DESDEO

Release 1.1.2

Jul 18, 2021

Contents

Decision Support for computationally Demanding Optimization problems

DESDEO is a free and open-source Python-based framework for developing and experimenting with interactive multiobjective optimization. It contains implementations of some interactive methods and modules that can be utilized to implement further methods.

We welcome you to utilize DESDEO and develop it further with us.

For news and other information related to DESDEO, see the official website.

CHAPTER 1

Packages

1.1 Introduction

1.1.1 DESDEO

Decision Support for computationally Demanding Optimization problems

DESDEO is a free and open-source Python-based framework for developing and experimenting with interactive multiobjective optimization.

DESDEO contains implementations of some interactive methods and modules that can be utilized to implement further methods.

We welcome you to utilize DESDEO and develop it further with us.

DESDEO brings interactive methods closer to researchers and practitioners worldwide by providing them with implementations of interactive methods.

DESDEO is part of DEMO (Decision analytics utilizing causal models and multiobjective optimization which is the thematic research area of the University of Jyväskylä.

1.1.2 Mission

The mission of DESDEO is to increase awareness of the benefits of interactive methods make interactive methods more easily available and applicable. Thanks to the open architecture, interactive methods are easier to be utilized and further developed. The framework consists of reusable components that can be utilized for implementing new methods or modifying the existing methods. The framework is released under a permissive open source license.

1.1.3 Multiobjective optimization

In multiobjective optimization, several conflicting objective functions are to be optimized simultaneously. Because of the conflicting nature of the objectives, it is not possible to obtain individual optima of the objectives simultaneously but one must trade-off between the objectives.

1.1.4 Interactive methods in multiobjective optimization

Interactive methods are iterative by nature where a decision maker (who has substance knowledge) can direct the solution process with one's preference information to find the most preferred balance between the objectives. In interactive methods, the amount of information to be considered at a time is limited and, thus, the cognitive load set on the decision maker is not too demanding. Furthermore, the decision maker learns about the interdependencies among the objectives and also the feasibility of one's preferences.

1.1.5 The Research Projects Behind DESDEO

About the Multiobjective Optimization Group

The Multiobjective Optimization Group developes theory, methodology and open-source computer implementations for solving real-world decision-making problems. Most of the research concentrates on multiobjective optimization (MO) in which multiple conflicting objectives are optimized simultaneously and a decision maker (DM) is supported in finding a preferred compromise.

About the DESDEO research project

DESDEO contains implementations of some interactive methods and modules that can be utilized to implement further methods. DESDEO brings interactive methods closer to researchers and practitioners world-wide, by providing them with implementations of interactive methods.

Interactive methods are useful tools for decision support in finding the most preferred balance among conflicting objectives. They support the decision maker in gaining insight in the trade-offs among the conflicting objectives. The decision maker can also conveniently learn about the feasibility of one's preferences and update them, if needed.

DESDEO is part of DEMO (Decision analytics utilizing causal models and multiobjective optimization) which is the thematic research area of the University of Jyväskylä (jyu.fi/demo).

We welcome you to utilize DESDEO and develop it further with us.

About DAEMON

The mission of DAEMON is method and software development for making better data-driven decisions. The project considers data and decision problems from selected fields as cases to inspire the research and demonstrate the added value.

In DAEMON, we support optimizing conflicting objectives simultaneously by applying interactive multiobjective optimization methods, where a decision maker (DM) incorporates one's domain expertise and preferences in the solution process. Overall, we model and formulate optimization problems based on data, so that DMs can identify effective strategies to better understand trade-offs and balance between conflicting objectives. In this, we incorporate machine learning tools, visualize trade-offs to DMs and consider uncertainties affecting the decisions.

