desdeo_mcdm

Release 1.0

Multiobjective Optimization Group

Jun 06, 2021

CONTENTS

1 Requirements **3**

2 Installation **5**
2.1 For users 5
2.2 For developers 5

3 Currently implemented methods **59**

4 Coming soon **61**

5 Indices and tables **63**

Python Module Index **65**

Index **67**

Contains interactive optimization methods for solving multiobjective optimizaton problems. This package is part of the DESDEO framework.

CHAPTER ONE

REQUIREMENTS

- Python 3.7 (3.8 is **NOT** supported at the moment).
- Poetry dependency manager : Only for developers.

See *pyproject.toml* for Python package requirements.

CHAPTER TWO

INSTALLATION

To install and use this package on a *nix-based system, follow one of the following procedures.

2.1 For users

First, create a new virtual environment for the project. Then install the package using the following command:

```
$ pip install desdeo-mcdm
```

2.2 For developers

Download the code or clone it with the following command:

```
$ git clone https://github.com/industrial-optimization-group/desdeo-mcdm
```

Then, create a new virtual environment for the project and install the package in it:

```
$ cd desdeo-mcdm
$ poetry init
$ poetry install
```

2.2.1 Background concepts

NAUTILUS

In NAUTILUS, starting from the nadir point, a solution is obtained at each iteration which dominates the previous one. Although only the last solution will be Pareto optimal, the decision maker never looses sight of the Pareto optimal set, and the search is oriented so that (s)he progressively focusses on the preferred part of the Pareto optimal set. Each new solution is obtained by minimizing an achievement scalarizing function including preferences about desired improvements in objective function values.

The decision maker has two possibilities to provide her/his preferences:

1. The decision maker can rank the objectives according to the relative importance of improving each current objective value.

Note: This ranking is not a global preference ranking of the objectives, but represents the local importance of improving each of the current objective values at that moment.

2. The decision maker can specify percentages reflecting how (s)he would like to improve the current objective values, by answering to the following question:

"Assuming you have one hundred points available, how would you distribute them among the current objective values so that the more points you allocate, the more improvement on the corresponding current objective value is desired?"

After each iteration round, the decision maker specifies whether (s)he wishes to continue with the previous preference information, or define a new one.

In addition to this, the decision maker can influence the solution finding process by taking a step back to previous iteration point. This enables the decision maker to provide new preferences and change the direction of solution seeking process. Furthermore, the decision maker can also take a half-step in case (s)he feels that a full step limits the reachable area of Pareto optimal set too much.

NAUTILUS is specially suitable for avoiding undesired anchoring effects, for example in negotiation support problems, or just as a means of finding an initial Pareto optimal solution for any interactive procedure.

NIMBUS

As its name suggests, NIMBUS (Nondifferentiable Interactive Multiobjective BUndle-based optimization System) is a multiobjective optimization system able to handle even non-differentiable functions. It will optimize (minimize or maximize) several functions simultaneously, creating a group of different solutions. One cannot say which one of them is the best, because the system cannot know the criteria affecting the 'goodness' of the desired solution. The user is the one that makes the decision.

Mathematically, all the generated solutions are 'equal', so it is important that the user can influence the solution process. The user may want to choose which of the functions should be optimized most, the limits of the objectives, etc. In NIMBUS, this phase is called a 'classification'. Searching for the desired solution means finding the best compromise between many different goals. If we want to get lower values for one function, we must be ready to accept the growth of another function. This is because the solutions produced by NIMBUS are Pareto optimal. This means that there is no possibility to achieve better solutions for some component of the problem without worsening some other component(s).

The Reference Point Method

In the Reference Point Method, the Decision Maker (DM) specifies desirable aspiration levels for objective functions. Vectors formed of these aspiration levels are then used to derive scalarizing functions having minimal values at weakly, properly or Pareto optimal solutions. It is important that reference points are intuitive and easy for the DM to specify, their consistency is not an essential requirement. Before the solution process starts, some information is given to the DM about the problem. If possible, the ideal objective vector and the (approximated) nadir objective vector are presented.

At each iteration, the DM is asked to give desired aspiration levels for the objective functions. Using this information to formulate a reference point, achievement function is minimized and a (weakly, properly or) Pareto optimal solution is obtained. This solution is then presented to the DM. In addition, k other (weakly, properly or) Pareto optimal solutions are calculated using perturbed reference points, where k is the number of objectives in the problem. The alternative solutions are also presented to the DM. If (s)he finds any of the k + 1 solutions satisfactory, the solution process is ended. Otherwise, the DM is asked to present a new reference point and the iteration described above is repeated.

The idea in perturbed reference points is that the DM gets better understanding of the possible solutions around the current solution. If the reference point is far from the Pareto optimal set, the DM gets a wider description of the Pareto optimal set and if the reference point is near the Pareto optimal set, then a finer description of the Pareto optimal set is given.

In this method, the DM has to specify aspiration levels and compare objective vectors. The DM is free to change her/his mind during the process and can direct the solution process without being forced to understand complicated concepts and their meaning. On the other hand, the method does not necessarily help the DM to find more satisfactory solutions.

NAUTILUS 2

Similarly to NAUTILUS, starting from the nadir point, a solution is obtained at each iteration which dominates the previous one. Although only the last solution will be Pareto optimal, the Decision Maker (DM) never looses sight of the Pareto optimal set, and the search is oriented so that (s)he progressively focusses on the preferred part of the Pareto optimal set. Each new solution is obtained by minimizing an achievement scalarizing function including preferences about desired improvements in objective function values.

NAUTILUS 2 introduces a new preference handling technique which is easily understandable for the DM and allows the DM to conveniently control the solution process. Preferences are given as direction of improvement for objectives. In NAUTILUS 2, the DM has three ways to do this:

1. The DM sets the direction of improvement directly.
2. The DM defines the improvement ratio between two different objectives f_i and f_j. For example, if the DM wishes that the improvement of fi by one unit should be accompanied with the improvement of f_j by $_{ij}$ units. Here, the DM selects an objective $f_i (i = 1, \ldots, k)$ and for each of the other objectives f_j sets the value $_{ij}$. Then, the direction of improvement is defined by

$$_i = 1 \; and \; _j =_{ij}, \; ji.$$

3. As a generalization of the approach 2, the DM sets values of improvement ratios freely for some selected pairs of objective functions.

As with NAUTILUS, after each iteration round, the decision maker specifies whether (s)he wishes to continue with the previous preference information, or define a new one.

In addition to this, the decision maker can influence the solution finding process by taking a step back to the previous iteration point. This enables the decision maker to provide new preferences and change the direction of the solution seeking process. Furthermore, the decision maker can also take a half-step in case (s)he feels that a full step limits the reachable area of the Pareto optimal set too much.

2.2.2 API Documentation

desdeo_mcdm.interactive Package

This module contains interactive methods and related requests implemented as classes.

Functions

validate_response(n_objectives, z_current, ...)	Validate decision maker's response.
validate_preferences(n_objectives, response)	Validate decision maker's preferences.
validate_n2_preferences(n_objectives, response)	Validate decision maker's preferences in NAUTILUS 2.
validate_n_iterations(n_it)	Validate decision maker's preference for number of iterations.

validate_response

desdeo_mcdm.interactive.**validate_response**(*n_objectives*, *z_current*, *nadir*, *response*, *first_iteration_bool*)

Validate decision maker's response.

Parameters

- **n_objectives** (*int*) – Number of objectives.
- **z_current** (*np.ndarray*) – Current iteration point.
- **nadir** (*np.ndarray*) – Nadir point.
- **response** (*Dict*) – Decision maker's response containing preference information.
- **first_iteration_bool** (*bool*) – Indicating whether the iteration round is the first one (True) or not (False).

Raises ***NautilusException*** – In case Decision maker's response is not valid.

Return type None

validate_preferences

desdeo_mcdm.interactive.**validate_preferences**(*n_objectives*, *response*)

Validate decision maker's preferences.

Parameters

- **n_objectives** (*int*) – Number of objectives in problem.
- **response** (*Dict*) – Decision maker's response containing preference information.

Raises ***NautilusException*** – In case preference info is not valid.

Return type None

validate_n2_preferences

`desdeo_mcdm.interactive.`**`validate_n2_preferences`**(*n_objectives*, *response*)

Validate decision maker's preferences in NAUTILUS 2.

Parameters

- **`n_objectives`** (`int`) – Number of objectives in problem.
- **`response`** (`Dict`) – Decision maker's response containing preference information.

Raises ***`NautilusException`*** – In case preference info is not valid.

Return type `None`

validate_n_iterations

`desdeo_mcdm.interactive.`**`validate_n_iterations`**(*n_it*)

Validate decision maker's preference for number of iterations.

Parameters **`n_it`** (`int`) – Number of iterations.

Raises ***`NautilusException`*** – If number of iterations given is not a positive integer greater than zero.

Return type `None`

Classes

`ENautilus`(pareto_front, ideal, nadir[, . . .])	
`ENautilusException`	Raised when an exception related to ENautilus is encountered.
`ENautilusInitialRequest`(ideal, nadir)	A request class to handle the initial preferences.
`ENautilusRequest`(ideal, nadir, points, . . .)	A request class to handle the intermediate requests.
`ENautilusStopRequest`(preferred_point)	A request class to handle termination.
`Nautilus`(problem, ideal, nadir[, epsilon, . . .])	Implements the basic NAUTILUS method as presented in **[Miettinen_2010]**.
`NautilusV2`(problem, starting_point, ideal, nadir)	Implements the NAUTILUS 2 method as presented in **[Miettinen_2015]**.
`NautilusException`	Raised when an exception related to Nautilus is encountered.
`NautilusInitialRequest`(ideal, nadir)	A request class to handle the Decision maker's initial preferences for the first iteration round.
`NautilusRequest`(z_current, nadir, . . .)	A request class to handle the Decision maker's preferences after the first iteration round.
`NautilusStopRequest`(x_h, f_h)	A request class to handle termination.
`NautilusNavigator`(pareto_front, ideal, nadir)	Implementations of the NAUTILUS Navigator algorithm.
`NautilusNavigatorException`	Raised when an exception related to NAUTILUS Navigator is encountered.
`NautilusNavigatorRequest`(ideal, nadir, . . .)	Request to handle interactions with NAUTILUS Navigator.
`NIMBUS`(problem[, scalar_method])	Implements the synchronous NIMBUS algorithm.

continues on next page

Table 2 – continued from previous page

NimbusException	Risen when an error related to NIMBUS is encountered.
NimbusClassificationRequest(method, ref)	A request to handle the classification of objectives in the synchronous NIMBUS method.
NimbusIntermediateSolutionsRequest(…)	A request to handle the computation of intermediate points between two previously computed points.
NimbusMostPreferredRequest(solution_vectors, …)	A request to handle the indication of a preferred point.
NimbusSaveRequest(solution_vectors, …)	A request to handle archiving of the solutions computed with NIMBUS.
NimbusStopRequest(solution_final, …)	A request to handle the termination of Synchronous NIMBUS.
ReferencePointMethod(problem, ideal, nadir)	Implements the Reference Point Method as presented in [Wierzbicki_1982].
RPMException	Raised when an exception related to Reference Point Method (RFM) is encountered.
RPMInitialRequest(ideal, nadir)	A request class to handle the Decision Maker's initial preferences for the first iteration round.
RPMRequest(f_current, f_additionals, ideal, …)	A request class to handle the Decision Maker's preferences after the first iteration round.
RPMStopRequest(x_h, f_h)	A request class to handle termination.

ENautilus

class desdeo_mcdm.interactive.**ENautilus**(*pareto_front*, *ideal*, *nadir*, *objective_names=None*, *minimize=None*)

Bases: desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod

Methods Summary

calculate_bounds(pareto_front, …)	Calculate the new bounds of the reachable points on the Pareto optimal front from each of the intermediate points.
calculate_distances(intermediate_points, …)	**rtype** ndarray
calculate_intermediate_points(…)	Calcualtes the intermediate points between representative points an a preferred point.
calculate_reachable_point_indices(…)	Calculate the indices of the reachable Pareto optimal solutions based on lower and upper bounds.
calculate_representative_points(…)	Calcualtes the most representative points on the Pareto front.
handle_initial_request(request)	Handles the initial request by parsing the response appropiately.
handle_request(request)	Handles the intermediate requests.
iterate(request)	Perform the next logical iteratino step based on the given request type.
start()	**rtype** ENautilusInitialRequest

Methods Documentation

calculate_bounds(*pareto_front*, *intermediate_points*)

Calculate the new bounds of the reachable points on the Pareto optimal front from each of the intermediate points.

Parameters

- **pareto_front** (*np.ndarray*) – The Pareto optimal front.
- **intermediate_points** (*np.ndarray*) – The current intermedaite points as a 2D array.

Returns The lower and upper bounds for each of the intermediate points.

Return type Tuple[np.ndarray, np.ndarray]

calculate_distances(*intermediate_points*, *zbars*, *nadir*)

Return type `ndarray`

calculate_intermediate_points(*preferred_point*, *zbars*, *n_iterations_left*)

Calcualtes the intermediate points between representative points an a preferred point.

Parameters

- **preferred_point** (*np.ndarray*) – The preferred point, 1D array.
- **zbars** (*np.ndarray*) – The representative points, 2D array.
- **n_iterations_left** (*int*) – The number of iterations left.

Returns The intermediate points as a 2D array.

Return type np.ndarray

calculate_reachable_point_indices(*pareto_front*, *lower_bounds*, *upper_bounds*)

Calculate the indices of the reachable Pareto optimal solutions based on lower and upper bounds.

Returns List of the indices of the reachable solutions.

Return type List[int]

calculate_representative_points(*pareto_front*, *subset_indices*, *n_points*)

Calcualtes the most representative points on the Pareto front. The points are clustered using k-means.

Parameters

- **pareto_front** (*np.ndarray*) – The Pareto front.
- **subset_indices** (*List[int]*) – A list of indices representing the
- **of the points on the Pareto front for which the** (*subset*) –
- **points should be calculated.** (*representative*) –
- **n_points** (*int*) – The number of representative points to be calculated.

