desdeo_emo

Release 1.0

Multiobjective Optimization Group

Sep 21, 2021

CONTENTS

The evolutionary algorithms package within the *desdeo* framework.

Currently supported:

- Multi-objective optimization with visualization and interaction support.
- Preference is accepted as a reference point.
- Surrogate modelling (neural networks and genetic trees) evolved via EAs.
- Surrogate assisted optimization
- Constraint handling using *RVEA*
- IOPIS optimization using *RVEA* and *NSGA-III*

Currently **NOT** supported:

- Binary and integer variables.

To test the code, open the binder link and read example.ipynb.

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-emo
```

2.2 For developers

Download the code or clone it with the following command:

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

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

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

CHAPTER

THREE

CURRENTLY IMPLEMENTED METHODS

Algorithm	Reference
RVEA	R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016
NSGA-III	K. Deb and H. Jain, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints," in IEEE Transactions on Evolutionary Computation, vol. 18, no. 4, pp. 577-601, Aug. 2014, doi: 10.1109/TEVC.2013.2281535.
MOEA/D	Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," in IEEE Transactions on Evolutionary Computation, vol. 11, no. 6, pp. 712-731, Dec. 2007, doi: 10.1109/TEVC.2007.892759.
PPGA	Laumanns, M., Rudolph, G., & Schwefel, H. P. (1998). A spatial predator-prey approach to multi-objective optimization: A preliminary study. In International Conference on Parallel Problem Solving from Nature (pp. 241-249). Springer, Berlin, Heidelberg.
IOPIS-RVEA	Saini B.S., Hakanen J., Miettinen K. (2020) A New Paradigm in Interactive Evolutionary Multiobjective Optimization. In: Bäck T. et al. (eds) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020. Lecture Notes in Computer Science, vol 12270. Springer, Cham. https://doi.org/10.1007/978-3-030-58115-2_17
IOPIS-NSGA-III	Saini B.S., Hakanen J., Miettinen K. (2020) A New Paradigm in Interactive Evolutionary Multiobjective Optimization. In: Bäck T. et al. (eds) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020. Lecture Notes in Computer Science, vol 12270. Springer, Cham. https://doi.org/10.1007/978-3-030-58115-2_17

3.1 Background concepts

3.1.1 Evolutionary algorithms

Evolutionary algorithms (EAs) are optimization algorithms that emulate the process of evolution via natural selection to find optimal solutions to single- or multiobjective optimization problems (MOPs). This is achieved by taking a population of candidate solutions, known as individuals. The individuals mix and match their properties with other individuals in a crossover process to form a new batch of candidate solutions, known as offspring. The method also involves a random change in the properties of the offspring, which occurs via a process called mutation. Finally, during the selection step, the approach decides which individuals remain for the next generation based on a fitness criterion. The surviving population members then undergo the same steps as mentioned above and slowly converge towards optimality. The general structure of an EA is the following:

```
Step 1: t= 0

Step 2: Initialize P(t)

Step 3: Evaluate P(t)

Step 4: While not terminate do

    P'(t) = variation [P(t)];------------->(Crossover and mutation)

    evaluate [P'(t)];

    P(t+1) = select [P'(t) U P(t)];

    t = t + 1;
end
```

Different EAs differ in the way they handle the population; conduct crossover, mutation, and selection; and calculate the fitness criteria. The `desdeo-emo` package currently provides implementations of ** Decomposition-based EAs**, which specialize in multiobjective optimization. The package also provides the *EvoNN*, *BioGP*, and *EvoDN2* algorithms which can be used to train surrogate models in an evolutionary manner.

3.1.2 Evolutionary operators

- **Selection:** Motivation is to preserve the best (make multiple copies) and eliminates the worst.

  E.g., Roulette wheel, Tournament, steady-state, etc.

- **Mutation:** Keep diversity in the population.

  E.g., Polynomial mutation, random mutation, one-point mutation, etc.

- **Crossover:** Create new solutions by considering more than one individual.

  E.g., Simulated binary crossover, Linear crossover, blend crossover, uniform, one-point, etc.

3.1.3 Multiobjective Evolutionary Algorithms

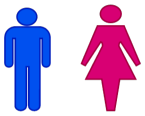
Multiobjective Evolutionary Algorithms (MOEAs) are Evolutionary algorithms employed to solve multiobjective optimization problems (MOPs). They use a population of solutions to obtain a diverse set of trade-off solutions close to the Pareto optimal front. There are mainly three types of MOEAs:

- **Dominance-based MOEAs:** These approaches sort the individuals of the population using the dominance relationship, obtaining multiple convergence layers. In addition, an explicit diversity preservation scheme is employed in the last layer to maintain a diverse set of solutions. Dominance-based MOEAs have some advantages like a simple principle, easy understanding, and fewer parameters than other types of MOEAs. However, their ability to guarantee convergence degrades when the number of objectives exceeds three, mainly due to the loss of selection pressure. Among the most popular approaches following this principle are: NSGA-II and SPEA2
- **Indicator-based MOEAs:** These algorithms employ a performance indicator (e.g., hypervolume) as a selection criterion to rank non-dominated solutions. However, they require a high computational complexity to compute the indicator values, especially in problems with more than three objectives. Some of the well-known indicator-based MOEAs are IBEA, SIBEA, etc.
- **Decomposition-based MOEAs:** These techniques transform a MOP into a set of single-objective problems (or subMOPs) using scalarizing functions (e.g., Tchebycheff, PBI, etc.). These are solved simultaneously using an

Components of an Evolutionary Algorithm

Population

Set of individuals (solutions)



Parent

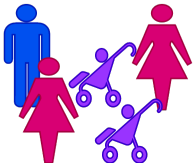
Member of current generation

Evolutionary Operators

Crossover, mutation, and selection

Children

Members of next generation



Generation

Successively created populations (EA iteration)

Evolutionary Algorithm. Most of these algorithms utilize reference vectors (also known as weight vectors) as a mechanism to maintain the population's diversity. The evolutionary operators for these techniques are similar to the ones employed in the rest of the EAs. However, during the selection process, they use an aggregated fitness value instead of a vector. Some of the most representative MOEAs in this category are MOEA/D, NSGA-III, and RVEA.

3.2 Examples

3.2.1 Basics of desdeo-emo

```
[ ]: import plotly.graph_objects as go
     import numpy as np
     import pandas as pd

     from desdeo_problem import variable_builder, ScalarObjective, MOProblem
     from desdeo_problem.testproblems.TestProblems import test_problem_builder

     from desdeo_emo.EAs.NSGAIII import NSGAIII
     from desdeo_emo.EAs.RVEA import RVEA
     from desdeo_emo.utilities.plotlyanimate import animate_init_, animate_next_
```

Coello MOP7

Definition

MOP5 P_{true} disconnected and unsymmetric, PF_{true} connected (a 3-D Pareto curve)	$F = (f_1(x,y), f_2(x,y), f_3(x,y))$, where $f_1(x,y) = 0.5 * (x^2 + y^2) + \sin(x^2 + y^2),$ $f_2(x,y) = \frac{(3x - 2y + 4)^2}{8} + \frac{(x - y + 1)^2}{27} + 15,$ $f_3(x,y) = \frac{1}{(x^2 + y^2 + 1)} - 1.1e^{(-x^2 - y^2)}$	$-30 \leq x, y \leq 30$

Pareto set and front

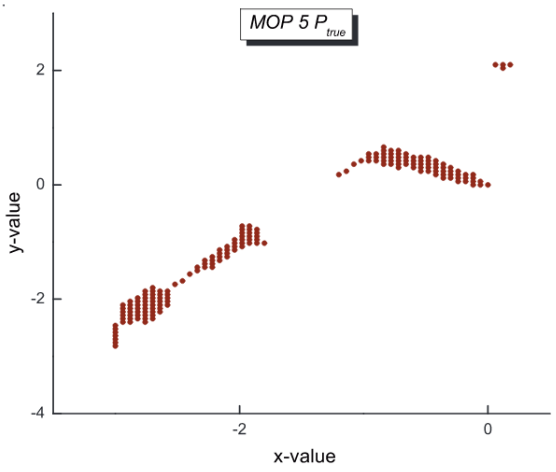


Fig. 4.9. MOP5 P_{true}

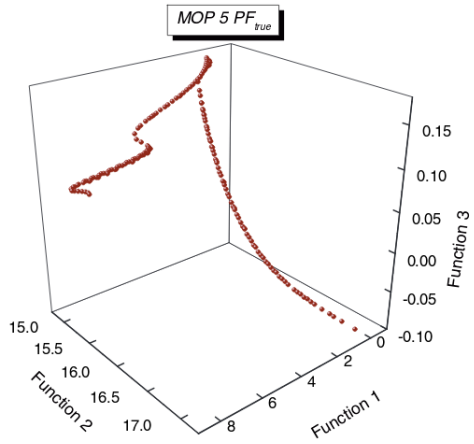


Fig. 4.10. MOP5 PF_{true}

Define objective functions

[2]:
```
def f_1(x):
    term1 = ((x[:,0] - 2) ** 2) / 2
    term2 = ((x[:,1] + 1) ** 2) / 13
    return term1 + term2 + 3

def f_2(x):
    term1 = ((x[:, 0] + x[:, 1] - 3) ** 2) / 36
    term2 = ((-x[:, 0] + x[:, 1] + 2) ** 2) / 8
    return term1 + term2 - 17

def f_3(x):
    term1 = ((x[:, 0] + (2 * x[:, 1]) - 1) ** 2) / 175
    term2 = ((-x[:, 0] + 2* x[:, 1]) ** 2) / 17
    return term1 + term2 - 13
```

Note that the expected input `x` is two dimensional. It should be a 2-D numpy array.

Create Variable objects

[3]:
```
help(variable_builder)
```

```
Help on function variable_builder in module desdeo_problem.problem.Variable:

variable_builder(names: List[str], initial_values: Union[List[float], numpy.ndarray],␣
↪lower_bounds: Union[List[float], numpy.ndarray] = None, upper_bounds:␣
↪Union[List[float], numpy.ndarray] = None) -> List[desdeo_problem.problem.Variable.
↪Variable]
```

(continues on next page)

(continued from previous page)

```
    Automatically build all variable objects.

    Args:
        names (List[str]): Names of the variables
        initial_values (np.ndarray): Initial values taken by the variables.
        lower_bounds (Union[List[float], np.ndarray], optional): Lower bounds of the
            variables. If None, it defaults to negative infinity. Defaults to None.
        upper_bounds (Union[List[float], np.ndarray], optional): Upper bounds of the
            variables. If None, it defaults to positive infinity. Defaults to None.

    Raises:
        VariableError: Lengths of the input arrays are different.

    Returns:
        List[Variable]: List of variable objects
```

[4]:
```
list_vars = variable_builder(['x', 'y'],
                             initial_values = [0,0],
                             lower_bounds=[-400, -400],
                             upper_bounds=[400, 400])
list_vars
```

[4]:
```
[<desdeo_problem.problem.Variable.Variable at 0x1fe27917448>,
 <desdeo_problem.problem.Variable.Variable at 0x1fe27917488>]
```

Create Objective objects

[5]:
```
f1 = ScalarObjective(name='f1', evaluator=f_1)
f2 = ScalarObjective(name='f2', evaluator=f_2)
f3 = ScalarObjective(name='f3', evaluator=f_3)
list_objs = [f1, f2, f3]
```

Create the problem object

[6]:
```
problem = MOProblem(variables=list_vars, objectives=list_objs)
```

Using the EAs

Pass the problem object to the EA, pass parameters as arguments if required.

[7]:
```
help(NSGAIII)
```

```
Help on class NSGAIII in module desdeo_emo.EAs.NSGAIII:

class NSGAIII(desdeo_emo.EAs.BaseEA.BaseDecompositionEA)
 |  NSGAIII(problem: desdeo_problem.problem.Problem.MOProblem, population_size: int =␣
↪None, population_params: Dict = None, n_survive: int = None, initial_population:␣
↪desdeo_emo.population.Population.Population = None, lattice_resolution: int = None,␣
↪selection_type: str = None, a_priori: bool = False, interact: bool = False, use_
↪surrogates: bool = False, n_iterations: int = 10, n_gen_per_iter: int = 100, total_
↪function_evaluations: int = 0)
```

(continues on next page)

(continued from previous page)

```
 |
 |  Python Implementation of NSGA-III. Based on the pymoo package.
 |
 |  Most of the relevant code is contained in the super class. This class just assigns
 |  the NSGAIII selection operator to BaseDecompositionEA.
 |
 |  Parameters
 |  ----------
 |  problem : MOProblem
 |      The problem class object specifying the details of the problem.
 |  population_size : int, optional
 |      The desired population size, by default None, which sets up a default value
 |      of population size depending upon the dimensionaly of the problem.
 |  population_params : Dict, optional
 |      The parameters for the population class, by default None. See
 |      desdeo_emo.population.Population for more details.
 |  initial_population : Population, optional
 |      An initial population class, by default None. Use this if you want to set up
 |      a specific starting population, such as when the output of one EA is to be
 |      used as the input of another.
 |  lattice_resolution : int, optional
 |      The number of divisions along individual axes in the objective space to be
 |      used while creating the reference vector lattice by the simplex lattice
 |      design. By default None
 |  selection_type : str, optional
 |      One of ["mean", "optimistic", "robust"]. To be used in data-driven␣
↪optimization.
 |      To be used only with surrogate models which return an "uncertainity" factor.
 |      Using "mean" is equivalent to using the mean predicted values from the␣
↪surrogate
 |      models and is the default case.
 |      Using "optimistic" results in using (mean - uncertainity) values from the
 |      the surrogate models as the predicted value (in case of minimization). It is
 |      (mean + uncertainity for maximization).
 |      Using "robust" is the opposite of using "optimistic".
 |  a_priori : bool, optional
 |      A bool variable defining whether a priori preference is to be used or not.
 |      By default False
 |  interact : bool, optional
 |      A bool variable defining whether interactive preference is to be used or
 |      not. By default False
 |  n_iterations : int, optional
 |      The total number of iterations to be run, by default 10. This is not a hard
 |      limit and is only used for an internal counter.
 |  n_gen_per_iter : int, optional
 |      The total number of generations in an iteration to be run, by default 100.
 |      This is not a hard limit and is only used for an internal counter.
 |  total_function_evaluations :int, optional
 |      Set an upper limit to the total number of function evaluations. When set to
 |      zero, this argument is ignored and other termination criteria are used.
 |
 |  Method resolution order:
 |      NSGAIII
 |      desdeo_emo.EAs.BaseEA.BaseDecompositionEA
 |      desdeo_emo.EAs.BaseEA.BaseEA
 |      builtins.object
 |
```

(continues on next page)

(continued from previous page)

```
 |  Methods defined here:
 |
 |  __init__(self, problem: desdeo_problem.problem.Problem.MOProblem, population_size:
↪ int = None, population_params: Dict = None, n_survive: int = None, initial_
↪population: desdeo_emo.population.Population.Population = None, lattice_resolution:␣
↪int = None, selection_type: str = None, a_priori: bool = False, interact: bool =␣
↪False, use_surrogates: bool = False, n_iterations: int = 10, n_gen_per_iter: int =␣
↪100, total_function_evaluations: int = 0)
 |      Initialize EA here. Set up parameters, create EA specific objects.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from desdeo_emo.EAs.BaseEA.BaseDecompositionEA:
 |
 |  end(self)
 |      Conducts non-dominated sorting at the end of the evolution process
 |
 |      Returns:
 |          tuple: The first element is a 2-D array of the decision vectors of the␣
↪non-dominated solutions.
 |              The second element is a 2-D array of the corresponding objective␣
↪values.
 |
 |  manage_preferences(self, preference=None)
 |      Run the interruption phase of EA.
 |
 |      Use this phase to make changes to RVEA.params or other objects.
 |      Updates Reference Vectors (adaptation), conducts interaction with the user.
 |
 |  request_plot(self) -> desdeo_tools.interaction.request.SimplePlotRequest
 |
 |  request_preferences(self) -> Union[NoneType, Tuple[desdeo_tools.interaction.
↪request.PreferredSolutionPreference, desdeo_tools.interaction.request.
↪NonPreferredSolutionPreference, desdeo_tools.interaction.request.
↪ReferencePointPreference, desdeo_tools.interaction.request.BoundPreference]]
 |
 |  requests(self) -> Tuple
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from desdeo_emo.EAs.BaseEA.BaseEA:
 |
 |  check_FE_count(self) -> bool
 |      Checks whether termination criteria via function evaluation count has been
 |          met or not.
 |
 |      Returns:
 |          bool: True is function evaluation count limit NOT met.
 |
 |  continue_evolution(self) -> bool
 |      Checks whether the current iteration should be continued or not.
 |
 |  continue_iteration(self)
 |      Checks whether the current iteration should be continued or not.
 |
 |  iterate(self, preference=None) -> Tuple
 |      Run one iteration of EA.
 |
 |      One iteration consists of a constant or variable number of
```

(continues on next page)

(continued from previous page)

