DESDEO Documentation

Vesa Ojalehto, Frankie Robertson

May 20, 2019
## CONTENTS:

1 DESDEO README ........................................ 1
   1.1 Try in your browser ................................ 1
   1.2 What is interactive multiobjective optimization? 1
   1.3 Installation ........................................ 1
       1.3.1 From conda-forge using Conda ............... 1
       1.3.2 From PyPI using pip ............................ 2
   1.4 Getting started with example problems .......... 2
   1.5 Development ....................................... 2
       1.5.1 Set-up ......................................... 2
       1.5.2 Tests .......................................... 2
       1.5.3 Release process .............................. 3

2 Background ............................................. 5
   2.1 What is NIMBUS? .................................... 5
       2.1.1 Mathematical approach ....................... 5
   2.2 Classification in NIMBUS ........................... 5
       2.2.1 Classification background .................... 5
       2.2.2 Classification using the widget ............ 6
           2.2.2.1 Classifying the cylinder problem ...... 7
       2.2.3 Classification without the widget ........... 7
       2.2.4 Specifying subproblems ...................... 7
   2.3 Estimates of the ICV and Nadir .................... 7
   2.4 What is NAUTILUS? ................................ 8
   2.5 Glossary ........................................... 8
   2.6 Architecture ...................................... 9
   2.7 Further Reading .................................. 10

3 Topical guides ......................................... 11
   3.1 How to get your multi-objective optimization problem working with DESDEO .... 11
       3.1.1 Creating the model in Python ............... 11
       3.1.2 Importing a list of solutions as data ..... 12
       3.1.3 Interfacing to another language via Thrift 12

4 API documentation ..................................... 13
   4.1 desdeo.core.ResultFactory ......................... 14
   4.2 desdeo.method ...................................... 14
   4.3 desdeo.method.base ................................ 17
   4.4 desdeo.method.NAUTILUS .......................... 17
   4.5 desdeo.method.NIMBUS ............................. 19
   4.6 desdeo.optimization ............................... 20
DESDEO is a free and open source Python-based framework for developing and experimenting with interactive multiobjective optimization.

Documentation is available.

Background and publications available on the University of Jyväskylä Research Group in Industrial Optimization web pages.

1.1 Try in your browser

You can try a guided example problem in your browser: choose how to deal with river pollution using NIMBUS. You can also browse the other examples.

1.2 What is interactive multiobjective optimization?

There exist many methods to solve multiobjective optimization problems. Methods which introduce some preference information into the solution process are commonly known as multiple criteria decision making methods. When using so called interactive methods, the decision maker (DM) takes an active part in an iterative solution process by expressing preference information at several iterations. According to the given preferences, the solution process is updated at each iteration and one or several new solutions are generated. This iterative process continues until the DM is sufficiently satisfied with one of the solutions found.

Many interactive methods have been proposed and they differ from each other e.g. in the way preferences are expressed and how the preferences are utilized when new solutions. The aim of the DESDEO is to implement aspects common for different interactive methods, as well as provide framework for developing and implementing new methods.

1.3 Installation

1.3.1 From conda-forge using Conda

This is the recommended installation method, especially for those who are newer to Python. First download and install the Anaconda Python distribution.

Next, run the following commands in a terminal:

```
conda config --add channels conda-forge
conda install desdeo desdeo-vis
```
Note: if you prefer not to install the full Anaconda distribution, you can install miniconda instead.

### 1.3.2 From PyPI using pip

Assuming you have Pip and Python 3 installed, you can install desdeo from PyPI by running the following command in a terminal:

```bash
pip install desdeo[vis]
```

This installs desdeo and desdeo-vis, which you will also want in most cases.

### 1.4 Getting started with example problems

To proceed with this section, you must first install Jupyter notebook. If you’re using Anaconda, you already have it! You can copy the example notebooks to the current directory by running:

```bash
python -m desdeo_notebooks
```

You can then open them using Jupyter notebook by running:

```bash
jupyter notebook
```

After trying out the examples, the next step is to read the full documentation.

### 1.5 Development

#### 1.5.1 Set-up

You should install the git pre-commit hook so that code formatting is kept consistent automatically. This is configured using the pre-commit utility. See the installation instructions. In short, pre-commit hook can be installed as

```bash
pip install --upgrade pre-commit
pre-commit install
```

If you are using pipenv for development, you can install desdeo and its dependencies after obtaining a git checkout like so:

```bash
pipenv install -e .[docs,dev,vis]
```

#### 1.5.2 Tests

Tests use pytest. After installing pytest you can run:

```bash
pytest tests
```
1.5.3 Release process

1. Make a release commit in which the version is incremented in setup.py and an entry added to HISTORY.md
2. Make a git tag of this commit with `git tag v$VERSION`
3. Push – including the tags with `git push --tags`
4. Upload to PyPI with `python setup.py sdist bdist_wheel` and `twine upload dist/*`
This section contains background and is meant to serve as a quick guide or reference to the key concepts and practical issues required to make use of the interactive multi-objective optimization techniques in DESDEO.

2.1 What is NIMBUS?

As its name suggests, NIMBUS (Nondifferentiable Interactive Multiobjective BUndle-based optimization System) is a multiobjective optimization system being able to handle even nondifferentiable functions. It will optimize (minimize or maximize) several functions at the same time, 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. Usually, an example is the best way to make things clear:

2.1.1 Mathematical approach

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, what are the limits of the objectives, etc. In NIMBUS, this phase is called a ‘classification’. We will discuss this procedure later.

Searching for the desired solution means actually finding the best compromise between many separate goals. If we want to get lower values for one function, we must be ready to accept the growth of another function. This is due to the fact that 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).

2.2 Classification in NIMBUS

The solution process with NIMBUS is iterative. Since there is usually not only one absolutely right solution, you are asked to ‘guide the solver to a desired direction’. The classification is a process in which the desires of the user are expressed. You can choose which of the function values should be decreased from the current level and which of the functions are less important (i.e. their values can be increased).

2.2.1 Classification background

In NIMBUS, preferences are expressed by choosing a class for each of the objective functions.

When considering minimization, the class alternatives are:
The value should be minimized.

The value should be minimized until the specified limit is reached.

The current value is OK.

Value can be increased. Value should be kept below the specified upper bound.

Value can change freely.

For maximization, directional signs are inverted:

The value should be maximized.

The value should be maximized until the specified limit is reached.

The current value is OK.

Value can be decreased. Value should be kept above the specified lower bound.

Value can change freely.

If the second or the fourth alternative is selected, you are asked to specify the limits: an aspiration level or an upper/lower bound respectively for the function values:

- **Aspiration level** defines a desired value for the objective function.
- **Upper/lower bound** defines the limit value that the function should not exceed, if possible.

Since we are dealing with Pareto optimal solutions (compromises) we must be willing to give up something in order to improve some other objective. That is why the classification is feasible only if at least one objective function is in the first two classes and at least one objective function is in the last two classes.

In other words, you must determine at least one function whose value should be made better. However that can not be done if there are no functions whose value can be worsened.

### 2.2.2 Classification using the widget

You may find it useful to follow along with the cylinder notebook while reading this section.

The current solution is shown graphically as a parallel coordinate plot. The classification can be made by either clicking points on the axes, or by manually adjusting the classification selection boxes and limit fields.

By default, maximization and minimization are displayed in their original units. You may wish to reformulate the problem so everything is minimized. This means all maximized functions are negated. To enable this, click settings and then check Reformulate maximization as minimization. To change the default, add the following to the beginning of your notebooks:

```python
from desdeo_vis.conf import conf
cnf(max_as_min=True)
```

Let us assume that the function under classification should be minimized. When you click on the corresponding axis, the system considers the axis value at the point you clicked on as a new limit value and inserts the numerical data into the limit field automatically. Selecting the point below the current solution means that function should be minimized (<) until that limit is reached. If the point is selected above the current solution, we allow the function value to increase (>) to that limit. Clicking a point below the whole bar means that the function should be minimized as much as possible (<). Correspondingly, if the value is selected above the whole bar, the function value can change freely (<>). If the current value of some function is satisfactory (=), you can express it by clicking the numeric value beside the bar.

In the case of maximization, the logic above is reversed. For example, if you click a point above the current solution, it indicates that the function should be maximized as much as possible. The desired extreme point of each function is indicated by a small black triangle inside the top or the bottom of each bar.
You can refine the graphical classification by changing the class of each objective function using the selection box or adjusting the value in its limit field.

### 2.2.2.1 Classifying the cylinder problem

This section walks you through creating a classification with the widget for the cylinder notebook.

The first solution we get from NIMBUS is reasonable. However, we may decide at this point that we want to increase the cylinder’s volume as much as possible, while still keeping the surface area and height difference low.

To do this, we select $(<>)$ from the volume dropdown, because we allow (for now) the volume to be varied freely. The next column describes the solution for the surface area function. We want to know how much the volume will be when the surface area is 1900, so we select $(>=)$ from the dropdown and enter 1900 into the box. For height difference we select $(<=)$.

### 2.2.3 Classification without the widget

It is also possible to make a classification without the widget. Possibly reasons you might do this are because you are constructing an artificial decision maker, you are making your own preference selection widget, or because you are unable to use Jupyter notebook. In this case, maximizations are always reformulated as minimizations.

The preference information is specified using a Python object called `desdeo.preference.NIMBUSClassification`. If we wanted to make the same classification as above, it can be done like so:

```python
classification = NIMBUSClassification(method, [
    ('>=', 1205.843),
    ('<=', 378.2263),
    ('=', 0.0)
])
```

### 2.2.4 Specifying subproblems

We can specify the maximum number of new solutions generated by the classification given. It’s also possible to specify particular scalarization functions. See `desdeo.method.NIMBUS.next_iteration()` for more information.

