DESDEO Documentation

Vesa Ojalehto, Frankie Robertson

May 21, 2019

CONTENTS:

CHAPTER

ONE

DESDEO README

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:

```
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:

```
python -m desdeo_notebooks
```

You can then open them using Jupyter notebook by running:

```
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.

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

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

1.5.2 Tests

Tests use pytest. After installing pytest you can run:

```
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/*`

CHAPTER

TWO

BACKGROUND

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:

```
from desdeo_vis.conf import conf
conf(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:

```
 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

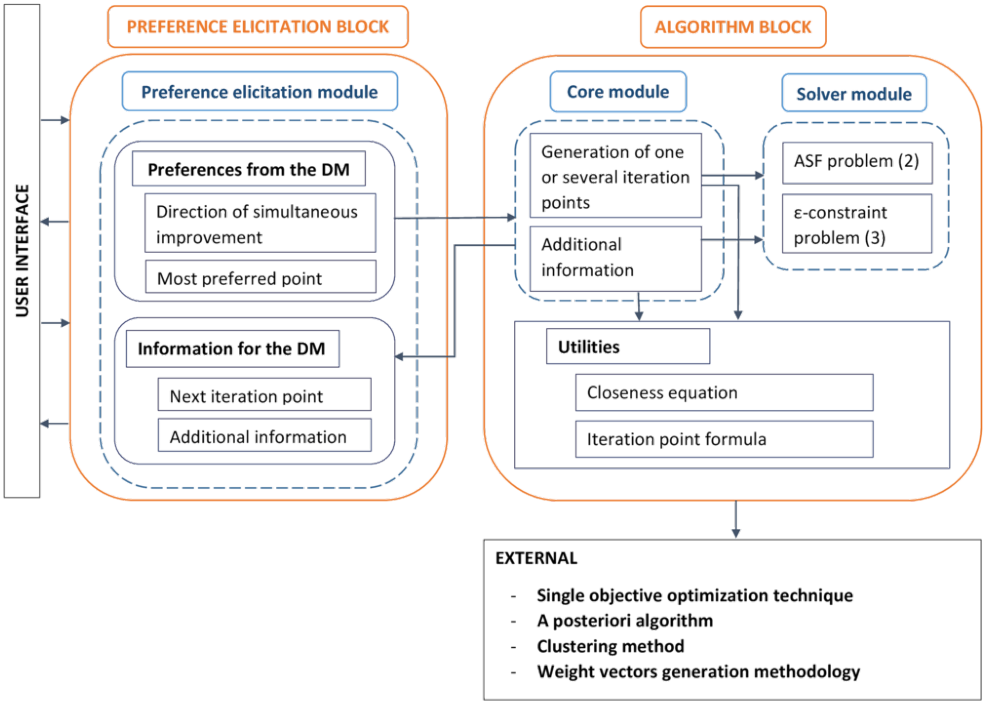

Fig. 1: Overview of the current DESDEO architecture.

2.7 Further Reading

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

Miettinen, K., *Nonlinear multiobjective optimization*, Springer Science & Business Media, 2012.
URL: https://www.springer.com/gp/book/9780792382782

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.

CHAPTER

THREE

API DOCUMENTATION

desdeo.core	
desdeo.method	This package contains methods for interactively solving multi-objective optimisation problems.
desdeo.optimization	This package contains methods for solving single-objective optimisation problems.
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.
desdeo.problem	This package contains tools for modelling multi-objective optimisation problems.
desdeo.problem.toy	This module contains simple "toy" problems suitable for demonstrating different interactive multi-objective optimization methods.
desdeo.result	This package contains classes for representing results obtained from running the methods in *desdeo.method*.
desdeo.utils	DESDEO Utilities

3.1 desdeo.core

3.2 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[1]_

References

[1] Miettinen, K.; Eskelinen, P.; Ruiz, F. & Luque, M., NAUTILUS method: An interactive technique in multiobjective optimization based on the nadir point, European Journal of Operational Research, 2010 , 206 , 426-434.

__init__(*method*, *method_class*)

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

__module__ = 'desdeo.method.NAUTILUS'

next_iteration(*preference=None*)
Return next iteration bounds

print_current_iteration()

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

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

Return type None

__module__ = 'desdeo.method.NAUTILUS'

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).

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

__module__ = 'desdeo.method.NAUTILUS'

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 ClNIMBUSClassificationdefault:None)

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

__module__ = 'desdeo.method.NIMBUS'

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*)

3.3 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 fucntions. These are contained in *OptimizationProblem*. Both are used as primitives by the methods defined in *desdeo.method*.

class desdeo.optimization.**PointSearch**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimizationMethod*

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationMethod'

class desdeo.optimization.**SciPyDE**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimalSearch*

__abstractmethods__ = frozenset({})

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

__module__ = 'desdeo.optimization.OptimizationMethod'

class desdeo.optimization.**SciPy**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimalSearch*

