

---

**desdeo\_tools**

*Release 1.0*

**Multiobjective Optimization Group**

**Jun 26, 2020**



## CONTENTS:

<b>1</b>	<b>desdeo-tools</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	Documentation . . . . .	1
<b>2</b>	<b>API Documentation</b>	<b>3</b>
2.1	desdeo_tools.interaction Package . . . . .	3
2.2	desdeo_tools.scalarization Package . . . . .	5
2.3	desdeo_tools.solver Package . . . . .	14
<b>3</b>	<b>Examples</b>	<b>19</b>
3.1	Example on using the scalarization methods for scalarizing and minimizing a problem which is based on discrete data . . . . .	19
3.2	Example on the usage of Scalarizer and ScalarSolver . . . . .	22
<b>4</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>
	<b>Index</b>	<b>31</b>



## DESDEO-TOOLS

Generic tools and design language used in the DESDEO framework. For example, this package contains classes for interacting with optimization methods implemented in `desdeo-mcmd` and `desdeo-emo`, and tools for solving for a representation of a Pareto optimal front for a multiobjective optimization.

### 1.1 Installation

Requires `poetry`. See `pyproject.toml` for Python package requirements.

To install and use the this package with `poetry` on a \*nix based system, issue the following commands:

1. `git clone https://github.com/industrial-optimization-group/desdeo-mcmd`
2. `cd desdeo-tools`
3. `poetry init`
4. `poetry install`

### 1.2 Documentation

Documentation and usage examples for this package can be found [here](#)



## API DOCUMENTATION

### 2.1 `desdeo_tools.interaction` Package

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

#### 2.1.1 Functions

---

`validate_ref_point_data_type(reference_point)`

---

`validate_ref_point_dimensions(...)`

---

`validate_ref_point_with_ideal(...)`

---

`validate_ref_point_with_ideal_and_nadir(...)`

---

`validate_with_ref_point_nadir(...)`

---

##### **`validate_ref_point_data_type`**

`desdeo_tools.interaction.validate_ref_point_data_type` (*reference\_point*)

##### **`validate_ref_point_dimensions`**

`desdeo_tools.interaction.validate_ref_point_dimensions` (*dimensions\_data*, *reference\_point*)

### validate\_ref\_point\_with\_ideal

`desdeo_tools.interaction.validate_ref_point_with_ideal` (*dimensions\_data*, *reference\_point*)

### validate\_ref\_point\_with\_ideal\_and\_nadir

`desdeo_tools.interaction.validate_ref_point_with_ideal_and_nadir` (*dimensions\_data*, *reference\_point*)

### validate\_with\_ref\_point\_nadir

`desdeo_tools.interaction.validate_with_ref_point_nadir` (*dimensions\_data*, *reference\_point*)

## 2.1.2 Classes

<code>PrintRequest</code> (message[, request_id])	Methods can use this request class to send out textual information to be displayed to the decision maker.
<code>SimplePlotRequest</code> (data, message[, ...])	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).
<code>ReferencePointPreference</code> (dimensions_data[, ...])	Methods can use this request class to ask the DM to provide their preferences in the form of a reference point.

### PrintRequest

**class** `desdeo_tools.interaction.PrintRequest` (*message*, *request\_id=None*)

Bases: `desdeo_tools.interaction.request.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.

### SimplePlotRequest

**class** `desdeo_tools.interaction.SimplePlotRequest` (*data*, *message*, *dimensions\_data=None*, *chart\_title=None*, *request\_id=None*)

Bases: `desdeo_tools.interaction.request.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.



## ReferencePointPreference

```
class desdeo_tools.interaction.ReferencePointPreference (dimensions_data, message=None, interaction_priority='required', preference_validator=None, request_id=None)
```

Bases: `desdeo_tools.interaction.request.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.