### 2.3 Estimates of the ICV and Nadir

The result of the optimization is a vector, where the components are the values of the objective functions. When optimizing the functions individually and creating the vector of these values, we get the ICV; that is, the Ideal Criterion Vector.

The ICV tells us the best solution that exists for each objective, when the functions are treated independently. However, the ICV vector is infeasible because it is usually impossible to get the best of everything at the same time - one must make compromises. For minimized functions ICV represents the lower bounds in the set of Pareto optimal solutions and the values are shown on the x-axis of the bar graph. For maximized functions ICV represents the upper bounds in the Pareto optimal set and the values can be found at the top of the bars. If the problem is complicated (that is, nonconvex) the actual components of ICV are difficult to calculate. Thus, to make sure, we refer to ICV as an estimated ICV.

The nadir is in some sense the opposite of the ICV. It consists of component values for the ‘worst case’ solution vector. For minimized functions Nadir represents the upper bounds in the set of Pareto optimal solutions and the values can
be found at the top of the bars. For maximized functions Nadir represents the lower bounds in the Pareto optimal set and the values are shown on the x-axis of the bar graph. In practise, the Nadir vector is only an estimation because it is also difficult (even impossible, in the general case) to calculate.

The estimated components of the ICV and the Nadir vector are updated during the calculations, whenever the solver founds improved values. If you know the exact values or better estimates of the ICV and Nadir vectors, you can correct the estimates of the system by setting the ideal and nadir properties of your subclass of `desdeo.problem.PythonProblem`.

### 2.4 What is NAUTILUS?

Most interactive methods developed for solving multiobjective optimization problems sequentially generate Pareto optimal solutions and the decision maker must always trade-off to get a new solution. Instead, the family of interactive trade-off-free methods called NAUTILUS starts from the worst possible objective values and improves every objective function step by step according to the preferences of the decision maker.

Recently, the NAUTILUS family has been presented as a general NAUTILUS framework consisting of several modules. To extend the applicability of interactive methods, it is recommended that a reliable software implementation, which can be easily connected to different simulators or modelling tools, is publicly available. In this paper, we bridge the gap between presenting an algorithm and implementing it and propose a general software framework for the NAUTILUS family which facilitates the implementation of all the NAUTILUS methods, and even other interactive methods. This software framework has been designed following an object-oriented architecture and consists of several software blocks designed to cover independently the different requirements of the NAUTILUS framework. To enhance wide applicability, the implementation is available as open source code.

### 2.5 Glossary

- **Pareto optimality** A criterion vector \( z^* \) (consisting of the values of the objective functions at a point \( x^* \)) is *Pareto optimal* if none of its components can be improved without impairing at least one of the other components. In this case, \( x^* \) is also called Pareto optimal. Synonyms for Pareto optimality are efficiency, noninteriority and Edgeworth-Pareto optimality.

- **Weak Pareto optimality** A criterion vector \( z^* \) (consisting of the values of the objective functions at a point \( x^* \)) is *weakly Pareto optimal* if there does not exist any other vector for which all the components are better. In this case, \( x^* \) is also called weakly Pareto optimal.

- **Ideal criterion vector (ICV)** The ideal criterion vector consists of the best possible values each objective function can achieve. The ICV represents the lower bounds of the set of Pareto optimal solutions. (That is, Pareto optimal set) For minimized functions the ICV is given as the Lowest Value, and for maximized functions as the Highest Value.

- **Current solution** Current values of the objective functions.

- **Nadir vector (or nadir point)** Estimated upper bounds of the solutions in the Pareto optimal set. The nadir vector represents the worst values that each objective function can attain in the Pareto optimal set. For minimized functions the Nadir is given as the Highest Value, and for maximized functions as the Lowest Value.

- **(Sub)gradient** A gradient of a function consists of its partial derivatives subject to each variable. A gradient vector exists for differentiable functions. For nondifferentiable functions a more general concept *subgradient* is used.
Aspiration levels  For each minimized function in the class <= and maximized function in the class >= you must specify an aspiration level. The aspiration level is the value which you desire function value should be decreased or increased to.

NOTE: For minimized functions the aspiration level must be between the lowest value and the current value of the objective function. For maximized function the aspiration level must be between the current and highest value of the objective function.

Upper and lower bounds  For each minimized function in the class >= and maximized function in the class <= you must specify a boundary value. The upper or lower bounds are the largest or smallest allowable objective function value respectively.

NOTE: For minimized function the upper bound value must be between the current and highest value of the objective function. For maximized function the lower bound value must be between the current and lowest value of the objective function.

2.6  Architecture

Fig. 1: Overview of the current DESDEO architecture.
2.7 Further Reading

For an in-depth treatment of the whole field of multi-objective optimization, including interactive and non-interactive methods, see:


For more information about the specific techniques implemented in DESDEO, see primarily the publications on the DESDEO project page as well as the other publications of the University of Jyväskylä optimization group.
3.1 How to get your multi-objective optimization problem working with DESDEO

This section describes the different ways you can get your data, model or simulation corresponding to a multi-objective optimization problem working with DESDEO. Currently, there are three approaches to adding a new problem: creating the model in Python, importing a list of solutions as data from a CSV, or interfacing to another language via Thrift.

3.1.1 Creating the model in Python

This approach is based around the classes in desdeo.problem.porcelain. This section walks through the process of defining the cylinder problem: desdeo.problem.toy.CylinderProblem. You can work through the cylinder problem notebook to gain familiarity with the cylinder problem.

```python
from desdeo.problem.porcelain import Objective, PorcelainProblem, Variable

class CylinderProblem(PorcelainProblem):
    """
    A description of the problem goes here.
    """

    r = Variable(low=5, high=15, start=10, name="Radius")
    h = Variable(low=5, high=25, start=10, name="Height")

    @Objective("Volume", maximized=True)
    def volume(r, h):
        return math.pi * r ** 2 * h
```

The first step is to import the classes we need and create a new class which inherits from desdeo.problem.porcelain.PorcelainProblem. You should always add a docstring describing the problem so it can be displayed in the notebook interface.

Next, we can define variables using the desdeo.problem.porcelain.Variable class. Here, two variables \( r \) and \( h \) are defined as the radius and height and given a range of acceptable values, a starting value and a display name.

Now we define one of the objectives: maximizing the volume of the cylinder. First we define a function that takes all variables \( r \) and \( h \) in the form we have just named them. The body of the function is the formula for the volume of a cylinder \( \pi r^2 h \), translated to Python code. The function is then decorated with the desdeo.problem.porcelain.Objective class where it is given a display name and the direction (maximized versus minimized) is chosen.
We now define the other objectives, setting `maximized=False` to define them as minimization problems.

```python
@Objective("Surface Area", maximized=False)
def surface_area(r, h):
    return 2 * math.pi * r ** 2 + 2 * math.pi * r * h

@Objective("Height Difference", maximized=False)
def height_diff(r, h):
    return abs(h - 15.0)
```

Lastly, we define the inner class `Meta` which is where extra settings are stored. Here, we just define the display name for the problem.

```
class Meta:
    name = "Cylinder Problem"
```

### 3.1.2 Importing a list of solutions as data

In this approach, you start by generating a list of solutions and write out their corresponding objective values to a CSV file. DESDEO can then be used to find the row corresponding to the best solution according to some preferences.

The main class involved here is `desdeo.problem.PreGeneratedProblem`. Usage is simple, just pass in a `filename` parameter:

```
PreGeneratedProblem(
    filename="/path/to/my/file.csv"
)
```

`PreGeneratedProblem` only works with the `desdeo.optimization.PointSearch` (single-objective) optimization method. So a full example using ENAUTILUS would look like:

```
ENAUTILUS(
    PreGeneratedProblem(
        filename="/path/to/my/file.csv"
    ),
    PointSearch,
)
```

### 3.1.3 Interfacing to another language via Thrift

In some cases, your multi-objective optimization problem may be defined in terms of simulation software you already have available. In order to transfer information from the simulation to DESDEO, a Thrift interface is provided. (Coming soon)
## API DOCUMENTATION

<table>
<thead>
<tr>
<th>Package/Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>desdeo.core.ResultFactory</code></td>
<td>This package contains methods for interactively solving multi-objective optimisation problems.</td>
</tr>
<tr>
<td><code>desdeo.method</code></td>
<td>This package contains methods for solving single-objective optimisation problems.</td>
</tr>
<tr>
<td><code>desdeo.method.base</code></td>
<td>This module contains the abstract base class for all interactive methods.</td>
</tr>
<tr>
<td><code>desdeo.method.NAUTILUS</code></td>
<td>NAUTILUS method variants.</td>
</tr>
<tr>
<td><code>desdeo.method.NIMBUS(problem, method_class)</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.optimization</code></td>
<td>This package contains methods for solving single-objective optimisation problems.</td>
</tr>
<tr>
<td><code>desdeo.optimization.OptimizationProblem</code></td>
<td>This module contains single objective optimization problems.</td>
</tr>
<tr>
<td><code>desdeo.optimization.OptimizationMethod</code></td>
<td>This module contains methods for solving single-objective optimisation problems.</td>
</tr>
<tr>
<td><code>desdeo.preference</code></td>
<td>This package contains various classes acting as containers for preference information given by a decision maker about which objectives they are concerned with and to what degree.</td>
</tr>
<tr>
<td><code>desdeo.preference.base</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.preference.direct</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.preference.nimbus</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.problem</code></td>
<td>This package contains tools for modelling multi-objective optimisation problems.</td>
</tr>
<tr>
<td><code>desdeo.problem.porcelain</code></td>
<td>This module contains simple “toy” problems suitable for demonstrating different interactive multi-objective optimization methods.</td>
</tr>
<tr>
<td><code>desdeo.problem.toy</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.problem.RangeEstimators</code></td>
<td>Contains tools to estimate the range of objective values in the Patero optimal set.</td>
</tr>
<tr>
<td><code>desdeo.result</code></td>
<td>This package contains classes for representing results obtained from running the methods in <code>desdeo.method</code>.</td>
</tr>
<tr>
<td><code>desdeo.utils</code></td>
<td>DESDEO Utilities</td>
</tr>
<tr>
<td><code>desdeo.utils.exceptions</code></td>
<td></td>
</tr>
<tr>
<td><code>desdeo.utils.warnings</code></td>
<td></td>
</tr>
</tbody>
</table>
4.1 desdeo.core.ResultFactory

class desdeo.core.ResultFactory.BoundsFactory (optimization_method)
    Bases: desdeo.core.ResultFactory.ResultFactory