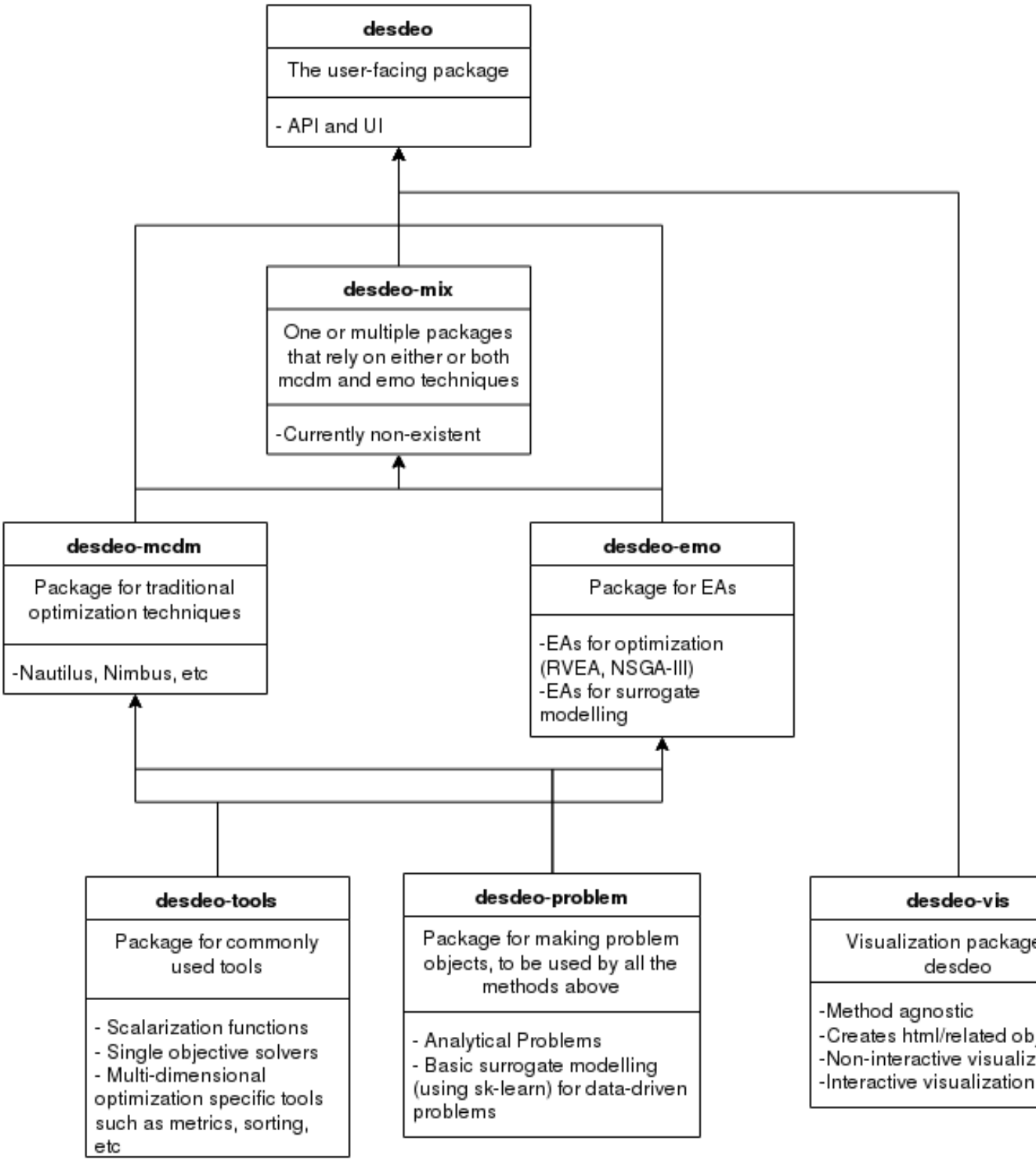
1.1.6 Publications Related to DESDEO

See publications.

1.2 Software

1.2.1 Modular Structure

The DESDEO framework consists of various modules defined as Python software packages. The modular structure can be seen in the image below, and a short description of the different packages follows. The descriptions contain links to the invdividual documentation pages for each of the package.


desdeo
The user-facing package
- API and UI
desdeo-mix
One or multiple packages that rely on either or both mcdm and emo techniques
-Currently non-existent
desdeo-mcdm
Package for traditional optimization techniques
-Nautilus, Nimbus, etc
desdeo-emo
Package for EAs
-EAs for optimization (RVEA, NSGA-III)
-EAs for surrogate modelling
desdeo-tools
Package for commonly used tools
- Scalarization functions
- Single objective solvers
- Multi-dimensional optimization specific tools such as metrics, sorting, etc
desdeo-problem
Package for making problem objects, to be used by all the methods above
- Analytical Problems
- Basic surrogate modelling (using sk-learn) for data-driven problems
desdeo-vis
Visualization package
desdeo
-Method agnostic
-Creates html/related ob
-Non-interactive visualiz
-Interactive visualization

desdeo-problem

The desdeo_problem package contains tools and classes for defining and modelling multiobjective optimization problems. The defined problem classes can be used in the other packages in the DESDEO framework, such as desdeo_mcdm and *desdeo-emo*.

desdeo-tools

The desdeo_tools package contains tools to facilitate different tasks in the other packages in the DESDEO framework. These tools include, for example, scalarization routines and various solvers.

desdeo-emo

The desdeo_emo package contains evolutionary algorithms for solving multiobjective optimization problems. These algorithms include, for example, interactive RVEA.

desdeo-mcdm

The desdeo_mcdm package contains traditional methods for performing interactive multiobjective optimization. These methods include, but are not limited to, Synchronous NIMBUS and E-NAUTILUS, for example.

desdeo-vis

(*coming soon*) The desdeo_vis package contains tools for building visualizations.

desdeo-mix

(*coming soon*) The desdeo_mix package contains routines and algorithms which make use of the other packages in the DESDEO framework, and warrant their own implementation to be included as part of the framework.

1.2.2 Installation

To install a single package, see that package's documentation. To install the whole DESDEO framework, follow one of the alternatives given below according to the operating system on the machine you plan to install DESDEO on.

Linux and Windows

The recommended way of installing the DESDEO framework is to use pip by invoking the command:

```
pip install desdeo
```

For developing, usage of the poetry dependency management tool is recommended. However, no framework code should be contributed to the main DESDEO package. Instead, contributions should go to the relevant subpackage, which are listed here. See the documentation for the individual for further details. See *Modular Structure* for links to the individual documentations.

OSX

Instruction coming eventually.

1.3 Guides and Examples

1.3.1 Guides

A quick overview of some of DESDEO's main capabilities

Note: This notebook contains the code shown in the three use cases considered in the article titled *DESDEO: an open framework for interactive multiobjective optimization*. Cells containing code found in the article will be marked accordingly (by a comment on the first line). The code shown may vary from the one in the article due to new additions and/or changes in the DESDEO framework. This is to ensure that the code shown can be run and experimented with in the future as well.

