ParMOO User's Manual

Release 0.3.1+dev

Tyler H. Chang, Stefan M. Wild, et al.

Sep 25, 2023

Contents

Chapter 1

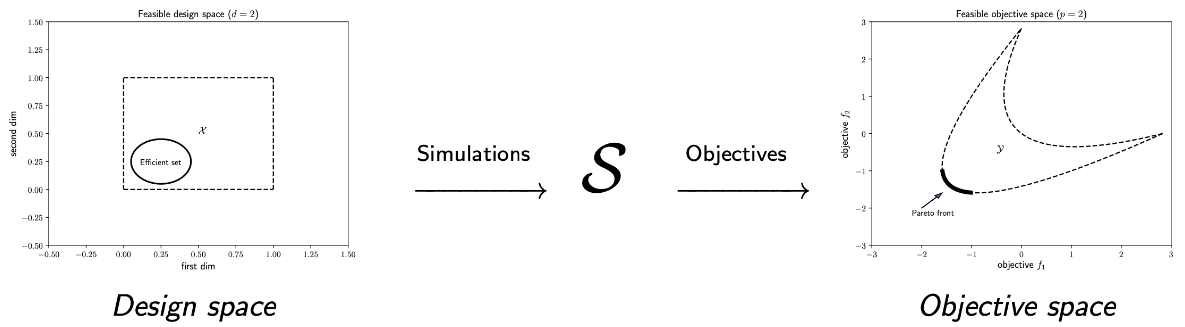
User's Guide

1.1 Quickstart

ParMOO is a parallel multiobjective optimization solver that seeks to exploit simulation-based structure in objective and constraint functions.

To exploit structure, ParMOO models *simulations* separately from *objectives* and *constraints*. In our language:

- a **design variable** is an input to the problem, which we can directly control;
- a **simulation** is an expensive or time-consuming process, including real-world experimentation, which is treated as a blackbox function of the design variables and evaluated sparingly;
- an **objective** is an algebraic function of the design variables and/or simulation outputs, which we would like to optimize; and
- a **constraint** is an algebraic function of the design variables and/or simulation outputs, which cannot exceed a specified bound.



To solve a multiobjective optimization problem (MOOP), we use surrogate models of the simulation outputs, together with the algebraic definition of the objectives and constraints.

In order to achieve scalable parallelism, we use libEnsemble to distribute batches of simulation evaluations across parallel resources.

1.1.1 Dependencies

ParMOO has been tested on Unix/Linux and MacOS systems.

ParMOO's base has the following dependencies:

- Python 3.8+
- numpy – for data structures and performant numerical linear algebra
- scipy – for scientific calculations needed for specific modules
- pyDOE – for generating experimental designs
- pandas – for exporting the resulting databases

Additional dependencies are needed to use the additional features in `parmoo.extras`:

- libEnsemble – for managing parallel simulation evaluations

And for using the Pareto front visualization library in `parmoo.viz`:

- plotly – for generating interactive plots
- dash – for hosting interactive plots in your browser
- kaleido – for exporting static plots post-interaction

1.1.2 Installation

The easiest way to install ParMOO is via the Python package index, PyPI (commonly called `pip`):

```
pip install < --user > parmoo
```

where the braces around `< --user >` indicate that the `--user` flag is optional.

To install *all* dependencies (including libEnsemble) use:

```
pip install < --user > "parmoo[extras]"
```

You can also clone this project from our GitHub and `pip` install it in-place, so that you can easily pull the latest version or checkout the `develop` branch for pre-release features. On Debian-based systems with a bash shell, this looks like:

```
git clone https://github.com/parmoo/parmoo
cd parmoo
pip install -e .
```

Alternatively, the latest release of ParMOO (including all required and optional dependencies) can be installed from the `conda-forge` channel using:

```
conda install --channel=conda-forge parmoo
```

Before doing so, it is recommended to create a new conda environment using:

```
conda create --name channel-name
conda activate channel-name
```

For detailed instructions, see `Advanced Installation`.

1.1.3 Testing

If you have pytest with the pytest-cov plugin and flake8 installed, then you can test your installation.

```
python3 setup.py test
```

These tests are run regularly using GitHub Actions.

1.1.4 Basic Usage

ParMOO uses numpy in an object-oriented design, based around the `MOOP` class. To get started, create a `MOOP` object, using the `constructor`.

```
from parmoo import MOOP
from parmoo.optimizers import LocalGPS

my_moop = MOOP(LocalGPS)
```

To summarize the framework, in each iteration ParMOO models each simulation using a computationally cheap surrogate, then solves one or more scalarizations of the objectives, which are specified by acquisition functions. Read more about this framework at our `Learn About MOOPs` page. In the above example, `optimizers.LocalGPS` is the class of optimizers that the `my_moop` will use to solve the scalarized surrogate problems.

Next, add design variables to the problem as follows using the `MOOP.addDesign(*args)` method. In this example, we define one continuous and one categorical design variable. Other options include integer, custom, and raw (using raw variables is not recommended except for expert users).

```
# Add a single continuous design variable in the range [0.0, 1.0]
my_moop.addDesign({'name': "x1", # optional, name
                   'des_type': "continuous", # optional, type of variable
                   'lb': 0.0, # required, lower bound
                   'ub': 1.0, # required, upper bound
                   'tol': 1.0e-8 # optional tolerance
                  })
# Add a second categorical design variable with 3 levels
my_moop.addDesign({'name': "x2", # optional, name
                   'des_type': "categorical", # required, type of variable
                   'levels': ["good", "bad"] # required, category names
                  })
```

Next, add simulations to the problem as follows using the `MOOP.addSimulation(*args)` method. In this example, we define a toy simulation `sim_func(x)`.

```
import numpy as np
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
```

(continues on next page)

(continued from previous page)

```
# Define a toy simulation for the problem, whose outputs are quadratic
def sim_func(x):
   if x["x2"] == "good":
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])
# Add the simulation to the problem
my_moop.addSimulation({'name': "MySim", # Optional name for this simulation
                       'm': 2, # This simulation has 2 outputs
                       'sim_func': sim_func, # Our sample sim from above
                       'search': LatinHypercube, # Use a LH search
                       'surrogate': GaussRBF, # Use a Gaussian RBF surrogate
                       'hyperparams': {}, # Hyperparams passed to internals
                       'sim_db': { # Optional dict of precomputed points
                                  'search_budget': 10 # Set search budget
                                 },
                      })
```

Now we can add objectives and constraints using `MOOP.addObjective(*args)` and `MOOP.addConstraint(*args)`. In this example, there are 2 objectives (each corresponding to a single simulation output) and one constraint.

```
# First objective just returns the first simulation output
def f1(x, s): return s["MySim"][0]
my_moop.addObjective({'name': "f1", 'obj_func': f1})
# Second objective just returns the second simulation output
def f2(x, s): return s["MySim"][1]
my_moop.addObjective({'name': "f2", 'obj_func': f2})
# Add a single constraint, that x[0] >= 0.1
def c1(x, s): return 0.1 - x["x1"]
my_moop.addConstraint({'name': "c1", 'constraint': c1})
```

Finally, we must add one or more acquisition functions using `MOOP.addAcquisition(*args)`. These are used to scalarize the surrogate problems. The number of acquisition functions typically determines the number of simulation evaluations per batch. This is useful to know if you are using a parallel solver.

```
from parmoo.acquisitions import RandomConstraint

# Add 3 acquisition functions
for i in range(3):
   my_moop.addAcquisition({'acquisition': RandomConstraint,
                           'hyperparams': {}})
```

Finally, the MOOP is solved using the `MOOP.solve(budget)` method, and the results can be viewed using `MOOP.getPF()`.

```
import pandas as pd

my_moop.solve(5) # Solve with 5 iterations of ParMOO algorithm
results = my_moop.getPF(format="pandas") # Extract the results as pandas df
```

After executing the above block of code, the `results` variable points to a pandas dataframe, each of whose rows corresponds to a nondominated objective value in the `my_moop` object's final database. You can reference individual columns in the `results` array by using the `name` keys that were assigned during `my_moop`'s construction, or plot the results by using the `viz` library.

Congratulations, you now know enough to get started solving MOOPs!

1.1.5 Minimal Working Example

Putting it all together, we get the following minimal working example.

```
import numpy as np
import pandas as pd
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import RandomConstraint
from parmoo.optimizers import LocalGPS

# Fix the random seed for reproducibility
np.random.seed(0)

my_moop = MOOP(LocalGPS)

my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': ["good", "bad"]})

def sim_func(x):
   if x["x2"] == "good":
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

def f1(x, s): return s["MySim"][0]
def f2(x, s): return s["MySim"][1]
my_moop.addObjective({'name': "f1", 'obj_func': f1})
my_moop.addObjective({'name': "f2", 'obj_func': f2})

def c1(x, s): return 0.1 - x["x1"]
my_moop.addConstraint({'name': "c1", 'constraint': c1})

for i in range(3):
   my_moop.addAcquisition({'acquisition': RandomConstraint,
                           'hyperparams': {}})
```

(continues on next page)

(continued from previous page)

```python
my_moop.solve(5)
results = my_moop.getPF(format="pandas")

# Display solution
print(results)

# Plot results -- must have extra viz dependencies installed
from parmoo.viz import scatter
# The optional arg `output` exports directly to jpg instead of interactive mode
scatter(my_moop, output="jpeg")
```

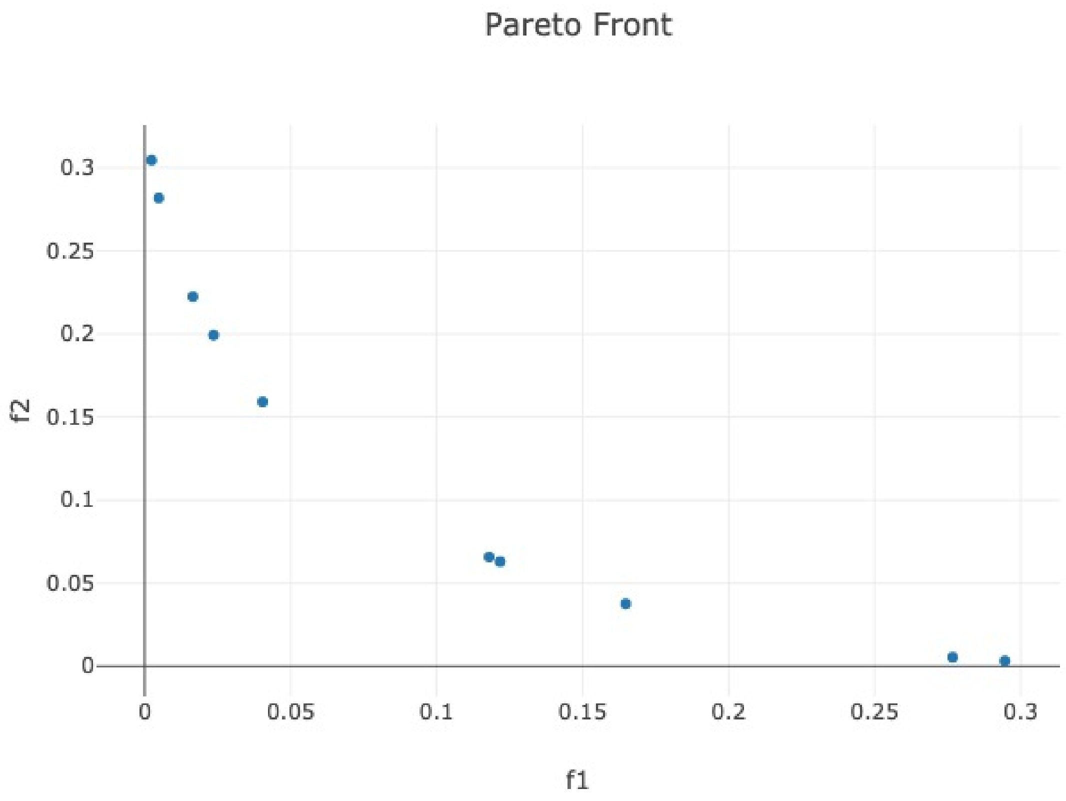
The above code saves all (approximate) Pareto optimal solutions in the `results` variable, and prints the `results` variable to the standard output:

```
         x1    x2        f1        f2        c1
0  0.742840  good  0.294675  0.003267 -0.642840
1  0.726092  good  0.276773  0.005462 -0.626092
2  0.605914  good  0.164766  0.037669 -0.505914
3  0.548931  good  0.121753  0.063036 -0.448931
4  0.543499  good  0.117991  0.065793 -0.443499
5  0.401011  good  0.040405  0.159192 -0.301011
6  0.353552  good  0.023578  0.199316 -0.253552
7  0.328402  good  0.016487  0.222404 -0.228402
8  0.269175  good  0.004785  0.281775 -0.169175
9  0.248183  good  0.002322  0.304502 -0.148183
```

And produces the following figure of the Pareto points:

1.1.6 Next Steps

- If you want to take advantage of all that ParMOO has to offer, please see `Writing a ParMOO Script`.
- If you would like more information on multiobjective optimization terminology and ParMOO's methodology, see the `Learn About MOOPs` page.
- For a full list of basic usage tutorials, see `More Tutorials`.
- To start solving MOOPs on parallel hardware, install libEnsemble and see the `libEnsemble tutorial`.
- See some of our pre-built solvers in the parmoo_solver_farm.
- To interactively explore your solutions, install its extra dependencies and use our built-in `viz` tool.
- For more advice, consult our FAQs.


Pareto Front
f2
0.3
0.25
0.2
0.15
0.1
0.05
0
0
0.05
0.1
0.15
0.2
0.25
0.3
f1

1.1.7 Resources

To seek support or report issues, e-mail:

- `parmoo@mcs.anl.gov`

Our full documentation is hosted on:

- ReadTheDocs

Please read our LICENSE and CONTRIBUTING files.

1.2 Advanced Installation

ParMOO can be installed with `pip` or directly from its GitHub source.

ParMOO's base has the following dependencies, which may be automatically installed depending on your choice of method:

- Python 3.6+
- numpy – for data structures and performant numerical linear algebra
- scipy – for scientific calculations needed for specific modules
- pyDOE – for generating experimental designs
- pandas – for exporting the resulting databases

Additional dependencies are needed to use the additional features in `parmoo.extras`:

- libEnsemble – for managing parallel simulation evaluations

And for using the Pareto front visualization library in `parmoo.viz`:

- plotly – for generating interactive plots
- dash – for hosting interactive plots in your browser
- kaleido – for exporting static plots post-interaction

If you want to run the tests (in `parmoo.tests`), then you will also need:

- pytest,
- pytest-cov, and
- flake8.

1.2.1 pip

The easiest way to install is via the PyPI package manager (`pip` utility). To install the latest release:

```
pip install < --user > parmoo
```

where the braces around < `--user` > indicate that the `--user` flag is optional.

Note that the default install will not install the extra dependencies, such as libEnsemble.

To install *all* dependencies, use:

```
pip install < --user > parmoo[extras]
```

To check the installation by running the full test suite, use:

```
python3 setup.py test
```

which will also install the test dependencies (pytest, pytest-cov, and flake8).

1.2.2 Conda Forge

For some users (in particular, this is the recommended method for Windows users), the preferred method for obtaining the latest release of ParMOO may be through the `conda` package manager. The latest release of ParMOO is available through the `conda-forge` channel. Note that `conda` does not support optional dependencies, so the following command will automatically fetch all required and optional dependencies:

```
conda install --channel=conda-forge parmoo
```

Before running the above command, it is recommended to create a new conda environment to avoid conflicts. Do so using:

```
conda create --name channel-name
conda activate channel-name
```

After performing a `conda-forge` installation of ParMOO, you can run our unit tests to make sure your installation is working:

```
py.test --pyargs parmoo.tests.unit_tests
```

1.2.3 Install from GitHub source

You may want to install ParMOO from its GitHub source code, so that you can easily pull the latest updates.

The easiest way to do this is to clone it from our GitHub and then `pip` install it in-place by using the `-e .` option. In a bash shell, that looks like this.

```
git clone https://github.com/parmoo/parmoo
cd parmoo
pip install -e .
```

This command will use the `setup.py` file to generate an `egg` inside the `parmoo` base directory.

Alternatively, you could just add the `parmoo` base directory to your `PYTHONPATH` environment variable. In the bash shell, this looks like:

```
git clone https://github.com/parmoo/parmoo
cd parmoo
export PYTHONPATH=$PYTHONPATH:`pwd`
```

However, this technique will not install any of ParMOO's dependencies.

Additionally, if you would like to use libEnsemble to handle parallel function evaluations (from `extras.libe`), you will need to also install libEnsemble.

To install libEnsemble with PyPI, use

```
pip3 install libensemble
```

or visit the libEnsemble_documentation for detailed installation instructions.

After installation, you can run the tests using either:

```
python3 setup.py test
```

(if you used the `pip install -e .` method), or:

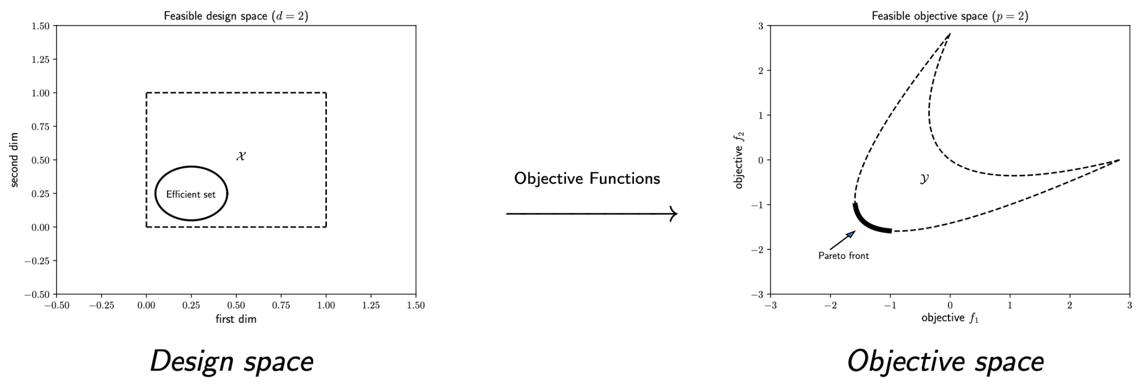
```
parmoo/tests/run-tests.sh -cu<rl>
```

1.3 Learn about MOOPs

1.3.1 Overview

A *multiobjective optimization problem* (MOOP) is an optimization problem involving multiple, potentially conflicting, objectives. MOOPs arise in many areas of science and engineering, for example, when designing a product or fitting a model according to multiple performance criteria.

The goal of a MOOP is to find numerous solutions that describe the tradeoffs between these (potentially conflicting) *objectives*. The solution to a MOOP is a set of achievable objective values (and corresponding design points), describing the inherent tradeoffs in the problem. This tradeoff curve is called the *Pareto front*. Real-world MOOPs may also involve some *constraints*—additional hard rules that every solution must adhere to.



In a multiobjective *simulation-based* optimization problem, the objectives are derived from the outputs of one or more expensive simulations.

Here, we use the term *simulation* in its broadest sense. In our terminology, a simulation could refer to any data-generating process of sufficient complexity, including

- numerical simulations run on a computer,
- physical experiments performed in a laboratory environment, and
- other nontrivial data-generating campaigns.

A simulation's expense can be judged by any of several factors, including

- the amount of time that is required to complete the simulation;
- the simulation's occupation of valuable scientific resources, such as computing nodes or laboratory equipment;
- the need for expert/human involvement to validate simulation outputs; and
- consumption of raw resources such as energy or laboratory materials.

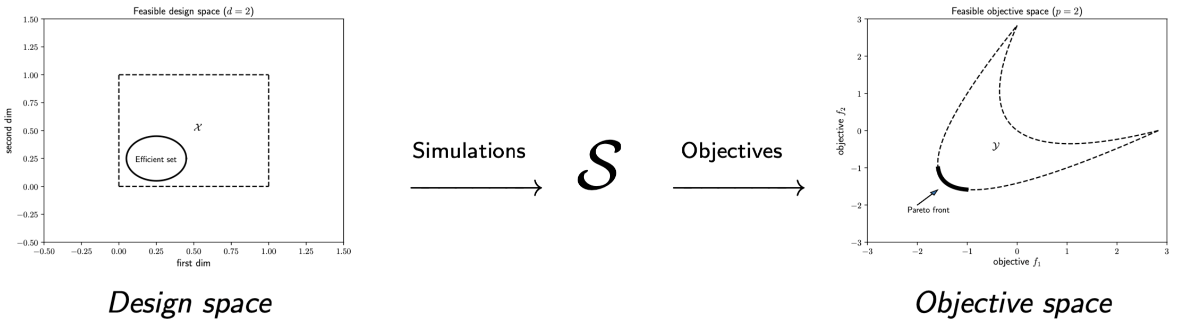
ParMOO is designed to solve multiobjective simulation-based optimization problems by exploiting the simulation-based structure in such problems.

One of the key concepts in ParMOO is the distinction between simulations and objectives.

In ParMOO, a simulation is an expensive or time-consuming process: it may require significant computational resources and may have a nonnegligible execution time.

$$\mathbf{S} : \mathcal{X} \rightarrow \mathcal{S}, \qquad \mathcal{X} \subset \mathbb{R}^n, \mathcal{S} \subset \mathbb{R}^m.$$

A single MOOP may involve multiple simulations, with varying costs, which map from the design variables to an intermediate space



ParMOO provides a framework for solving these problems, while parallelizing simulation evaluations.

In ParMOO, both objectives and constraints are algebraic functions of the design variables and/or the simulation outputs. After evaluating the simulation(s), evaluating the objective is assumed to be computationally inexpensive and tends to present a smaller opportunity for parallelization.

Objectives are expressed as

$$\mathbf{F} : \mathcal{X} \times \mathcal{S} \rightarrow \mathcal{Y}, \qquad \mathcal{Y} \subset \mathbb{R}^o.$$

Constraints are expressed as

$$\mathbf{G} : \mathcal{X} \times \mathcal{S} \rightarrow \mathbb{R}^p.$$

ParMOO allows its users to separately specify simulations, objective functions, and constraint functions. ParMOO will then utilize simulations sparingly, but it may use many objective and constraint evaluations to solve problems of the form

$$\min_{\mathbf{x} \in \mathcal{X}} \mathbf{F}(\mathbf{x}, \mathbf{S}(\mathbf{x})) \quad s.t. \quad \mathbf{G}(\mathbf{x}, \mathbf{S}(\mathbf{x})) \leq \mathbf{0}^*.$$

* Here "$\min$" is understood in the Pareto sense.

1.3.2 ParMOO Algorithm

ParMOO uses response surface methodology to solve MOOPs. This means that ParMOO fits a computationally cheaper surrogate to each output simulation, then optimizes various scalarizations of your problem using these surrogates instead of running the expensive simulations.