Returns A 2D array of the most representative points. If the subset of Pareto efficient points is less than n_points, returns the subset of the Pareto front.

Return type np.ndarray

handle_initial_request(*request*)

Handles the initial request by parsing the response appropiately.

Return type `ENautilusRequest`

handle_request(*request*)
Handles the intermediate requests.

Return type `Union[ENautilusRequest, ENautilusStopRequest]`

iterate(*request*)
Perform the next logical iteratino step based on the given request type.

Return type `Union[ENautilusRequest, ENautilusStopRequest]`

start()

Return type `ENautilusInitialRequest`

ENautilusException

exception desdeo_mcdm.interactive.**ENautilusException**
Raised when an exception related to ENautilus is encountered.

ENautilusInitialRequest

class desdeo_mcdm.interactive.**ENautilusInitialRequest**(*ideal*, *nadir*)
Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle the initial preferences.

Attributes Summary

response

Methods Summary

init_with_method(method)	
validator(response)	**rtype** None

Attributes Documentation

response

Methods Documentation

classmethod init_with_method(*method*)

validator(*response*)

Return type None

ENautilusRequest

class desdeo_mcdm.interactive.**ENautilusRequest**(*ideal*, *nadir*, *points*, *lower_bounds*, *upper_bounds*, *distances*, *minimize*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle the intermediate requests.

Attributes Summary

response	

Methods Summary

validator(response)	**rtype** None

Attributes Documentation

response

Methods Documentation

validator(*response*)

Return type None

ENautilusStopRequest

class desdeo_mcdm.interactive.**ENautilusStopRequest**(*preferred_point*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle termination.

Nautilus

class `desdeo_mcdm.interactive.`**`Nautilus`**(*problem*, *ideal*, *nadir*, *epsilon=1e-06*, *objective_names=None*, *minimize=None*)

Bases: `desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod`

Implements the basic NAUTILUS method as presented in .

In NAUTILUS, starting from the nadir point, a solution is obtained at each iteration which dominates the previous one. Although only the last solution will be Pareto optimal, the decision maker never looses sight of the Pareto optimal set, and the search is oriented so that (s)he progressively focusses on the preferred part of the Pareto optimal set. Each new solution is obtained by minimizing an achievement scalarizing function including preferences about desired improvements in objective function values.

The decision maker has **two possibilities** to provide her/his preferences:

1. The decision maker can **rank** the objectives according to the **relative** importance of improving each current objective value.

Note: This ranking is not a global preference ranking of the objectives, but represents the local importance of improving each of the current objective values **at that moment**.

2. The decision maker can specify **percentages** reflecting how (s)he would like to improve the current objective values, by answering to the following question:

"Assuming you have one hundred points available, how would you distribute them among the current objective values so that the more points you allocate, the more improvement on the corresponding current objective value is desired?"

After each iteration round, the decision maker specifies whether (s)he wishes to continue with the previous preference information, or define a new one.

In addition to this, the decision maker can influence the solution finding process by taking a **step back** to previous iteration point. This enables the decision maker to provide new preferences and change the direction of solution seeking process. Furthermore, the decision maker can also take a **half-step** in case (s)he feels that a full step limits the reachable area of Pareto optimal set too much.

NAUTILUS is specially suitable for avoiding undesired anchoring effects, for example in negotiation support problems, or just as a means of finding an initial Pareto optimal solution for any interactive procedure.

Parameters

- **`problem`** (*`MOProblem`*) – Problem to be solved.
- **`ideal`** (*`np.ndarray`*) – The ideal objective vector of the problem.
- **`nadir`** (*`np.ndarray`*) – The nadir objective vector of the problem. This may also be the "worst" objective vector provided by the Decision maker if the approximation of Nadir vector is not applicable or if the Decision maker wishes to provide even worse objective vector than what the approximated Nadir vector is.
- **`epsilon`** (*`float`*) – A small number used in calculating the utopian point.
- **`objective_names`** (*`Optional[List[str]], optional`*) – Names of the objectives. List must match the number of columns in ideal.
- **`minimize`** (*`Optional[List[int]], optional`*) – Multipliers for each objective. '-1' indicates maximization and '1' minimization. Defaults to all objective values being minimized.

Raises *NautilusException* – One or more dimension mismatches are encountered among the supplies arguments.

Methods Summary

`calculate_bounds`(objectives, n_objectives, . . .)	Calculate the new bounds using Epsilon constraint method.
`calculate_distance`(z_current, nadir, f_current)	Calculates the distance from current iteration point to the Pareto optimal set.
`calculate_iteration_point`(itn, z_prev, f_current)	Calculate next iteration point towards the Pareto optimal solution.
`calculate_preferential_factors`(pref_method, . . .)	Calculate preferential factors based on the Decision maker's preference information.
`handle_initial_request`(request)	Handles the initial request by parsing the response appropriately.
`handle_request`(request)	Handle Decision maker's requests after the first iteration round, so called **intermediate requests.**
`iterate`(request)	Perform the next logical iteration step based on the given request type.
`solve_asf`(ref_point, x0, . . .)	Solve Achievement scalarizing function.
`start`()	Start the solution process with initializing the first request.

Methods Documentation

calculate_bounds(*objectives*, *n_objectives*, *x0*, *epsilons*, *bounds*, *constraints*, *method*)
Calculate the new bounds using Epsilon constraint method.

Parameters

- **objectives** (*np.ndarray*) – The objective function values for each input vector.
- **n_objectives** (*int*) – Total number of objectives.
- **x0** (*np.ndarray*) – Initial values for decision variables.
- **epsilons** (*np.ndarray*) – Previous iteration point.
- **bounds** (*Union[np.ndarray, None*) – Bounds for decision variables.
- **constraints** (*Callable*) – Constraints of the problem.
- **method** (*Union[ScalarMethod, str, None]*) – The optimization method the scalarizer should be minimized with.

Returns New lower bounds for objective functions.

Return type new_lower_bounds (np.ndarray)

calculate_distance(*z_current*, *nadir*, *f_current*)
Calculates the distance from current iteration point to the Pareto optimal set.

Parameters

- **z_current** (*np.ndarray*) – Current iteration point.
- **nadir** (*np.ndarray*) – Nadir vector.

- **`f_current`** (*`np.ndarray`*) – Current optimal objective vector.

Returns Distance to the Pareto optimal set.

Return type np.ndarray

`calculate_iteration_point` (*itn*, *z_prev*, *f_current*)
Calculate next iteration point towards the Pareto optimal solution.

Parameters

- **`itn`** (*`int`*) – Number of iterations left.
- **`z_prev`** (*`np.ndarray`*) – Previous iteration point.
- **`f_current`** (*`np.ndarray`*) – Current optimal objective vector.

Returns Next iteration point.

Return type np.ndarray

`calculate_preferential_factors` (*pref_method*, *pref_info*, *nadir*, *utopian*)
Calculate preferential factors based on the Decision maker's preference information. These preferential factors are used as weights for objectives when solving an Achievement scalarizing function. The Decision maker (DM) has **two possibilities** to provide her/his preferences:

1. The DM can rank the objectives according to the **relative** importance of improving each current objective value.

Note: This ranking is not a global preference ranking of the objectives, but represents the local importance of improving each of the current objective values **at that moment**.

2. The DM can specify percentages reflecting how (s)he would like to improve the current objective values, by answering to the following question:

"Assuming you have one hundred points available, how would you distribute them among the current objective values so that the more points you allocate, the more improvement on the corresponding current objective value is desired?"

Parameters

- **`pref_method`** (*`int`*) – Preference information method (either ranks (1) or percentages (2)).
- **`pref_info`** (*`np.ndarray`*) – Preference information on how the DM wishes to improve the values of each objective function.
- **`nadir`** (*`np.ndarray`*) – Nadir vector.
- **`utopian`** (*`np.ndarray`*) – Utopian vector.

Returns Weights assigned to each of the objective functions in achievement scalarizing function.

Return type np.ndarray

Examples

```
>>> pref_method = 1  # ranks
>>> pref_info = np.array([2, 2, 1, 1])  # first and second objective are the␣
↪most important to improve
>>> nadir = np.array([-4.75, -2.87, -0.32, 9.71])
>>> utopian = np.array([-6.34, -3.44, -7.5, 0.])
>>> calculate_preferential_factors(pref_method, pref_info, nadir, utopian)
array([0.31446541, 0.87719298, 0.13927577, 0.10298661])
```

```
>>> pref_method = 2  # percentages
>>> pref_info = np.array([10, 30, 40, 20])  # DM wishes to improve most the␣
↪value of objective 3, then 2,4,1
>>> nadir = np.array([-4.75, -2.87, -0.32, 9.71])
>>> utopian = np.array([-6.34, -3.44, -7.5, 0.])
>>> calculate_preferential_factors(pref_method, pref_info, nadir, utopian)
array([6.28930818, 5.84795322, 0.34818942, 0.51493306])
```

handle_initial_request (*request*)

Handles the initial request by parsing the response appropriately.

Parameters **request** (NautilusInitialRequest) – Initial request including Decision maker's initial preferences.

Returns New request with updated solution process information.

Return type *NautilusRequest*

handle_request (*request*)

Handle Decision maker's requests after the first iteration round, so called **intermediate requests.**

Parameters **request** (NautilusRequest) – Intermediate request including Decision maker's response.

Returns In case last iteration, request to stop the solution process. Otherwise, new request with updated solution process information.

Return type Union[*NautilusRequest*, *NautilusStopRequest*]

iterate (*request*)

Perform the next logical iteration step based on the given request type.

Parameters **request** (*Union[*NautilusInitialRequest, NautilusRequest*]*) – Either initial or intermediate request.

Returns A new request with content depending on the Decision maker's preferences.

Return type Union[*NautilusRequest*, *NautilusStopRequest*]

solve_asf (*ref_point*, *x0*, *preferential_factors*, *nadir*, *utopian*, *objectives*, *variable_bounds*, *method*)

Solve Achievement scalarizing function.

Parameters

- **ref_point** (*np.ndarray*) – Reference point.
- **x0** (*np.ndarray*) – Initial values for decision variables.
- **preferential_factors** (*np.ndarray*) – preferential factors on how much would the decision maker wish to improve the values of each objective function.
- **nadir** (*np.ndarray*) – Nadir vector.

- **utopian** (*np.ndarray*) – Utopian vector.
- **objectives** (*np.ndarray*) – The objective function values for each input vector.
- **variable_bounds** (*Optional[np.ndarray]*) – Lower and upper bounds of each variable as a 2D numpy array. If undefined variables, None instead.
- **method** (*Union[ScalarMethod, str, None]*) – The optimization method the scalarizer should be minimized with

Returns A dictionary with at least the following entries: 'x' indicating the optimal variables found, 'fun' the optimal value of the optimized function, and 'success' a boolean indicating whether the optimization was conducted successfully.

Return type Dict

start()

Start the solution process with initializing the first request.

Returns Initial request.

Return type *NautilusInitialRequest*

NautilusV2

class desdeo_mcdm.interactive.**NautilusV2**(*problem*, *starting_point*, *ideal*, *nadir*, *epsilon=1e-06*, *objective_names=None*, *minimize=None*)

Bases: desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod

Implements the NAUTILUS 2 method as presented in .

Similarly to NAUTILUS, starting from the nadir point, a solution is obtained at each iteration which dominates the previous one. Although only the last solution will be Pareto optimal, the Decision Maker (DM) never looses sight of the Pareto optimal set, and the search is oriented so that (s)he progressively focusses on the preferred part of the Pareto optimal set. Each new solution is obtained by minimizing an achievement scalarizing function including preferences about desired improvements in objective function values.

NAUTILUS 2 introduces a new preference handling technique which is easily understandable for the DM and allows the DM to conveniently control the solution process. Preferences are given as direction of improvement for objectives. In NAUTILUS 2, the DM has **three ways** to do this:

1. The DM sets the direction of improvement directly.
2. The DM defines the **improvement ratio** between two different objectives fi and fj. For example, if the DM wishes that the improvement of fi by one unit should be accompanied with the improvement of fj by ij units. Here, the DM selects an objective fi (i=1,. . . ,k) and for each of the other objectives fj sets the value ij. Then, the direction of improvement is defined by

   ```
   i=1 and j=ij, ji.
   ```

3. As a generalization of the approach 2, the DM sets values of improvement ratios freely for some **selected pairs** of objective functions.

As with NAUTILUS, after each iteration round, the decision maker specifies whether (s)he wishes to continue with the previous preference information, or define a new one.

In addition to this, the decision maker can influence the solution finding process by taking a **step back** to the previous iteration point. This enables the decision maker to provide new preferences and change the direction of the solution seeking process. Furthermore, the decision maker can also take a **half-step** in case (s)he feels that a full step limits the reachable area of the Pareto optimal set too much.

Parameters

- **`problem`** (*`MOProblem`*) – Problem to be solved.
- **`starting_point`** (*`np.ndarray`*) – Objective vector used as a starting point for method.
- **`ideal`** (*`np.ndarray`*) – The ideal objective vector of the problem being represented by the Pareto front.
- **`nadir`** (*`np.ndarray`*) – The nadir objective vector of the problem being represented by the Pareto front.
- **`epsilon`** (*`float`*) – A small number used in calculating the utopian point. By default 1e-6.
- **`objective_names`** (*`Optional[List[str]], optional`*) – Names of the objectives. The length of the list must match the number of columns in ideal.
- **`minimize`** (*`Optional[List[int]], optional`*) – Multipliers for each objective. '-1' indicates maximization and '1' minimization. Defaults to all objective values being minimized.

Raises **`NautilusException`** – One or more dimension mismatches are encountered among the supplies arguments.

