettings A copy of the settings, without any 'auto' parameter values.

CHAPTER 9

rbfopt_test_functions module

Test functions.

This module implements several known mathematical functions, that can be used to test RBFOpt.

class rbfopt_test_functions.**TestBlackBox**(*name*)

Bases: rbfopt.rbfopt_black_box.RbfoptBlackBox

A black-box constructed from a known test function.

Parameters

name [string] The name of the function to be implemented.

evaluate(*point*)

Evaluate the black-box function.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

float Value of the function at x.

evaluate_noisy(*point*)

Evaluate a fast approximation of the black-box function.

Returns an approximation of the value of evaluate(), hopefully much more quickly, and provides 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.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

1D numpy.ndarray[float] 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.

get_dimension()
Return the dimension of the problem.

Returns

int The dimension of the problem.

get_var_lower()
Return the array of lower bounds on the variables.

Returns

1D numpy.ndarray[float] Lower bounds of the decision variables.

get_var_type()
Return the type of each variable.

Returns

1D numpy.ndarray[char] An array of length equal to dimension, specifying the type of each variable. Possible types are 'R' for real (continuous) variables, and 'I' for integer (discrete) variables.

get_var_upper()
Return the array of upper bounds on the variables.

Returns

1D numpy.ndarray[float] Upper bounds of the decision variables.

has_evaluate_noisy()
Indicate whether evaluate_noisy is available.

Indicate if a fast but potentially noisy version of evaluate is available through the function evaluate_noisy. 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.

Returns

bool Is evaluate_noisy available?

class rbfopt_test_functions.**TestNoisyBlackBox**(*name*, *max_rel_error=0.1*, *max_abs_error=0.1*)
Bases: rbfopt.rbfopt_black_box.RbfoptBlackBox

A noisy black-box constructed from a known test function.

Parameters

name [string] The name of the function to be implemented.

max_rel_error: float Maximum relative error.

max_abs_error: float Maximum absolute error.

evaluate(*point*)
Evaluate the black-box function.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

float Value of the function at x.

evaluate_noisy(*point*)
Evaluate a fast approximation of the black-box function.

Returns an approximation of the value of evaluate(), hopefully much more quickly, and provides 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.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

1D numpy.ndarray[float] 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.

get_dimension()
Return the dimension of the problem.

Returns

int The dimension of the problem.

get_var_lower()
Return the array of lower bounds on the variables.

Returns

1D numpy.ndarray[float] Lower bounds of the decision variables.

get_var_type()
Return the type of each variable.

Returns

1D numpy.ndarray[char] An array of length equal to dimension, specifying the type of each variable. Possible types are 'R' for real (continuous) variables, and 'I' for integer (discrete) variables.

get_var_upper()
Return the array of upper bounds on the variables.

Returns

1D numpy.ndarray[float] Upper bounds of the decision variables.

has_evaluate_noisy()
Indicate whether evaluate_noisy is available.

Indicate if a fast but potentially noisy version of evaluate is available through the function evaluate_noisy. 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.

Returns

bool Is evaluate_noisy available?

class rbfopt_test_functions.**branin**
Branin function of the Dixon-Szego test set.

additional_optima = <Mock name='mock.array()' id='139727845492176'>

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.397887357729739

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**camel**
Six-hump Camel function of the Dixon-Szego test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1.0316284535

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**ex4_1_1**
ex4_1_1 function of the GlobalLib test set.

dimension = 1

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -7.4873

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**ex4_1_2**
ex4_1_2 function of the GlobalLib test set.

dimension = 1

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -663.4993631230575

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**ex8_1_1**
ex8_1_1 function of the GlobalLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -2.02181

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**ex8_1_4**
ex8_1_4 function of the GlobalLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**gear**
gear function of the MINLPLib test set.

dimension = 4

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**gear4**
gear4 function of the MINLPLib test set.

dimension = 5

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 1.643428

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**goldsteinprice**
Goldstein & Price function of the Dixon-Szego test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 3

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**hartman3**
Hartman3 function of the Dixon-Szego test set.