```
 |      generations. This method leaves EA.params unchanged, except the current
 |      iteration count and gen count.
 |
 |  start(self)
 |      Mimics the structure of the mcdm methods. Returns the request objects from␣
↪self.retuests().
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from desdeo_emo.EAs.BaseEA.BaseEA:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

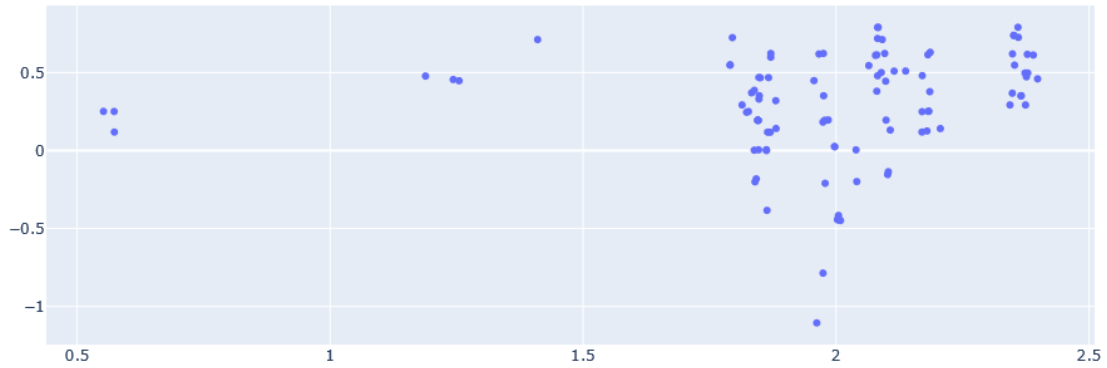
[8]:
```
evolver = NSGAIII(problem,
                  n_iterations=10,
                  n_gen_per_iter=100,
                  population_size=100)
```

[9]:
```
while evolver.continue_evolution():
    evolver.iterate()
```

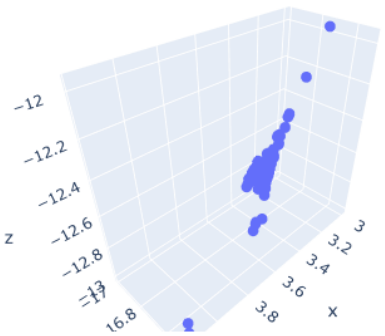
Extracting optimized decision variables and objective values

[10]:
```
individuals, solutions = evolver.end()

fig1 = go.Figure(
    data=go.Scatter(
        x=individuals[:,0],
        y=individuals[:,1],
        mode="markers"))
fig1
```



[11]:
```
fig2 = go.Figure(data=go.Scatter3d(x=solutions[:,0],
                                   y=solutions[:,1],
                                   z=solutions[:,2],
                                   mode="markers",
                                   marker_size=5))
fig2
```



[13]:
```
pd.DataFrame(solutions).to_csv("MOP7_true_front.csv")
```

Coello MOP5

Definition

MOP5 P_{true} disconnected and unsymmetric, PF_{true} connected (a 3-D Pareto curve)	$F = (f_1(x,y), f_2(x,y), f_3(x,y))$, where $f_1(x,y) = 0.5 * (x^2 + y^2) + \sin(x^2 + y^2),$ $f_2(x,y) = \frac{(3x - 2y + 4)^2}{8} + \frac{(x - y + 1)^2}{27} + 15,$ $f_3(x,y) = \frac{1}{(x^2 + y^2 + 1)} - 1.1e^{(-x^2 - y^2)}$	$-30 \leq x, y \leq 30$

Pareto set and front

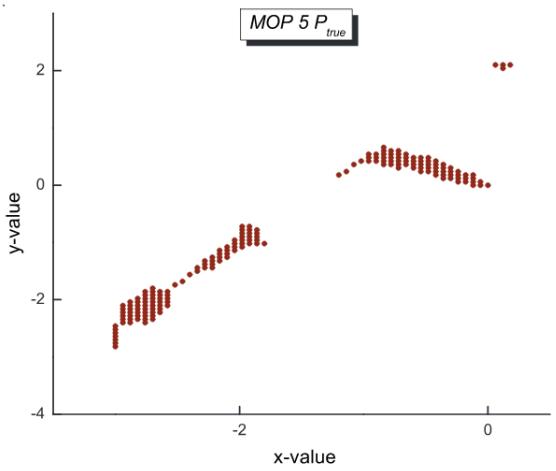


Fig. 4.9. MOP5 P_{true}

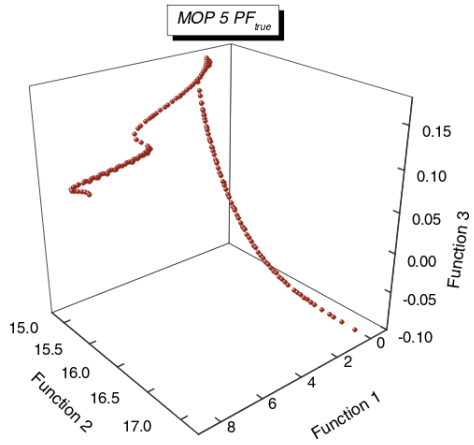


Fig. 4.10. MOP5 PF_{true}

[14]:
```
def f_4(x):
    term = x[:, 0]**2 + x[:, 1]**2
    return 0.5*term + np.sin(term)

def f_5(x):
    term1 = ((3*x[:, 0] - 2*x[:,1] + 4)**2)/8
    term2 = ((x[:, 0] - x[:,1] + 1)**2)/27
    return term1 + term2 + 15

def f_6(x):
    term = x[:, 0]**2 + x[:, 1]**2
    return (1/(term + 1)) - 1.1 * np.exp(-term)
```

[15]:
```
list_vars = variable_builder(['x', 'y'],
                             initial_values = [0,0],
                             lower_bounds=[-30, -30],
                             upper_bounds=[30, 30])
```

[16]:
```
f1 = ScalarObjective(name='f1', evaluator=f_4)
f2 = ScalarObjective(name='f2', evaluator=f_5)
f3 = ScalarObjective(name='f3', evaluator=f_6)
```

[17]:
```
problem = MOProblem(variables=list_vars, objectives=[f1, f2, f3])
```

[23]:
```
evolver = NSGAIII(problem)
```

[24]:
```
individual, solutions = evolver.end()
figure = animate_init_(solutions, filename="MOP5.html")
```

```
Plot saved as:  MOP5.html
View the plot by opening the file in browser.
To view the plot in Jupyter Notebook, use the IFrame command.
```

Refresh the plot page after each iteration to get the updated animation

[25]:
```
while evolver.continue_evolution():
    print(f"Running iteration {evolver._iteration_counter+1}")
    evolver.iterate()
    non_dominated = evolver.population.non_dominated_fitness()
    figure = animate_next_(
        evolver.population.objectives[non_dominated],
        figure,
        filename="MOP5.html",
        generation=evolver._iteration_counter,
    )
```

```
Running iteration 1
Running iteration 2
Running iteration 3
Running iteration 4
Running iteration 5
Running iteration 6
Running iteration 7
Running iteration 8
Running iteration 9
Running iteration 10
```

Interaction in EAs

The reference point method has been implemented.

[26]:
```
help(test_problem_builder)
```

```
Help on function test_problem_builder in module desdeo_problem.testproblems.
↪TestProblems:

test_problem_builder(name: str, n_of_variables: int = None, n_of_objectives: int =␣
↪None) -> desdeo_problem.problem.Problem.MOProblem
    Build test problems. Currently supported: ZDT1-4, ZDT6, and DTLZ1-7.

    Args:
        name (str): Name of the problem in all caps. For example: "ZDT1", "DTLZ4",␣
↪etc.
        n_of_variables (int, optional): Number of variables. Required for DTLZ␣
↪problems,
            but can be skipped for ZDT problems as they only support one variable␣
↪value.
        n_of_objectives (int, optional): Required for DTLZ problems,
            but can be skipped for ZDT problems as they only support one variable␣
↪value.

    Raises:
        ProblemError: When one of many issues occur while building the MOProblem
            instance.

    Returns:
        MOProblem: The test problem object
```

[27]:
```
problem = test_problem_builder(name="DTLZ1", n_of_variables=30, n_of_objectives=3)
```

[28]:
```
evolver = RVEA(problem, interact=True, n_iterations=5, n_gen_per_iter=400)
figure = animate_init_(
    evolver.population.objectives[evolver.population.non_dominated_fitness()],
    filename="dtlz1.html")
```

```
Plot saved as:  dtlz1.html
View the plot by opening the file in browser.
To view the plot in Jupyter Notebook, use the IFrame command.
```

[29]:
```
pref, plot = evolver.start()
```

Loop over the following three cells, updating the preferences as desired

Try changing the preference once the Pareto front has been reached. Note the format of providing the preference. Check out the content of pref and plot

[30]:
```
print(pref[0].content['message'])
```

```
Please provide preferences. There is four ways to do this. You can either:

        1: Select preferred solution(s)
        2: Select non-preferred solution(s)
```

(continues on next page)

(continued from previous page)

```
        3: Specify a reference point worse than or equal to the ideal point
        4: Specify desired ranges for objectives.

In case you choose

1, please specify index/indices of preferred solutions in a numpy array (indexing␣
↪starts from 0).
For example:
        numpy.array([1]), for choosing the solutions with index 1.
        numpy.array([2, 4, 5, 16]), for choosing the solutions with indices 2, 4, 5,␣
↪and 16.

2, please specify index/indices of non-preferred solutions in a numpy array (indexing␣
↪starts from 0).
For example:
        numpy.array([3]), for choosing the solutions with index 3.
        numpy.array([1, 2]), for choosing the solutions with indices 1 and 2.

3, please provide a reference point worse than or equal to the ideal point:

f1    2.63548
f2    0.98086
f3    1.17816
Name: ideal, dtype: object
The reference point will be used to focus the reference vectors towards the preferred␣
↪region.
If a reference point is not provided, the previous state of the reference vectors is␣
↪used.
If the reference point is the same as the ideal point, the reference vectors are␣
↪spread uniformly in the objective space.

4, please specify desired lower and upper bound for each objective, starting from
the first objective and ending with the last one. Please specify the bounds as a␣
↪numpy array containing
lists, so that the first item of list is the lower bound and the second the upper␣
↪bound, for each
objective.
        For example: numpy.array([[1, 2], [2, 5], [0, 3.5]]), for problem with three␣
↪objectives.
Ideal vector:
f1    2.63548
f2    0.98086
f3    1.17816
Name: ideal, dtype: object
Nadir vector:
f1    inf
f2    inf
f3    inf
Name: nadir, dtype: object.
```

[39]:
```
response = evolver.population.ideal_fitness_val + [0.5,0.7,0.1]
pref[2].response = pd.DataFrame([response], columns=pref[2].content['dimensions_data
↪'].columns)
```

[40]:
```
pref, plot = evolver.iterate(pref[2])
figure = animate_next_(
```

(continues on next page)

(continued from previous page)

```
    plot.content['data'].values,
    figure,
    filename="dtlz1.html",
    generation=evolver._iteration_counter,
)

message = (f"Current generation number:{evolver._current_gen_count}. "
           f"Is looping back recommended: {'Yes' if evolver.continue_evolution() else
↪'No'}")
print(message)
```

```
Current generation number:2000. Is looping back recommended: No
```

[ ]:

3.2.2 The River Pollution Problem

[1]:
```
from desdeo_emo.EAs.RVEA import RVEA
from desdeo_problem import variable_builder, ScalarObjective, VectorObjective,␣
↪MOProblem

import numpy as np
import pandas as pd
from desdeo_emo.utilities.plotlyanimate import animate_init_, animate_next_
```

[2]:
```
# create the problem
def f_1(x):
    return 4.07 + 2.27 * x[:, 0]

def f_2(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):
    return 8.21 - 0.71 / (1.09 - x[:, 0]**2)

def f_4(x):
    return 0.96 - 0.96 / (1.09 - x[:, 1]**2)

# def f_5(x):
    # return -0.96 + 0.96 / (1.09 - x[:, 1]**2)

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

[3]:
```
f1 = ScalarObjective(name="f1", evaluator=f_1, maximize=True)
f2 = ScalarObjective(name="f2", evaluator=f_2, maximize=True)
f3 = ScalarObjective(name="f3", evaluator=f_3, maximize=True)
f4 = ScalarObjective(name="f4", evaluator=f_4, maximize=True)
f5 = ScalarObjective(name="f5", evaluator=f_5, maximize=False)
```

[4]:
```
varsl = variable_builder(["x_1", "x_2"],
        initial_values=[0.5, 0.5],
```

(continues on next page)

(continued from previous page)

```
        lower_bounds=[0.3, 0.3],
        upper_bounds=[1.0, 1.0]
        )
```

[5]:
```
problem = MOProblem(variables=varsl, objectives=[f1, f2, f3, f4, f5])
```

[6]:
```
evolver = RVEA(problem, interact=True, n_iterations=5, n_gen_per_iter=100)
figure = animate_init_(evolver.population.objectives, filename="river.html")
```

```
Plot saved as:  river.html
View the plot by opening the file in browser.
To view the plot in Jupyter Notebook, use the IFrame command.
```

[7]:
```
pref, plot = evolver.start()
```

[8]:
```
print(plot.content["dimensions_data"])
```

```
                f1       f2       f3          f4        f5
minimize        -1       -1       -1          -1         1
ideal     6.33643  3.40833  7.49815 -0.00301699  0.013136
nadir        -inf     -inf     -inf        -inf       inf
```

[9]:
```
print(pref[0].content['message'])
```

```
Please provide preferences. There is four ways to do this. You can either:

        1: Select preferred solution(s)
        2: Select non-preferred solution(s)
        3: Specify a reference point worse than or equal to the ideal point
        4: Specify desired ranges for objectives.

In case you choose

1, please specify index/indices of preferred solutions in a numpy array (indexing␣
↪starts from 0).
For example:
        numpy.array([1]), for choosing the solutions with index 1.
        numpy.array([2, 4, 5, 16]), for choosing the solutions with indices 2, 4, 5,␣
↪and 16.

2, please specify index/indices of non-preferred solutions in a numpy array (indexing␣
↪starts from 0).
For example:
        numpy.array([3]), for choosing the solutions with index 3.
        numpy.array([1, 2]), for choosing the solutions with indices 1 and 2.

3, please provide a reference point worse than or equal to the ideal point:

f1       6.33643
f2       3.40833
f3       7.49815
f4   -0.00301699
f5      0.013136
Name: ideal, dtype: object
The reference point will be used to focus the reference vectors towards the preferred␣
↪region.
```

(continues on next page)

(continued from previous page)

```
If a reference point is not provided, the previous state of the reference vectors is␣
↪used.
If the reference point is the same as the ideal point, the reference vectors are␣
↪spread uniformly in the objective space.