Methods Summary

`calculate_bounds`(objectives, n_objectives, ...)	Calculate the new bounds using Epsilon constraint method.
`calculate_distance`(z_current, ...)	Calculates the distance from current iteration point to the Pareto optimal set.
`calculate_doi`(n_objectives, pref_info)	Calculate direction of improvement based on improvement ratios between pairs of objective functions.
`calculate_iteration_point`(itn, z_prev, f_current)	Calculate next iteration point towards the Pareto optimal solution.
`calculate_preferential_factors`(n_objectives, ...)	Calculate preferential factors based on decision maker's preference information.
`handle_initial_request`(request)	Handles the initial request by parsing the response appropriately.
`handle_request`(request)	Handle Decision maker's requests after the first iteration round, so-called **intermediate requests.**
`iterate`(request)	Perform the next logical iteration step based on the given request type.
`solve_asf`(ref_point, x0, ... [, ...])	Solve achievement scalarizing function.
`start`()	Start the solution process with initializing the first request.

Methods Documentation

calculate_bounds (*objectives*, *n_objectives*, *x0*, *epsilons*, *bounds*, *constraints*, *method*)
Calculate the new bounds using Epsilon constraint method.

Parameters

- **objectives** (*np.ndarray*) – The objective function values for each input vector.
- **n_objectives** (*int*) – Total number of objectives.
- **x0** (*np.ndarray*) – Initial values for decision variables.
- **epsilons** (*np.ndarray*) – Previous iteration point.
- **bounds** (*Union[np.ndarray, None]*) – Bounds for decision variables.
- **constraints** (*Callable*) – Constraints of the problem.
- **method** (*Union[ScalarMethod, str, None]*) – The optimization method the scalarizer should be minimized with.

Returns New lower bounds for objective functions.

Return type np.ndarray

calculate_distance (*z_current*, *starting_point*, *f_current*)
Calculates the distance from current iteration point to the Pareto optimal set.

Parameters

- **z_current** (*np.ndarray*) – Current iteration point.
- **starting_point** (*np.ndarray*) – Starting iteration point.
- **f_current** (*np.ndarray*) – Current optimal objective vector.

Returns Distance to the Pareto optimal set.

Return type np.ndarray

calculate_doi (*n_objectives*, *pref_info*)
Calculate direction of improvement based on improvement ratios between pairs of objective functions.

Parameters

- **n_objectives** (*int*) – Number of objectives.
- **pref_info** (*np.ndarray*) – Preference information on how the DM wishes to improve the values of each objective function.

Returns Direction of improvement.

Return type np.ndarray

calculate_iteration_point (*itn*, *z_prev*, *f_current*)
Calculate next iteration point towards the Pareto optimal solution.

Parameters

- **itn** (*int*) – Number of iterations left.
- **z_prev** (*np.ndarray*) – Previous iteration point.
- **f_current** (*np.ndarray*) – Current optimal objective vector.

Returns Next iteration point.

Return type np.ndarray

calculate_preferential_factors(*n_objectives*, *pref_method*, *pref_info*)
Calculate preferential factors based on decision maker's preference information.

Parameters

- **n_objectives** (*int*) – Number of objectives in problem.
- **pref_method** (*int*) – Preference information method, either: Direction of improvement (1), improvement ratios between a selected objective and rest of the objectives (2), or improvement ratios freely for some selected pairs of objectives (3).
- **pref_info** (*np.ndarray*) – Preference information on how the DM wishes to improve the values of each objective function. **See the examples below**.

Returns Direction of improvement. Used as weights assigned to each of the objective functions in the achievement scalarizing function.

Return type np.ndarray

Examples

```
>>> n_objectives = 4
>>> pref_method = 1  # deltas directly
>>> pref_info = np.array([1, 2, 1, 2]),  # second and fourth objective are␣
↪the most important to improve
>>> calculate_preferential_factors(n_objectives, pref_method, pref_info)
np.array([1, 2, 1, 2])
```

```
>>> n_objectives = 4
>>> pref_method = 2  # improvement ratios between one selected objective and␣
↪each other objective
>>> pref_info = np.array([1, 1.5, (7/3), 0.5])  # first objective's ratio is␣
↪set to one
>>> calculate_preferential_factors(n_objectives, pref_method, pref_info)
np.array([1, 1.5, (7/3), 0.5])
```

```
>>> n_objectives = 4
>>> pref_method = 3  # improvement ratios between freely selected pairs of␣
↪objectives
# format the tuples like this: (('index of objective', 'index of objective'),
↪'improvement ratio between the objectives')
>>> pref_info = np.array([((1, 2), 0.5), ((3, 4), 1), ((2, 3), 1.5)],␣
↪dtype=object)
>>> calculate_preferential_factors(n_objectives, pref_method, pref_info)
np.array([1., 0.5, 0.75, 0.75])
```

Note: Remember to specify "dtype=object" in **pref_info** array when using preference method 3.

handle_initial_request(*request*)
Handles the initial request by parsing the response appropriately.

Parameters request (NautilusInitialRequest) – Initial request including Decision maker's initial preferences.

Returns New request with updated solution process information.

Return type *NautilusRequest*

handle_request(*request*)

Handle Decision maker's requests after the first iteration round, so-called **intermediate requests.**

Parameters **request** (NautilusRequest) – Intermediate request including Decision maker's response.

Returns In case last iteration, request to stop the solution process. Otherwise, new request with updated solution process information.

Return type Union[*NautilusRequest*, *NautilusStopRequest*]

iterate(*request*)

Perform the next logical iteration step based on the given request type.

Parameters **request** (*Union[*NautilusInitialRequest, NautilusRequest*]*) – Either initial or intermediate request.

Returns A new request with content depending on the Decision maker's preferences.

Return type Union[*NautilusRequest*, *NautilusStopRequest*]

solve_asf(*ref_point*, *x0*, *preferential_factors*, *nadir*, *utopian*, *objectives*, *variable_bounds=None*, *method=None*)

Solve achievement scalarizing function.

Parameters

- **ref_point** (*np.ndarray*) – Reference point.
- **x0** (*np.ndarray*) – Initial values for decision variables.
- **preferential_factors** (*np.ndarray*) – Preferential factors indicating how much would the decision maker wish to improve the values of each objective function.
- **nadir** (*np.ndarray*) – Nadir vector.
- **utopian** (*np.ndarray*) – Utopian vector.
- **objectives** (*np.ndarray*) – The objective function values for each input vector.
- **variable_bounds** (*Optional[np.ndarray]*) – Lower and upper bounds of each variable as a 2D numpy array. If undefined variables, None instead.
- **method** (*Union[ScalarMethod, str, None]*) – The optimization method the scalarizer should be minimized with.

Returns A dictionary with at least the following entries: 'x' indicating the optimal variables found, 'fun' the optimal value of the optimized function, and 'success' a boolean indicating whether the optimization was conducted successfully.

Return type Dict

start()

Start the solution process with initializing the first request.

Returns Initial request.

Return type *NautilusInitialRequest*

NautilusException

exception `desdeo_mcdm.interactive.`**`NautilusException`**

Raised when an exception related to Nautilus is encountered.

NautilusInitialRequest

class `desdeo_mcdm.interactive.`**`NautilusInitialRequest`**(*ideal*, *nadir*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle the Decision maker’s initial preferences for the first iteration round.

Attributes Summary

response	

Methods Summary

init_with_method(method)	Initialize request with given instance of Nautilus method.

Attributes Documentation

`response`

Methods Documentation

classmethod init_with_method(*method*)

Initialize request with given instance of Nautilus method.

Parameters **`method`** (`Nautilus`) – Instance of Nautilus-class.

Returns Initial request.

Return type *NautilusInitialRequest*

NautilusRequest

class `desdeo_mcdm.interactive.`**`NautilusRequest`**(*z_current*, *nadir*, *lower_bounds*, *upper_bounds*, *distance*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle the Decision maker’s preferences after the first iteration round.

Attributes Summary

response

Attributes Documentation

response

NautilusStopRequest

class desdeo_mcdm.interactive.**NautilusStopRequest**(*x_h*, *f_h*)

Bases: desdeo_tools.interaction.request.BaseRequest

A request class to handle termination.

NautilusNavigator

class desdeo_mcdm.interactive.**NautilusNavigator**(*pareto_front*, *ideal*, *nadir*, *objective_names=None*, *minimize=None*)

Bases: desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod

Implementations of the NAUTILUS Navigator algorithm.

Methods Summary

calculate_bounds(pareto_front, nav_point)	Calculate the new bounds of the reachable points on the Pareto optimal front from a navigation point.
calculate_distance(nav_point, projection, nadir)	Calculate the distance to the Pareto optimal front from a navigation point.
calculate_navigation_point(projection, ...)	Calculate a new navigation point based on the projection of the preference point to the Pareto optimal front.
calculate_reachable_point_indices(...)	Calculate the indices of the reachable Pareto optimal solutions based on lower and upper bounds.
handle_request(request)	Handle the Request and its contents.
iterate(request)	Perform the next logical step based on the response in the Request.
solve_nautilus_asf_problem(pareto_f, ...)	Forms and solves the achievement scalarizing function to find the closesto point on the Pareto optimal front to the given reference point.
start()	Returns the first Request object to begin iterating.
update(ref_point, speed, go_to_previous, stop)	Update the inernal state of self.

Methods Documentation

calculate_bounds (*pareto_front*, *nav_point*)

Calculate the new bounds of the reachable points on the Pareto optimal front from a navigation point.

Parameters

- **pareto_front** (*np.ndarray*) – The Pareto optimal front.
- **nav_point** (*np.ndarray*) – The current navigation point.

Returns The lower and upper bounds.

Return type Tuple[np.ndarray, np.ndarray]

calculate_distance (*nav_point*, *projection*, *nadir*)

Calculate the distance to the Pareto optimal front from a navigation point. The distance is calculated to the supplied projection which is assumed to lay on the front.

Parameters

- **nav_point** (*np.ndarray*) – The navigation point.
- **projection** (*np.ndarray*) – The point of the Pareto optimal front the distance is calculated to.
- **nadir** (*np.ndarray*) – The nadir point of the Pareto optimal set.

Returns The distance.

Return type float

calculate_navigation_point (*projection*, *nav_point*, *steps_remaining*)

Calculate a new navigation point based on the projection of the preference point to the Pareto optimal front.

Parameters

- **projection** (*np.ndarray*) – The point on the Pareto optimal front
- **to the preference point given by a decision maker.** (*closest*) –
- **nav_point** (*np.ndarray*) – The previous navigation point.
- **steps_remaining** (*int*) – How many steps are remaining in the navigation.

Returns The new navigation point.

Return type np.ndarray

calculate_reachable_point_indices (*pareto_front*, *lower_bounds*, *upper_bounds*)

Calculate the indices of the reachable Pareto optimal solutions based on lower and upper bounds.

Returns List of the indices of the reachable solutions.

Return type List[int]

handle_request (*request*)

Handle the Request and its contents.

Parameters **request** (NautilusNavigatorRequest) – A Request with a defined response.

Returns Some of the contents of the response are invalid.

Return type *NautilusNavigatorRequest*

iterate(*request*)
Perform the next logical step based on the response in the Request.

Return type NautilusNavigatorRequest

solve_nautilus_asf_problem(*pareto_f*, *subset_indices*, *ref_point*, *ideal*, *nadir*)
Forms and solves the achievement scalarizing function to find the closesto point on the Pareto optimal front to the given reference point.

Parameters

- **pareto_f** (*np.ndarray*) – The whole Pareto optimal front.
- **subset_indices** (*[type]*) – Indices of the currently reachable solutions.
- **ref_point** (*np.ndarray*) – The reference point indicating a decision
- **preference.** (*maker's*) –
- **ideal** (*np.ndarray*) – Ideal point.
- **nadir** (*np.ndarray*) – Nadir point.

Returns Index of the closest point according the minimized value of the ASF.

Return type int

start()
Returns the first Request object to begin iterating.

Returns The Request.

Return type *NautilusNavigatorRequest*

update(*ref_point*, *speed*, *go_to_previous*, *stop*, *step_number=None*, *nav_point=None*, *lower_bounds=None*, *upper_bounds=None*, *reachable_idx=None*, *distance=None*, *steps_remaining=None*)
Update the inernal state of self.

Parameters

- **ref_point** (*np.ndarray*) – A reference point given by a decision maker.
- **speed** (*int*) – An integer value between 1-5 indicating the navigation speed.
- **go_to_previous** (*bool*) – If True, the parameters indicate the state
- **a previous state** (*of*) –
- **the request is handled accordingly.** (*and*) –
- **stop** (*bool*) – If the navigation should stop. If True, self.update return None.
- **step_number** (*Optional[int], optional*) – Current step number, or
- **step number if go_to_previous is True. Defaults to None.** (*previous*) –
- **nav_point** (*Optional[np.ndarray], optional*) – The current
- **point. Relevant if go_to_previous is True. Defaults to** (*navigation*) –
- **None.** –
- **lower_bounds** (*Optional[np.ndarray], optional*) – Lower bounds of
- **reachable objective vector valus. Relevant if go_to_previous** (*the*) –

- **True. Defaults to None.** (*is*) –
- **upper_bounds** (*Optional[np.ndarray], optional*) – Upper bounds of
- **reachable objective vector valus. Relevant if go_to_previous** –
- **True. Defaults to None.** –
- **reachable_idx** (*Optional[List[int]], optional*) – Indices of the
- **Pareto optimal solutions. Relevant if go_to_previous is** (*reachable*) –
- **Defaults to None.** (*True.*) –
- **distance** (*Optional[float], optional*) – Distance to the Pareto
- **front. Relevant if go_to_previous is True. Defaults to** (*optimal*) –
- **None.** –
- **steps_remaining** (*Optional[int], optional*) – Remaining steps in the
- **Relevant if go_to_previous is True. Defaults to None.** (*navigation.*) –

Returns Some of the given parameters are erraneous.

Return type *NautilusNavigatorRequest*

NautilusNavigatorException

exception desdeo_mcdm.interactive.**NautilusNavigatorException**

Raised when an exception related to NAUTILUS Navigator is encountered.

