desdeo_tools

Release 1.0

Multiobjective Optimization Group

Nov 03, 2021

CONTENTS

This package contains generic tools and design language used in the DESDEO framework. For example, it includes classes for interacting with optimization methods implemented in *desdeo-mcdm* and *desdeo-emo*, and tools for solving a representation of a Pareto optimal front for a multiobjective optimization problem.

CHAPTER

ONE

REQUIREMENTS

- Python 3.7 or newer.
- Poetry dependency manager : Only for developers.

See *pyproject.toml* for Python package requirements.

CHAPTER

TWO

INSTALLATION

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

2.1 For users

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

```
$ pip install desdeo_tools
```

2.2 For developers

Download the code or clone it with the following command:

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

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

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

2.2.1 Examples

Example on using the scalarization methods for scalarizing and minimizing a problem which is based on discrete data

In this example, we will go through the following two topics: 1. How to define a scalarization method for scalarizing discrete data representing a multiobjective optimization problem; 2. How to find a solution to the scalarized problem.

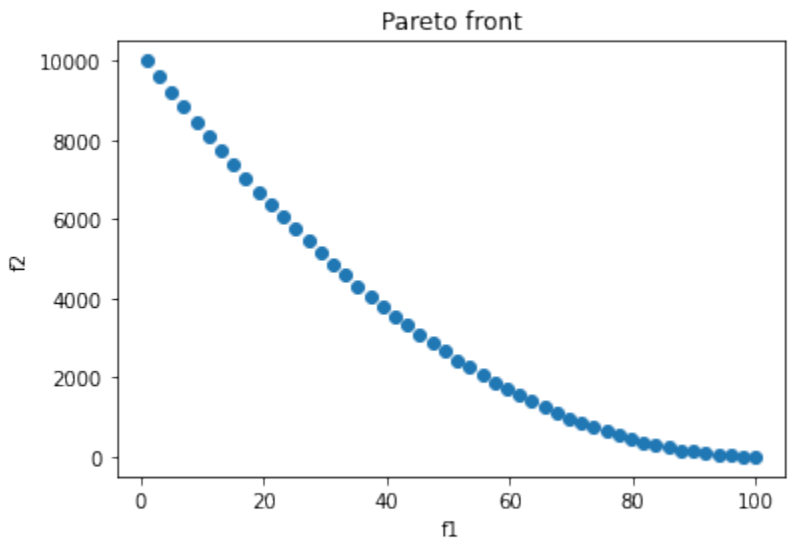
We will start by defining simple 2-dimensional data representing a set of Pareto optimal solutions.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

f1 = np.linspace(1, 100, 50)
f2 = f1[::-1]**2

plt.scatter(f1, f2)
```

(continues on next page)

(continued from previous page)

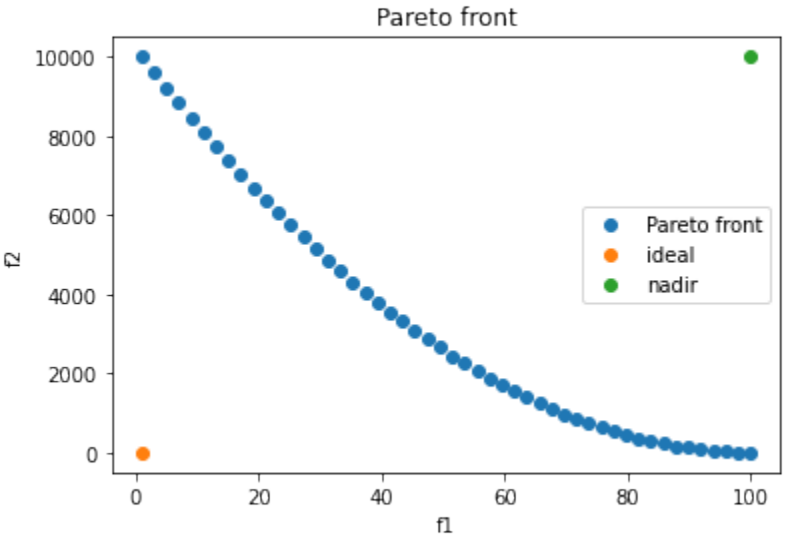
```
plt.title("Pareto front")
plt.xlabel("f1")
plt.ylabel("f2")
plt.show()
```



Let us pretend the points represent the Pareto front for a problem with two objectives to be minimized. We can easily determine the ideal and nadir points as follows:

[2]:
```
pfront = np.stack((f1, f2)).T

ideal = np.min(pfront, axis=0)
nadir = np.max(pfront, axis=0)

print(f"Ideal point: {ideal}")
print(f"Nadir point: {nadir}")

plt.scatter(f1, f2, label="Pareto front")
plt.scatter(ideal[0], ideal[1], label="ideal")
plt.scatter(nadir[0], nadir[1], label="nadir")
plt.title("Pareto front")
plt.xlabel("f1")
plt.ylabel("f2")
plt.legend()
plt.show()
```

```
Ideal point: [1. 1.]
Nadir point: [  100. 10000.]
```



Next, suppose we would like to find a solution close to the point (80, 2500), let us define that point as a reference point.

[3]:
```
z = np.array([80, 2500])
```

Clearly, `z` is not on the Pareto front. We can find a closest solution by scalarizing the problem using an achievement scalarizing function (ASF) and minimizing the related achievement scalarizing optimization problem. We will do that next.

[4]:
```
from desdeo_tools.scalarization.ASF import PointMethodASF
from desdeo_tools.scalarization.Scalarizer import DiscreteScalarizer
from desdeo_tools.solver.ScalarSolver import DiscreteMinimizer

# define the achievement scalarizing function
asf = PointMethodASF(nadir, ideal)
# the scalarizer
dscalarizer = DiscreteScalarizer(asf, scalarizer_args={"reference_point": z})
# the solver (minimizer)
dminimizer = DiscreteMinimizer(dscalarizer)

solution_i = dminimizer.minimize(pfront)

print(f"Index of the objective vector minimizing the ASF problem: {solution_i}")
```

```
Index of the objective vector minimizing the ASF problem: 32
```

When a scalar problem is minimized using a `DiscreteMinimizer`, the result will be the index of the objective vector in the supplied `vector` argument minimizing the `DiscreteScalarizer` defined in `DiscreteMinimizer`. This is done because it is assumed that the corresponding decision variables are also kept in a vector somewhere, and the variables are ordered in a manner where the `i`th element in `vectors` corresponds to the `i`th variables in the vector storing the variables.

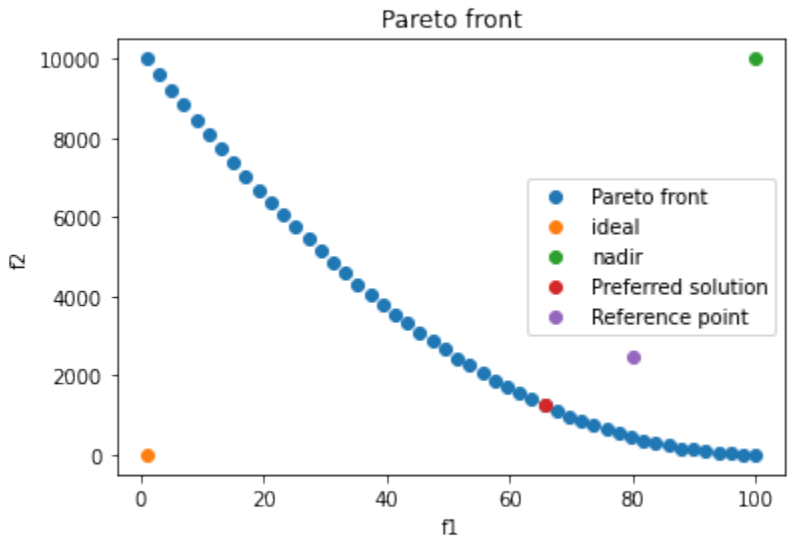
Anyway, let us plot the solution:

[5]:
```
plt.scatter(f1, f2, label="Pareto front")
plt.scatter(ideal[0], ideal[1], label="ideal")
plt.scatter(nadir[0], nadir[1], label="nadir")
```

(continues on next page)

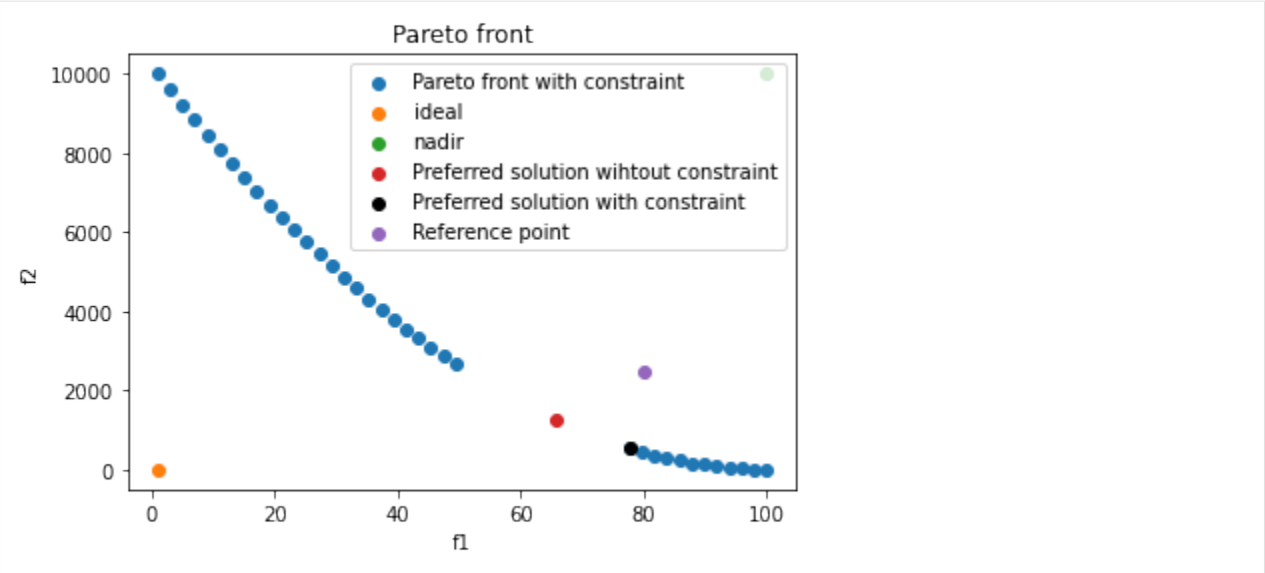
(continued from previous page)