4, please specify desired lower and upper bound for each objective, starting from
the first objective and ending with the last one. Please specify the bounds as a␣
↪numpy array containing
lists, so that the first item of list is the lower bound and the second the upper␣
↪bound, for each
objective.
        For example: numpy.array([[1, 2], [2, 5], [0, 3.5]]), for problem with three␣
↪objectives.
Ideal vector:
f1       6.33643
f2       3.40833
f3       7.49815
f4   -0.00301699
f5      0.013136
Name: ideal, dtype: object
Nadir vector:
f1   -inf
f2   -inf
f3   -inf
f4   -inf
f5    inf
Name: nadir, dtype: object.
```

[16]:
```
pref[2].response = pd.DataFrame([[6.3,3.3,7,-2,0.3]],
                                columns=pref[2].content['dimensions_data'].columns)
```

[17]:
```
pref, plot = evolver.iterate(pref[2])
figure = animate_next_(
    plot.content['data'].values,
    figure,
    filename="river.html",
    generation=evolver._iteration_counter,
)

message = (f"Current generation number:{evolver._current_gen_count}. "
           f"Is looping back recommended: {'Yes' if evolver.continue_evolution() else
↪'No'}")
print(message)
```

```
Current generation number:400. Is looping back recommended: Yes
```

[ ]:

3.3 API Reference

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

3.3.1 desdeo_emo

Subpackages

desdeo_emo.EAs

This module contains classes implementing different Evolutionary algorithms.

Submodules

desdeo_emo.EAs.BaseEA

Module Contents

Classes

BaseEA	This class provides the basic structure for Evolutionary algorithms.
BaseDecompositionEA	The Base class for decomposition based EAs.

exception desdeo_emo.EAs.BaseEA.**eaError**
Bases: Exception

Raised when an error related to EA occurs

class desdeo_emo.EAs.BaseEA.**BaseEA**(*a_priori: bool = False*, *interact: bool = False*, *selection_operator: Type[desdeo_emo.selection.SelectionBase.SelectionBase] = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *use_surrogates: bool = False*)
This class provides the basic structure for Evolutionary algorithms.

start(*self*)
Mimics the structure of the mcdm methods. Returns the request objects from self.retuests().

end(*self*)
To be run at the end of the evolution process.

_next_gen(*self*)
Run one generation of an EA. Change nothing about the parameters.

iterate(*self*, *preference=None*) → Tuple
Run one iteration of EA.

One iteration consists of a constant or variable number of generations. This method leaves EA.params unchanged, except the current iteration count and gen count.

[1] Created with sphinx-autoapi

continue_iteration(*self*)
Checks whether the current iteration should be continued or not.

continue_evolution(*self*) → bool
Checks whether the current iteration should be continued or not.

check_FE_count(*self*) → bool

Checks whether termination criteria via function evaluation count has been met or not.

Returns True is function evaluation count limit NOT met.

Return type bool

manage_preferences(*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

requests(*self*) → Tuple

class desdeo_emo.EAs.BaseEA.**BaseDecompositionEA**(*problem: desdeo_problem.MOProblem*, *selection_operator: Type[desdeo_emo.selection.SelectionBase.SelectionBase] = None*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *a_priori: bool = False*, *interact: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *lattice_resolution: int = None*, *use_surrogates: bool = False*)

Bases: *BaseEA*

The Base class for decomposition based EAs.

This class contains most of the code to set up the parameters and operators. It also contains the logic of a simple decomposition EA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **selection_operator** (*Type[SelectionBase], optional*) – The selection operator to be used by the EA, by default None.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.

- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

_next_gen (*self*)
Run one generation of decomposition based EA. Intended to be used by next_iteration.

manage_preferences (*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

_select (*self*) → list
Describe a selection mechanism. Return indices of selected individuals.

Returns List of indices of individuals to be selected.

Return type list

request_plot (*self*) → desdeo_tools.interaction.SimplePlotRequest

request_preferences (*self*) → Union[None, Tuple[desdeo_tools.interaction.PreferredSolutionPreference, desdeo_tools.interaction.NonPreferredSolutionPreference, desdeo_tools.interaction.ReferencePointPreference, desdeo_tools.interaction.BoundPreference]]

requests (*self*) → Tuple

end (*self*)
Conducts non-dominated sorting at the end of the evolution process

Returns

The first element is a 2-D array of the decision vectors of the non-dominated solutions.
The second element is a 2-D array of the corresponding objective values.

Return type tuple

desdeo_emo.EAs.IOPIS

Module Contents

Classes

BaseIOPISDecompositionEA	The Base class for decomposition based EAs.
IOPIS_RVEA	The python version reference vector guided evolutionary algorithm.
IOPIS_NSGAIII	The Base class for decomposition based EAs.

class desdeo_emo.EAs.IOPIS.**BaseIOPISDecompositionEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *lattice_resolution: int = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *use_surrogates: bool = False*)

Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*, *desdeo_emo.EAs.BaseEA.BaseEA*

The Base class for decomposition based EAs.

This class contains most of the code to set up the parameters and operators. It also contains the logic of a simple decomposition EA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **selection_operator** (*Type[SelectionBase], optional*) – The selection operator to be used by the EA, by default None.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False

- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

manage_preferences(*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

request_preferences(*self*) → Union[None, desdeo_tools.interaction.ReferencePointPreference]

_select(*self*) → List
Describe a selection mechanism. Return indices of selected individuals.

Returns List of indices of individuals to be selected.

Return type list

class desdeo_emo.EAs.IOPIS.**IOPIS_RVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *alpha: float = None*, *lattice_resolution: int = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*, *use_surrogates: bool = False*)
Bases: *BaseIOPISDecompositionEA*, *desdeo_emo.EAs.RVEA.RVEA*

The python version reference vector guided evolutionary algorithm.

Most of the relevant code is contained in the super class. This class just assigns the APD selection operator to BaseDecompositionEA.

NOTE: The APD function had to be slightly modified to accomodate for the fact that this version of the algorithm is interactive, and does not have a set termination criteria. There is a time component in the APD penalty function formula of the type: (t/t_max)^alpha. As there is no set t_max, the formula has been changed. See below, the documentation for the argument: penalty_time_component

See the details of RVEA in the following paper

R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.

- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **alpha** (*float, optional*) – The alpha parameter in the APD selection mechanism. Read paper for details.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.
- **penalty_time_component** (*Union[str, float], optional*) – The APD formula had to be slightly changed. If penalty_time_component is a float between [0, 1], (t/t_max) is replaced by that constant for the entire algorithm. If penalty_time_component is "original", the original intent of the paper is followed and (t/t_max) is calculated as (current generation count/total number of generations). If penalty_time_component is "function_count", (t/t_max) is calculated as (current function evaluation count/total number of function evaluations) If penalty_time_component is "interactive", (t/t_max) is calculated as (Current gen count within an iteration/Total gen count within an iteration). Hence, time penalty is always zero at the beginning of each iteration, and one at the end of each iteration. Note: If the penalty_time_component ever exceeds one, the value one is used as the penalty_time_component. If no value is provided, an appropriate default is selected. If *interact* is true, penalty_time_component is "interactive" by default. If *interact* is false, but *total_function_evaluations* is provided, penalty_time_component is "function_count" by default. If *interact* is false, but *total_function_evaluations* is not provided, penalty_time_component is "original" by default.

_time_penalty_constant(*self*)
Returns the constant time penalty value.

_time_penalty_original(*self*)
Calculates the appropriate time penalty value, by the original formula.

_time_penalty_interactive(*self*)
Calculates the appropriate time penalty value.

_time_penalty_function_count(*self*)
Calculates the appropriate time penalty value.

class desdeo_emo.EAs.IOPIS.**IOPIS_NSGAIII**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *lattice_resolution: int = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *use_surrogates: bool = False*)

Bases: *BaseIOPISDecompositionEA*

The Base class for decomposition based EAs.

This class contains most of the code to set up the parameters and operators. It also contains the logic of a simple decomposition EA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **selection_operator** (*Type[SelectionBase], optional*) – The selection operator to be used by the EA, by default None.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

desdeo_emo.EAs.MOEAD

Module Contents

Classes

MOEA_D	Python implementation of MOEA/D

class desdeo_emo.EAs.MOEAD.**MOEA_D**(*problem: desdeo_problem.MOProblem*, *scalarization_function: desdeo_tools.scalarization.MOEADSF = PBI()*, *n_neighbors: int = 20*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *lattice_resolution: int = None*, *use_repair: bool = True*, *n_parents: int = 2*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*)

Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

Python implementation of MOEA/D

in IEEE Transactions on Evolutionary Computation, vol. 11, no. 6, pp. 712-731, Dec. 2007, doi: 10.1109/TEVC.2007.892759.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **scalarization_function** (*MOEADSF*) – The scalarization function to compare the solutions. Some implementations can be found in desdeo-tools/scalarization/MOEADSF. By default it uses the PBI function.
- **n_neighbors** (*int, optional*) – Number of reference vectors considered in the neighborhoods creation. The default number is 20.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **n_parents** (*int, optional*) – Number of individuals considered for the generation of offspring solutions. The default option is 2.
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **use_surrogates** (*bool, optional*) – A bool variable defining whether surrogate problems are to be used or not. By default False

- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

_next_gen (*self*)
Run one generation of decomposition based EA. Intended to be used by next_iteration.

_select (*self*, *current_neighborhood*, *offspring_fx*) → list
Describe a selection mechanism. Return indices of selected individuals.

Returns List of indices of individuals to be selected.

Return type list

desdeo_emo.EAs.NSGAIII

Module Contents

Classes

NSGAIII	Python Implementation of NSGA-III. Based on the pymoo package.

class desdeo_emo.EAs.NSGAIII.**NSGAIII**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *n_survive: int = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *lattice_resolution: int = None*, *selection_type: str = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*)
Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

Python Implementation of NSGA-III. Based on the pymoo package.

Most of the relevant code is contained in the super class. This class just assigns the NSGAIII selection operator to BaseDecompositionEA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the

output of one EA is to be used as the input of another.

- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **selection_type** (*str, optional*) – One of ["mean", "optimistic", "robust"]. To be used in data-driven optimization. To be used only with surrogate models which return an "uncertainity" factor. Using "mean" is equivalent to using the mean predicted values from the surrogate models and is the default case. Using "optimistic" results in using (mean - uncertainity) values from the the surrogate models as the predicted value (in case of minimization). It is (mean + uncertainity for maximization). Using "robust" is the opposite of using "optimistic".
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

desdeo_emo.EAs.PPGA

Module Contents

Classes

PPGA	Predatory-Prey genetic algorithm.
Lattice	The 2-dimensional toroidal lattice in which the predators and prey are placed.

class desdeo_emo.EAs.PPGA.**PPGA**(*problem*, *population_size: int = 100*, *population_params=None*, *initial_population=None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 10*, *predator_pop_size: int = 50*, *prey_max_moves: int = 10*, *prob_prey_move: float = 0.3*, *offspring_place_attempts: int = 10*, *kill_interval: int = 7*, *max_rank: int = 20*, *neighbourhood_radius: int = 3*)

Bases: *desdeo_emo.EAs.BaseEA.BaseEA*

Predatory-Prey genetic algorithm.

A population of prey signify the various models or solutions to the problem at hand. Weaker prey, i.e. bad models or solutions, are killed by predators. The predators and prey are placed in a lattice, in which they are free to roam.

In each generation, each predator gets a certain number of turns to move about and hunt in its neighbourhood, killing the weaker prey, according to a fitness criteria. After this, each prey gets a certain number of moves to

pursue a random walk and to reproduce with other prey. Each reproduction step generates two new prey from two parents, by crossing over their attributes and adding random mutations. After each prey has completed its move, the whole process starts again.

As the weaker individuals get eliminated in each generation, the population as a whole becomes more fit, i.e. the individuals get closer to the true pareto-optimal solutions.

If you have any questions about the code, please contact:

Bhupinder Saini: bhupinder.s.saini@jyu.fi Project researcher at University of Jyväskylä.

Parameters **population** (*object*) – The population object

Notes

The algorithm has been created earlier in MATLAB, and this Python implementation has been using that code as a basis. See references [4] for the study during which the original MATLAB version was created. Python code has been written by Niko Rissanen under the supervision of professor Nirupam Chakraborti.

For the MATLAB implementation, see: N. Chakraborti. Data-Driven Bi-Objective Genetic Algorithms EvoNN and BioGP and Their Applications in Metallurgical and Materials Domain. In Datta, Shubhabrata, Davim, J. Paulo (eds.), Computational Approaches to Materials Design: Theoretical and Practical Aspects, pp. 346-369, 2016.

References

[1] Laumanns, M., Rudolph, G., & Schwefel, H. P. (1998). A spatial predator-prey approach to multi-objective optimization: A preliminary study. In International Conference on Parallel Problem Solving from Nature (pp. 241-249). Springer, Berlin, Heidelberg.

[2] Li, X. (2003). A real-coded predator-prey genetic algorithm for multiobjective optimization. In International Conference on Evolutionary Multi-Criterion Optimization (pp. 207-221). Springer, Berlin, Heidelberg.

[3] Chakraborti, N. (2014). Strategies for evolutionary data driven modeling in chemical and metallurgical Systems. In Applications of Metaheuristics in Process Engineering (pp. 89-122). Springer, Cham.

[4] Pettersson, F., Chakraborti, N., & Saxén, H. (2007). A genetic algorithms based multi-objective neural net applied to noisy blast furnace data. Applied Soft Computing, 7(1), 387-397.

_next_gen (*self*)
Run one generation of PPGA.

Intended to be used by next_iteration.

Parameters **population** (*"Population"*) – Population object

select (*self*, *population*, *max_rank=20*) → list
Of the population, individuals lower than max_rank are selected. Return indices of selected individuals.

Parameters

- **population** (Population) – Contains the current population and problem information.
- **max_rank** (*int*) – Select only individuals lower than max_rank

Returns List of indices of individuals to be selected.

Return type list

manage_preferences(*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

class desdeo_emo.EAs.PPGA.**Lattice**(*size_x*, *size_y*, *population*, *predator_pop_size*, *target_pop_size*, *prob_prey_move*, *prey_max_moves*, *offspring_place_attempts*, *neighbourhood_radius*)
The 2-dimensional toroidal lattice in which the predators and prey are placed.

size_x
Width of the lattice.

Type int

size_y
Height of the lattice.

Type int

lattice
2d array for the lattice.

Type ndarray

predator_pop
The predator population.

Type ndarray

predators_loc
Location (x, y) of predators on the lattice.

Type list

preys_loc
Location (x, y) of preys on the lattice.

Type list

init_predators(*self*)
Initialize the predator population, linearly distributed in [0,1] and place them in the lattice randomly.

init_prey(*self*)
Find an empty position in the lattice and place the prey.

move_prey(*self*)
Find an empty position in prey neighbourhood for the prey to move in, and choose a mate for breeding if any available.

Returns **mating_pop** – List of parent indices to use for mating

Return type list

place_offspring(*self*, *offspring*)
Try to place the offsprings to the lattice. If no empty spot found within number of max attempts, do not place.

Parameters **offspring** (*int*) – number of offsprings

Returns Successfully placed offspring indices.

Return type list

move_predator(*self*)
Find an empty position in the predator neighbourhood for the predators to move in, move the predator and kill the weakest prey in its neighbourhood, if any. Repeat until > predator_max_moves.

update_lattice(*self*, *selected=None*)
Update prey positions in the lattice.

Parameters **selected** (*list*) – Indices of preys to be removed from the lattice.

static lattice_wrap_idx(*index*, *lattice_shape*)
Returns periodic lattice index for a given iterable index.

Parameters

- **index** (*tuple*) – one integer for each axis
- **lattice_shape** (*tuple*) – the shape of the lattice to index to

static neighbours(*arr*, *x*, *y*, *n=3*)
Given a 2D-array, returns an n*n array whose "center" element is arr[x,y]

Parameters

- **arr** (*ndarray*) – A 2D-array where to get the neighbouring cells
- **x** (*int*) – X coordinate for the center element
- **y** (*int*) – Y coordinate for the center element
- **n** (*int*) – Radius of the neighbourhood

Returns

- *The neighbouring cells of x, y in radius n*n.*
- *Defaults to Moore neighbourhood (n=3).*

desdeo_emo.EAs.RVEA

Module Contents

Classes

RVEA	The python version reference vector guided evolutionary algorithm.
oRVEA	Feature incorporated in the RVEA class using the "selection_type" argument.
robust_RVEA	Feature incorporated in the RVEA class using the "selection_type" argument.