NautilusNavigatorRequest

class desdeo_mcdm.interactive.**NautilusNavigatorRequest**(*ideal, nadir, reachable_lb, reachable_ub, reachable_idx, step_number, steps_remaining, distance, allowed_speeds, current_speed, navigation_point*)

Bases: desdeo_tools.interaction.request.BaseRequest

Request to handle interactions with NAUTILUS Navigator. See the NautilusNavigator class for further details.

Attributes Summary

response	

Methods Summary

init_with_method(method)	
validator(response)	**rtype** None

Attributes Documentation

response

Methods Documentation

classmethod init_with_method(*method*)

validator(*response*)

Return type None

NIMBUS

class desdeo_mcdm.interactive.**NIMBUS**(*problem*, *scalar_method=None*)

Bases: desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod

Implements the synchronous NIMBUS algorithm.

Parameters

- **problem** (*MOProblem*) – The problem to be solved.
- **scalar_method** (*Optional[ScalarMethod], optional*) – The method used to solve the various ASF minimization problems present in the method. Defaults to None.

Methods Summary

calculate_new_solutions(number_of_solutions, ...)	Calcualtes new solutions based on classifications supplied by the decision maker by
compute_intermediate_solutions(solutions, ...)	Computs intermediate solution between two solutions computed earlier.
create_plot_request(objectives, msg)	Used to create a plot request for visualizing objective values.
handle_classification_request(request)	Handles a classification request.
handle_intermediate_solutions_request(request)	Handles an intermediate solutions request.
handle_most_preferred_request(request)	Handles a preferres solution request.
handle_save_request(request)	Handles a save request.

continues on next page

Table 16 – continued from previous page

`iterate`(request)	Implements a finite state machine to iterate over the different steps defined in Synchronous NIMBUS based on a supllied request.
`request_classification`()	**rtype** `Tuple[NimbusClassificationRequest, SimplePlotRequest]`
`request_most_preferred_solution`(solutions, …)	Create a NimbusMostPreferredRequest.
`request_stop`()	Create a NimbusStopRequest based on self.
`save_solutions_to_archive`(objectives, …)	Save solutions to the archive.
`start`()	Return the first request to start iterating NIMBUS.
`update_current_solution`(solutions, …)	Update the state of self with a new current solution and the corresponding objective values.

Methods Documentation

`calculate_new_solutions`(*number_of_solutions*, *levels*, *improve_inds*, *improve_until_inds*, *acceptable_inds*, *impaire_until_inds*, *free_inds*)

Calcualtes new solutions based on classifications supplied by the decision maker by solving ASF problems.

Parameters

- **`number_of_solutions`** (*`int`*) – Number of solutions, should be between 1 and 4.
- **`levels`** (*`np.ndarray`*) – Aspiration and upper bounds relevant to the some of the classifications.
- **`improve_inds`** (*`np.ndarray`*) – Indices corresponding to the objectives which should be improved.
- **`improve_until_inds`** (*`np.ndarray`*) – Like above, but improved until an aspiration level is reached.
- **`acceptable_inds`** (*`np.ndarray`*) – Indices of objectives which are acceptable as they are now.
- **`impaire_until_inds`** (*`np.ndarray`*) – Indices of objectives which may be impaired until an upper limit is reached.
- **`free_inds`** (*`np.ndarray`*) – Indices of objectives which may change freely.

Returns A save request with the newly computed soutions, and a plot request to visualize said solutions.

Return type Tuple[*NimbusSaveRequest*, SimplePlotRequest]

`compute_intermediate_solutions`(*solutions*, *n_desired*)

Computs intermediate solution between two solutions computed earlier.

Parameters

- **`solutions`** (*`np.ndarray`*) – The solutions between which the intermediat solutions should be computed.

- **n_desired** (*int*) – The number of intermediate solutions desired.

Raises ***NimbusException*** –

Returns A save request with the compured intermediate points, and a plot request to visualize said points.

Return type Tuple[*NimbusSaveRequest*, SimplePlotRequest]

create_plot_request (*objectives*, *msg*)
Used to create a plot request for visualizing objective values.

Parameters

- **objectives** (*np.ndarray*) – A 2D numpy array containing objective vectors to be visualized.
- **msg** (*str*) – A message to be displayed in the context of a visualization.

Returns A plot request to create a visualization.

Return type SimplePlotRequest

handle_classification_request (*request*)
Handles a classification request.

Parameters **request** (*NimbusClassificationReuest*) – A classification request with the response attribute set.

Returns A NIMBUS save request and a plot request with the solutions the decision maker can choose from to save for alter use.

Return type Tuple[*NimbusSaveRequest*, SimplePlotRequest]

handle_intermediate_solutions_request (*request*)
Handles an intermediate solutions request.

Parameters **request** (NimbusIntermediateSolutionsRequest) – A NIMBUS intermediate solutions request with the response attribute set.

Returns Return either a save request or a preferred solution request. The former is returned if the decision maker wishes to see intermediate points, the latter otherwise. Also a plot request is returned with the solutions available in it.

Return type Tuple[Union[*NimbusSaveRequest*, *NimbusMostPreferredRequest*], SimplePlotRequest,]

handle_most_preferred_request (*request*)
Handles a preferres solution request.

Parameters **request** (NimbusMostPreferredRequest) – A NIMBUS preferred solution request with the response attribute set.

Returns Return a classificaiton request if the decision maker wishes to continue. If the decision maker wishes to stop, return a stop request. Also return a plot request with all the solutions saved so far.

Return type Tuple[Union[*NimbusClassificationRequest*, *NimbusStopRequest*], SimplePlotRequest]

handle_save_request (*request*)
Handles a save request.

Parameters **request** (NimbusSaveRequest) – A save request with the response attribute set.

Returns Return an intermediate solution request where the decision maker can specify whether they would like to see intermediate solution between two previously computed solutions. The plot request has the available solutions.

Return type Tuple[*NimbusIntermediateSolutionsRequest*, SimplePlotRequest]

iterate(*request*)
Implements a finite state machine to iterate over the different steps defined in Synchronous NIMBUS based on a supllied request.

Parameters **request** (*Union[NimbusClassificationRequest, NimbusSaveRequest, NimbusIntermediateSolutionsRequest, NimbusMostPreferredRequest, NimbusStopRequest,]*) – A request based on the next step in the NIMBUS algorithm is taken.

Raises ***NimbusException*** – If a wrong type of request is supllied based on the current state NIMBUS is in.

Returns The next logically sound request.

Return type Tuple[Union[*NimbusClassificationRequest*,*NimbusSaveRequest*,*NimbusIntermediateSolutionsRequest*,],U
None],]

request_classification()

Return type `Tuple[NimbusClassificationRequest, SimplePlotRequest]`

request_most_preferred_solution(*solutions*, *objectives*)
Create a NimbusMostPreferredRequest.

Parameters

- **solutions** (*np.ndarray*) – A 2D numpy array of decision variable vectors.
- **objectives** (*np.ndarray*) – A 2D numpy array of objective value vectors.

Returns The requests based on the given arguments.

Return type Tuple[*NimbusMostPreferredRequest*, SimplePlotRequest]

Note: The ‘i’th decision variable vector in *solutions* should correspond to the ‘i’th objective value vector in *objectives*.

request_stop()
Create a NimbusStopRequest based on self.

Returns A stop request and a plot request with the final solution chosen in it.

Return type Tuple[*NimbusStopRequest*, SimplePlotRequest]

save_solutions_to_archive(*objectives*, *decision_variables*, *indices*)
Save solutions to the archive. Saves also the corresponding objective function values.

Parameters

- **objectives** (*np.ndarray*) – Available objectives.
- **decision_variables** (*np.ndarray*) – Available solutions.
- **indices** (*List[int]*) – Indices of the solutions to be saved.

Returns An intermediate solutions request asking the decision maker whether they would like to generate intermediata solutions between two existing solutions. Also returns a plot request to visualize the available solutions between which the intermediate solutions should be computed.

Return type Tuple[*NimbusIntermediateSolutionsRequest*, None]

start()

Return the first request to start iterating NIMBUS.

Returns The first request and and a plot request to visualize relevant data.

Return type Tuple[*NimbusClassificationRequest*, SimplePlotRequest]

update_current_solution(*solutions*, *objectives*, *index*)

Update the state of self with a new current solution and the corresponding objective values. This solution is used in the classification phase of synchronous NIMBUS.

Parameters

- **solutions** (*np.ndarray*) – A 2D numpy array of decision variable vectors.
- **objectives** (*np.ndarray*) – A 2D numpy array of objective value vectors.
- **index** (*int*) – The index of the solution in *solutions* and *objectives*.

Returns The requests based on the given arguments.

Return type Tuple[*NimbusMostPreferredRequest*, SimplePlotRequest]

Note: The 'i'th decision variable vector in *solutions* should correspond to the 'i'th objective value vector in *objectives*.

NimbusException

exception desdeo_mcdm.interactive.**NimbusException**

Risen when an error related to NIMBUS is encountered.

NimbusClassificationRequest

class desdeo_mcdm.interactive.**NimbusClassificationRequest**(*method*, *ref*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request to handle the classification of objectives in the synchronous NIMBUS method.

Parameters

- **method** (NIMBUS) – The instance of the NIMBUS method the request should be initialized for.
- **ref** (*np.ndarray*) – Objective values used as a reference the decision maker is classifying the objectives.

self.**_valid_classifications**

The valid classifications. Defaults is ['<', '<=', '=', '>=', '0']

Type List[str]

Attributes Summary

response	

Methods Summary

validator(response)	Validates a dictionary containing the response of a decision maker.

Attributes Documentation

response

Methods Documentation

validator(*response*)

Validates a dictionary containing the response of a decision maker. Should contain the keys 'classifications', 'levels', and 'number_of_solutions'.

'classifications' should be a list of strings, where the number of elements is equal to the number of objectives being classified, and the elements are found in *_valid_classifications*. 'levels' should have either aspiration levels or bounds for each objective depending on that objective's classification. 'number_of_solutions' should be an integer between 1 and 4 indicating the number of intermediate solutions to be computed.

Parameters **response** (*Dict*) – See the documentation for *validator*.

Raises ***NimbusException*** – Some discrepancy is encountered in the parsing of the response.

Return type None

NimbusIntermediateSolutionsRequest

class desdeo_mcdm.interactive.**NimbusIntermediateSolutionsRequest**(*solution_vectors*, *objective_vectors*)

Bases: desdeo_tools.interaction.request.BaseRequest

A request to handle the computation of intermediate points between two previously computed points.

Parameters

- **solution_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing a decision variable vector.
- **objective_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing an objective vector.

Note: The objective vector at position 'i' in *objective_vectors* should correspond to the decision variables at position 'i' in *solution_vectors*. Only the two first entries in each of the lists is relevant. The rest is ignored.

Attributes Summary

response	

Methods Summary

validator(response)	Validates a response dictionary.

Attributes Documentation

response

Methods Documentation

validator(*response*)

Validates a response dictionary. The dictionary should contain the keys 'indices' and 'number_of_solutions'.

'indices' should be a list of integers representing an index to the lists *solutions_vectors* and *objective_vectors*. 'number_of_solutions' should be an integer greater or equal to 1.

Parameters **response** (*Dict*) – See the documentation for *validator*.

Raises ***NimbusException*** – Some discrepancy is encountered in the parsing of *response*.

NimbusMostPreferredRequest

class desdeo_mcdm.interactive.**NimbusMostPreferredRequest**(*solution_vectors*, *objective_vectors*)

Bases: desdeo_tools.interaction.request.BaseRequest

A request to handle the indication of a preferred point.

Parameters

- **solution_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing a decision variable vector.
- **objective_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing an objective vector.

Note: The objective vector at position 'i' in *objective_vectors* should correspond to the decision variables at position 'i' in *solution_vectors*. Only the two first entries in each of the lists is relevant. The preferred solution will be selected from *objective_vectors*.

Attributes Summary

response	

Methods Summary

validator(response)	Validates a response dictionary.

Attributes Documentation

response

Methods Documentation

validator(*response*)

Validates a response dictionary. The dictionary should contain the keys 'index' and 'continue'.

'index' is an integer and should indicate the index of the preferred solution is *objective_vectors*. 'continue' is a boolean and indicates whether to stop or continue the iteration of Synchronous NIMBUS.

Parameters **response** (*Dict*) – See the documentation for *validator*.

Raises ***NimbusException*** – Some discrepancy is encountered in the parsing of *response*.

NimbusSaveRequest

class desdeo_mcdm.interactive.**NimbusSaveRequest**(*solution_vectors*, *objective_vectors*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request to handle archiving of the solutions computed with NIMBUS.

Parameters

- **solution_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing a decision variable vector.
- **objective_vectors** (*List[np.ndarray]*) – A list of numpy arrays each representing an objective vector.

Note: The objective vector at position 'i' in *objective_vectors* should correspond to the decision variables at position 'i' in *solution_vectors*.

Attributes Summary

response	

Methods Summary

validator(response)	Validates a response dictionary.

Attributes Documentation

response

Methods Documentation

validator(*response*)

Validates a response dictionary. The dictionary should contain the keys 'indices'.

'indices' should be a list of integers representing an index to the lists *solutions_vectors* and *objective_vectors*.

Parameters **response** (*Dict*) – See the documentation for *validator*.

Raises ***NimbusException*** – Some discrepancy is encountered in the parsing of *response*.

Return type None

NimbusStopRequest

class desdeo_mcdm.interactive.**NimbusStopRequest**(*solution_final*, *objective_final*)

Bases: desdeo_tools.interaction.request.BaseRequest

A request to handle the termination of Synchronous NIMBUS.

Parameters

- **solutions_final** (*np.ndarray*) – A numpy array containing the final decision variable values.
- **objective_final** (*np.ndarray*) – A numpy array containing the final objective variables which correspond to
- **solution_final.** –

Note: This request expects no response.