```
plt.scatter(pfront[solution_i][0], pfront[solution_i][1], label="Preferred solution")
plt.scatter(z[0], z[1], label="Reference point")
plt.title("Pareto front")
plt.xlabel("f1")
plt.ylabel("f2")
plt.legend()
plt.show()
```



Suppose now that there is the following constraint to our problem: values of `f1` should be less than 50 or more than 77. We can easily deal with this situation as well, and we will conclude our example here.

[6]:
```
# define the constraint function, it should return either True of False for each
# objective vector defined in its argument.
def con(fs):
    fs = np.atleast_2d(fs)

    return np.logical_or(fs[:, 0] < 50, fs[:, 0] > 77)

dminimizer_con = DiscreteMinimizer(dscalarizer, con)

solution_con = dminimizer_con.minimize(pfront)

mask = con(pfront)
plt.scatter(f1[mask], f2[mask], label="Pareto front with constraint")
plt.scatter(ideal[0], ideal[1], label="ideal")
plt.scatter(nadir[0], nadir[1], label="nadir")
plt.scatter(pfront[solution_i][0], pfront[solution_i][1], label="Preferred solution␣
↪wihtout constraint")
plt.scatter(pfront[solution_con][0], pfront[solution_con][1], label="Preferred␣
↪solution with constraint", color="black")
plt.scatter(z[0], z[1], label="Reference point")
plt.title("Pareto front")
plt.xlabel("f1")
plt.ylabel("f2")
plt.legend()
plt.show()
```



Example on the usage of Scalarizer and ScalarSolver

This notebook will go through a simple example on how to scalarize a vector valued funciton and solve it using a minimizer.

Suppose we are tasked with baking a birthday cake for our friend. We will be modelling the cake as a cylinder with a height h and radius r, both in centimeters. Therefore, the cake will have a volume of

$$V(r, h) = \pi r^2 \times h$$

and a surface area queal to

$$A(r, h) = 2\pi r^2 + \pi r h.$$

Just to keep the cake realistical, let us limit the radius to be greater than 2.5cm and less than 15cm, that is $2.5 < r < 15$. The height should not exceed 50cm and be no less than 10cm: $10 < h < 50$.

We are baking the cake for a very particular friend who just fancies cake crust, and he does not really care for the filling. This implies that we would like to bake a cake which has a surface are A as large as possible while having a volume V as small as possible. In other words, we wish to maximize the surface area of the cake and minimize the volume.

Unfortunately our friend is also very picky about ratios and he has requested that the ratio of the radius and height of the cake should not exceed the golden ratio 1.618.

This can be formualted as a multi-objective optimization problem with two objectives and two constraints. Formally

$$\begin{aligned} \min_{r,h} \{V(r, h), -A(r, h)\} \\ s.t. \quad \frac{r}{h} < 1.618, \\ 2.5 < r < 15, \\ 10 < h < 50. \end{aligned}$$

We will begin by expressing all of this in Python:

[1]:
```
import numpy as np

# objectives

def volume(r, h):
    return np.pi*r**2*h

def area(r, h):
    return 2*np.pi**2 + np.pi*r*h

def objective(xs):
    # xs is a 2d array like, which has different values for r and h on its first and␣
↪second columns respectively.
    xs = np.atleast_2d(xs)
    return np.stack((volume(xs[:, 0], xs[:, 1]), -area(xs[:, 0], xs[:, 1]))).T

# bounds

r_bounds = np.array([2.5, 15])
h_bounds = np.array([10, 50])
bounds = np.stack((r_bounds, h_bounds))

# constraints

def con_golden(xs):
    # constraints are defined in DESDEO in a way were a positive value indicates an␣
↪agreement with a constraint, and
    # a negative one a disagreement.
    xs = np.atleast_2d(xs)
    return -(xs[:, 0] / xs[:, 1] - 1.618)
```

To solve this problem, we will need to scalarize it. However, before we will be able to scalarize `objective` we will need some scalarization function:

[2]:
```
def simple_sum(xs):
    xs = np.atleast_2d(xs)
    return np.sum(xs, axis=1)
```

Now we are in a position where we can scalarize `objective` using `simple_sum`:

[3]:
```
from desdeo_tools.scalarization.Scalarizer import Scalarizer

scalarized_objective = Scalarizer(objective, simple_sum)
```

In DESDEO, optimization will always mean minimization, at least internally. This is why we will be using a `ScalarMinimizer` to optimize `scalaralized_objective`.

[4]:
```
from desdeo_tools.solver.ScalarSolver import ScalarMinimizer
from scipy.optimize import NonlinearConstraint

# by setting the method to be none, we will actually be using the minimizer␣
↪implemented
# in the SciPy library.

minimizer = ScalarMinimizer(scalarized_objective, bounds, constraint_evaluator=con_
↪golden, method=None)
```

(continues on next page)

(continued from previous page)

```
# we need to supply an initial guess
x0 = np.array([2.6, 11])
sum_res = minimizer.minimize(x0)

# the optimal solution and function value
x_optimal, f_optimal = sum_res["x"], sum_res["fun"]
objective_optimal = objective(sum_res["x"]).squeeze()

print(f"\"Optimal\" cake specs: radius: {x_optimal[0]}cm, height: {x_optimal[1]}cm.")
print(f"\"Optimal\" cake dimensions: volume: {objective_optimal[0]}, area: {-
↪objective_optimal[1]}.")
```

```
"Optimal" cake specs: radius: 2.50000100052373cm, height: 10.000001000042642cm.
"Optimal" cake dimensions: volume: 196.34971764710062, area: 98.27906442862323.
```

Are we happy with this solution? No... Clearly the area of the cake could be bigger. Let us next solve for a representation of the Pareto optimal front for the defined problem. We can do this by using an achievement scalarizing function and solving the scalarized problem with a bunch of evenly generated reference points. We start by calculating the ideal and nadir points, then create a simple achievement scalarizing function, and finally generate an evenly spread set of reference points and solve the original problem by scalarizing it with the achievement scalarizing function using the generated reference points and minimizing it invidually with each reference point.

[5]:
```
# define a new scalarizing function so that each of the objectives can be optimized␣
↪independently
def weighted_sum(xs, ws):
    # ws stand for weights
    return np.sum(ws * xs, axis=1)


# minimize the first objective
weighted_scalarized_objective = Scalarizer(objective, weighted_sum, scalarizer_args={
↪"ws": np.array([1, 0])})
minimizer._scalarizer = weighted_scalarized_objective
res = minimizer.minimize(x0)
first_obj_vals = objective(res["x"])

# minimize the second objective
weighted_scalarized_objective._scalarizer_args = {"ws": np.array([0, 1])}
res = minimizer.minimize(x0)
second_obj_vals = objective(res["x"])