### Attributes Summary

---

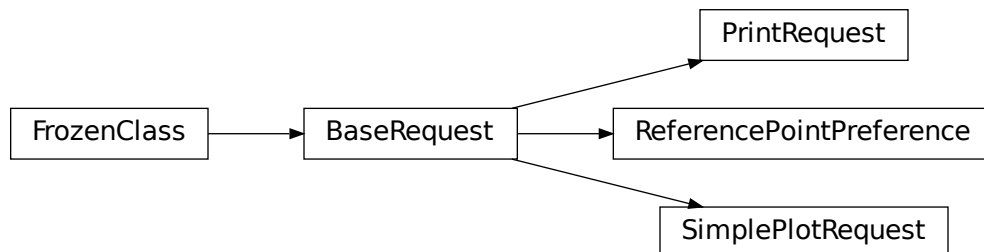
*response*

---

### Attributes Documentation

**response**

## 2.1.3 Class Inheritance Diagram



## 2.2 desdeo\_tools.scalarization Package

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

## 2.2.1 Classes

<code>AugmentedGuessASF(nadir, ideal, index_to_exclude)</code>	Implementation of the augmented GUESS related ASF as presented in <i>Miettinen 2006</i> equation (3.4)
<code>MaxOfTwoASF(nadir, ideal, lt_inds, lte_inds)</code>	Implements the ASF as defined in eq.
<code>PointMethodASF(nadir, ideal[, rho, rho_sum])</code>	Implementation of the reference point based ASF as presented in <i>Miettinen 2006</i> equation (3.3)
<code>ReferencePointASF(preferential_factors, ...)</code>	Uses a reference point $q$ and preferential factors to scalarize a MOO problem.
<code>SimpleASF(weights)</code>	Implements a simple order-representing ASF.
<code>StomASF(ideal[, rho, rho_sum])</code>	Implementation of the satisfying trade-off method (STOM) as presented in <i>Miettinen 2006</i> equation (3.2)
<code>DiscreteScalarizer(scalarizer[, scalarizer_args])</code>	Implements a class to scalarize discrete vectors given a scalarizing function.
<code>Scalarizer(evaluator, scalarizer[, ...])</code>	Implements a class for scalarizing vector valued functions with a given scalarization function.

### AugmentedGuessASF

**class** `desdeo_tools.scalarization.AugmentedGuessASF` (*nadir*, *ideal*, *index\_to\_exclude*, *rho=1e-06*, *rho\_sum=1e-06*)

Bases: `desdeo_tools.scalarization.ASF.ASFBase`

Implementation of the augmented GUESS related ASF as presented in *Miettinen 2006* equation (3.4)

#### 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
- **excluded in calculating the first term of the ASF.** (*be*) –
- **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.** –

#### Methods Summary

<code>__call__(objective_vectors, reference_point)</code>	Evaluate the ASF.
-----------------------------------------------------------	-------------------

## Methods Documentation

`__call__` (*objective\_vectors*, *reference\_point*)  
Evaluate the ASF.

### Parameters

- **objective\_vectors** (*np.ndarray*) – The objective vectors to calculate
- **values.** (*the*) –
- **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.

---

## MaxOfTwoASF

**class** `desdeo_tools.scalarization.MaxOfTwoASF` (*nadir*, *ideal*, *lt\_inds*, *lte\_inds*, *rho=1e-06*,  
*rho\_sum=1e-06*)

Bases: `desdeo_tools.scalarization.ASF.ASFBase`

Implements the ASF as defined in eq. 3.1 [Miettinen 2006](#)

### 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
- **until some value is reached.** (*reduced*) –
- **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

**Type** List[int]

decreased.

**lte\_inds**

Indices of the objectives categorized to be

**Type** List[int]

reduced until some value is reached.

**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

**Type** float

term.

## Methods Summary

---

<code>__call__</code> (objective_vector, reference_point)	Evaluate the ASF.
-----------------------------------------------------------	-------------------

---

## Methods Documentation

`__call__` (objective\_vector, reference\_point)

Evaluate the ASF.

### Parameters

- **objective\_vectors** (*np.ndarray*) – The objective vectors to calculate
- **values.** (*the*) –
- **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.

---

## PointMethodASF

**class** desdeo\_tools.scalarization.**PointMethodASF** (*nadir, ideal, rho=1e-06, rho\_sum=1e-06*)

Bases: desdeo\_tools.scalarization.ASF.ASFBase

Implementation of the reference point based ASF as presented in *Miettinen 2006* equation (3.3)

### 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.** –

---

**Note:** Lack of better name...

---

## Methods Summary

---

<code>__call__</code> ( <i>objective_vectors</i> , <i>reference_point</i> )	Evaluate the ASF.
-----------------------------------------------------------------------------	-------------------

---

## Methods Documentation

`__call__` (*objective\_vectors*, *reference\_point*)

Evaluate the ASF.