class desdeo_emo.EAs.RVEA.**RVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *alpha: float = 2*, *lattice_resolution: int = None*, *selection_type: str = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*)
Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

The python version reference vector guided evolutionary algorithm.

Most of the relevant code is contained in the super class. This class just assigns the APD selection operator to BaseDecompositionEA.

NOTE: The APD function had to be slightly modified to accomodate for the fact that this version of the algorithm is interactive, and does not have a set termination criteria. There is a time component in the APD penalty function formula of the type: (t/t_max)^alpha. As there is no set t_max, the formula has been changed. See below, the documentation for the argument: penalty_time_component

See the details of RVEA in the following paper

R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (Population, *optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **alpha** (*float, optional*) – The alpha parameter in the APD selection mechanism. Read paper for details.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **selection_type** (*str, optional*) – One of ["mean", "optimistic", "robust"]. To be used in data-driven optimization. To be used only with surrogate models which return an "uncertainity" factor. Using "mean" is equivalent to using the mean predicted values from the surrogate models and is the default case. Using "optimistic" results in using (mean - uncertainity) values from the the surrogate models as the predicted value (in case of minimization). It is (mean + uncertainity for maximization). Using "robust" is the opposite of using "optimistic".
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.
- **penalty_time_component** (*Union[str, float], optional*) – The APD formula had to be slightly changed. If penalty_time_component is a float between [0, 1],

(t/t_max) is replaced by that constant for the entire algorithm. If penalty_time_component is "original", the original intent of the paper is followed and (t/t_max) is calculated as (current generation count/total number of generations). If penalty_time_component is "function_count", (t/t_max) is calculated as (current function evaluation count/total number of function evaluations) If penalty_time_component is "interactive", (t/t_max) is calculated as (Current gen count within an iteration/Total gen count within an iteration). Hence, time penalty is always zero at the beginning of each iteration, and one at the end of each iteration. Note: If the penalty_time_component ever exceeds one, the value one is used as the penalty_time_component. If no value is provided, an appropriate default is selected. If *interact* is true, penalty_time_component is "interactive" by default. If *interact* is false, but *total_function_evaluations* is provided, penalty_time_component is "function_count" by default. If *interact* is false, but *total_function_evaluations* is not provided, penalty_time_component is "original" by default.

_time_penalty_constant(*self*)
Returns the constant time penalty value.

_time_penalty_original(*self*)
Calculates the appropriate time penalty value, by the original formula.

_time_penalty_interactive(*self*)
Calculates the appropriate time penalty value.

_time_penalty_function_count(*self*)
Calculates the appropriate time penalty value.

class desdeo_emo.EAs.RVEA.**oRVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *alpha: float = 2*, *lattice_resolution: int = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*)

Bases: *RVEA*

Feature incorporated in the RVEA class using the "selection_type" argument. To be depreciated.

class desdeo_emo.EAs.RVEA.**robust_RVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *alpha: float = 2*, *lattice_resolution: int = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*)

Bases: *RVEA*

Feature incorporated in the RVEA class using the "selection_type" argument. To be depreciated.

desdeo_emo.EAs.TournamentEA

Module Contents

Classes

TournamentEA	This class provides the basic structure for Evolutionary algorithms.

class desdeo_emo.EAs.TournamentEA.**TournamentEA**(*problem*, *initial_population: desdeo_emo.population.Population.Population*, *n_gen_per_iter: int = 10*, *n_iterations: int = 10*, *tournament_size: int = 5*, *population_size: int = 500*)

Bases: *desdeo_emo.EAs.BaseEA.BaseEA*

This class provides the basic structure for Evolutionary algorithms.

_next_gen(*self*)

Run one generation of decomposition based EA.

This method leaves method.params unchanged. Intended to be used by next_iteration.

Parameters **population** (*"Population"*) – Population object

select(*self*) → list

Select parents for recombination using tournament selection. Chooses two parents, which are needed for crossover.

Returns **parents** – List of indices of individuals to be selected.

Return type list

desdeo_emo.EAs.slowRVEA

Legacy code. To be updated or depreciated.

Module Contents

Classes

slowRVEA	RVEA variant that impliments slow reference vector movement.

class desdeo_emo.EAs.slowRVEA.**slowRVEA**(*population: pyrvea.Population.Population.Population*, *ea_parameters*)

Bases: pyrvea.EAs.RVEA.RVEA

RVEA variant that impliments slow reference vector movement.

set_params(*self*, *population: pyrvea.Population.Population.Population*, *generations_per_iteration: int = 10*, *iterations: int = 10*, *Alpha: float = 2*, *ref_point: list = None*, *old_point: list = None*, ***kwargs*)

Set up the parameters. Save in RVEA.params. Note, this should be changed to align with the current structure.

Parameters

- **population** (Population) – Population object
- **Alpha** (*float*) – The alpha parameter of APD selection.
- **plotting** (*bool*) – Useless really.

_run_interruption(*self*, *population: pyrvea.Population.Population.Population*)

Package Contents

Classes

BaseEA	This class provides the basic structure for Evolutionary algorithms.
BaseDecompositionEA	The Base class for decomposition based EAs.
RVEA	The python version reference vector guided evolutionary algorithm.
NSGAIII	Python Implementation of NSGA-III. Based on the pymoo package.
PPGA	Predatory-Prey genetic algorithm.
TournamentEA	This class provides the basic structure for Evolutionary algorithms.
IOPIS_NSGAIII	The Base class for decomposition based EAs.
IOPIS_RVEA	The python version reference vector guided evolutionary algorithm.
MOEA_D	Python implementation of MOEA/D

class desdeo_emo.EAs.**BaseEA**(*a_priori: bool = False*, *interact: bool = False*, *selection_operator: Type[*desdeo_emo.selection.SelectionBase.SelectionBase*] = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *use_surrogates: bool = False*)

This class provides the basic structure for Evolutionary algorithms.

start(*self*)
Mimics the structure of the mcdm methods. Returns the request objects from self.retuests().

end(*self*)
To be run at the end of the evolution process.

_next_gen(*self*)
Run one generation of an EA. Change nothing about the parameters.

iterate(*self*, *preference=None*) → Tuple
Run one iteration of EA.

One iteration consists of a constant or variable number of generations. This method leaves EA.params unchanged, except the current iteration count and gen count.

continue_iteration(*self*)
Checks whether the current iteration should be continued or not.

continue_evolution(*self*) → bool
Checks whether the current iteration should be continued or not.

check_FE_count(*self*) → bool

Checks whether termination criteria via function evaluation count has been met or not.

Returns True is function evaluation count limit NOT met.

Return type bool

manage_preferences(*self*, *preference=None*)

Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

requests(*self*) → Tuple

class desdeo_emo.EAs.**BaseDecompositionEA**(*problem: desdeo_problem.MOProblem*, *selection_operator: Type[*desdeo_emo.selection.SelectionBase.SelectionBase*] = None*, *population_size: int = None*, *population_params: Dict = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *a_priori: bool = False*, *interact: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *lattice_resolution: int = None*, *use_surrogates: bool = False*)

Bases: *BaseEA*

The Base class for decomposition based EAs.

This class contains most of the code to set up the parameters and operators. It also contains the logic of a simple decomposition EA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **selection_operator** (*Type[SelectionBase], optional*) – The selection operator to be used by the EA, by default None.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False

- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

_next_gen (*self*)
Run one generation of decomposition based EA. Intended to be used by next_iteration.

manage_preferences (*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

_select (*self*) → list
Describe a selection mechanism. Return indices of selected individuals.

Returns List of indices of individuals to be selected.

Return type list

request_plot (*self*) → desdeo_tools.interaction.SimplePlotRequest

request_preferences (*self*) → Union[None, Tuple[desdeo_tools.interaction.PreferredSolutionPreference, desdeo_tools.interaction.NonPreferredSolutionPreference, desdeo_tools.interaction.ReferencePointPreference, desdeo_tools.interaction.BoundPreference]]

requests (*self*) → Tuple

end (*self*)
Conducts non-dominated sorting at the end of the evolution process

Returns

The first element is a 2-D array of the decision vectors of the non-dominated solutions.
The second element is a 2-D array of the corresponding objective values.

Return type tuple

class desdeo_emo.EAs.**RVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *alpha: float = 2*, *lattice_resolution: int = None*, *selection_type: str = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*)
Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

The python version reference vector guided evolutionary algorithm.

Most of the relevant code is contained in the super class. This class just assigns the APD selection operator to BaseDecompositionEA.

NOTE: The APD function had to be slightly modified to accomodate for the fact that this version of the algorithm is interactive, and does not have a set termination criteria. There is a time component in the APD penalty function formula of the type: (t/t_max)^alpha. As there is no set t_max, the formula has been changed. See below, the documentation for the argument: penalty_time_component

See the details of RVEA in the following paper

R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (Population, *optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **alpha** (*float, optional*) – The alpha parameter in the APD selection mechanism. Read paper for details.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **selection_type** (*str, optional*) – One of ["mean", "optimistic", "robust"]. To be used in data-driven optimization. To be used only with surrogate models which return an "uncertainity" factor. Using "mean" is equivalent to using the mean predicted values from the surrogate models and is the default case. Using "optimistic" results in using (mean - uncertainity) values from the the surrogate models as the predicted value (in case of minimization). It is (mean + uncertainity for maximization). Using "robust" is the opposite of using "optimistic".
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.
- **penalty_time_component** (*Union[str, float], optional*) – The APD formula had to be slightly changed. If penalty_time_component is a float between [0, 1], (t/t_max) is replaced by that constant for the entire algorithm. If penalty_time_component is "original", the original intent of the paper is followed and (t/t_max) is calculated as (current generation count/total number of generations). If penalty_time_component is "function_count", (t/t_max) is calculated as (current function evaluation count/total number of function evaluations) If penalty_time_component is "interactive", (t/t_max) is calculated as (Current gen count within an iteration/Total gen count within an iteration). Hence, time penalty is always zero at the beginning of each iteration, and one at the end of each iteration. Note: If the penalty_time_component ever exceeds one, the value one is used

as the penalty_time_component. If no value is provided, an appropriate default is selected. If *interact* is true, penalty_time_component is "interactive" by default. If *interact* is false, but *total_function_evaluations* is provided, penalty_time_component is "function_count" by default. If *interact* is false, but *total_function_evaluations* is not provided, penalty_time_component is "original" by default.

_time_penalty_constant(*self*)
Returns the constant time penalty value.

_time_penalty_original(*self*)
Calculates the appropriate time penalty value, by the original formula.

_time_penalty_interactive(*self*)
Calculates the appropriate time penalty value.

_time_penalty_function_count(*self*)
Calculates the appropriate time penalty value.

class desdeo_emo.EAs.**NSGAIII**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *n_survive: int = None*, *initial_population:* desdeo_emo.population.Population.Population *= None*, *lattice_resolution: int = None*, *selection_type: str = None*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*)
Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

Python Implementation of NSGA-III. Based on the pymoo package.

Most of the relevant code is contained in the super class. This class just assigns the NSGAIII selection operator to BaseDecompositionEA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (Population, *optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **selection_type** (*str, optional*) – One of ["mean", "optimistic", "robust"]. To be used in data-driven optimization. To be used only with surrogate models which return an "uncertainity" factor. Using "mean" is equivalent to using the mean predicted values from the surrogate models and is the default case. Using "optimistic" results in using (mean - uncertainity) values from the the surrogate models as the predicted value (in case of minimization). It is (mean + uncertainity for maximization). Using "robust" is the opposite of using "optimistic".
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False

- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

class desdeo_emo.EAs.**PPGA**(*problem*, *population_size: int = 100*, *population_params=None*, *initial_population=None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 10*, *predator_pop_size: int = 50*, *prey_max_moves: int = 10*, *prob_prey_move: float = 0.3*, *offspring_place_attempts: int = 10*, *kill_interval: int = 7*, *max_rank: int = 20*, *neighbourhood_radius: int = 3*)

Bases: *desdeo_emo.EAs.BaseEA.BaseEA*

Predatory-Prey genetic algorithm.

A population of prey signify the various models or solutions to the problem at hand. Weaker prey, i.e. bad models or solutions, are killed by predators. The predators and prey are placed in a lattice, in which they are free to roam.

In each generation, each predator gets a certain number of turns to move about and hunt in its neighbourhood, killing the weaker prey, according to a fitness criteria. After this, each prey gets a certain number of moves to pursue a random walk and to reproduce with other prey. Each reproduction step generates two new prey from two parents, by crossing over their attributes and adding random mutations. After each prey has completed its move, the whole process starts again.

As the weaker individuals get eliminated in each generation, the population as a whole becomes more fit, i.e. the individuals get closer to the true pareto-optimal solutions.

If you have any questions about the code, please contact:

Bhupinder Saini: bhupinder.s.saini@jyu.fi Project researcher at University of Jyväskylä.

Parameters **population** (*object*) – The population object

Notes

The algorithm has been created earlier in MATLAB, and this Python implementation has been using that code as a basis. See references [4] for the study during which the original MATLAB version was created. Python code has been written by Niko Rissanen under the supervision of professor Nirupam Chakraborti.

For the MATLAB implementation, see: N. Chakraborti. Data-Driven Bi-Objective Genetic Algorithms EvoNN and BioGP and Their Applications in Metallurgical and Materials Domain. In Datta, Shubhabrata, Davim, J. Paulo (eds.), Computational Approaches to Materials Design: Theoretical and Practical Aspects, pp. 346-369, 2016.

References

[1] Laumanns, M., Rudolph, G., & Schwefel, H. P. (1998). A spatial predator-prey approach to multi-objective optimization: A preliminary study. In International Conference on Parallel Problem Solving from Nature (pp. 241-249). Springer, Berlin, Heidelberg.

[2] Li, X. (2003). A real-coded predator-prey genetic algorithm for multiobjective optimization. In International Conference on Evolutionary Multi-Criterion Optimization (pp. 207-221). Springer, Berlin, Heidelberg.

[3] Chakraborti, N. (2014). Strategies for evolutionary data driven modeling in chemical and metallurgical Systems. In Applications of Metaheuristics in Process Engineering (pp. 89-122). Springer, Cham.

[4] Pettersson, F., Chakraborti, N., & Saxén, H. (2007). A genetic algorithms based multi-objective neural net applied to noisy blast furnace data. Applied Soft Computing, 7(1), 387-397.

_next_gen (*self*)
Run one generation of PPGA.

Intended to be used by next_iteration.

Parameters **population** (*"Population"*) – Population object

select (*self*, *population*, *max_rank=20*) → list
Of the population, individuals lower than max_rank are selected. Return indices of selected individuals.

Parameters

- **population** (Population) – Contains the current population and problem information.
- **max_rank** (*int*) – Select only individuals lower than max_rank

Returns List of indices of individuals to be selected.

Return type list

manage_preferences (*self*, *preference=None*)
Run the interruption phase of EA.

Use this phase to make changes to RVEA.params or other objects. Updates Reference Vectors (adaptation), conducts interaction with the user.

class desdeo_emo.EAs.**TournamentEA**(*problem*, *initial_population: desdeo_emo.population.Population.Population*, *n_gen_per_iter: int = 10*, *n_iterations: int = 10*, *tournament_size: int = 5*, *population_size: int = 500*)
Bases: *desdeo_emo.EAs.BaseEA.BaseEA*

This class provides the basic structure for Evolutionary algorithms.

_next_gen (*self*)
Run one generation of decomposition based EA.

This method leaves method.params unchanged. Intended to be used by next_iteration.

Parameters **population** (*"Population"*) – Population object

select (*self*) → list
Select parents for recombination using tournament selection. Chooses two parents, which are needed for crossover.

Returns **parents** – List of indices of individuals to be selected.

Return type list

class desdeo_emo.EAs.**IOPIS_NSGAIII**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *lattice_resolution: int = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *use_surrogates: bool = False*)

Bases: BaseIOPISDecompositionEA

The Base class for decomposition based EAs.