# payoff table
po_table = np.stack((first_obj_vals, second_obj_vals)).squeeze()

ideal = np.diagonal(po_table)
nadir = np.max(po_table, axis=0)


from desdeo_tools.scalarization.ASF import PointMethodASF

# evenly spread reference points
zs = np.mgrid[ideal[0]:nadir[0]:1500, ideal[1]:nadir[1]:150].reshape(2, -1).T


asf = PointMethodASF(nadir, ideal)
asf_scalarizer = Scalarizer(objective, asf, scalarizer_args={"reference_point": None})
minimizer._scalarizer = asf_scalarizer
```

(continues on next page)

(continued from previous page)

```
fs = np.zeros(zs.shape)

for i, z in enumerate(zs):
    asf_scalarizer._scalarizer_args={"reference_point": z}
    res = minimizer.minimize(x0)
    # assuming minimization is always a success
    fs[i] = objective(res["x"])


# plot the Pareto solutions in the original scale
import matplotlib.pyplot as plt

plt.title("Cake options")
plt.scatter(fs[:, 0], -fs[:, 1], label="Cake options")
plt.scatter(nadir[0], -nadir[1], label="nadir")
plt.scatter(ideal[0], -ideal[1], label="ideal")
plt.xlabel("Volume")
plt.ylabel("Surface area")
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7f25d59e7a60>
```

Observing the Pareto optimal front, it is clear that our previous optimal objective values `objective_optimal` are just one available option. We show our friend the available options and he decides that he wants a cake with a volume of `25000` and a surface area of `2000`. Great, now we just have to figure out the radius and height of such a cake. This should be easy:

[6]:
```
# final reference point chosen by our friend
z = np.array([25000, -2000])
asf_scalarizer._scalarizer_args={"reference_point": z}
res = minimizer.minimize(x0)

final_r, final_h = res["x"][0], res["x"][1]
final_obj = objective(res["x"]).squeeze()
final_V, final_A = final_obj[0], final_obj[1]

print(f"Final cake specs: radius: {final_r}cm, height: {final_h}cm.")
print(f"Final cake dimensions: volume: {final_V}, area: {-final_A}.")
print(final_r/final_h)
```