### Parameters

- **objective\_vectors** (*np.ndarray*) – The objective vectors to calculate
- **values.** (*the*) –
- **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.

---

## ReferencePointASF

**class** `desdeo_tools.scalarization.ReferencePointASF` (*preferential\_factors*, *nadir*,  
*utopian\_point*, *rho=0.1*)

Bases: `desdeo_tools.scalarization.ASF.ASFBase`

Uses a reference point *q* and preferential factors to scalarize a MOO problem. Defined in [Miettinen 2010](#) equation (2).

### 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
- **Defaults to 0.1.** (*ASF.*) –

**preferential\_factors**

The preferential factors.

**Type** np.ndarray

**nadir**

The nadir point of the MOO problem to be

**Type** np.ndarray

**scalarized.**

**utopian\_point**

The utopian point of the MOO problem to be

**Type** np.ndarray

**scalarized.**

**rho**

A small number to be used to scale the sm factor in the

**Type** float

**ASF. Defaults to 0.1.**

## Methods Summary

---

<code>__call__(objective_vector, reference_point)</code>	Evaluate the ASF.
----------------------------------------------------------	-------------------

---

## Methods Documentation

`__call__(objective_vector, reference_point)`

Evaluate the ASF.

**Parameters**

- **objective\_vectors** (*np.ndarray*) – The objective vectors to calculate
- **values.** (*the*) –
- **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.

---

## SimpleASF

**class** `desdeo_tools.scalarization.SimpleASF` (*weights*)

Bases: `desdeo_tools.scalarization.ASF.ASFBase`

Implements a simple order-representing ASF.

### Parameters

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

### weights

A weight vector that holds weights. It's

Type `np.ndarray`

**length should match the number of objectives in the underlying MOO problem the achievement problem aims to solve.**

## Methods Summary

---

<code>__call__</code> ( <i>objective_vector</i> , <i>reference_point</i> )	Evaluate the simple order-representing ASF.
----------------------------------------------------------------------------	---------------------------------------------

---

## Methods Documentation

`__call__` (*objective\_vector*, *reference\_point*)

Evaluate the simple order-representing ASF.

### Parameters

- **objective\_vector** (*np.ndarray*) – A vector representing a solution in
- **solution space.** (*the*) –
- **reference\_point** (*np.ndarray*) – A vector representing a reference
- **in the solution space.** (*point*) –

### Raises

- **ASFError** – The dimensions of the objective vector and reference
- **point don't match.** –

---

**Note:** The shaped of *objective\_vector* and *reference\_point* must match.

---

**Return type** `Union[float, ndarray]`

## StomASF

**class** `desdeo_tools.scalarization.StomASF` (*ideal*, *rho*=1e-06, *rho\_sum*=1e-06)

Bases: `desdeo_tools.scalarization.ASF.ASFBase`

Implementation of the satisfying trade-off method (STOM) as presented in *Miettinen 2006* equation (3.2)

### 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

**Type** `float`

### **term.**

## Methods Summary

---

<code>__call__</code> ( <i>objective_vectors</i> , <i>reference_point</i> )	Evaluate the ASF.
-----------------------------------------------------------------------------	-------------------

---

## Methods Documentation

`__call__` (*objective\_vectors*, *reference\_point*)

Evaluate the ASF.

### Parameters

- **objective\_vectors** (*np.ndarray*) – The objective vectors to calculate
- **values.** (*the*) –
- **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.

---



## DiscreteScalarizer

**class** desdeo\_tools.scalarization.**DiscreteScalarizer** (*scalarizer*, *scalarizer\_args=None*)

Bases: object

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

### Methods Summary

<code>__call__(vectors)</code>	Call self as a function.
<code>evaluate(vectors)</code>	<b>rtype</b> ndarray

### Methods Documentation

`__call__(vectors)`

Call self as a function.

**evaluate** (*vectors*)

**Return type** ndarray

## Scalarizer

**class** desdeo\_tools.scalarization.**Scalarizer** (*evaluator*, *scalarizer*, *evaluator\_args=None*, *scalarizer\_args=None*)

Bases: object

Implements a class for scalarizing vector valued functions with a given scalarization function.