dimension = 3

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -3.86278214782076

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**hartman6**
Hartman6 function of the Dixon-Szego test set.

dimension = 6

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -3.32236801141551

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**least**
least function of the GlobalLib test set.

dimension = 3

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 14085.1398

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs02**
nvs02 function of the MINLPLib test set.

dimension = 5

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -5.964184

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs03**
nvs03 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 16.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs04**
nvs04 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.72

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs06**
nvs06 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 1.7703125

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs07**
nvs07 function of the MINLPLib test set.

dimension = 3

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs09**
nvs09 function of the MINLPLib test set.

dimension = 10

static evaluate(*x*)

i = 9

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -43.134336918035

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs14**
nvs14 function of the MINLPLib test set.

dimension = 5

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -40358.15477

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs15**
nvs15 function of the MINLPLib test set.

dimension = 3

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 1.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**nvs16**
nvs16 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.703125

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**perm0_8**
perm0 function of dimension 8 from Arnold Neumaier. http://www.mat.univie.ac.at/~neum/glopt/my_problems.html We use parameters (8, 100) here.

dimension = 8

static evaluate(*x*)

i = 7

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 1000.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**perm_6**
perm function of dimension 6 from Arnold Neumaier. http://www.mat.univie.ac.at/~neum/glopt/my_problems.html We use parameters (6, 60) here.

dimension = 6

static evaluate(*x*)

i = 5

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 1000.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**prob03**
prob03 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 10.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**rbrock**
rbrock function of the GlobalLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schaeffer_f7_12_1**

Schaeffer F7 function.

dimension = 12

static evaluate(*x*)

i = 11

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -10

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schaeffer_f7_12_2**

Schaeffer F7 function.

dimension = 12

static evaluate(*x*)

i = 11

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 10

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_10_1**

schoen function of dimension 10 with 50 stationary points.

dimension = 10

static evaluate(*x*)

i = 9

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_10_1_int**
schoen function of dimension 10 with 50 stationary points.

Mixed integer version.

dimension = 10

static evaluate(*x*)

i = 9

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_10_2**
schoen function of dimension 10 with 50 stationary points.

dimension = 10

static evaluate(*x*)

i = 9

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_10_2_int**
schoen function of dimension 10 with 50 stationary points.

Mixed integer version.

dimension = 10

static evaluate(*x*)

i = 9

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_6_1**
schoen function of dimension 6 with 50 stationary points.

dimension = 6

static evaluate(*x*)

i = 5

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_6_1_int**
schoen function of dimension 6 with 50 stationary points.

Mixed integer version.

dimension = 6

static evaluate(*x*)

i = 5

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_6_2**
schoen function of dimension 6 with 50 stationary points.

dimension = 6

static evaluate(*x*)

i = 5

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**schoen_6_2_int**
schoen function of dimension 6 with 50 stationary points.

Mixed integer version.

dimension = 6

static evaluate(*x*)

i = 5

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -1000

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**shekel10**
Shekel10 function of the Dixon-Szego test set.

dimension = 4

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -10.536409816692

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**shekel5**
Shekel5 function of the Dixon-Szego test set.

dimension = 4

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -10.1531996790582

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**shekel7**
Shekel7 function of the Dixon-Szego test set.

dimension = 4

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -10.4029405668187

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**sporttournament06**
sporttournament06 function of the MINLPLib test set.

dimension = 15

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -12.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**st_miqp1**
st_miqp1 function of the MINLPLib test set.

dimension = 5

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 281.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**st_miqp3**

st_miqp3 function of the MINLPLib test set.

dimension = 2

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = -6.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

class rbfopt_test_functions.**st_test1**

st_test1 function of the MINLPLib test set.

dimension = 5

static evaluate(*x*)

optimum_point = <Mock name='mock.array()' id='139727845492176'>

optimum_value = 0.0

var_lower = <Mock name='mock.array()' id='139727845492176'>

var_type = <Mock name='mock.array()' id='139727845492176'>

var_upper = <Mock name='mock.array()' id='139727845492176'>

CHAPTER 10

rbfopt_user_black_box module

Black-box function from user data.