    __abstractmethods__ = frozenset({})
    __init__ (optimization_method)
        Initialize self. See help(type(self)) for accurate signature.
        __module__ = 'desdeo.core.ResultFactory'

    result (prev_point, upper=False)

class desdeo.core.ResultFactory.IterationPointFactory (optimization_method)
    Bases: desdeo.core.ResultFactory.ResultFactory

    __abstractmethods__ = frozenset({})
    __init__ (optimization_method)
        Initialize self. See help(type(self)) for accurate signature.
        __module__ = 'desdeo.core.ResultFactory'

    result (preferences, prev_point)
        Return type Tuple[numpy.ndarray, List[float]]

class desdeo.core.ResultFactory.ResultFactory
    Bases: abc.ABC
    Abstract base class for result factories

    __abstractmethods__ = frozenset({})
    __module__ = 'desdeo.core.ResultFactory'

4.2 desdeo.method

This package contains methods for interactively solving multi-objective optimisation problems. Currently this includes the NIMBUS methods and several variants of the NAUTILUS method.

class desdeo.method.NAUTILUSv1 (method, method_class)
    Bases: desdeo.method.NAUTILUS.NAUTILUS

    The first NAUTILUS method variant [MIETTINEN2010].

    References

    __abstractmethods__ = frozenset({})
    __init__ (method, method_class)
        Initialize self. See help(type(self)) for accurate signature.
        __module__ = 'desdeo.method.NAUTILUS'

    _update_fh ()
        Return next iteration bounds
print_current_iteration()

class desdeo.method.ENAUTILUS(method, method_class)
    Bases: desdeo.method.NAUTILUS.NAUTILUS
    \_abstractmethods\_ = frozenset({})
    \_init\_(method, method_class)
        Initialize self. See help(type(self)) for accurate signature.
        Return type None
    \_module\_ = 'desdeo.method.NAUTILUS'
    \_update\_zh(term1, term2)
    next\_iteration(*args, preference=None, **kwargs)
        Return solution(s) for the next iteration
    print\_current\_iteration()
    select\_point(point)

class desdeo.method.NNAUTILUS(method, method_class)
    Bases: desdeo.method.NAUTILUS.NAUTILUS
    NAVIGATOR NAUTILUS method
    fh
        Current non-dominated point
        Type list of floats
    zh
        Current iteration point
        Type list of floats
    fh\_up
        Upper boundary for iteration points reachable from iteration point zh
        Type list of floats
    fh\_lo
        Lower boundary for iteration points reachable from iteration point zh
        Type list of floats

exception NegativeIntervalWarning
    Bases: desdeo.utils.warnings.UnexpectedCondition
    \_module\_ = 'desdeo.method.NAUTILUS'
    \_str\_()  
        Return str(self).
    \_abstractmethods\_ = frozenset({})
    \_init\_(method, method_class)
        Initialize self. See help(type(self)) for accurate signature.
        \_module\_ = 'desdeo.method.NAUTILUS'
    \_update\_fh()
    next\_iteration(ref\_point, bounds=None)
        Calculate the next iteration point to be shown to the DM
Parameters

- `ref_point(list of float)` – point given by the DM (Reference) -
- `update_points()`

class desdeo.method.NIMBUS(problem, method_class)
    Bases: desdeo.method.base.InteractiveMethod
    ' Abstract class for optimization methods

    _preference
        Preference, i.e., classification information information for current iteration

        Type  CINIMBUSClassificationdefault:None)

    __abstractmethods__ = frozenset({})

    __init__(problem, method_class)
        Initialize self. See help(type(self)) for accurate signature.

        __module__ = 'desdeo.method.NIMBUS'

    _get_ach()

    _init_iteration(*args, **kwargs)
        Return type ResultSet

    _next_iteration(*args, **kwargs)
        Return type ResultSet

    between(objs1, objs2, n=1)
        Generate n solutions which attempt to trade-off objs1 and objs2.

        Parameters

            - `objs1(List[float])` – First boundary point for desired objective function values
            - `objs2(List[float])` – Second boundary point for desired objective function values
            - `n` – Number of solutions to generate

    next_iteration(*args, **kwargs)
        Generate the next iteration’s solutions using the DM’s preferences and the NIMBUS scalarization functions.

        Parameters

            - `preference(NIMBUSClassification)` – Preference classifications obtained from the DM
            - `scalars(list of strings)` – List containing one or more of the scalarizing functions: NIM, ACH, GUESS, STOM
            - `num_scalars(number)` – The number of scalarizing functions to use (mutually exclusive with scalars)
4.3 desdeo.method.base

This module contains the abstract base class for all interactive methods.

```python
class desdeo.method.base.InteractiveMethod(problem, method_class)
    Bases: abc.ABC

    Abstract base class for interactive multiobjective methods.
    __abstractmethods__ = frozenset({'_init_iteration', '_next_iteration'})

    __init__(problem, method_class)
        Initialize self. See help(type(self)) for accurate signature.
    __module__ = 'desdeo.method.base'

    _init_iteration(*args, **kwargs)
    _next_iteration(*args, **kwargs)

    init_iteration(*args, **kwargs)
        Return the initial solution(s)

    next_iteration(*args, **kwargs)
        Return solution(s) for the next iteration
```

4.4 desdeo.method.NAUTILUS

NAUTILUS method variants. The first NAUTILUS variant was introduced in [MIETTINEN2010].

References

```python
class desdeo.method.NAUTILUS.ENAUTILUS(method, method_class)
    Bases: desdeo.method.NAUTILUS.NAUTILUS

    __abstractmethods__ = frozenset({})

    __init__(method, method_class)
        Initialize self. See help(type(self)) for accurate signature.
        Return type None
    __module__ = 'desdeo.method.NAUTILUS'

    _update_zh(term1, term2)

    next_iteration(*args, preference=None, **kwargs)
        Return solution(s) for the next iteration

    print_current_iteration()

    select_point(point)

class desdeo.method.NAUTILUS.NAUTILUS(method, method_class)
    Bases: desdeo.method.base.InteractiveMethod

    __abstractmethods__ = frozenset({})

    __init__(method, method_class)
        Initialize self. See help(type(self)) for accurate signature.
```

4.3. desdeo.method.base
Return type: None

__module__ = 'desdeo.method.NAUTILUS'

__init__iteration__(*args, **kwargs)

__next__iteration__(*args, **kwargs)

__next__zh(term1, term2)

__update_fh()

__update__zh(term1, term2)

distance(where, target)

class desdeo.method.NAUTILUS.NAUTILUSv1(method, method_class)

Bases: desdeo.method.NAUTILUS.NAUTILUS

The first NAUTILUS method variant [MIETTINEN2010].

References

__abstractmethods__ = frozenset({})

__init__(method, method_class)

    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.method.NAUTILUS'

__update_fh()

next__iteration(preference=None)

    Return next iteration bounds

print_current__iteration()

class desdeo.method.NAUTILUS.NNAUTILUS(method, method_class)

Bases: desdeo.method.NAUTILUS.NAUTILUS

NAVIGATOR NAUTILUS method

fh

    Current non-dominated point

        Type  list of floats

zh

    Current iteration point

        Type  list of floats

fh_up

    Upper boundary for iteration points reachable from iteration point zh

        Type  list of floats

fh_lo

    Lower boundary for iteration points reachable from iteration point zh

        Type  list of floats

definition NegativeIntervalWarning

    Bases: desdeo.utils.warnings.UnexpectedCondition

        __module__ = 'desdeo.method.NAUTILUS'
__str__()
Return str(self).

__abstractmethods__ = frozenset({})

__init__(method, method_class)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.method.NAUTILUS'

___update_fh___

next_iteration(ref_point, bounds=None)
Calculate the next iteration point to be shown to the DM

Parameters
- ref_point (list of float) – point given by the DM

update_points()

4.5 desdeo.method.NIMBUS

class desdeo.method.NIMBUS(problem, method_class)
Bases: desdeo.method.base.InteractiveMethod

Abstract class for optimization methods

_preference
Preference, i.e., classification information information for current iteration

Type CINIMBUSClassificationdefault:None)

___init__(problem, method_class)
Initialize self. See help(type(self)) for accurate signature.

Methods

___init__(problem, method_class) Initialize self.

between(objs1, objs2[, n]) Generate n solutions which attempt to trade-off objs1 and objs2.

init_iteration(*args, **kwargs) Return the initial solution(s)

next_iteration(*args, **kwargs) Generate the next iteration’s solutions using the DM’s preferences and the NIMBUS scalarization functions.