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationMethod'

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

3.3.1 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:

$$\begin{aligned} &\text{minimize} \\ &\max_{i=1,\ldots,k} \{\, \mu_i(\ldots)\} + \rho \sum_{i=1}^{k} \mu_i(\ldots) \\ &\text{subject to} \\ &\mathbf{x} \in S \end{aligned}$$

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'

class desdeo.optimization.OptimizationProblem.**EpsilonConstraintProblem**(*mo_problem*, *obj_bounds=None*)
Bases: *desdeo.optimization.OptimizationProblem.OptimizationProblem*

Epsilon constraint problem

$$\begin{aligned} &\text{minimize} \\ &f_r(\mathbf{x}) \\ &\text{subject to} \\ &f_j(\mathbf{x}) \leq z_j, j = 1, \ldots, k, j \neq r, \\ &\mathbf{x} \in S, \end{aligned}$$

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*, *obj_bounds=None*)
Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

class desdeo.optimization.OptimizationProblem.**MaxEpsilonConstraintProblem**(*mo_problem*, *obj_bounds=None*)
Bases: *desdeo.optimization.OptimizationProblem.EpsilonConstraintProblem*

Epsilon constraint problem where the objective is to be maximized

$$\begin{aligned} &\text{maximize} && f_r(\mathbf{x}) \\ &\text{subject to} && \\ &f_j(\mathbf{x}) \leq z_j, j = 1, \ldots, k, j \neq r, && \\ &\mathbf{x} \in S && \end{aligned}$$

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

__abstractmethods__ = frozenset({})

__init__(*mo_problem*, *obj_bounds=None*)
Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

class desdeo.optimization.OptimizationProblem.**NIMBUSAchievementProblem**(*mo_problem*, ***kwargs*)
Bases: *desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin*, *desdeo.optimization.OptimizationProblem.SimpleAchievementProblem*

Finds new solution by solving NIMBUS version of the achivement problem.

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

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationProblem'

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

Bases: *desdeo.optimization.OptimizationProblem.SimpleAchievementProblem*

Finds new solution by solving NIMBUS version of the GUESS method.

$$\text{minimize} \quad \max_{i \notin I^\diamond} \left[\frac{f_i(\mathbf{x}) - z_i^{\text{nad}}}{z_i^{\text{nad}} - \bar{z}_i} \right] + \rho \sum_{i=1}^{k} \frac{f_i(\mathbf{x})}{z_i^{\text{nad}} - \bar{z}_i}$$

$$\text{subject to} \quad \mathbf{x} \in S.$$

In this implementation z^{nad} is *eps* larger than the true nadir to protect against the case where $\bar{z}_i = z_i^{\text{nad}}$ causing division by zero.

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationProblem'

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.

$$\text{minimize} \quad \max_{\substack{i \in I^< \\ j \in I^\leq}} \left[\frac{f_i(\mathbf{x}) - z_i^\star}{z_i^{\text{nad}} - z_i^{\star\star}}, \frac{f_j(\mathbf{x}) - \hat{z}_j}{z_j^{\text{nad}} - z_j^{\star\star}} \right] + \rho \sum_{i=1}^{k} \frac{f_i(\mathbf{x})}{z_i^{\text{nad}} - z_i^{\star\star}}$$

$$\text{subject to} \quad f_i(\mathbf{x}) \leq f_i(\mathbf{x}^c) \text{ for all } i \in I^< \cup I^\leq \cup I^=,$$

$$f_i(\mathbf{x}) \leq \varepsilon_i \text{ for all } i \in I^\geq,$$

$$\mathbf{x} \in S$$

__abstractmethods__ = frozenset({})

__init__(*mo_problem*, ***kwargs*)

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

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

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).

$$\text{minimize} \quad \max_{i=1,\ldots,k} \left[\frac{f_i(\mathbf{x}) - z_i^{\star\star}}{\bar{z}_i - z_i^{\star\star}} \right] + \rho \sum_{i=1}^{k} \frac{f_i(\mathbf{x})}{\bar{z}_i - z_i^{\star\star}}$$

$$\text{subject to} \quad \mathbf{x} \in S$$

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationProblem'