This class contains most of the code to set up the parameters and operators. It also contains the logic of a simple decomposition EA.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **selection_operator** (*Type[SelectionBase], optional*) – The selection operator to be used by the EA, by default None.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

class desdeo_emo.EAs.**IOPIS_RVEA**(*problem: desdeo_problem.MOProblem*, *population_size: int = None*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *alpha: float = None*, *lattice_resolution: int = None*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*, *time_penalty_component: Union[str, float] = None*, *use_surrogates: bool = False*)

Bases: BaseIOPISDecompositionEA, *desdeo_emo.EAs.RVEA.RVEA*

The python version reference vector guided evolutionary algorithm.

Most of the relevant code is contained in the super class. This class just assigns the APD selection operator to BaseDecompositionEA.

NOTE: The APD function had to be slightly modified to accomodate for the fact that this version of the algorithm is interactive, and does not have a set termination criteria. There is a time component in the APD penalty function formula of the type: (t/t_max)^alpha. As there is no set t_max, the formula has been changed. See below, the documentation for the argument: penalty_time_component

See the details of RVEA in the following paper

R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **population_size** (*int, optional*) – The desired population size, by default None, which sets up a default value of population size depending upon the dimensionaly of the problem.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (Population, *optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **alpha** (*float, optional*) – The alpha parameter in the APD selection mechanism. Read paper for details.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.
- **penalty_time_component** (*Union[str, float], optional*) – The APD formula had to be slightly changed. If penalty_time_component is a float between [0, 1], (t/t_max) is replaced by that constant for the entire algorithm. If penalty_time_component is "original", the original intent of the paper is followed and (t/t_max) is calculated as (current generation count/total number of generations). If penalty_time_component is "function_count", (t/t_max) is calculated as (current function evaluation count/total number of function evaluations) If penalty_time_component is "interactive", (t/t_max) is calculated as (Current gen count within an iteration/Total gen count within an iteration). Hence, time penalty is always zero at the beginning of each iteration, and one at the end of each iteration. Note: If the penalty_time_component ever exceeds one, the value one is used

as the penalty_time_component. If no value is provided, an appropriate default is selected. If *interact* is true, penalty_time_component is "interactive" by default. If *interact* is false, but *total_function_evaluations* is provided, penalty_time_component is "function_count" by default. If *interact* is false, but *total_function_evaluations* is not provided, penalty_time_component is "original" by default.

_time_penalty_constant(*self*)
Returns the constant time penalty value.

_time_penalty_original(*self*)
Calculates the appropriate time penalty value, by the original formula.

_time_penalty_interactive(*self*)
Calculates the appropriate time penalty value.

_time_penalty_function_count(*self*)
Calculates the appropriate time penalty value.

class desdeo_emo.EAs.**MOEA_D**(*problem: desdeo_problem.MOProblem*, *scalarization_function: desdeo_tools.scalarization.MOEADSF = PBI()*, *n_neighbors: int = 20*, *population_params: Dict = None*, *initial_population: desdeo_emo.population.Population.Population = None*, *lattice_resolution: int = None*, *use_repair: bool = True*, *n_parents: int = 2*, *a_priori: bool = False*, *interact: bool = False*, *use_surrogates: bool = False*, *n_iterations: int = 10*, *n_gen_per_iter: int = 100*, *total_function_evaluations: int = 0*)

Bases: *desdeo_emo.EAs.BaseEA.BaseDecompositionEA*

Python implementation of MOEA/D

in IEEE Transactions on Evolutionary Computation, vol. 11, no. 6, pp. 712-731, Dec. 2007, doi: 10.1109/TEVC.2007.892759.

Parameters

- **problem** (*MOProblem*) – The problem class object specifying the details of the problem.
- **scalarization_function** (*MOEADSF*) – The scalarization function to compare the solutions. Some implementations can be found in desdeo-tools/scalarization/MOEADSF. By default it uses the PBI function.
- **n_neighbors** (*int, optional*) – Number of reference vectors considered in the neighborhoods creation. The default number is 20.
- **population_params** (*Dict, optional*) – The parameters for the population class, by default None. See desdeo_emo.population.Population for more details.
- **initial_population** (*Population, optional*) – An initial population class, by default None. Use this if you want to set up a specific starting population, such as when the output of one EA is to be used as the input of another.
- **lattice_resolution** (*int, optional*) – The number of divisions along individual axes in the objective space to be used while creating the reference vector lattice by the simplex lattice design. By default None
- **n_parents** (*int, optional*) – Number of individuals considered for the generation of offspring solutions. The default option is 2.
- **a_priori** (*bool, optional*) – A bool variable defining whether a priori preference is to be used or not. By default False
- **interact** (*bool, optional*) – A bool variable defining whether interactive preference is to be used or not. By default False

- **use_surrogates** (*bool, optional*) – A bool variable defining whether surrogate problems are to be used or not. By default False
- **n_iterations** (*int, optional*) – The total number of iterations to be run, by default 10. This is not a hard limit and is only used for an internal counter.
- **n_gen_per_iter** (*int, optional*) – The total number of generations in an iteration to be run, by default 100. This is not a hard limit and is only used for an internal counter.
- **total_function_evaluations** (*int, optional*) – Set an upper limit to the total number of function evaluations. When set to zero, this argument is ignored and other termination criteria are used.

_next_gen (*self*)
Run one generation of decomposition based EA. Intended to be used by next_iteration.

_select (*self*, *current_neighborhood*, *offspring_fx*) → list
Describe a selection mechanism. Return indices of selected individuals.

Returns List of indices of individuals to be selected.

Return type list

desdeo_emo.population

This module provides classes and methods which implement populations in an EA.

Submodules

desdeo_emo.population.CreateIndividuals

Module Contents

Functions

create_new_individuals(design, problem, pop_size=None)	Create new individuals to the population.

desdeo_emo.population.CreateIndividuals.**create_new_individuals** (*design*, *problem*, *pop_size=None*)
Create new individuals to the population.

The individuals can be created randomly, by LHS design, or can be passed by the user.

Design does not apply in case of EvoNN and EvoDN2 problem, where neural networks are created as individuals.

Parameters

- **design** (*str, optional*) – Describe the method of creation of new individuals. "RandomDesign" creates individuals randomly. "LHSDesign" creates individuals using Latin hypercube sampling. "EvoNN" creates Artificial Neural Networks as individuals. "EvoDN2" creates Deep Neural Networks.
- **problem** (*baseProblem*) – An object of the class Problem

- **pop_size** (*int, optional*) – Number of individuals in the population. If none, some default population size based on number of objectives is chosen.

Returns **individuals** – A list of individuals.

Return type list

desdeo_emo.population.Population

Module Contents

Classes

BasePopulation	Helper class that provides a standard way to create an ABC using
Population	Helper class that provides a standard way to create an ABC using

class desdeo_emo.population.Population.**BasePopulation**(*problem: desdeo_problem.MOProblem*, *pop_size: int*, *pop_params: Dict = None*)

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

property ideal_objective_vector(*self*) → numpy.ndarray

property ideal_fitness_val(*self*) → numpy.ndarray

abstract add(*self*, *offsprings: Union[List, numpy.ndarray]*) → List

Evaluate and add offspring to the population.

Parameters **offsprings** (*Union[List, np.ndarray]*) – List or array of individuals to be evaluated and added to the population.

Returns Indices of the evaluated individuals

Return type List

abstract keep(*self*, *indices: List*)

Save the population members given by the list of indices for the next generation. Delete the rest.

Parameters **indices** (*List*) –

List of indices of the population members to be kept for the next generation.

abstract delete(*self*, *indices: List*)

Delete the population members given by the list of indices for the next generation. Keep the rest.

Parameters **indices** (*List*) – List of indices of the population members to be deleted.

abstract mate(*self*, *mating_individuals: List = None*, *params: Dict = None*) → Union[List, numpy.ndarray]

Perform crossover and mutation over the population members.

Parameters

- **mating_individuals** (*List, optional*) –

  List of individuals taking part in recombination. By default None, which recombinated all individuals in random order.

- **params** (*Dict, optional*) – Parameters for the mutation or crossover operator, by default None.

Returns The offspring population

Return type Union[List, np.ndarray]

class desdeo_emo.population.Population.**Population**(*problem: desdeo_problem.MOProblem*, *pop_size: int*, *pop_params: Dict = None*, *use_surrogates: bool = False*)

Bases: *BasePopulation*

Helper class that provides a standard way to create an ABC using inheritance.

add(*self*, *offsprings: Union[List, numpy.ndarray]*, *use_surrogates: bool = False*) → List
Evaluate and add offspring to the population.

Parameters

- **offsprings** (*Union[List, np.ndarray]*) – List or array of individuals to be evaluated and added to the population.
- **use_surrogates** (*bool*) – If true, use surrogate models rather than true function evaluations.
- **use_surrogates** – If true, use surrogate models rather than true function evaluations.

Returns Indices of the evaluated individuals

Return type List

keep(*self*, *indices: List*)

Save the population members given by the list of indices for the next generation. Delete the rest.

Parameters **indices** (*List*) –

List of indices of the population members to be kept for the next generation.

delete(*self*, *indices: List*)

Delete the population members given by the list of indices for the next generation. Keep the rest.

Parameters **indices** (*List*) – List of indices of the population members to be deleted.

mate(*self*, *mating_individuals: List = None*) → Union[List, numpy.ndarray]
Perform crossover and mutation over the population members.

Parameters

- **mating_individuals** (*List, optional*) –

  List of individuals taking part in recombination. By default None, which recombinated all individuals in random order.

- **params** (*Dict, optional*) – Parameters for the mutation or crossover operator, by default None.

Returns The offspring population

Return type Union[List, np.ndarray]

update_ideal(*self*)

replace(*self*, *indices: List*, *individual: numpy.ndarray*, *evaluation: tuple*)

Replace the population members given by the list of indices by the given individual and its evaluation.
Keep the rest of the population unchanged.

Parameters

- **indices** (*List*) – List of indices of the population members to be replaced.
- **individual** (*np.ndarray*) – Decision variables of the individual that will replace the positions given in the list.
- **evaluation** (*tuple*) – Result of the evaluation of the objective function, constraints, etc. obtained using the evaluate method.

repair(*self*, *individual*)
Repair the variables of an individual which are not in the boundary defined by the problem :param individual: Decision variables of the individual.

Returns

Return type The new decision vector with the variables in the boundary defined by the problem

reevaluate_fitness(*self*)

non_dominated_fitness(*self*)

non_dominated_objectives(*self*)

desdeo_emo.population.Population_old

Module Contents

Classes

Population	Define the population.

class desdeo_emo.population.Population_old.**Population**(*problem: desdeo_problem.MOProblem*, *assign_type: str = 'RandomDesign'*, *pop_size=None*, *recombination_type=None*, *crossover_type='simulated_binary_crossover'*, *mutation_type='bounded_polynomial_mutation'*, **args*)

Define the population.

add(*self*, *new_pop: list*)
Evaluate and add individuals to the population. Update ideal and nadir point.

Parameters **new_pop** (*list*) – Decision variable values for new population.

append_individual (*self*, *ind: numpy.ndarray*)
Evaluate and add individual to the population.

Parameters **ind** (*np.ndarray*) –

evaluate_individual (*self*, *ind: numpy.ndarray*)
Evaluate individual.

Returns objective values, constraint violation, and fitness.

Parameters **ind** (*np.ndarray*) –

eval_fitness (*self*, *obj*)
Calculate fitness based on objective values. Fitness = obj if minimized.

update_fitness (*self*)
Include or exclude objectives from fitness calculation. Problem.minimize should be a list of booleans of same length as the number of objectives.

delete (*self*, *indices*, *preserve=False*)
Remove from population individuals which are in indices if preserve=False, otherwise preserve them and remove all others.

Parameters

- **indices** (*array_like*) – Indices of individuals to keep or delete.
- **preserve** (*bool*) – Whether to delete individuals at indices from current population, or preserve them and delete others.

evolve (*self*, *EA: BaseEA = None*, *ea_parameters: dict = None*)
Evolve the population with interruptions.

Evolves the population based on the EA sent by the user.

Parameters

- **EA** (*"BaseEA"*) – Should be a derivative of BaseEA (Default value = None)
- **ea_parameters** (*dict*) – Contains the parameters needed by EA (Default value = None)

mate (*self*, *mating_pop=None*, *params=None*)
Conduct crossover and mutation over the population.

plot_init_ (*self*)
Initialize animation object. Return figure

plot_objectives (*self*, *iteration: int = None*)
Plot the objective values of individuals.

Parameters **iteration** (*int*) – Iteration count.

plot_pareto (*self*, *name*, *show_all=False*)
Plot the pareto front. REMOVE THIS IN THE FUTURE.

Parameters

- **name** (*str*) – Name to append to the plot filename.
- **show_all** (*bool*) – Show all solutions, including those not on the pareto front.

hypervolume (*self*, *ref_point*)
Calculate hypervolume. Uses package pygmo. Add checks to prevent errors.

Parameters **ref_point** –

non_dominated(*self*)
Fix this. check if nd2 and nds mean the same thing

update_ideal_and_nadir(*self*, *new_objective_vals: list = None*)
Updates self.ideal and self.nadir in the fitness space.

Uses the entire population if new_objective_vals is none.

Parameters **new_objective_vals** (*list, optional*) – Objective values for a newly added individual (the default is None, which calculated the ideal and nadir for the entire population.)

desdeo_emo.population.SurrogatePopulation

Module Contents

Classes

SurrogatePopulation	Helper class that provides a standard way to create an ABC using

class desdeo_emo.population.SurrogatePopulation.**SurrogatePopulation**(*problem*, *pop_size: int*, *initial_pop*, *xover*, *mutation*, *recombination*)

Bases: *desdeo_emo.population.Population.Population*, *desdeo_emo.population.Population.BasePopulation*

Helper class that provides a standard way to create an ABC using inheritance.

Package Contents

Classes

Population	Helper class that provides a standard way to create an ABC using
SurrogatePopulation	Helper class that provides a standard way to create an ABC using

Functions

create_new_individuals(design, problem, pop_size=None)	Create new individuals to the population.

desdeo_emo.population.**create_new_individuals**(*design*, *problem*, *pop_size=None*)

Create new individuals to the population.

The individuals can be created randomly, by LHS design, or can be passed by the user.

Design does not apply in case of EvoNN and EvoDN2 problem, where neural networks are created as individuals.

Parameters

- **design** (*str, optional*) – Describe the method of creation of new individuals. "RandomDesign" creates individuals randomly. "LHSDesign" creates individuals using Latin hypercube sampling. "EvoNN" creates Artificial Neural Networks as individuals. "EvoDN2" creates Deep Neural Networks.
- **problem** (*baseProblem*) – An object of the class Problem
- **pop_size** (*int, optional*) – Number of individuals in the population. If none, some default population size based on number of objectives is chosen.

Returns **individuals** – A list of individuals.

Return type list

class desdeo_emo.population.**Population**(*problem: desdeo_problem.MOProblem*, *pop_size: int*, *pop_params: Dict = None*, *use_surrogates: bool = False*)

Bases: *BasePopulation*

Helper class that provides a standard way to create an ABC using inheritance.

add(*self*, *offsprings: Union[List, numpy.ndarray]*, *use_surrogates: bool = False*) → List

Evaluate and add offspring to the population.

Parameters

- **offsprings** (*Union[List, np.ndarray]*) – List or array of individuals to be evaluated and added to the population.
- **use_surrogates** (*bool*) – If true, use surrogate models rather than true function evaluations.
- **use_surrogates** – If true, use surrogate models rather than true function evaluations.

Returns Indices of the evaluated individuals

Return type List

keep(*self*, *indices: List*)

Save the population members given by the list of indices for the next generation. Delete the rest.

Parameters **indices** (*List*) –

List of indices of the population members to be kept for the next generation.

delete(*self*, *indices: List*)

Delete the population members given by the list of indices for the next generation. Keep the rest.

Parameters **indices** (*List*) – List of indices of the population members to be deleted.

mate(*self*, *mating_individuals: List = None*) → Union[List, numpy.ndarray]
Perform crossover and mutation over the population members.

Parameters

- **mating_individuals** (*List, optional*) –

  List of individuals taking part in recombination. By default None, which recombinated all individuals in random order.

- **params** (*Dict, optional*) – Parameters for the mutation or crossover operator, by default None.

Returns The offspring population

Return type Union[List, np.ndarray]

update_ideal(*self*)

replace(*self*, *indices: List*, *individual: numpy.ndarray*, *evaluation: tuple*)

Replace the population members given by the list of indices by the given individual and its evaluation. Keep the rest of the population unchanged.

Parameters

- **indices** (*List*) – List of indices of the population members to be replaced.
- **individual** (*np.ndarray*) – Decision variables of the individual that will replace the positions given in the list.