### Methods Summary

<code>__call__(xs)</code>	Wrapper to the evaluate method.
<code>evaluate(xs)</code>	Evaluates the scalarized function with the given arguments and returns a scalar value for each vector of variables given in a numpy array.

### Methods Documentation

`__call__(xs)`

Wrapper to the evaluate method.

**Return type** ndarray

**evaluate** (*xs*)

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

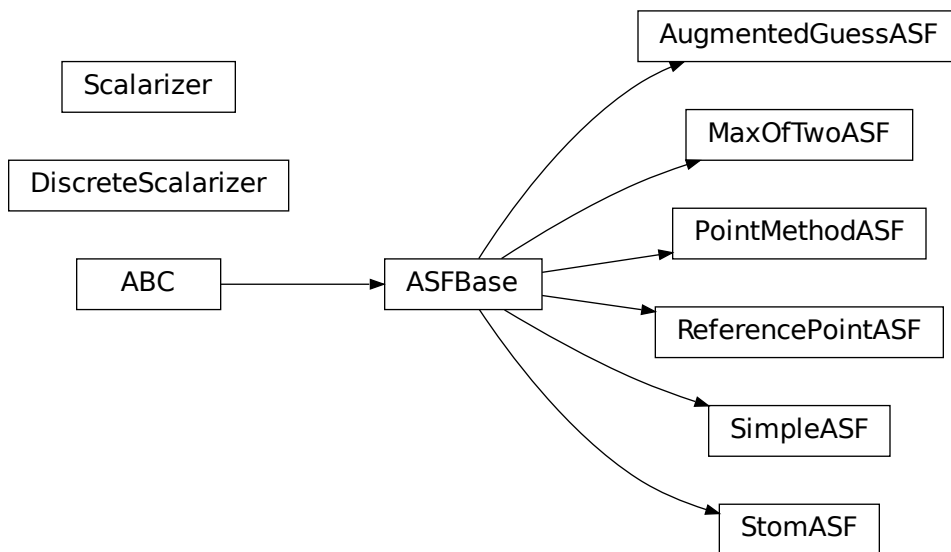
**Parameters**

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

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

**Return type** np.ndarray

## 2.2.2 Class Inheritance Diagram



## 2.3 desdeo\_tools.solver Package

This module implements methods for solving scalar valued functions.

### 2.3.1 Classes

<i>DiscreteMinimizer</i> (discrete_scalarizer[, ...])	Implements a class for finding the minimum value of a discrete of scalarized vectors.
<i>ScalarMethod</i> (method[, method_args, use_scipy])	A class the define and implement methods for minimizing scalar valued functions.
<i>ScalarMinimizer</i> (scalarizer, bounds[, ...])	Implements a class for minimizing scalar valued functions with bounds set for the variables, and constraints.
<i>ScalarSolverException</i>	

## DiscreteMinimizer

**class** desdeo\_tools.solver.**DiscreteMinimizer** (*discrete\_scalarizer*, *constraint\_evaluator=None*) *con-*

Bases: object

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

### Methods Summary

---

<i>minimize</i> (vectors)	Find the index of the element in vectors which minimizes the scalar value returned by the scalarizer.
---------------------------	-------------------------------------------------------------------------------------------------------

---

### Methods Documentation

**minimize** (*vectors*)

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
- **should be computed for.** (*value*) –

#### Raises

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

**Returns** The index of the vector in vectors which minimizes the value computed with the given scalarizer.

**Return type** int

## ScalarMethod

**class** desdeo\_tools.solver.**ScalarMethod** (*method*, *method\_args=None*, *use\_scipy=False*)

Bases: object

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

### Methods Summary

---

<i>__call__</i> (obj_fun, x0, bounds, ...)	Minimizes a scalar valued function.
--------------------------------------------	-------------------------------------

---

## Methods Documentation

`__call__` (*obj\_fun*, *x0*, *bounds*, *constraint\_evaluator*)

Minimizes a scalar valued function.

### Parameters

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

## ScalarMinimizer

**class** `desdeo_tools.solver.ScalarMinimizer` (*scalarizer*, *bounds*, *constraint\_evaluator=None*, *method=None*)

Bases: `object`

Implements a class for minimizing scalar valued functions with bounds set for the variables, and constraints.