class desdeo.optimization.OptimizationProblem.**NadirStarStarScaleMixin**
Bases: object

This mixin implements *_set_scaling_weights* as:

$$\frac{1}{z_i^{\text{nad}} - z_i^{\star\star}}$$

__dict__ = mappingproxy({'__module__': 'desdeo.optimization.OptimizationProblem', '__

__module__ = 'desdeo.optimization.OptimizationProblem'

__weakref__
list of weak references to the object (if defined)

class desdeo.optimization.OptimizationProblem.**NautilusAchievementProblem**(*mo_problem*, ***kwargs*)
Bases: *desdeo.optimization.OptimizationProblem.NadirStarStarScaleMixin*, *desdeo.optimization.OptimizationProblem.SimpleAchievementProblem*

Solves problems of the form:

$$\text{minimize} \quad \max_{i=1,\ldots,k} \{ \mu_i (f_i(\mathbf{x}) - q_i) \} + \rho \sum_{i=1}^{k} \frac{f_i(\mathbf{x}) - q_i}{z_i^{\text{nad}} - z_i^{\star\star}}$$

$$\text{subject to} \quad \mathbf{x} \in S$$

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationProblem'

class desdeo.optimization.OptimizationProblem.**OptimizationProblem**(*mo_problem*)
Bases: object

Single objective optimization problem

problem
The multi-objective problem that the single-objective problem is posed in terms of

__abstractmethods__ = frozenset({'_evaluate'})

__dict__ = mappingproxy({'__module__': 'desdeo.optimization.OptimizationProblem', '__

__init__(*mo_problem*)
Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

__weakref__
list of weak references to the object (if defined)

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*

__abstractmethods__ = frozenset({'_evaluate', '_set_preferences'})

__init__(*mo_problem*, ***kwargs*)

Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

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'

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

Bases: *desdeo.optimization.OptimizationProblem.AchievementProblemBase*

Solves a simple form of achievement scalarizing function

$$\text{minimize} \quad \max_{i=1,\ldots,k} \{ \mu_i(f_i(\mathbf{x}) - q_i) \} + \rho \sum_{i=1}^{k} \mu_i(f_i(\mathbf{x}))$$

$$\text{subject to} \quad \mathbf{x} \in S$$

If ach_pen=True is passed to the constructor, the full achivement function is used as the penatly, causing us to instead solve[1]_

$$\text{minimize} \quad \max_{i=1,\ldots,k} \{ \mu_i(f_i(\mathbf{x}) - q_i) \} + \rho \sum_{i=1}^{k} \mu_i(f_i(\mathbf{x}) - q_i)$$

$$\text{subject to} \quad \mathbf{x} \in S$$

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

References

[1] A. P. Wierzbicki, The use of reference objectives in multiobjective optimization, in: G. Fandel, T. Gal (Eds.), Multiple Criteria Decision Making, Theory and Applications, Vol. 177 of Lecture Notes in Economics and Mathematical Systems, Springer, 1980, pp. 468-486.

__abstractmethods__ = frozenset({'_get_rel', '_set_scaling_weights'})

__init__(*mo_problem*, ***kwargs*)
Constructor

Return type None

__module__ = 'desdeo.optimization.OptimizationProblem'

desdeo.optimization.OptimizationProblem.**v_ach**(*f*, *w*, *r*)

desdeo.optimization.OptimizationProblem.**v_pen**(*f*, *w*, *r*)

3.3.2 desdeo.optimization.OptimizationMethod

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

class desdeo.optimization.OptimizationMethod.**OptimalSearch**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimizationMethod*

Abstract class for optimal search

__abstractmethods__ = frozenset({'_objective', '_search'})

__module__ = 'desdeo.optimization.OptimizationMethod'

class desdeo.optimization.OptimizationMethod.**OptimizationMethod**(*optimization_problem*)
Bases: object

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'})

__dict__ = mappingproxy({'__module__': 'desdeo.optimization.OptimizationMethod', '__d

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