- **evaluation** (*tuple*) – Result of the evaluation of the objective function, constraints, etc. obtained using the evaluate method.

repair(*self*, *individual*)
Repair the variables of an individual which are not in the boundary defined by the problem :param individual: Decision variables of the individual.

Returns

Return type The new decision vector with the variables in the boundary defined by the problem

reevaluate_fitness(*self*)

non_dominated_fitness(*self*)

non_dominated_objectives(*self*)

class desdeo_emo.population.**SurrogatePopulation**(*problem*, *pop_size: int*, *initial_pop*, *xover*, *mutation*, *recombination*)
Bases: *desdeo_emo.population.Population.Population*, *desdeo_emo.population.Population.BasePopulation*

Helper class that provides a standard way to create an ABC using inheritance.

desdeo_emo.recombination

This module provides implementations of various crossover and mutation operators.

Submodules

desdeo_emo.recombination.BoundedPolynomialMutation

Module Contents

Classes

BP_mutation	

class desdeo_emo.recombination.BoundedPolynomialMutation.**BP_mutation**(*lower_limits: numpy.ndarray*, *upper_limits: numpy.ndarray*, *ProM: float = None*, *DisM: float = 20*)

do(*self*, *offspring: numpy.ndarray*)

Conduct bounded polynomial mutation. Return the mutated individuals.

Parameters **offspring** (*np.ndarray*) – The array of offsprings to be mutated.

Returns The mutated offsprings

Return type np.ndarray

desdeo_emo.recombination.SimulatedBinaryCrossover

Module Contents

Classes

SBX_xover	Simulated binary crossover.

class desdeo_emo.recombination.SimulatedBinaryCrossover.**SBX_xover**(*ProC: float = 1*, *DisC: float = 30*)

Simulated binary crossover.

Parameters

ProC [float, optional] [description], by default 1

DisC [float, optional] [description], by default 30

do(*self*, *pop: numpy.ndarray*, *mating_pop_ids: list = None*) → numpy.ndarray

Consecutive members of mating_pop_ids are crossed over in pairs. Example: if mating_pop_ids = [0, 2, 3, 6, 5] then the individuals are crossover as: [0, 2], [3, 6], [5, 0]. Note: if the number of elements is odd, the last individual is crossed over with the first one.

Parameters

- **pop** (*np.ndarray*) – Array of all individuals
- **mating_pop_ids** (*list, optional*) –

  Indices of population members to mate, by default None, which shuffles and mates whole population

Returns The offspring produced as a result of crossover.

Return type np.ndarray

desdeo_emo.recombination.biogp_mutation

Module Contents

Classes

BioGP_mutation	

Functions

mutate(offspring, individuals, params, *args)	Perform BioGP mutation functions.

desdeo_emo.recombination.biogp_mutation.**mutate**(*offspring*, *individuals*, *params*, **args*)

Perform BioGP mutation functions.

Standard mutation: Randomly select and regrow a subtree of an individual.

Small mutation: Randomly select a node within a tree and replace it with either a function of the same arity, or another value from the terminal set.

Mono parental: Randomly swap two subtrees within the same individual.

Parameters

- **offspring** (*list*) – List of individuals to mutate.
- **individuals** (*list*) – List of all individuals.
- **params** (*dict*) – Parameters for breeding. If None, use defaults.

class desdeo_emo.recombination.biogp_mutation.**BioGP_mutation**(*probability_mutation: float*)

do(*self*, *offspring*)

desdeo_emo.recombination.biogp_xover

Module Contents

Classes

BioGP_xover	

Functions

mate(mating_pop, individuals: list, params)	Perform BioGP crossover functions. Produce two offsprings by swapping genetic

desdeo_emo.recombination.biogp_xover.**mate**(*mating_pop*, *individuals: list*, *params*)

Perform BioGP crossover functions. Produce two offsprings by swapping genetic material of the two parents.

Standard crossover: Swap two random subtrees between the parents.

Height-fair crossover: Swap two random subtrees between the parents at the selected depth.

Parameters

- **mating_pop** (*list*) – List of indices of individuals to mate. If None, choose from population randomly. Each entry should contain two indices, one for each parent.
- **individuals** (*list*) – List of all individuals.
- **params** (*dict*) – Parameters for evolution. If None, use defaults.

Returns **offspring** – The offsprings produced as a result of crossover.

Return type list

class desdeo_emo.recombination.biogp_xover.**BioGP_xover**(*probability_crossover: float = 0.9*, *probability_standard: float = 0.5*)

do(*self*, *pop*, *mating_pop_ids*)

desdeo_emo.recombination.evodn2_xover_mutation

Module Contents

Classes

EvoDN2Recombination	

Functions

mate(mating_pop, individuals: list, params, crossover_type=None, mutation_type=None)	Swap nodes between two partners and mutate based on standard deviation.

desdeo_emo.recombination.evodn2_xover_mutation.**mate**(*mating_pop*, *individuals: list*, *params*, *crossover_type=None*, *mutation_type=None*)

Swap nodes between two partners and mutate based on standard deviation.

Parameters

- **mating_pop** (*list*) – List of indices of individuals to mate. If None, choose from population randomly. Each entry should contain two indices, one for each parent.
- **individuals** (*list*) – List of all individuals.
- **params** (*dict*) – Parameters for evolution. If None, use defaults.

Returns **offspring** – The offsprings produced as a result of crossover and mutation.

Return type list

class desdeo_emo.recombination.evodn2_xover_mutation.**EvoDN2Recombination**(*evolver: BaseEA*, *ProC: float = 0.8*, *ProM: float = 0.3*, *mutation_strength: float = 1.0*)

do(*self*, *pop*, *mating_pop_ids: list = None*)

desdeo_emo.recombination.evonn_xover_mutation

Module Contents

Classes

EvoNNRecombination	

Functions

mate(mating_pop, individuals: list, params, crossover_type=None, mutation_type=None)	Swap nodes between two partners and mutate based on standard deviation.

desdeo_emo.recombination.evonn_xover_mutation.**mate**(*mating_pop*, *individuals: list*, *params*, *crossover_type=None*, *mutation_type=None*)

Swap nodes between two partners and mutate based on standard deviation.

Parameters

- **mating_pop** (*list*) – List of indices of individuals to mate. If None, choose from population randomly. Each entry should contain two indices, one for each parent.
- **individuals** (*list*) – List of all individuals.
- **params** (*dict*) – Parameters for evolution. If None, use defaults.

Returns **offspring** – The offsprings produced as a result of crossover and mutation.

Return type list

class desdeo_emo.recombination.evonn_xover_mutation.**EvoNNRecombination**(*evolver: BaseEA*, *ProC: float = 0.8*, *ProM: float = 0.3*, *mutation_strength: float = 1.0*, *mutation_type: str = 'gaussian'*)

do(*self*, *pop*, *mating_pop_ids: list = None*)

Package Contents

Classes

BioGP_mutation	
BioGP_xover	
BP_mutation	
EvoDN2Recombination	
EvoNNRecombination	
SBX_xover	Simulated binary crossover.

class desdeo_emo.recombination.**BioGP_mutation**(*probability_mutation: float*)

do(*self*, *offspring*)

class desdeo_emo.recombination.**BioGP_xover**(*probability_crossover: float = 0.9*, *probability_standard: float = 0.5*)

do(*self*, *pop*, *mating_pop_ids*)

class desdeo_emo.recombination.**BP_mutation**(*lower_limits: numpy.ndarray*, *upper_limits: numpy.ndarray*, *ProM: float = None*, *DisM: float = 20*)

do(*self*, *offspring: numpy.ndarray*)

Conduct bounded polynomial mutation. Return the mutated individuals.

Parameters **offspring** (*np.ndarray*) – The array of offsprings to be mutated.

Returns The mutated offsprings

Return type np.ndarray

class desdeo_emo.recombination.**EvoDN2Recombination**(*evolver: BaseEA*, *ProC: float = 0.8*, *ProM: float = 0.3*, *mutation_strength: float = 1.0*)

do(*self*, *pop*, *mating_pop_ids: list = None*)

class desdeo_emo.recombination.**EvoNNRecombination**(*evolver: BaseEA*, *ProC: float = 0.8*, *ProM: float = 0.3*, *mutation_strength: float = 1.0*, *mutation_type: str = 'gaussian'*)

do(*self*, *pop*, *mating_pop_ids: list = None*)

class desdeo_emo.recombination.**SBX_xover**(*ProC: float = 1*, *DisC: float = 30*)

Simulated binary crossover.

Parameters

ProC [float, optional] [description], by default 1

DisC [float, optional] [description], by default 30

do(*self*, *pop: numpy.ndarray*, *mating_pop_ids: list = None*) → numpy.ndarray

Consecutive members of mating_pop_ids are crossed over in pairs. Example: if mating_pop_ids = [0, 2, 3, 6, 5] then the individuals are crossover as: [0, 2], [3, 6], [5, 0]. Note: if the number of elements is odd, the last individual is crossed over with the first one.

Parameters

- **pop** (*np.ndarray*) – Array of all individuals
- **mating_pop_ids** (*list, optional*) –

  Indices of population members to mate, by default None, which shuffles and mates whole population

Returns The offspring produced as a result of crossover.

Return type np.ndarray

desdeo_emo.selection

This module provides implementations of various selection operators.

Submodules

desdeo_emo.selection.APD_Select

Module Contents

Classes

APD_Select	The selection operator for the RVEA algorithm. Read the following paper for more

class desdeo_emo.selection.APD_Select.**APD_Select**(*pop:* desdeo_emo.population.Population.Population, *time_penalty_function: Callable*, *alpha: float = 2*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the RVEA algorithm. Read the following paper for more details. R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.
- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop:* desdeo_emo.population.Population.Population, *vectors:* desdeo_emo.utilities.ReferenceVectors.ReferenceVectors) → List[int]
Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

desdeo_emo.selection.APD_Select_constraints

Module Contents

Classes

APD_Select	The selection operator for the RVEA algorithm. Read the following paper for more

class desdeo_emo.selection.APD_Select_constraints.**APD_Select**(*pop:* desdeo_emo.population.Population.Population, *time_penalty_function: Callable*, *alpha: float = 2*, *selection_type: str = None*)
Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the RVEA algorithm. Read the following paper for more details. R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.
- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop:* desdeo_emo.population.Population.Population, *vectors:* desdeo_emo.utilities.ReferenceVectors.ReferenceVectors) → List[int]
Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

_calculate_fitness(*self*, *pop*) → numpy.ndarray

desdeo_emo.selection.IOPIS_APD

Module Contents

Classes

IOPIS_APD_Select	The selection operator for the IOPIS/RVEA algorithm.

class desdeo_emo.selection.IOPIS_APD.**IOPIS_APD_Select**(*time_penalty_function: Callable*, *scalarization_methods: List*, *alpha: float = 2*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the IOPIS/RVEA algorithm.

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.
- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*, *reference_point: numpy.ndarray*) → List[int]

Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

desdeo_emo.selection.IOPIS_NSGAIII

Module Contents

Classes

IOPIS_NSGAIII_select	The NSGA-III selection operator. Code is heavily based on the version of nsga3 in

class desdeo_emo.selection.IOPIS_NSGAIII.**IOPIS_NSGAIII_select**(*scalarization_nethods*, *pop: desdeo_emo.population.Population.Population*, *n_survive: int = None*, *selection_type: str = None*)

Bases: *desdeo_emo.selection.NSGAIII_select.NSGAIII_select*

The NSGA-III selection operator. Code is heavily based on the version of nsga3 in the pymoo package by msu-coinlab.

Parameters

- **pop** (Population) – [description]
- **n_survive** (*int, optional*) – [description], by default None

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*, *reference_point: numpy.ndarray*) → List[int]

Select individuals for mating for NSGA-III.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

desdeo_emo.selection.MOEAD_select

Module Contents

Classes

MOEAD_select	The MOEAD selection operator.

class desdeo_emo.selection.MOEAD_select.**MOEAD_select**(*pop: desdeo_emo.population.Population.Population*, *SF_type: desdeo_tools.scalarization.MOEADSF.MOEADSFBase*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The MOEAD selection operator.

Parameters

- **pop** (Population) – The population of individuals
- **SF_type** (*MOEADSFBase*) – The scalarizing function employed to evaluate the solutions

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*, *ideal_point*, *current_neighborhood*, *offspring_fx*) → List[int]

Select the individuals that are kept in the neighborhood.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.
- **ideal_point** – Ideal vector found so far
- **current_neighborhood** – Neighborhood to be updated
- **offspring_fx** – Offspring solution to be compared with the rest of the neighborhood

Returns List of indices of the selected individuals

Return type List[int]

_evaluate_SF(*self*, *neighborhood*, *weights*, *ideal_point*)

desdeo_emo.selection.NSGAIII_select

Module Contents

Classes

NSGAIII_select	The NSGA-III selection operator. Code is heavily based on the version of nsga3 in

class desdeo_emo.selection.NSGAIII_select.**NSGAIII_select**(*pop: desdeo_emo.population.Population.Population*, *n_survive: int = None*, *selection_type: str = None*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The NSGA-III selection operator. Code is heavily based on the version of nsga3 in the pymoo package by msu-coinlab.

Parameters

- **pop** (Population) – [description]
- **n_survive** (*int, optional*) – [description], by default None

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*) → List[int]
Select individuals for mating for NSGA-III.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

get_extreme_points_c(*self*, *F*, *ideal_point*, *extreme_points=None*)
Taken from pymoo

get_nadir_point(*self*, *extreme_points*, *ideal_point*, *worst_point*, *worst_of_front*, *worst_of_population*)

niching(*self*, *F*, *n_remaining*, *niche_count*, *niche_of_individuals*, *dist_to_niche*)

associate_to_niches(*self*, *F*, *ref_dirs*, *ideal_point*, *nadir_point*, *utopian_epsilon=0.0*)

calc_niche_count(*self*, *n_niches*, *niche_of_individuals*)

calc_perpendicular_distance(*self*, *N*, *ref_dirs*)

_calculate_fitness(*self*, *pop*) → numpy.ndarray

desdeo_emo.selection.SelectionBase

Module Contents

Classes

SelectionBase	The base class for the selection operator.

class desdeo_emo.selection.SelectionBase.**SelectionBase**
Bases: abc.ABC

The base class for the selection operator.

abstract do(*self*, *fitness: numpy.ndarray*, **args*) → List[int]

Use the selection operator over the given fitness values. Return the indices individuals with the best fitness values according to the operator.

Parameters **fitness** (*np.ndarray*) – Fitness of the individuals from which the next generation is to be selected.

Returns The list of selected individuals

Return type List[int]

desdeo_emo.selection.oAPD

Module Contents

Classes

Optimistic_APD_Select	The selection operator for the RVEA algorithm. Read the following paper for more

class desdeo_emo.selection.oAPD.**Optimistic_APD_Select**(*pop: desdeo_emo.population.Population.Population*, *time_penalty_function: Callable*, *alpha: float = 2*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the RVEA algorithm. Read the following paper for more details. R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.
- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*) → List[int]

Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

desdeo_emo.selection.robust_APD

Module Contents

Classes

robust_APD_Select	The selection operator for the RVEA algorithm. Read the following paper for more

class desdeo_emo.selection.robust_APD.**robust_APD_Select**(*pop: desdeo_emo.population.Population.Population*, *time_penalty_function: Callable*, *alpha: float = 2*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the RVEA algorithm. Read the following paper for more details. R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.
- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*) → List[int]

Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

desdeo_emo.selection.tournament_select

Module Contents

Functions

tour_select(fitness, tournament_size)	Tournament selection. Choose number of individuals to participate

desdeo_emo.selection.tournament_select.**tour_select**(*fitness*, *tournament_size*)

Tournament selection. Choose number of individuals to participate and select the one with the best fitness.

Parameters

- **fitness** (*array_like*) – An array of each individual's fitness.
- **tournament_size** (*int*) – Number of participants in the tournament.

Returns The index of the best individual.

Return type int

Package Contents

Classes

APD_Select	The selection operator for the RVEA algorithm. Read the following paper for more
NSGAIII_select	
MOEAD_select	The MOEAD selection operator.

Functions

tour_select(fitness, tournament_size)	Tournament selection. Choose number of individuals to participate

class desdeo_emo.selection.**APD_Select**(*pop: desdeo_emo.population.Population.Population*, *time_penalty_function: Callable*, *alpha: float = 2*, *selection_type: str = None*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The selection operator for the RVEA algorithm. Read the following paper for more details. R. Cheng, Y. Jin, M. Olhofer and B. Sendhoff, A Reference Vector Guided Evolutionary Algorithm for Many-objective Optimization, IEEE Transactions on Evolutionary Computation, 2016

Parameters

- **pop** (Population) – The population instance
- **time_penalty_function** (*Callable*) – A function that returns the time component in the penalty function.

- **alpha** (*float, optional*) – The RVEA alpha parameter, by default 2

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*) → List[int]
Select individuals for mating on basis of Angle penalized distance.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

_partial_penalty_factor(*self*) → float

Calculate and return the partial penalty factor for APD calculation. This calculation does not include the angle related terms, hence the name. If the calculated penalty is outside [0, 1], it will round it up/down to 0/1

Returns The partial penalty value

Return type float

_calculate_fitness(*self*, *pop*) → numpy.ndarray

class desdeo_emo.selection.**NSGAIII_select**(*pop: desdeo_emo.population.Population.Population*, *n_survive: int = None*, *selection_type: str = None*)
Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The NSGA-III selection operator. Code is heavily based on the version of nsga3 in the pymoo package by msu-coinlab.

Parameters

- **pop** (Population) – [description]
- **n_survive** (*int, optional*) – [description], by default None

do(*self*, *pop: desdeo_emo.population.Population.Population*, *vectors: desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*) → List[int]
Select individuals for mating for NSGA-III.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.

Returns List of indices of the selected individuals

Return type List[int]

get_extreme_points_c(*self*, *F*, *ideal_point*, *extreme_points=None*)
Taken from pymoo

get_nadir_point(*self*, *extreme_points*, *ideal_point*, *worst_point*, *worst_of_front*, *worst_of_population*)