## Methods Summary

<code>get_presets()</code>	Return the list of preset minimizers available.
<code>minimize(x0)</code>	Minimizes the scalarizer given an initial guess x0.

## Methods Documentation

### `get_presets()`

Return the list of preset minimizers available.

### `minimize(x0)`

Minimizes the scalarizer given an initial guess x0.

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

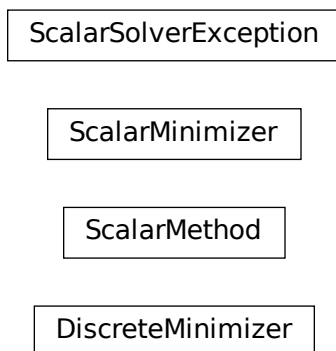
**Returns** A dictionary with at least the following entries: ‘x’ indicating the optimal variables found, ‘fun’ the optimal value of the optimized function, and ‘success’ a boolean indicating whether the optimization was conducted successfully.

**Return type** Dict

## ScalarSolverException

**exception** `desdeo_tools.solver.ScalarSolverException`

## 2.3.2 Class Inheritance Diagram





## EXAMPLES

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

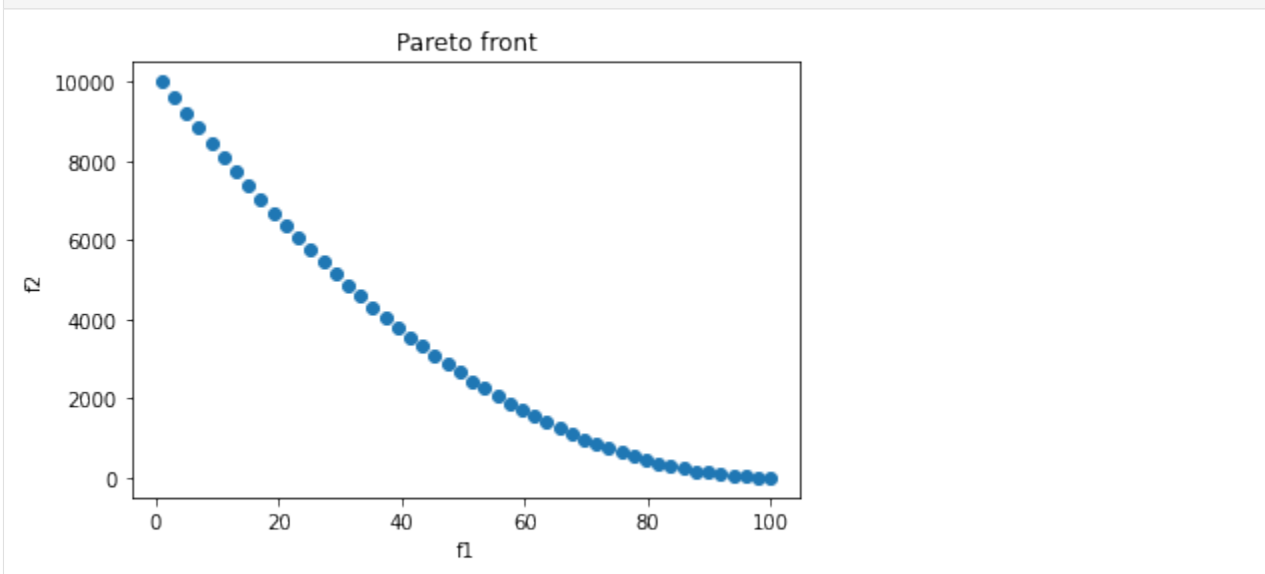
In this example, we will go through the following two topics: 1. How to define a scalarization method for scalarizing discrete data representing a multiobjective optimization problem; 2. How to find a solution to the scalarized problem.