ReferencePointMethod

class `desdeo_mcdm.interactive.`**`ReferencePointMethod`**(*problem*, *ideal*, *nadir*, *epsilon=1e-06*, *objective_names=None*, *minimize=None*)

Bases: `desdeo_mcdm.interactive.InteractiveMethod.InteractiveMethod`

Implements the Reference Point Method as presented in .

In the Reference Point Method, the Decision Maker (DM) specifies **desirable aspiration levels** for objective functions. Vectors formed of these aspiration levels are then used to derive scalarizing functions having minimal values at weakly, properly or Pareto optimal solutions. It is important that reference points are intuitive and easy for the DM to specify, their consistency is not an essential requirement. Before the solution process starts, some information is given to the DM about the problem. If possible, the ideal objective vector and the (approximated) nadir objective vector are presented.

At each iteration, the DM is asked to give desired aspiration levels for the objective functions. Using this information to formulate a **reference point**, achievement function is minimized and a (weakly, properly or) Pareto optimal solution is obtained. This solution is then presented to the DM. In addition, k other (weakly, properly or) Pareto optimal solutions are calculated using **perturbed reference points**, where k is the number of objectives in the problem. The alternative solutions are also presented to the DM. If (s)he finds any of the k + 1 solutions satisfactory, the solution process is ended. Otherwise, the DM is asked to present a new reference point and the iteration described above is repeated.

The idea in perturbed reference points is that the DM gets **better understanding** of the possible solutions around the current solution. If the reference point is far from the Pareto optimal set, the DM gets a **wider** description of the Pareto optimal set and if the reference point is near the Pareto optimal set, then a **finer** description of the Pareto optimal set is given.

In this method, the DM has to specify aspiration levels and compare objective vectors. The DM is **free to change** her/his mind during the process and can direct the solution process without being forced to understand complicated concepts and their meaning. On the other hand, the method does not necessarily help the DM to find more satisfactory solutions.

Parameters

- **`problem`** (*`MOProblem`*) – Problem to be solved.
- **`ideal`** (*`np.ndarray`*) – The ideal objective vector of the problem.
- **`nadir`** (*`np.ndarray`*) – The nadir objective vector of the problem. This may also be the "worst" objective vector provided by the Decision Maker if the approximation of Nadir vector is not applicable or if the Decision Maker wishes to provide even worse objective vector than what the approximated Nadir vector is.
- **`epsilon`** (*`float`*) – A small number used in calculating the utopian point.
- **`epsilon`** – A small number used in calculating the utopian point. Default value is 1e-6.
- **`objective_names`** (*`Optional[List[str]], optional`*) – Names of the objectives. The length of the list must match the number of elements in ideal vector.
- **`minimize`** (*`Optional[List[int]], optional`*) – Multipliers for each objective. '-1' indicates maximization and '1' minimization. Defaults to all objective values being minimized.

Raises ***`RPMException`*** – Dimensions of ideal, nadir, objective_names, and minimize-list do not match.

Methods Summary

`calculate_prp`(ref_point, f_current)	Calculate perturbed reference points.
`handle_initial_request`(request)	Handles the initial request by parsing the response appropriately.
`handle_request`(request)	Handle the Decision Maker's requests after the first iteration round, so-called **intermediate requests.**
`iterate`(request)	Perform the next logical iteration step based on the given request type.
`solve_asf`(ref_point, x0, ...[, ...])	Solve Achievement scalarizing function.
`start`()	Start the solution process with initializing the first request.

Methods Documentation

`calculate_prp`(*ref_point*, *f_current*)
Calculate perturbed reference points.

Parameters

- **`ref_point`** (*`np.ndarray`*) – Current reference point.
- **`f_current`** (*`np.ndarray`*) – Current solution.

Returns Perturbed reference points.

Return type np.ndarray

`handle_initial_request`(*request*)
Handles the initial request by parsing the response appropriately.

Parameters **`request`** (`RPMInitialRequest`) – Initial request including the Decision Maker's initial preferences.

Returns New request with updated solution process information.

Return type *RPMRequest*

`handle_request`(*request*)
Handle the Decision Maker's requests after the first iteration round, so-called **intermediate requests.**

Parameters **`request`** (`RPMRequest`) – Intermediate request including the Decision Maker's response.

Returns In case last iteration, request to stop the solution process. Otherwise, new request with updated solution process information.

Return type Union[*RPMRequest*, *RPMStopRequest*]

`iterate`(*request*)
Perform the next logical iteration step based on the given request type.

Parameters **`request`** (*`Union[RPMInitialRequest, RPMRequest]`*) – Either initial or intermediate request.

Returns A new request with content depending on the Decision Maker's preferences.

Return type Union[*RPMRequest*, *RPMStopRequest*]

solve_asf(*ref_point*, *x0*, *preferential_factors*, *nadir*, *utopian*, *objectives*, *variable_bounds=None*, *method=None*)

Solve Achievement scalarizing function.

Parameters

- **ref_point** (*np.ndarray*) – Reference point.
- **x0** (*np.ndarray*) – Initial values for decision variables.
- **preferential_factors** (*np.ndarray*) – Preferential factors on how much would the Decision Maker wish to improve the values of each objective function.
- **nadir** (*np.ndarray*) – Nadir vector.
- **utopian** (*np.ndarray*) – Utopian vector.
- **objectives** (*np.ndarray*) – The objective function values for each input vector.
- **variable_bounds** (*Optional[np.ndarray)*) – Lower and upper bounds of each variable as a 2D numpy array. If undefined variables, None instead.
- **method** (*Union[ScalarMethod, str, None]*) – The optimization method the scalarizer should be minimized with.

Returns A dictionary with at least the following entries: 'x' indicating the optimal variables found, 'fun' the optimal value of the optimized function, and 'success' a boolean indicating whether the optimization was conducted successfully.

Return type dict

start()

Start the solution process with initializing the first request.

Returns Initial request.

Return type *RPMInitialRequest*

RPMException

exception desdeo_mcdm.interactive.**RPMException**

Raised when an exception related to Reference Point Method (RFM) is encountered.

RPMInitialRequest

class desdeo_mcdm.interactive.**RPMInitialRequest**(*ideal*, *nadir*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle the Decision Maker's initial preferences for the first iteration round.

Attributes Summary

response

Methods Summary

init_with_method(method)	Initialize request with given instance of ReferencePointMethod.

Attributes Documentation

response

Methods Documentation

classmethod init_with_method(*method*)

Initialize request with given instance of ReferencePointMethod.

Parameters **method** (ReferencePointMethod) – Instance of ReferencePointMethod-class.

Returns Initial request.

Return type *RPMInitialRequest*

RPMRequest

class desdeo_mcdm.interactive.**RPMRequest**(*f_current*, *f_additionals*, *ideal*, *nadir*)

Bases: desdeo_tools.interaction.request.BaseRequest

A request class to handle the Decision Maker's preferences after the first iteration round.

Attributes Summary

response

Attributes Documentation

response

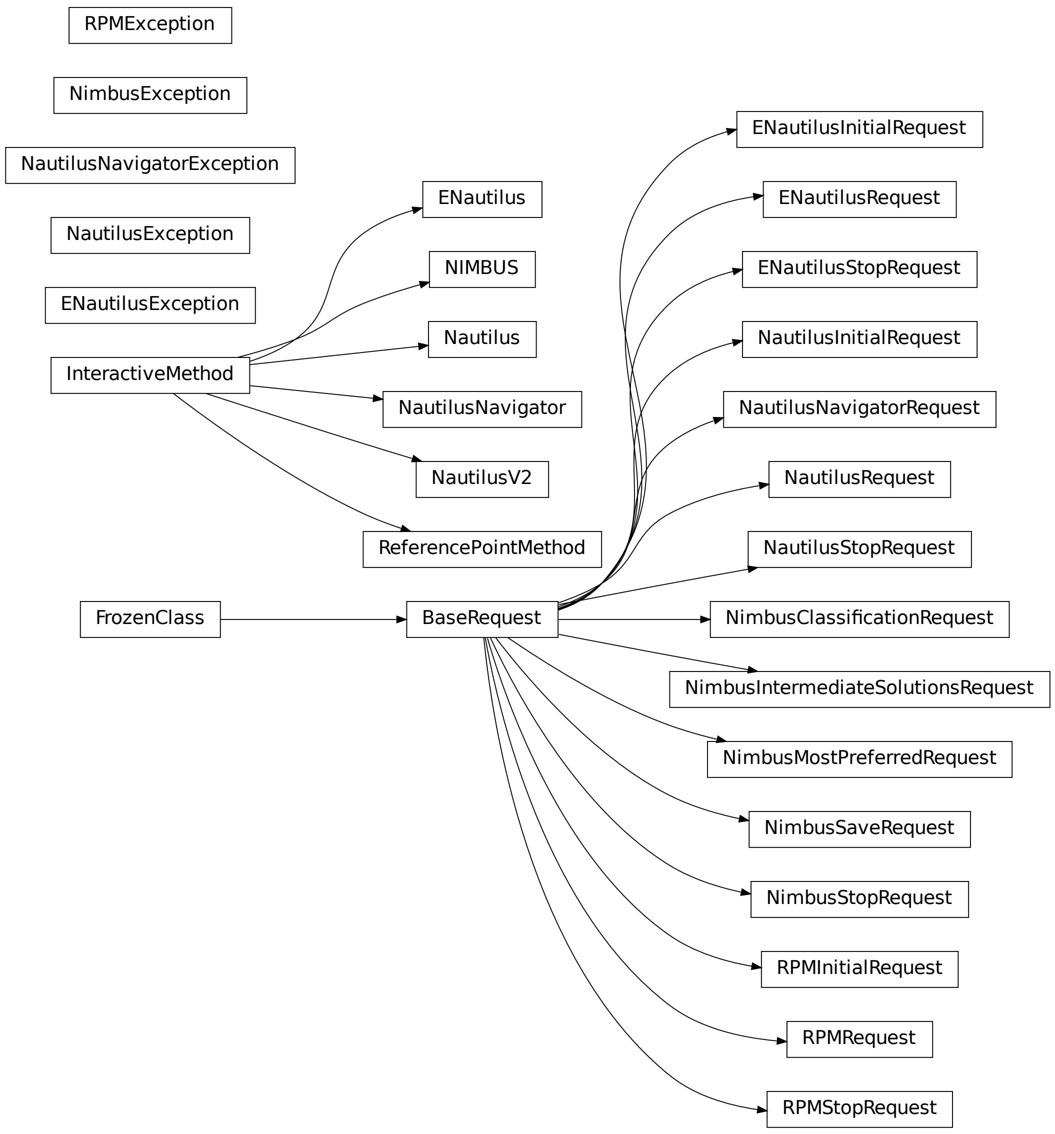
RPMStopRequest

class desdeo_mcdm.interactive.**RPMStopRequest**(*x_h*, *f_h*)

Bases: `desdeo_tools.interaction.request.BaseRequest`

A request class to handle termination.

Class Inheritance Diagram



desdeo_mcdm.utilities Package

This module contains various utilities used in different interactive methods.

Functions

payoff_table_method(problem[, …])	Uses the payoff table method to solve for the ideal and nadir points of a MOProblem.
payoff_table_method_general(…[, …])	Solves a representation for the nadir and ideal points for a multiobjective minimization problem with objectives defined as the result of some objective evaluator.
solve_pareto_front_representation(problem[, …])	Pass through to solve_pareto_front_representation_general when the problem for which the front is being calculated for is defined as an MOProblem object.
solve_pareto_front_representation_general(…)	Computes a representation of a Pareto efficient front from a multiobjective minimizatino problem.
weighted_scalarizer(xs, ws)	A simple linear weight based scalarizer.

payoff_table_method

`desdeo_mcdm.utilities.`**`payoff_table_method`**(*problem*, *initial_guess=None*, *solver_method='scipy_de'*)

Uses the payoff table method to solve for the ideal and nadir points of a MOProblem. Call through to payoff_table_method_general.

Parameters

- **`problem`** (*`MOProblem`*) – The problem defined as a MOProblem class instance.
- **`initial_guess`** (*`Optional[np.ndarray]`*) – The initial guess of decision variables to be used in the solver. If None, uses the lower bounds defined for the variables in MOProblem. Defaults to None.
- **`solver_method`** (*`Optional[Union[ScalarMethod, str]]`*) – The method used to minimize the invidual problems in the payoff table method. Defaults to 'scipy_de'.

Returns The ideal and nadir points

Return type Tuple[np.ndarray, np.ndarray]

payoff_table_method_general

`desdeo_mcdm.utilities.`**`payoff_table_method_general`**(*objective_evaluator*, *n_of_objectives*, *variable_bounds*, *constraint_evaluator=None*, *initial_guess=None*, *solver_method='scipy_de'*)

Solves a representation for the nadir and ideal points for a multiobjective minimization problem with objectives defined as the result of some objective evaluator.

Parameters

- **`objective_evaluator`** (*`Callable[[np.ndarray], np.ndarray]`*) – The evaluator which returns the objective values given a set of variabels.

- **n_of_objectives** (*int*) – Number of objectives returned by calling objective_evaluator.
- **variable_bounds** (*np.ndarray*) – The lower and upper bounds of the variables passed as argument to objective_evaluator. Should be a 2D numpy array with the limits for each variable being on each row. The first column should contain the lower bounds, and the second column the upper bounds. Use np.inf to indicate no bounds.
- **constraint_evaluator** (*Optional[Callable[[np.ndarray], np.ndarray]], optional*) – An evaluator accepting the same arguments as objective_evaluator, which returns the constraint values of the multiobjective minimization problem being solved. A negative constraint value indicates a broken constraint. Defaults to None.
- **initial_guess** (*Optional[np.ndarray], optional*) – The initial guess used for the variable values while solving the payoff table. The relevancy of this parameter depends on the solver_method being used. Defaults to None.
- **solver_method** (*Optional[Union[ScalarMethod, str]], optional*) – The method to solve the scalarized problems in the payoff table method. Defaults to "scipy_de", which ignores initial_guess.

Returns The representations computed using the payoff table for the ideal and nadir points respectively.