```
Final cake specs: radius: 12.612270698952173cm, height: 49.999999cm.
Final cake dimensions: volume: 24986.558053433215, area: 2000.8700178252593.
0.2522454190239518
```

That is a big cake!

[ ]:

2.2.2 API Reference

This page contains auto-generated API reference documentation[1].

maps

This module implements methods for defining functions which map vectors to vectors (of same or different lengths).

Examples include simple transformations such as translation or rotation, and complicated mappings such as the creation of Preference Incorporated Spaces.

Submodules

maps.preference_incorporated_space_RP

Module Contents

Classes

__PreferenceIncorporatedSpace	
classificationPIS	Implements the preference incorporated space mapping which uses the classification preference.

exception maps.preference_incorporated_space_RP.**PreferenceIncorporatedSpaceError**
Bases: Exception

Raised when an error related to the preference incorporated space is encountered.

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

class maps.preference_incorporated_space_RP.**__PreferenceIncorporatedSpace**(*scalarizers: List[Type[desdeo_tools.s utopian: numpy.ndarray, nadir: numpy.ndarray, preference: dict, rho: float = 1e-06*)

update_map(*self*, *utopian: numpy.ndarray*, *nadir: numpy.ndarray*, *preference: dict*, *scalarizers: List[Type[desdeo_tools.scalarization.GLIDE_II.GLIDEBase]] = None*, *rho: float = 1e-06*)

__call__(*self*, *objective_vector: numpy.ndarray*)

[1] Created with sphinx-autoapi

evaluate_constraints(*self*, *objective_vector: numpy.ndarray*)

class maps.preference_incorporated_space_RP.**classificationPIS**(*scalarizers: List[Type[desdeo_tools.scalarization.GLI*
utopian: numpy.ndarray, *nadir: numpy.ndarray*, *rho: float = 1e-06*)

Implements the preference incorporated space mapping which uses the classification preference.

Parameters

- **scalarizers** (*List[Type[GLIDEBase]]*) – Scalarizers to be used to create the PIS. Should include atleast one scalarizer. NIMBUS should not be included as it is added automatically.
- **utopian** (*np.ndarray*) – The utopian point of the problem.
- **nadir** (*np.ndarray*) – The nadir point of the problem.
- **rho** (*float, optional*) – The augmentation factor used in the different scalarizers. Defaults to 1e-6.

update_map(*self*, *utopian: numpy.ndarray*, *nadir: numpy.ndarray*, *scalarizers: List[Type[desdeo_tools.scalarization.GLIDE_II.GLIDEBase]] = None*, *rho: float = 1e-06*)

update_preference(*self*, *preference: dict*)

__call__(*self*, *objective_vector: numpy.ndarray*)

utilities

This module implements various small tools that may be useful during optimization using either the MCDM or the EA methods.

Submodules

utilities.distance_to_reference_point

Module Contents

Functions

distance_to_reference_point(obj: numpy.ndarray, reference_point: numpy.ndarray) → Tuple	Computes the closest solution to a reference point using achievement scalarizing function.

utilities.distance_to_reference_point.**distance_to_reference_point**(*obj: numpy.ndarray*, *reference_point: numpy.ndarray*) → Tuple

Computes the closest solution to a reference point using achievement scalarizing function.

Parameters

- **obj** (*np.ndarray*) – Array of the solutions. Should be 2d-array.
- **reference_point** (*np.ndarray*) – The reference point array. Should be one dimensional array.

Returns Returns a tuple containing the closest solution to a reference point and the index of it in obj.

Return type Tuple

utilities.fast_non_dominated_sorting

Module Contents

Functions

dominates(x: numpy.ndarray, y: numpy.ndarray) → bool	Returns true if x dominates y.
non_dominated(data: numpy.ndarray) → numpy.ndarray	Finds the non-dominated front from a population of solutions.
fast_non_dominated_sort(data: numpy.ndarray) → numpy.ndarray	Conduct fast non-dominated sorting on a population of solutions.
fast_non_dominated_sort_indices(data: numpy.ndarray) → List[numpy.ndarray]	Conduct fast non-dominated sorting on a population of solutions.

utilities.fast_non_dominated_sorting.**dominates**(*x: numpy.ndarray*, *y: numpy.ndarray*) → bool

Returns true if x dominates y.

Parameters

- **x** (*np.ndarray*) – First solution. Should be a 1-D array of numerics.
- **y** (*np.ndarray*) – Second solution. Should be the same shape as x.

Returns True if x dominates y, false otherwise.

Return type bool

utilities.fast_non_dominated_sorting.**non_dominated**(*data: numpy.ndarray*) → numpy.ndarray

Finds the non-dominated front from a population of solutions.

Parameters **data** (*np.ndarray*) – 2-D array of solutions, with each row being a single solution.

Returns

Boolean array of same length as number of solutions (rows). The value is true if corresponding solution is non-dominated. False otherwise

Return type np.ndarray

utilities.fast_non_dominated_sorting.**fast_non_dominated_sort**(*data: numpy.ndarray*) → numpy.ndarray

Conduct fast non-dominated sorting on a population of solutions.

Parameters **data** (*np.ndarray*) – 2-D array of solutions, with each row being a single solution.

Returns

n x f boolean array. n is the number of solutions, f is the number of fronts. The value of an array element is true if the corresponding solution id (column) belongs in the corresponding front (row).

Return type np.ndarray

utilities.fast_non_dominated_sorting.**fast_non_dominated_sort_indices**(*data: numpy.ndarray*) → List[numpy.ndarray]

Conduct fast non-dominated sorting on a population of solutions.

This function returns identical results as *fast_non_dominated_sort*, but in a different format. This function returns an array of solution indices for each front, packed in a list.

Parameters **data** (*np.ndarray*) – 2-D array of solutions, with each row being a single solution.

Returns

A list with f elements where f is the number of fronts in the data, arranged in ascending order. Each element is a numpy array of the indices of solutions belonging to the corresponding front.

Return type List[np.ndarray]

utilities.frozen

Freeze a class, i.e., prevent setting new attributes outside __init__.

raises TypeError Raised when setting a new attribute in a frozen class.

Module Contents

Classes

FrozenClass

class utilities.frozen.**FrozenClass**

Bases: object

__isfrozen = False

__setattr__(*self*, *key*, *value*)

Implement setattr(self, name, value).

_freeze(*self*)

utilities.lattice_generators

A file to contain different kinds of lattice generation algorithms.

Module Contents

Functions

fibonacci_sphere(samples: int = 1000) → numpy.ndarray	Generate a very even lattice of points on a 3d sphere using the fibonacci sphere

utilities.lattice_generators.**fibonacci_sphere**(*samples: int = 1000*) → numpy.ndarray

Generate a very even lattice of points on a 3d sphere using the fibonacci sphere or fibonacci spiral algorithm.

Parameters **samples** (*int, optional*) – Number of points to be generated. Defaults to 1000.

Returns The lattice of points as a 2-D (samples, 3) numpy array.

Return type np.ndarray

utilities.polytopes

Module which handles polytopes.

Module Contents

Functions

inherently_nondominated(A: numpy.ndarray, epsilon: Optional[float] = 1e-06, method: Optional[str] = 'highs') → bool	Check if a polytope is inherently nondominated:
polytope_dominates(k1: numpy.ndarray, k2: numpy.ndarray, epsilon: Optional[float] = 1e-06, method: Optional[str] = 'highs') → bool	Check if polytope p(k1) dominates polytope p(k2) with epsilon certainty
generate_polytopes(simplices: numpy.ndarray) → numpy.ndarray	Generate polytopes from an array of indices which form simplices

utilities.polytopes.**inherently_nondominated**(*A: numpy.ndarray*, *epsilon: Optional[float] = 1e-06*, *method: Optional[str] = 'highs'*) → bool

Check if a polytope is inherently nondominated: A polytope is inherently nondominated iff the polytope does not dominate itself.

Parameters

- **A** (*np.ndarray*) – A polytope to be checked.
- **epsilon** (*Optional[float], optional*) – precision parameter, see polytope_dominates for further details. Defaults to 1e-6.
- **method** (*Optional[str], optional*) – Algorithm used to solve the optimization problems. Defaults to 'highs'.

Returns is the given set inherently nondominated.

Return type bool

utilities.polytopes.**polytope_dominates**(*k1: numpy.ndarray*, *k2: numpy.ndarray*, *epsilon: Optional[float] = 1e-06*, *method: Optional[str] = 'highs'*) → bool

Check if polytope p(k1) dominates polytope p(k2) with epsilon certainty by solving linear optimization problems [min_x c^T*x] using linprog from scipy.optimize.

Parameters

- **k1** (*np.ndarray*) – Corners of first polytope
- **k2** (*np.ndarray*) – Corners of seconds polytope
- **epsilon** (*Optional[float], optional*) – precision parameter. Defaults to 1e-6
- **method** (*Optional[str], optional*) – Algorithm used to solve the optimization problems. Defaults to 'highs'. See scipy.optimize.linprog for further details.

Returns Does polytope p(k1) dominate polytope p(k2).

Return type bool

utilities.polytopes.**generate_polytopes**(*simplices: numpy.ndarray*) → numpy.ndarray

Generate polytopes from an array of indices which form simplices

Parameters **arr** (*np.ndarray*) – An array of indices which form simplices. In PAINT this is the array of simplices which form the Delaunay triangulation.

Returns

np.ndarray: An array of indices which form the polytopes that are generated from the given array. If a polytope has fewer outcomes than there are columns in the given array the first value of the row representing the polytope is repeated until the lengths match.

utilities.preference_converters

Provides implementations that convert one type of preference information to another.

Module Contents

Functions

classification_to_reference_point(classification_preference: dict, ideal: numpy.ndarray, nadir: numpy.ndarray) → dict	Convert classification type of preference (e.g. NIMBUS) to reference point

utilities.preference_converters.**classification_to_reference_point**(*classification_preference: dict*, *ideal: numpy.ndarray*, *nadir: numpy.ndarray*) → dict

Convert classification type of preference (e.g. NIMBUS) to reference point preference.

Parameters

- **classification_preference** (*dict*) – A dict containing keys 'current solution', 'levels', and 'classifications'. Read the NIMBUS paper for more details.
- **ideal** (*np.ndarray*) – The ideal point of the problem.
- **nadir** (*np.ndarray*) – The nadir point of the problem.

Returns

The preference in the form of a reference point. Contains one key: "reference point", which maps to the preference in a numpy array structure.

Return type dict

utilities.quality_indicator

Module Contents

Functions

epsilon_indicator(s1: numpy.ndarray, s2: numpy.ndarray) → float	Computes the additive epsilon-indicator between two solutions.
epsilon_indicator_ndims(front: numpy.ndarray, reference_point: numpy.ndarray) → list	Computes the additive epsilon-indicator between reference point and current one-dimensional vector of front.