This module contains the definition of a black box function constructed from user data that can be optimized by RBFOpt.

Licensed under Revised BSD license, see LICENSE. (C) Copyright International Business Machines Corporation 2017.

class rbfopt_user_black_box.**RbfoptUserBlackBox**(*dimension*, *var_lower*, *var_upper*, *var_type*, *obj_funct*, *obj_funct_noisy=None*)

Bases: rbfopt.rbfopt_black_box.RbfoptBlackBox

A black-box function from user data that can be optimized.

A class that implements the necessary methods to describe the black-box function to be minimized, and gets all the required data from the user.

Parameters

dimension [int] Dimension of the problem.

var_lower [1D numpy.ndarray[float]] Lower bounds of the decision variables.

var_upper [1D numpy.ndarray[float]] Upper bounds of the decision variables.

var_type [1D numpy.ndarray[char]] An array of length equal to dimension, specifying the type of each variable. Possible types are 'R' for real (continuous) variables, and 'I' for integer (discrete) variables.

obj_funct [Callable[1D numpy.ndarray[float]]] The function to optimize. Must take a numpy array as argument, and return a float.

obj_funct_noisy [Callable[1D numpy.ndarray[float]] or None] The noisy but fast version of the function to optimize. If given, it must take a numpy array as argument, and return a numpy array with three floats, in the following order: the approximate function value, its lower variation, and its upper variation, where where lower <= 0 and upper >= 0 and the

true function value is contained between value + lower and value + upper. If it is None, we assume that there is no fast version of the objective function.

See also:

`rbfopt_black_box.BlackBox`

evaluate(*x*)
Evaluate the black-box function.

Parameters

x [List[float]] Value of the decision variables.

Returns

float Value of the function at x.

evaluate_noisy(*x*)
Evaluate a fast approximation of the black-box function.

Returns an approximation of the value of evaluate(), hopefully much more quickly, and provides 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.

Parameters

x [1D numpy.ndarray[float]] Value of the decision variables.

Returns

1D numpy.ndarray[float] 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.

get_dimension()
Return the dimension of the problem.

Returns

int The dimension of the problem.

get_var_lower()
Return the array of lower bounds on the variables.

Returns

List[float] Lower bounds of the decision variables.

get_var_type()
Return the type of each variable.

Returns

1D numpy.ndarray[char] An array of length equal to dimension, specifying the type of each variable. Possible types are 'R' for real (continuous) variables, and 'I' for integer (discrete) variables.

get_var_upper()
Return the array of upper bounds on the variables.

Returns

List[float] Upper bounds of the decision variables.

has_evaluate_noisy()

Indicate whether evaluate_noisy is available.

Indicate if a fast but potentially noisy version of evaluate is available through the function evaluate_noisy. 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.

Returns

bool Is evaluate_noisy available?

CHAPTER 11

rbfopt_utils module

Utility functions.

This module contains a number of subroutines that are used by the other modules. In particular it contains most of the subroutines that do the calculations using numpy, as well as utility functions for various modules.

rbfopt_utils.**bulk_evaluate_rbf**(*settings*, *points*, *n*, *k*, *node_pos*, *rbf_lambda*, *rbf_h*, *return_distances=u'no'*)

Evaluate the RBF interpolant at all points in a given list.

Evaluate the RBF interpolant at all points in a given list. This version uses numpy and should be faster than individually evaluating the RBF at each single point, provided that the list of points is large enough. It also computes the distance or the minimum distance of each point from the interpolation nodes, if requested, since this comes almost for free.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

points [2D numpy.ndarray[float]] The list of points in R^n where we want to evaluate the interpolant.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

node_pos [2D numpy.ndarray[float]] List of coordinates of the interpolation points.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to he polynomial. List of dimension given by get_size_P_matrix().

return_distances [string] If 'no', do nothing. If 'min', return the minimum distance of each point to interpolation nodes. If 'all', return the full distance matrix to the interpolation nodes.