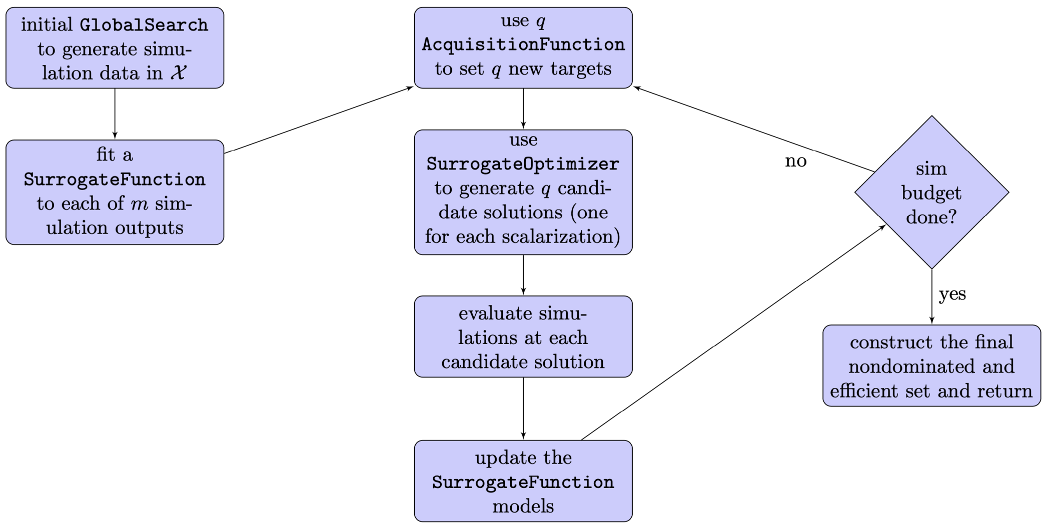
The key difference between ParMOO and other response surface techniques is that ParMOO uses its surrogates to model the *simulation* response surfaces, not to directly model the *objective* values. This allows ParMOO to exploit additional information about your problem, in situations where it is available.

Common use cases include the following.

- One or more objectives does not depend on any simulation outputs and therefore can be evaluated directly without concern for computational expense.
- The MOOP involves multiple simulations, each with differing costs, and therefore one simulation can be evaluated far more times than the other.
- The objectives have some exploitable structure, for example, the sum-of-squared simulation outputs, which readily admits additional information about the shape of the objective response surfaces.

This process has several key components, which ParMOO allows users to interchange.

- Before fitting any surrogates or performing any scalarizations, ParMOO must search the design space using one of the `GlobalSearch` implementations from the `parmoo.searches` module.
- After some data has been generated, ParMOO fits and updates a surrogate for each simulation output using one of the `SurrogateFunction` implementations from the `parmoo.surrogates` module.
- After fitting surrogates, ParMOO must scalarize the objectives so that it can solve the surrogate problems and produce candidate design points using one of the `AcquisitionFunction` implementations from the `parmoo.acquisitions` module.
- ParMOO must solve the scalarized surrogate optimization problems using one of the `SurrogateOptimizer` implementations from the `parmoo.optimizers` module.


initial GlobalSearch to generate simulation data in $\mathcal{X}$
fit a SurrogateFunction to each of m simulation outputs
use q AcquisitionFunction to set q new targets
use SurrogateOptimizer to generate q candidate solutions (one for each scalarization)
evaluate simulations at each candidate solution
update the SurrogateFunction models
no
sim budget done?
yes
construct the final nondominated and efficient set and return

You may mix and match built-in techniques to generate your own unique MOOP solver, or you may implement your own techniques by employing one of the abstract base classes defined in `parmoo.structs`.

1.3.3 Glossary

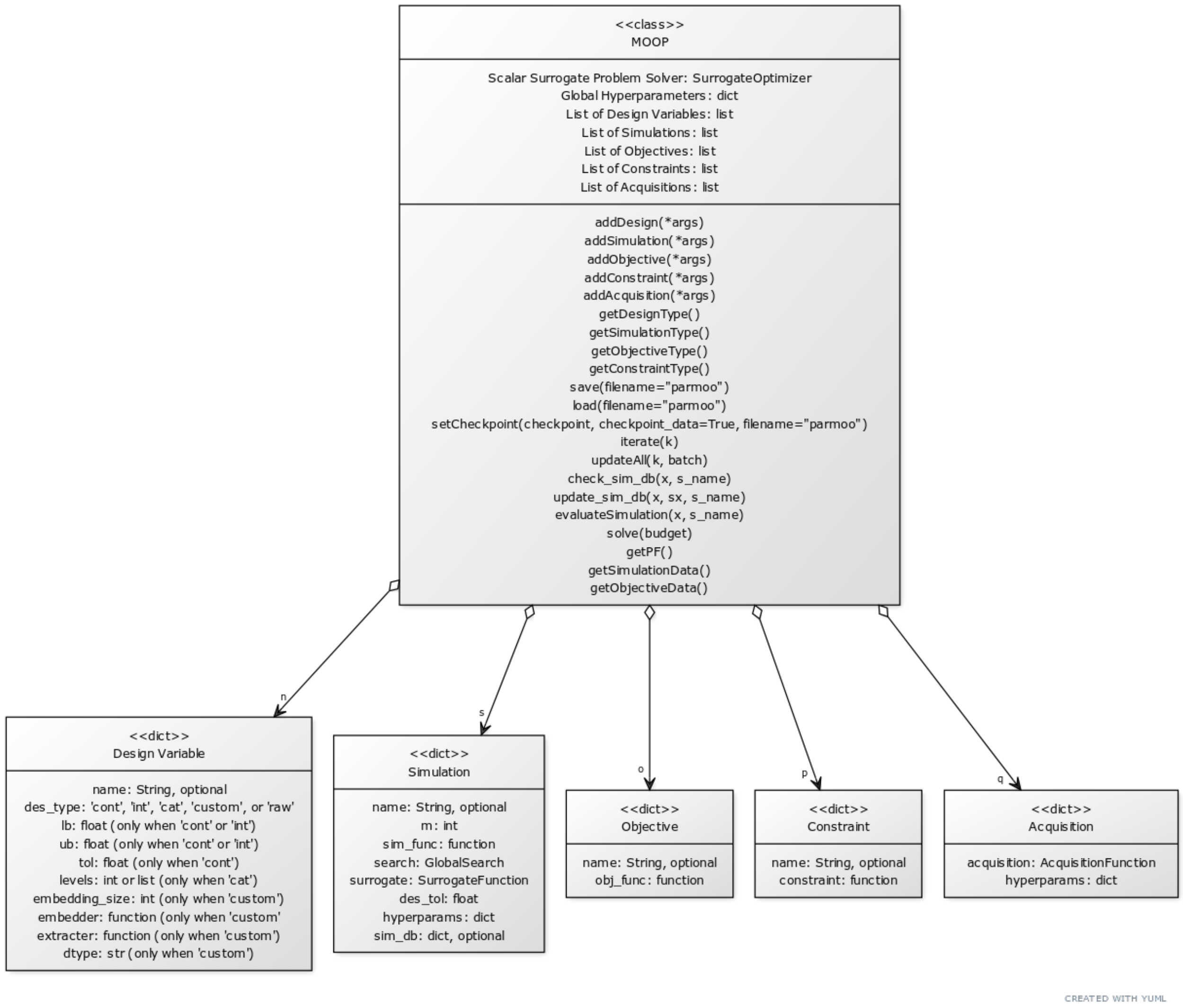
- **Acquisition function:** An *acquisition function* is our language for a family of scalarizing functions, which can be used to specify targets on the Pareto front. Acquisition functions may use objective scores, gradient values, and/or uncertainty information in order to guide ParMOO's search for an approximation to the Pareto front.
  - **Ex.–** common acquisition functions from the literature include weighted sums (averages) of objective values, the epsilon constraint method, and expected hypervolume improvement.
- **Design variable:** A *design variable* is an input to your simulations, which can be controlled within some reasonable bounds.
  - **Ex.-** when designing an air foil using a fluid dynamics simulation, one design variable might be the angle of attack.
  - ParMOO currently supports continuous and categorical design variables.
- **Design space:** The *design space* is the underlying vector space where you could represent all possible design variable combinations.
  - **Ex.-** if you have n continuous design variables, then your design space would be all of $\mathbb{R}^n$.
- **Constraint:** A *constraint* is a requirement that every solution point must satisfy.
  - **Ex.-** if your simulation code fails whenever $x_1 > x_2$, then you might impose the constraint: $x_2 - x_1 \leq 0$.
- **Hard constraint:** A *hard constraint* cannot be violated by the MOOP solver. ParMOO will never attempt to evaluate a point that violates a hard constraint.
  - **Ex.-** your simulation code does not need to be defined for points that are outside the upper/lower bounds on the design variables.
- **Soft constraint:** A *soft constraint* must be satisfied for a point to be considered a solution, but ParMOO may violate it during the course of the optimization process.
  - **Ex.-** all nonlinear constraints are soft constraints for ParMOO, and ParMOO will evaluate design points that violate these constraints, especially early in the optimization process.
- **Bound constraint:** A *bound constraint* is a simple upper/lower bound on the range of design values. In ParMOO, these are treated as hard constraints, while all other constraints are considered to be soft.
- **Feasible design space:** The *feasible design space* is the subset of the design space where all constraints (both hard and soft) are satisfied. In other words, this is the set of all "legal" designs.
  - **Ex.-** if you have *n* continuous design variables, constrained to the unit cube, then your entire design space is still R^n, but your *feasible* design space is the cube $[0, 1]^n$.
- **Simulation:** A *simulation* can refer to any complex process for generating scientific or engineering data. This includes both numerical simulations and laboratory experiments. The data that is gathered from your simulation might used to compute your objectives, constraints, or both.
  - **Ex.-** if you are designing a material, your simulation may be a molecular dynamics code **or** a process for synthesizing new materials in the laboratory.
  - Each simulation may have a single output or many outputs, which will be passed on as inputs to your objectives and/or constraints.
- **Objective:** An *objective* is one of possibly many criteria that you will use to rank the "goodness" of a particular design configuration. By convention, we assume that your goal is to minimize all objectives.

  - **Ex.-** if you are designing materials, you may want to minimize the production of unwanted byproducts.
  - If your goal is actually to maximize an objective f_{max}, you may supply the negated value of that objective $-f_{max}$ to ParMOO.
- **Feasible objective space:** The *feasible objective space* is the image of the feasible design space – i.e., the set of all objective values that can be obtained, by evaluating every objective at configurations from the feasible design space.
  - In practice, you will not know your feasible objective space *a priori*.
- **Nondominated:** A point $\mathbf{y}^*$ in a set $\mathcal{V} \subset \mathbb{R}^p$ is *nondominated* if for all $\mathbf{y} \in \mathcal{V}$, either $\mathbf{y} = \mathbf{y}^*$ or $\mathbf{y}^*$ is less than $\mathbf{y}$ in at least one of its p components.
  - Objective values that are feasible and nondominated in the set of all observations make up the solution set returned by ParMOO.
- **Pareto optimal:** A point in the feasible objective space is *Pareto optimal* for a given MOOP if it is nondominated in the feasible objective space.
  - This is a member of the true solution set for a MOOP.
  - In practice, we cannot typically guarantee that any point in a multiobjective simulation optimization problem is Pareto optimal, so we return solutions that are nondominated among all other objective values that we have observed.
- **Pareto front:** The *Pareto front* is the set of all Pareto optimal objective points.
  - This is the true solution to a multiobjective optimization problem.
- **Efficient set:** The *efficient set* is the set of all corresponding design configurations that produce points on the Pareto front.
  - These are the solutions in the feasible design space, which are the pre-image of the Pareto front.
- **Surrogate:** A *surrogate* is a computational model that approximates another underlying function.
  - **Ex.-** a trained artificial neural network, Gaussian process, RBF model, or spline interpolant.
- **Scalarization:** A *scalarization* technique reduces a MOOP into a single-objective optimization problem. Typically, solving the scalarized problem should produce a solution that is efficient/Pareto optimal.
  - **Ex.-** minimize the weighted sum of all objectives in a MOOP to obtain a single efficient point/Pareto optimal value.
- **Design of experiments/experimental design:** An *experimental design* is a set of design points that are in some sense space filling and could be evaluated to gain some initial data for a particular simulation.
  - **Ex.-** generate 100 uniform random samples within the feasible design space.

1.4 Write a ParMOO Script

1.4.1 The MOOP class

The `MOOP` class is the fundamental data structure in ParMOO. Below is a UML diagram showing the key public methods and dependencies.


<<class>>
MOOP
Scalar Surrogate Problem Solver: SurrogateOptimizer
Global Hyperparameters: dict
List of Design Variables: list
List of Simulations: list
List of Objectives: list
List of Constraints: list
List of Acquisitions: list
addDesign(*args)
addSimulation(*args)
addObjective(*args)
addConstraint(*args)
addAcquisition(*args)
getDesignType()
getSimulationType()
getObjectiveType()
getConstraintType()
save(filename="parmoo")
load(filename="parmoo")
setCheckpoint(checkpoint, checkpoint_data=True, filename="parmoo")
iterate(k)
updateAll(k, batch)
check_sim_db(x, s_name)
update_sim_db(x, sx, s_name)
evaluateSimulation(x, s_name)
solve(budget)
getPF()
getSimulationData()
getObjectiveData()
n
s
o
p
q
<<dict>>
Design Variable
name: String, optional
des_type: 'cont', 'int', 'cat', 'custom', or 'raw'
lb: float (only when 'cont' or 'int')
ub: float (only when 'cont' or 'int')
tol: float (only when 'cont')
levels: int or list (only when 'cat')
embedding_size: int (only when 'custom')
embedder: function (only when 'custom'
extracter: function (only when 'custom')
dtype: str (only when 'custom')
<<dict>>
Simulation
name: String, optional
m: int
sim_func: function
search: GlobalSearch
surrogate: SurrogateFunction
des_tol: float
hyperparams: dict
sim_db: dict, optional
<<dict>>
Objective
name: String, optional
obj_func: function
<<dict>>
Constraint
name: String, optional
constraint: function
<<dict>>
Acquisition
acquisition: AcquisitionFunction
hyperparams: dict
CREATED WITH YUML

To create an instance of the MOOP class, use the constructor.

```
from parmoo import MOOP
moop = MOOP(optimizer, hyperparams=hp)
```

In the above code snippet, `optimizer` should be an implementation of the SurrogateOptimizer Abstract-Base-Class (ABC), and the optional input `hp` is a dictionary of hyperparameters for the `optimizer` object. The `optimizer` is the surrogate optimization problem solver that will be used to generate candidate solutions for the MOOP. The choice of surrogate optimizer determines what information will be required when defining each objective and constraint.

- If you use a derivative-free technique, such as LocalGPS, then you do not need to provide derivative information for your objective or constraint functions.
- If you use a derivative-based technique, such as LBFGSB, then you need to provide an additional input to your objectives and constraint functions, which can be set to evaluate their derivatives with respect to design inputs and simulation outputs.

To avoid issues, it is best to define your MOOP in the following order.

1. Add design variables using `MOOP.addDesign(*args)`.
2. Add simulations using `MOOP.addSimulation(*args)`.
3. Add objectives using `MOOP.addObjective(*args)`.
4. Add constraints using `MOOP.addConstraint(*args)`.
5. Add acquisitions using `MOOP.addAcquisition(*args)`.

All of these methods accept one or more `args`, each of which is a dictionary, as detailed in the corresponding sections below.

1.4.2 The name Key and ParMOO Output Types

Each of the design, simulation, objective, and constraint dictionaries may contain an optional `name` key. By default, the `name` of the simulations, objectives, and constraints default to `{sim|f|c}i`, where `sim` is for a simulation, `f` is for an objective, `c` is for a constraint, and `i=1,2,...` is determined by the order in which each was added.

For example, if you add 3 simulations, then they will automatically be named `sim1`, `sim2`, and `sim3` unless a different name, was specified for one or more by including the `name` key. Similarly, objectives are named `f1`, `f2`, ..., and constraints are named `c1`, `c2`, ...

The design variables are the only exception to this rule.

Working with Named Outputs

When every design variable is given a name, then ParMOO formats its output in a numpy structured array, using the given/default names to specify each field. This operation mode is recommended, especially for first-time users.

After adding all design variables, simulations, objectives, and constraints to the MOOP, you can check the numpy dtype for each of these by using

- `MOOP.getDesignType()`,
- `MOOP.getSimulationType()`,
- `MOOP.getObjectiveType()`, and
- `MOOP.getConstraintType()`.

```
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

my_moop = MOOP(LocalGPS)

# Define a simulation to use below
def sim_func(x):
    if x["MyCat"] == 0:
        return np.array([(x["MyDes"]) ** 2, (x["MyDes"] - 1.0) ** 2])
    else:
        return np.array([99.9, 99.9])

# Add a design variable, simulation, objective, and constraint.
# Note the 'name' keys for each
my_moop.addDesign({'name': "MyDes",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "MyCat",
                   'des_type': "categorical",
                   'levels': 2})

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'name': "MyObj",
                      'obj_func': lambda x, s: sum(s["MySim"])})

my_moop.addConstraint({'name': "MyCon",
                       'constraint': lambda x, s: 0.1 - x["MyDes"]})

# Extract numpy dtypes for all of this MOOP's inputs/outputs
des_dtype = my_moop.getDesignType()
obj_dtype = my_moop.getObjectiveType()
sim_dtype = my_moop.getSimulationType()

# Display the dtypes as strings
print("Design variable type:   " + str(des_dtype))
print("Simulation output type: " + str(sim_dtype))
print("Objective type:         " + str(obj_dtype))
```

The result is the following.

```
Design variable type:   [('MyDes', '<f8'), ('MyCat', '<i4')]
Simulation output type: [('MySim', '<f8', (2,))]
```

(continues on next page)

(continued from previous page)

```
Objective type:         [('MyObj', '<f8')]
```

Working with Unnamed Outputs

If even a single design variable is left with a blank `name` key, then all input/output pairs are returned in a Python dictionary, with the following keys:

- `x_vals`: (`np.ndarray`) of length $d \times n$ – number of data points by number of design variables;
- `s_vals`: (`np.ndarray`) of length $d \times m$ – number of data points by number of outputs for a particular simulation;
- `f_vals`: (`np.ndarray`) of length $d \times o$ – number of data points by number of objectives;
- `c_vals`: (`np.ndarray`) of length $d \times c$ – number of data points by number of constraints, if any.

Note that the value of d (number of data points0 may vary by database). Each column in each of `x_vals`, `s_vals`, `f_vals`, and `c_vals` will correspond to a specific design variable, simulation output, objective function, or constraint, determined by the order in which they were added to the MOOP.

For first-time users, this execution mode may be confusing; however, for advanced users, the convenience of using numpy.ndarrays over structured arrays may be preferable.

You can still use the type-getter methods from the previous section to check the dtype of each output, knowing that

- `MOOP.getDesignType()` is the dtype of the `x_vals` key (when present).
- `MOOP.getSimulationType()` is the dtype of the `s_vals` key (when present),
- `MOOP.getObjectiveType()` is the dtype of the `f_vals` key (when present), and
- `MOOP.getConstraintType()` is the dtype of the `c_vals` key (when present).

```python

import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

# Fix the random seed for reproducibility
np.random.seed(0)

my_moop = MOOP(LocalGPS)

# Define a simulation to use below
def sim_func(x):
    return np.array([(x[0]) ** 2, (x[0] - 1.0) ** 2])

# Add a design variable, simulation, objective, and constraint, w/o name key
my_moop.addDesign({'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})

my_moop.addSimulation({'m': 2,
                       'sim_func': sim_func,
```

(continues on next page)

(continued from previous page)

```
                        'search': LatinHypercube,
                        'surrogate': GaussRBF,
                        'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'obj_func': lambda x, s: sum(s)})

my_moop.addConstraint({'constraint': lambda x, s: 0.1 - x[0]})

# Extract numpy dtypes for all of this MOOP's inputs/outputs
des_dtype = my_moop.getDesignType()
sim_dtype = my_moop.getSimulationType()
obj_dtype = my_moop.getObjectiveType()
const_dtype = my_moop.getConstraintType()

# Display the dtypes as strings
print("Design variable type:   " + str(des_dtype))
print("Simulation output type: " + str(sim_dtype))
print("Objective type:         " + str(obj_dtype))
print("Constraint type:        " + str(const_dtype))
print()

# Add one acquisition and solve with 0 iterations to initialize databases
my_moop.addAcquisition({'acquisition': UniformWeights})
my_moop.solve(0)

# Extract final objective and simulation databases
obj_db = my_moop.getObjectiveData()
sim_db = my_moop.getSimulationData()

# Print the objective database dtypes
print("Objective database keys: " + str([key for key in obj_db.keys()]))
for key in obj_db.keys():
    print("\t'" + key + "'" + " dtype: " + str(obj_db[key].dtype))
    print("\t'" + key + "'" + " shape: " + str(obj_db[key].shape))
print()

# Print the simulation database dtypes
print("Simulation database type: " + str(type(sim_db)))
print("Simulation database length: " + str(len(sim_db)))
for i, dbi in enumerate(sim_db):
    print("\tsim_db[" + str(i) + "] database keys: " +
          str([key for key in dbi.keys()]))
    for key in dbi.keys():
        print("\t\t'" + key + "'" + " dtype: " + str(dbi[key].dtype))
        print("\t\t'" + key + "'" + " shape: " + str(dbi[key].shape))
```

The result is below. Note that in this example, there are $d = 20$ points in both the objective and simulation databases.

```
Design variable type:   ('<f8', (1,))
Simulation output type: ('<f8', (2,))
Objective type:         ('<f8', (1,))
Constraint type:        ('<f8', (1,))
```

(continues on next page)

(continued from previous page)

```

Objective database keys: ['x_vals', 'f_vals', 'c_vals']
        'x_vals' dtype: float64
        'x_vals' shape: (20, 1)
        'f_vals' dtype: float64
        'f_vals' shape: (20, 1)
        'c_vals' dtype: float64
        'c_vals' shape: (20, 1)

Simulation database type: <class 'list'>
Simulation database length: 1
        sim_db[0] database keys: ['x_vals', 's_vals']
                'x_vals' dtype: float64
                'x_vals' shape: (20, 1)
                's_vals' dtype: float64
                's_vals' shape: (20, 2)
```

1.4.3 Adding Design Variables

Design variables are added to your `MOOP` object using the `addDesign(*args)` method. ParMOO currently supports several types of design variables:

- `continuous` (or `real` or `cont`),
- `integer` (or `int`),
- `categorical` (or `cat`),
- `custom`,
- `raw` – not recommended, for advanced users only.

To add a continuous variable, use the following format.

```
# Add a continuous design variable
moop.addDesign({'name': "MyContVar", # optional
                'des_type': "continuous",
                'lb': 0.0,
                'ub': 1.0,
                'des_tol': 1.0e-8})
```

- Note that when the `des_type` key is omitted, its value defaults to `continuous`.
- For continuous design variables, both a lower (`lb`) and upper (`ub`) bound must be specified. These bounds are *hard constraints*, meaning that no simulations or objectives will be evaluated outside of these bounds.
- The optional key `des_tol` specifies a minimum step size between values for this design variable (default value is 10^{-8}). For this design variable, any two values that are closer than `des_tol` will be treated as exactly equal.

To add an integer design variable, use the following format.

```python
# Add an integer design variable
moop.addDesign({'name': "MyIntVar", # optional
                'des_type': "integer",
                'lb': 0,
                'ub': 100})
```

- The lb and ub keys must be integer-valued, and serve the same purpose as with continuous design variables.

To add a categorical design variable, use the following format.

```python
# Add a categorical design variable
moop.addDesign({'name': "MyCatVar", # optional
                'des_type': "categorical",
                'levels': 3})
```

- The levels key is either an integer specifying the number of categories taken on by this design variable (ParMOO will index these levels by $0, 1, \ldots, \text{levels} - 1$) or a list of strings specifying the name for each category (ParMOO will use these names for the levels, e.g., ["first cat", "second cat", ...]).