preference_indicator(s1: numpy.ndarray, s2: numpy.ndarray, min_asf_value: float, ref_point: numpy.ndarray, delta: float) → float	Computes the preference-based quality indicator.
hypervolume_indicator(front: numpy.ndarray, reference_point: numpy.ndarray) → float	Computes the hypervolume-indicator between reference front and current approximating point.

Attributes

po_front

utilities.quality_indicator.**epsilon_indicator**(*s1: numpy.ndarray*, *s2: numpy.ndarray*) → float

Computes the additive epsilon-indicator between two solutions.

Parameters

- **s1** (*np.ndarray*) – Solution 1. Should be an one-dimensional array.
- **s2** (*np.ndarray*) – Solution 2. Should be an one-dimensional array.

Returns The maximum distance between the values in s1 and s2.

Return type float

utilities.quality_indicator.**epsilon_indicator_ndims**(*front: numpy.ndarray*, *reference_point: numpy.ndarray*) → list

Computes the additive epsilon-indicator between reference point and current one-dimensional vector of front.

Parameters

- **front** (*np.ndarray*) – The front that the current reference point is being compared to. Should be set of arrays, where the rows are the solutions and the columns are the objective dimensions.
- **reference_point** (*np.ndarray*) – The reference point that is compared. Should be one-dimensional array.

Returns The list of indicator values.

Return type list

utilities.quality_indicator.**preference_indicator**(*s1: numpy.ndarray*, *s2: numpy.ndarray*, *min_asf_value: float*, *ref_point: numpy.ndarray*, *delta: float*) → float

Computes the preference-based quality indicator.

Parameters

- **s1** (*np.ndarray*) – Solution 1. Should be an one-dimensional array.
- **s2** (*np.ndarray*) – Solution 2. Should be an one-dimensional array.
- **ref_point** (*np.ndarray*) – The reference point should be same shape as front.
- **min_asf_value** (*float*) – Minimum value of achievement scalarization of the reference_front. Used in normalization.
- **delta** (*float*) – The spesifity delta allows to set the amplification of the indicator to be closer or farther from the reference point. Smaller delta means that all solutions are in smaller range around the reference point.

Returns

The maximum distance between the values in s1 and s2 taking into account the reference point and spesifity.

Return type float

utilities.quality_indicator.**hypervolume_indicator**(*front: numpy.ndarray*, *reference_point: numpy.ndarray*) → float

Computes the hypervolume-indicator between reference front and current approximating point.

Parameters

- **front** (*np.ndarray*) – The front that is compared. Should be set of arrays, where the rows are the solutions and the columns are the objective dimensions.
- **reference_point** (*np.ndarray*) – The reference point that the current front is being compared to. Should be 1D array.

Returns Measures the volume of the objective space dominated by an approximation set.

Return type float

utilities.quality_indicator.**po_front**

interaction

This module contains classes implementing different interactions to be used to communicate between different optimization algorithms and users.

Submodules

interaction.request

Module Contents

Classes

BaseRequest	The base class for all Request classes. Request classes are to be used
PrintRequest	Methods can use this request class to send out textual information to be
SimplePlotRequest	Methods can use this request class to send out some data to be shown to
ReferencePointPreference	Methods can use this request class to ask the DM to provide their preferences
PreferredSolutionPreference	Methods can use this class to ask the Decision maker to provide their preferences in form of preferred solution(s).
NonPreferredSolutionPreference	Methods can use this class to ask the Decision maker to provide their preferences in form of non-preferred
BoundPreference	Methods can use this class to ask the Decision maker to provide their preferences in form of preferred lower and

exception interaction.request.**RequestError**

Bases: Exception

Raised when an error related to the Request class is encountered.

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

class interaction.request.**BaseRequest**(*request_type: str*, *interaction_priority: str*, *content=None*, *request_id: int = None*)

Bases: desdeo_tools.utilities.frozen.FrozenClass

The base class for all Request classes. Request classes are to be used to handle interaction between the user and the methods, as well as within various methods. This class is frozen, so no variables other than that already defined in current __init__ can be defined in derived classes.

Initialize a BaseRequest class. This method contains a lot of boilerplate.

Parameters

- **request_type** (*str*) – The type of request. Currently, one of [“print”, “simple_plot”, “reference_point_preference”, “classification_preference”].
- **interaction_priority** (*str*) – The priority of preference, as decided by the method. One of [“no_interaction”, “not_required”, “recommended”, “required”], with trivial meanings.

- **content** (*[type], optional*) – The data relevant to the request packet. For example, if the request type is print, content may contain strings to be printed. Typically a dict. Defaults to None.
- **request_id** (*int, optional*) – A unique identifier. Defaults to None.

Raises

- ***RequestError*** – If request type is not recognized
- ***RequestError*** – If request priority is not recognized
- ***RequestError*** – If request id is not an integer

property request_type(*self*)

property interaction_priority(*self*)

property request_id(*self*)

property content(*self*)

property response(*self*)

class interaction.request.**PrintRequest**(*message: Union[str, List[str]], request_id: int = None*)

Bases: *BaseRequest*

Methods can use this request class to send out textual information to be displayed to the decision maker. This could be a single message in the form of a string, or multiple messages in a list of strings. The method of displaying these messages is left to the UI.

Initialise the PrintRequest.

Parameters

- **message** (*Union[str, List[str]]*) – A single message (str) or a list of messages to be displayed to the decision maker
- **request_id** (*int, optional*) – A unique identifier for this request. Defaults to None.

Raises

- ***RequestError*** – If message is not a str or a list
- ***RequestError*** – If message is a list but one or more elements are not str.

class interaction.request.**SimplePlotRequest**(*data: pandas.DataFrame, message: Union[str, List[str]], dimensions_data: pandas.DataFrame = None, chart_title: str = None, request_id=None*)

Bases: *BaseRequest*

Methods can use this request class to send out some data to be shown to the decision maker (typically in the form of a plot). This data is usually a set of solutions, stored in the content variable of this class. The manner of visualization is left to the UI.

content is a dict that contains the following keys: "data" (pandas.DataFrame): The data to be plotted. "dimensional_data" (pandas.Dataframe): The data contained in this key can be

used to scale the data to be plotted.

"chart_title" (str): A recommended title for the visualization. "message" (Union[str, List[str]]): A message or list of messages to be

displayed to the decision maker.

Initialize the request packet

Parameters

- **data** (*pd.DataFrame*) – The data to be plotted.
- **message** (*Union[str, List[str]]*) – A message or list of messages to be displayed to the decision maker.
- **dimensions_data** (*pd.DataFrame, optional*) – Data used to used to scale the data to be plotted. Defaults to None.
- **chart_title** (*str, optional*) – A recommended title for the visualization. Defaults to None.
- **request_id** (*[type], optional*) – A unique identifier. Defaults to None.

Raises

- ***RequestError*** – data is not a pandas DataFrame.
- ***RequestError*** – dimensions_data is not a pandas DataFrame or None.
- ***RequestError*** – A mismatch in the column names of data and dimensions_data.
- ***RequestError*** – If dimensions_data DataFrame contains indices other that "minimize", "ideal", or "nadir".
- ***RequestError*** – If chart_title is not str or None.
- ***RequestError*** – If message is not a str or a list.
- ***RequestError*** – If message is a list but one or more elements are not str.

class interaction.request.**ReferencePointPreference**(*dimensions_data: pandas.DataFrame*, *message: str = None*, *interaction_priority: str = 'required'*, *preference_validator: Callable = None*, *request_id: int = None*)

Bases: *BaseRequest*

Methods can use this request class to ask the DM to provide their preferences in the form of a reference point. This reference point is validated according to the needs of the method that initializes this class object, before the reference point can be accepted in the response variable.

Initialize the request class.

Parameters

- **dimensions_data** (*pd.DataFrame*) – Minimal data that should be shown to the decision maker. If a lot of data needs to be shown (i.e., with a visualization), use SimplePlotRequest or related classes for that purpose, and this class for the interaction with the decision maker.
- **message** (*str, optional*) – Message to be displayed to a decision maker. Defaults to None.
- **interaction_priority** (*str, optional*) – The importance of the interaction as decided by the method. If equal to "required", the method will not continue without a DM preference. If equal to "recommended", the interaction is recommended, but not required for the continuation of the method. The case "not_required" is similar to "recommended". Defaults to "required".

- **preference_validator** (*Callable, optional*) – A callable function that tests whether a reference point provided by the DM is valid or not. Defaults to None.
- **request_id** (*int, optional*) – A unique identifier. Defaults to None.

Raises

- ***RequestError*** – dimensions_data is not a pandas DataFrame.
- ***RequestError*** – If dimensions_data DataFrame contains indices other that "minimize", "ideal", or "nadir".
- ***RequestError*** – If message is not a str or a list.
- ***RequestError*** – If message is a list but one or more elements are not str.

class interaction.request.**PreferredSolutionPreference**(*n_solutions: int*, *message: str = None*, *interaction_priority: str = 'required'*, *preference_validator: Callable = None*, *request_id: int = None*)

Bases: *BaseRequest*

Methods can use this class to ask the Decision maker to provide their preferences in form of preferred solution(s).

Initialize preference-class with information about problem.

Parameters

- **n_solutions** (*int*) – Number of solutions in total.
- **message** (*str*) – Message to be displayed to the Decision maker.
- **interaction_priority** (*str*) – Level of priority.
- **preference_validator** (*Callable*) – Function that validates the Decision maker's preferences.