This overview should give the novice user interested in DESDEO a brief overview of how the framework may be utilized. We will consider a multiobjective optimization problem with five objectives and two variables. The problem will be treated as an analytical problem, a data-based problem, and a computationally expensive problem. We will show how the problem can be solved utilizing MCDM and EMO methods both separately and in tandem (hybridizing them). **Note:** Link to article with additional information to be added later.

The probelm

The considered problem - the river pollution problem - in the use cases is presented here. Its analytical definiton can be stated as:

$$
\begin{aligned}
\min \quad & f_1(\mathbf{x}) = && -4.07 - 2.27x_1 \\
\min \quad & f_2(\mathbf{x}) = && -2.60 - 0.03x_1 - 0.02x_2 \\
& && -\tfrac{0.01}{1.39-x_1^2} - \tfrac{0.30}{1.39-x_2^2} \\
\min \quad & f_3(\mathbf{x}) = && -8.21 + \tfrac{0.71}{1.09-x_1^2} \\
\min \quad & f_4(\mathbf{x}) = && -0.96 + \tfrac{0.96}{1.09-x_2^2} \\
\min \quad & f_5(\mathbf{x}) = && \max\{|x_1 - 0.65|, |x_2 - 0.65|\} \\
\\
\text{s.t.} \quad & && 0.3 \leq x_1, x_2 \leq 1.0,
\end{aligned}
\tag{1.1}
$$

where each objective is to be minimized subject to box-constraints imposed on the variables.

Use case 1: solving problems with analytical formulations

If we wish to solve a problem with an analytical formulation, such as the river pollution problem, we can proceed as follows. First, we will define the problem in DESDEO. We import some required modules from *desdeo-problem* for this purpose. *desdeo-problem* contains modules that are used to define multiobjective optimization problems. We import the following modules:

[1]:
```
# Source code 1
import numpy as np

from desdeo_problem.problem import MOProblem
```

(continues on next page)

(continued from previous page)

```
from desdeo_problem.problem import Variable
from desdeo_problem.problem import ScalarObjective
```

Next ,we define the objective functions of the river pollution problem first as Python functions and then wrap them inside instances of the `ScalarObjective` class. The *scalar* in the name implies that the objective is itself a scalar-valued function (i.e., $\mathbb{R}^n \to \mathbb{R}$ for some positive n). The resulting objects are then stored in a list. In code:

[2]:
```
# Source code 2
def f_1(x: np.ndarray) -> np.ndarray:
    x = np.atleast_2d(x)  # This step is to guarantee that the function works when␣
↪called with a single decision variable vector as well.
    return -4.07 - 2.27*x[:, 0]

def f_2(x: np.ndarray) -> np.ndarray:
    x = np.atleast_2d(x)
    return -2.60 - 0.03*x[:, 0] - 0.02*x[:, 1] - 0.01 / (1.39 - x[:, 0]**2) - 0.30 /␣
↪(1.39 + x[:, 1]**2)

def f_3(x: np.ndarray) -> np.ndarray:
    x = np.atleast_2d(x)
    return -8.21 + 0.71 / (1.09 - x[:, 0]**2)

def f_4(x: np.ndarray) -> np.ndarray:
    x = np.atleast_2d(x)
    return -0.96 - 0.96 / (1.09 - x[:, 1]**2)

def f_5(x: np.ndarray) -> np.ndarray:
    return np.max([np.abs(x[:, 0] - 0.65), np.abs(x[:, 1] - 0.65)], axis=0)

objective_1 = ScalarObjective(name="f_1", evaluator=f_1)
objective_2 = ScalarObjective(name="f_2", evaluator=f_2)
objective_3 = ScalarObjective(name="f_3", evaluator=f_3)
objective_4 = ScalarObjective(name="f_4", evaluator=f_4)
objective_5 = ScalarObjective(name="f_5", evaluator=f_5)

objectives = [objective_1, objective_2, objective_3, objective_4, objective_5]
```

When defining objectives, it is expected that they may be called with either a single set of decision variables or a set of such sets. I.e.,:

[3]:
```
# single set of decision variables
x_single = np.array([0.8, 0.5])

# set of sets
x_multi = np.array([[0.8, 0.5], [0.31, 0.88], [0.34, 0.33]])

# Both sets work when evaluated with the defined objective functions.
# Notice how we get one value for each provided set of decision variables.
print(f"f_1({x_single}) = ", f_1(x_single))
print(f"f_1({x_multi}) = ", f_1(x_multi))
```

```
f_1([0.8 0.5]) =  [-5.886]
f_1([[0.8  0.5 ]
 [0.31 0.88]
 [0.34 0.33]]) =  [-5.886  -4.7737 -4.8418]
```

The variables of the problem are defined in a very similar fashion compared to the objectives:

[4]:
```
# Source code 3
x_1 = Variable("x_1", 0.5, 0.3, 1.0)
x_2 = Variable("x_2", 0.5, 0.3, 1.0)

variables = [x_1, x_2]
```

Notice the arguments given to the initializer of 'Variable': the last and second-to-last define the variabels upper and lower bounds, respectively, while the second argument defines a variable's initial value and current value, which is sometimes useful information. The upper and lower bounds are optional; if not provided, appropiate infinims will be assumed for the bounds. The documentation should confirm these claims regarding the arguments:

[5]:
```
help(Variable)
```

```
Help on class Variable in module desdeo_problem.problem.Variable:

class Variable(builtins.object)
 |  Variable(name: str, initial_value: float, lower_bound: float = -inf, upper_bound:␣
↪float = inf) -> None
 |
 |  Simple variable with a name, initial value and bounds.
 |
 |  Args:
 |      name (str): Name of the variable
 |      initial_value (float): The initial value of the variable.
 |      lower_bound (float, optional): Lower bound of the variable. Defaults
 |          to negative infinity.
 |      upper_bound (float, optional): Upper bound of the variable. Defaults
 |          to positive infinity.
 |
 |  Attributes:
 |      name (str): Name of the variable.
 |      initial_value (float): Initial value of the variable.
 |      lower_bound (float): Lower bound of the variable.
 |      upper_bound (float): Upper bound of the variable.
 |      current_value (float): The current value the variable holds.
 |
 |  Raises:
 |      VariableError: Bounds are incorrect.
 |
 |  Methods defined here:
 |
 |  __init__(self, name: str, initial_value: float, lower_bound: float = -inf, upper_
↪bound: float = inf) -> None
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  get_bounds(self) -> Tuple[float, float]
 |      Return the bounds of the variables as a tuple.
 |
 |      Returns:
 |          tuple(float, float): A tuple consisting of (lower_bound,
 |              upper_bound)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
```

(continues on next page)

(continued from previous page)

```
|  __weakref__
|      list of weak references to the object (if defined)
|
|  current_value
|
|  initial_value
|
|  name
```

Most of the classes, their methods, and functions should be equipped with docstrings so that the `help` utility should give some insight on how to use them.

We are now ready to define the multiobjective optimization problem itself. Most of the work has already been done in defining the objectives and variables:

[6]:
```
# Source code 4
mo_problem = MOProblem(variables=variables, objectives=objectives)
```

At this point, we could start solving the problem with various methods found in DESDEO or we can use the problem in various other ways as well to gain additional insight. Since we will be using Synchronous NIMBUS to solve the problem in this use case, we need knowledge of the upper and lower bounds of the possible objective values contained in the set of feasible solutions. For this, we can compute the ideal and nadir points of the problem. The simplest way to compute the ideal and approximate the nadir is to use a pay-off table. A method based on the pay-off table to compute the ideal and (approximation of) the nadir point is found in the *utilities* module of the *desdeo-mcdm* package. The ideal and nadir points are then stored as attributes of the `mo_problem` object to readily access them later on. We proceed as follows:

[7]:
```
# Source code 5
from desdeo_mcdm.utilities import payoff_table_method

ideal, nadir = payoff_table_method(mo_problem)

mo_problem.ideal = ideal
mo_problem.nadir = nadir
```

The ideal and nadir points should give us an idea of the ranges of the objectives. Indeed, this is the case:

[8]:
```
print(f"Ideal point: {mo_problem.ideal}")
print(f"Nadir point: {mo_problem.nadir}")
print(f"Ideal strictly better than nadir?: {np.all(mo_problem.ideal <= mo_problem.
↪nadir)}")
```

```
Ideal point: [ -6.33999773  -2.8643435   -7.49999957 -11.62604407   0.        ]
Nadir point: [-4.75100227 -2.76563265 -0.32128642 -1.92000058  0.349999  ]
Ideal strictly better than nadir?: True
```

We can now actually begin solving the river pollution problem using the Synchronous NIMBUS method. As we have already defined an instance of `MOProblem`, initializing and starting the method is straight forward:

[9]:
```
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

# Source code 6
from desdeo_mcdm.interactive.NIMBUS import NIMBUS
```

(continues on next page)

(continued from previous page)

```
nimbus = NIMBUS(mo_problem)

classification_request, _ = nimbus.start()
```

The `start` method returns two *requests* of which the first one is of interest here. Its `contents` attribute is a `dict` the keys of which we can inspect:

```
[10]: classification_request.content.keys()
```

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

In the `content` of each *request*, at least the `message` entry exists. This entry contains information on how to proceed:

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

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

We can interact with the method by defining the *response*, an attribute of the *request*, following the instructions contained in the `message` printed above. To help us in this task, we can first inspect the current objective values as:

```
[12]: classification_request.content["objective_values"]
```

```
[12]: array([ -5.74637016,  -2.77951722,  -6.90637204, -11.62642964,
          0.349999  ])
```

We may then define our *response* with the required classifications to continute iterating the method:

```
[13]: # Source code 7
      response = {
          "classifications": ["<=", "0", "=", ">=", "<"],
          "levels": [-6.2, 0, 0, -3.0, 0],
          "number_of_solutions": 2,
      }

      classification_request.response = response

      save_request, _ = nimbus.iterate(classification_request)
```

We got a new request (`save_request`) which contains newly computed solutions based on the classifications given. These new solutions are:

```
[14]: save_request.content["objectives"]
```

```
[14]: [array([-5.74637016, -2.79903207, -6.90637204, -3.00000115,  0.13702734]),
       array([-5.54550116e+00, -2.80835322e+00, -7.14632853e+00, -2.39820101e+00,
              5.08845965e-07])]
```

This request is similar to the earlier one: its `content` attribute will contain at least a `message` entry with instructions on what needs to be defined in the `response` attribute of `save_request`. In Synchronous NIMBUS, the next

step would be to indicate whether we would like some of the computed solutions to be saved into an archive for later viewing. Further steps would consist of computing intermediate solutions, providing new classifications, and choosing a new preferred solution. However, we will conclude this use case here. Further information on the implementation of Synchronous NIMBUS in DESDEO can be found in the documentation of *desde-mcdm*.