__NIMBUS__SCALARS = ['NIM', 'ACH', 'GUESS', 'STOM']

__abstractmethods__ = frozenset({})

___init__(problem, method_class)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.method.NIMBUS'

_get_ach()
4.6 desdeo.optimization

This package contains methods for solving single-objective optimisation problems. These are contained in OptimizationMethod. It also contains scalarisation functions, used for converting multi-objective problems into single-objective functions. These are contained in OptimizationProblem. Both are used as primitives by the methods defined in desdeo.method.

class desdeo.optimization.PointSearch(optimization_problem)

__abstractmethods__ = frozenset({})
__module__ = 'desdeo.optimization.OptimizationMethod'

_search(**params)

The actual search for the optimal solution

This is an abstract class that must be implemented by the subclasses

Parameters  **params(dict [optional]) – Parameters for single objective optimization method

Return type  Tuple[numpy.ndarray, List[float]]

class desdeo.optimization.SciPyDE(optimization_problem)

Optimal search using scipy.optimize.differential_evolution().

__abstractmethods__ = frozenset({})
__init__(optimization_problem)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.optimization.OptimizationMethod'

_objective(x)
Return objective function value

Parameters x (list of values) – Decision variable vector to be calculated

__search(**params)
The actual search for the optimal solution

This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) – Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

class desdeo.optimization.SciPy(optimization_problem)
Optimal search using scipy.optimize.minimize().

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationMethod'

_const(x, *ncon)

_objective(x)
Return objective function value

Parameters x (list of values) – Decision variable vector to be calculated

__search(max=False, **params)
The actual search for the optimal solution

This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) – Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

4.7 desdeo.optimization.OptimizationProblem

This module contains single objective optimization problems. Principally there are scalarization functions for converting multi-objective problems into single-objective functions.

class desdeo.optimization.OptimizationProblem.AchievementProblemBase(mo_problem, **kwargs)
Bases: desdeo.optimization.OptimizationProblem.ScalarizedProblem

Solves problems of the form:

\[
\begin{align*}
\text{minimize} & \\
\max_{i=1,\ldots,k} \{ \mu_i(\ldots) \} + \rho \sum_{i=1}^k \mu_i(\ldots) \\
\text{subject to} & \\
x \in S
\end{align*}
\]

4.7. desdeo.optimization.OptimizationProblem
This is an abstract base class. Implementors should override \_ach, \_augmentation and \_set\_scaling\_weights.

\_abstractmethods\_ = frozenset({'\_ach', '\_augmentation', '\_set\_scaling\_weights'})

\_init\_ (mo\_problem, **kwargs)

Constructor

Return type None

\_module\_ = 'desdeo.optimization.OptimizationProblem'

\_ach \( (\text{objectives}) \)

Calculate achievement term

Return type List[\text{float}]

\_augmentation \( (\text{objectives}) \)

Calculate augmentation term

Return type List[\text{float}]

\_evaluate \( (\text{objectives}) \)

Return type Tuple[\text{List[float]}, \text{Optional[\text{numpy.ndarray}]()]}

\_set\_preferences()

\_set\_scaling\_weights()

class desdeo.optimization.OptimizationProblem.EpsilonConstraintProblem(mo\_problem, \text{obj\_bounds=None})

Bases: desdeo.optimization.OptimizationProblem.OptimizationProblem

Epsilon constraint problem

\begin{align*}
\text{minimize} \\
\quad f_r(x) \\
\text{subject to} \\
\quad f_j(x) \leq z_j, j = 1, \ldots, k; j \neq r, \\
\quad x \in S,
\end{align*}

bounds

Boundary value for the each of the objectives. The objective with boundary of None is to be minimized

Type List of numerical values

\_abstractmethods\_ = frozenset({})

\_init\_ (mo\_problem, \text{obj\_bounds=None})

Constructor

Return type None

\_module\_ = 'desdeo.optimization.OptimizationProblem'

\_evaluate \( (\text{objectives}) \)

Return type Tuple[\text{List[float]}, \text{Optional[\text{numpy.ndarray}]()]}

class desdeo.optimization.OptimizationProblem.MaxEpsilonConstraintProblem(mo\_problem, \text{obj\_bounds=None})

Epsilon constraint problem where the objective is to be maximized

\[
\begin{align*}
\text{maximize} & \quad f_r(x) \\
\text{subject to} & \quad f_j(x) \leq z_j, j = 1, \ldots, k, j \neq r, \\
& \quad x \in S
\end{align*}
\]

This is a special case of using epsilon constraint, to be very clear when using maximized scalarizing function.

**bounds**
Boundary value for the each of the objectives. The objective with boundary of None is to be minimized

**Type** List of numerical values

```python
__abstractmethods__ = frozenset({})
__init__(mo_problem, obj_bounds=None)
Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'
class desdeo.optimization.OptimizationProblem.NIMBUSAchievementProblem(mo_problem, **kwargs)
Finds new solution by solving NIMBUS version of the achivement problem.

\[
\begin{align*}
\text{minimize} & \quad \max_{i=1, \ldots, k} \left[ \frac{f_i(x) - \bar{z}_i}{z^\text{nad}_i - \bar{z}_i} \right] + \rho \sum_{i=1}^k \frac{f_i(x)}{z^\text{nad}_i - \bar{z}_i} \\
\text{subject to} & \quad x \in S.
\end{align*}
\]

__abstractmethods__ = frozenset({})
__module__ = 'desdeo.optimization.OptimizationProblem'
_get_rel()
class desdeo.optimization.OptimizationProblem.NIMBUSGuessProblem(mo_problem, **kwargs)
Bases: desdeo.optimization.OptimizationProblem.SimpleAchievementProblem
Finds new solution by solving NIMBUS version of the GUESS method.

\[
\begin{align*}
\text{minimize} & \quad \max_{i \in I} \left[ \frac{f_i(x) - \bar{z}_i}{z^\text{nad}_i - \bar{z}_i} \right] + \rho \sum_{i=1}^k \frac{f_i(x)}{z^\text{nad}_i - \bar{z}_i} \\
\text{subject to} & \quad x \in S.
\end{align*}
\]

In this implementation \( z^\text{nad} \) is \( \text{eps} \) larger than the true nadir to protect against the case where \( \bar{z}_i = z^\text{nad}_i \) causing division by zero.
class desdeo.optimization.OptimizationProblem.NIMBUSProblem(mo_problem, **kwargs)
Bases: desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin,
desdeo.optimization.OptimizationProblem.SimpleAchievementProblem
Finds new solution by solving NIMBUS scalarizing function.

minimize

\[
\max_{i \in I^< \cup I^<= \cup I^=} \left[ \frac{f_i(x) - z^*_i}{z^\text{nad}_i - z^*_i} - f_j(x) - \bar{z}_j \right] + \rho \sum_{i=1}^k \frac{f_i(x)}{z^\text{nad}_i - z^{**}_i}
\]

subject to

\[ f_i(x) \leq f_i(x^c) \quad \text{for all} \quad i \in I^< \cup I^-= \cup I^=, \]

\[ f_i(x) \leq \epsilon_i \quad \text{for all} \quad i \in I^>, \]

\[ x \in S \]

class desdeo.optimization.OptimizationProblem.NIMBUSStomProblem(mo_problem, **kwargs)
Bases: desdeo.optimization.OptimizationProblem.SimpleAchievementProblem
Finds new solution by solving NIMBUS version of the satisficing trade-off method (STOM).

minimize

\[
\max_{i=1,...,k} \left[ \frac{f_i(x) - z^{**}_i}{z_i - z^{**}_i} \right] + \rho \sum_{i=1}^k \frac{f_i(x)}{z_i - z^{**}_i}
\]

subject to

\[ x \in S \]
class desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin

Bases: object

This mixin implements _set_scaling_weights as:

\[
\frac{1}{z_i - z_i^{**}}
\]

class desdeo.optimization.OptimizationProblem.NautilusAchievementProblem

Bases: desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin,
      desdeo.optimization.OptimizationProblem.SimpleAchievementProblem

Solves problems of the form:

\[
\min_{x \in S} \left\{ \mu_i(f_i(x) - q_i) + \rho \sum_{i=1}^k \frac{f_i(x) - q_i}{z_i^{nad} - z_i^{**}} \right\}
\]

\[\text{subject to}\]

\[x \in S\]

class desdeo.optimization.OptimizationProblem.OptimizationProblem

Bases: abc.ABC

Single objective optimization problem

problem

The multi-objective problem that the single-objective problem is posed in terms of
__abstractmethods__ = frozenset({"_evaluate")

__init__(mo_problem)
Constructor

Return type None

__module__ = "desdeo.optimization.OptimizationProblem"
__evaluate__(objectives)
Return type Tuple[List[float], Optional[numpy.ndarray]]
evaluate(objectives)
Evaluate value of the objective function and possible additional constraints
Parameters objectives (list of objective values) –
Return type Tuple[List[float], Optional[numpy.ndarray]]
Returns
• objective (list of floats) – Objective function values corresponding to objectives
• constraint (2-D matrix of floats) – Constraint function values corresponding to objectives per row. None if no constraints are added

class desdeo.optimization.OptimizationProblem.ScalarizedProblem(mo_problem, **kwargs)
Bases: desdeo.optimization.OptimizationProblem.OptimizationProblem, abc.ABC
__abstractmethods__ = frozenset({"_evaluate", "_set_preferences")

__init__(mo_problem, **kwargs)
Constructor

Return type None

__module__ = "desdeo.optimization.OptimizationProblem"
__evaluate__(objectives)
Return type Tuple[List[float], Optional[numpy.ndarray]]

_set_preferences()
evaluate(objectives)
Evaluate value of the objective function and possible additional constraints
Parameters objectives (list of objective values) –
Return type Tuple[List[float], Optional[numpy.ndarray]]
Returns
• objective (list of floats) – Objective function values corresponding to objectives
• constraint (2-D matrix of floats) – Constraint function values corresponding to objectives per row. None if no constraints are added

set_preferences(preference, last_solution)
class desdeo.optimization.OptimizationProblem.SelectedOptimizationProblem(mo_problem, n)
Bases: desdeo.optimization.OptimizationProblem.OptimizationProblem

Converts a multi-objective optimization problem to a single-objective one by selecting only a single objective.
__abstractmethods__ = frozenset({})
__init__(mo_problem, n)

Parameters n (int) – The index of the objective to be considered

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

_evaluate(objectives)

Return type Tuple[List[float], Optional[numpy.ndarray]]

class desdeo.optimization.OptimizationProblem.SimpleAchievementProblem(mo_problem, **kwargs)

Bases: desdeo.optimization.OptimizationProblem.AchievementProblemBase

Solves a simple form of achievement scalarizing function

\[
\begin{align*}
\text{minimize} \\
\max_{i=1,...,k} \{ \mu_i(f_i(x) - q_i) \} + \rho \sum_{i=1}^{k} \mu_i(f_i(x)) \\
\text{subject to} \\
x \in S
\end{align*}
\]

If ach_pen=True is passed to the constructor, the full achievement function is used as the penalty, causing us to instead solve[WIERZBICKI1980]_

\[
\begin{align*}
\text{minimize} \\
\max_{i=1,...,k} \{ \mu_i(f_i(x) - q_i) \} + \rho \sum_{i=1}^{k} \mu_i(f_i(x) - q_i) \\
\text{subject to} \\
x \in S
\end{align*}
\]

This is an abstract base class. Implementors should override _get_rel and _set_scaling_weights.

References

This module contains methods for solving single-objective optimization problems.