Note that because a numpy ndarray cannot contain string entries, when operating with unnamed variables, the levels key may only contain the integer number of levels, and named categories cannot be used with unnamed design variables.

To add a custom design variable, use the following format.

```python
# Add a custom design variable
moop.addDesign({'name': "MyCustomVar", # optional
                'des_type': "custom",
                'embedding_size': 1,
                'embedder': my_embedding_func,
                'extracter': my_extracting_func,
                'dtype': "U25" # optional
                })
```

- The embedding_size key tells ParMOO how many dimensions the embedding for this variable will be.
- The embedder key should be a function that maps the input type to to a point in the embedding_size-dimensional unit hypercube.
- The extracter key should be a function that maps an arbitrary point in the embedding_size-dimensional unit hypercube back to the input type (such that extracter(embedder(x)) = x).
- Optionally, the dtype key is a Python str specifying the numpy dtype of the input (defaults to U25, i.e., a maximum 25-character string).

Note that because a numpy ndarray cannot contain string entries, when operating with unnamed variables, the `dtype` key is ignored, and the input must have a numeric type.

To add a raw design variable, use the following format. Please note that raw design variables are not recommended, and one will typically need to write custom `search`, `surrogate`, `optimizer`, and `acquisition` functions/classes to accomodate a raw variable. This feature is only included to allow flexibility for expert users.

```
# Add a raw design variable
moop.addDesign({'name': "MyRawVar", # optional
                'des_type': "raw"})
```

Note that for every MOOP, at least one design variable is required before solving.

1.4.4 Adding Simulations

Before you can add a simulation to your `MOOP`, you must define the simulation function.

The simulation function can be either a Python function or a callable object.

The expected signature of your simulation function depends on whether you are working with `named or unnamed outputs`.

When working with named variables, the simulation should take a single numpy structured array as input, whose keys match the design variable names. The simulation function returns a numpy.ndarray containing the simulation output(s).

For example, with three design variables named `x1`, `x2`, and `x3`, you might define the quadratic $\mathbf{S}(\mathbf{x}) = \|\mathbf{x}\|^2$ as follows.

```
def quadratic_sim(x):
    return np.array([x["x1"] ** 2 + x["x2"] ** 2 + x["x3"] ** 2])
```

If you are working with unnamed variables, then the simulation will accept a numpy.ndarray of length `n`, where the indices correspond to the order in which the design variables were added to the MOOP. The example above would change as follows.

```
def quadratic_sim(x):
    return np.array([x[0] ** 2 + x[1] ** 2 + x[2] ** 2])
```

To add your simulation to the `MOOP` object, use the `addSimulation(*args)` method.

```
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF

moop.addSimulation({'name': "MySim", # optional
                    'm': 1, # number of outputs
                    'sim_func': quadratic_sim, # simulation function
                    'search': LatinHypercube, # search technique
                    'surrogate': GaussRBF, # surrogate model
                    'hyperparams': {'search_budget': 20}})
```

In the above example,

- `name` is used as described in `named or unnamed outputs`;

- m specifies the number of outputs for this simulation;
- sim_func is given a reference to the simulation function;
- search specifies the GlobalSearch that you will use when generating data for this particular simulation;
- surrogate specifies the class of SurrogateFunction that you will use to model this particular simulation's output;
- hyperparams is a dictionary of hyperparameter values that will be passed to the surrogate and search technique objects. One particularly important key in the hyperparams dictionary is the search_budget key, which specifies how many simulation evaluations should be used during the initial search phase.

If you wish, you may create a MOOP without any simulations.

Using a Precomputed Simulation Database

If you would like to specify a precomputed database, use the MOOP.update_sim_db(x, sx, s_name) method to add all simulation data into ParMOO's database after creating your MOOP but before solving. Be careful not to add duplicate points, because these could cause numerical issues when fitting surrogate models.

```
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

my_moop = MOOP(LocalGPS)

my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': 3})

def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'name': "f1", 'obj_func': lambda x, s: s["MySim"][0]})
my_moop.addObjective({'name': "f2", 'obj_func': lambda x, s: s["MySim"][1]})

my_moop.addAcquisition({'acquisition': UniformWeights})
```

(continues on next page)

(continued from previous page)

```python
# Precompute one simulation value for demo
des_val = np.zeros(1, dtype=[("x1", float), ("x2", int)])[0]
sim_val = sim_func(des_val)

# Add the precomputed simulation value from above
my_moop.update_sim_db(des_val, sim_val, "MySim")

# Get and display initial database
sim_db = my_moop.getSimulationData()
print(sim_db)
```

The output of the above code is shown below.

```
{'MySim': array([(0., 0, [0.04, 0.64])],
      dtype=[('x1', '<f8'), ('x2', '<i4'), ('out', '<f8', (2,))])}
```

1.4.5 Adding Objectives

Objectives are algebraic functions of your design variables and simulation outputs. *ParMOO always minimizes objectives.* If you would like to maximize instead, re-define the problem by minimizing the negative-value of your objective.

Just like with simulation functions, ParMOO accepts either a Python function or a callable object for each objective. Make sure you match the expected signature, which depends on whether you are using `named or unnamed outputs`.

For named outputs, your objective function should accept two numpy structured arrays and return a single scalar output. The following objective minimizes the output of the simulation output named `MySim`.

```python
def min_sim(x, sim):
    return sim["MySim"]
```

Similarly, the following objective minimizes the squared value of the design variable named `MyDes`.

```python
def min_des(x, sim):
    return x["MyDes"] ** 2
```

If you are using a gradient-based `SurrogateOptimizer`, then you are required to supply an additional input named `der`, which defaults to 0. The `der` input is used as follows:

- `der=0` (default) implies that no derivatives are taken, and you will return the objective function value;
- `der=1` implies that you will return an array of derivatives with respect to each design variable; and
- `der=2` implies that you will return an array of derivative with respect to each simulation output.

Note that for categorical variables ParMOO does not use the partial derivatives given here, and it is acceptable to fill these slots with a garbage value or leave them uninitialized.

Modifying the above two objectives to support derivative-based solvers, we get the following.

```python
def min_sim(x, sim, der=0):
    if der == 0:
        # No derivative, just return the value of sim["MySim"]
        return sim["MySim"]
    elif der == 1:
        # Derivative wrt each design variable is 0
```

(continues on next page)

(continued from previous page)

```python
        return np.zeros(1, x.dtype)[0]
    elif der == 2:
        # Derivative wrt other simulations is 0, but df/d"MySim"=1
        result = np.zeros(1, sim.dtype)[0]
        result["MySim"] = 1.0
        return  result

def min_des(x, sim, der=0):
    if der == 0:
        # No derivative, just return the value of x["MyDes"] ** 2
        return x["MyDes"] ** 2
    elif der == 1:
        # Derivative wrt other design vars is 0, but df/d"MyDes"=2"MyDes"
        result = np.zeros(1, x.dtype)[0]
        result["MyDes"] = 2.0 * x["MyDes"]
        return result
    elif der == 2:
        # Derivative wrt each simulations is 0
        return np.zeros(1, sim.dtype)[0]
```

For a full example showing how to solve a MOOP using a derivative-based solver, see `Solving a MOOP with Derivative-Based Solvers` in `Basic Tutorials`.

When using ParMOO with unnamed outputs, each objective should accept two 1D numpy.ndarrays instead. The above example would be modified as follows, assuming that `MySim` was the first simulation and `MyDes` was the first design variable added to the MOOP.

```python
def min_sim(x, sim, der=0):
    if der == 0:
        # No derivative, just return the value of sim[0]
        return sim[0]
    elif der == 1:
        # Derivative wrt each design variable is 0
        return np.zeros(x.size)
    elif der == 2:
        # Derivative wrt other simulations is 0, but df/dsim[0]=1
        return np.eye(sim.size)[0]

def min_des(x, sim, der=0):
    if der == 0:
        # No derivative, just return the value of x["MyDes"] ** 2
        return x[0] ** 2
    elif der == 1:
        # Derivative wrt other design vars is 0, but df/d"MyDes"=2"MyDes"
        result = np.zeros(x.size)
        result[0] = 2.0 * x[0]
        return result
    elif der == 2:
        # Derivative wrt each simulations is 0
        return np.zeros(sim.size)
```

To add the objective(s), use the `MOOP.addObjective(*args)` method.

```
moop.addObjective({'name': "Min MySim",
                   'obj_func': min_sim})

moop.addObjective({'name': "Min MyDes",
                   'obj_func': min_des})
```

Note that for every MOOP, at least one objective is required before solving.

1.4.6 Adding Constraints

Adding constraints is similar to adding objectives. The main difference is in how ParMOO treats constraint functions. Although ParMOO may evaluate infeasible design points along the way, ParMOO will search for solutions where all constraints are less than or equal to zero.

For example, to add the constraint that the simulation `MySim` must have output greater than or equal to 0 and that the design variable `MyDes` must be less than or equal to 0.9, you would define the following constraint functions.

```
def sim_constraint(x, sim):
    return -1.0 * sim["MySim"]

def des_constraint(x, sim):
    return x["MyDes"] - 0.9
```

As with objectives, if you want to use a gradient-based `SurrogateOptimizer`, you must modify the above constraint functions as follows.

```
def sim_constraint(x, sim, der=0):
    if der == 0:
        return -1.0 * sim["MySim"]
    elif der == 1:
        return np.zeros(1, x.dtype)[0]
    elif der == 2:
        result = np.zeros(1, sim.dtype)[0]
        result["MySim"] = -1.0
        return result

def des_constraint(x, sim, der=0):
    if der == 0:
        return x["MyDes"] - 0.9
    elif der == 1:
        result =  np.zeros(1, x.dtype)[0]
        result["MyDes"] = 1.0
        return result
    elif der == 2:
        return np.zeros(1, sim.dtype)[0]
```

If you are operating with unnamed variables, use indices similarly as with the objectives.

```
def sim_constraint(x, sim, der=0):
    if der == 0:
        return -1.0 * sim[0]
    elif der == 1:
        return np.zeros(x.size)
```

(continues on next page)

(continued from previous page)

```
    elif der == 2:
        return -np.eye(sim.size)[0]

def des_constraint(x, sim, der=0):
    if der == 0:
        return x[0] - 0.9
    elif der == 1:
        return np.eye(x.size)[0]
    elif der == 2:
        return np.zeros(sim.size)
```

To add the constraint(s), use the `MOOP.addConstraint(*args)` method.

```
moop.addConstraint({'name': "Constrain MySim",
                    'constraint': sim_constraint})

moop.addConstraint({'name': "Constrain MyDes",
                    'constraint': des_constraint})
```

You are not required to add any constraints of this form to your MOOP before solving.

1.4.7 Adding Acquisitions

After you have added all of the design variables, simulations, objectives, and constraints to your MOOP, you must add one or more acquisitions using the `MOOP.addAcquisition(*args)` method.

```
from parmoo.acquisitions import RandomConstraint, FixedWeights

moop.addAcquisition({'acquisition': RandomConstraint})
moop.addAcquisition({'acquisition': FixedWeights,
                     'hyperparams': {'weights': np.array([0.5, 0.5])}})
```

The acquisition dictionary may contain two keys:

- `acquisition` (required) specifies one `AcquisitionFunction` that you would like to use for this problem; and
- `hyperparams` (optional) specifies a dictionary of hyperparameter values that are used by the specified `AcquisitionFunction`.

The number of acquisitions added determines the batch size for each of ParMOO's batches of simulation evaluations (which could be done in parallel). **In general, if there are q acquisition functions and s simulations, then ParMOO will generate batches of q*s simulations**. In other words, each simulation is evaluated once per acquisition function in each iteration of ParMOO's algorithm.

1.4.8 Logging and Checkpointing

When solving large or expensive problems, it is often a good idea to activate ParMOO's logging and/or checkpointing features.

Logging

For diagnostics, ParMOO logs its progress at the `logging.INFO` level. To display these log messages, turn on Python's `INFO`-level logging.

```
import logging
logging.basicConfig(level=logging.INFO)
```

If you would like to also print a formatted timestamp, use the Python logger's built-in formatting options.

```
import logging
logging.basicConfig(level=logging.INFO,
                    [format='%(asctime)s %(levelname)-8s %(message)s',
                     datefmt='%Y-%m-%d %H:%M:%S'])
```

Be aware that when using ParMOO together with libEnsemble, `libE` already comes with its own logging tools, which are recommended, and ParMOO's logging tools will not work.

Checkpointing

A ParMOO can be run with checkpointing turned on, so that your MOOP can be paused and resumed later, and your simulation data can be recovered after a crash. Checkpointing is off by default. To turn it on, use the method:

- `setCheckpoint(checkpoint, [checkpoint_data, filename])`

```
moop.setCheckpoint(True, checkpoint_data=True, filename="parmoo")
```

The first argument tells ParMOO to save its internal class attributes and databases, so that they can be reloaded in the future. In the above example, this save data will be written to a file in the calling directory, with the name `parmoo.moop`.

In order to save the problem definition, ParMOO needs to store information for reloading all of your functions. For this to work:

- All functions (such as simulation functions, objective functions, and constraint functions) are defined in the global scope;
- All modules are reloaded before attempting to recover a previously-saved MOOP object (by calling the `load(filename)` method);
- ParMOO cannot reload `lambda` functions. Use only regular functions and callable objects when checkpointing.

If the option argument `checkpoint_data` is set to `True` (default), the ParMOO will also save a second copy of all simulation evaluations in a human-readable JSON file in the same directory, with the name `parmoo.simdb.json`. This file is not used by ParMOO, it is only provided for user-convenience.

Reloading After Crash or Early Stop

After a crash or early termination, reload the saved `.moop` file to resume. **Make sure that you first import any external modules and redefine any functions that are needed by ParMOO (with the exact same signatures).**

```python
from parmoo import MOOP
from optimizers import [optimizer]

# Create a new MOOP object
moop = MOOP([optimizer])
# Reload the old problem
moop.load(filename="parmoo") # Use your savefile name, omitting ".moop"
```

Then resume your solve with an increased budget.

```python
# Resume solve with increased budget
moop.solve(6)
```

Example

The example below shows how the Quickstart demo can be modified to use logging and checkpointing, including an example of how to load a MOOP from a saved checkpoint file and resume running.

```python

import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS
import logging

# Fix the random seed for reproducibility
np.random.seed(0)

# Create a new MOOP
my_moop = MOOP(LocalGPS)

# Add 1 continuous and 1 categorical design variable
my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': 3})

# Create a simulation function
def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

# Add the simulation function to the MOOP
```

(continues on next page)

(continued from previous page)

```
my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

# Define the 2 objectives as named Python functions
def obj1(x, s): return s["MySim"][0]
def obj2(x, s): return s["MySim"][1]

# Define the constraint as a function
def const(x, s): return 0.1 - x["x1"]

# Add 2 objectives
my_moop.addObjective({'name': "f1", 'obj_func': obj1})
my_moop.addObjective({'name': "f2", 'obj_func': obj2})

# Add 1 constraint
my_moop.addConstraint({'name': "c1", 'constraint': const})

# Add 3 acquisition functions (generates batches of size 3)
for i in range(3):
   my_moop.addAcquisition({'acquisition': UniformWeights,
                           'hyperparams': {}})

# Turn on logging with timestamps
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)-8s %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S')

# Use checkpointing without saving a separate data file (in "parmoo.moop" file)
my_moop.setCheckpoint(True, checkpoint_data=False, filename="parmoo")

# Solve the problem with 4 iterations
my_moop.solve(4)

# Create a new MOOP object and reload the MOOP from parmoo.moop file
new_moop = MOOP(LocalGPS)
new_moop.load("parmoo")

# Do another iteration
new_moop.solve(5)

# Display the solution
results = new_moop.getPF()
print(results, "\n dtype=" + str(results.dtype))
```

The result is the following.

```
[(0.79013666, 0, 3.48261275e-01, 9.72855201e-05, -0.69013666)
 (0.7895263 , 0, 3.47541259e-01, 1.09698380e-04, -0.6895263 )
```

(continues on next page)

(continued from previous page)

```
(0.73488156, 0, 2.86098283e-01, 4.24041125e-03, -0.63488156)
(0.70656124, 0, 2.56604287e-01, 8.73080244e-03, -0.60656124)
(0.68409101, 0, 2.34344111e-01, 1.34348928e-02, -0.58409101)
(0.67237225, 0, 2.23135544e-01, 1.62888423e-02, -0.57237225)
(0.5784217 , 0, 1.43202981e-01, 4.90969442e-02, -0.4784217 )
(0.53031761, 0, 1.09109723e-01, 7.27285920e-02, -0.43031761)
(0.51322778, 0, 9.81116425e-02, 8.22383058e-02, -0.41322778)
(0.49723345, 0, 8.83477213e-02, 9.16675863e-02, -0.39723345)
(0.44987019, 0, 6.24351129e-02, 1.22590882e-01, -0.34987019)
(0.40591372, 0, 4.24004606e-02, 1.55303995e-01, -0.30591372)
(0.39028872, 0, 3.62097978e-02, 1.67863331e-01, -0.29028872)
(0.38995793, 0, 3.60840145e-02, 1.68134501e-01, -0.28995793)
(0.38287784, 0, 3.34443041e-02, 1.73990897e-01, -0.28287784)
(0.25435646, 0, 2.95462529e-03, 2.97726867e-01, -0.15435646)
(0.20137796, 0, 1.89878434e-06, 3.58348342e-01, -0.10137796)]
dtype=[('x1', '<f8'), ('x2', '<i4'), ('f1', '<f8'), ('f2', '<f8'), ('c1', '<f8')]
```

1.4.9 Methods for Solving

Once you have finished creating your `MOOP` object and adding all design variables, simulations, objectives, constraints, and acquisitions, you are ready to solve your problem.

The easiest way to solve is by using `MOOP.solve(k)`. Here, `k` is the number of iterations of ParMOO's algorithm that you would like to perform. Note that a value of `k=0` is legal, and will result in ParMOO generating and evaluating an experimental design and fitting its surrogates, without ever attempting to solve a single scalarized surrogate problems.

```
# Evaluate an experimental design, then performing 5 iterations
moop.solve(5)
```

Note that the above command will perform all simulation evaluations serially. To generate a batch of simulations that you could evaluate in parallel, use `MOOP.iterate(k)`, where `k` is the iteration index. You can let ParMOO handle the simulation evaluations with `MOOP.evaluateSimulation(x, s_name)`, or you can evaluate the simulations yourself and add them to the simulation database using `MOOP.update_sim_db(x, sx, s_name)`. Afterward, call `MOOP.updateAll(k, batch)` to update the surrogate models and objective database.

```
# Do 5 iterations letting ParMOO handle simulation evaluation
# Note that the i=0 iteration will just generate an experimental design
for i in range(5):
    # Get batch
    batch = moop.iterate(i)
    # Let ParMOO evaluate design point x for simulation s_name
    for (x, s_name) in batch:
        moop.evaluateSimulation(x, s_name)
    # Update ParMOO models
    moop.updateAll(i, batch)
```

or

```
# Solve another MOOP, doing simulation evaluation manually
for i in range(5):
    # Get batch
    batch = moop.iterate(i)
```

(continues on next page)

(continued from previous page)

```
    # User evaluates design point x for simulation s_name
    for (x, s_name) in batch:
        ### User code to evaluate x with sim["s_name"] goes HERE ###
        ### Store results in variable sx ###
        moop.update_sim_db(x, sx, s_name)
    # Update ParMOO models
    moop.updateAll(i, batch)
```

Additional ParMOO solver execution paradigms (including those where ParMOO will handle parallel execution on the user's behalf) are included under `Additional ParMOO Plugins and Features`.

1.4.10 Viewing Your Results

After solving the MOOP, you can view the results using `MOOP.getPF()`.

```
soln = moop.getPF()
```

The output format defaults to a numpy structured array. However, you can change it to a pandas dataframe using the optional `format` argument.

```
soln = moop.getPF(format="pandas")
```

Note that `format="pandas"` is only supported when working with `named outputs`.

To get the full simulation and objective databases, you can also use `MOOP.getSimulationData()` and `MOOP.getObjectiveData()`.

```
sim_db = moop.getSimulationData()
obj_db = moop.getObjectiveData()
```

To understand the format of these outputs, please revisit the section on `The name Key and ParMOO Output Types`.

Finally, if you have installed ParMOO with its extra dependencies (see the Advanced Installation), then you can visualize your results using any of the `viz.scatter()`, `viz.parallel_coordinates()`, or `viz.radar()` functions.

```
from parmoo.viz import scatter

scatter(moop)
```

Note that these plots are interactive and will render in a Dash app hosted locally on your computer. There are known issues when using the Chrome browser.

For more information, view the complete `viz API page`.

1.4.11 Built-in and Custom Components

By now you can see that the performance of ParMOO is determined by your choices of

- AcquisitionFunction,
- GlobalSearch,
- SurrogateFunction, and
- SurrogateOptimizer.

You can find the current options for each of these in the following modules.

- parmoo.acquisitions
- parmoo.searches
- parmoo.surrogates
- parmoo.optimizers

You can also create your own custom implementations for each of the above, by implementing one of the abstract base classes in structs.

1.5 Extras and Plugins

1.5.1 Running in Parallel using libEnsemble

libEnsemble is a Python library to "coordinate concurrent evaluation of dynamic ensembles of calculations." Read more about libEnsemble by visiting the libEnsemble documentation.

The libE_MOOP class is used to solve MOOPs using libEnsemble. The libE_MOOP class inherits from MOOP and supports all of the public methods in its API.

To create an instance of the libE_MOOP class, import it from the extras.libe module and then create a MOOP, just as you normally would. The solve() method has been redefined to create a libEnsemble Persistent Generator function, which libEnsemble can call to generate batches of simulation evaluations, which it will distribute over available resources.

Below we reproduce the example from the Quickstart guide, using a libE_MOOP object.

Note that it is always recommended that you turn on checkpointing when using libEnsemble. Since setCheckpoint method does not support the usage of Python lambda functions, each of the objectives and constraints is explicitly defined.

```python
import numpy as np
from parmoo.extras.libe import libE_MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

# When running with MPI, we need to keep track of which thread is the manager
# using libensemble.tools.parse_args()
from libensemble.tools import parse_args
_, is_manager, _, _ = parse_args()
```

(continues on next page)

(continued from previous page)

```python

# All functions are defined below.

def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

def obj_f1(x, s):
    return s["MySim"][0]

def obj_f2(x, s):
    return s["MySim"][1]

def const_c1(x, s):
    return 0.1 - x["x1"]

# When using libEnsemble with Python MP, the "solve" command must be enclosed
# in an "if __name__ == '__main__':" block, as shown below
if __name__ == "__main__":
    # Fix the random seed for reproducibility
    np.random.seed(0)

    # Create a libE_MOOP
    my_moop = libE_MOOP(LocalGPS)

    # Add 2 design variables (one continuous and one categorical)
    my_moop.addDesign({'name': "x1",
                       'des_type': "continuous",
                       'lb': 0.0, 'ub': 1.0})
    my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                       'levels': 3})

    # Add the simulation (note the budget of 20 sim evals during search phase)
    my_moop.addSimulation({'name': "MySim",
                           'm': 2,
                           'sim_func': sim_func,
                           'search': LatinHypercube,
                           'surrogate': GaussRBF,
                           'hyperparams': {'search_budget': 20}})

    # Add the objectives
    my_moop.addObjective({'name': "f1", 'obj_func': obj_f1})
    my_moop.addObjective({'name': "f2", 'obj_func': obj_f2})

    # Add the constraint
    my_moop.addConstraint({'name': "c1", 'constraint': const_c1})

    # Add 3 acquisition functions
    for i in range(3):
       my_moop.addAcquisition({'acquisition': UniformWeights,
```

(continues on next page)

(continued from previous page)

```
                                          'hyperparams': {}})

    # Turn on checkpointing -- creates files parmoo.moop & parmoo.surrogate.1
    my_moop.setCheckpoint(True, checkpoint_data=False, filename="parmoo")

    # Use sim_max = 30 to perform just 30 simulations
    my_moop.solve(sim_max=30)

    # Display the solution -- this "if" clause is needed when running with MPI
    if is_manager:
        results = my_moop.getPF(format="pandas")
        print(results)
```

To run a ParMOO/libEnsemble script, first make sure that libEnsemble is installed. You can find instructions on how to do so under libEnsemble's Advanced Installation documentation.