Use case 2: data-based problem

Here we demonstrate how a data-based multiobjective optimization problem may be solved using DESDEO and an evolutionary method found in *desdeo-emo*.

We will assume that the river pollution problem has no known explicit form. Instead, the have only data on the probem in the form of variable and objective vector pairs. To emulate a data-based problem, we have pre-computed a small number (100) of solutions consisting of decision variable and objective values. We can then use these points to train a surrogate model which is computationally less expensive to evaluate and can therefore be readily used in an interactive method. We assume to have already computed the points mentioned earlier and that they are stored in a file `River_Pollution.csv` in a CSV format. The first two columns will consist of the decision variable values and the last five columns will consists of the objective values. We can use this data and together with *desdeo-problem* and pandas to formulate a data-based problem in DESDEO:

[15]:
```
# Source code 8
import pandas as pd
from desdeo_problem.problem import DataProblem


training_data = pd.read_csv("./data/River_pollution.csv", comment="#")

problem = DataProblem(
    data=training_data,
    variable_names=["x_0", "x_1"],
    objective_names=["f_1", "f_2", "f_3", "f_4", "f_5"],
    bounds=pd.DataFrame(
        [[0.3, 0.3], [1.0, 1.0]],
        columns=["x_0", "x_1"],
        index=["lower_bound", "upper_bound"]),
)
```

Next, we can train a surrogate to model each of the objectives. It is possible, and sometimes very desiderable, to use different surrogates for different objectives, but in this example we opt to use the same type of surrogate for each objective. We choose to use Gaussian regression:

[16]:
```
# Source code 9
from desdeo_problem.surrogatemodels.SurrogateModels import GaussianProcessRegressor

problem.train(
    models=GaussianProcessRegressor,
    model_parameters={"optimizer":"fmin_l_bfgs_b"}
)
```

The `GaussianProcessRegressor` we chose to use is a wrapper for the Gaussian process found in scikit-learn. Informatino about the hyperparameters given to the `optimizer` in Source code 9 can be found in scikit-learn's documentation. However, different surrogate models are also available and readily implemented, if absent.

To solve the problem being modeled by the surrogates, we choose and evolutionary method this time, namely the interactive variant of RVEA implemented in the *desdeo-emo* package. We will be supplying reference points (three other types of preference may also be supplied; more information can be found in the documentation of *desdeo-emo*). Since RVEA comes in a non-interactive version as well, we will have to specify explicitly to be interested in the

interactive version using the `interact` flag. Also, we will have to specify that we will be using a problem based on surragates in a similar fashion:

[17]:
```
warnings.filterwarnings("ignore", category=UserWarning)
# Source code 10
from desdeo_emo.EAs import RVEA

evolver = RVEA(
    problem,
    interact=True,
    use_surrogates=True
)

(_, _, refp_request, _), _ = evolver.requests()

refp_request.response = pd.DataFrame([[-5.7, -2.8, -6.9, -3.0, 0.1]],
                                     columns=["f_1", "f_2", "f_3", "f_4", "f_5"]
                                    )

(_, _, refp_request, _), _ = evolver.iterate(refp_request)
```

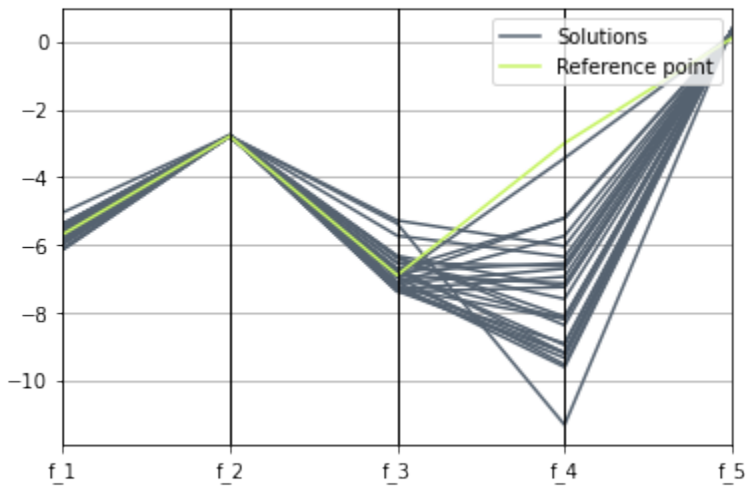
We have now computed new solutions according to our given preferences. Let us inspect them together with the reference point using a plot:

[18]:
```
import matplotlib.pyplot as plt
from pandas.plotting import parallel_coordinates

objectives_df = pd.DataFrame(data=evolver.population.objectives)
objectives_df.columns = ["f_1", "f_2", "f_3", "f_4", "f_5"]
objectives_df.insert(0, "type", "Solutions")
objectives_df = objectives_df.append(
    {"type": "Reference point", "f_1": -5.7, "f_2": -2.8, "f_3": -6.9, "f_4": -3.0,
↪"f_5": 0.1}, ignore_index=True)

parallel_coordinates(objectives_df, class_column="type", color=["#556270", "#C7F464"])
```

[18]:
```
<AxesSubplot:>
```



As seen from the plot above, quite many of the solutions roughly follow the given reference point. If none of the solutions found is attractive, we could supply new preferences to RVEA or we could otherwise tweak its parameters. However, what we have seen so far should give the rough idea how a data-based problem can be modeled and solved

using an evolutionary method in DESDEO. Lastly, we will consider the case of solving a computationally expensive multiobjective optimization problem.