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)
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:

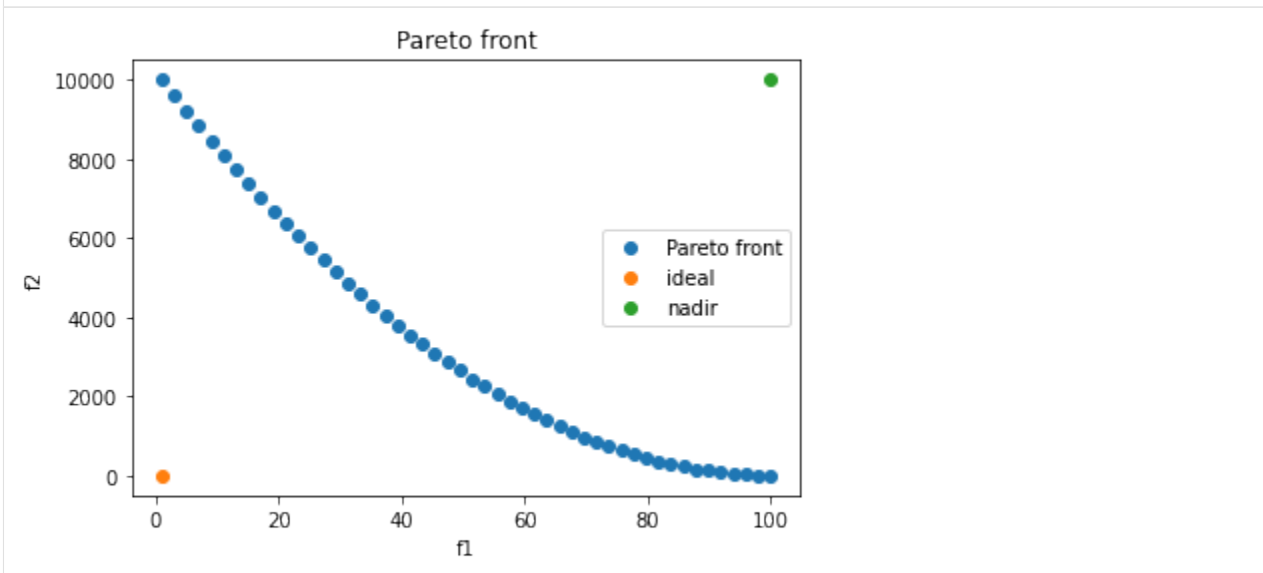
```
[2]: pfront = np.stack((f1, f2)).T

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)
```

(continues on next page)



(continued from previous page)

```
dminimizer = DiscreteMinimizer(dscalarizer)

solution_i = dminimizer.minimize(pfront)

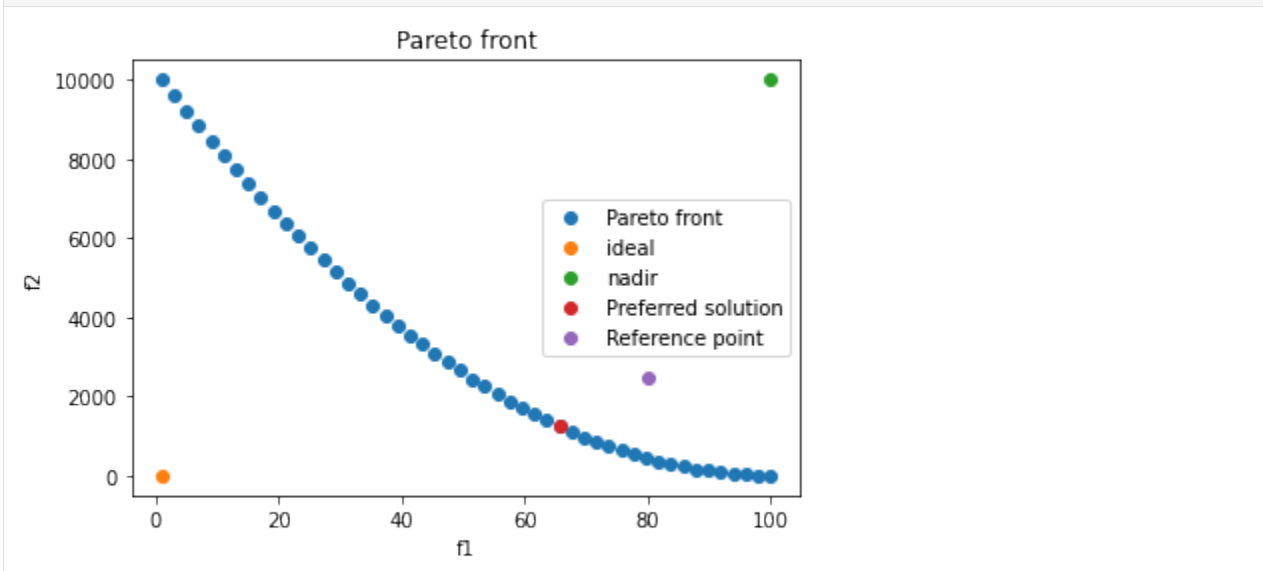
print(f"Index of the objective vector minimizing the ASF problem: {solution_i}")

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")
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  $f_1$  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 or 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)
```