Next, run libEnsemble as described in the Running libEnsemble section. Common methods of running a libEnsemble script are with MPI

```
mpirun -np N python3 libe_basic_ex.py
```

and with Python's built-in multiprocessing module.

```
python3 libe_basic_ex.py --comms local --nworkers N
```

- Note: When running a `libE_MOOP` with Python multiprocessing, MacOS and Windows systems default to using the `spawn` method. When using the `spawn` method, one must enclose the `libE_MOOP.solve()` command inside an `if __name__ == '__main__':` block, as shown in the example above. Read more about the issue here:

  https://libensemble.readthedocs.io/en/main/running_libE.html#local-comms

The result from running the example is shown below.

```
         x1   x2        f1        f2        c1
0  0.742825   0  0.294659  0.003269 -0.642825
1  0.680283   0  0.230672  0.014332 -0.580283
2  0.616501   0  0.173473  0.033672 -0.516501
3  0.580369   0  0.144680  0.048238 -0.480369
4  0.555222   0  0.126183  0.059916 -0.455222
5  0.518980   0  0.101749  0.078972 -0.418980
6  0.475477   0  0.075888  0.105315 -0.375477
7  0.302503   0  0.010507  0.247503 -0.202503
8  0.201285   0  0.000002  0.358460 -0.101285
```

1.6 References

1.6.1 Additional Resources

To seek support or report issues, e-mail:

- parmoo@mcs.anl.gov

For permissions, see the following:

- LICENSE
- CONTRIBUTING

Our full user guide is hosted here:

- ReadTheDocs

1.6.2 Reference Format

Please use one of the following to cite ParMOO.

Our JOSS paper:

```
@article{parmoo,
    author={Chang, Tyler H. and Wild, Stefan M.},
    title={{ParMOO}: A {P}ython library for parallel multiobjective simulation␣
↪optimization},
    year={2023},
    journal = {Journal of Open Source Software},
    volume = {8},
    number = {82},
    pages = {4468},
    doi = {10.21105/joss.04468}
}
```

Our online documentation:

```
@techreport{parmoo-docs,
    title       = {{ParMOO}: {P}ython library for parallel multiobjective simulation␣
↪optimization},
    author      = {Chang, Tyler H. and Wild, Stefan M. and Dickinson, Hyrum},
    institution = {Argonne National Laboratory},
    number      = {Version 0.3.1+dev},
    year        = {2023},
    url         = {https://parmoo.readthedocs.io/en/latest}
}
```

Chapter 2

API

2.1 ParMOO API

2.1.1 Basic ParMOO Classes/Objects

ParMOO uses an object-oriented design, where users start by creating a `MOOP` object, defining their problem.

The type of `MOOP` that you use determines what kind of resources you will use to solve the problem.

Current options are:

Base (serial) MOOP Class

This is the base class for solving MOOPs with ParMOO.

```
from parmoo import moop
```

Use this class to define and solve a MOOP. The `MOOP.solve(...)` method will perform simulations serially for this class.

Contains the MOOP class for defining multiobjective optimization problems.

parmoo.moop.MOOP is the base class for defining and solving multiobjective optimization problems (MOOPs). Each MOOP object may contain several simulations, specified using dictionaries.

class moop.**MOOP**(*opt_func*, *hyperparams=None*)

Class for defining a multiobjective optimization problem (MOOP).

Upon initialization, supply a scalar optimization procedure and dictionary of hyperparameters using the default constructor:

- `MOOP.__init__(ScalarOpt, [hyperparams={}])`

Class methods are summarized below.

To define the MOOP, add each design variable, simulation, objective, and constraint (in that order) by using the following functions:

- `MOOP.addDesign(*args)`
- `MOOP.addSimulation(*args)`

- `MOOP.addObjective(*args)`
- `MOOP.addConstraint(*args)`

Next, define your solver.

Acquisition functions (used for scalarizing problems/setting targets) are added using:

- `MOOP.addAcquisition(*args)`

After creating a MOOP, the following methods may be useful for getting the numpy.dtype of the input/output arrays:

- `MOOP.getDesignType()`
- `MOOP.getSimulationType()`
- `MOOP.getObjectiveType()`
- `MOOP.getConstraintType()`

The following methods are used to save/load ParMOO objects from memory:

- `MOOP.save([filename="parmoo"])`
- `MOOP.load([filename="parmoo"])`

To turn on checkpointing, use:

- `MOOP.setCheckpoint(checkpoint, [checkpoint_data, filename])`

ParMOO also offers logging. To turn on logging, activate INFO-level logging by importing Python's built-in logging module.

After defining the MOOP and setting up checkpointing and logging info, use the following method to solve the MOOP (serially):

- `MOOP.solve(iter_max=None, sim_max=None)`

The following methods are used for solving the MOOP and managing the internal simulation/objective databases:

- `MOOP.check_sim_db(x, s_name)`
- `MOOP.update_sim_db(x, sx, s_name)`
- `MOOP.evaluateSimulation(x, s_name)`
- `MOOP.addData(x, sx)`
- `MOOP.iterate(k, ib)`
- `MOOP.updateAll(k, batch)`

Finally, the following methods are used to retrieve data after the problem has been solved:

- `MOOP.getPF(format='ndarray')`
- `MOOP.getSimulationData(format='ndarray')`
- `MOOP.getObjectiveData(format='ndarray')`

The following methods are not recommended for external usage:

- `MOOP.__extract__(x)`
- `MOOP.__embed__(x)`
- `MOOP.__generate_encoding__()`

- `MOOP.__unpack_sim__(sx)`
- `MOOP.__pack_sim__(sx)`
- `MOOP.fitSurrogates()`
- `MOOP.updateSurrogates()`
- `MOOP.resetSurrogates(center)`
- `MOOP.evaluateSurrogates(x)`
- `MOOP.surrogateUncertainty(x)`
- `MOOP.evaluateObjectives(x)`
- `MOOP.evaluateConstraints(x)`
- `MOOP.evaluatePenalty(x)`
- `MOOP.evaluateGradients(x)`

`__init__`(*opt_func*, *hyperparams=None*)

Initializer for the MOOP class.

Parameters

- **opt_func** (`SurrogateOptimizer`) – A solver for the surrogate problems.
- **hyperparams** (*dict, optional*) – A dictionary of hyperparameters for the opt_func, and any other procedures that will be used.

Returns

A new MOOP object with no design variables, objectives, or constraints.

Return type

`MOOP`

`__extract__`(*x*)

Extract a design variable from an n-dimensional vector.

Parameters

x (*numpy.ndarray*) – A 1D numpy.ndarray containing the embedded design variable.

Returns

Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the extracted design variable values. Note that when working in unnamed mode, the design variable indices were assigned in the order that they were added to the MOOP using *MOOP.addDesign(*args)*.

Return type

numpy.ndarray or numpy structured array

`__embed__`(*x*)

Embed a design input as n-dimensional vector for ParMOO.

Parameters

x (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when working with named design variables) or a 1D numpy.ndarray containing design variable values. Note that when working in unnamed mode, the design variable indices were assigned in the order that they were added to the MOOP using *MOOP.addDesign(*args)*.

Returns

A 1D array containing the embedded design vector.

Return type
numpy.ndarray

`__generate_encoding__()`

Generate the encoding matrices for this MOOP.

`__unpack_sim__(sx)`

Extract a simulation output from a m-dimensional vector.

Parameters
sx (*numpy.ndarray*) – A 1D numpy.ndarray containing the vectorized simulation output(s).

Returns
Either a numpy structured array (when operating with named variables) or the unmodified input sx.

Return type
numpy.ndarray or numpy structured array

`__pack_sim__(sx)`

Pack a simulation output into a m-dimensional vector.

Parameters
sx (*numpy.ndarray or numpy structured array*) – A numpy structured array (when operating with named variables) or a m-dimensional vector.

Returns
A 1D numpy.ndarray of length m.

Return type
numpy.ndarray

`addDesign(*args)`

Add a new design variables to the MOOP.

Append new design variables to the problem. Note that every design variable must be added before any simulations or acquisition functions can be added since the number of design variables is used to infer the size of the simulation databases and the acquisition function initialization.

Parameters
args (*dict*) – Each argument is a dictionary representing one design variable. The dictionary contains information about that design variable, including:

- 'name' (str, optional): The name of this design if any are left blank, then ALL names are considered unspecified.
- 'des_type' (str): The type for this design variable. Currently supported options are:
  - 'continuous' (or 'cont' or 'real')
  - 'categorical' (or 'cat')
  - 'integer' (or 'int')
  - 'custom'
  - 'raw' – for advanced use only, not recommended
- 'lb' (float): When des_type is 'continuous' or 'integer', this specifies the lower bound for the design variable. This value must be specified, and must be strictly less than 'ub' (below) up to the tolerance (below).

- 'ub' (float): When des_type is 'continuous' or 'integer', this specifies the upper bound for the design variable. This value must be specified, and must be strictly greater than 'lb' (above) up to the tolerance (below).
- 'des_tol' (float): When des_type is 'continuous', this specifies the tolerance, i.e., the minimum spacing along this dimension, before two design values are considered to have equal values in this dimension. If not specified, the default value is 1.0e-8 * (ub - lb).
- 'levels' (int or list): When des_type is 'categorical', this specifies the number of levels for the variable (when int) or the names of each valid category (when a list).
- 'embedding_size' (int): When des_type is 'custom', this specifies the dimension of the custom embedding.
- 'dtype' (str): When des_type is 'custom', this contains a string specifying the numpy dtype of the custom input. Only used when operating with named variables, otherwise it must be numeric. When using named variables, defaults to 'U25'.
- 'embedder': When des_type is 'custom', this is a custom embedding function, which maps the input to a point in the unit hypercube of dimension 'embedding_size'.
- 'extracter': When des_type is 'custom', this is a custom extracting function, which maps a point in the unit hypercube of dimension 'embedding_size' to a legal input value of type 'dtype'.

`addSimulation`(**args*)

Add new simulations to the MOOP.

Append new simulation functions to the problem.

Parameters

args (*dict*) – Each argument is a dictionary representing one simulation function. The dictionary must contain information about that simulation function, including:

- name (str, optional): The name of this simulation (defaults to "sim" + str(i), where i = 1, 2, 3, … for the first, second, third, … simulation added to the MOOP).
- m (int): The number of outputs for this simulation.
- sim_func (function): An implementation of the simulation function, mapping from R^n -> R^m. The interface should match: *sim_out = sim_func(x)*.
- search (GlobalSearch): A GlobalSearch object for performing the initial search over this simulation's design space.
- surrogate (SurrogateFunction): A SurrogateFunction object specifying how this simulation's outputs will be modeled.
- hyperparams (dict): A dictionary of hyperparameters, which will be passed to the surrogate and search routines. Most notably, the 'search_budget': (int) can be specified here.

`addObjective`(**args*)

Add a new objective to the MOOP.

Append a new objective to the problem. The objective must be an algebraic function of the design variables and simulation outputs. Note that all objectives must be specified before any acquisition functions can be added.

Parameters

***args** (*dict*) – Python dictionary containing objective function information, including:

- 'name' (str, optional): The name of this objective (defaults to "obj" + str(i), where i = 1, 2, 3, … for the first, second, third, … simulation added to the MOOP).

- 'obj_func' (function): An algebraic objective function that maps from R^n X R^m –> R. Interface should match: *cost = obj_func(x, sim_func(x), der=0)*, where *der* is an optional argument specifying whether to take the derivative of the objective function
  - 0 – no derivative taken, return f(x, sim_func(x))
  - 1 – return derivative wrt x, or
  - 2 – return derivative wrt sim(x).
- 'exp_func' (function): An algebraic objective function that calculates the expected value and distribution of an objective, given that surrogate outputs are Gaussian distributed with given mean and variance. Accepts three inputs. The first input represents x, the second input is the expected value of S(x), and the third input represents the standard deviation of S(x) – assuming S(x) is Gaussian distributed. The output is the expected value of f(x, S). Interface should match: `cost = exp_func(x, sim_mean, sim_std_dev, der=0,

  sd=False)`,

  where *der* is an optional argument specifying whether to evaluate the derivative
  - 0 – no derivative taken, return f(x, sim_func(x))
  - 1 – return derivative wrt x,
  - 2 – return derivative wrt expected value of sim(x), or
  - 3 – return derivative wrt std deviation of sim(x).

`addConstraint`(**args*)

Add a new constraint to the MOOP.

Append a new constraint to the problem. The constraint can be a linear or nonlinear inequality constraint, and may depend on the design variables and/or the simulation outputs.

Parameters

args (*dict*) – Python dictionary containing constraint function information, including:

- 'name' (str, optional): The name of this constraint (defaults to "const" + str(i), where i = 1, 2, 3, … for the first, second, third, … constraint added to the MOOP).
- 'constraint' (function): An algebraic constraint function that maps from R^n X R^m –> R and evaluates to zero or a negative number when feasible and positive otherwise. Interface should match: *violation = constraint(x, sim_func(x), der=0)*, where *der* is an optional argument specifying whether to take the derivative of the constraint function
  - 0 – no derivative taken, return c(x, sim_func(x))
  - 1 – return derivative wrt x, or
  - 2 – return derivative wrt sim(x).

  Note that any `constraint(x, sim_func(x), der=0) <= 0` indicates that x is feaseible, while `constraint(x, sim_func(x), der=0) > 0` indicates that x is infeasible, violating the constraint by an amount proportional to the output. It is the user's responsibility to ensure that after adding all constraints, the feasible region is nonempty and has nonzero measure in the design space.
- 'exp_func' (function): An algebraic objective function that calculates the expected value and distribution of the constraint penalty, given that surrogate outputs are Gaussian distributed with given mean and variance. Accepts three inputs. The first input represents x, the second input is the expected value of S(x), and the third input represents the standard deviation of S(x) – assuming S(x) is Gaussian distributed. The output is the expected value of c(x, S). Interface should match: `penalty = exp_func(x, sim_mean, sim_std_dev, der=0,

sd=False)`,

where *der* is an optional argument specifying whether to evaluate the derivative

- 0 – no derivative taken, return exp val c(x, Sx)
- 1 – return derivative wrt x,
- 2 – return derivative wrt expected value of sim(x), or
- 3 – return derivative wrt std deviation of sim(x).

`addAcquisition`(**args*)

Add an acquisition function to the MOOP.

Append a new acquisition function to the problem. In each iteration, each acquisition is used to generate one or more points to evaluate Typically, each acquisition generates one evaluation per simulation function.

Parameters

args (*dict*) – Python dictionary of acquisition function info, including:

- 'acquisition' (AcquisitionFunction): An acquisition function that maps from R^o –> R for scalarizing outputs.
- 'hyperparams' (dict): A dictionary of hyperparameters for the acquisition functions. Can be omitted if no hyperparameters are needed.

`setCheckpoint`(*checkpoint*, *checkpoint_data=True*, *filename='parmoo'*)

Set ParMOO's checkpointing feature.

Note that for checkpointing to work, all simulation, objective, and constraint functions must be defined in the global scope. ParMOO also cannot save lambda functions.

Parameters

- **checkpoint** (*bool*) – Turn checkpointing on (True) or off (False).
- **checkpoint_data** (*bool, optional*) – Also save raw simulation output in a separate JSON file (True) or rely on ParMOO's internal simulation database (False). When omitted, this parameter defaults to False.
- **filename** (*str, optional*) – Set the base checkpoint filename/path. The checkpoint file will have the JSON format and the extension ".moop" appended to the end of filename. Additional checkpoint files may be created with the same filename but different extensions, depending on the choice of AcquisitionFunction, SurrogateFunction, and GlobalSearch. When omitted, this parameter defaults to "parmoo" and is saved inside current working directory.

`getDesignType`()

Get the numpy dtype of all design points for this MOOP.

Use this type when allocating a numpy array to store the design points for this MOOP object.

Returns

The numpy dtype of this MOOP's design points. If no design variables have yet been added, returns None.

Return type

np.dtype

`getSimulationType`()

Get the numpy dtypes of the simulation outputs for this MOOP.

Use this type if allocating a numpy array to store the simulation outputs of this MOOP object.

Returns
The numpy dtype of this MOOP's simulation outputs. If no simulations have been given, returns None.

Return type
np.dtype

`getObjectiveType()`

Get the numpy dtype of an objective point for this MOOP.

Use this type if allocating a numpy array to store the objective values of this MOOP object.

Returns
The numpy dtype of this MOOP's objective points. If no objectives have yet been added, returns None.

Return type
np.dtype

`getConstraintType()`

Get the numpy dtype of the constraint violations for this MOOP.

Use this type if allocating a numpy array to store the constraint scores output of this MOOP object.

Returns
The numpy dtype of this MOOP's constraint violation outputs. If no constraint functions have been given, returns None.

Return type
np.dtype

`check_sim_db`(*x*, *s_name*)

Check self.sim_db[s_name] to see if the design x was evaluated.

x (np.ndarray or numpy structured array): A 1d numpy.ndarray or numpy
structured array specifying the design point to check for.

s_name (str or int): The name or index of the simulation where
(x, sx) will be added. Note, indices are assigned in the order the simulations were listed during initialization.

Returns
returns None if x is not in self.sim_db[s_name] (up to the design tolerance). Otherwise, returns the corresponding value of sx.

Return type
None or numpy.ndarray

`update_sim_db`(*x*, *sx*, *s_name*)

Update sim_db[s_name] by adding a design/simulation output pair.

x (np.ndarray or numpy structured array): A 1d numpy.ndarray or numpy
structured array specifying the design point to add.

sx (np.ndarray): A 1d numpy.ndarray containing the corresponding
simulation output.

s_name (str or int): The name or index of the simulation to whose
database the pair (x, sx) will be added. Note, when using unnamed variables and simulations, the simulation indices were assigned in the same order that the simulations were added to the MOOP (using *MOOP.addSimulation(*args)*) during initialization.

`evaluateSimulation`(*x*, *s_name*)

Evaluate sim_func[s_name] and store the result in the database.

Parameters

- **x** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the values of the design variable to evaluate. Note, when operating with unnamed variables, design variables are indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- **s_name** (*str, int*) – The name or index of the simulation to evaluate. Note, when operating with unnamed variables, simulation indices were assigned in the order that the simulations were added to the MOOP using *MOOP.addSimulation(*args)*.

Returns
A 1d numpy.ndarray containing the output from the sx = simulation[s_name](x).

Return type
numpy.ndarray

`fitSurrogates`()

Fit the surrogate models using the current sim databases.

Warning: Not recommended for external usage!

`updateSurrogates`()

Update the surrogate models using the current sim databases.

Warning: Not recommended for external usage!

`resetSurrogates`(*center*)

Reset the surrogates using SurrogateFunction.setCenter(center).

Warning: Not recommended for external usage!

Parameters
center (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) coordinates of the new center in the rescaled design space.

Returns
The minimum over the recommended trust region radius for all surrogates.

Return type
float

`evaluateSurrogates`(*x*)

Evaluate all simulation surrogates.

Warning: Not recommended for external usage!

Parameters
x (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.

Returns
A 1d numpy.ndarray containing the (embedded) result of the surrogate model evaluations.

Return type
numpy.ndarray

`surrogateUncertainty`(*x*, *grad=False*)

Evaluate uncertainty (standard deviation) of all surrogates.

Assumes a Gaussian distribution on simulation outputs.

Warning: Not recommended for external usage!

Parameters

- **x** (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.
- **grad** (*bool*) – Specifies whether or not to evalaute the gradients.

Returns

(When grad is False) a 1d numpy.ndarray containing the std deviation of the surrogates at x. (When grad is True) a 2d numpy.ndarray containing the Jacobian of gradients of all surrogates uncertainties at x.

Return type

numpy.ndarray

`evaluateObjectives`(*x*)

Evaluate all objectives using the simulation surrogates as needed.

Warning: Not recommended for external usage!

Parameters

x (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.

Returns

A 1d numpy.ndarray containing the result of the evaluation.

Return type

numpy.ndarray

`evaluateConstraints`(*x*)

Evaluate the constraints using the simulation surrogates as needed.

Warning: Not recommended for external usage!

Parameters

x (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.

Returns

A 1d numpy.ndarray containing the list of constraint violations at x (zero if no violation).

Return type

numpy.ndarray

`evaluatePenalty`(*x*, *sx=None*)

Evaluate the penalized objective using the surrogates as needed.

Warning: Not recommended for external usage!

Parameters

x (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.

Returns

A 1d numpy.ndarray containing the result of the evaluation.

Return type
numpy.ndarray

`evaluateGradients`(*x*)

Evaluate the gradient of the penalized objective using surrogates.

Warning: Not recommended for external usage!

Parameters
x (*numpy.ndarray*) – A 1d numpy.ndarray containing the (embedded) design point to evaluate.

Returns
A 1d numpy.ndarray containing the result of the evaluation.

Return type
numpy.ndarray

`addData`(*x*, *sx*)

Update the internal objective database by truly evaluating x.

Parameters

- **x** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the value of the design variable to add to ParMOO's database. When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- **sx** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the values of the corresponding simulation outputs for ALL simulations involved in this MOOP. In named mode, sx['s_name'][:] contains the output(s) for sim_func['s_name']. In unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*. Then, if each simulation i has m_i outputs (i = 0, 1, ...):
  - sx[:m_0] contains the output(s) of sim_func[0],
  - sx[m_0:m_0 + m_1] contains output(s) of sim_func[1],
  - sx[m_0 + m_1:m_0 + m_1 + m_2] contains the output(s) for sim_func[2], etc.

`iterate`(*k*, *ib=None*)

Perform an iteration of ParMOO's solver and generate candidates.

Generates a batch of suggested candidate points (design points) or (candidate point, simulation name) pairs and returns to the user for further processing. Note, this method may produce duplicates.

Parameters

- **k** (*int*) – The iteration counter (corresponding to MOOP.iteration).
- **ib** (*int, optional*) – The index of the acquisition function to optimize and add to the current batch. Defaults to None, which optimizes all acquisition functions and adds all resulting candidates to the batch.

Returns

A list of design points (numpy structured or 1D arrays) or tuples (design points, simulation name) specifying the unfiltered list of candidates that ParMOO recommends for true simulation evaluations. Specifically:

- Each item or the first entry in tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- If the item is a tuple, then the second entry in the tuple is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Return type
(list)

`filterBatch`(**args*)

Filter a batch produced by ParMOO's MOOP.iterate method.

Accepts one or more batches of candidate design points, produced by the MOOP.iterate() method and checks both the batch and ParMOO's database for redundancies. Any redundant points (up to the design tolerance) are replaced by model improving points, using each surrogate's Surrogate.improve() method.