Use case 3: computationally expensive problem

In the case of computationally expensive problems, we may encounter unacceptably long waiting times when solving the problems interactively. This is obviously not desired. In such cases, one solution is to compute a representation of the Pareto optimal front *a priori* and then use the front in an interactive method to explore the available solutions. In this way, we can avoid long waiting times during the interactive solution process.

Suppose then that the river pollution problem defined at the beginning of this notebook was very expensive. To generate an approximate representaton of the Pareto optimal front, we use the implementation of NSGAIII found in *desdeo-tools*. We use NSGAIII's non-interactive variant to evolve the population for 1000 generations (default) and then extract the non-dominated set of solutions from the final population by invoking the `end` method. This non-dominated set will then work as our representation of the Pareto front (as is often done). In code:

[19]:
```
# Source code 11
from desdeo_emo.EAs import NSGAIII

evolver = NSGAIII(mo_problem, interact=False)

while evolver.continue_evolution():
    evolver.iterate()

individuals, pareto_front = evolver.end()
```

We then use the E-NAUTILUS method found in *desdeo-mcdm*. E-NAUTILUS is a good choice of interactive method when we have already computed a representation of the Pareto front. Moreover, E-NAUTILUS is a tradeoff free method, which means that during the interactive process, we will be able to always improve our solution. This is because E-NAUTILUS starts from a wortst solution (often the nadir) from which the Pareto front is then gradually approched by taking steps towards it in each iteration according to expressed preferences. We can import E-NAUTILUS and instantiate it as:

[20]:
```
# Source code 12
from desdeo_mcdm.interactive import ENautilus

# 'pareto_front' stores the set of solutions computed using NSGA-III
method = ENautilus(pareto_front, ideal, nadir)

enautilus_request = method.start()
```

As seen, we have used the previously computed representation of the Pareto front, ideal point, and nadir point. Before we can start iterating the method, we will have to specify the number of intermediate solutions we wish to see in each iteration and the number of iterations to iterate the method for:

[21]:
```
print(enautilus_request.content["message"])
```

```
Please specify the number of iterations as 'n_iterations' to be carried out, and how␣
↪many intermediate points to show as 'n_points'.
```

[22]:
```
# Source code 13
response = {"n_iterations": 5, "n_points": 3}

enautilus_request.response = response

enautilus_request = method.iterate(enautilus_request)
```

In the new request, we have now intermediate points and the bounds of the currently reachable solution on Pareto front. We can inspect them:

[23]:
```
print("Intermediate points:\n", enautilus_request.content["points"])
print("Upper bounds:\n", enautilus_request.content["upper_bounds"])
print("Lower bounds:\n", enautilus_request.content["lower_bounds"])
```

```
Intermediate points:
 [[-5.00567056 -2.76999623 -1.49188962 -3.53924431  0.34839229]
 [-4.92637229 -2.77351336 -1.66963492 -2.04805943  0.29001499]
 [-5.06880182 -2.77651715 -0.32125135 -2.14502368  0.3499992 ]]
Upper bounds:
 [[-4.78970405 -2.77826009 -0.89527016 -1.92932773  0.35      ]
 [-4.94393521 -2.77205621 -5.00891907 -1.97414337  0.35      ]
 [-4.75774637 -2.77526782 -0.49390469 -1.92932773  0.35      ]]
Lower bounds:
 [[ -6.31360419  -2.80612481  -7.27014419 -11.20114082   0.21653705]
 [ -6.18512349  -2.83954576  -7.45610845  -5.59351819   0.02531614]
 [ -6.33771131  -2.82751575  -7.27014419 -11.62627225   0.02531614]]
```

From the intermediate points and bounds shown, we can then choose our most preferred one to continue iterating. We can do this by expressing the index (starting from zero) of the point we most prefer. Choosing the last intermediate point looks like this:

[24]:
```
# Source code 14
response = {"preferred_point_index": 2}

enautilus_request.response = response

enautilus_request = method.iterate(enautilus_request)
```

Next, we would have to choose a new preferred solution from the newly computed intermediate points. We would continue until we reach the desired number of iterations and end up with a solution on the representation of the Pareto front given when instantiating `ENautilus`. Since the solution found would still be picked from a set of (approximate) representations of Pareto optimal solutions, we could still improve the solution using the (expensive) original problem and a achievement scalarizing functions, for example.

The point of this use case was, however, to show how to handle a computationally expensive problem, which we have now demonstrated. Moreover, we have also shown how an evolutionary method (NSGA-III) has been hybridized with an MCDM method (E-NAUTILUS). This was a remarkably simple example, but more advanced applications are also possible.