```python
class desdeo.optimization.OptimizationMethod.OptimalSearch(optimization_problem)

Abstract class for optimal search

__abstractmethods__ = frozenset({'_objective', '_search'})
__module__ = 'desdeo.optimization.OptimizationMethod'

_objective(x)
    Return objective function value

Parameters x (list of values) - Decision variable vector to be calculated
```

```python
class desdeo.optimization.OptimizationMethod.OptimizationMethod(optimization_problem)
    Bases: abc.ABC

Abstract class for optimization methods

__max
    True if the objective function is to be maximized
    Type  bool (default:False)

__ceoff
    Coefficient for the objective function
    Type  float

__abstractmethods__ = frozenset({'_search'})
__init__(optimization_problem)
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.optimization.OptimizationMethod'

_search(**params)
    The actual search for the optimal solution

This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) - Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

search(max=False, **params)
    Search for the optimal solution

This sets up the search for the optimization and calls the _search method

Parameters

• max (bool (default False)) - If true find maximum of the objective function instead of minimum
```
**params (dict [optional]) – Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

class desdeo.optimization.OptimizationMethod.PointSearch(optimization_problem)

__abstractmethods__ = frozenset({})
__module__ = 'desdeo.optimization.OptimizationMethod'

_search(**params)
The actual search for the optimal solution
This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) – Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

class desdeo.optimization.OptimizationMethod.SciPy(optimization_problem)

Optimal search using scipy.optimize.minimize().
__abstractmethods__ = frozenset({})
__module__ = 'desdeo.optimization.OptimizationMethod'

_const(x, *ncon)

_objective(x)
Return objective function value

Parameters x (list of values) – Decision variable vector to be calculated

_search (max=False, **params)
The actual search for the optimal solution
This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) – Parameters for single objective optimization method

Return type Tuple[numpy.ndarray, List[float]]

class desdeo.optimization.OptimizationMethod.SciPyDE (optimization_problem)

Optimal search using scipy.optimize.differential_evolution().
__abstractmethods__ = frozenset({})
__init__(optimization_problem)
Initialize self. See help(type(self)) for accurate signature.
__module__ = 'desdeo.optimization.OptimizationMethod'

_objective(x)
Return objective function value

Parameters x (list of values) – Decision variable vector to be calculated
The actual search for the optimal solution

This is an abstract class that must be implemented by the subclasses

Parameters **params (dict [optional]) – Parameters for single objective optimization

Return type Tuple[numpy.ndarray, List[float]]

4.9 desdeo.preference

This package contains various classes acting as containers for preference information given by a decision maker about which objectives they are concerned with and to what degree. It is sued by the methods defined in desdeo.method.

class desdeo.preference.NIMBUSClassification (method, functions, **kwargs)

Preferences by NIMBUS classification

_classification

NIMBUSClassification information pairing objective n to a classification with value if needed

Type Dict (objn_n, (class,value))

_maxmap

Minimization - maximiation mapping of classification symbols

Type NIMBUSClassification (default:None)

__abstractmethods__ = frozenset({})

__getitem__ (key)

Shortcut to query a classification.

__init__ (method, functions, **kwargs)

Initialize the classification information

Parameters functions (list ((class,value)) – Function classification information

Return type None

__module__ = 'desdeo.preference.nimbus'

__setitem__ (key, value)

Shortcut to manipulate a single classification.

_as_reference_point ()

Return classification information as reference point

Return type numpy.ndarray

_maxmap = {'<': '>', '<=': '>', '=': '=', '>': '<', '>=': '<='}

with_class (cls)

Return functions with the class

class desdeo.preference.ReferencePoint (method, reference_point=None)

Bases: desdeo.preference.base.PreferenceInformation

__abstractmethods__ = frozenset({})

__init__ (method, reference_point=None)

Initialize self. See help(type(self)) for accurate signature.
Return type None
__module__ = 'desdeo.preference.base'

reference_point()
Return reference point corresponding to the given preference information

4.10 desdeo.preference.base

class desdeo.preference.base.Direction(method)
    Bases: desdeo.preference.base.PreferenceInformation

    __abstractmethods__ = frozenset({})
    __module__ = 'desdeo.preference.base'

    check_input(data)
    Return type str

default_input()
    Return type numpy.ndarray

class desdeo.preference.base.PreferenceInformation(method)
    Bases: abc.ABC

    __abstractmethods__ = frozenset({})
    __init__(method)
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'desdeo.preference.base'

    weights()
    Return type numpy.ndarray

    weights()
        Return weight vector corresponding to the given preference information

class desdeo.preference.base.ReferencePoint(method, reference_point=None)
    Bases: desdeo.preference.base.PreferenceInformation

    __abstractmethods__ = frozenset({})
    __init__(method, reference_point=None)
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'desdeo.preference.base'

    reference_point()
        Return reference point corresponding to the given preference information

4.10. desdeo.preference.base 31
4.11 desdeo.preference.direct

```python
class desdeo.preference.direct.DirectSpecification(method, direction, reference_point=None):
    Bases: desdeo.preference.base.Direction, desdeo.preference.base.ReferencePoint
    __abstractmethods__ = frozenset({})
    __init__(method, direction, reference_point=None):
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'desdeo.preference.direct'

    _weights():
        Return type numpy.ndarray

    reference_point():
        Return reference point corresponding to the given preference information
        Return type numpy.ndarray

class desdeo.preference.direct.PairwiseRanking(method, selected_obj, other_ranking):
    Bases: desdeo.preference.base.Direction
    __abstractmethods__ = frozenset({})
    __init__(method, selected_obj, other_ranking):
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'desdeo.preference.direct'

    _weights():
        Return type numpy.ndarray

class desdeo.preference.direct.PercentageSpecification(method, percentages):
    Bases: desdeo.preference.base.Direction
    __abstractmethods__ = frozenset({})
    __init__(method, percentages):
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'desdeo.preference.direct'