Parameters

- ***args** (*list of numpy.ndarrays or tuples*) – The list of
- **method.** (*unfiltered candidates returned by the MOOP.iterate()*) –
- **points** (*A list of design*) –
- **tuples** (*design points, simulation name*) –
- **simulation** (*list of candidates that ParMOO recommends for true*) –
- **Specifically** (*evaluations.*) –
  - Each item or the first entry in tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
  - If the item is a tuple, then the second entry in the tuple is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Returns

A filtered list of ordered pairs (tuples), specifying the (design points, simulation name) that ParMOO suggests for evaluation. Specifically:

- The first entry in each tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- The second entry is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Return type
(list)

updateAll(*k*, *batch*)

Update all surrogates given a batch of freshly evaluated data.

Parameters

- **k** (*int*) – The iteration counter (corresponding to MOOP.iteration).
- **batch** (*list*) – A list of ordered pairs (tuples), each specifying a design point that was evaluated in this iteration. For each tuple in the list:
  - The first entry in each tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
  - The second entry is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

solve(*iter_max=None*, *sim_max=None*)

Solve a MOOP using ParMOO.

If desired, be sure to turn on checkpointing before starting the solve, using:

`MOOP.setCheckpoint(checkpoint, [checkpoint_data, filename])`

and turn on INFO-level logging for verbose output, using:

`` import logging logging.basicConfig(level=logging.INFO,

[format='%(asctime)s %(levelname)-8s %(message)s',
datefmt='%Y-%m-%d %H:%M:%S'])

``

Parameters
iter_max (*int*) – The max limit for ParMOO's internal iteration counter. ParMOO keeps track of how many iterations it has completed internally. This value k specifies the stopping criteria for ParMOO.

getPF(*format='ndarray'*)

Extract nondominated and efficient sets from internal databases.

Parameters
format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

A discrete approximation of the Pareto front and efficient set.

If operating with named variables, then this is a 1d numpy structured array whose fields match the names for design variables, objectives, and constraints (if any).

Otherwise, this is a dict containing the following keys:

- x_vals (numpy.ndarray): A 2d numpy.ndarray containing a list of nondominated points discretely approximating the Pareto front.
- f_vals (numpy.ndarray): A 2d numpy.ndarray containing the list of corresponding efficient design points.

- c_vals (numpy.ndarray): A 2d numpy.ndarray containing the list of corresponding constraint satisfaction scores, all less than or equal to 0.

`getSimulationData`(*format='ndarray'*)

Extract all computed simulation outputs from the MOOP's database.

Parameters

format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

(dict or list) Either a dictionary or list of dictionaries containing every point where a simulation was evaluated.

If operating with named variables, then the result is a dict. Each key is the name for a different simulation, and each value is a 1d numpy structured array whose keys match the names for each design variables plus an additional 'out' key for simulation outputs.

Otherwise, this is a list of s (number of simulations) dicts, each dict containing the following keys:

- x_vals (numpy.ndarray): A 2d array containing a list of design points that have been evaluated for this simulation.
- s_vals (numpy.ndarray): A 1d or 2d array containing the list of corresponding simulation outputs.

`getObjectiveData`(*format='ndarray'*)

Extract all computed objective scores from this MOOP's database.

Parameters

format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

A database of all designs that have been fully evaluated, and their corresponding objective scores.

If operating with named variables, then this is a 1d numpy structured array whose fields match the names for design variables, objectives, and constraints (if any).

Otherwise, this is a dict containing the following keys:

- x_vals (numpy.ndarray): A 2d array containing a list of all fully evaluated design points.
- f_vals (numpy.ndarray): A 2d array containing the list of corresponding objective values.
- c_vals (numpy.ndarray): A 2d array containing the list of corresponding constraint satisfaction scores, all less than or equal to 0.

`save`(*filename='parmoo'*)

Serialize and save the MOOP object and all of its dependencies.

Parameters

filename (*str, optional*) – The filepath to serialized checkpointing file(s). Do not include file extensions, they will be appended automaically. This method may create several additional save files with this same name, but different file extensions, in order to recursively save dependency objects (such as surrogate models). Defaults to the value "parmoo" (filename will be "parmoo.moop").

load(*filename='parmoo'*)

Load a serialized MOOP object and all of its dependencies.

Parameters

filename (*str, optional*) – The filepath to the serialized checkpointing file(s). Do not include file extensions, they will be appended automaically. This method may also load from other save files with the same name, but different file extensions, in order to recursively load dependency objects (such as surrogate models) as needed. Defaults to the value "parmoo" (filename will be "parmoo.moop").

savedata(*x*, *sx*, *s_name*, *filename='parmoo'*)

Save the current simulation database for this MOOP.

Parameters

filename (*str, optional*) – The filepath to the checkpointing file(s). Do not include file extensions, they will be appended automaically. Defaults to the value "parmoo" (filename will be "parmoo.simdb.json").

Parallel Solvers using the libE_MOOP Class

Solve MOOPs in parallel using ParMOO together with libEnsemble.

```
from parmoo.extras import libE_MOOP
```

The `libE_MOOP.solve(...)` method will distribute simulations across parallel resources using libEnsemble for this class.

Contains the libE_MOOP class and parmoo_persis_gen function.

Use the libE_MOOP class to define and solve multiobjective optimization problems (MOOPs) with parallel simulation evaluations. The libE_MOOP class extends the base class parmoo.moop.MOOP for defining and solving MOOPs.

The parmoo_persis_gen function can be used as a generator function in libEnsemble. To do so, create a regular *parmoo.MOOP* object and add it to the *gen_specs* dict, then import and use *parmoo_persis_gen* as the libE gen func.

class extras.libe.**libE_MOOP**(*opt_func*, *hyperparams=None*)

Class for solving a MOOP using libEnsemble to manage parallelism.

Upon initialization, supply a scalar optimization procedure and dictionary of hyperparameters using the default constructor:

- `moop = libE_MOOP.__init__(ScalarOpt, [hyperparams={}])`

Class methods are summarized below.

To define the MOOP, add each design variable, simulation, objective, and constraint (in that order) by using the following functions:

- `libE_MOOP.addDesign(*args)`
- `libE_MOOP.addSimulation(*args)`
- `libE_MOOP.addObjective(*args)`
- `libE_MOOP.addConstraint(*args)`

Next, define your solver.

Acquisition functions (used for scalarizing problems/setting targets) are added using:

- `libE_MOOP.addAcquisition(*args)`

After creating a MOOP, the following methods may be useful for getting the numpy.dtype of the input/output arrays:

- `libE_MOOP.getDesignType()`
- `libE_MOOP.getSimulationType()`
- `libE_MOOP.getObjectiveType()`
- `libE_MOOP.getConstraintType()`

The following methods are used to save/load ParMOO objects from memory:

- `libE_MOOP.save([filename="parmoo"])`
- `libE_MOOP.load([filename="parmoo"])`

To turn on checkpointing (recommended), use:

- `libE_MOOP.setCheckpoint(checkpoint, [checkpoint_data, filename])`

ParMOO's logging feature is not active for the *libE_MOOP* class since libEnsemble already provides this feature.

After defining the MOOP and setting up checkpointing and logging, use the following method to solve the MOOP (using libEnsemble to distribute simulation evaluations):

- ``**libE_MOOP.solve(iter_max=None, sim_max=None, wt_max=864000,**
  profile=False)``

The following methods are used for managing ParMOO's internal simulation/objective databases. Note that these databases are maintained separately from libEnsemble's simulation database:

- `libE_MOOP.check_sim_db(x, s_name)`
- `libE_MOOP.update_sim_db(x, sx, s_name)`
- `libE_MOOP.evaluateSimulation(x, s_name)`
- `libE_MOOP.addData(x, sx)`
- `libE_MOOP.iterate(k, ib)`
- `libE_MOOP.updateAll(k, batch)`

Finally, the following methods are used to retrieve data after the problem has been solved:

- `libE_MOOP.getPF(format='ndarray')`
- `libE_MOOP.getSimulationData(format='ndarray')`
- `libE_MOOP.getObjectiveData(format='ndarray')`

Other private methods from the MOOP class do not work for a libE_MOOP.

`__init__`(*opt_func*, *hyperparams=None*)

Initializer for the libE interface to the MOOP class.

Parameters

- **opt_func** (*`SurrogateOptimizer`*) – A solver for the surrogate problems.
- **hyperparams** (*`dict, optional`*) – A dictionary of hyperparameters for the opt_func, and any other procedures that will be used.

Returns

A new libE_MOOP object with no design variables,
objectives, or constraints.

Return type
`libE_MOOP`

addDesign(**args*)

Add a new design variables to the libE_MOOP.

Append new design variables to the problem. Note that every design variable must be added before any simulations or acquisition functions can be added since the number of design variables is used to infer the size of simulation databases and acquisition function policies.

Parameters
args (*dict*) – Each argument is a dictionary representing one design variable. The dictionary contains information about that design variable, including:

- 'name' (str, optional): The name of this design if any are left blank, then ALL names are considered unspecified.
- 'des_type' (str): The type for this design variable. Currently supported options are:
  - 'continuous'
  - 'categorical'
- 'lb' (float): When des_type is 'continuous', this specifies the lower bound for the design variable. This value must be specified, and must be strictly less than 'ub' (below) up to the tolerance (below).
- 'ub' (float): When des_type is 'continuous', this specifies the upper bound for the design variable. This value must be specified, and must be strictly greater than 'lb' (above) up to the tolerance (below).
- 'tol' (float): When des_type is 'continuous', this specifies the tolerance, i.e., the minimum spacing along this dimension, before two design values are considered to have equal values in this dimension. If not specified, the default value is 1.0e-8.
- 'levels' (int): When des_type is 'categorical', this specifies the number of levels for the variable.

addSimulation(**args*)

Add new simulations to the libE_MOOP.

Append new simulation functions to the problem.

Parameters
args (*dict*) – Each argument is a dictionary representing one simulation function. The dictionary must contain information about that simulation function, including:

- name (str, optional): The name of this simulation (defaults to "sim" + str(i), where i = 1, 2, 3, … for the first, second, third, … simulation added to the MOOP).
- m (int): The number of outputs for this simulation.
- sim_func (function): An implementation of the simulation function, mapping from R^n -> R^m. The interface should match: *sim_out = sim_func(x)*.
- search (GlobalSearch): A GlobalSearch object for performing the initial search over this simulation's design space.
- surrogate (SurrogateFunction): A SurrogateFunction object specifying how this simulation's outputs will be modeled.
- hyperparams (dict): A dictionary of hyperparameters, which will be passed to the surrogate and search routines. Most notably, 'search_budget': (int) can be specified here.

`addObjective`(**args*)

Add a new objective to the libE_MOOP.

Append a new objective to the problem. The objective must be an algebraic function of the design variables and simulation outputs. Note that all objectives must be specified before any acquisition functions can be added.

Parameters

args (*dict*) – Python dictionary containing objective function information, including:

- 'name' (str, optional): The name of this objective (defaults to "obj" + str(i), where i = 1, 2, 3, … for the first, second, third, … simulation added to the MOOP).
- 'obj_func' (function): An algebraic objective function that maps from R^n X R^m –> R. Interface should match: *cost = obj_func(x, sim_func(x), der=0)*, where *der* is an optional argument specifying whether to take the derivative of the objective function
  - 0 – no derivative taken, return f(x, sim_func(x))
  - 1 – return derivative wrt x, or
  - 2 – return derivative wrt sim(x).

`addConstraint`(**args*)

Add a new constraint to the libE_MOOP.

Append a new constraint to the problem. The constraint can be a linear or nonlinear inequality constraint, and may depend on the design variables and/or the simulation outputs.

Parameters

***args** (*dict*) – Python dictionary containing constraint function information, including:

- 'name' (str, optional): The name of this constraint (defaults to "const" + str(i), where i = 1, 2, 3, … for the first, second, third, … constraint added to the MOOP).
- 'constraint' (function): An algebraic constraint function that maps from R^n X R^m –> R and evaluates to zero or a negative number when feasible and positive otherwise. Interface should match: *violation = constraint(x, sim_func(x), der=0)*, where *der* is an optional argument specifying whether to take the derivative of the constraint function
  - 0 – no derivative taken, return c(x, sim_func(x))
  - 1 – return derivative wrt x, or
  - 2 – return derivative wrt sim(x).

  Note that any `constraint(x, sim_func(x), der=0) <= 0` indicates that x is feaseible, while `constraint(x, sim_func(x), der=0) > 0` indicates that x is infeasible, violating the constraint by an amount proportional to the output. It is the user's responsibility to ensure that after adding all constraints, the feasible region is nonempty and has nonzero measure in the design space.

`addAcquisition`(**args*)

Add an acquisition function to the libE_MOOP.

Append a new acquisition function to the problem. In each iteration, each acquisition is used to generate one or more points to evaluate Typically, each acquisition generates one evaluation per simulation function.

Parameters

args (*dict*) – Python dictionary of acquisition function info, including:

- 'acquisition' (AcquisitionFunction): An acquisition function that maps from R^o –> R for scalarizing outputs.

- 'hyperparams' (dict): A dictionary of hyperparameters for the acquisition functions. Can be omitted if no hyperparameters are needed.

`setCheckpoint`(*checkpoint*, *checkpoint_data=True*, *filename='parmoo'*)

Set ParMOO's checkpointing feature.

Note that for checkpointing to work, all simulation, objective, and constraint functions must be defined in the global scope. ParMOO also cannot save lambda functions.

Parameters

- **checkpoint** (*bool*) – Turn checkpointing on (True) or off (False).
- **checkpoint_data** (*bool, optional*) – Also save raw simulation output in a separate JSON file (True) or rely on ParMOO's internal simulation database (False). When omitted, this parameter defaults to False.
- **filename** (*str, optional*) – Set the base checkpoint filename/path. The checkpoint file will have the JSON format and the extension ".moop" appended to the end of filename. Additional checkpoint files may be created with the same filename but different extensions, depending on the choice of AcquisitionFunction, SurrogateFunction, and GlobalSearch. When omitted, this parameter defaults to "parmoo" and is saved inside current working directory.

`getDesignType`()

Get the numpy dtype of all design points for this MOOP.

Use this type when allocating a numpy array to store the design points for this MOOP object.

Returns
The numpy dtype of this MOOP's design points. If no design variables have yet been added, returns None.

Return type
np.dtype

`getSimulationType`()

Get the numpy dtypes of the simulation outputs for this MOOP.

Use this type if allocating a numpy array to store the simulation outputs of this MOOP object.

Returns
The numpy dtype of this MOOP's simulation outputs. If no simulations have been given, returns None.

Return type
np.dtype

`getObjectiveType`()

Get the numpy dtype of an objective point for this MOOP.

Use this type if allocating a numpy array to store the objective values of this MOOP object.

Returns
The numpy dtype of this MOOP's objective points. If no objectives have yet been added, returns None.

Return type
np.dtype

`getConstraintType()`

Get the numpy dtype of the constraint violations for this MOOP.

Use this type if allocating a numpy array to store the constraint scores output of this MOOP object.

Returns
The numpy dtype of this MOOP's constraint violation output. If no constraints have been given, returns None.

Return type
np.dtype

`check_sim_db`(*x*, *s_name*)

Check self.sim_db[s_name] to see if the design x was evaluated.

x (np.ndarray or numpy structured array): A 1d numpy.ndarray or numpy
structured array specifying the design point to check for.

s_name (str or int): The name or index of the simulation where
(x, sx) will be added. Note, indices are assigned in the order the simulations were listed during initialization.

Returns
returns None if x is not in self.sim_db[s_name] (up to the design tolerance). Otherwise, returns the corresponding value of sx.

Return type
None or numpy.ndarray

`update_sim_db`(*x*, *sx*, *s_name*)

Update sim_db[s_name] by adding a design/simulation output pair.

x (np.ndarray or numpy structured array): A 1d numpy.ndarray or numpy
structured array specifying the design point to add.

sx (np.ndarray): A 1d numpy.ndarray containing the corresponding
simulation output.

s_name (str or int): The name or index of the simulation to whose
database the pair (x, sx) will be added. Note, when using unnamed variables and simulations, the simulation indices were assigned in the same order that the simulations were added to the MOOP (using *MOOP.addSimulation(*args)*) during initialization.

`evaluateSimulation`(*x*, *s_name*)

Evaluate sim_func[s_name] and store the result in the database.

Parameters

- **x** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the values of the design variable to evaluate. Note, when operating with unnamed variables, design variables are indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- **s_name** (*str, int*) – The name or index of the simulation to evaluate. Note, when operating with unnamed variables, simulation indices were assigned in the order that the simulations were added to the MOOP using *MOOP.addSimulation(*args)*.

Returns
A 1d numpy.ndarray containing the output from the sx = simulation[s_name](x).

Return type
numpy.ndarray

`addData`(*x*, *sx*)

Update the internal objective database by truly evaluating x.

Parameters

- **x** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the value of the design variable to add to ParMOO's database. When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- **sx** (*numpy.ndarray or numpy structured array*) – Either a numpy structured array (when using named variables) or a 1D numpy.ndarray containing the values of the corresponding simulation outputs for ALL simulations involved in this MOOP. In named mode, sx['s_name'][:] contains the output(s) for sim_func['s_name']. In unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*. Then, if each simulation i has m_i outputs (i = 0, 1, . . .):
  - sx[:m_0] contains the output(s) of sim_func[0],
  - sx[m_0:m_0 + m_1] contains output(s) of sim_func[1],
  - sx[m_0 + m_1:m_0 + m_1 + m_2] contains the output(s) for sim_func[2], etc.

`iterate`(*k*, *ib=None*)

Perform an iteration of ParMOO's solver and generate candidates.

Generates a batch of suggested candidate points (design points) or (candidate point, simulation name) pairs and returns to the user for further processing. Note, this method may produce duplicates.

Parameters

- **k** (*int*) – The iteration counter (corresponding to MOOP.iteration).
- **ib** (*int, optional*) – The index of the acquisition function to optimize and add to the current batch. Defaults to None, which optimizes all acquisition functions and adds all resulting candidates to the batch.

Returns

A list of design points (numpy structured or 1D arrays) or tuples (design points, simulation name) specifying the unfiltered list of candidates that ParMOO recommends for true simulation evaluations. Specifically:

- Each item or the first entry in tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- If the item is a tuple, then the second entry in the tuple is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Return type
(list)

`filterBatch`(**xbatch*)

Filter a batch produced by ParMOO's MOOP.iterate method.

Accepts one or more batches of candidate design points, produced by the MOOP.iterate() method and checks both the batch and ParMOO's database for redundancies. Any redundant points (up to the design tolerance) are replaced by model improving points, using each surrogate's Surrogate.improve() method.

Parameters

- ***xbatch** (*list of numpy.ndarrays or tuples*) – The list of
- **method.** (*unfiltered candidates returned by the MOOP.iterate()*) –
- **points** (*A list of design*) –
- **tuples** (*design points, simulation name*) –
- **simulation** (*list of candidates that ParMOO recommends for true*) –
- **Specifically** (*evaluations.*) –
  - Each item or the first entry in tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
  - If the item is a tuple, then the second entry in the tuple is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Returns

A filtered list of ordered pairs (tuples), specifying the (design points, simulation name) that ParMOO suggests for evaluation. Specifically:

- The first entry in each tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.
- The second entry is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

Return type

(list)

`updateAll`(*k*, *batch*)

Update all surrogates given a batch of freshly evaluated data.

Parameters

- **k** (*int*) – The iteration counter (corresponding to the value in libE_MOOP.moop.iteration).
- **batch** (*list*) – A list of ordered pairs (tuples), each specifying a design point that was evaluated in this iteration. For each tuple in the list:
  - The first entry in each tuple is either a numpy structured array (when operating with named variables) or a 1D numpy.ndarray (in unnamed mode). When operating with unnamed variables, the indices were assigned in the order that the design variables were added to the MOOP using *MOOP.addDesign(*args)*.

- The second entry is either the (str) name of the simulation to evaluate (when operating with named variables) or the (int) index of the simulation to evaluate (when operating in unnamed mode). Note, in unnamed mode, simulation indices were assigned in the order that they were added using *MOOP.addSimulation(*args)*.

`moop_sim`(*H*, *persis_info*, *sim_specs*, *_*)

Evaluates the sim function for a collection of points given in `H['x']`.

`solve`(*iter_max=None*, *sim_max=None*, *wt_max=864000*, *profile=False*)

Solve a MOOP using ParMOO + libEnsemble.

If desired, be sure to turn on checkpointing before starting the solve, using:

`MOOP.setCheckpoint(checkpoint, [checkpoint_data, filename])`

ParMOO will solve the MOOP and use libEnsemble to distribute simulations over available resources.

Parameters

- **iter_max** (*int*) – The max number of ParMOO iterations to be performed by libEnsemble (default is unlimited).
- **sim_max** (*int*) – The max number of simulation to be performed by libEnsemble (default is unlimited).
- **wt_max** (*int*) – The max number of seconds that the simulation may run for (the default is 864000 secs, i.e., 10 days).
- **profile** (*bool*) – Specifies whether to run libE with the profiler.

`getPF`(*format='ndarray'*)

Extract nondominated and efficient sets from internal databases.

Parameters

format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

A discrete approximation of the Pareto front and efficient set.

If operating with named variables, then this is a 1d numpy structured array whose fields match the names for design variables, objectives, and constraints (if any).

Otherwise, this is a dict containing the following keys:

- x_vals (numpy.ndarray): A 2d numpy.ndarray containing a list of nondominated points discretely approximating the Pareto front.
- f_vals (numpy.ndarray): A 2d numpy.ndarray containing the list of corresponding efficient design points.
- c_vals (numpy.ndarray): A 2d numpy.ndarray containing the list of corresponding constraint satisfaction scores, all less than or equal to 0.

`getSimulationData`(*format='ndarray'*)

Extract all computed simulation outputs from the MOOP's database.

Parameters

format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

(dict or list) Either a dictionary or list of dictionaries containing every point where a simulation was evaluated.

If operating with named variables, then the result is a dict. Each key is the name for a different simulation, and each value is a 1d numpy structured array whose keys match the names for each design variables plus an additional 'out' key for simulation outputs.

Otherwise, this is a list of s (number of simulations) dicts, each dict containing the following keys:

- x_vals (numpy.ndarray): A 2d array containing a list of design points that have been evaluated for this simulation.
- s_vals (numpy.ndarray): A 1d or 2d array containing the list of corresponding simulation outputs.

`getObjectiveData`(*format='ndarray'*)

Extract all computed objective scores from this MOOP's database.

Parameters

format (*str, optional*) – Either 'ndarray' (default) or 'pandas', in order to produce output as a numpy structured array or pandas dataframe. Note: format='pandas' is only valid for named inputs.

Returns

A database of all designs that have been fully evaluated, and their corresponding objective scores.

If operating with named variables, then this is a 1d numpy structured array whose fields match the names for design variables, objectives, and constraints (if any).

Otherwise, this is a dict containing the following keys:

- x_vals (numpy.ndarray): A 2d array containing a list of all fully evaluated design points.
- f_vals (numpy.ndarray): A 2d array containing the list of corresponding objective values.
- c_vals (numpy.ndarray): A 2d array containing the list of corresponding constraint satisfaction scores, all less than or equal to 0.

`save`(*filename='parmoo'*)

Serialize and save the MOOP object and all of its dependencies.

Parameters

filename (*str, optional*) – The filepath to serialized checkpointing file(s). Do not include file extensions, they will be appended automaically. May create several save files with extensions of this name, in order to recursively save dependencies objects. Defaults to the value "parmoo" (filename will be "parmoo.moop").