Returns

1D numpy.ndarray[float] or (1D numpy.ndarray[float], 1D numpy.ndarray[float]) Value of the RBF interpolant at each point; if compute_min_dist is True, additionally returns the minimum distance of each point from the interpolation nodes.

rbfopt_utils.**bulk_get_min_distance**(*points*, *other_points*)

Get the minimum distances between two sets of points.

Compute the minimum distance of each point in the first set to the points in the second set. This is faster than using get_min_distance repeatedly, for large sets of points.

Parameters

points [2D numpy.ndarray[float]] The points in R^n that we compute the distances from.

other_points [2D numpy.ndarray[float]] The list of points we want to compute the distances to.

Returns

1D numpy.ndarray[float] Minimum distance between each point in points and the other_points.

See also:

get_min_distance

rbfopt_utils.**bulk_transform_domain**(*settings*, *var_lower*, *var_upper*, *points*, *reverse=False*)

Rescale the domain.

Rescale the function domain according to the chosen strategy.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

points [2D numpy.ndarray[float]] Point in the domain to be rescaled.

reverse [bool] False if we transform from the original domain to the transformed space, True if we want to apply the reverse.

Returns

2D numpy.ndarray[float] Rescaled points.

Raises

ValueError If the requested rescaling strategy is not implemented.

rbfopt_utils.**compute_gap**(*settings*, *fmin*)

Compute the optimality gap w.r.t. the target value.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

fmin [float] Best known function value discovered so far. Note that this value should already take into account possible noise at the best point.

Returns

float The current optimality gap, i.e. relative distance from target value.

rbfopt_utils.**distance**(*p1*, *p2*)
Compute Euclidean distance between two points.

Compute Euclidean distance between two points.

Parameters

p1 [1D numpy.ndarray[float]] First point.

p2 [1D numpy.ndarray[float]] Second point.

Returns

float Euclidean distance.

rbfopt_utils.**evaluate_rbf**(*settings*, *point*, *n*, *k*, *node_pos*, *rbf_lambda*, *rbf_h*)
Evaluate the RBF interpolant at a given point.

Evaluate the RBF interpolant at a given point.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

point [1D numpy.ndarray[float]] The point in R^n where we want to evaluate the interpolant.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

node_pos [2D numpy.ndarray[float]] List of coordinates of the interpolation points.

rbf_lambda [1D numpy.ndarray[float]] The lambda coefficients of the RBF interpolant, corresponding to the radial basis functions. List of dimension k.

rbf_h [1D numpy.ndarray[float]] The h coefficients of the RBF interpolant, corresponding to he polynomial. List of dimension given by get_size_P_matrix().

Returns

float Value of the RBF interpolant at the given point.

rbfopt_utils.**get_all_corners**(*var_lower*, *var_upper*)
Compute all corner points of a box.

Compute and return all the corner points of the given box. Note that this number is exponential in the dimension of the problem.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

Returns

2D numpy.ndarray[float] All the corner points.

rbfopt_utils.**get_best_rbf_model**(*settings*, *n*, *k*, *node_pos*, *node_val*, *num_nodes_to_check*)
Compute which type of RBF yields the best model.

Compute which RBF interpolant yields the best surrogate model, using cross validation to determine the lowest leave-one-out error.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the space where the point lives.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] Location of current interpolation nodes (one on each row.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

num_nodes_to_check [int] Number of nodes on which quality should be tested.

Returns

str The type of RBF that currently yields the best surrogate model, based on leave-one-out error. This will be one of the supported types of RBF.

rbfopt_utils.**get_degree_polynomial**(*settings*)

Compute the degree of the polynomial for the interpolant.

Return the degree of the polynomial that should be used in the RBF expression to ensure unisolvence and convergence of the optimization method.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

Returns

int Degree of the polynomial

Raises

ValueError If the matrix type is not implemented.

rbfopt_utils.**get_fmax_current_iter**(*settings*, *n*, *k*, *current_step*, *node_val*)

Compute the largest function value for target value computation.