Return type Tuple[np.ndarray, np.ndarray]

solve_pareto_front_representation

desdeo_mcdm.utilities.**solve_pareto_front_representation**(*problem*, *step=0.1*, *eps=1e-06*, *solver_method='scipy_de'*)

Pass through to solve_pareto_front_representation_general when the problem for which the front is being calculated for is defined as an MOProblem object.

Computes a representation of a Pareto efficient front from a multiobjective minimizatino problem. Does so by generating an evenly spaced set of reference points (in the objective space), in the space spanned by the supplied ideal and nadir points. The generated reference points are then used to formulate achievement scalaraization problems, which when solved, yield a representation of a Pareto efficient solution.

Parameters

- **problem** (*MOProblem*) – The multiobjective minimization problem for which the front is to be solved for.
- **step** (*Optional[Union[np.ndarray, float]], optional*) – Either a float or an array of floats. If a single float is given, generates reference points with the objectives having values a step apart between the ideal and nadir points. If an array of floats is given, use the steps defined in the array for each objective's values. Default to 0.1.
- **eps** (*Optional[float], optional*) – An offset to be added to the nadir value to keep the nadir inside the range when generating reference points. Defaults to 1e-6.
- **solver_method** (*Optional[Union[ScalarMethod, str]], optional*) – The method used to minimize the achievement scalarization problems arising when calculating Pareto efficient solutions. Defaults to "scipy_de".

Returns A tuple containing representations of the Pareto optimal variable values, and the correspon-sing objective values.

Return type Tuple[np.ndarray, np.ndarray]

solve_pareto_front_representation_general

`desdeo_mcdm.utilities.`**`solve_pareto_front_representation_general`**(*objective_evaluator*, *n_of_objectives*, *variable_bounds*, *step=0.1*, *eps=1e-06*, *ideal=None*, *nadir=None*, *constraint_evaluator=None*, *solver_method='scipy_de'*)

Computes a representation of a Pareto efficient front from a multiobjective minimizatino problem. Does so by generating an evenly spaced set of reference points (in the objective space), in the space spanned by the supplied ideal and nadir points. The generated reference points are then used to formulate achievement scalaraization problems, which when solved, yield a representation of a Pareto efficient solution. The result is guaranteed to contain only non-dominated solutions.

Parameters

- **`objective_evaluator`** (*`Callable[[np.ndarray], np.ndarray]`*) – A vector valued function returning objective values given an array of decision variables.
- **`n_of_objectives`** (*`int`*) – Numbr of objectives returned by objective_evaluator.
- **`variable_bounds`** (*`np.ndarray`*) – The upper and lower bounds of the decision variables. Bound for each variable should be on the rows, with the first column containing lower bounds, and the second column upper bounds. Use np.inf to indicate no bounds.
- **`step`** (*`Optional[Union[np.ndarray, float]], optional`*) – Etiher an float or an array of floats. If a single float is given, generates reference points with the objectives having values a step apart between the ideal and nadir points. If an array of floats is given, use the steps defined in the array for each objective's values. Default to 0.1.
- **`eps`** (*`Optional[float], optional`*) – An offset to be added to the nadir value to keep the nadir inside the range when generating reference points. Defaults to 1e-6.
- **`ideal`** (*`Optional[np.ndarray], optional`*) – The ideal point of the problem being solved. Defaults to None.
- **`nadir`** (*`Optional[np.ndarray], optional`*) – The nadir point of the problem being solved. Defaults to None.
- **`constraint_evaluator`** (*`Optional[Callable[[np.ndarray], np.ndarray]], optional`*) – An evaluator returning values for the constraints defined for the problem. A negative value for a constraint indicates a breach of that constraint. Defaults to None.
- **`solver_method`** (*`Optional[Union[ScalarMethod, str]], optional`*) – The method used to minimize the achievement scalarization problems arising when calculating Pareto efficient solutions. Defaults to "scipy_de".

Raises

- **`MCDMUtilityException`** – Mismatching sizes of the supplied ideal and

- **nadir points between the step, when step is an array. Or the type of –**
- **step is something else than np.ndarray of float. –**

Returns A tuple containing representationns of the Pareto optimal variable values, and the corresponsing objective values.

Return type Tuple[np.ndarray, np.ndarray]

Note: The objective evaluator should be defined such that minimization is expected in each of the objectives.

weighted_scalarizer

`desdeo_mcdm.utilities.`**`weighted_scalarizer`**`(`*xs*, *ws*`)`

A simple linear weight based scalarizer.

Parameters

- **xs** (*np.ndarray*) – Values to be scalarized.
- **ws** (*np.ndarray*) – Weights to multiply each value in the summation of xs.

Returns An array of scalar values with length equal to the first dimension of xs.

Return type np.ndarray

2.2.3 Examples

NAUTILUS Navigator example

This example goes through the basic functionalities of the NAUTILUS Navigator method.

We will consider a simple 2D Pareto front which we will define next alongside the method itself. Both objectives are to be minimized.

Because of the nature of navigation based interactive optimization methods, the idea of NAUTILUS Navigator is best demonstrated using some graphical user interface. One such interface can be found online.

[1]:
```
import numpy as np
from desdeo_mcdm.interactive.NautilusNavigator import NautilusNavigator

# half of a parabola to act as a Pareto front
f1 = np.linspace(1, 100, 50)
f2 = f1[::-1] ** 2

front = np.stack((f1, f2)).T
ideal = np.min(front, axis=0)
nadir = np.max(front, axis=0)

method = NautilusNavigator((front), ideal, nadir)
```

To start, we can invoke the `start` method.

```
[2]: req_first = method.start()

     print(req_first)
     print(req_first.content.keys())
```

```
<desdeo_mcdm.interactive.NautilusNavigator.NautilusNavigatorRequest object at␣
↪0x7ff8807cfd60>
dict_keys(['message', 'ideal', 'nadir', 'reachable_lb', 'reachable_ub', 'reachable_idx
↪', 'step_number', 'steps_remaining', 'distance', 'allowed_speeds', 'current_speed',
↪'navigation_point'])
```

The returned object is a NautilusNavigatorRequest. The keys should give an idea of what the contents of the request are. We will explain most of the in this example.

At the moment, the `nadir`, `reachable_lb` and `reachable_ub` are most interesting to us. Navigation starts from the nadir and will proceed toward the Pareto optimal front enclosed between the limits defined in `reachable_lb` and `reachable_ub`.

To interact with the method, we must fill out the `response` member of `req`. Let's see the contents of the message in `req` next.

```
[3]: print(req_first.content["message"])
```

```
Please supply aspirations levels for each objective between the upper and lower␣
↪bounds as `reference_point`. Specify a speed between 1-5 as `speed`. If going to a␣
↪previous step is desired, please set `go_to_previous` to True, otherwise it should␣
↪be False. Lastly, if stopping is desired, `stop` should be True, otherweise it␣
↪should be set to False.
```

We should define the required values and set them as keys of a dictionary. Before that, it is useful to see the bounds to know the currently feasible objective values.

```
[4]: print(req_first.content["reachable_lb"])
     print(req_first.content["reachable_ub"])
```

```
[1. 1.]
[  100. 10000.]
```

```
[5]: reference_point = np.array([50, 6000])
     go_to_previous = False
     stop = False
     speed = 1

     response = dict(reference_point=reference_point, go_to_previous=False, stop=False,␣
     ↪speed=1)
```

`go_to_previous` should be set to `False` unless we desire going to a previous point. `stop` should be `True` if we wish to stop, otherwise it should be `False`. `speed` is the speed of the navigation. It is not used internally in the method. To continue, we call `iterate` with suppliying the `req` object with a defined `response` attribute. We should get a new request as a return value.

```
[6]: req_first.response = response
     req_snd = method.iterate(req_first)

     print(req_snd.content["reachable_lb"])
     print(req_snd.content["reachable_ub"])
```

```
[3.02040816 9.12286547]
[  100. 10000.]
```

We see that the bounds have narrowed down as they should.

In reality, `iterate` should be called multiple times in succession with the same `response` contents. We can do this in a loop until the 30th step is computed, for example. NB: Steps are internally zero-index based.

[7]:
```
previous_requests = [req_first, req_snd]
req = req_snd
while method._step_number < 30:
    req.response = response
    req = method.iterate(req)

    previous_requests.append(req)

print(req.content["reachable_lb"])
print(req.content["reachable_ub"])
print(req.content["step_number"])
```

```
[ 11.10204082 449.61307788]
[  81.81632653 8081.64306539]
30
```

The region of reachable Pareto optimal solutions had narrowed down. Suppose now we wish to return to a previous step and change our preferences. Let’s say, step 14.

[8]:
```
# fetch the 14th step saved previously
req_14 = previous_requests[13]
print(req_14.content["reachable_lb"])
print(req_14.content["reachable_ub"])
print(req_14.content["step_number"])

req_14.response["go_to_previous"] = True
req_14.response["reference_point"] = np.array([50, 5000])
new_response = req_14.response
```

```
[  5.04081633 123.25531029]
[  91.91836735 9208.16493128]
14
```

When going to a previous point, the method assumes thath the state the method was in during that point is fully defined in the request object given to it when calling `iterate` with `go_to_previous` being `True`. This is why we saved the request previously in a list.

[9]:
```
req_14_new = method.iterate(req_14)
req = req_14_new

# remember to unser go_to_previous!
new_response["go_to_previous"] = False

# continue iterating for 16 steps
while method._step_number < 30:
    req.response = new_response
    req = method.iterate(req)

print("Old 30th step")
print(previous_requests[29].content["reachable_lb"])
print(previous_requests[29].content["reachable_ub"])
print(previous_requests[29].content["step_number"])

print("New 30th step")
```

(continues on next page)

(continued from previous page)

```
print(req.content["reachable_lb"])
print(req.content["reachable_ub"])
print(req.content["step_number"])
```

```
Old 30th step
[ 11.10204082 449.61307788]
[  81.81632653 8081.64306539]
30
New 30th step
[ 11.10204082 368.01332778]
[  81.81632653 8081.64306539]
30
```

We can see a difference in the limits when we changed the preference point.

To find the final solution, we can iterate till the end.

```
[10]: while method._step_number < 100:
          req.response = new_response
          req = method.iterate(req)

      print(req.content["reachable_idx"])
```

```
19
```

When finished navigating, the method will return the index of the reached solution based on the supplied Pareto front. It is assumed that if decision variables also exist for the problem, they are stored elwhere. The final index returned can then be used to find the corresponding decision variables to the found solution in objective space.

```
[ ]:
```

Example on the usage of NIMBUS

This notebook will go through a simple example to illustrate how the synchronous variant of NIMBUS has been implemented in the DESDEO framework.

We will be solving the Kursawe function originally defined in this article

Let us begin by importing some libraries and defining the problem.

```
[1]: import numpy as np

     import matplotlib.pyplot as plt
     from desdeo_problem.Problem import MOProblem
     from desdeo_problem.Variable import variable_builder
     from desdeo_problem.Objective import _ScalarObjective

     def f_1(xs: np.ndarray):
         xs = np.atleast_2d(xs)
         xs_plusone = np.roll(xs, 1, axis=1)
         return np.sum(-10*np.exp(-0.2*np.sqrt(xs[:, :-1]**2 + xs_plusone[:, :-1]**2)),␣
     ↪axis=1)

     def f_2(xs: np.ndarray):
         xs = np.atleast_2d(xs)
         return np.sum(np.abs(xs)**0.8 + 5*np.sin(xs**3), axis=1)
```

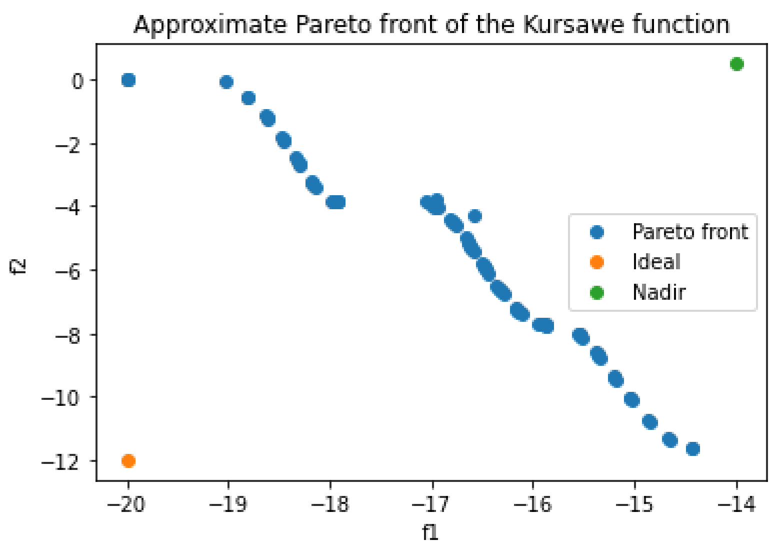
(continues on next page)

(continued from previous page)

```
varsl = variable_builder(
    ["x_1", "x_2", "x_3"],
    initial_values=[0, 0, 0],
    lower_bounds=[-5, -5, -5],
    upper_bounds=[5, 5, 5],
)

f1 = _ScalarObjective(name="f1", evaluator=f_1)
f2 = _ScalarObjective(name="f2", evaluator=f_2)

problem = MOProblem(variables=varsl, objectives=[f1, f2], ideal=np.array([-20, -12]),␣
↪nadir=np.array([-14, 0.5]))
```

To check out problem, let us compute a representation of the Pareto optimal front of solutions:

```
[2]: from desdeo_mcdm.utilities.solvers import solve_pareto_front_representation

p_front = solve_pareto_front_representation(problem, step=1.0)[1]

plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
plt.xlabel("f1")
plt.ylabel("f2")
plt.title("Approximate Pareto front of the Kursawe function")
plt.legend()
plt.show()
```



Now we can get to the NIMBUS part. Let us define an instance of the NIMBUS method utilizing our problem defined earlier, and start by invoking the instance's `start` method:

```
[3]: from desdeo_mcdm.interactive.NIMBUS import NIMBUS

method = NIMBUS(problem, "scipy_de")

classification_request, plot_request = method.start()
```

Let us look at the keys in the dictionary contained in the `classification_request`:

[4]:
```
print(classification_request.content.keys())
```

```
dict_keys(['message', 'objective_values', 'classifications', 'levels', 'number_of_
↪solutions'])
```

Message should give us some more information:

[5]:
```
print(classification_request.content["message"])
```

```
Please classify each of the objective values in one of the following categories:
        1. values should improve '<'
        2. values should improve until some desired aspiration level is reached '<='
        3. values with an acceptable level '='
        4. values which may be impaired until some upper bound is reached '>='
        5. values which are free to change '0'
Provide the aspiration levels and upper bounds as a vector. For categories 1, 3, and␣
↪5,the value in the vector at the objective's position is ignored. Suppy also the␣
↪number of maximumsolutions to be generated.
```

We should therefore classify each of the objectives found beind the `objective_values` -key in the dictionary in `classification_request.content`. Let's print them:

[6]:
```
print(classification_request.content["objective_values"])
```

```
[-15.88641547  -7.74757837]
```

Instead of printing the values, we could have also used the `plot_request` object. However, we are inspecting only one set of objective values for the time being, so a raw print of the values should be enough. Let us classify the objective values next. We can get a hint of what the classification should look like by inspecting the value found using the `classifications` -key in `classification_request.content`:

[7]:
```
print(classification_request.content["classifications"])
```

```
[None]
```

Therefore it should be a list. Suppose we wish to improve (decrease in value) the first objective, and impair (increase in value) the second objective till some upper bound is reached. We should define our preferences as a dictionary `classification_request.response` with the keys `classifications` and `number_of_solutions` (we have to define the number of new solutions we wish to compute). The key `levels` will contain the upper bound for the second objective.