    _weights():
        Return type numpy.ndarray

    check_input(input):
        Return type str

    default_input():
        Return type numpy.ndarray

class desdeo.preference.direct.RelativeRanking(method, ranking):
    Bases: desdeo.preference.base.Direction
```

Chapter 4. API documentation
4.12 desdeo.preference.nimbus

class desdeo.preference.nimbus.NIMBUSClassification(method, functions, **kwargs)

Preferences by NIMBUS classification

_classification
  NIMBUSClassification information pairing objective n to a classification with value if needed
  
  Type  Dict (objn_n, (class,value))

_maxmap
  Minimization - maximiation mapping of classification symbols
  
  Type  NIMBUSClassification (default:None)

__abstractmethods__ = frozenset({})

__init__ (method, functions, **kwargs)

Initialize the classification information

Parameters

functions (list ((class,value)) – Function classification information

Return type  None

__module__ = 'desdeo.preference.nimbus'

__getitem__ (key)

Shortcut to query a classification.

__init__ (method, functions, **kwargs)

Initialize the classification information

Parameters

functions (list ((class,value)) – Function classification information

Return type  None

__module__ = 'desdeo.preference.nimbus'

__setitem__ (key, value)

Shortcut to manipulate a single classification.

_as_reference_point ()

Return classification information as reference point

Return type  numpy.ndarray

_maxmap = {'<': '>', '<=': '>=', '==': '==', '>': '<', '>=': '<='}

with_class (cls)

Return functions with the class

4.13 desdeo.problem

This package contains tools for modelling multi-objective optimisation problems.
class desdeo.problem.PythonProblem(nobj, nconst=0, ideal=None, nadir=None, maximized=None, objectives=None, name=None, points=None)

Bases: desdeo.problem.Problem.MOProblem

__abstractmethods__ = frozenset({'evaluate'})
__module__ = 'desdeo.problem.Problem'

class desdeo.problem.PreGeneratedProblem(filename=None, points=None, delim=',', **kwargs)

Bases: desdeo.problem.Problem.MOProblem

A problem where the objective function values have been pregenerated

__abstractmethods__ = frozenset({})
__init__(filename=None, points=None, delim=',', **kwargs)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.problem.Problem'

evaluate(population=None)

Evaluate the objective and constraint functions for population and return tuple (objective, constraint) values

Parameters population (list of variable values) -- Description

class desdeo.problem.Variable(bounds=None, starting_point=None, name="")

Bases: object

bounds

lower and upper boundaries of the variable

Type list of numeric values

name

Name of the variable

Type string

starting_point

Starting point for the variable

Type numeric value

__dict__ = mappingproxy({'__module__': 'desdeo.problem.Problem', '__doc__': '

Attributes

----------

bounds : list of numeric values

name : string

starting_point : numeric value

\n


__init__(bounds=None, starting_point=None, name="")

Constructor

__module__ = 'desdeo.problem.Problem'

__weakref__

list of weak references to the object (if defined)

class desdeo.problem.MOProblem(nobj, nconst=0, ideal=None, nadir=None, maximized=None, objectives=None, name=None, points=None)

Bases: abc.ABC

Abstract base class for multiobjective problem

variables

MOProblem decision variable information

Type list of Variables

Chapter 4. API documentation
ideal
   Ideal, i.e, the worst values of objective functions

nadir
   Nadir, i.e, the best values of objective functions

maximized
   Indicates maximized objectives

__abstractmethods__ = frozenset({"evaluate")

__init__(nobj, nconst=0, ideal=None, nadir=None, maximized=None, objectives=None,
          name=None, points=None)
   Initialize self. See help(type(self)) for accurate signature.

   Return type None

__module__ = 'desdeo.problem.Problem'

add_variables (variables, index=None)

   Parameters
   • variable (list of variables or single variable) – Add variables as
     problem variables
   • index (int) – Location to add variables, if None add to the end

   Return type None

as_minimized (v)

bounds ()

evaluate (population)
   Evaluate the objective and constraint functions for population and return tuple (objective,constraint) values

   Parameters population (list of variable values) – Description

nof_objectives ()
   Return type Optional[int]

nof_variables ()
   Return type int

objective_bounds ()
   Return objective bounds

   Returns
   • lower (list of floats) – Lower boundaries for the objectives
   • Upper (list of floats) – Upper boundaries for the objectives

4.14 desdeo.problem.porcelain

class desdeo.problem.porcelain.Constraint (name)
   Bases: object

   __call__ (inner)
   Call self as a function.

   __dict__ = mappingproxy({"__module__": 'desdeo.problem.porcelain', '__init__': <function

class desdeo.problem.porcelain.Objective(name, maximized=False, ideal=None, nadir=None)

Bases: object

__call__(inner)
Call self as a function.

__dict__ = mappingproxy({'__module__': 'desdeo.problem.porcelain', '__init__': <function Objective.__init__>, '__call__': ... '__dict__' of 'Objective' objects>, '__weakref__': <attribute '__weakref__' of 'Objective' objects>, '__doc__': None})

__init__(name, maximized=False, ideal=None, nadir=None)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.problem.porcelain'
__weakref__
list of weak references to the object (if defined)

class desdeo.problem.porcelain.PorcelainProblem
Bases: desdeo.problem.Problem.PythonProblem

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.problem.porcelain'

_porcelain_constrs = []

_porcelain_name = 'PorcelainProblem'

_porcelain_objs = []

_porcelain_vars = []
evaluate(population)
Evaluate the objective and constraint functions for population and return tuple (objective,constraint) values

Parameters population(list of variable values) – Description

class desdeo.problem.porcelain.PorcelainProblemMeta
Bases: abc.ABCMeta

__module__ = 'desdeo.problem.porcelain'

static __new__(cls, name, bases, attrs)
Create and return a new object. See help(type) for accurate signature.

class desdeo.problem.porcelain.Variable(low, high, start, name)
Bases: object

__dict__ = mappingproxy({'__module__': 'desdeo.problem.porcelain', '__init__': <function Variable.__init__>, '__dict__': <attribute '__dict__' of 'Variable' objects>, '__weakref__': <attribute '__weakref__' of 'Variable' objects>, '__doc__': None})

__init__(low, high, start, name)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'desdeo.problem.porcelain'
__weakref__
list of weak references to the object (if defined)
4.15 desdeo.problem.toy

This module contains simple “toy” problems suitable for demonstrating different interactive multi-objective optimization methods.

class desdeo.problem.toy.RiverPollution
    Bases: desdeo.problem.porcelain.PorcelainProblem

    River pollution problem by Narula and Weistroffer [NARULA1989].

    The problem has four objectives and two variables

    The problem describes a (hypothetical) pollution problem of a river, where a fishery and a city are polluting water. The decision variables represent the proportional amounts of biochemical oxygen demanding material removed from water in two treatment plants located after the fishery and after the city.

    The first and second objective functions describe the quality of water after the fishery and after the city, respectively, while objective functions three and four represent the percent return on investment at the fishery and the addition to the tax rate in the city, respectively.

References

__module__ = 'desdeo.problem.toy.river_pollution'
__porc_constrs = []
__porc_name = 'River pollution problem'
__porc_objs = [('wq_fishery', <desdeo.problem.porcelain.Objective object>), ('wq_city', ,

class desdeo.problem.toy.CylinderProblem
    Bases: desdeo.problem.porcelain.PorcelainProblem

    In this problem consider a cell shaped like a cylinder with a circular cross-section.

    The shape of the cell is here determined by two quantities, its radius \( r \) and its height \( h \). We want to maximize the volume of the cylinder and minimize the surface area. In addition to this, cylinder’s height should be close to 15 units, i.e. we minimize the absolute difference between the height and 15.

    Finally the cylinder’s height must be greater or equal to its width. Thus there are 2 decision variables, 3 objectives and 1 constraint in this problem.

__module__ = 'desdeo.problem.toy.cylinder'
__porc_constrs = []
__porc_name = 'Cylinder Problem'
__porc_objs = [('volume', <desdeo.problem.porcelain.Objective object>), ('surface_area', ,
__porc_vars = [('r', <desdeo.problem.porcelain.Variable object>), ('h', <desdeo.problem.porcelain.Variable object>)]

class desdeo.problem.toy.RiverPollutionRobust
    Bases: desdeo.problem.porcelain.PorcelainProblem

    __module__ = 'desdeo.problem.toy.river_pollution_robust'
    __porc_constrs = []
    __porc_name = 'River pollution robust'
    __porc_objs = [('wq_fishery', <desdeo.problem.porcelain.Objective object>), ('wq_city',]
4.16 desdeo.problem.RangeEstimators

Contains tools to estimate the range of objective values in the Patero optimal set. The vector with the best objective values is called the ideal, and the vector with the worst values is called the nadir.

```
deso.problem.RangeEstimators.default_estimate(opt_meth, mo_prob, dp=2)
```

The recommended nadir/ideal estimator - use a payoff table and then round off the result.

**Return type**
Tuple[List[float], List[float]]

```
deso.problem.RangeEstimators.estimate_payoff_table(opt_meth_cls, mo_prob)
```

Estimates the ideal and nadir by using a payoff table. This should give a good estimate for the ideal, but can be very inaccurate for the nadir. For an explanation of why, see [DEB2010].

**References**

```
deso.problem.RangeEstimators.pad(idnad, pad_nadir=0.05, pad_ideal=0.0)
```

Pad an ideal/nadir estimate. This is mainly useful for padding the nadir estimated by a payoff table for safety purposes.

```
deso.problem.RangeEstimators.round_off(idnad, dp=2)
```

Round off an ideal/nadir estimate e.g. so that it’s looks nicer when plotted. This function is careful to round so only ever move the values away from the contained range.

4.17 desdeo.result

This package contains classes for representing results obtained from running the methods in desdeo.method.

```
class desdeo.result.ResultSet(data=None, meta=None)
```

Class to store sets of results.

```
__dict__ = mappingproxy({'

__module__ = 'desdeo.result.Result', '

__doc__': '

Class to store sets of results.

__init__(data=None, meta=None)

Construct a Result from data, such as a list of (decision, objective) pairs.

__module__ = 'desdeo.result.Result'

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

decision_vars

items

meta
objectives_vars

4.18 desdeo.utils

DESDEO Utilities

desdeo.utils.as_minimized(values, maximized)
Return vector values as minimized

   Return type List[float]
desdeo.utils.reachable_points(points, lower, upper)
desdeo.utils.isin(value, values)
Check that value is in values
desdeo.utils.new_points(factory, solution, weights=None)
Generate approximate set of points
   Generate set of Pareto optimal solutions projecting from the Pareto optimal solution using weights to determine the direction.
   Parameters
      • factory (IterationPointFactory) – IterationPointFactory with suitable optimization problem
      • solution – Current solution from which new solutions are projected
      • weights (Optional[List[List[float]]]) – Direction of the projection, if not given generate with :func:random_weights
   Return type List[Tuple[numpy.ndarray, List[float]]]
desdeo.utils.random_weights(nobj, nweight)
Generate nw random weight vectors for nobj objectives as per Tchebycheff method [SteCho83]
   Parameters
      • nobj (int) – Number of objective functions
      • nweight (int) – Number of weights vectors to be generated
   Returns nobj x nweight matrix of weight vectors
   Return type List[List[float]]

4.19 desdeo.utils.exceptions

exception desdeo.utils.exceptions.DESDEOException
Bases: Exception
   __module__ = 'desdeo.utils.exceptions'
   __weakref__
      list of weak references to the object (if defined)

exception desdeo.utils.exceptions.OptimizationProblemError
Bases: desdeo.utils.exceptions.DESDEOException
   __module__ = 'desdeo.utils.exceptions'
exception desdeo.utils.exceptions.PreferenceUndefinedError
   Bases: desdeo.utils.exceptions.DESDEOException
   __module__ = 'desdeo.utils.exceptions'

exception desdeo.utils.exceptions.ProblemDefinitionError
   Bases: desdeo.utils.exceptions.DESDEOException
   __module__ = 'desdeo.utils.exceptions'

4.20 desdeo.utils.warnings

exception desdeo.utils.warnings.UnexpectedCondition
   Bases: UserWarning
   __module__ = 'desdeo.utils.warnings'
   __weakref__
      list of weak references to the object (if defined)

exception desdeo.utils.warnings.UnmetPrecondition
   Bases: desdeo.utils.warnings.UnexpectedCondition
   __module__ = 'desdeo.utils.warnings'
CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search
BIBLIOGRAPHY