Compute the largest function value used to determine the target value. This is given by the sorted value in position :math: *sigma_n*.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the space where the point lives.

k [int] Number of nodes, i.e. interpolation points.

current_step [int] The current step in the cyclic search strategy.

node_val [1D numpy.ndarray[float]] List of function values.

Returns

float The value that should be used to determine the range of the function values when computing the target value.

See also:

get_sigma_n

rbfopt_utils.**get_lhd_corr_points**(*var_lower*, *var_upper*, *num_trials=50*)

Compute a latin hypercube design with min correlation.

Compute a list of (n+1) points in the given box, where n is the dimension of the space. The selected points are picked according to a random latin hypercube design with minimum correlation criterion. This function relies on the library pyDOE.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

num_trials [int] Maximum number of generated LHs to choose from.

Returns

2D numpy.ndarray[float] List of points in the latin hypercube design.

rbfopt_utils.**get_lhd_maximin_points**(*var_lower*, *var_upper*, *num_trials=50*)

Compute a latin hypercube design with maximin distance.

Compute an array of (n+1) points in the given box, where n is the dimension of the space. The selected points are picked according to a random latin hypercube design with maximin distance criterion.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

num_trials [int] Maximum number of generated LHs to choose from.

Returns

2D numpy.ndarray[float] List of points in the latin hypercube design.

rbfopt_utils.**get_lower_corners**(*var_lower*, *var_upper*)

Compute the lower corner points of a box.

Compute a list of (n+1) corner points of the given box, where n is the dimension of the space. The selected points are the bottom left (i.e. corresponding to the origin in the 0-1 hypercube) and the n adjacent ones.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

Returns

2D numpy.ndarray[float] The lower corner points.

rbfopt_utils.**get_matrix_inverse**(*settings*, *Amat*)

Compute the inverse of a matrix.

Compute the inverse of a given matrix, zeroing out small coefficients to improve sparsity.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

Amat [numpy.matrix] The matrix to invert.

Returns

numpy.matrix The matrix Amat^{-1}.

Raises

numpy.linalg.LinAlgError If the matrix cannot be inverted for numerical reasons.

rbfopt_utils.**get_min_distance**(*point*, *other_points*)

Compute minimum distance from a set of points.

Compute the minimum Euclidean distance between a given point and a list of points.

Parameters

point [1D numpy.ndarray[float]] The point we compute the distances from.

other_points [2D numpy.ndarray[float]] The list of points we want to compute the distances to.

Returns

float Minimum distance between point and the other_points.

rbfopt_utils.**get_min_distance_and_index**(*point*, *other_points*)

Compute the distance and index of the point with minimum distance.

Compute the distance value and the index of the point in a matrix that achieves minimum Euclidean distance to a given point.

Parameters

point [1D numpy.ndarray[float]] The point we compute the distances from.

other_points [2D numpy.ndarray[float]] The list of points we want to compute the distances to.

Returns

(float, int) The distance value and the index of the point in other_points that achieved minimum distance from point.

rbfopt_utils.**get_model_quality_estimate**(*settings*, *n*, *k*, *node_pos*, *node_val*, *num_nodes_to_check*)

Compute an estimate of model quality.

Computes an estimate of model quality, performing cross-validation. It only checks the best num_nodes_to_check nodes.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the space where the point lives.

k [int] Number of nodes, i.e. interpolation points.

node_pos [2D numpy.ndarray[float]] Location of current interpolation nodes (one on each row).

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

num_nodes_to_check [int] Number of nodes on which quality should be tested.

Returns

float An estimate of the leave-one-out cross-validation error, which can be interpreted as a measure of model quality.

Raises

ValueError If the RBF type is not implemented.

rbfopt_utils.**get_most_common_element**(*array*, *to_exclude=[]*)

Get most common element in a list.

Parameters

array [List[any]] The list whose most common element is sought.

to_exclude [List[any]] A list of elements to exclude from the count.

Returns

Any The most common element, None if all elements were excluded.

rbfopt_utils.**get_one_ready_index**(*results*)
Get index of a single async computation result that is ready.