`load`(*filename='parmoo'*)

Load a serialized MOOP object and all of its dependencies.

Parameters

filename (*str, optional*) – The filepath to the serialized checkpointing file(s). Do not include file extensions, they will be appended automaically. This method may also load from other save files with the same name, but different file extensions, in order to recursively load dependency objects (such as surrogate models) as needed. Defaults to the value "parmoo" (filename will be "parmoo.moop").

savedata(*x*, *sx*, *s_name*, *filename='parmoo'*)

Save the current simulation database for this MOOP.

Parameters

filename (*str, optional*) – The filepath to the checkpointing file(s). Do not include file extensions, they will be appended automaically. Defaults to the value "parmoo" (filename will be "parmoo.simdb.json").

2.1.2 Acquisition Functions

Add one of these to your `MOOP` object to generate additional scalarizations per iteration. In general, ParMOO typically generates one candidate solution per simulation per acquisition function, so the number of acquisition functions determines the number of candidate simulations evaluated (in parallel) per iteration/batch.

```
from parmoo import acquisitions
```

Current options are:

Weighted Sum Methods

Implementations of the weighted-sum scalarization technique.

This module contains implementations of the AcquisitionFunction ABC, which use the weighted-sum technique.

The classes include:

- `UniformWeights` (sample convex weights from a uniform distribution)
- `FixedWeights` (uses a fixed scalarization, which can be set upon init)

class acquisitions.weighted_sum.**UniformWeights**(*o*, *lb*, *ub*, *hyperparams*)

Randomly generate scalarizing weights.

Generates uniformly distributed scalarization weights, by randomly sampling the probability simplex.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the UniformWeights class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for tuning the acquisition function.

Returns

A new UniformWeights generator.

Return type

`UniformWeights`

`useSD()`

Query whether this method uses uncertainties.

When False, allows users to shortcut expensive uncertainty computations.

`setTarget`(*data*, *penalty_func*, *history*)

Randomly generate a new vector of scalarizing weights.

Parameters

- `data` (*dict*) – A dictionary specifying the current function evaluation database.
- `penalty_func` (*function*) – A function of one (x) or two (x, sx) inputs that evaluates the (penalized) objectives.
- `history` (*dict*) – Another unused argument for this function.

Returns

A 1d array containing the 'best' feasible starting point for the scalarized problem (if any previous evaluations were feasible) or the point in the existing database that is most nearly feasible.

Return type

numpy.ndarray

`scalarize`(*f_vals*, *x_vals*, *s_vals_mean*, *s_vals_sd*)

Scalarize a vector of function values using the current weights.

Parameters

- `f_vals` (*numpy.ndarray*) – A 1d array specifying the function values to be scalarized.
- `x_vals` (*np.ndarray*) – A 1D array specifying a vector the design point corresponding to f_vals (unused by this method).
- `s_vals_mean` (*np.ndarray*) – A 1D array specifying the expected simulation outputs for the x value being scalarized (unused by this method).
- `s_vals_sd` (*np.ndarray*) – A 1D array specifying the standard deviation for each of the simulation outputs (unused by this method).

Returns

The scalarized value.

Return type

float

`scalarizeGrad`(*f_vals*, *g_vals*)

Scalarize a Jacobian of gradients using the current weights.

Parameters

- `f_vals` (*numpy.ndarray*) – A 1d array specifying the function values for the scalarized gradient (not used here).
- `g_vals` (*numpy.ndarray*) – A 2d array specifying the gradient values to be scalarized.

Returns

The 1d array for the scalarized gradient.

Return type

np.ndarray

class acquisitions.weighted_sum.**FixedWeights**(*o*, *lb*, *ub*, *hyperparams*)

Use fixed scalarizing weights.

Use a fixed scalarization scheme, based on a fixed weighted sum.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the FixedWeights class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for tuning the acquisition function. May contain the following key:
  - 'weights' (numpy.ndarray): A 1d array of length o that, when present, specifies the scalarization weights to use. When absent, the default weights are w = [1/o, . . . , 1/o].

Returns

A new FixedWeights generator.

Return type

FixedWeights

useSD()

Querry whether this method uses uncertainties.

When False, allows users to shortcut expensive uncertainty computations.

setTarget(*data*, *penalty_func*, *history*)

Randomly generate a feasible starting point.

Parameters

- **data** (*dict*) – A dictionary specifying the current function evaluation database.
- **penalty_func** (*function*) – A function of one (x) or two (x, sx) inputs that evaluates the (penalized) objectives.
- **history** (*dict*) – Another unused argument for this function.

Returns

A 1d array containing the 'best' feasible starting point for the scalarized problem (if any previous evaluations were feasible) or the point in the existing database that is most nearly feasible.

Return type

numpy.ndarray

scalarize(*f_vals*, *x_vals*, *s_vals_mean*, *s_vals_sd*)

Scalarize a vector of function values using the current weights.

Parameters

- **f_vals** (*numpy.ndarray*) – A 1d array specifying the function values to be scalarized.
- **x_vals** (*np.ndarray*) – A 1D array specifying a vector the design point corresponding to f_vals (unused by this method).

- **s_vals_mean** (*np.ndarray*) – A 1D array specifying the expected simulation outputs for the x value being scalarized (unused by this method).
- **s_vals_sd** (*np.ndarray*) – A 1D array specifying the standard deviation for each of the simulation outputs (unused by this method).

Returns
The scalarized value.

Return type
float

scalarizeGrad(*f_vals*, *g_vals*)

Scalarize a Jacobian of gradients using the current weights.

Parameters

- **f_vals** (*numpy.ndarray*) – A 1d array specifying the function values for the scalarized gradient (not used here).
- **g_vals** (*numpy.ndarray*) – A 2d array specifying the gradient values to be scalarized.

Returns
The 1d array for the scalarized gradient.

Return type
np.ndarray

Epsilon Constraint Methods

Implementations of the epsilon-constraint-style scalarizations.

This module contains implementations of the AcquisitionFunction ABC, which use the epsilon constraint method.

The classes include:

- `RandomConstraint` (randomly set a ub for all but 1 objective)

class acquisitions.epsilon_constraint.**RandomConstraint**(*o*, *lb*, *ub*, *hyperparams*)

Improve upon a randomly set target point.

Randomly sets a target point inside the current Pareto front. Attempts to improve one of the objective values by reformulating all other objectives as constraints, upper bounded by their target value.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the RandomConstraint class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for tuning the acquisition function.

Returns
A new RandomConstraint scalarizer.

Return type
`RandomConstraint`

`useSD()`

Querry whether this method uses uncertainties.

When False, allows users to shortcut expensive uncertainty computations.

`setTarget`(*data*, *penalty_func*, *history*)

Randomly generate a target based on current nondominated points.

Parameters

- **data** (*dict*) – A dictionary specifying the current function evaluation database. It contains two mandatory fields:
  - 'x_vals' (numpy.ndarray): A 2d array containing the list of design points.
  - 'f_vals' (numpy.ndarray): A 2d array containing the corresponding list of objective values.
- **penalty_func** (*function*) – A function of one (x) or two (x, sx) inputs that evaluates the (penalized) objectives.
- **history** (*dict*) – A persistent dictionary that could be used by the implementation of the AcquisitionFunction to pass data between iterations; also unused by this scheme.

Returns
A 1d array containing the 'best' feasible starting point for the scalarized problem (if any previous evaluations were feasible) or the point in the existing database that is most nearly feasible.

Return type
numpy.ndarray

`scalarize`(*f_vals*, *x_vals*, *s_vals_mean*, *s_vals_sd*)

Scalarize a vector of function values using the current bounds.

Parameters

- **f_vals** (*numpy.ndarray*) – A 1d array specifying the function values to be scalarized.
- **x_vals** (*np.ndarray*) – A 1D array specifying a vector the design point corresponding to f_vals (unused by this method).
- **s_vals_mean** (*np.ndarray*) – A 1D array specifying the expected simulation outputs for the x value being scalarized (unused by this method).
- **s_vals_sd** (*np.ndarray*) – A 1D array specifying the standard deviation for each of the simulation outputs (unused by this method).

Returns
The scalarized value.

Return type
float

`scalarizeGrad`(*f_vals*, *g_vals*)

Scalarize a Jacobian of gradients using the current bounds.

Parameters

- **f_vals** (*numpy.ndarray*) – A 1d array specifying the function values for the scalarized gradient, which are used to penalize exceeding the bounds.

- **g_vals** (*numpy.ndarray*) – A 2d array specifying the gradient values to be scalarized.

Returns
The 1d array for the scalarized gradient.

Return type
np.ndarray

class acquisitions.epsilon_constraint.**EI_RandomConstraint**(*o*, *lb*, *ub*, *hyperparams*)

Expected improvement of a randomly set target point.

Randomly sets a target point inside the current Pareto front. Attempts to improve one of the objective values by reformulating all other objectives as constraints, upper bounded by their target value. Uses surrogate uncertainties to maximize expected improvement in the target objective subject to constraints.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the RandomConstraint class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for tuning the acquisition function. Including
  - mc_sample_size (int): The number of samples to use for monte carlo integration (defaults to 10 * m ** 2).

Returns
A new RandomConstraint scalarizer.

Return type
RandomConstraint

useSD()

Querry whether this method uses uncertainties.

When False, allows users to shortcut expensive uncertainty computations.

setTarget(*data*, *penalty_func*, *history*)

Randomly generate a target based on current nondominated points.

Parameters

- **data** (*dict*) – A dictionary specifying the current function evaluation database. It contains two mandatory fields:
  - 'x_vals' (numpy.ndarray): A 2d array containing the list of design points.
  - 'f_vals' (numpy.ndarray): A 2d array containing the corresponding list of objective values.
- **penalty_func** (*function*) – A function of one (x) or two (x, sx) inputs that evaluates the (penalized) objectives.
- **history** (*dict*) – A persistent dictionary that could be used by the implementation of the AcquisitionFunction to pass data between iterations; also unused by this scheme.

Returns
A 1d array containing the 'best' feasible starting point for the scalarized problem (if any previous evaluations were feasible) or the point in the existing database that is most nearly feasible.

Return type
numpy.ndarray

`scalarize`(*f_vals*, *x_vals*, *s_vals_mean*, *s_vals_sd*)

Scalarize a vector of function values using the current bounds.

Parameters

- **f_vals** (*numpy.ndarray*) – A 1d array specifying the function values to be scalarized.
- **x_vals** (*np.ndarray*) – A 1D array specifying a vector the design point corresponding to f_vals (unused by this method).
- **s_vals_mean** (*np.ndarray*) – A 1D array specifying the expected simulation outputs for the x value being scalarized (unused by this method).
- **s_vals_sd** (*np.ndarray*) – A 1D array specifying the standard deviation for each of the simulation outputs (unused by this method).

Returns
The scalarized value.

Return type
float

`scalarizeGrad`(*f_vals*, *g_vals*)

Not implemented for this acquisition function, do not use gradient-based methods.

2.1.3 Surrogate Optimizers

When initializing a new `MOOP` object (see `MOOP Classes`), you must provide a surrogate optimization problem solver, which will be used to generate candidate solutions for each iteration.

```
from parmoo import optimizers
```

Note that when using a gradient-based technique, you must provide gradient evaluation options for all objective and constraint functions, by adding code to handle the optional ``der`` input.

```
def f(x, sx, der=0):
    # When using gradient-based solvers, define extra if-cases for
    # handling der=1 (calculate df/dx) and der=2 (caldculate df/dsx).
```

GPS Search Techniques (gradient-free)

Implementations of the SurrogateOptimizer class.

This module contains implementations of the SurrogateOptimizer ABC, which are based on the GPS polling strategy for direct search.

Note that these strategies are all gradient-free, and therefore does not require objective, constraint, or surrogate gradients methods to be defined.

The classes include:

- `LocalGPS` – Generalized Pattern Search (GPS) algorithm
- `GlobalGPS` – global random search, followed by GPS

class `optimizers.gps_search.`**`LocalGPS`**(*o*, *lb*, *ub*, *hyperparams*)

Use Generalized Pattern Search (GPS) to identify local solutions.

Applies GPS to the surrogate problem, in order to identify design points that are locally Pareto optimal, with respect to the surrogate problem.

`__init__`(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the LocalGPS class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure. It may contain the following:
  - opt_budget (int): The GPS iteration limit (default: 1000).
  - opt_restarts (int): Number of multisolve restarts per scalarization (default: n+1).

Returns

A new SurrogateOptimizer object.

Return type

SurrogateOptimizer

`solve`(*x*)

Solve the surrogate problem using generalized pattern search (GPS).

Parameters

x (*np.ndarray*) – A 2d array containing a list of feasible design points used to warm start the search.

Returns

A 2d numpy.ndarray of potentially efficient design points that were found by the GPS optimizer.

Return type

np.ndarray

class `optimizers.gps_search.`**`GlobalGPS`**(*o*, *lb*, *ub*, *hyperparams*)

Use randomized search globally followed by GPS locally.

Use `RandomSearch` to globally search the design space (search phase) followed by `LocalGPS` to refine the potentially efficient solutions (poll phase).

`__init__`(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the GlobalGPS class.

Parameters

- **o** (*int*) – The number of objectives.

- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure. It may contain the following:
  - opt_budget (int): The function evaluation budget (default: 10,000)
  - gps_budget (int): The number of the total opt_budget evaluations that will be used by GPS (default: half of opt_budget).

Returns
A new SurrogateOptimizer object.

Return type
`SurrogateOptimizer`

`solve`(*x*)

Solve the surrogate problem by using random search followed by GPS.

Parameters
x (*np.ndarray*) – A 2d array containing a list of feasible design points used to warm start the search.

Returns
A 2d numpy.ndarray containing a list of potentially efficient design points that were found by the optimizers.

Return type
np.ndarray

Random Search Techniques (gradient-free)

Implementations of the SurrogateOptimizer class.

This module contains implementations of the SurrogateOptimizer ABC, which are based on randomized search strategies.

Note that these strategies are all gradient-free, and therefore does not require objective, constraint, or surrogate gradients methods to be defined.

The classes include:

- `RandomSearch` – search globally by generating random samples

`class optimizers.random_search.RandomSearch`(*o*, *lb*, *ub*, *hyperparams*)

Use randomized search to identify potentially efficient designs.

Randomly search the design space and use the surrogate models to predict whether each search point is potentially Pareto optimal.

`__init__`(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the RandomSearch class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.

- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure. It may contain the following:
  - opt_budget (int): The sample size (default 10,000)

Returns
A new SurrogateOptimizer object.

Return type
SurrogateOptimizer

solve(*x*)

Solve the surrogate problem using random search.

Parameters
x (*np.ndarray*) – A 2d array containing a list of feasible design points used to warm start the search.

Returns
A 2d numpy.ndarray containing a list of potentially efficient design points that were found by the random search.

Return type
np.ndarray

L-BFGS-B Variations (gradient-based)

Implementations of the SurrogateOptimizer class.

This module contains implementations of the SurrogateOptimizer ABC, which are based on the L-BFGS-B quasi-Newton algorithm.

Note that all of these methods are gradient based, and therefore require objective, constraint, and surrogate gradient methods to be defined.

The classes include:

- LBFGSB – Limited-memory bound-constrained BFGS (L-BFGS-B) method
- TR_LBFGSB – L-BFGS-B is applied within a trust region

class optimizers.lbfgsb.**LBFGSB**(*o*, *lb*, *ub*, *hyperparams*)

Use L-BFGS-B and gradients to identify local solutions.

Applies L-BFGS-B to the surrogate problem, in order to identify design points that are locally Pareto optimal with respect to the surrogate problem.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the LBFGSB class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.

  - **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure. It may contain the following:
    - opt_budget (int): The evaluation budget per solve (default: 1000).
    - opt_restarts (int): Number of multisolve restarts per scalarization (default: n+1).

Returns
A new SurrogateOptimizer object.

Return type
SurrogateOptimizer

solve(*x*)

Solve the surrogate problem using L-BFGS-B.

Parameters
x (*np.ndarray*) – A 2d array containing a list of design points used to warm start the search.

Returns
A 2d numpy.ndarray of potentially efficient design points that were found by L-BFGS-B.

Return type
np.ndarray

class optimizers.lbfgsb.**TR_LBFGSB**(*o*, *lb*, *ub*, *hyperparams*)

Use L-BFGS-B and gradients to identify solutions within a trust region.

Applies L-BFGS-B to the surrogate problem, in order to identify design points that are locally Pareto optimal with respect to the surrogate problem.

__init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the TR_LBFGSB class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure. It may contain the following:
  - opt_budget (int): The evaluation budget per solve (default: 1000).
  - opt_restarts (int): Number of multisolve restarts per scalarization (default: 2).

Returns
A new SurrogateOptimizer object.

Return type
SurrogateOptimizer

solve(*x*)

Solve the surrogate problem using L-BFGS-B.

Parameters
x (*np.ndarray*) – A 2d array containing a list of design points used to warm start the search.

Returns
A 2d numpy.ndarray of potentially efficient design points that were found by L-BFGS-B.

Return type
np.ndarray

2.1.4 Search Techniques

A search technique is associated with each simulation when the simulation dictionary is added to the `MOOP` object. This technique is used for generating simulation data prior to the first iteration of ParMOO, so that the initial surrogate models can be fit.

For most search techniques, it is highly recommended that you supply the following optional hyperparameter keys/values:

- `search_budget (int)`: specifies how many samples will be generated for this simulation.

```
from parmoo import searches
```

Available search techniques are as follows:

Latin Hypercube Sampling

Implementations of the GlobalSearch class.

This module contains implementations of the GlobalSearch ABC, which are based on the Latin hypercube design.

The classes include:

- `LatinHypercube` – Latin hypercube sampling

`class searches.latin_hypercube.`**`LatinHypercube`**`(`*m*, *lb*, *ub*, *hyperparams*`)`

Implementation of a Latin hypercube search.

This GlobalSearch strategy uses a Latin hypercube design to sample in the design space.

`__init__``(`*m*, *lb*, *ub*, *hyperparams*`)`

Constructor for the LatinHypercube GlobalSearch class.

Parameters

- **m** (*int*) – The number of simulation outputs (unused by this class).
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the LatinHypercube design. It may contain:
  - search_budget (int): The sim eval budget for the search

Returns
A new LatinHypercube object.

Return type
`LatinHypercube`

`startSearch``(`*lb*, *ub*`)`

Begin a new Latin hypercube sampling.

Parameters

- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The dimension must match n.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match n.

Returns
A 2d array, containing the list of design points to be evaluated.

Return type
np.ndarray

resumeSearch()

Resume a previous Latin hypercube sampling.

Returns
A 2d array, containing the list of design points to be evaluated.

Return type
np.ndarray

2.1.5 Surrogate Functions

A surrogte model is associated with each simulation when its simulation dictionary is added to the `MOOP` object. This technique is used for generatng an approximation to the simulation's response surface, based on data gathered during the solve.

```
from parmoo import surrogates
```

Available techniques are:

Gaussian Process (RBF) Models

Implementations of the SurrogateFunction class.

This module contains implementations of the SurrogateFunction ABC, which rely on Gaussian basis functions (i.e., Gaussian processes).

The classes include:

- `GaussRBF` – fits Gaussian radial basis functions (RBFs)
- `LocalGaussRBF` – fits Gaussian radial basis functions (RBFs) locally

class surrogates.gaussian_proc.**GaussRBF**(*m*, *lb*, *ub*, *hyperparams*)

A RBF surrogate model, using a Gaussian basis.

This class implements a RBF surrogate with a Gaussian basis, using the SurrogateFunction ABC.

__init__(*m*, *lb*, *ub*, *hyperparams*)

Constructor for the GaussRBF class.

Parameters

- **m** (*int*) – The number of objectives to fit.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.

- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the RBF models, including:
  - des_tols (numpy.ndarray, optional): A 1d array whose length matches lb and ub. Each entry is a number (greater than 0) specifying the design space tolerance for that variable. By default, des_tols = [1.0e-8, . . . , 1.0e-8].

Returns
A new GaussRBF object.

Return type
GaussRBF

fit(*x*, *f*)

Fit a new Gaussian RBF to the given data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of design points.
- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

update(*x*, *f*)

Update an existing Gaussian RBF using new data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of new design points, with which to update the surrogate models.
- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

evaluate(*x*)

Evaluate the Gaussian RBF at a design point.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which to the Gaussian RBF should be evaluated.

Returns
A 1d array containing the predicted objective value at x.

Return type
numpy.ndarray

gradient(*x*)

Evaluate the gradients of the Gaussian RBF at a design point.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of the RBF should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the RBF interpolants at x.

Return type
numpy.ndarray

stdDev(*x*)

Evaluate the standard deviation (uncertainty) of the Gaussian RBF at x.

Parameters
`x` (*numpy.ndarray*) – A 1d array containing the design point at which the standard deviation should be evaluated.

Returns
A 1d array containing the standard deviation at x.

Return type
numpy.ndarray

stdDevGrad(*x*)

Evaluate the gradient of the standard deviation of the GaussRBF at x.

Parameters
`x` (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of standard deviation should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the standard deviation at x.

Return type
numpy.ndarray

save(*filename*)

Save important data from this class so that it can be reloaded.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data should be saved.

load(*filename*)

Reload important data into this class after a previous save.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data has been saved.

class surrogates.gaussian_proc.**LocalGaussRBF**(*m*, *lb*, *ub*, *hyperparams*)

A local RBF surrogate model, using a Gaussian basis.

This class implements a local RBF surrogate with a Gaussian basis, using the SurrogateFunction ABC.

__init__(*m*, *lb*, *ub*, *hyperparams*)

Constructor for the LocalGaussRBF class.

Parameters

- `m` (*int*) – The number of objectives to fit.
- `lb` (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- `ub` (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- `hyperparams` (*dict*) – A dictionary of hyperparameters for the RBF models, including:
  - des_tols (numpy.ndarray, optional): A 1d array whose length matches lb and ub. Each entry is a number (greater than 0) specifying the design space tolerance for that variable. By default, des_tols = [1.0e-8, …, 1.0e-8].

Returns
A new LocalGaussRBF object.

Return type
`LocalGaussRBF`

`fit`(*x*, *f*)

Fit a new Gaussian RBF to the given data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of design points.
- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

`update`(*x*, *f*)

Update an existing Gaussian RBF using new data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of new design points, with which to update the surrogate models.
- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

`setCenter`(*center*)

Set the new trust region center and refit the local RBF.

Parameters
center (*numpy.ndarray*) – A 1d array containing the new trust region center.

Returns
The standard deviation used for fitting the surrogates, which should be used as the trust region radius for a local optimizer.

Return type
float

`evaluate`(*x*)

Evaluate the Gaussian RBF at a design point.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which to the Gaussian RBF should be evaluated.

Returns
A 1d array containing the predicted objective value at x.

Return type
numpy.ndarray

`gradient`(*x*)

Evaluate the gradients of the Gaussian RBF at a design point.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of the RBF should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the RBF interpolants at x.

Return type
numpy.ndarray

stdDev(*x*)

Evaluate the standard deviation (uncertainty) of the Gaussian RBF at x.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the standard deviation should be evaluated.

Returns
A 1d array containing the standard deviation at x.

Return type
numpy.ndarray

stdDevGrad(*x*)

Evaluate the gradient of the standard deviation of the GaussRBF at x.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of standard deviation should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the standard deviation at x.