PYTHON MODULE INDEX

d

desdeo.core.ResultFactory, 14
desdeo.method, 14
desdeo.method.base, 17
desdeo.method.NAUTILUS, 17
desdeo.optimization, 20
desdeo.optimization.OptimizationMethod, 28
desdeo.optimization.OptimizationProblem, 21
desdeo.preference, 30
desdeo.preference.base, 31
desdeo.preference.direct, 32
desdeo.preference.nimbus, 33
desdeo.problem, 33
desdeo.problem.porcelain, 35
desdeo.problem.RangeEstimators, 38
desdeo.problem.toy, 37
desdeo.result, 38
desdeo.utils, 39
desdeo.utils.exceptions, 39
desdeo.utils.warnings, 40
Symbols

_NIMBUS__SCALARS  (desdeo.method.NIMBUS at-
tribute), 16, 19
__abstractmethods__
  deo.core.ResultFactory.BoundsFactory attribute), 14
__abstractmethods__
  deo.core.ResultFactory.IterationPointFactory attribute), 14
__abstractmethods__
  deo.core.ResultFactory.ResultFactory attribute), 14
__abstractmethods__
  desdeo.method.ENAUTILUS attribute), 15
__abstractmethods__
  desdeo.method.NAUTILUS.ENAUTILUS attribute), 17
__abstractmethods__
  desdeo.method.NAUTILUS.NAUTILUS attribute), 17
__abstractmethods__
  desdeo.method.NAUTILUS.NAUTILUSv1 attribute), 18
__abstractmethods__
  desdeo.method.NAUTILUS.NNAUTILUS attribute), 19
__abstractmethods__
  desdeo.method.NAUTILUSv1 attribute), 14
__abstractmethods__
  desdeo.method.NIMBUS attribute), 16, 19
__abstractmethods__
  desdeo.method.NAUTILUS attribute), 15
__abstractmethods__
  desdeo.method.base.InteractiveMethod attribute), 17
__abstractmethods__
  desdeo.optimization.OptimizationMethod.OptimalSearch attribute), 28
__abstractmethods__
  desdeo.optimization.OptimizationMethod.PointSearch attribute), 29
__abstractmethods__
__abstractmethods__
__abstractmethods__
  desdeo.optimization.OptimizationProblem.AchievementProblemBase attribute), 22
__abstractmethods__
  desdeo.optimization.OptimizationProblem.EpsilonConstraintProblem attribute), 22
__abstractmethods__
  desdeo.optimization.OptimizationProblem.MaxEpsilonConstraintProblem attribute), 23
__abstractmethods__
  desdeo.optimization.OptimizationProblem.NIMBUSAchievementProblem attribute), 23
__abstractmethods__
  desdeo.optimization.OptimizationProblem.NIMBUSGuessProblem attribute), 23
__abstractmethods__
  desdeo.optimization.OptimizationProblem.NIMBUSProblem attribute), 24
__abstractmethods__
  desdeo.optimization.OptimizationProblem.NIMBUSStomProblem attribute), 25
__abstractmethods__
  desdeo.optimization.OptimizationProblem.NautilusAchievementProblem attribute), 25
__abstractmethods__
  desdeo.optimization.OptimizationProblem.OptimizationProblem attribute), 25
__abstractmethods__
  desdeo.optimization.OptimizationProblem.SelectedOptimizationProblem attribute), 26
__abstractmethods__
  desdeo.optimization.OptimizationProblem.ScalarizedProblem attribute), 26
__abstractmethods__ (desdeo.optimization.PointSearch attribute), 20
__abstractmethods__ (desdeo.optimization.SciPy attribute), 21
__abstractmethods__ (desdeo.optimization.SciPyDE attribute), 20
__abstractmethods__ (desdeo.preference.NIMBUSClassification attribute), 30
__abstractmethods__ (desdeo.preference.ReferencePoint attribute), 30
__abstractmethods__ (desdeo.preference.base.Direction attribute), 31
__abstractmethods__ (desdeo.preference.base.PreferenceInformation attribute), 31
__abstractmethods__ (desdeo.preference.base.ReferencePoint attribute), 31
__abstractmethods__ (desdeo.preference.direct.DirectSpecification attribute), 32
__abstractmethods__ (desdeo.preference.direct.PercentageSpecification attribute), 32
__abstractmethods__ (desdeo.preference.direct.RelativeRanking attribute), 32
__abstractmethods__ (desdeo.preference.nimbus.NIMBUSClassification attribute), 33
__abstractmethods__ (desdeo.problem.MOProblem attribute), 35
__abstractmethods__ (desdeo.problem.PreGeneratedProblem attribute), 34
__abstractmethods__ (desdeo.problem.pythonProblem attribute), 34
__call__() (desdeo.problem.porcelain.Constraint method), 35
__call__() (desdeo.problem.porcelain.Objective method), 36
__dict__ (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin attribute), 25
__dict__ (desdeo.problem.porcelain.Variable attribute), 34
__dict__ (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin method), 25
__dict__ (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin attribute), 25
__dict__ (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin method), 25
__getitem__() (desdeo.preference.NIMBUSClassification method), 30
__getitem__() (desdeo.preference.nimbus.NIMBUSClassification method), 33
__init__() (desdeo.core.ResultFactory.BoundsFactory method), 14
__init__() (desdeo.core.ResultFactory.IterationPointFactory method), 14
__init__() (desdeo.method.ENAUTILUS method), 15
__init__() (desdeo.method.NAUTILUS.ENAUTILUS method), 17
__init__() (desdeo.method.NAUTILUS.NAUTILUS method), 17
__init__() (desdeo.method.NAUTILUS.NAUTILUSv1 method), 18
__init__() (desdeo.method.NAUTILUS.NNAUTILUS method), 19
__init__() (desdeo.method.NAUTILUS.NNAUTILUSv1 method), 19
__init__() (desdeo.method.NIMBUS method), 16, 19
__init__() (desdeo.method.NNAUTILUS method), 15
__init__() (desdeo.method.base.InteractiveMethod method), 17
__init__() (desdeo.optimization.OptimizationProblem.AchievementProblemBase method), 22
__init__() (desdeo.optimization.OptimizationProblem.EpsilonConstraintProblem method), 22
__init__() (desdeo.optimization.OptimizationProblem.MaxEpsilonConstraintProblem method), 23
__init__() (desdeo.optimization.OptimizationProblem.NIMBUSProblem method), 24
__init__() (desdeo.optimization.OptimizationProblem.OptimizationProblem method), 26
__init__() (desdeo.optimization.OptimizationProblem.OptimizationMethod method), 26
__init__() (desdeo.optimization.OptimizationProblem.OptimizationMethod method), 27
__init__() (desdeo.optimization.OptimizationProblem.ScalarizedProblem method), 26
__init__() (desdeo.optimization.OptimizationProblem.SimpleAchievementProblem method), 27
Index
__module__ (desdeo.preference.base.ReferencePoint attribute), 31
__module__ (desdeo.preference.direct.DirectSpecification attribute), 32
__module__ (desdeo.preference.direct.PairwiseRanking attribute), 32
__module__ (desdeo.preference.direct.PercentageSpecification attribute), 32
__module__ (desdeo.preference.direct.RelativeRanking attribute), 33
__module__ (desdeo.preference.nimbus.NIMBUSClassification attribute), 33
__module__ (desdeo.problem.MOPProblem attribute), 35
__module__ (desdeo.problem.PreGeneratedProblem attribute), 34
__module__ (desdeo.problem.PorcelainProblemMeta attribute), 36
__module__ (desdeo.problem.Variable attribute), 34
__module__ (desdeo.problem.porcelain.Constraint at-tribute), 33
__module__ (desdeo.problem.porcelain.Objective attribute), 36
__module__ (desdeo.problem.porcelain.Variable at-ttribute), 33
__module__ (desdeo.problem.toy.RiverPollutionRobust attribute), 37
__module__ (desdeo.problem.toy.RiverPollutionRobust attribute), 37
__module__ (desdeo.result.ResultSet attribute), 38
__module__ (desdeo.result.ResultSet attribute), 38
__module__ (desdeo.utils.exceptions.DESDEOException attribute), 40
__module__ (desdeo.utils.warnings.UnexpectedCondition attribute), 40
__module__ (desdeo.utils.warnings.UnmetPrecondition attribute), 40
__module__ (desdeo.utils.exceptions.PreferenceUndefinedError attribute), 40
__module__ (desdeo.utils.exceptions.ProblemDefinitionError attribute), 40
__module__ (desdeo.utils.warnings.UnexpectedCondition attribute), 40
__module__ (desdeo.utils.warnings.UnmetPrecondition attribute), 40
__new__() (desdeo.problem.porcelain.PorcelainProblemMeta static method), 36
__repr__() (desdeo.result.ResultSet attribute), 38
__setitem__() (desdeo.preference.nimbus.NIMBUSClassification method), 30
__setitem__() (desdeo.preference.nimbus.NIMBUSClassification method), 30
__setitem__() (desdeo.preference.nimbus.NIMBUSClassification method), 30
__str__() (desdeo.method.NAUTILUS.NNAUTILUS.NegativeIntervalWarning method), 18
__str__() (desdeo.method.NAUTILUS.NNAUTILUS.NegativeIntervalWarning method), 18
__weakref__(desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin attribute), 25
__weakref__(desdeo.problem.Variable attribute), 34
__weakref__(desdeo.problem.porcelain.Constraint attribute), 36
__weakref__(desdeo.problem.porcelain.Objective attribute), 36
__weakref__(desdeo.problem.porcelain.Variable attribute), 36
__weakref__(desdeo.result.ResultSet attribute), 38
__weakref__(desdeo.result.ResultSet attribute), 38
__weakref__(desdeo.utils.exceptions.DESDEOException attribute), 39
ach() (desdeo.optimization.OptimizationProblem.AchievementProblemBase method), 22
ach() (desdeo.optimization.OptimizationProblem.SimpleAchievementProblem method), 27
ach() (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin method), 22
ach() (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin method), 22
ach() (desdeo.optimization.OptimizationProblem.AchievementProblemBase method), 27
ach() (desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin method), 22
ach() (desdeo.optimization.OptimizationProblem.AchievementProblemBase method), 27
ach() (desdeo.optimization.OptimizationProblem.SimpleAchievementProblem method), 27
coeff() (desdeo.optimization.OptimizationMethod.OptimizationMethod attribute), 28
classification (desdeo.preference.nimbus.NIMBUSClassification attribute), 30
classification (desdeo.preference.nimbus.NIMBUSClassification attribute), 30
classification (desdeo.preference.nimbus.NIMBUSClassification attribute), 30
classification (desdeo.preference.nimbus.NIMBUSClassification attribute), 30
_classification (desdeo.preference.nimbus.NIMBUSClassification attribute), 30
_new_() (desdeo.problem.porcelain.PorcelainProblemMeta static method), 36
_const() (desdeo.optimization.SciPy method), 21
_evaluate() (desdeo.optimization.OptimizationProblem.AchievementProblemBase method), 22
___setitem__() (desdeo.preference.NIMBUSClassification method), 30