Given a list containing results of asynchronous computations dispatched to a worker pool, obtain the index of the last computation that has concluded. (Last is better to use the pop() function in the list.)

Parameters

results [List[(multiprocessing.pool.AsyncResult, any)]] A list of tasks, where each task is a list and the first element is the output of a call to apply_async. The other elements of the list will never be scanned by this function, and they could be anything.

Returns

int Index of last computation that completed, or len(results) if no computation is ready.

rbfopt_utils.**get_random_corners**(*var_lower*, *var_upper*)
Compute some randomly selected corner points of the box.

Compute a list of (n+1) corner points of the given box, where n is the dimension of the space. The selected points are picked randomly.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

Returns

2D numpy.ndarray[float] A List of random corner points.

rbfopt_utils.**get_rbf_coefficients**(*settings*, *n*, *k*, *Amat*, *node_val*)
Compute the coefficients of the RBF interpolant.

Solve a linear system to compute the coefficients of the RBF interpolant.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

Amat [numpy.matrix] Matrix [Phi P; P^T 0] defining the linear system. Must be a square matrix of appropriate size.

node_val [1D numpy.ndarray[float]] List of values of the function at the nodes.

Returns

(1D numpy.ndarray[float], 1D numpy.ndarray[float]) Lambda coefficients (for the radial basis functions), and h coefficients (for the polynomial).

rbfopt_utils.**get_rbf_function**(*settings*)
Return a radial basis function.

Return the radial basis function appropriate function as indicated by the settings.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

Returns

—

Callable[numpy.ndarray] A callable radial basis function that can be applied on floats and numpy.ndarray.

rbfopt_utils.**get_rbf_matrix**(*settings*, *n*, *k*, *node_pos*)

Compute the matrix for the RBF system.

Compute the matrix A = [Phi P; P^T 0] of equation (3) in the paper by Costa and Nannicini.

Parameters

settings [*rbfopt_settings.RbfoptSettings*.] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. the size of the space.

k [int] Number of interpolation nodes.

node_pos [2D numpy.ndarray[float]] List of coordinates of the nodes.

Returns

numpy.matrix The matrix A = [Phi P; P^T 0].

Raises

ValueError If the type of RBF function is not supported.

rbfopt_utils.**get_sigma_n**(*k*, *current_step*, *num_global_searches*, *num_initial_points*)

Compute sigma_n.

Compute the index :math: *sigma_n*, where :math: *sigma_n* is a function described in the paper by Gutmann (2001). The same function is called :math: *alpha_n* in a paper of Regis & Shoemaker (2007).

Parameters

k [int] Number of nodes, i.e. interpolation points.

current_step [int] The current step in the cyclic search strategy.

num_global_searches [int] The number of global searches in a cycle.

num_initial_points [int] Number of points for the initialization phase.

Returns

int The value of sigma_n.

rbfopt_utils.**get_size_P_matrix**(*settings*, *n*)

Compute size of the P part of the RBF matrix.

Return the number of columns in the P part of the matrix [Phi P; P^T 0] that is used through the algorithm.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

n [int] Dimension of the problem, i.e. number of variables.

Returns

int Number of columns in the matrix

Raises

ValueError If the matrix type is not implemented.

rbfopt_utils.**get_uniform_lhs**(*n*, *num_samples*)

Generate random Latin Hypercube samples.

Generate points using Latin Hypercube sampling from the uniform distribution in the unit hypercube.

Parameters

n [int] Dimension of the space, i.e. number of variables.

num_samples [num_samples] Number of samples to be generated.

Returns

2D numpy.ndarray[float] A list of n-dimensional points in the unit hypercube.

rbfopt_utils.**init_environment**(*settings*)

Initialize the random seed and disable Pyomo output.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

rbfopt_utils.**initialize_nodes**(*settings*, *var_lower*, *var_upper*, *integer_vars*)

Compute the initial sample points.

Compute an initial list of nodes using the initialization strategy indicated in the algorithmic settings.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

integer_vars [1D numpy.ndarray[int]] A List containing the indices of the integrality constrained variables. If empty, all variables are assumed to be continuous.