Return type
numpy.ndarray

improve(*x*, *global_improv*)

Suggests a design to evaluate to improve the RBF model near x.

Parameters

- **x** (*numpy.ndarray*) – A 1d array containing the design point at which the RBF should be improved.
- **global_improv** (*Boolean*) – When True, returns a point for global improvement, ignoring the value of x.

Returns
A 2d array containing the list of design points that should be evaluated to improve the RBF models.

Return type
numpy.ndarray

save(*filename*)

Save important data from this class so that it can be reloaded.

Parameters
filename (*string*) – The relative or absolute path to the file where all reload data should be saved.

load(*filename*)

Reload important data into this class after a previous save.

Parameters
filename (*string*) – The relative or absolute path to the file where all reload data has been saved.

2.1.6 Built-in Problem Libraries

We provide several modules containing common objective and constraint functions, which match the ParMOO interface and already support gradient-based solvers.

You can import these and use them to help define your MOOP.

```
from parmoo import objectives
from parmoo import constraints
```

We also provide templates for defining callable objects, which match ParMOO's interface.

```
from parmoo.simulations import sim_func
from parmoo.objectives import obj_func
from parmoo.constraints import const_func
```

Current options are:

Simulation Templates (ABCs)

Abstract base class (ABC) for simulation function outputs.

Defines an ABC for the callable `sim_func` class.

class `simulations.sim_func.`**`sim_func`**(*des*)

Abstract base class (ABC) for simulation function outputs.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method is already implemented, and is the constructor.

The `__call__` method is left to be implemented, and performs the simulation evaluation.

`__init__`(*des*)

Constructor for simulation functions.

Parameters

des (*numpy.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.

abstract `__call__`(*x*)

Make sim_func objects callable.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

Objective Templates (ABCs)

Abstract base class (ABC) for objective functions.

Defines an ABC for the callable `obj_func` class.

class `objectives.obj_func.`**`obj_func`**(*des*, *sim*)

Abstract base class (ABC) for objective function outputs.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x, sx, der=0)`

The `__init__` method is already implemented, and is the constructor.

The `__call__` method is left to be implemented, and performs the objective evaluation.

`__init__`(*des*, *sim*)

Constructor for objective functions.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*list, tuple, or int*) – Either the numpy.dtype of the simultation outputs (list or tuple) or the number of simulation outputs (assumed to all be continuous, unnamed).

abstract **`__call__`**(*x*, *sx*, *der=0*)

Make obj_func objects callable.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this objective for the input x.

Return type

float

Objective Function Library

This module contains a library of common objective functions, matching ParMOO's interface.

The common objectives are:

- `single_sim_out` – min or max a single simulation output
- `sos_sim_out` – min or max the sum-of-squares for several sim outputs
- `sum_sim_out` – min or max the (absolute) sum of several sim outputs

class `objectives.obj_lib.`**`single_sim_out`**(*des*, *sim*, *sim_ind*, *goal='min'*)

Class for optimizing a single simulation's output.

Minimize or maximize a single simulation output. This simulation's value will be used as an objective.

If minimizing:

```
def obj_func(x, sx, der=0):  return sx[self.sim_ind]
```

If maximizing:

`def obj_func(x, sx, der=0):  return -sx[self.sim_ind]`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_ind, goal='min')`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` returns sim[self.sim_ind].

`__init__`(*des*, *sim*, *sim_ind*, *goal='min'*)

Constructor for single_sim_out class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.
- **sim_ind** (*int, str, or tuple*) – The index or name of the simulation output to minimize or maximize. Use an integer index when des & sim contain unnamed types. Use a str name when des & sim contain named types. Use a tuple when sim[sim_ind] has multiple outputs, where the first entry is the name of the simulation, and the second entry is the index of that simulation's output to minimize/maximize.
- **goal** (*str*) – Either 'min' to minimize or 'max' to maximize. Defaults to 'min'.

`__call__`(*x*, *sim*, *der=0*)

Define objective evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns

The output of this objective for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type

float or numpy.array

class `objectives.obj_lib.`**`sos_sim_out`**(*des*, *sim*, *sim_inds*, *goal='min'*)

Class for optimizing the sum-of-squared simulation outputs.

Minimize or maximize the sum-of-squared simulation outputs. This sum-of-squares (SOS) will be used as an objective.

If minimizing:

`def obj_func(x, sx, der=0):  return sum([sx[i]**2 for i in sim_inds])`

If maximizing:

`def obj_func(x, sx, der=0):  return -sum([sx[i]**2 for i in sim_inds])`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_inds, goal='min')`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` evaluate the sum-of-square outputs.

`__init__`(*des*, *sim*, *sim_inds*, *goal='min'*)

Constructor for sos_sim_out class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.
- **sim_inds** (*list*) – The list of indices or names of the simulation outputs to sum over.
- **goal** (*str*) – Either 'min' to minimize SOS or 'max' to maximize SOS. Defaults to 'min'.

`__call__`(*x*, *sim*, *der=0*)

Define objective evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns

The output of this objective for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type
float or numpy.array

class `objectives.obj_lib.`**`sum_sim_out`**(*des*, *sim*, *sim_inds*, *goal='min'*, *absolute=False*)

Class for optimizing the sum of simulation outputs.

Minimize or maximize the (absolute) sum of simulation output. This sum will be used as an objective.

If minimizing:

`def obj_func(x, sx, der=0):  return sum([sx[i] for i in sim_inds])`

If maximizing:

`def obj_func(x, sx, der=0):  return -sum([sx[i] for i in sim_inds])`

If minimizing absolute sum:

`def obj_func(x, sx, der=0):  return sum([abs(sx[i]) for i in sim_inds])`

If maximizing absolute sum:

`def obj_func(x, sx, der=0):  return -sum([abs(sx[i]) for i in sim_inds])`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_inds, goal='min', absolute=False)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` evaluate the (absolute) sum outputs.

`__init__`(*des*, *sim*, *sim_inds*, *goal='min'*, *absolute=False*)

Constructor for sum_sim_out class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.
- **sim_inds** (*list*) – The list of indices or names of the simulation outputs to sum over.
- **goal** (*str*) – Either 'min' to minimize sum or 'max' to maximize sum. Defaults to 'min'.
- **absolute** (*bool*) – True to min/max absolute sum, False to min/max raw sum. Defaults to False.

`__call__`(*x*, *sim*, *der=0*)

Define objective evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x

  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The output of this objective for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type
float or numpy.array

Constraint Function Templates (ABCs)

Abstract base class (ABC) for constraint functions.

Defines an ABC for the callable `const_func` class.

`class constraints.const_func.`**`const_func`**(*des*, *sim*)

Abstract base class (ABC) for constraint functions.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x, sx, der=0)`

The `__init__` method is already implemented, and is the constructor.

The `__call__` method is left to be implemented, and performs the constraint evaluation.

`__init__`(*des*, *sim*)

Constructor for constraint functions.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*list, tuple, or int*) – Either the numpy.dtype of the simultation outputs (list or tuple) or the number of simulation outputs (assumed to all be continuous, unnamed).

`abstract` **`__call__`**(*x*, *sim*, *der=0*)

Make const_func objects callable.

Parameters
x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The constraint violation for the input x.

Return type
float

Constraint Function Library

This module contains a library of common constraint functions, matching ParMOO's interface.

The common constraints are:

- `single_sim_bound` – min or max bound on a single simulation output
- `sos_sim_bound` – min or max bound on the SOS for several sim outputs
- `sum_sim_out` – min or max bound on the (abs) sum of several sim outputs

`class constraints.const_lib.`**`single_sim_bound`**(*des*, *sim*, *sim_ind*, *type='upper'*, *bound=0.0*)

Class for bounding a single simulation's output.

Upper or lower bound a single simulation output.

If upper-bounding:

`def const_func(x, sx, der=0):  return sx[self.sim_ind] - upper_bound`

If lower-bounding:

`def obj_func(x, sx, der=0):  return lower_bound - sx[self.sim_ind]`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_ind, type='min', bound=0.0)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the const_func ABC.

The `__call__` returns the slace (negative when feasible).

`__init__`(*des*, *sim*, *sim_ind*, *type='upper'*, *bound=0.0*)

Constructor for single_sim_bound class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.
- **sim_ind** (*int, str, or tuple*) – The index or name of the simulation output to minimize or maximize. Use an integer index when des & sim contain unnamed types. Use a str name when des & sim contain named types. Use a tuple when sim[sim_ind] has multiple outputs, where the first entry is the name of the simulation, and the second entry is the index of that simulation's output to minimize/maximize.
- **type** (*str*) – Either 'lower' to lower-bound or 'upper' to upper-bound. Defaults to 'upper'.
- **bound** (*float*) – The lower/upper bound for this constraint. Defaults to 0.

`__call__`(*x*, *sim*, *der=0*)

Define simulation evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The (negative when feasible) slack in this constraint for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type
float or numpy.array

class constraints.const_lib.**sos_sim_bound**(*des*, *sim*, *sim_inds*, *type='upper'*, *bound=0.0*)

Class for constraining the sum-of-squared simulation outputs.

Upper or lower bound the sum-of-squared simulation outputs.

If upper bounding:

`def obj_func(x, sx):  return sum([sx[i]**2 for all i]) - upper_bound`

If lower bounding:

`def obj_func(x, sx):  return lower_bound - sum([sx[i]**2 for all i])`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_inds, type='upper', bound=0.0)`
- `__call__(x, sx, der=0)`

The `__init__` method inherits from the const_func ABC.

The `__call__` evaluate the slack (negative values are feasible).

`__init__`(*des*, *sim*, *sim_inds*, *type='upper'*, *bound=0.0*)

Constructor for sos_sim_bound class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.
- **sim_inds** (*list*) – The list of indices or names of the simulation outputs to sum over.
- **type** (*str*) – Either 'lower' to lower-bound or 'upper' to upper-bound. Defaults to 'upper'.
- **bound** (*float*) – The lower/upper bound for this constraint. Defaults to 0.

`__call__`(*x*, *sim*, *der=0*)

Define constraint evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The (negative when feasible) slack in this constraint for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type
float or numpy.array

`class constraints.const_lib.`**`sum_sim_bound`**(*des*, *sim*, *sim_inds*, *type='upper'*, *bound=0.0*, *absolute=False*)

Class for bounding the sum of simulation outputs.

Upper or lower bound the (absolute) sum of simulation output.

If upper bounding:

`def const_func(x, sx):  return sum([sx[i] for all i]) - upper_bound`

If lower bounding:

`def const_func(x, sx):  return lower_bound - sum([sx[i] for all i])`

If upper bounding absolute sum:

`def const_func(x, sx):  return sum([abs(sx[i]) forall i]) - upper_bound`

If lower bounding absolute sum:

`def const_func(x, sx):  return lower_bound - sum([abs(sx[i]) forall i])`

Also supports derivative usage.

Contains 2 methods:

- `__init__(des, sim, sim_inds, type='upper', bound=0, absolute=False)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the const_func ABC.

The `__call__` evaluate the slack (negative values are feasible).

`__init__`(*des*, *sim*, *sim_inds*, *type='upper'*, *bound=0.0*, *absolute=False*)

Constructor for sum_sim_bound class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be unnamed.
- **sim** (*np.dtype or int*) – Either the numpy.dtype of the simulation outputs or the number of simulation outputs, assumed to all be unnamed.

- **sim_inds** (*list*) – The list of indices or names of the simulation outputs to sum over.
- **type** (*str*) – Either 'lower' to lower-bound or 'upper' to upper-bound. Defaults to 'upper'.
- **bound** (*float*) – The lower/upper bound for this constraint. Defaults to 0.
- **absolute** (*bool*) – True to bound absolute sum, False to bound raw sum. Defaults to False.

__call__(*x*, *sim*, *der=0*)

Define constraint evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns

The (negative when feasible) slack in this constraint for the input x (der=0), the gradient with respect to x (der=1), or the gradient with respect to sim (der=2).

Return type

float or numpy.array

2.1.7 The DTLZ Problem Library

To facilitate testing and comparison with other methods, we provide built-in implementations of the DTLZ problems as ParMOO Simulations and Objectives (with gradients defined).

```
from parmoo.simulations import dtlz
from parmoo.objectives import dtlz
```

DTLZ Problems as Simulations

This module contains simulation function implementations of the DTLZ test suite, as described in:

- Deb, Thiele, Laumanns, and Zitzler. "Scalable test problems for evolutionary multiobjective optimization" in Evolutionary Multiobjective Optimization, Theoretical Advances and Applications, Ch. 6 (pp. 105–145). Springer-Verlag, London, UK, 2005. Abraham, Jain, and Goldberg (Eds).

When run with default settings, the outputs of `dtlz{1-7}_sim` have been confirmed against the outputs from the corresponding problems in `pymoo` up to 8 decimal places of precision.

- Blank and Deb. "pymoo: Multi-Objective Optimization in Python." IEEE Access 8 (pp. 89497–89509). 2020.

One drawback of the original DTLZ problems is that their global minima (Pareto points) always corresponded to design points that satisfy x_i = 0.5, for i = number of objectives, . . . , number of design points

or

x_i = 0, for i = number of objectives, . . . , number of design points.

This was appropriate for testing evolutionary algorithms, but for many deterministic algorithms, these solutions may represent either the best- or worst-case scenarios.

To make these problems applicable for deterministic algorithms, the solution sets must be configurable offset by a user-specified amount, as proposed in:

- Chang. Mathematical Software for Multiobjective Optimization Problems. Ph.D. dissertation, Virginia Tech, Dept. of Computer Science, 2020.

For the problems DTLZ8 and DTLZ9, only objective outputs are given by the simulation function herein. To fully define the problem, also use one or more of the corresponding constraint classes included in `parmoo.constraints.dtlz` [NOT YET IMPLEMENTED].

The full list of simulation functions in this module includes the kernel functions:

- `g1_sim`
- `g2_sim`
- `g3_sim`
- `g4_sim`

and the 9 DTLZ problems in simulation form, with each simulation output corresponding to an objective:

- `dtlz1_sim`
- `dtlz2_sim`
- `dtlz3_sim`
- `dtlz4_sim`
- `dtlz5_sim`
- `dtlz6_sim`
- `dtlz7_sim`
- `dtlz8_sim`
- `dtlz9_sim`

`class simulations.dtlz.`**`g1_sim`**`(`*des*, *num_obj=3*, *offset=0.5*`)`

Class defining 1 of 4 kernel functions used in the DTLZ problem suite.

g1 = 100 ((n - o + 1) +
sum_{i=o}^n ((x_i - offset)^2 - cos(20pi(x_i - offset))))

Contains 2 methods:

- `__init__(des, num_obj)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the g1 kernel.

`__init__`(*des*, *num_obj=3*, *offset=0.5*)

Constructor for g1 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used to calculate the value of g1. Note that regardless of the number of objectives, the number of simulation outputs from g1 is always 1.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__`(*x*)

Define simulation evaluation.

Parameters
x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

class simulations.dtlz.**g2_sim**(*des*, *num_obj=3*, *offset=0.5*)

Class defining 2 of 4 kernel functions used in the DTLZ problem suite.

g2 = (x_o - offset)^2 + . . . + (x_n - offset)^2

Contains 2 methods:

- `__init__(des)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the g2 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.5*)

Constructor for g2 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used to calculate the value of g2. Note that regardless of the number of objectives, the number of simulation outputs from g2 is always 1.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__`(*x*)

Define simulation evaluation.

Parameters
x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

class `simulations.dtlz.g3_sim`(*des*, *num_obj=3*, *offset=0.0*)

Class defining 3 of 4 kernel functions used in the DTLZ problem suite.

g3 = |x_o - offset|^.1 + … + |x_n - offset|^.1

Contains 2 methods:

- `__init__(des)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the g3 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for g3 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used to calculate the value of g3. Note that regardless of the number of objectives, the number of simulation outputs from g3 is always 1.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, …, number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters
x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

class `simulations.dtlz.g4_sim`(*des*, *num_obj=3*, *offset=0.0*)

Class defining 4 of 4 kernel functions used in the DTLZ problem suite.

g4 = 1 + (9 * (**|x_o - offset|** + … + **|x_n - offset|**) / (n + 1 - o))

Contains 2 methods:

- `__init__(des)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the g4 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for g4 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used to calculate the value of g4. Note that regardless of the number of objectives, the number of simulation outputs from g4 is always 1.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

class simulations.dtlz.**dtlz1_sim**(*des*, *num_obj=3*, *offset=0.5*)

Class defining the DTLZ1 problem with offset minimizer.

DTLZ1 has a linear Pareto front, with all nondominated points on the hyperplane F_1 + F_2 + . . . + F_o = 0.5. DTLZ1 has 11^k - 1 "local" Pareto fronts where k = n - o + 1, and 1 "global" Pareto front.

Contains 2 methods:

- `__init__(des)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ1 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.5*)

Constructor for DTLZ1 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

`class simulations.dtlz.dtlz2_sim(des, num_obj=3, offset=0.5)`

Class defining the DTLZ2 problem with offset minimizer.

DTLZ2 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ2 has no "local" Pareto fronts, besides the true Pareto front.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ2 problem.

`__init__(des, num_obj=3, offset=0.5)`

Constructor for DTLZ2 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__(x)`

Define simulation evaluation.

Parameters
x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

`class simulations.dtlz.dtlz3_sim(des, num_obj=3, offset=0.5)`

Class defining the DTLZ3 problem with offset minimizer.

DTLZ3 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ3 has 3^k - 1 "local" Pareto fronts where k = n - o + 1, and 1 "global" Pareto front.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ3 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.5*)

Constructor for DTLZ3 class.

Parameters

- `des` (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- `num_obj` (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- `offset` (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__`(*x*)

Define simulation evaluation.

Parameters

`x` (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

class simulations.dtlz.**dtlz4_sim**(*des*, *num_obj=3*, *offset=0.5*, *alpha=100.0*)

Class defining the DTLZ4 problem with offset minimizer.

DTLZ4 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ4 has no "local" Pareto fronts, besides the true Pareto front, but by tuning the optional parameter alpha, one can adjust the solution density, making it harder for MOO algorithms to produce a uniform distribution of solutions.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ4 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.5*, *alpha=100.0*)

Constructor for DTLZ4 class.

Parameters

- `des` (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- `num_obj` (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- `offset` (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.
- `alpha` (*optional, float or int*) – The uniformity parameter used for controlling the uniformity of the distribution of solutions across the Pareto front. Must be greater than or equal to 1. A value of 1 results in DTLZ2. Default value is 100.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

`x` (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

`class simulations.dtlz.dtlz5_sim`(*des*, *num_obj=3*, *offset=0.5*)

Class defining the DTLZ5 problem with offset minimizer.

DTLZ5 has a lower-dimensional Pareto front embedded in the objective space, given by an arc of the unit sphere in the positive orthant. DTLZ5 has no "local" Pareto fronts, besides the true Pareto front.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ5 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.5*)

Constructor for DTLZ5 class.

Parameters

- `des` (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- `num_obj` (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- `offset` (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.5.

`__call__`(*x*)

Define simulation evaluation.

Parameters

`x` (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

`class simulations.dtlz.dtlz6_sim`(*des*, *num_obj=3*, *offset=0.0*)

Class defining the DTLZ6 problem with offset minimizer.

DTLZ6 has a lower-dimensional Pareto front embedded in the objective space, given by an arc of the unit sphere in the positive orthant. DTLZ6 has no "local" Pareto fronts, but tends to show very little improvement until the algorithm is very close to its solution set.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ6 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for DTLZ6 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, …, number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

class simulations.dtlz.**dtlz7_sim**(*des*, *num_obj=3*, *offset=0.0*)

Class defining the DTLZ7 problem with offset minimizer.

DTLZ7 has a discontinuous Pareto front, with solutions on the 2^(o-1) discontinuous nondominated regions of the surface:

F_m = o - F_1 (1 + sin(3pi F_1)) - … - F_{o-1} (1 + sin3pi F_{o-1}).

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ7 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for DTLZ7 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, …, number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

`class simulations.dtlz.`**`dtlz8_sim`**(*des*, *num_obj=3*, *offset=0.0*)

Class defining the DTLZ8 problem with offset minimizer.

DTLZ8 is a constrained MOOP, whose Pareto front combines a region of a plane with a line segment. To fully define DTLZ8, you must also use the parmoo.constraints.dtlz module.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ8 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for DTLZ8 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

`class simulations.dtlz.`**`dtlz9_sim`**(*des*, *num_obj=3*, *offset=0.0*)

Class defining the DTLZ9 problem with offset minimizer.

DTLZ9 is a constrained MOOP, whose Pareto front is a subregion of the arc traced out by the solution to DTLZ5. To fully define DTLZ8, you must also use the parmoo.constraints.dtlz module.

Contains 2 methods:

- `__init__(des, sim)`
- `__call__(x)`

The `__init__` method inherits from the sim_func ABC.

The `__call__` method performs an evaluation of the DTLZ9 problem.

`__init__`(*des*, *num_obj=3*, *offset=0.0*)

Constructor for DTLZ9 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **num_obj** (*int, optional*) – The number of objectives, which is used as the number of simulation outputs.
- **offset** (*optional, float*) – The location of the global minimizers is x_i = offset for i = number of objectives, . . . , number of design variables. The default value is offset = 0.0.

`__call__`(*x*)

Define simulation evaluation.

Parameters

x (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.

Returns

The output of this simulation for the input x.

Return type

numpy.ndarray

`simulations.dtlz.`**`__check_optionals__`**(*num_obj=3*, *offset=0.5*, *alpha=100.0*)

Check DTLZ optional inputs for illegal values.

Not recommended for external usage.

DTLZ Problems as Objectives

This module contains objective function implementations of the DTLZ test suite, as described in:

Deb, Thiele, Laumanns, and Zitzler. "Scalable test problems for evolutionary multiobjective optimization" in Evolutionary Multiobjective Optimization, Theoretical Advances and Applications, Ch. 6 (pp. 105–145). Springer-Verlag, London, UK, 2005. Abraham, Jain, and Goldberg (Eds).

Since DTLZ[1-7] depended upon kernel functions (implemented in parmoo.simulations.dtlz), each of these problems is implemented here as an algebraic, differentiable objective, with the kernel function output as an input. The problems DTLZ8 and DTLZ9 do not support this modification, so they are omitted.

TODO: DTLZ5, DTLZ6, and DTLZ7 have not yet been added.

To use this module, first import one or more of the following simulation/kernel functions from parmoo.simulations.dtlz:

- `g1_sim`
- `g2_sim`
- `g3_sim`
- `g4_sim`

The 7 DTLZ problems included here are:

- `dtlz1_obj`
- `dtlz2_obj`
- `dtlz3_obj`
- `dtlz4_obj`

`class objectives.dtlz.`**`dtlz1_obj`**(*des*, *sim*, *obj_ind*, *num_obj=3*)

Class defining the DTLZ1 objectives.

Use this class in combination with the g1_sim() class from the parmoo.simulations.dtlz module

DTLZ1 has a linear Pareto front, with all nondominated points on the hyperplane F_1 + F_2 + … + F_o = 0.5. DTLZ1 has 11^k - 1 "local" Pareto fronts where k = n - m + 1, and 1 "global" Pareto front.

Contains 2 methods:

- `__init__(des, sim, obj_ind)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` method performs an evaluation of the DTLZ1 problem.

`__init__`(*des*, *sim*, *obj_ind*, *num_obj=3*)

Constructor for DTLZ1 class.