- **request_id** (*int*) – Identification number of request.

class interaction.request.**NonPreferredSolutionPreference**(*n_solutions: int*, *message: str = None*, *interaction_priority: str = 'required'*, *preference_validator: Callable = None*, *request_id: int = None*)

Bases: *BaseRequest*

Methods can use this class to ask the Decision maker to provide their preferences in form of non-preferred solution(s).

Initialize preference-class with information about problem.

Parameters

- **n_solutions** (*int*) – Number of solutions in total.
- **message** (*str*) – Message to be displayed to the Decision maker.
- **interaction_priority** (*str*) – Level of priority.
- **preference_validator** (*Callable*) – Function that validates the Decision maker's preferences.

- **request_id** (*int*) – Identification number of request.

class interaction.request.**BoundPreference**(*dimensions_data: pandas.DataFrame, n_objectives: int, message: str = None, interaction_priority: str = 'required', preference_validator: Callable = None, request_id: int = None*)

Bases: *BaseRequest*

Methods can use this class to ask the Decision maker to provide their preferences in form of preferred lower and upper bounds for objective values.

Initialize preference-class with information about problem.

Parameters

- **dimensions_data** (*pd.DataFrame*) – DataFrame including information whether an objective is minimized or maximized, for each objective. In addition, includes ideal and nadir vectors.
- **n_objectives** (*int*) – Number of objectives in problem.
- **message** (*str*) – Message to be displayed to the Decision maker.
- **interaction_priority** (*str*) – Level of priority.
- **preference_validator** (*Callable*) – Function that validates the Decision maker's preferences.
- **request_id** (*int*) – Identification number of request.

interaction.validators

Module Contents

Functions

validate_ref_point_with_ideal_and_nadir(dimensions_data: pandas.DataFrame, reference_point: pandas.DataFrame)	
validate_ref_point_with_ideal(dimensions_data: pandas.DataFrame, reference_point: pandas.DataFrame)	
validate_with_ref_point_nadir(dimensions_data: pandas.DataFrame, reference_point: pandas.DataFrame)	
validate_ref_point_dimensions(dimensions_data: pandas.DataFrame, reference_point: pandas.DataFrame)	
validate_ref_point_data_type(reference_point: pandas.DataFrame)	
validate_specified_solutions(indices: numpy.ndarray, n_solutions: int) → None	Validate the Decision maker's choice of preferred/non-preferred solutions.
validate_bounds(dimensions_data: pandas.DataFrame, bounds: numpy.ndarray, n_objectives: int) → None	Validate the Decision maker's desired lower and upper bounds for objective values.

exception interaction.validators.**ValidationError**
Bases: Exception

Raised when an error related to the validation is encountered.

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

interaction.validators.**validate_ref_point_with_ideal_and_nadir**(*dimensions_data: pandas.DataFrame*, *reference_point: pandas.DataFrame*)

interaction.validators.**validate_ref_point_with_ideal**(*dimensions_data: pandas.DataFrame*, *reference_point: pandas.DataFrame*)

interaction.validators.**validate_with_ref_point_nadir**(*dimensions_data: pandas.DataFrame*, *reference_point: pandas.DataFrame*)

interaction.validators.**validate_ref_point_dimensions**(*dimensions_data: pandas.DataFrame*, *reference_point: pandas.DataFrame*)

interaction.validators.**validate_ref_point_data_type**(*reference_point: pandas.DataFrame*)

interaction.validators.**validate_specified_solutions**(*indices: numpy.ndarray*, *n_solutions: int*) → None
Validate the Decision maker's choice of preferred/non-preferred solutions.

Parameters

- **indices** (*np.ndarray*) – Index/indices of preferred solutions specified by the Decision maker.
- **n_solutions** (*int*) – Number of solutions in total.

Returns:

Raises ***ValidationError*** – In case the preference is invalid.

interaction.validators.**validate_bounds**(*dimensions_data: pandas.DataFrame*, *bounds: numpy.ndarray*, *n_objectives: int*) → None
Validate the Decision maker's desired lower and upper bounds for objective values.

Parameters

- **dimensions_data** (*pd.DataFrame*) – DataFrame including information whether an objective is minimized or maximized, for each objective. In addition, includes ideal and nadir vectors.
- **bounds** (*np.ndarray*) – Desired lower and upper bounds for each objective.
- **n_objectives** (*int*) – Number of objectives in problem.

Returns:

Raises ***ValidationError*** – In case desired bounds are invalid.

solver

This module implements methods for solving scalar valued functions.

Submodules

solver.ScalarSolver

Implements methods for solving scalar valued functions.

Module Contents

Classes

ScalarMethod	A class the define and implement methods for minimizing scalar valued functions.
ScalarMinimizer	Implements a class for minimizing scalar valued functions with bounds set for the
DiscreteMinimizer	Implements a class for finding the minimum value of a discrete of scalarized vectors.

Attributes

ideal

exception solver.ScalarSolver.**ScalarSolverException**

Bases: Exception

Common base class for all non-exit exceptions.

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

class solver.ScalarSolver.**ScalarMethod**(*method: Callable*, *method_args=None*, *use_scipy: Optional[bool] = False*)

A class the define and implement methods for minimizing scalar valued functions.

Parameters

- **method** (*Callable*) – A callable minimizer function which expects a callable scalar valued function to be minimized. The function should accept as its first argument a two dimensional numpy array and should return a dictionary with at least the keys: "x" the found optimal solution, "success" boolean indicating if the minimization was successfull, "message" a string of additional info.
- **method_args** (*Dict, optional*) – Any other keyword arguments to be supplied to the method. Defaults to None.
- **use_scipy** (*Optional[bool]*) – Whether to use scipy's NonLinearConstraint to handle the constraints.

__call__(*self*, *obj_fun: Callable*, *x0: numpy.ndarray*, *bounds: numpy.ndarray*, *constraint_evaluator: Callable*) → Dict
Minimizes a scalar valued function.

Parameters

- **obj_fun** (*Callable*) – A callable scalar valued function that accepts a two dimensional numpy array as its first arguments.
- **x0** (*np.ndarray*) – An initial guess.
- **bounds** (*np.ndarray*) – The upper and lower bounds for each variable accepted by obj_fun. Expects a 2D numpy array with each row representing the lower and upper bounds of a variable. The first column should contain the lower bounds and the last column the upper bounds. Use np.inf to indicate no bound.
- **constraint_evaluator** (*Callable*) – Should accepts exactly the same arguments as obj_fun. Returns a scalar value for each constraint present. This scalar value should be positive if a constraint holds, and negative otherwise.

Returns

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

Return type Dict

class solver.ScalarSolver.**ScalarMinimizer**(*scalarizer: desdeo_tools.scalarization.Scalarizer.Scalarizer*, *bounds: numpy.ndarray*, *constraint_evaluator: Callable = None*, *method: Optional[Union[*ScalarMethod*, str]] = None*)
Implements a class for minimizing scalar valued functions with bounds set for the variables, and constraints.

Parameters

- **scalarizer** (Scalarizer) – A Scalarizer to be minimized.
- **bounds** (*np.ndarray*) – The bounds of the independent variables the scalarizer is called with.
- **constraint_evaluator** (*Callable, optional*) – A Callable which representing a vector valued constraint function. The array the constraint function returns should be two dimensional with each row corresponding to the constraint function values when evaluated. A value of less than zero is understood as a non valid constraint. Defaults to None.
- **method** (*Optional[Union[Callable, str]], optional*) – The optimization method the scalarizer should be minimized with. It should accepts as keyword the arguments 'bounds' and 'constraints' which will be used to pass it the bounds and constraint_evaluator. If none is supplied, uses the minimizer implemented in SciPy. Otherwise a str can be given to use one of the preset solvers available. Use the method 'get_presets' to get a list of available preset solvers. Defaults to None.

get_presets(*self*)
Return the list of preset minimizers available.

minimize(*self*, *x0: numpy.ndarray*) → Dict
Minimizes the scalarizer given an initial guess x0.

Parameters **x0** (*np.ndarray*) – A numpy array containing an initial guess of variable values.

Returns

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

Return type Dict

class solver.ScalarSolver.**DiscreteMinimizer**(*discrete_scalarizer: desdeo_tools.scalarization.Scalarizer.DiscreteScalarizer*, *constraint_evaluator: Optional[Callable[[numpy.ndarray], numpy.ndarray]] = None*)

Implements a class for finding the minimum value of a discrete of scalarized vectors.

Parameters

- **discrete_scalarizer** (DiscreteScalarizer) – A discrete scalarizer which takes as its arguments an array of vectors and returns a scalar value for each vector.
- **(Optional[Callable[[np.ndarray]** (*constraint_evaluator*) –

:param : :param np.ndarray]]: An evaluator which returns True if a

given vector(s) adheres to given constraints, and False otherwise. Defaults to None.

Parameters optional) – An evaluator which returns True if a given vector(s) adheres to given constraints, and False otherwise. Defaults to None.

minimize(*self*, *vectors: numpy.ndarray*) → dict

Find the index of the element in vectors which minimizes the scalar value returned by the scalarizer. If multiple minimum values are found, returns the index of the first occurrence.

Parameters vectors (*np.ndarray*) – The vectors for which the minimum scalar value should be computed for.

Raises ***ScalarSolverException*** – None of the given vectors adhere to the given constraints.

Returns

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

Return type Dict

solver.ScalarSolver.**ideal**

scalarization

This module implements methods for defining functions to scalarize vector valued functions. These are knows as `Scalarizer`s. It also provides achievement scalarizing functions to be used with the scalarizers.

Submodules

scalarization.ASF

Module Contents

Classes

ASFBase	A base class for representing achievement scalarizing functions.
SimpleASF	Implements a simple order-representing ASF.
ReferencePointASF	Uses a reference point q and preferential factors to scalarize a MOO problem.
MaxOfTwoASF	Implements the ASF used in NIMBUS, which takes the maximum of two terms.
StomASF	Implementation of the satisfying trade-off method (STOM).
PointMethodASF	Implementation of the reference point based ASF.
AugmentedGuessASF	Implementation of the augmented GUESS related ASF.
GuessASF	Implementation of the naive or GUESS ASF.