Returns

2D numpy.ndarray[float] Matrix containing at least n+1 corner points, one for each row, where n is the dimension of the space. The number and position of points depends on the chosen strategy.

Raises

RuntimeError If a set of feasible and linearly independent sample points cannot be computed within the prescribed number of iterations.

rbfopt_utils.**norm**(*p*)

Compute the L2-norm of a vector

Compute the L2 (Euclidean) norm.

Parameters

p [1D numpy.ndarray[float]] The point whose norm should be computed.

Returns

float The norm of the point.

rbfopt_utils.**results_ready**(*results*)

Check if some asynchronous results completed.

Given a list containing results of asynchronous computations dispatched to a worker pool, verify if some of them are ready for processing.

Parameters

results [List[(multiprocessing.pool.AsyncResult, any)]] A list of tasks, where each task is a list and the first element is the output of a call to apply_async. The other elements of the list will never be scanned by this function, and they could be anything.

Returns

bool True if at least one result has completed.

rbfopt_utils.**round_integer_bounds**(*var_lower*, *var_upper*, *integer_vars*)
Round the variable bounds to integer values.

Round the values of the integer-constrained variable bounds, in the usual way: lower bounds are rounded up, upper bounds are rounded down.

Parameters

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

integer_vars [1D numpy.ndarray[int]] A list containing the indices of the integrality constrained variables. If empty, all variables are assumed to be continuous.

rbfopt_utils.**round_integer_vars**(*point*, *integer_vars*)
Round a point to the closest integer.

Round the values of the integer-constrained variables to the closest integer value. The values are rounded in-place.

Parameters

point [1D numpy.ndarray[float]] The point to be rounded.

integer_vars [1D numpy.ndarray[int]] A list of indices of integer variables.

rbfopt_utils.**transform_domain**(*settings*, *var_lower*, *var_upper*, *point*, *reverse=False*)
Rescale the domain.

Rescale the function domain according to the chosen strategy.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

point [1D numpy.ndarray[float]] Point in the domain to be rescaled.

reverse [bool] False if we transform from the original domain to the transformed space, True if we want to apply the reverse.

Returns

1D numpy.ndarray[float] Rescaled point.

Raises

ValueError If the requested rescaling strategy is not implemented.

rbfopt_utils.**transform_domain_bounds**(*settings*, *var_lower*, *var_upper*)
Rescale the variable bounds.

Rescale the bounds of the function domain according to the chosen strategy.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

var_lower [1D numpy.ndarray[float]] List of lower bounds of the variables.

var_upper [1D numpy.ndarray[float]] List of upper bounds of the variables.

Returns

(1D numpy.ndarray[float], 1D numpy.ndarray[float]) Rescaled bounds as (lower, upper).

Raises

ValueError If the requested rescaling strategy is not implemented.

rbfopt_utils.**transform_function_values**(*settings*, *node_val*, *fmin*, *fmax*, *node_err_bounds*)

Rescale function values.

Rescale and adjust function values according to the chosen strategy and to the occurrence of large fluctuations (high dynamism). May not rescale at all if rescaling is off.

Parameters

settings [*rbfopt_settings.RbfoptSettings*] Global and algorithmic settings.

node_val [1D numpy.ndarray[float]] List of function values at the interpolation nodes.

fmin [float] Minimum function value found so far.

fmax [float] Maximum function value found so far.

node_err_bounds [2D numpy.ndarray[float]] The lower and upper variation of the function value for the nodes in node_pos. The variation is assumed 0 for nodes evaluated in accurate mode.

Returns

(1D numpy.ndarray[float], float, float, 2D numpy.ndarray[float],

Callable[float]) A tuple (scaled_function_values, scaled_fmin, scaled_fmax, scaled_error_bounds, rescale_function) containing a list of rescaled function values, the rescaled minimum, the rescaled maximum, the rescaled error bounds (one node per row), and a callable function to apply the same scaling to further function values if needed.

Raises

ValueError If the function scaling strategy requested is not implemented.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V