Summary

We have seen in the considered use cases how an analytically defined problem can be solved using an MCDM method (NIMBUS); how a data-based problem can be modeled using surrogates and solved utilizing an evolutionary method (interactive RVEA); and we have seen how we can combine both evolutionary (NSGAIII) and MCDM (E-NAUTILUS) methods to solve a computationally expensive problem. In addition, we have seen the basic procedure on how to use interactive methods found in DESDEO (the request-response structure). While these examples were quite simple, they were still and excellent showcase of the basics in applying DESDEO to solve multiobjective optimization problems.

Lastly, it is good to observe that interacting with the methods as we have in this example, was cumbersome. We highly suggest that instead some sort of user interface is provided to interface to these method. This can facilitate greatly the interaction between humans and interactive methods found in DESDEO. We are currently developing some necessary building blocks to build such an interface, but we are not ready to discuss it here yet. But stay tuned!

More guides to be added eventually.

1.3.2 Examples

- How to use interactive methods: examples in desdeo-mcdm's documentation
- Examples on how to apply useful utilities: examples desdeo-tools's documentation
- How to define different problems: examples in desdeo-problem's documentation

1.4 Contributing

We welcome anybody interested to contribute to the packages found in the DESDEO framework. These contributions can be anything. Contributions do not have to necessarely be ground-breaking. From fixing typos in the documentation to implementing new multiobjective optimization methods, everything is welcome!

1.4.1 Guidelines and Conventions

The guidelines for code to be included in DESDEO should follow a few simple rules:

- Use spaces instead of tabs. Four spaces are used for indenting blocks when writing Python.
- Each line of code should not exceed 120 characters.
- Try to use type annotations for functions using Python's type hints.
- For naming classes use `CamelCase`, and for naming functions and methods use `snake_case`.
- Do not use global or file level declarations. Try to always encapsulate your declarations inside classes.
- Classes should be designed with duck typing conventions in mind.

Docstring style

The docstring style used throughout the DESDEO framework follows the style dictated by Google. Each function, class, and method should be documented.

1.4.2 Software development

This sections contains some basic tips to get started in contributing to DESDEO. We expect contributors to be proficient in at least the basics of Python.

Version control

DESDEO and all of its packages are under version control managed with `git`. It is highly suggested that anybody wanting to contribute to DESDEO should be first familiar with the basic principles of `git`. If terms such as *branching*, *committing*, *merging*, *staging*, and *cloning* are alien to the reader, a brief review of the basics of `git` is in place before contributing to DESDEO can start. At least if the reader wishes to make the experience as painless as possible...

For resources to learn git, git's official documentation is a very good place to start. Many other resources exists on the web as well. We will not be listing them all. However, if a gamified approach is desired instead of having to read documentation, an in-browser tutorial is available here.

Project management

The packages in DESDEO depend on many external Python packages. Sometimes a particular version of an external package is needed, which does not match the current version of the same external package on the computer's system DESDEO is going to be developed on. This can lead to many dependency problems. It is therefore highly recommended to use *virtual environments* when developing DESDEO to separate the packages needed by DESDEO from the packages present on the system's level.

Poetry is a modern and powerful tool to manage Python package dependencies and for building Python packages. `Poetry` is used throughout the DESDEO framework to manage and build its packages. While `Poetry` is *not* a tool for managing virtual environments, it nonetheless offers very simple commands to trivialize the task. Anybody contributing code to DESDEO should be familiar with `Poetry`.

Sometimes the Python version installed on a system is not compatible with the Python version required in DESDEO (3.7.x). In this case, it is recommended to use an external tool, such as pyenv, to facilitate the task of switching from one Python version to an other.

Example on how to get started

Once the reader is familiar with `git` and `Poetry`, starting to develop DESDEO should proceed relatively painlessly. Suppose we have a feature X which we wish to implement in DESDEO's `desdeo-mcdm` package. It is higly recommended to first `fork` the `desdeo-package` on GitHub on a separate repository and then cloning that repository. When doing so, use the forked repository's URL instead of the main repotiry's URL when cloning the project using `git`. Make sure to also setup your forked repository to be configured properly to be able to synchronize it later with the upstream repository.

We proceed by cloning the repository on our local machine:

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

Next, we should switch to the newly created directory:

```
$ cd desdeo-mcdm
```

We can now easily use `Poetry` to first create a Python virtual environment by issuing Poetry's `shell`- command and then use Poetry's `install`-command to install the `desdeo-mcdm` package locally in our newly created virtual environment:

```
$ poetry shell
$ poetry install
```

The `install` command might take a while. Once that is done, `desdeo-mcdm` should now be installed in our virtual environment and be fully usable in it.

To start implemeting our new feature X, we should start by making a new `branch` and switching to it using `git`. This ensures that the changes we make are relative to `desdeo-mcdm`'s master branch, at least the version of it which was available at the time of cloning it. Let us create a new branch now for our featrue X named `feature-X`:

```
$ git branch feature-X
$ git checkout feature-X
```

We are now ready to start implementing our changes to the package. Frequent `committing` is encouraged.

Suppose that we are now done impelenting our feature X. We now wish it was included to the master branch of `desdeo-mcdm`. To do so, we need to first switch back to the master branch

```
$ git checkout master
```

Then we need to make sure the master branch is up-to-date with the upstream version of the branch by issuing a pull. (If a `forked` repository is being used, the repository must first be synchronized with the upstream repository.)

```
$ git pull
```

Lastly, we will have to merge our `feature-X` branch containing our changes with the master branch

```
$ git merge feature-X
```

If all went well, we should now be ready to issue a `pull request`. However, if any conflicts emerge during the merging of the branches, these conflicts should be addressed before making a `pull request`. When the merge is free of conflicts, a `pull request` can be issued on GitHub or from the terminal. A maintainer of the repository will then review your changes and either accept them into the upstream or request revisions to be made before the new code can be accepted.