__module__ = 'desdeo.optimization.OptimizationMethod'

__weakref__
list of weak references to the object (if defined)

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 mximum 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*)
Bases: *desdeo.optimization.OptimizationMethod.OptimizationMethod*

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationMethod'

class desdeo.optimization.OptimizationMethod.**SciPy**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimalSearch*

__abstractmethods__ = frozenset({})

__module__ = 'desdeo.optimization.OptimizationMethod'

class desdeo.optimization.OptimizationMethod.**SciPyDE**(*optimization_problem*)
Bases: *desdeo.optimization.OptimizationMethod.OptimalSearch*

__abstractmethods__ = frozenset({})

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

__module__ = 'desdeo.optimization.OptimizationMethod'

3.4 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*)

Bases: desdeo.preference.PreferenceInformation.ReferencePoint

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)

__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.PreferenceInformation'

__setitem__(*key*, *value*)

Shortcut to manipulate a single classification.

with_class(*cls*)

Return functions with the class

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

Bases: desdeo.preference.PreferenceInformation.PreferenceInformation

__init__(*method*, *reference_point=None*)

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

Return type None

__module__ = 'desdeo.preference.PreferenceInformation'

reference_point()

Return reference point corresponding to the given preference information

3.5 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

__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 beeen pregenerated

__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__': '\n Attr

__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: object

Abstract base class for multiobjective problem

variables
MOProblem decision variable information

Type list of Variables

ideal
Ideal, i.e, the worst values of objective functions

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

maximized
Indicates maximized objectives

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

__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

__metaclass__
alias of abc.ABCMeta

__module__ = 'desdeo.problem.Problem'

__weakref__
list of weak references to the object (if defined)

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

3.6 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 [1]

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

[1] Narula, S. & Weistroffer, H. A flexible method for nonlinear multicriteria decision-making problems Systems, IEEE Transactions on Man and Cybernetics, 1989, 19, 883-887.

__module__ = 'desdeo.problem.toy.river_pollution'

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'

3.7 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*)

Bases: object

Class to store sets of results.

__dict__ = mappingproxy({'__module__': 'desdeo.result.Result', '__doc__': '\n Class

__init__(*data=None*, *meta=None*)

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

Return type None

__module__ = 'desdeo.result.Result'

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

decision_vars

items

meta

objective_vars

3.8 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*)
Generatate nw random weight vectors for nof 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]

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[SteCho83] Steuer, R. E. & Choo, E.-U. An interactive weighted Tchebycheff procedure for multiple objective programming, Mathematical programming, Springer, 1983, 26, 326-344

PYTHON MODULE INDEX

d

INDEX

Symbols