niching(*self*, *F*, *n_remaining*, *niche_count*, *niche_of_individuals*, *dist_to_niche*)

associate_to_niches(*self*, *F*, *ref_dirs*, *ideal_point*, *nadir_point*, *utopian_epsilon=0.0*)

calc_niche_count (*self*, *n_niches*, *niche_of_individuals*)

calc_perpendicular_distance (*self*, *N*, *ref_dirs*)

_calculate_fitness (*self*, *pop*) → numpy.ndarray

desdeo_emo.selection.**tour_select** (*fitness*, *tournament_size*)

Tournament selection. Choose number of individuals to participate and select the one with the best fitness.

Parameters

- **fitness** (*array_like*) – An array of each individual's fitness.
- **tournament_size** (*int*) – Number of participants in the tournament.

Returns The index of the best individual.

Return type int

class desdeo_emo.selection.**MOEAD_select** (*pop:* desdeo_emo.population.Population.Population, *SF_type: desdeo_tools.scalarization.MOEADSF.MOEADSFBase*)

Bases: *desdeo_emo.selection.SelectionBase.SelectionBase*

The MOEAD selection operator.

Parameters

- **pop** (Population) – The population of individuals
- **SF_type** (*MOEADSFBase*) – The scalarizing function employed to evaluate the solutions

do (*self*, *pop:* desdeo_emo.population.Population.Population, *vectors:* desdeo_emo.utilities.ReferenceVectors.ReferenceVectors, *ideal_point*, *current_neighborhood*, *offspring_fx*) → List[int]

Select the individuals that are kept in the neighborhood.

Parameters

- **pop** (Population) – The current population.
- **vectors** (ReferenceVectors) – Class instance containing reference vectors.
- **ideal_point** – Ideal vector found so far
- **current_neighborhood** – Neighborhood to be updated
- **offspring_fx** – Offspring solution to be compared with the rest of the neighborhood

Returns List of indices of the selected individuals

Return type List[int]

_evaluate_SF (*self*, *neighborhood*, *weights*, *ideal_point*)

desdeo_emo.surrogatemodels

This module provides implementations of EAs which can be used for training surrogate models.

Submodules

desdeo_emo.surrogatemodels.BioGP

Module Contents

Classes

Node	A node object representing a function or terminal node in the tree.
LinearNode	The parent node of the tree, from which a number of subtrees emerge, as defined
BioGP	Helper class that provides a standard way to create an ABC using

Functions

negative_r2_score(y_true, y_pred)	

desdeo_emo.surrogatemodels.BioGP.**negative_r2_score**(*y_true*, *y_pred*)

class desdeo_emo.surrogatemodels.BioGP.**Node**(*value*, *depth*, *max_depth*, *max_subtrees*, *prob_terminal*, *function_set*, *terminal_set*)

A node object representing a function or terminal node in the tree.

Parameters

- **value** (*function, str or float*) – A function node has as its value a function. Terminal nodes contain variables which are either float or str.
- **depth** (*int*) – The depth the node is at.
- **function_set** (*array_like*) – The function set to use when creating the trees.
- **terminal_set** (*array_like*) – The terminals (variables and constants) to use when creating the trees.

predict(*self*, *decision_variables=None*)

node_label(*self*)

draw(*self*, *dot*, *count*)

get_sub_nodes(*self*)

Get all nodes belonging to the subtree under the current node.

Returns **nodes** – A list of nodes in the subtree.

Return type list

grow_tree(*self*, *max_depth=None*, *method='grow'*, *depth=0*, *ind=None*)

Create a random tree recursively using either grow or full method.

Parameters

- **max_depth** (*int*) – The maximum depth of the tree.

- **method** (*str*) – Methods: 'grow', 'full'. For the 'grow' method, nodes are chosen at random from both functions and terminals. The 'full' method chooses nodes from the function set until the max depth is reached, and then terminals are chosen.
- **depth** (*int*) – Current depth.
- **ind** – The starting node from which to begin growing trees.

class desdeo_emo.surrogatemodels.BioGP.**LinearNode**(*max_depth*, *max_subtrees*, *prob_terminal*, *function_set*, *terminal_set*, *error_lim*, *complexity_scalar=0.5*, *value='linear'*, *depth=0*)

Bases: *Node*

The parent node of the tree, from which a number of subtrees emerge, as defined by the user. The linear node takes a weighted sum of the output from the subtrees and also uses a bias value. The weights and the bias are calculated by the linear least square technique.

Parameters

- **value** (*function, str or float*) – A function node has as its value a function. Terminal nodes contain variables which are either float or str.
- **depth** (*int*) – The depth the node is at.

calculate_linear(*self*, *X_train*, *y_train*)

class desdeo_emo.surrogatemodels.BioGP.**BioGP**(*training_algorithm: Type[*desdeo_emo.EAs.BaseEA.BaseEA*] = PPGA*, *pop_size: int = 500*, *probability_crossover: float = 0.9*, *probability_mutation: float = 0.3*, *max_depth: int = 5*, *max_subtrees: int = 4*, *prob_terminal: float = 0.5*, *complexity_scalar: float = 0.5*, *error_lim: float = 0.001*, *init_method: str = 'ramped_half_and_half'*, *model_selection_criterion: str = 'min_error'*, *loss_function: str = 'mse'*, *single_obj_generations: int = 10*, *function_set=('add', 'sub', 'mul', 'div')*, *terminal_set=None*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit(*self*, *X: pandas.DataFrame*, *y: pandas.DataFrame*)

_create_individuals(*self*)

_model_performance(*self*, *trees: LinearNode*, *X: numpy.ndarray = None*, *y: numpy.ndarray = None*)

predict(*self*, *X: numpy.ndarray*)

select(*self*)

static add(*x*, *y*)

static sub(*x*, *y*)

static mul(*x*, *y*)

static div(*x*, *y*)

static sqrt(*x*)

static log(*x*)

static sin(*x*)

static cos(*x*)

static tan(*x*)

static neg(*x*)

desdeo_emo.surrogatemodels.EvoDN2

Module Contents

Classes

EvoDN2	Helper class that provides a standard way to create an ABC using

Functions

negative_r2_score(y_true, y_pred)

desdeo_emo.surrogatemodels.EvoDN2.**negative_r2_score**(*y_true*, *y_pred*)

class desdeo_emo.surrogatemodels.EvoDN2.**EvoDN2**(*num_subnets: int = 4*, *num_subsets: int = 4*, *max_layers: int = 4*, *max_nodes: int = 4*, *p_omit: float = 0.2*, *w_low: float = - 5.0*, *w_high: float = 5.0*, *subsets: list = None*, *activation_function: str = 'sigmoid'*, *loss_function: str = 'mse'*, *training_algorithm: desdeo_emo.EAs.BaseEA.BaseEA = PPGA*, *pop_size: int = 500*, *model_selection_criterion: str = 'min_error'*, *verbose: int = 0*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit(*self*, *X: numpy.ndarray*, *y: numpy.ndarray*)

_model_performance(*self*, *individuals: numpy.ndarray = None*, *X: numpy.ndarray = None*, *y_true: numpy.ndarray = None*)

_feed_forward(*self*, *subnets*, *X*)

_calculate_linear(*self*, *previous_layer_output*)

Calculate the final layer using LLSQ or

Parameters **non_linear_layer** (*np.ndarray*) – Output of the activation function

Returns

- **linear_layer** (*np.ndarray*) – The optimized weight matrix of the upper part of the network
- **predicted_values** (*np.ndarray*) – The prediction of the model

activate (*self*, *x*)

predict (*self*, *X*)

select (*self*)

_create_individuals (*self*)

desdeo_emo.surrogatemodels.EvoNN

Module Contents

Classes

EvoNN	Helper class that provides a standard way to create an ABC using

Functions

negative_r2_score(y_true, y_pred)	

desdeo_emo.surrogatemodels.EvoNN.**negative_r2_score** (*y_true*, *y_pred*)

class desdeo_emo.surrogatemodels.EvoNN.**EvoNN** (*num_hidden_nodes: int = 20*, *p_omit: float = 0.2*, *w_low: float = - 5.0*, *w_high: float = 5.0*, *activation_function: str = 'sigmoid'*, *loss_function: str = 'mse'*, *training_algorithm: Type[*desdeo_emo.EAs.BaseEA.BaseEA*] = PPGA*, *pop_size: int = 500*, *model_selection_criterion: str = 'akaike_corrected'*, *recombination_type: str = 'evonn_xover_mutation'*, *crossover_type: str = 'standard'*, *mutation_type: str = 'gaussian'*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit (*self*, *X: numpy.ndarray*, *y: numpy.ndarray*)

_model_performance (*self*, *first_layer: numpy.ndarray = None*, *X: numpy.ndarray = None*, *y_true: numpy.ndarray = None*)

predict (*self*, *X: numpy.ndarray = None*, *first_layer: numpy.ndarray = None*, *training: bool = False*)

activate (*self*, *x*)

calculate_linear (*self*, *previous_layer_output*)
Calculate the final layer using LLSQ or

Parameters **non_linear_layer** (*np.ndarray*) – Output of the activation function

Returns

- **linear_layer** (*np.ndarray*) – The optimized weight matrix of the upper part of the network
- **predicted_values** (*np.ndarray*) – The prediction of the model
- **training_error** (*float*) – The model's training error

_create_individuals(*self*)

select(*self*)

desdeo_emo.surrogatemodels.Problem

Module Contents

Classes

surrogateProblem	The base class from which every other class representing a problem should

class desdeo_emo.surrogatemodels.Problem.**surrogateProblem**(*performance_evaluator*)

Bases: desdeo_problem.problem.ProblemBase

The base class from which every other class representing a problem should derive.

evaluate(*self*, *model_parameters*, *use_surrogates=False*)

Evaluates the problem using an ensemble of input vectors. Uses surrogate models if available. Otherwise, it uses the true evaluator.

Parameters

- **decision_vectors** (*np.ndarray*) – An array of decision variable
- **vectors.** (*input*) –
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially
- **function or a surrogate model to evaluate the objectives.** (*expensive*) –

Returns

Dict with the following keys:

'objectives' (np.ndarray): The objective function values for each input vector.

'constraints' (Union[np.ndarray, None]): The constraint values of the problem corresponding each input vector.

'fitness' (np.ndarray): Equal to objective values if objective is to be minimized. Multiplied by (-1) if objective to be maximized.

'uncertainity' (Union[np.ndarray, None]): The uncertainity in the objective values.

Return type (Dict)

evaluate_constraint_values(*self*)

Evaluate just the constraint function values using the attributes decision_vectors and objective_vectors

Note: Currently not supported by ScalarMOProblem

get_variable_bounds(*self*)

get_objective_names(*self*)

Package Contents

Classes

BioGP	Helper class that provides a standard way to create an ABC using
EvoNN	Helper class that provides a standard way to create an ABC using
EvoDN2	Helper class that provides a standard way to create an ABC using
surrogateProblem	The base class from which every other class representing a problem should

class desdeo_emo.surrogatemodels.**BioGP**(*training_algorithm: Type[*desdeo_emo.EAs.BaseEA.BaseEA*] = PPGA*, *pop_size: int = 500*, *probability_crossover: float = 0.9*, *probability_mutation: float = 0.3*, *max_depth: int = 5*, *max_subtrees: int = 4*, *prob_terminal: float = 0.5*, *complexity_scalar: float = 0.5*, *error_lim: float = 0.001*, *init_method: str = 'ramped_half_and_half'*, *model_selection_criterion: str = 'min_error'*, *loss_function: str = 'mse'*, *single_obj_generations: int = 10*, *function_set=('add', 'sub', 'mul', 'div')*, *terminal_set=None*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit(*self*, *X: pandas.DataFrame*, *y: pandas.DataFrame*)

_create_individuals(*self*)

_model_performance(*self*, *trees:* LinearNode, *X: numpy.ndarray = None*, *y: numpy.ndarray = None*)

predict(*self*, *X: numpy.ndarray*)

select(*self*)

static add(*x*, *y*)

static sub(*x*, *y*)

static mul(*x*, *y*)

static div(*x*, *y*)

static sqrt(*x*)

static log(*x*)

static sin(*x*)

static cos(*x*)

static tan(*x*)

static neg(*x*)

class desdeo_emo.surrogatemodels.**EvoNN**(*num_hidden_nodes: int = 20*, *p_omit: float = 0.2*, *w_low: float = - 5.0*, *w_high: float = 5.0*, *activation_function: str = 'sigmoid'*, *loss_function: str = 'mse'*, *training_algorithm: Type[*desdeo_emo.EAs.BaseEA.BaseEA*] = PPGA*, *pop_size: int = 500*, *model_selection_criterion: str = 'akaike_corrected'*, *recombination_type: str = 'evonn_xover_mutation'*, *crossover_type: str = 'standard'*, *mutation_type: str = 'gaussian'*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit(*self*, *X: numpy.ndarray*, *y: numpy.ndarray*)

_model_performance(*self*, *first_layer: numpy.ndarray = None*, *X: numpy.ndarray = None*, *y_true: numpy.ndarray = None*)

predict(*self*, *X: numpy.ndarray = None*, *first_layer: numpy.ndarray = None*, *training: bool = False*)

activate(*self*, *x*)

calculate_linear(*self*, *previous_layer_output*)

Calculate the final layer using LLSQ or

Parameters **non_linear_layer** (*np.ndarray*) – Output of the activation function

Returns

- **linear_layer** (*np.ndarray*) – The optimized weight matrix of the upper part of the network
- **predicted_values** (*np.ndarray*) – The prediction of the model
- **training_error** (*float*) – The model's training error

_create_individuals(*self*)

select(*self*)

class desdeo_emo.surrogatemodels.**EvoDN2**(*num_subnets: int = 4*, *num_subsets: int = 4*, *max_layers: int = 4*, *max_nodes: int = 4*, *p_omit: float = 0.2*, *w_low: float = - 5.0*, *w_high: float = 5.0*, *subsets: list = None*, *activation_function: str = 'sigmoid'*, *loss_function: str = 'mse'*, *training_algorithm: desdeo_emo.EAs.BaseEA.BaseEA = PPGA*, *pop_size: int = 500*, *model_selection_criterion: str = 'min_error'*, *verbose: int = 0*)

Bases: desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor

Helper class that provides a standard way to create an ABC using inheritance.

fit(*self*, *X: numpy.ndarray*, *y: numpy.ndarray*)

_model_performance(*self*, *individuals: numpy.ndarray = None*, *X: numpy.ndarray = None*, *y_true: numpy.ndarray = None*)

_feed_forward(*self*, *subnets*, *X*)

_calculate_linear(*self*, *previous_layer_output*)
Calculate the final layer using LLSQ or

Parameters **non_linear_layer** (*np.ndarray*) – Output of the activation function

Returns

- **linear_layer** (*np.ndarray*) – The optimized weight matrix of the upper part of the network
- **predicted_values** (*np.ndarray*) – The prediction of the model

activate(*self*, *x*)

predict(*self*, *X*)

select(*self*)

_create_individuals(*self*)

class desdeo_emo.surrogatemodels.**surrogateProblem**(*performance_evaluator*)
Bases: desdeo_problem.problem.ProblemBase

The base class from which every other class representing a problem should derive.

evaluate(*self*, *model_parameters*, *use_surrogates=False*)
Evaluates the problem using an ensemble of input vectors. Uses surrogate models if available. Otherwise, it uses the true evaluator.

Parameters

- **decision_vectors** (*np.ndarray*) – An array of decision variable
- **vectors.** (*input*) –
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially
- **function or a surrogate model to evaluate the objectives.** (*expensive*) –

Returns

Dict with the following keys:

'objectives' (np.ndarray): The objective function values for each input vector.

'constraints' (Union[np.ndarray, None]): The constraint values of the problem corresponding each input vector.

'fitness' (np.ndarray): Equal to objective values if objective is to be minimized. Multiplied by (-1) if objective to be maximized.

'uncertainity' (Union[np.ndarray, None]): The uncertainity in the objective values.

Return type (Dict)

evaluate_constraint_values(*self*)
Evaluate just the constraint function values using the attributes decision_vectors and objective_vectors

Note: Currently not supported by ScalarMOProblem

get_variable_bounds(*self*)

get_objective_names(*self*)

desdeo_emo.utilities

This module provides miscellaneous tools

Submodules

desdeo_emo.utilities.IsNotebook

Module Contents

Functions

IsNotebook() → bool	Checks if the current environment is a Jupyter Notebook or a console.

desdeo_emo.utilities.IsNotebook.**IsNotebook**() → bool
Checks if the current environment is a Jupyter Notebook or a console.

Returns True if notebook. False if console

Return type bool

desdeo_emo.utilities.ReferenceVectors

Module Contents

Classes

ReferenceVectors	Class object for reference vectors.

Functions

normalize(vectors)	Normalize a set of vectors.
shear(vectors, degrees: float = 5)	Shear a set of vectors lying on the plane z=0 towards the z-axis, such that the
rotate(initial_vector, rotated_vector, other_vectors)	Calculate the rotation matrix that rotates the initial_vector to the
householder(vector)	Return reflection matrix via householder transformation.
rotate_toward(initial_vector, final_vector, other_vectors, degrees: float = 5)	Rotate other_vectors (with the centre at initial_vector) towards final_vector

desdeo_emo.utilities.ReferenceVectors.**normalize**(*vectors*)
Normalize a set of vectors.

The length of the returned vectors will be unity.

Parameters **vectors** (*np.ndarray*) – Set of vectors of any length, except zero.

desdeo_emo.utilities.ReferenceVectors.**shear**(*vectors*, *degrees: float = 5*)

Shear a set of vectors lying on the plane z=0 towards the z-axis, such that the resulting vectors 'degrees' angle away from the z axis.

z is the last element of the vector, and has to be equal to zero.

Parameters

- **vectors** (*numpy.ndarray*) – The final element of each vector should be zero.
- **degrees** (*float, optional*) – The angle that the resultant vectors make with the z axis. Unit is radians. (the default is 5)

desdeo_emo.utilities.ReferenceVectors.**rotate**(*initial_vector*, *rotated_vector*, *other_vectors*)

Calculate the rotation matrix that rotates the initial_vector to the rotated_vector. Apply that rotation on other_vectors and return. Uses Householder reflections twice to achieve this.

desdeo_emo.utilities.ReferenceVectors.**householder**(*vector*)

Return reflection matrix via householder transformation.

desdeo_emo.utilities.ReferenceVectors.**rotate_toward**(*initial_vector*, *final_vector*, *other_vectors*, *degrees: float = 5*)

Rotate other_vectors (with the centre at initial_vector) towards final_vector by an angle degrees.

Parameters

- **initial_vector** (*np.ndarray*) – Centre of the vectors to be rotated.
- **final_vector** (*np.ndarray*) – The final position of the center of other_vectors.
- **other_vectors** (*np.ndarray*) – The array of vectors to be rotated
- **degrees** (*float, optional*) – The amount of rotation (the default is 5)

Returns

- **rotated_vectors** (*np.ndarray*) – The rotated vectors
- **reached** (*bool*) – True if final_vector has been reached

class desdeo_emo.utilities.ReferenceVectors.**ReferenceVectors**(*lattice_resolution: int = None*, *number_of_objectives: int = None*, *creation_type: str = 'Uniform'*, *vector_type: str = 'Spherical'*, *ref_point: list = None*)

Class object for reference vectors.

_create(*self*, *creation_type: str = 'Uniform'*)

Create the reference vectors.

Parameters **creation_type** (*str, optional*) – 'Uniform' creates the reference vectors uniformly using simplex lattice design. 'Focused' creates reference vectors symmetrically around a central reference vector. By default 'Uniform'.

normalize(*self*)

Normalize the reference vectors to a unit hypersphere.

neighbouring_angles (*self*) → numpy.ndarray
Calculate neighbouring angles for normalization.

adapt (*self*, *fitness: numpy.ndarray*)
Adapt reference vectors. Then normalize.

Parameters **fitness** (*np.ndarray*) –

interactive_adapt_1 (*self*, *z: numpy.ndarray*, *n_solutions: int*, *translation_param: float = 0.2*) → None
Adapt reference vectors using the information about prefererred solution(s) selected by the Decision maker.

Parameters