[8]:
```
response = {
    "classifications": ["<", ">="],
    "number_of_solutions": 3,
    "levels": [0, -5]
}
classification_request.response = response
```

To continue, just feed `classification_request` back to the method through the `step` method:

[9]:
```
save_request, plot_request = method.iterate(classification_request)
```

We got a new request as a response. Let us inspect it:

[10]:
```
print(save_request.content.keys())
print(save_request.content["message"])
print(save_request.content["objectives"])
```

```
dict_keys(['message', 'solutions', 'objectives', 'indices'])
Please specify which solutions shown you would like to save for later viewing. Supply␣
↪the indices of such solutions as a list, or supply an empty list if none of the␣
↪shown soulutions should be saved.
[array([-1.99999999e+01,  2.34193302e-06]), array([-1.99999998e+01,  3.34684282e-06]),
↪ array([-18.46633851,  -1.81760788])]
```

Suppose the first and last solutions result in nice objective values.

```
[11]: response = {"indices": [0, 2]}
      save_request.response = response

      intermediate_request, plot_request = method.iterate(save_request)
```

```
[12]: print(intermediate_request.content.keys())
      print(intermediate_request.content["message"])
```

```
dict_keys(['message', 'solutions', 'objectives', 'indices', 'number_of_desired_
↪solutions'])
Would you like to see intermediate solutions between two previously computed solutions?
↪ If so, please supply two indices corresponding to the solutions.
```

We do not desire to see intermediate results.

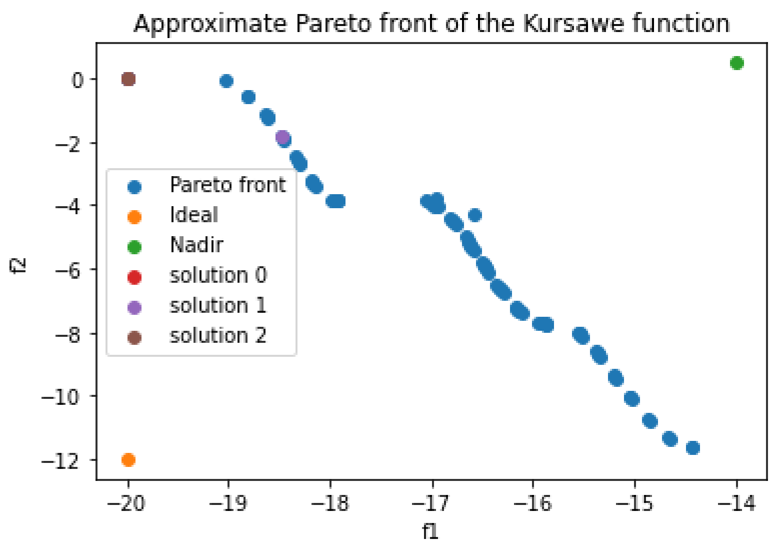
```
[13]: response = {"number_of_desired_solutions": 0, "indices": []}
      intermediate_request.response = response

      preferred_request, plot_request = method.iterate(intermediate_request)
```

```
[14]: print(preferred_request.content.keys())
      print(preferred_request.content["message"])
```

```
dict_keys(['message', 'solutions', 'objectives', 'index', 'continue'])
Please select your most preferred solution and whether you would like to continue.
```

We should select our most preferred solution. Let us plot the objective values to inspect them better:

```
[15]: plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
      plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
      plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
      for i, z in enumerate(preferred_request.content["objectives"]):
          plt.scatter(z[0], z[1], label=f"solution {i}")
      plt.xlabel("f1")
      plt.ylabel("f2")
      plt.title("Approximate Pareto front of the Kursawe function")
      plt.legend()
      plt.show()
```



Solutions at indices 0 and 2 seem to be overlapping in the objective space. We decide to select the solution at index 1, and to continue the iterations.

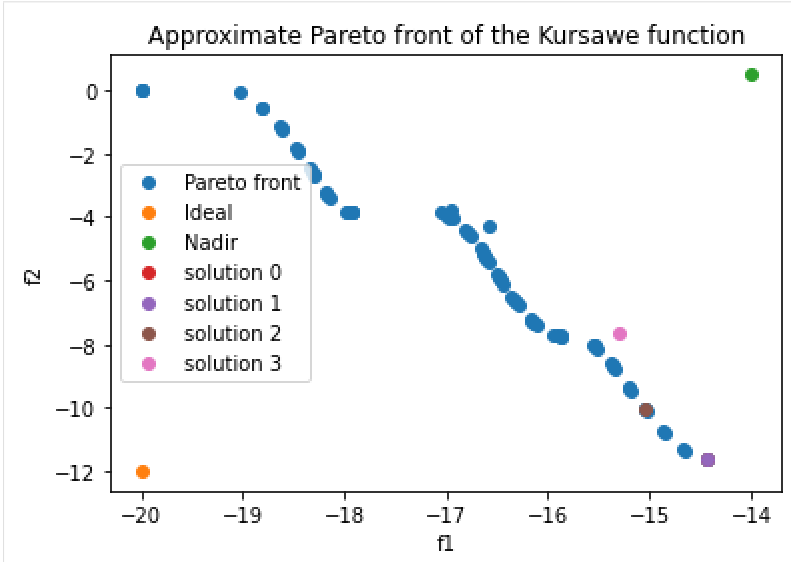
[16]:
```
response = {"index": 1, "continue": True}
preferred_request.response = response

classification_request, plot_request = method.iterate(preferred_request)
```

Back at the classification pahse of the NIMBUS method.

[17]:
```
response = {
    "classifications": [">=", "<"],
    "number_of_solutions": 4,
    "levels": [-16, -1]
}
classification_request.response = response

save_request, plot_request = method.iterate(classification_request)
```

Let us plot some of the solutions again:

[18]:
```
plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
for i, z in enumerate(save_request.content["objectives"]):
    plt.scatter(z[0], z[1], label=f"solution {i}")
plt.xlabel("f1")
plt.ylabel("f2")
plt.title("Approximate Pareto front of the Kursawe function")
plt.legend()
plt.show()
```

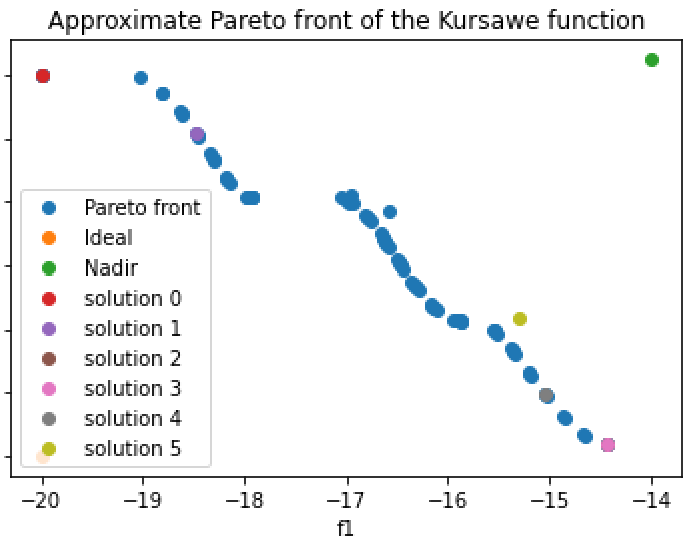


NIMBUS really took to heart our request to detoriate the first objective… Suppose we like all of the solutions:

[19]:
```
response = {"indices": [0, 1, 2, 3]}
save_request.response = response

intermediate_request, plot_request = method.iterate(save_request)
```

Let us plot everything we have so far:

[20]:
```
plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
for i, z in enumerate(intermediate_request.content["objectives"]):
    plt.scatter(z[0], z[1], label=f"solution {i}")
plt.xlabel("f1")
plt.ylabel("f2")
plt.title("Approximate Pareto front of the Kursawe function")
plt.legend()
plt.show()
```

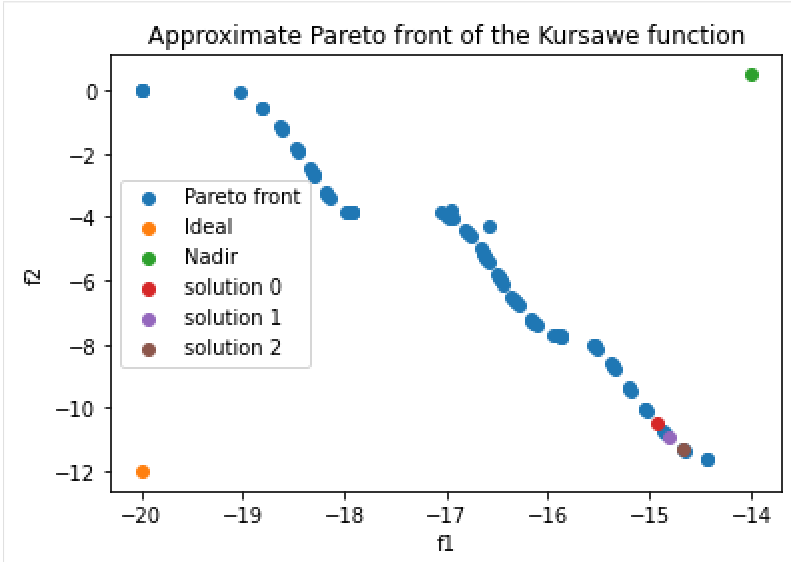


Assume we really like what we have between solution 3 and 4. Let NIMBUS compute 3 intermediate solutions between them:

[21]:
```
response = {
    "indices": [3, 4],
    "number_of_desired_solutions": 3,
    }
intermediate_request.response = response

save_request, plot_request = method.iterate(intermediate_request)
```

Plot the intermediate solutions:

[22]:
```
plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
for i, z in enumerate(save_request.content["objectives"]):
    plt.scatter(z[0], z[1], label=f"solution {i}")
plt.xlabel("f1")
plt.ylabel("f2")
plt.title("Approximate Pareto front of the Kursawe function")
plt.legend()
plt.show()
```



Nice, we are really getting there, even if we have no goal set… Let us save solution 1:

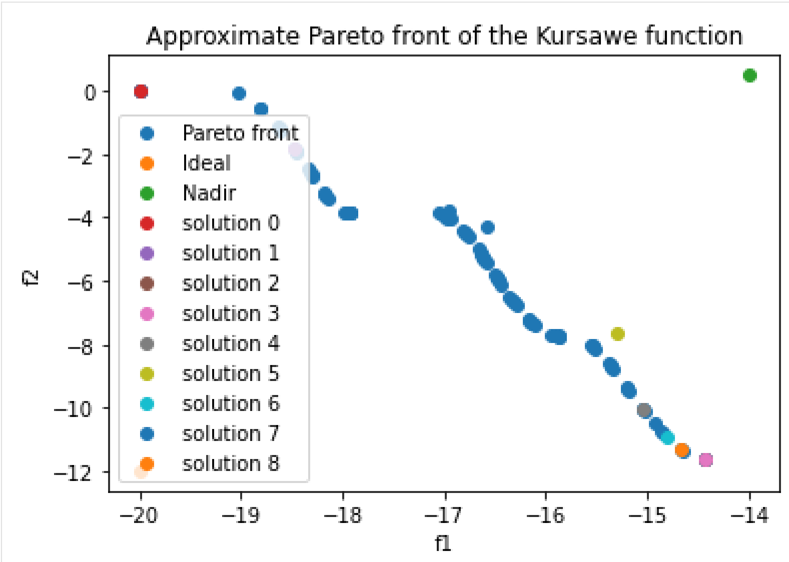
[23]:
```
response = {"indices": [1]}
save_request.response = response

intermediate_request, plot_request = method.iterate(save_request)
```

We do not wish to generate any more intermediate solutions.