1.4.3 Documentation

Introduction

To build the documentation for the DESDEO framework and its various modules, Sphinx is used. Sphinx offers excellent utilities for automatically generating API documentation based on existing documentation located in source code, and for adding custom content.

Automatically generated documentation and custom content is specified as reStructuredText. ReStructuredText is a markup language just like Markdown or html, but offers the possibility to extend the language for specific domains. The file extention `.rst` is used for files containing reStructuredText content. Sphinx can then be used to generate documentation in various formats, such as html and pdf, based on content provided as reStructuredText.

Resources to get started with Sphinx

The official documentation offers a good guide for getting started. It is advised to read through the guide before contributing to the documentation. After reading the guide, the reader is encouraged to check out the contents of the source file used to generate the current page. The source file can be accessed by going to the top of this page and following the *Edit on GitHub*-link. It is also advised to check out the content of the *docs* file found in the main repository of the DESDEO framework. After checking the source file used to generate this page, the user should be familiar with at least basic sectioning, hyperlinks, code blocks, note blocks, and lists.

Other useful resources include:

- Official documentation for reStructuredText.
- ReStructuredText syntax cheatsheet.
- A conference talk about Sphinx given during PyCon 2018. (YouTube has also many other videos on Sphinx as well)
- A more through tutorial written by the matplotlib developers on how to achieve a documentation similar to theirs.

Extensions

In the DESDEO framework, some Sphinx extensions are used to faciliate automatic documentation generation. At least the following extensions are used:

Included in Sphinx:

- Sphinx.ext.autodoc for automatically generating documentation based on docstrings.
- Sphinx.ext.napoleon for parsing the Google styled docstrings.
- Sphinx.ext.viewcode for accessing the documented source code from the documentation itself.

User provided extensions:

- Sphinx-autodoc-typehints for better type hints.
- automodapi for even better automatic API documentation generation.
- nbsphinx for converting Python notebooks into rst pages.

Building and testing the documentation

If Sphinx has been setup following the official quick guide, the documentation can be build by running the commnad

```
make html
```

in the root direcotry containing the documentation. This will produce documentation in an html format residing in the `_build` folder in the documentation's root directory. To view the documentation built, use any web browser. For example, with Firefox, this is achieved by issuing the command

```
firefox _build/html/index.html
```

Note: The directory `_build` generated by `sphinx-quickstart` should not be under version control.

Deployment

Note: Most of the content in this section is relevant only when setting up the documentation for the first time for a module.

The documentation for each of the DESDEO modules is hosted on readthedocs.org. For the documentation to be build correctly, a YAML congiguration file named `.readthedocs.yml` should be present in the root directory of the project (not the root directory of the documentation!) A minimal configarion file could look like this:

```
# Required
version: 2

# Build documentation in the docs/ directory with Sphinx
Sphinx:
configuration: docs/conf.py

# Optionally set the version of Python and requirements required to build your docs
python:
version: 3.7
install:
    - requirements: docs/requirements.txt
```

Especially the locations of the configuration files `docs/conf.py` and `docs/requirements.txt` are important to enable readthedocs to correctly build the documentation.

Note: The requirements file should contain the requirements for **building the documentation**. It does not necessarely need to contain all the requirements of the module the documentation is being build for. However, for building the documentation for some of the modules, like `desdeo-mcdm` for example, the whole module needs to be installed for Sphinx to be able to compile the documentation. In that case, having the project's whole requirements in the requirements file pointed at in `.readthedocs.yml` is justified.

If a `requirements.txt` if required, but *poetry* is used to manage dependencies, then the command

```
poetry export --dev -f requirements.txt > requirements.txt
```

can be used to generate a requirements file.

For more configuration options, go here. The whole documentation for readthedocs can be found here.

Caveats

Some common caveats with Sphinx:

- The intendation Sphinx expects in the reStructuredText files is **three spaces** to specify the scope of the *options* and *content* of a *directive*. Options should follow the directive immediately on the following line, one option per line, and the content should be separated by one blank line from the options (if no options are provided, the blank line should be between the directive and the contents). For example, the following is correct:

```
.. toctree::
   :maxdepht: 2

   content
   morecontent
```

  The following, however, is **incorrect**:

```
.. toctree::
   :maxdepht: 2
   content
   morecontent
```

- If the contents of an item in a list span more than one line, the lines following the first line should have their indentation starting at the same level as the content on the first line. I.e.:

```
- This is the first line
  this is the second line
  this is the third line
  notice the indentation
```

1.5 Glossary

decision maker, DM A domain expert with adequate expertise related to an optimization problem able to provide preference information.

Pareto optimal solution A feasible solution to a multiobjective optimization problem which corresponds to an objective vector that cannot be exchanged for any other objective vector resulting from some other feasible solution wihtout having to make a trade-off in at least one objective value.

Pareto front The set of Pareto optimal solutions.

nadir (point) The worst possible objective values of a Pareto front.

ideal (point) The best possible objective values of a Pareto front.

reference point Desirable aspiration levels for each objective function.

Index