- **z** (*np.ndarray*) – Preferred solution(s).
- **n_solutions** (*int*) – Number of solutions in total.
- **translation_param** (*float*) – Parameter determining how close the reference vectors are to the central vector
- **defined by using the selected solution** (***v***) –

Returns:

interactive_adapt_2 (*self*, *z: numpy.ndarray*, *n_solutions: int*, *predefined_distance: float = 0.2*) → None
Adapt reference vectors by using the information about non-preferred solution(s) selected by the Decision maker. After the Decision maker has specified non-preferred solution(s), Euclidian distance between normalized solution vector(s) and each of the reference vectors are calculated. Those reference vectors that are **closer** than a predefined distance are either **removed** or **re-positioned** somewhere else.

Note: At the moment, only the **removal** of reference vectors is supported. Repositioning of the reference vectors is **not** supported.

Note: In case the Decision maker specifies multiple non-preferred solutions, the reference vector(s) for which the distance to **any** of the non-preferred solutions is less than predefined distance are removed.

Note: Future developer should implement a way for a user to say: "Remove some percentage of objecive space/reference vectors" rather than giving a predefined distance value.

Parameters

- **z** (*np.ndarray*) – Non-preferred solution(s).
- **n_solutions** (*int*) – Number of solutions in total.
- **predefined_distance** (*float*) – The reference vectors that are closer than this distance are either removed or
- **somewhere else.** (*re-positioned*) –
- **value** (*Default*) – 0.2

Returns:

iteractive_adapt_3 (*self*, *ref_point*, *translation_param=0.2*)
Adapt reference vectors linearly towards a reference point. Then normalize.

The details can be found in the following paper: Hakanen, Jussi & Chugh, Tinkle & Sindhya, Karthik & Jin, Yaochu & Miettinen, Kaisa. (2016). Connections of Reference Vectors and Different Types of Preference Information in Interactive Multiobjective Evolutionary Algorithms.

Parameters

- **ref_point** –
- **translation_param** – (Default value = 0.2)

interactive_adapt_4 (*self*, *preferred_ranges: numpy.ndarray*) → None

Adapt reference vectors by using the information about the Decision maker's preferred range for each of the objective. Using these ranges, Latin hypercube sampling is applied to generate m number of samples between within these ranges, where m is the number of reference vectors. Normalized vectors constructed of these samples are then set as new reference vectors.

Parameters **preferred_ranges** (*np.ndarray*) – Preferred lower and upper bound for each of the objective function values.

Returns:

slow_interactive_adapt (*self*, *ref_point*)

Basically a wrapper around rotate_toward. Slowly rotate ref vectors toward ref_point. Return a boolean value to tell if the ref_point has been reached.

Parameters **ref_point** (*list or np.ndarray*) – The reference vectors will slowly move towards the ref_point.

Returns True if ref_point has been reached. False otherwise.

Return type boolean

add_edge_vectors (*self*)

Add edge vectors to the list of reference vectors.

Used to cover the entire orthant when preference information is provided.

desdeo_emo.utilities.newRV

Module Contents

Classes

newRV	pass

Functions

rotate(initial_vector, rotated_vector, other_vectors)	Calculate the rotation matrix that rotates the initial_vector to the
normalize(vector)	Normalize and return a vector.
householder(vector)	Return reflection matrix via householder transformation.
dist_based_translation(vectors)	Translates points towards origin based on distance.

continues on next page

Table 51 – continued from previous page

main()

class desdeo_emo.utilities.newRV.**newRV**(*lattice_resolution: int = None, number_of_objectives: int = None, creation_type: str = 'Uniform', vector_type: str = 'Spherical', ref_point: list = None*)

Bases: *desdeo_emo.utilities.ReferenceVectors.ReferenceVectors*

pass

rotate_to_axis(*self*, *ref_point*)

revert_rotation(*self*, *ref_point*)

project_to_hyperplane(*self*)
Projects the reference vectors to the hyperplane xn = 1.

translate_to_hypersphere(*self*)
Reverse of preject_to_hyperplane().

interact_v2(*self*, *ref_point*)
New kind of interaction.

interact_v3(*self*, *ref_point*)
New kind of interaction. More coverage.

desdeo_emo.utilities.newRV.**rotate**(*initial_vector*, *rotated_vector*, *other_vectors*)
Calculate the rotation matrix that rotates the initial_vector to the rotated_vector. Apply that rotation on other_vectors and return. Uses Householder reflections twice to achieve this.

desdeo_emo.utilities.newRV.**normalize**(*vector*)
Normalize and return a vector.

desdeo_emo.utilities.newRV.**householder**(*vector*)
Return reflection matrix via householder transformation.

desdeo_emo.utilities.newRV.**dist_based_translation**(*vectors*)
Translates points towards origin based on distance.

desdeo_emo.utilities.newRV.**main**()

desdeo_emo.utilities.plotlyanimate

Module Contents

Functions

*animate_init_*(data: Union[numpy.ndarray, pandas.DataFrame, list], filename: str) → dict	Plot the first (or zeroth) iteration of a population.
*animate_next_*(data: Union[numpy.ndarray, pandas.DataFrame, list], figure: dict, filename: str, generation: int = None) → dict	Plot the next set of individuals in an animation.
*animate_2d_init_*(data: Union[numpy.ndarray, pandas.DataFrame, list], filename: str) → dict	Initiate a 2D scatter animation.

continues on next page

Table 52 – continued from previous page

*animate_2d_next_*(data: Union[numpy.ndarray, pandas.DataFrame, list], figure: dict, filename: str, generation: int) → dict	Plot the next set of individuals in a 2D scatter animation.
*animate_3d_init_*(data: Union[numpy.ndarray, pandas.DataFrame, list], filename: str) → dict	Plot the first (or zeroth) iteration of a population.
*animate_3d_next_*(data: Union[numpy.ndarray, pandas.DataFrame, list], figure: dict, filename: str, generation: int) → dict	Plot the next set of individuals in an animation.
*animate_parallel_coords_init_*(data: Union[numpy.ndarray, pandas.DataFrame, list], filename: str) → dict	Plot the first (or zeroth) iteration of a population.
*animate_parallel_coords_next_*(data: Union[numpy.ndarray, pandas.DataFrame, list], figure: dict, filename: str, generation: int) → dict	Plot the next set of individuals in an animation.
test()	
test2()	

desdeo_emo.utilities.plotlyanimate.**animate_init_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *filename: str*) → dict

Plot the first (or zeroth) iteration of a population.

Intended as a frames object. Plots Scatter for 2D and 3D data. Plots parallel coordinate plot for higher dimensional data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – Contains the data to be plotted. Each row is an individual's objective values.
- **filename** (*str*) – Contains the name of the file to which the plot is saved.

Returns Plotly figure object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_next_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *figure: dict*, *filename: str*, *generation: int = None*) → dict

Plot the next set of individuals in an animation.

Plots scatter for 2D and 3D data, parallel coordinate plot for 4D and up.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – The objective values to be plotted
- **figure** (*dict*) – Plotly figure object compatible dict
- **filename** (*str*) – Name of the file to which the plot is saved
- **generation** (*int*) – Iteration Number

Returns Plotly Figure Object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_2d_init_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *filename: str*) → dict

Initiate a 2D scatter animation.

Only for 2D data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – Objective values
- **filename** (*str*) – Name of the file to which plot is saved

Returns Plotly Figure Object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_2d_next_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *figure: dict*, *filename: str*, *generation: int*) → dict

Plot the next set of individuals in a 2D scatter animation.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – The objective values to be plotted
- **figure** (*dict*) – Plotly figure object compatible dict
- **filename** (*str*) – Name of the file to which the plot is saved
- **generation** (*int*) – Iteration Number

Returns Plotly Figure Object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_3d_init_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *filename: str*) → dict

Plot the first (or zeroth) iteration of a population.

Intended as a frames object. Plots Scatter 3D data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – Contains the data to be plotted. Each row is an individual's objective values.
- **filename** (*str*) – Contains the name of the file to which the plot is saved.

Returns Plotly figure object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_3d_next_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *figure: dict*, *filename: str*, *generation: int*) → dict

Plot the next set of individuals in an animation.

Plots scatter for 3D data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – The objective values to be plotted
- **figure** (*dict*) – Plotly figure object compatible dict
- **filename** (*str*) – Name of the file to which the plot is saved
- **generation** (*int*) – Iteration Number

Returns Plotly Figure Object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_parallel_coords_init_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *filename: str*) → dict

Plot the first (or zeroth) iteration of a population.

Intended as a frames object. Plots parallel coordinate plot for >3D data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – Contains the data to be plotted. Each row is an individual's objective values.
- **filename** (*str*) – Contains the name of the file to which the plot is saved.

Returns Plotly figure object

Return type dict

desdeo_emo.utilities.plotlyanimate.**animate_parallel_coords_next_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *figure: dict*, *filename: str*, *generation: int*) → dict

Plot the next set of individuals in an animation.

Plots parallel coordinate plot for 4D and up.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – The objective values to be plotted
- **figure** (*dict*) – Plotly figure object compatible dict
- **filename** (*str*) – Name of the file to which the plot is saved
- **generation** (*int*) – Iteration Number

Returns Plotly Figure Object

Return type dict

desdeo_emo.utilities.plotlyanimate.**test**()

desdeo_emo.utilities.plotlyanimate.**test2**()

desdeo_emo.utilities.symmetric_vectors

Module Contents

Functions

shear(vectors, degrees: float = 5)	Shear a set of vectors lying on the plane z=0 towards the z-axis, such that the
normalize(vectors)	Normalize a set of vectors.
rotate(initial_vector, rotated_vector, other_vectors)	Calculate the rotation matrix that rotates the initial_vector to the
householder(vector)	Return reflection matrix via householder transformation.
rotate_toward(initial_vector, final_vector, other_vectors, degrees: float = 5)	Rotate other_vectors (with the centre at initial_vector) towards final_vector
main()	

desdeo_emo.utilities.symmetric_vectors.**shear**(*vectors*, *degrees: float = 5*)

Shear a set of vectors lying on the plane z=0 towards the z-axis, such that the resulting vectors 'degrees' angle away from the z axis.

z is the last element of the vector, and has to be equal to zero.

Parameters

- **vectors** (*numpy.ndarray*) – The final element of each vector should be zero.
- **degrees** (*float, optional*) – The angle that the resultant vectors make with the z axis. Unit is radians. (the default is 5)

desdeo_emo.utilities.symmetric_vectors.**normalize**(*vectors*)

Normalize a set of vectors.

The length of the returned vectors will be unity.

Parameters **vectors** (*np.ndarray*) – Set of vectors of any length, except zero.

desdeo_emo.utilities.symmetric_vectors.**rotate**(*initial_vector*, *rotated_vector*, *other_vectors*)

Calculate the rotation matrix that rotates the initial_vector to the rotated_vector. Apply that rotation on other_vectors and return. Uses Householder reflections twice to achieve this.

desdeo_emo.utilities.symmetric_vectors.**householder**(*vector*)

Return reflection matrix via householder transformation.

desdeo_emo.utilities.symmetric_vectors.**rotate_toward**(*initial_vector*, *final_vector*, *other_vectors*, *degrees: float = 5*)

Rotate other_vectors (with the centre at initial_vector) towards final_vector by an angle degrees.

Parameters

- **initial_vector** (*np.ndarray*) – Centre of the vectors to be rotated.
- **final_vector** (*np.ndarray*) – The final position of the center of other_vectors.
- **other_vectors** (*np.ndarray*) – The array of vectors to be rotated
- **degrees** (*float, optional*) – The amount of rotation (the default is 5)

Returns

- **rotated_vectors** (*np.ndarray*) – The rotated vectors
- **reached** (*bool*) – True if final_vector has been reached

desdeo_emo.utilities.symmetric_vectors.**main**()

Package Contents

Classes

ReferenceVectors	

Functions

IsNotebook() → bool	Checks if the current environment is a Jupyter Notebook or a console.
*animate_init_*(data: Union[numpy.ndarray, pandas.DataFrame, list], filename: str) → dict	Plot the first (or zeroth) iteration of a population.
*animate_next_*(data: Union[numpy.ndarray, pandas.DataFrame, list], figure: dict, filename: str, generation: int = None) → dict	Plot the next set of individuals in an animation.

desdeo_emo.utilities.**IsNotebook**() → bool

Checks if the current environment is a Jupyter Notebook or a console.

Returns True if notebook. False if console

Return type bool

desdeo_emo.utilities.**animate_init_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *filename: str*) → dict

Plot the first (or zeroth) iteration of a population.

Intended as a frames object. Plots Scatter for 2D and 3D data. Plots parallel coordinate plot for higher dimensional data.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – Contains the data to be plotted. Each row is an individual's objective values.
- **filename** (*str*) – Contains the name of the file to which the plot is saved.

Returns Plotly figure object

Return type dict

desdeo_emo.utilities.**animate_next_**(*data: Union[numpy.ndarray, pandas.DataFrame, list]*, *figure: dict*, *filename: str*, *generation: int = None*) → dict

Plot the next set of individuals in an animation.

Plots scatter for 2D and 3D data, parallel coordinate plot for 4D and up.

Parameters

- **data** (*Union[np.ndarray, pd.DataFrame, list]*) – The objective values to be plotted
- **figure** (*dict*) – Plotly figure object compatible dict
- **filename** (*str*) – Name of the file to which the plot is saved
- **generation** (*int*) – Iteration Number

Returns Plotly Figure Object

Return type dict

class desdeo_emo.utilities.**ReferenceVectors**(*lattice_resolution: int = None*, *number_of_objectives: int = None*, *creation_type: str = 'Uniform'*, *vector_type: str = 'Spherical'*, *ref_point: list = None*)

Class object for reference vectors.

_create(*self*, *creation_type: str = 'Uniform'*)

Create the reference vectors.

Parameters **creation_type** (*str, optional*) – 'Uniform' creates the reference vectors uniformly using simplex lattice design. 'Focused' creates reference vectors symmetrically around a central reference vector. By default 'Uniform'.

normalize(*self*)

Normalize the reference vectors to a unit hypersphere.

neighbouring_angles(*self*) → numpy.ndarray

Calculate neighbouring angles for normalization.

adapt(*self*, *fitness: numpy.ndarray*)

Adapt reference vectors. Then normalize.

Parameters **fitness** (*np.ndarray*) –

interactive_adapt_1(*self*, *z: numpy.ndarray*, *n_solutions: int*, *translation_param: float = 0.2*) → None

Adapt reference vectors using the information about prefererred solution(s) selected by the Decision maker.

Parameters

- **z** (*np.ndarray*) – Preferred solution(s).
- **n_solutions** (*int*) – Number of solutions in total.
- **translation_param** (*float*) – Parameter determining how close the reference vectors are to the central vector
- **defined by using the selected solution** (***v***) –

Returns:

interactive_adapt_2(*self*, *z: numpy.ndarray*, *n_solutions: int*, *predefined_distance: float = 0.2*) → None

Adapt reference vectors by using the information about non-preferred solution(s) selected by the Decision maker. After the Decision maker has specified non-preferred solution(s), Euclidian distance between normalized solution vector(s) and each of the reference vectors are calculated. Those reference vectors that are **closer** than a predefined distance are either **removed** or **re-positioned** somewhere else.

Note: At the moment, only the **removal** of reference vectors is supported. Repositioning of the reference vectors is **not** supported.

Note: In case the Decision maker specifies multiple non-preferred solutions, the reference vector(s) for which the distance to **any** of the non-preferred solutions is less than predefined distance are removed.

Note: Future developer should implement a way for a user to say: "Remove some percentage of objecive space/reference vectors" rather than giving a predefined distance value.

Parameters

- **z** (*np.ndarray*) – Non-preferred solution(s).
- **n_solutions** (*int*) – Number of solutions in total.
- **predefined_distance** (*float*) – The reference vectors that are closer than this distance are either removed or
- **somewhere else.** (*re-positioned*) –
- **value** (*Default*) – 0.2

Returns:

iteractive_adapt_3 (*self*, *ref_point*, *translation_param=0.2*)

Adapt reference vectors linearly towards a reference point. Then normalize.

The details can be found in the following paper: Hakanen, Jussi & Chugh, Tinkle & Sindhya, Karthik & Jin, Yaochu & Miettinen, Kaisa. (2016). Connections of Reference Vectors and Different Types of Preference Information in Interactive Multiobjective Evolutionary Algorithms.

Parameters

- **ref_point** –
- **translation_param** – (Default value = 0.2)

interactive_adapt_4 (*self*, *preferred_ranges: numpy.ndarray*) → None

Adapt reference vectors by using the information about the Decision maker's preferred range for each of the objective. Using these ranges, Latin hypercube sampling is applied to generate m number of samples between within these ranges, where m is the number of reference vectors. Normalized vectors constructed of these samples are then set as new reference vectors.

Parameters **preferred_ranges** (*np.ndarray*) – Preferred lower and upper bound for each of the objective function values.

Returns:

slow_interactive_adapt (*self*, *ref_point*)

Basically a wrapper around rotate_toward. Slowly rotate ref vectors toward ref_point. Return a boolean value to tell if the ref_point has been reached.

Parameters **ref_point** (*list or np.ndarray*) – The reference vectors will slowly move towards the ref_point.

Returns True if ref_point has been reached. False otherwise.

Return type boolean

add_edge_vectors (*self*)

Add edge vectors to the list of reference vectors.

Used to cover the entire orthant when preference information is provided.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U