(continues on next page)

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

(continued from previous page)

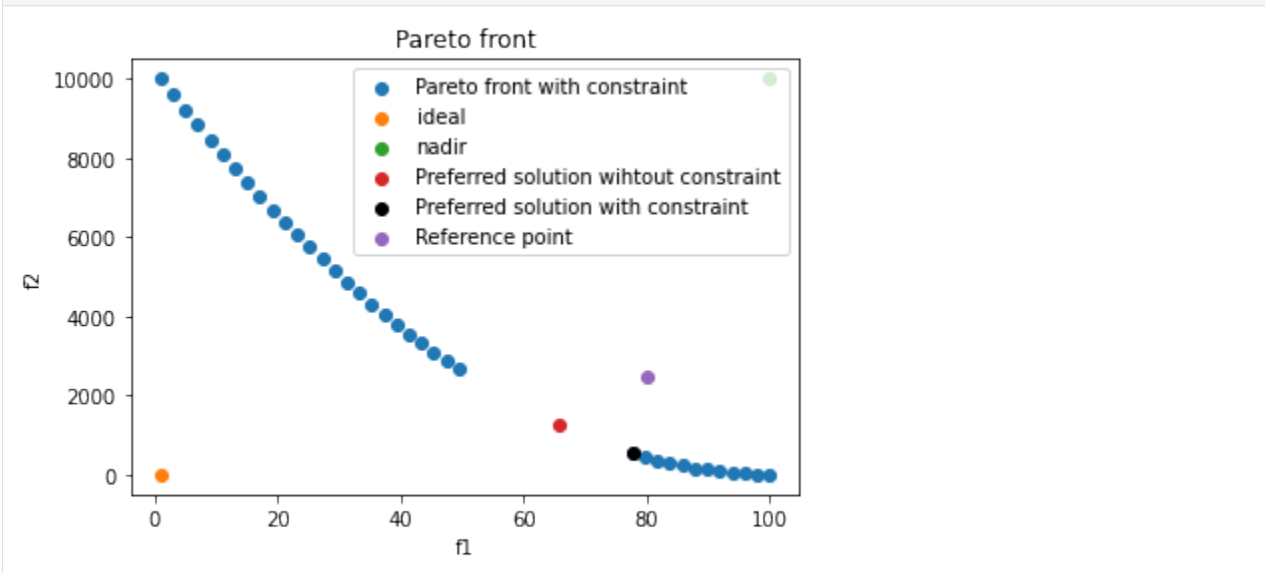
```

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()

```



## 3.2 Example on the usage of Scalarizer and ScalarSolver

This notebook will go through a simple example on how to scalarize a vector valued function 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 equal 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 area  $A$  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 formulated as a multi-objective optimization problem with two objectives and two constraints. Formally

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

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*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])))

# 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 `scalarized_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)

# 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"\nOptimal\ cake specs: radius: {x_optimal[0]}cm, height: {x_optimal[1]}cm.")
print(f"\nOptimal\ 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 individually 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"])
```

(continues on next page)

(continued from previous page)

```

# 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

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}.")

```

(continues on next page)

(continued from previous page)

```
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!

```
[ ]:
```

## INDICES AND TABLES

- genindex
- modindex
- search





## PYTHON MODULE INDEX

### d

`desdeo_tools.interaction`, 3  
`desdeo_tools.scalarization`, 5  
`desdeo_tools.solver`, 14