[24]:
```
response = {"number_of_desired_solutions": 0, "indices": []}
intermediate_request.response = response

preferred_request, plot_request = method.iterate(intermediate_request)
```

Let us plot everything we have, and select a final solution:

[25]:
```
plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
for i, z in enumerate(preferred_request.content["objectives"]):
    plt.scatter(z[0], z[1], label=f"solution {i}")
plt.xlabel("f1")
plt.ylabel("f2")
plt.title("Approximate Pareto front of the Kursawe function")
plt.legend()
plt.show()
```



We REALLY like solution 6, so let us go with that:

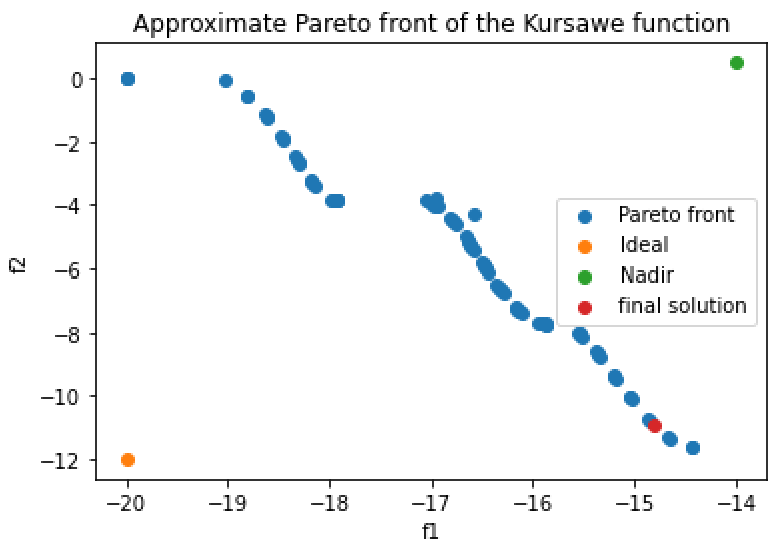
```
[26]: response = {
          "index": 6,
          "continue": False,
          }

      preferred_request.response = response

      print("hello")
      print(preferred_request)

      stop_request, plot_request = method.iterate(preferred_request)

      print(stop_request)
```

```
hello
<desdeo_mcdm.interactive.NIMBUS.NimbusMostPreferredRequest object at 0x7f5078418940>
<desdeo_mcdm.interactive.NIMBUS.NimbusStopRequest object at 0x7f507a67b670>
```

We are done, let us bask in the glory of the solution found:

```
[27]: print(f"Final decision variables: {stop_request.content['solution']}")

      plt.scatter(p_front[:, 0], p_front[:, 1], label="Pareto front")
      plt.scatter(problem.ideal[0], problem.ideal[1], label="Ideal")
      plt.scatter(problem.nadir[0], problem.nadir[1], label="Nadir")
      plt.scatter(stop_request.content["objective"][0], stop_request.content["objective
      ↪"][1], label=f"final solution")
      plt.xlabel("f1")
      plt.ylabel("f2")
      plt.title("Approximate Pareto front of the Kursawe function")
      plt.legend()
      plt.show()
```

```
Final decision variables: [-1.02767713 -1.09988959 -1.09984277]
```


Approximate Pareto front of the Kursawe function
f1
f2
0
−2
−4
−6
−8
−10
−12
−20
−19
−18
−17
−16
−15
−14
Pareto front
Ideal
Nadir
final solution

CHAPTER

THREE

CURRENTLY IMPLEMENTED METHODS

Algorithm	Reference
Synchronous NIMBUS	Miettinen, K., Mäkelä, M.M.: Synchronous approach in interactive multiobjective optimization. Eur. J. Oper. Res. 170(3), 909–922 (2006)
NAUTILUS Navigator	Ruiz, A. B., Ruiz, F., Miettinen, K., Delgado-Antequera, L., & Ojalehto, V. (2019). NAUTILUS Navigator : free search interactive multiobjective optimization without trading-off. Journal of Global Optimization, 74 (2), 213-231. doi:10.1007/s10898-019-00765-2
E-NAUTILUS	Ruiz, A., Sindhya, K., Miettinen, K., Ruiz, F., & Luque, M. (2015). E-NAUTILUS: A decision support system for complex multiobjective optimization problems based on the NAUTILUS method. European Journal of Operational Research, 246 (1), 218-231. doi:10.1016/j.ejor.2015.04.027
NAUTILUS	Kaisa Miettinen, Petri Eskelinen, Francisco Ruiz, Mariano Luque, NAUTILUS method: An interactive technique in multiobjective optimization based on the nadir point, European Journal of Operational Research, Volume 206, Issue 2, 2010, Pages 426-434, ISSN 0377-2217, https://doi.org/10.1016/j.ejor.2010.02.041.
Reference Point Method	Andrzej P. Wierzbicki, A mathematical basis for satisficing decision making, Mathematical Modelling, Volume 3, Issue 5,1982, Pages 391-405, ISSN 0270-0255, https://doi.org/10.1016/0270-0255(82)90038-0.
NAUTILUSv2	Miettinen, K., Podkopaev, D., Ruiz, F. et al. A new preference handling technique for interactive multiobjective optimization without trading-off. J Glob Optim 63, 633–652 (2015). https://doi.org/10.1007/s10898-015-0301-8

CHAPTER

FOUR

COMING SOON

- Pareto Navigator

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

desdeo_mcdm.interactive, 7

desdeo_mcdm.utilities, 42

INDEX

Symbols

_valid_classifications (*desdeo_mcdm.interactive.NimbusClassificationRequest.self attribute*), 32

C

calculate_bounds() (*desdeo_mcdm.interactive.ENautilus method*), 11
calculate_bounds() (*desdeo_mcdm.interactive.Nautilus method*), 15
calculate_bounds() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
calculate_bounds() (*desdeo_mcdm.interactive.NautilusV2 method*), 20
calculate_distance() (*desdeo_mcdm.interactive.Nautilus method*), 15
calculate_distance() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
calculate_distance() (*desdeo_mcdm.interactive.NautilusV2 method*), 20
calculate_distances() (*desdeo_mcdm.interactive.ENautilus method*), 11
calculate_doi() (*desdeo_mcdm.interactive.NautilusV2 method*), 20
calculate_intermediate_points() (*desdeo_mcdm.interactive.ENautilus method*), 11
calculate_iteration_point() (*desdeo_mcdm.interactive.Nautilus method*), 16
calculate_iteration_point() (*desdeo_mcdm.interactive.NautilusV2 method*), 20
calculate_navigation_point() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
calculate_new_solutions() (*desdeo_mcdm.interactive.NIMBUS method*), 29
calculate_preferential_factors() (*desdeo_mcdm.interactive.Nautilus method*), 16
calculate_preferential_factors() (*desdeo_mcdm.interactive.NautilusV2 method*), 20
calculate_prp() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 38
calculate_reachable_point_indices() (*desdeo_mcdm.interactive.ENautilus method*), 11
calculate_reachable_point_indices() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
calculate_representative_points() (*desdeo_mcdm.interactive.ENautilus method*), 11
compute_intermediate_solutions() (*desdeo_mcdm.interactive.NIMBUS method*), 29
create_plot_request() (*desdeo_mcdm.interactive.NIMBUS method*), 30

D

desdeo_mcdm.interactive
module, 7
desdeo_mcdm.utilities
module, 42

E

ENautilus (*class in desdeo_mcdm.interactive*), 10
ENautilusException, 12
ENautilusInitialRequest (*class in desdeo_mcdm.interactive*), 12
ENautilusRequest (*class in desdeo_mcdm.interactive*), 13

ENautilusStopRequest (*class in desdeo_mcdm.interactive*), 13

H

handle_classification_request() (*desdeo_mcdm.interactive.NIMBUS method*), 30
handle_initial_request() (*desdeo_mcdm.interactive.ENautilus method*), 11
handle_initial_request() (*desdeo_mcdm.interactive.Nautilus method*), 17
handle_initial_request() (*desdeo_mcdm.interactive.NautilusV2 method*), 21
handle_initial_request() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 38
handle_intermediate_solutions_request() (*desdeo_mcdm.interactive.NIMBUS method*), 30
handle_most_preferred_request() (*desdeo_mcdm.interactive.NIMBUS method*), 30
handle_request() (*desdeo_mcdm.interactive.ENautilus method*), 11
handle_request() (*desdeo_mcdm.interactive.Nautilus method*), 17
handle_request() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
handle_request() (*desdeo_mcdm.interactive.NautilusV2 method*), 21
handle_request() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 38
handle_save_request() (*desdeo_mcdm.interactive.NIMBUS method*), 30

I

init_with_method() (*desdeo_mcdm.interactive.ENautilusInitialRequest class method*), 13
init_with_method() (*desdeo_mcdm.interactive.NautilusInitialRequest class method*), 23
init_with_method() (*desdeo_mcdm.interactive.NautilusNavigatorRequest class method*), 28
init_with_method() (*desdeo_mcdm.interactive.RPMInitialRequest class method*), 40
iterate() (*desdeo_mcdm.interactive.ENautilus method*), 12
iterate() (*desdeo_mcdm.interactive.Nautilus method*), 17
iterate() (*desdeo_mcdm.interactive.NautilusNavigator method*), 25
iterate() (*desdeo_mcdm.interactive.NautilusV2 method*), 22
iterate() (*desdeo_mcdm.interactive.NIMBUS method*), 31
iterate() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 38

M

module
    desdeo_mcdm.interactive, 7
    desdeo_mcdm.utilities, 42

N

Nautilus (*class in desdeo_mcdm.interactive*), 14
NautilusException, 23
NautilusInitialRequest (*class in desdeo_mcdm.interactive*), 23
NautilusNavigator (*class in desdeo_mcdm.interactive*), 24
NautilusNavigatorException, 27
NautilusNavigatorRequest (*class in desdeo_mcdm.interactive*), 27
NautilusRequest (*class in desdeo_mcdm.interactive*), 23
NautilusStopRequest (*class in desdeo_mcdm.interactive*), 24
NautilusV2 (*class in desdeo_mcdm.interactive*), 18
NIMBUS (*class in desdeo_mcdm.interactive*), 28
NimbusClassificationRequest (*class in desdeo_mcdm.interactive*), 32
NimbusException, 32
NimbusIntermediateSolutionsRequest (*class in desdeo_mcdm.interactive*), 33
NimbusMostPreferredRequest (*class in desdeo_mcdm.interactive*), 34
NimbusSaveRequest (*class in desdeo_mcdm.interactive*), 35
NimbusStopRequest (*class in desdeo_mcdm.interactive*), 36

P

payoff_table_method() (*in module desdeo_mcdm.utilities*), 42
payoff_table_method_general() (*in module desdeo_mcdm.utilities*), 42

R

ReferencePointMethod (*class in desdeo_mcdm.interactive*), 37

request_classification() (*desdeo_mcdm.interactive.NIMBUS method*), 31

request_most_preferred_solution() (*desdeo_mcdm.interactive.NIMBUS method*), 31

request_stop() (*desdeo_mcdm.interactive.NIMBUS method*), 31

response (*desdeo_mcdm.interactive.ENautilusInitialRequest attribute*), 12

response (*desdeo_mcdm.interactive.ENautilusRequest attribute*), 13

response (*desdeo_mcdm.interactive.NautilusInitialRequest attribute*), 23

response (*desdeo_mcdm.interactive.NautilusNavigatorRequest attribute*), 28

response (*desdeo_mcdm.interactive.NautilusRequest attribute*), 24

response (*desdeo_mcdm.interactive.NimbusClassificationRequest attribute*), 33

response (*desdeo_mcdm.interactive.NimbusIntermediateSolutionsRequest attribute*), 34

response (*desdeo_mcdm.interactive.NimbusMostPreferredRequest attribute*), 35

response (*desdeo_mcdm.interactive.NimbusSaveRequest attribute*), 36

response (*desdeo_mcdm.interactive.RPMInitialRequest attribute*), 40

response (*desdeo_mcdm.interactive.RPMRequest attribute*), 40

RPMException, 39

RPMInitialRequest (*class in desdeo_mcdm.interactive*), 39

RPMRequest (*class in desdeo_mcdm.interactive*), 40

RPMStopRequest (*class in desdeo_mcdm.interactive*), 41

S

save_solutions_to_archive() (*desdeo_mcdm.interactive.NIMBUS method*), 31

solve_asf() (*desdeo_mcdm.interactive.Nautilus method*), 17

solve_asf() (*desdeo_mcdm.interactive.NautilusV2 method*), 22

solve_asf() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 38

solve_nautilus_asf_problem() (*desdeo_mcdm.interactive.NautilusNavigator method*), 26

solve_pareto_front_representation() (*in module desdeo_mcdm.utilities*), 43

solve_pareto_front_representation_general() (*in module desdeo_mcdm.utilities*), 44

start() (*desdeo_mcdm.interactive.ENautilus method*), 12

start() (*desdeo_mcdm.interactive.Nautilus method*), 18

start() (*desdeo_mcdm.interactive.NautilusNavigator method*), 26

start() (*desdeo_mcdm.interactive.NautilusV2 method*), 22

start() (*desdeo_mcdm.interactive.NIMBUS method*), 32

start() (*desdeo_mcdm.interactive.ReferencePointMethod method*), 39

U

update() (*desdeo_mcdm.interactive.NautilusNavigator method*), 26

update_current_solution() (*desdeo_mcdm.interactive.NIMBUS method*), 32

V

validate_n2_preferences() (*in module desdeo_mcdm.interactive*), 9

validate_n_iterations() (*in module desdeo_mcdm.interactive*), 9

validate_preferences() (*in module desdeo_mcdm.interactive*), 8

validate_response() (*in module desdeo_mcdm.interactive*), 8

validator() (*desdeo_mcdm.interactive.ENautilusInitialRequest method*), 13

validator() (*desdeo_mcdm.interactive.ENautilusRequest method*), 13

validator() (*desdeo_mcdm.interactive.NautilusNavigatorRequest method*), 28

validator() (*desdeo_mcdm.interactive.NimbusClassificationRequest method*), 33

validator() (*desdeo_mcdm.interactive.NimbusIntermediateSolutionsRe method*), 34

validator() (*desdeo_mcdm.interactive.NimbusMostPreferredRequest method*), 35

validator() (*desdeo_mcdm.interactive.NimbusSaveRequest method*), 36

W

weighted_scalarizer() (*in module desdeo_mcdm.utilities*), 45