Index
Index

A
AchievementProblemBase (class in desdeo.optimization.OptimizationProblem), 21

B
between() (desdeo.method.NIMBUS method), 16, 20

C
check_input() (desdeo.preference.base.Direction method), 31
check_input() (desdeo.preference.direct.PercentageSpecification method), 32

D
decision_vars (desdeo.result.ResultSet attribute), 38
default_estimate() (in module desdeo.problem.RangeEstimators), 38
default_input() (desdeo.preference.base.Direction method), 31
default_input() (desdeo.preference.direct.PercentageSpecification method), 32
desdeo.core.ResultFactory (module), 14
desdeo.method (module), 14
desdeo.method.base (module), 17
desdeo.method.NAUTILUS (module), 17
desdeo.optimization (module), 20
desdeo.optimization.OptimizationMethod (module), 28
desdeo.optimization.OptimizationProblem (module), 21
desdeo.preference (module), 30
desdeo.preference.base (module), 31
desdeo.preference.direct (module), 32
desdeo.preference.nimbus (module), 33
desdeo.problem (module), 33
desdeo.problem.porcelain (module), 35
desdeo.problem.RangeEstimators (module), 38
desdeo.problem.toy (module), 37

E
as_minimized() (desdeo.problem.MOProblem method), 35

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z
desdeo.result (module), 38
desdeo.utils (module), 39
desdeo.utils.exceptions (module), 39
desdeo.utils.warnings (module), 40
DESDEOException, 39
Direction (class in desdeo.preference.base), 31
DirectSpecification (class in desdeo.preference.direct), 32
distance() (desdeo.method.NAUTILUS.NAUTILUS method), 18

ENAUTILUS (class in desdeo.method), 15
ENAUTILUS (class in desdeo.method.NAUTILUS), 17
EpsilonConstraintProblem (class in desdeo.optimization.OptimizationProblem), 22
estimate_payoff_table() (in module desdeo.problem.RangeEstimators), 38
evaluate() (desdeo.optimization.OptimizationProblem method), 26
evaluate() (desdeo.optimization.OptimizationProblem method), 35
evaluate() (desdeo.problem.MOProblem method), 34

fh (desdeo.method.NAUTILUS.NNAUTILUS attribute), 18
fh (desdeo.method.NNAUTILUS attribute), 15
fh_lo (desdeo.method.NAUTILUS.NNAUTILUS attribute), 18
fh_lo (desdeo.method.NNAUTILUS attribute), 15
fh_up (desdeo.method.NAUTILUS.NNAUTILUS attribute), 18
fh_up (desdeo.method.NNAUTILUS attribute), 15

ideal (desdeo.problem.MOProblem attribute), 34
init_iteration() (desdeo.method.base.InteractiveMethod method), 17
InteractiveMethod (class in desdeo.method.base), 17
isin() (in module desdeo.utils), 39
items (desdeo.result.ResultSet attribute), 38
IterationPointFactory (class in desdeo.core.ResultFactory), 14

MaxEpsilonConstraintProblem (class in desdeo.optimization.OptimizationProblem), 22
maximized (desdeo.problem.MOProblem attribute), 35
meta (desdeo.result.ResultSet attribute), 38

MOPProblem (class in desdeo.problem), 34

nadir (desdeo.problem.MOProblem attribute), 35
NadirStarStarScaleMixin (class in desdeo.optimization.OptimizationProblem), 25
name (desdeo.problem.Variable attribute), 34

NIMBUS (class in desdeo.method), 16, 19
NIMBUS (class in desdeo.method.NAUTILUS), 17
NimBUSClassification (class in desdeo.preference), 30
NIMBUSClassification (class in desdeo.preference.nimbus), 33
NIMBUSGuessProblem (class in desdeo.optimization.OptimizationProblem), 23
NIMBUSProblem (class in desdeo.optimization.OptimizationProblem), 24
NIMBUSStomProblem (class in desdeo.optimization.OptimizationProblem), 25
<table>
<thead>
<tr>
<th><strong>N</strong></th>
<th><strong>I</strong></th>
<th><strong>A</strong></th>
<th><strong>T</strong></th>
<th><strong>U</strong></th>
<th><strong>L</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>NNAUTILUS (class in desdeo.method), 15</td>
<td>NNAUTILUS (class in desdeo.method.NAUTILUS), 18</td>
<td>NNAUTILUS.NegativeIntervalWarning, 15, 18</td>
<td>nof_objectives() (desdeo.problem.MOProblem method), 35</td>
<td>nof_variables() (desdeo.problem.MOProblem method), 35</td>
<td></td>
</tr>
<tr>
<td>Objective (class in desdeo.problem.porcelain), 36</td>
<td>objective_bounds() (desdeo.problem.MOProblem method), 35</td>
<td>Objective (class in desdeo.problem.porcelain), 36</td>
<td>objective_bounds() (desdeo.problem.MOProblem method), 35</td>
<td>OptimizationProblem (class in desdeo.optimization.OptimizationProblem), 25</td>
<td></td>
</tr>
<tr>
<td>OptimizationProblem (class in desdeo.optimization.OptimizationProblem), 25</td>
<td>OptimizationProblemError, 39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PorcelainProblem (class in desdeo.problem.porcelain), 36</td>
<td>PorcelainProblemMeta (class in desdeo.problem.porcelain), 36</td>
<td>PreferenceInformation (class in desdeo.preference.base), 31</td>
<td>PreferenceUndefinedError, 39</td>
<td>PreGeneratedProblem (class in desdeo.problem), 34</td>
<td></td>
</tr>
<tr>
<td>search() (desdeo.optimization.OptimizationMethod, 28)</td>
<td>selected_point() (desdeo.method.NAUTILUS method), 15</td>
<td>selected_point() (desdeo.method.NAUTILUS.NAUTILUSv1 method), 17</td>
<td>selected_point() (desdeo.method.NAUTILUS.NAUTILUSv1 method), 18</td>
<td>selected_point() (desdeo.method.NAUTILUS.NAUTILUSv1 method), 18</td>
<td></td>
</tr>
<tr>
<td>problem (desdeo.optimization.OptimizationProblem.OptimizationProblem attribute), 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResultFactory (class in desdeo.core.ResultFactory), 14</td>
<td>ResultSet (class in desdeo.result), 38</td>
<td>RiverPollution (class in desdeo.problem.toy), 37</td>
<td>RiverPollutionRobust (class in desdeo.problem.toy), 37</td>
<td>round_off() (in module desdeo.problem.RangeEstimators), 38</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search() (desdeo.optimization.OptimizationMethod.OptimizationMethod method), 28</td>
<td>selected_point() (desdeo.method.NAUTILUS method), 15</td>
<td>selected_point() (desdeo.method.NAUTILUS.NAUTILUS method), 17</td>
<td>selected_point() (desdeo.method.NAUTILUS.NAUTILUSv1 method), 18</td>
<td>selected_point() (desdeo.method.NAUTILUSv1 method), 18</td>
<td></td>
</tr>
<tr>
<td>SelectedOptimizationProblem (class in desdeo.optimization.OptimizationProblem), 26</td>
<td>set_preferences() (desdeo.optimization.OptimizationProblemScalarizedProblem attribute), 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
method), 26
SimpleAchievementProblem (class in desdeo.optimization.OptimizationProblem), 27
starting_point (desdeo.problem.Variable attribute), 34

U
UnexpectedCondition, 40
UnmetPrecondition, 40
update_points() (desdeo.method.NAUTILUS.NNAUTILUS method), 19
update_points() (desdeo.method.NNAUTILUS method), 16

V
v_ach() (in module desdeo.optimization.OptimizationProblem), 28
v_pen() (in module desdeo.optimization.OptimizationProblem), 28
Variable (class in desdeo.problem), 34
Variable (class in desdeo.problem.porcelain), 36
variables (desdeo.problem.MOProblem attribute), 34

W
weights() (desdeo.preference.base.PreferenceInformation method), 31
with_class() (desdeo.preference.nimbus.NIMBUSClassification method), 33
with_class() (desdeo.preference.NIMBUSClassification method), 30

Z
zh (desdeo.method.NAUTILUS.NNAUTILUS attribute), 18
zh (desdeo.method.NNAUTILUS attribute), 15