exception scalarization.ASF.**ASFError**
Bases: Exception

Raised when an error related to the ASF classes is encountered.

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

class scalarization.ASF.**ASFBase**
Bases: abc.ABC

A base class for representing achievement scalarizing functions. Instances of the implementations of this class should function as function.

abstract **__call__**(*self*, *objective_vector: numpy.ndarray*, *reference_point: numpy.ndarray*) → Union[float, numpy.ndarray]
Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

class scalarization.ASF.**SimpleASF**(*weights: numpy.ndarray*)
Bases: *ASFBase*

Implements a simple order-representing ASF.

Parameters **weights** (*np.ndarray*) – A weight vector that holds weights. It's length should match the number of objectives in the underlying MOO problem the achievement problem aims to solve.

weights
A weight vector that holds weights. It's length should match the number of objectives in the underlying MOO problem the achievement problem aims to solve.

Type np.ndarray

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_point: numpy.ndarray*) → Union[float, numpy.ndarray]
Evaluate the simple order-representing ASF.

Parameters

- **objective_vector** (*np.ndarray*) – A vector representing a solution in the solution space.
- **reference_point** (*np.ndarray*) – A vector representing a reference point in the solution space.

Note: The shaped of objective_vector and reference_point must match.

class scalarization.ASF.**ReferencePointASF**(*preferential_factors: numpy.ndarray*, *nadir: numpy.ndarray*, *utopian_point: numpy.ndarray*, *rho: float = 1e-06*)
Bases: *ASFBase*

Uses a reference point q and preferential factors to scalarize a MOO problem.

Parameters

- **preferential_factors** (*np.ndarray*) – The preferential factors.
- **nadir** (*np.ndarray*) – The nadir point of the MOO problem to be scalarized.
- **utopian_point** (*np.ndarray*) – The utopian point of the MOO problem to be scalarized.
- **rho** (*float*) – A small number to be used to scale the sm factor in the ASF. Defaults to 0.1.

preferential_factors
The preferential factors.

Type np.ndarray

nadir
The nadir point of the MOO problem to be scalarized.

Type np.ndarray

utopian_point
The utopian point of the MOO problem to be scalarized.

Type np.ndarray

rho
A small number to be used to scale the sm factor in the ASF. Defaults to 0.1.

Type float

References

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

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_point: numpy.ndarray*) → Union[float, numpy.ndarray]

Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

---

class scalarization.ASF.**MaxOfTwoASF**(*nadir: numpy.ndarray*, *ideal: numpy.ndarray*, *lt_inds: List[int]*, *lte_inds: List[int]*, *rho: float = 1e-06*, *rho_sum: float = 1e-06*)

Bases: *ASFBase*

Implements the ASF used in NIMBUS, which takes the maximum of two terms.

Parameters

- **nadir** (*np.ndarray*) – The nadir point.
- **ideal** (*np.ndarray*) – The ideal point.
- **lt_inds** (*List[int]*) – Indices of the objectives categorized to be decreased.
- **lte_inds** (*List[int]*) – Indices of the objectives categorized to be reduced until some value is reached.
- **rho** (*float*) – A small number to form the utopian point.
- **rho_sum** (*float*) – A small number to be used as a weight for the sum term.

nadir
The nadir point.

Type np.ndarray

ideal
The ideal point.

Type np.ndarray

lt_inds
Indices of the objectives categorized to be decreased.

Type List[int]

lte_inds
Indices of the objectives categorized to be reduced until some value is reached.

Type List[int]

rho
A small number to form the utopian point.

Type float

rho_sum
A small number to be used as a weight for the sum term.

Type float

References

Miettinen, K. & Mäkelä, Marko M. Synchronous approach in interactive multiobjective optimization European Journal of Operational Research, 2006, 170, 909-922

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_point: numpy.ndarray*) → Union[float, numpy.ndarray]
Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

---

class scalarization.ASF.**StomASF**(*ideal: numpy.ndarray*, *rho: float = 1e-06*, *rho_sum: float = 1e-06*)
Bases: *ASFBase*

Implementation of the satisfying trade-off method (STOM).

Parameters

- **ideal** (*np.ndarray*) – The ideal point.
- **rho** (*float*) – A small number to form the utopian point.
- **rho_sum** (*float*) – A small number to be used as a weight for the sum term.

ideal
The ideal point.

Type np.ndarray

rho
A small number to form the utopian point.

Type float

rho_sum
A small number to be used as a weight for the sum term.

Type float

References

Miettinen, K. & Mäkelä, Marko M. Synchronous approach in interactive multiobjective optimization European Journal of Operational Research, 2006, 170, 909-922

__call__(*self*, *objective_vectors: numpy.ndarray*, *reference_point: numpy.ndarray*) → Union[float, numpy.ndarray]

Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

---

class scalarization.ASF.**PointMethodASF**(*nadir: numpy.ndarray*, *ideal: numpy.ndarray*, *rho: float = 1e-06*, *rho_sum: float = 1e-06*)

Bases: *ASFBase*

Implementation of the reference point based ASF.

Parameters

- **nadir** (*np.ndarray*) – The nadir point.
- **ideal** (*np.ndarray*) – The ideal point.
- **rho** (*float*) – A small number to form the utopian point.
- **rho_sum** (*float*) – A small number to be used as a weight for the sum term.

References

Miettinen, K. & Mäkelä, Marko M. Synchronous approach in interactive multiobjective optimization European Journal of Operational Research, 2006, 170, 909-922

__call__(*self*, *objective_vectors: numpy.ndarray*, *reference_point: numpy.ndarray*)

Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

class scalarization.ASF.**AugmentedGuessASF**(*nadir: numpy.ndarray*, *ideal: numpy.ndarray*, *index_to_exclude: List[int]*, *rho: float = 1e-06*, *rho_sum: float = 1e-06*)

Bases: *ASFBase*

Implementation of the augmented GUESS related ASF.

Parameters

- **nadir** (*np.ndarray*) – The nadir point.
- **ideal** (*np.ndarray*) – The ideal point.
- **index_to_exclude** (*List[int]*) – The indices of the objective functions to be excluded in calculating the first term of the ASF.
- **rho** (*float*) – A small number to form the utopian point.
- **rho_sum** (*float*) – A small number to be used as a weight for the sum term.

References

Miettinen, K. & Mäkelä, Marko M. Synchronous approach in interactive multiobjective optimization European Journal of Operational Research, 2006, 170, 909-922

__call__(*self*, *objective_vectors: numpy.ndarray*, *reference_point: numpy.ndarray*)

Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

class scalarization.ASF.**GuessASF**(*nadir: numpy.ndarray*)

Bases: *ASFBase*

Implementation of the naive or GUESS ASF.

Parameters **nadir** (*np.ndarray*) – The nadir point of the problem being scalarized.

References

Miettinen, K., Mäkelä, M. On scalarizing functions in multiobjective optimization OR Spectrum 24, 193–213 (2002)

__call__(*self*, *objective_vectors: numpy.ndarray*, *reference_point: numpy.ndarray*)
Evaluate the ASF.

Parameters

- **objective_vectors** (*np.ndarray*) – The objective vectors to calculate the values.
- **reference_point** (*np.ndarray*) – The reference point to calculate the values.

Returns

Either a single ASF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

Note: The reference point may not always necessarily be feasible, but it's dimensions should match that of the objective vector.

scalarization.EpsilonConstraintMethod

Module Contents

Classes

EpsilonConstraintMethod	A class to represent a class for scalarizing MOO problems using the epsilon

Functions

volume(r, h)

exception scalarization.EpsilonConstraintMethod.**ECMError**
Bases: Exception

Raised when an error related to the Epsilon Constraint Method is encountered.

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

class scalarization.EpsilonConstraintMethod.**EpsilonConstraintMethod**(*objectives: Callable*, *to_be_minimized: int*, *epsilons: numpy.ndarray*, *constraints: Optional[Callable]*)

A class to represent a class for scalarizing MOO problems using the epsilon constraint method.

objectives
Objective functions.

Type Callable

to_be_minimized
Integer representing which objective function should be minimized.

Type int

epsilons
Upper bounds chosen by the decison maker. Epsilon constraint functions are defined in a following form:

f_i(x) <= eps_i

If the constraint function is of form f_i(x) >= eps_i

Remember to multiply the epsilon value with -1!

Type np.ndarray

constraints
Function that returns definitions of other constraints, if existing.

Type Optional[Callable]

evaluate_constraints(*self*, *xs*) → numpy.ndarray
Returns values of constraints with given decison variables.

Parameters **xs** (*np.ndarray*) – Decision variables.

Returns Values of constraint functions (both "original" constraints as well as epsilon constraints) in a vector.

__call__(*self*, *objective_vector: numpy.ndarray*) → Union[float, numpy.ndarray]
Returns the value of objective function to be minimized.

Parameters **objective_vector** (*np.ndarray*) – Values of objective functions.

Returns Value of objective function to be minimized.

scalarization.EpsilonConstraintMethod.**volume**(*r*, *h*)

scalarization.GLIDE_II

Module Contents

Classes

GLIDEBase	Implements the non-differentiable variant of GLIDE-II as proposed in
reference_point_method_GLIDE	Implements the reference point method of preference elicitation and scalarization
GUESS_GLIDE	Implements the GUESS method of preference elicitation and scalarization
AUG_GUESS_GLIDE	Implements the Augmented GUESS method of preference elicitation and scalarization
NIMBUS_GLIDE	Implements the NIMBUS method of preference elicitation and scalarization
STEP_GLIDE	Implements the STEP method of preference elicitation and scalarization
STOM_GLIDE	Implements the STOM method of preference elicitation and scalarization
AUG_STOM_GLIDE	Implements the Augmented STOM method of preference elicitation and scalarization
Tchebycheff_GLIDE	Implements the Tchebycheff method of preference elicitation and scalarization
PROJECT_GLIDE	Implements the PROJECT method of preference elicitation and scalarization

exception scalarization.GLIDE_II.**GLIDEError**
Bases: Exception

Raised when an error related to the ASF classes is encountered.

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

class scalarization.GLIDE_II.**GLIDEBase**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Implements the non-differentiable variant of GLIDE-II as proposed in Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Note: Additional contraints produced by the GLIDE-II formulation are implemented such that if the returned values are negative, the corresponding constraint is violated. The returned value may be positive. In such cases, the returned value is a measure of how close or far the corresponding feasible solution is from violating the constraint.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.

- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

__call__(*self*, *objective_vector: numpy.ndarray*, *preference: dict*) → numpy.ndarray
Evaluate the scalarization function value based on objective vectors and DM preference.

Parameters

- **objective_vector** (*np.ndarray*) – 2-dimensional array of objective values of solutions.
- **preference** (*dict*) – The preference given by the decision maker. The required dictionary keys and their meanings can be found in self.required_keys variable.

Returns

The scalarized value obtained by using GLIDE-II over objective_vector.

Return type np.ndarray

evaluate_constraints(*self*, *objective_vector: numpy.ndarray*, *preference: dict*) → Union[None, numpy.ndarray]
Evaluate the additional contraints generated by the GLIDE-II formulation.

Note: Additional contraints produced by the GLIDE-II formulation are implemented such that if the returned values are negative, the corresponding constraint is violated. The returned value may be positive. In such cases, the returned value is a measure of how close or far the corresponding feasible solution is from violating the constraint.

Parameters

- **objective_vector** (*np.ndarray*) – [description]
- **preference** (*dict*) – [description]

Returns [description]

Return type Union[None, np.ndarray]

property I_alpha(*self*)

property I_epsilon(*self*)

property mu(*self*)

property q(*self*)

property w(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**reference_point_method_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GLIDEBase*

Implements the reference point method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property I_epsilon(*self*)

property I_alpha(*self*)

property mu(*self*)

property w(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**GUESS_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GLIDEBase*

Implements the GUESS method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property I_epsilon(*self*)

property I_alpha(*self*)

property mu(*self*)

property w(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**AUG_GUESS_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GUESS_GLIDE*

Implements the Augmented GUESS method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

class scalarization.GLIDE_II.**NIMBUS_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GLIDEBase*

Implements the NIMBUS method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property improve_unconstrained(*self*)

property improve_constrained(*self*)

property satisfactory(*self*)

property relax_constrained(*self*)

property relax_unconstrained(*self*)

property I_alpha(*self*)

property I_epsilon(*self*)

property w(*self*)

property mu(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**STEP_GLIDE**(*utopian: numpy.ndarray = None, nadir: numpy.ndarray = None, rho: float = 1e-06, **kwargs*)

Bases: *GLIDEBase*

Implements the STEP method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property improve_constrained(*self*)

property satisfactory(*self*)

property relax_constrained(*self*)

property I_alpha(*self*)

property w(*self*)

property mu(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**STOM_GLIDE**(*utopian: numpy.ndarray = None, nadir: numpy.ndarray = None, rho: float = 1e-06, **kwargs*)

Bases: *GLIDEBase*

Implements the STOM method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Has no effect on STOM calculation. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property I_epsilon(*self*)

property I_alpha(*self*)

property mu(*self*)

property w(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**AUG_STOM_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *STOM_GLIDE*

Implements the Augmented STOM method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Has no effect on STOM calculation. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

class scalarization.GLIDE_II.**Tchebycheff_GLIDE**(*utopian: numpy.ndarray = None*, *nadir: numpy.ndarray = None*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GLIDEBase*

Implements the Tchebycheff method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property I_epsilon(*self*)

property I_alpha(*self*)

property mu(*self*)

property w(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

class scalarization.GLIDE_II.**PROJECT_GLIDE**(*current_objective_vector: numpy.ndarray*, *rho: float = 1e-06*, ***kwargs*)

Bases: *GLIDEBase*

Implements the PROJECT method of preference elicitation and scalarization using the non-differentiable variant of GLIDE-II as proposed in: Ruiz, Francisco, Mariano Luque, and Kaisa Miettinen. "Improving the computational efficiency in a global formulation (GLIDE) for interactive multiobjective optimization." Annals of Operations Research 197.1 (2012): 47-70.

Parameters

- **utopian** (*np.ndarray, optional*) – The utopian point. Defaults to None.
- **nadir** (*np.ndarray, optional*) – The nadir point. Defaults to None.
- **rho** (*float, optional*) – The augmentation term for the scalarization function. Defaults to 1e-6.

property I_epsilon(*self*)

property I_alpha(*self*)

property mu(*self*)

property w(*self*)

property q(*self*)

property epsilon(*self*)

property s_epsilon(*self*)

property delta_epsilon(*self*)

scalarization.MOEADSF

Module Contents

Classes

MOEADSFBase	A base class for representing scalarizing functions for the MOEA/D algorithm.
Tchebycheff	Implements the Tchebycheff scalarizing function.
WeightedSum	Implements the Weighted sum scalarization function
PBI	Implements the PBI scalarization function

exception scalarization.MOEADSF.**MOEADSFError**

Bases: Exception

Raised when an error related to the MOEADSF classes is encountered.

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

class scalarization.MOEADSF.**MOEADSFBase**

Bases: abc.ABC

A base class for representing scalarizing functions for the MOEA/D algorithm. Instances of the implementations of this class should work as function.

abstract __call__(*self*, *objective_vector: numpy.ndarray*, *reference_vector: numpy.ndarray*, *ideal_vector: numpy.ndarray*, *nadir_vector: numpy.ndarray*) → Union[float, numpy.ndarray]

Evaluate the SF.

Parameters

- **objective_vector** (*np.ndarray*) – The objective vector to calculate the values.
- **reference_vector** (*np.ndarray*) – The reference vector to calculate the values.
- **ideal_vector** (*np.ndarray*) – The ideal objective vector.
- **nadir_vector** (*np.ndarray*) – The nadir objective vector.

Returns

Either a single SF value or a vector of values if objective is a 2D array.

Return type Union[float, np.ndarray]

class scalarization.MOEADSF.**Tchebycheff**

Bases: *MOEADSFBase*

Implements the Tchebycheff scalarizing function.

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_vector: numpy.ndarray*, *ideal_vector: numpy.ndarray*) → Union[float, numpy.ndarray]

Evaluate the Tchebycheff scalarizing function for minimization problems.

Parameters

- **objective_vector** (*np.ndarray*) – A vector representing a solution in the objective space.
- **reference_vector** (*np.ndarray*) – A reference vector representing the direction
- **ideal_vector** (*np.ndarray*) – The ideal objective vector

Raises ***MOEADSFError*** – The dimensions of the objective vector and reference_vector don't match.

Note: The shaped of objective_vector and reference_vector must match.

---

class scalarization.MOEADSF.**WeightedSum**

Bases: *MOEADSFBase*

Implements the Weighted sum scalarization function

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_vector: numpy.ndarray*) → Union[float, numpy.ndarray]

Evaluate the WeightedSum scalarizing function.

Parameters

- **objective_vector** (*np.ndarray*) – A vector representing a solution in the objective space.
- **reference_vector** (*np.ndarray*) – A reference vector representing the direction

Raises ***MOEADSFError*** – The dimensions of the objective vector and reference_vector don't match.

Note: The shaped of objective_vector and reference_vector must match. A reference point is not needed.

---

class scalarization.MOEADSF.**PBI**(*theta: float = 5*)

Bases: *MOEADSFBase*

Implements the PBI scalarization function

Parameters **theta** (*float*) – A penalty parameter used by the function

theta
A penalty parameter used by the function

Type float

__call__(*self*, *objective_vector: numpy.ndarray*, *reference_vector: numpy.ndarray*, *ideal_vector: numpy.ndarray*) → Union[float, numpy.ndarray]
Evaluate the PBI scalarizing function for minimization problems.

Parameters

- **objective_vector** (*np.ndarray*) – A vector representing a solution in the objective space.
- **reference_vector** (*np.ndarray*) – A reference vector representing the direction
- **ideal_vector** (*np.ndarray*) – The ideal objecive vector

Raises ***MOEADSFError*** – The dimensions of the objective vector and reference_vector don't match.

Note: The shaped of objective_vector and reference_vector must match. The reference point is not needed.

---

scalarization.Scalarizer

Module Contents

Classes

Scalarizer	Implements a class for scalarizing vector valued functions with a
DiscreteScalarizer	Implements a class to scalarize discrete vectors given a scalarizing function.

Attributes

vectors

class scalarization.Scalarizer.**Scalarizer**(*evaluator: Callable*, *scalarizer: Callable*, *evaluator_args: Dict = None*, *scalarizer_args: Dict = None*)
Implements a class for scalarizing vector valued functions with a given scalarization function.

Parameters

- **evaluator** (*Callable*) – A Callable object returning a numpy array.
- **scalarizer** (*Callable*) – A function which should accepts as its arguments the output

of evaluator and return a single value.

- **evaluator_args** (*Any, optional*) – Optional arguments to be passed to evaluator. Defaults to None.
- **scalarizer_args** (*Any, optional*) – Optional arguments to be passed to scalarizer. Defaults to None.

evaluate(*self*, *xs: numpy.ndarray*) → numpy.ndarray

Evaluates the scalarized function with the given arguments and returns a scalar value for each vector of variables given in a numpy array.

Parameters **xs** (*np.ndarray*) – A 2D numpy array containing vectors of variables on each of its rows.

Returns

A 1D numpy array with the values returned by the scalarizer for each row in xs.

Return type np.ndarray

__call__(*self*, *xs: numpy.ndarray*) → numpy.ndarray

Wrapper to the evaluate method.

class scalarization.Scalarizer.**DiscreteScalarizer**(*scalarizer: Callable*, *scalarizer_args: Dict = None*)

Implements a class to scalarize discrete vectors given a scalarizing function.

evaluate(*self*, *vectors: numpy.ndarray*) → numpy.ndarray

__call__(*self*, *vectors: numpy.ndarray*)

scalarization.Scalarizer.**vectors**

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

m

s

u

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W