## Symbols

`__call__()` (*desdeo\_tools.scalarization.AugmentedGuessASF* method), 7  
`__call__()` (*desdeo\_tools.scalarization.DiscreteScalarizer* method), 13  
`__call__()` (*desdeo\_tools.scalarization.MaxOfTwoASF* method), 8  
`__call__()` (*desdeo\_tools.scalarization.PointMethodASF* method), 9  
`__call__()` (*desdeo\_tools.scalarization.ReferencePointASF* method), 10  
`__call__()` (*desdeo\_tools.scalarization.Scalarizer* method), 13  
`__call__()` (*desdeo\_tools.scalarization.SimpleASF* method), 11  
`__call__()` (*desdeo\_tools.scalarization.StomASF* method), 12  
`__call__()` (*desdeo\_tools.solver.ScalarMethod* method), 16

## A

*AugmentedGuessASF* (class in *desdeo\_tools.scalarization*), 6

## D

*desdeo\_tools.interaction* module, 3  
*desdeo\_tools.scalarization* module, 5  
*desdeo\_tools.solver* module, 14  
*DiscreteMinimizer* (class in *desdeo\_tools.solver*), 15  
*DiscreteScalarizer* (class in *desdeo\_tools.scalarization*), 13

## E

`evaluate()` (*desdeo\_tools.scalarization.DiscreteScalarizer* method), 13  
`evaluate()` (*desdeo\_tools.scalarization.Scalarizer* method), 13

## G

`ASF_presets()` (*desdeo\_tools.solver.ScalarMinimizer* method), 17

## I

*ideal* (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 7  
*ideal* (*desdeo\_tools.scalarization.StomASF* attribute), 12

## L

*lt\_inds* (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 7  
*lte\_inds* (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 8

## M

*MaxOfTwoASF* (class in *desdeo\_tools.scalarization*), 7  
`minimize()` (*desdeo\_tools.solver.DiscreteMinimizer* method), 15  
`minimize()` (*desdeo\_tools.solver.ScalarMinimizer* method), 17  
 module  
   *desdeo\_tools.interaction*, 3  
   *desdeo\_tools.scalarization*, 5  
   *desdeo\_tools.solver*, 14

## N

*nadir* (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 7  
*nadir* (*desdeo\_tools.scalarization.ReferencePointASF* attribute), 10

## P

*PointMethodASF* (class in *desdeo\_tools.scalarization*), 8  
*preferential\_factors* (*desdeo\_tools.scalarization.ReferencePointASF* attribute), 10  
*PrintRequest* (class in *desdeo\_tools.interaction*), 4

## R

ReferencePointASF (class in *desdeo\_tools.scalarization*), 9  
ReferencePointPreference (class in *desdeo\_tools.interaction*), 5  
response (*desdeo\_tools.interaction.ReferencePointPreference* attribute), 5  
rho (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 8  
rho (*desdeo\_tools.scalarization.ReferencePointASF* attribute), 10  
rho (*desdeo\_tools.scalarization.StomASF* attribute), 12  
rho\_sum (*desdeo\_tools.scalarization.MaxOfTwoASF* attribute), 8  
rho\_sum (*desdeo\_tools.scalarization.StomASF* attribute), 12

## S

Scalarizer (class in *desdeo\_tools.scalarization*), 13  
ScalarMethod (class in *desdeo\_tools.solver*), 15  
ScalarMinimizer (class in *desdeo\_tools.solver*), 16  
ScalarSolverException, 17  
SimpleASF (class in *desdeo\_tools.scalarization*), 11  
SimplePlotRequest (class in *desdeo\_tools.interaction*), 4  
StomASF (class in *desdeo\_tools.scalarization*), 12

## U

utopian\_point (*desdeo\_tools.scalarization.ReferencePointASF* attribute), 10

## V

validate\_ref\_point\_data\_type() (in module *desdeo\_tools.interaction*), 3  
validate\_ref\_point\_dimensions() (in module *desdeo\_tools.interaction*), 3  
validate\_ref\_point\_with\_ideal() (in module *desdeo\_tools.interaction*), 4  
validate\_ref\_point\_with\_ideal\_and\_nadir() (in module *desdeo\_tools.interaction*), 4  
validate\_with\_ref\_point\_nadir() (in module *desdeo\_tools.interaction*), 4

## W

weights (*desdeo\_tools.scalarization.SimpleASF* attribute), 11