Parameters

- **des** (*`np.dtype or int`*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*`list, tuple, or int`*) – Either the numpy.dtype of the simulation outputs (list or tuple) or the number of simulation outputs (assumed to all be unnamed).
- **obj_ind** (*`int`*) – The index of the DTLZ1 objective to return.
- **num_obj** (*`int, optional`*) – The number of objectives for this problem. Note that this effects the calculation of the objective value, but still only a single objective output is created per instance of this class. To add all objectives, create num_obj instances with obj_ind = 0, …, num_obj - 1.

`__call__`(*x*, *sim*, *der=0*)

Define simulation evaluation.

Parameters

- **x** (*`numpy.array`*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*`numpy.array`*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*`int, optional`*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

`class objectives.dtlz.dtlz2_obj`(*des*, *sim*, *obj_ind*, *num_obj=3*)

Class defining the DTLZ2 objectives.

Use this class in combination with the g2_sim() class from the parmoo.simulations.dtlz module.

DTLZ2 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ2 has no "local" Pareto fronts, besides the true Pareto front.

Contains 2 methods:

- `__init__(des, sim, obj_ind)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` method performs an evaluation of the DTLZ2 problem.

`__init__`(*des*, *sim*, *obj_ind*, *num_obj=3*)

Constructor for DTLZ2 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*list, tuple, or int*) – Either the numpy.dtype of the simulation outputs (list or tuple) or the number of simulation outputs (assumed to all be unnamed).
- **obj_ind** (*int*) – The index of the DTLZ2 objective to return.
- **num_obj** (*int, optional*) – The number of objectives for this problem. Note that this effects the calculation of the objective value, but still only a single objective output is created per instance of this class. To add all objectives, create num_obj instances with obj_ind = 0, . . . , num_obj - 1.

`__call__`(*x*, *sim*, *der=0*)

Define simulation evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

class `objectives.dtlz.`**`dtlz3_obj`**(*des*, *sim*, *obj_ind*, *num_obj=3*)

Class defining the DTLZ3 objectives.

Use this class in combination with the g1_sim() class from the parmoo.simulations.dtlz module.

DTLZ3 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ3 has 3^k - 1 "local" Pareto fronts where k = n - o + 1, and 1 "global" Pareto front.

Contains 2 methods:

- `__init__(des, sim, obj_ind)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` method performs an evaluation of the DTLZ3 problem.

`__init__`(*des*, *sim*, *obj_ind*, *num_obj=3*)

Constructor for DTLZ3 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*list, tuple, or int*) – Either the numpy.dtype of the simulation outputs (list or tuple) or the number of simulation outputs (assumed to all be unnamed).
- **obj_ind** (*int*) – The index of the DTLZ3 objective to return.
- **num_obj** (*int, optional*) – The number of objectives for this problem. Note that this effects the calculation of the objective value, but still only a single objective output is created per instance of this class. To add all objectives, create num_obj instances with obj_ind = 0, . . . , num_obj - 1.

`__call__`(*x*, *sim*, *der=0*)

Define simulation evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns
The output of this simulation for the input x.

Return type
numpy.ndarray

class `objectives.dtlz.`**`dtlz4_obj`**(*des*, *sim*, *obj_ind*, *num_obj=3*, *alpha=100.0*)

Class defining the DTLZ4 objectives.

Use this class in combination with the g2_sim() class from the parmoo.simulations.dtlz module.

DTLZ4 has a concave Pareto front, given by the unit sphere in objective space, restricted to the positive orthant. DTLZ4 has no "local" Pareto fronts, besides the true Pareto front, but by tuning the optional parameter alpha, one can adjust the solution density, making it harder for MOO algorithms to produce a uniform distribution of solutions.

Contains 2 methods:

- `__init__(des, sim, obj_ind)`
- `__call__(x, sim, der=0)`

The `__init__` method inherits from the obj_func ABC.

The `__call__` method performs an evaluation of the DTLZ4 problem.

`__init__`(*des*, *sim*, *obj_ind*, *num_obj=3*, *alpha=100.0*)

Constructor for DTLZ4 class.

Parameters

- **des** (*np.dtype or int*) – Either the numpy.dtype of the design variables or the number of design variables, assumed to all be continuous and unnamed.
- **sim** (*list, tuple, or int*) – Either the numpy.dtype of the simulation outputs (list or tuple) or the number of simulation outputs (assumed to all be unnamed).
- **obj_ind** (*int*) – The index of the DTLZ4 objective to return.
- **num_obj** (*int, optional*) – The number of objectives for this problem. Note that this effects the calculation of the objective value, but still only a single objective output is created per instance of this class. To add all objectives, create num_obj instances with obj_ind = 0, . . . , num_obj - 1.
- **alpha** (*optional, float or int*) – The uniformity parameter used for controlling the uniformity of the distribution of solutions across the Pareto front. Must be greater than or equal to 1. A value of 1 results in DTLZ2. Default value is 100.0.

`__call__`(*x*, *sim*, *der=0*)

Define simulation evaluation.

Parameters

- **x** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the design point to evaluate.
- **sim** (*numpy.array*) – A numpy.ndarray (unnamed) or numpy structured array (named), containing the corresponding simulation outputs.
- **der** (*int, optional*) – Specifies whether to take derivative (and wrt which variables).
  - der=1: take derivatives wrt x
  - der=2: take derivatives wrt sim
  - other: no derivatives

  Default value is der=0.

Returns

The output of this simulation for the input x.

Return type
numpy.ndarray

2.1.8 Extra Developer Tools

These are miscellaneous additional tools and definitions, which may be useful for developers.

ParMOO Solver and Component Definitions

To implement a new acquisition function, solver, surrogate, or search technique for ParMOO, you must match its interface. An interface definition for each of these methods is provided in the `structs` module in the corresponding Abstract Base Class (ABC).

```
from parmoo import structs
```

When implementing one of these techniques, you should extend the corresponding ABC, defined below.

Abstract base classes (ABCs) for ParMOO project.

This module contains several abstract base classes that can be used to create a flexible framework for surrogate based multiobjective optimization.

The classes include:

- AcquisitionFunction
- GlobalSearch
- SurrogateFunction
- SurrogateOptimizer

AcquisitionFunction

class structs.**AcquisitionFunction**(*o*, *lb*, *ub*, *hyperparams*)

ABC describing acquisition functions.

This class contains the following methods:

- `useSD()`
- `setTarget(data, constraint_func, history)`
- `scalarize(f_vals)`
- `scalarizeGrad(f_vals, g_vals)`
- `save(filename)`
- `load(filename)`

abstract __init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the AcquisitionFunction class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design space.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design space.

- **hyperparams** (*dict*) – A dictionary of hyperparameters that are passed to the acquisition function.

Returns
A new AcquisitionFunction object.

Return type
AcquisitionFunction

abstract setTarget(*data*, *penalty_func*, *history*)

Set a new target value or region for the AcquisitionFunction.

Parameters

- **data** (*dict*) – A dictionary specifying the current function evaluation database. It contains two mandatory fields:
  - 'x_vals' (numpy.ndarray): A 2d array containing the list of design points.
  - 'f_vals' (numpy.ndarray): A 2d array containing the corresponding list of objective values.
- **available** (*If gradients are*) –

  field:
  - 'g_vals' (numpy.ndarray): A 3d array containing the Jacobian of the objective function at each point in 'x_vals'.
- **additional** (*data may contain one*) –

  field:
  - 'g_vals' (numpy.ndarray): A 3d array containing the Jacobian of the objective function at each point in 'x_vals'.
- **penalty_func** (*function*) – A function of one (x) or two (x, sx) inputs that evaluates all (penalized) objective scores.
- **history** (*dict*) – A persistent dictionary that could be used by the implementation of the AcquisitionFunction to pass data between iterations.

Returns
A 1d array containing a feasible starting point for the scalarized problem.

Return type
numpy.ndarray

useSD()

Query whether this method uses uncertainties.

When False, allows users to shortcut expensive uncertainty computations.

Default implementation returns True, requiring full uncertainty computation for applicable models.

abstract scalarize(*f_vals*, *x_vals*, *s_vals_mean*, *s_vals_sd*)

Scalarize a vector-valued function using the AcquisitionFunction.

Parameters

- **f_vals** (*np.ndarray*) – A 1D array specifying a vector of function values to be scalarized.
- **x_vals** (*np.ndarray*) – A 1D array specifying a vector the design point corresponding to f_vals.

- **`s_vals_mean`** (*np.ndarray*) – A 1D array specifying the expected simulation outputs for the x value being scalarized.
- **`s_vals_sd`** (*np.ndarray*) – A 1D array specifying the standard deviation for each of the simulation outputs.

Returns
The scalarized value.

Return type
float

`scalarizeGrad`(*f_vals*, *g_vals*)

Scalarize a Jacobian of gradients using the current weights.

Parameters

- **`f_vals`** (*numpy.ndarray*) – A 1d array specifying the function values for the scalarized gradient.
- **`g_vals`** (*numpy.ndarray*) – A 2d array specifying the gradient values to be scalarized.

Returns
The 1d array for the scalarized gradient.

Return type
np.ndarray

`save`(*filename*)

Save important data from this class so that it can be reloaded.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the load method need not be implemented.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data should be saved.

`load`(*filename*)

Reload important data into this class after a previous save.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the save method need not be implemented.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data has been saved.

GlobalSearch

class `structs.`**`GlobalSearch`**(*o*, *lb*, *ub*, *hyperparams*)

ABC describing global search techniques.

This class contains the following methods.

- `startSearch(lb, ub)`
- `resumeSearch()`
- `save(filename)`

- `load(filename)`

abstract __init__(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the GlobalSearch class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design space.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design space.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the global search. It may contain any inputs specific to the search algorithm.

Returns
A new GlobalSearch object.

Return type
GlobalSearch

abstract startSearch(*lb*, *ub*)

Begin a new global search.

Parameters

- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The dimension must match n.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match n.

Returns
A 2d array, containing the list of design points to be evaluated.

Return type
numpy.ndarray

resumeSearch()

Resume a global search.

Returns
A 2d array, containing the list of design points to be evaluated.

Return type
numpy.ndarray

save(*filename*)

Save important data from this class so that it can be reloaded.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the load method need not be implemented.

Parameters
filename (*string*) – The relative or absolute path to the file where all reload data should be saved.

load(*filename*)

Reload important data into this class after a previous save.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the save method need not be implemented.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data has been saved.

SurrogateFunction

class `structs.SurrogateFunction`(*m*, *lb*, *ub*, *hyperparams*)

ABC describing surrogate functions.

This class contains the following methods.

- `fit(x, f)`
- `update(x, f)`
- `setCenter(x)` (default implementation provided)
- `evaluate(x)`
- `gradient(x)`
- `stdDev(x)`
- `stdDevGrad(x)`
- `improve(x, global_improv)` (default implementation provided)
- `save(filename)`
- `load(filename)`

abstract `__init__`(*m*, *lb*, *ub*, *hyperparams*)

Constructor for the SurrogateFunction class.

Parameters

- **m** (*int*) – The number of objectives to fit.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design region. The number of design variables is inferred from the dimension of lb.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design region. The dimension must match ub.
- **hyperparams** (*dict*) – A dictionary of hyperparameters to be used by the surrogate models, including:
  - des_tols (numpy.ndarray, optional): A 1d array whose length matches lb and ub. Each entry is a number (greater than 0) specifying the design space tolerance for that variable. By default, des_tols = [1.0e-8, . . . , 1.0e-8].

Returns
A new SurrogateFunction object.

Return type
`SurrogateFunction`

abstract `fit`(*x*, *f*)

Fit a new surrogate to the given data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of design points.

- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

abstract update(*x*, *f*)

Update an existing surrogate model using new data.

Parameters

- **x** (*numpy.ndarray*) – A 2d array containing the list of new design points, with which to update the surrogate models.
- **f** (*numpy.ndarray*) – A 2d array containing the corresponding list of objective values.

setCenter(*center*)

Set the center for the fit, if this is a local method.

Default implementation returns the diameter of the design space, resulting in a nonbinding trust region.

Parameters
center (*numpy.ndarray*) – A 1d array containing the center for this local fit.

abstract evaluate(*x*)

Evaluate the surrogate at a design point.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which to the surrogate should be evaluated.

Returns
A 1d array containing the predicted objective value at x.

Return type
numpy.ndarray

gradient(*x*)

Evaluate the gradient of the surrogate at a design point.

Note: this method need not be implemented when using a derivative free SurrogateOptimization solver.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of the surrogate should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the surrogate at x.

Return type
numpy.ndarray

stdDev(*x*)

Evaluate the standard deviation (uncertainty) of the surrogate at x.

Note: this method need not be implemented when the acquisition function does not use the model uncertainty.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the standard deviation should be evaluated.

Returns
A 1d array containing the standard deviation at x.

Return type
numpy.ndarray

`stdDevGrad`(*x*)

Evaluate the gradient of the standard deviation at x.

Note: this method need not be implemented when the acquisition function does not use both the model uncertainty and gradient.

Parameters
x (*numpy.ndarray*) – A 1d array containing the design point at which the gradient of standard deviation should be evaluated.

Returns
A 2d array containing the Jacobian matrix of the standard deviation at x.

Return type
numpy.ndarray

`improve`(*x*, *global_improv*)

Suggests a design to evaluate to improve the surrogate near x.

A default implementation is given based on random sampling. Re-implement the improve method to overwright the default policy.

Parameters

- **x** (*numpy.ndarray*) – A 1d array containing the design point at which the surrogate should be improved.
- **global_improv** (*Boolean*) – When True, returns a point for global improvement, ignoring the value of x.

Returns
A 2d array containing the list of design points that should be evaluated to improve the surrogate.

Return type
numpy.ndarray

`save`(*filename*)

Save important data from this class so that it can be reloaded.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the load method need not be implemented.

Parameters
filename (*string*) – The relative or absolute path to the file where all reload data should be saved.

`load`(*filename*)

Reload important data into this class after a previous save.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the save method need not be implemented.

Parameters
filename (*string*) – The relative or absolute path to the file where all reload data has been saved.

SurrogateOptimizer

class `structs.SurrogateOptimizer`(*o*, *lb*, *ub*, *hyperparams*)

ABC describing surrogate optimization techniques.

This class contains the following methods.

- `setObjective(obj_func)` (default implementation provided)
- `setSimulation(sim_func, sd_func)` (default implementation provided)
- `setConstraints(constraint_func)` (default implementation provided)
- `setPenalty(penaltyFunc, gradFunc)` (default implementation provided)
- `setReset(reset)` (default implementation provided)
- `addAcquisition(*args)` (default implementation provided)
- `solve(x)`
- `save(filename)`
- `load(filename)`

abstract `__init__`(*o*, *lb*, *ub*, *hyperparams*)

Constructor for the SurrogateOptimizer class.

Parameters

- **o** (*int*) – The number of objectives.
- **lb** (*numpy.ndarray*) – A 1d array of lower bounds for the design space.
- **ub** (*numpy.ndarray*) – A 1d array of upper bounds for the design space.
- **hyperparams** (*dict*) – A dictionary of hyperparameters for the optimization procedure.

Returns

A new SurrogateOptimizer object.

Return type

`SurrogateOptimizer`

`setObjective`(*obj_func*)

Add a vector-valued objective function that will be solved.

Parameters

obj_func (*function*) – A vector-valued function that can be evaluated to solve the surrogate optimization problem.

`setSimulation`(*sim_func*, *sd_func=None*)

Add a vector-valued simulation function, used to calculate objs.

Parameters

- **sim_func** (*function*) – A vector-valued function that can be evaluated to determine the surrogate-predicted simulation outputs.
- **sd_func** (*function*) – A vector-valued function that can be evaluated to determine the standard deviations of the surrogate predictions.

`setPenalty`(*penalty_func*, *grad_func*)

Add a matrix-valued gradient function for obj_func.

Parameters

- **penalty_func** (*function*) – A vector-valued penalized objective that incorporates a penalty for violating constraints.
- **grad_func** (*function*) – A matrix-valued function that can be evaluated to obtain the Jacobian matrix for obj_func.

`setConstraints`(*constraint_func*)

Add a constraint function that will be satisfied.

Parameters

constraint_func (*function*) – A vector-valued function from the design space whose components correspond to constraint violations. If the problem is unconstrained, a function that returns zeros could be provided.

`setReset`(*reset*)

Add a reset function for resetting surrogate updates.

Parameters

reset (*function*) – A function with one input, which will be called prior to solving the surrogate optimization problem with each acquisition function.

`addAcquisition`(**args*)

Add an acquisition function for the surrogate optimizer.

Parameters

***args** (*AcquisitionFunction*) – Acquisition functions that are used to scalarize the list of objectives in order to solve the surrogate optimization problem.

`abstract solve`(*x*)

Solve the surrogate problem.

Parameters

x (*numpy.ndarray*) – A 2d array containing a list of feasible design points used to warm start the search.

Returns

A 2d numpy.ndarray of potentially efficient design points that were found by the surrogate optimizer.

Return type

float

`save`(*filename*)

Save important data from this class so that it can be reloaded.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the load method need not be implemented.

Parameters

filename (*string*) – The relative or absolute path to the file where all reload data should be saved.

`load`(*filename*)

Reload important data into this class after a previous save.

Note: If this function is left unimplemented, ParMOO will reinitialize a fresh instance after a save/load. If this is the desired behavior, then this method and the save method need not be implemented.

Parameters
`filename` (*string*) – The relative or absolute path to the file where all reload data has been saved.

Utilities

This module contains additional utilities, which may be useful for some developers.

```
from parmoo import util
```

This module contains several auxiliary functions used throughout ParMOO.

These functions may also be of external interest. They are:

- xerror(o, lb, ub, hyperparams)
- check_sims(n, arg1, arg2, . . .)
- lex_leq(a, b)
- updatePF(data, nondom)
- unpack(x, dtype)

`util.`**`xerror`**(*o=1*, *lb=None*, *ub=None*, *hyperparams=None*)

Typecheck the input arguments for a class interface.

Parameters

- **o** (*int*) – The number of objectives should be an int greater than or equal to 1.
- **lb** (*np.ndarray*) – The lower bounds should be a 1d array.
- **ub** (*np.ndarray*) – The upper bounds should be a 1d array with the same length as lb, and must satisfy lb[:] < ub[:].
- **hyperparams** (*dict*) – The hyperparameters must be supplied in a dictionary.

`util.`**`check_sims`**(*n*, **args*)

Check simulation dictionaries for bad input.

Parameters

- **n** (*int*) – The dimension of the design space. Used for confirming any simulation databases provided in args.
- ***args** (*dict*) – An unpacked array of dictionaries, each specifying one of the simulations. The following keys are used:
  - name (String, optional): The name of this simulation (defaults to "sim" + str(i), where i = 1, 2, 3, . . . for the first, second, third, . . . simulation added to the MOOP).
  - m (int): The number of outputs for this simulation.
  - sim_func (function): An implementation of the simulation function, mapping from R^n -> R^m. The interface should match: *sim_out = sim_func(x, der=False)*, where *der* is an optional argument specifying whether to take the derivative of the simulation. Unless otherwise specified by your solver, *der* is always omitted by ParMOO's internal structures, and need not be implemented.
  - search (GlobalSearch): A GlobalSearch object for performing the initial search over this simulation's design space.

  - surrogate (SurrogateFunction): A SurrogateFunction object specifying how this simulation's outputs will be modeled.
  - des_tol (float): The tolerance for this simulation's design space; a new design point that is closer than des_tol to a point that is already in this simulation's database will not be reevaluated.
  - hyperparams (dict): A dictionary of hyperparameters, which will be passed to the surrogate and search routines. Most notably, search_budget (int) can be specified here.
  - sim_db (dict, optional): A dictionary of previous simulation evaluations. When present, contains:
    * x_vals (np.ndarray): A 2d array of pre-evaluated design points.
    * s_vals (np.ndarray): A 2d array of corresponding simulation outputs.
    * g_vals (np.ndarray): A 3d array of corresponding Jacobian values. This value is only needed if the provided SurrogateFunction uses gradients.

`util.`**`lex_leq`**`(`*a*, *b*`)`

Lexicographically compare two vectors from back to front.

Check whether the vector a is lexicographically less than or equal to b, starting from the last element and working back to the first element.

Parameters

- **a** (*numpy.ndarray*) – The first vector to compare.
- **b** (*numpy.ndarray*) – The second vector to compare.

Returns
Whether a <= b in the lexicographical sense.

Return type
Boolean

`util.`**`updatePF`**`(`*data*, *nondom*`)`

Update the Pareto front and efficient set by resorting.

Returns

A dictionary containing a discrete approximation of the Pareto front and efficient set.

- f_vals (numpy.ndarray): A list of nondominated points discretely approximating the Pareto front.
- x_vals (numpy.ndarray): A list of corresponding efficient design points.
- c_vals (numpy.ndarray): A list of corresponding constraint satisfaction scores, all less than or equal to 0.

Return type
dict

`util.`**`unpack`**`(`*x*, *dtype*`)`

Unpack an input vector of given dtype into a numpy.ndarray.

Parameters

- **x** (*numpy.ndarray or numpy structured array*) – The input vector, which needs to be unpacked.

- **dtype** (*numpy.dtype*) – The dtype of the input x.

Returns
x unpacked into a 1-dimensional numpy.ndarray.

Return type
numpy.ndarray

2.1.9 The Interactive Visualization (viz) Library

Easily generate a locally-hosted GUI for interactively visualizing your MOOP results.

The public interface can be accessed by importing the viz module, which contains three functions for creating interactive visualizations of the approximate Pareto front and objective data; these visualizations run on the browser in a Dash app. *There is a known issue when using parmoo.viz in the Chrome browser, Firefox or Safari is recommended.*

```
import parmoo.viz
```

Public Plotting Functions Running in Python Plotly and Dash

A plotting library for interactive visualization of MOOP objects.

To generate a plot, generate create a MOOP object and use it to solve a problem. Then call one of the following function, passing the MOOP object as the first argument:

- `viz.scatter(moop)`
- `viz.parallel_coordinates(moop)`
- `viz.radar(moop)`

Please take note of the following:

- Via the interactivity features, all of the arguments except for the `moop` field can be adjusted interactively through the pop-up GUI. The plot type can also be adjusted from the dropdown menu while running the GUI..
- The viz tool hosts a Dash app through the local port `http://127.0.0.1:8050/`. This means that only one visualization can be hosted at a time. Also, killing the Dash app will end the interactivity in the browser, although a static plot may remain in your broswer window.
- Finally, note that there is a known issue when using the Chrome browser. The Firefox or Safari browsers are recommended.

To interact with the plots:

1. To multi-select in scatterplots and radar plots, hold down `SHIFT` while making additional selections.
2. To remove all selections from a scatterplot and radar plot, select without holding down the `SHIFT` key or double click.
3. To select in parallel coordinates plots, click along an axis and then drag the cursor elsewhere along the same axis. The highlighted part of the axis is the area that is selected. There is no need to hold down SHIFT when making multiple selections in parallel coordinates plots.
4. To remove a parallel coordinates plot selection, click on the highlighted selection bar you wish to delete.
5. To reset plot interactions made through the toolbar in the top right of a graph (not interactions made through buttons, dropdowns, toggles, or input boxes), double click on the plot. This will undo selections, zoom, pan, etc.

The three basic plot options are detailed below.

`viz.plot.scatter`(*moop*, *db='pf'*, *output='dash'*, *points='constraint_satisfying'*, *height='auto'*, *width='auto'*, *font='auto'*, *fontsize='auto'*, *background_color='auto'*, *screenshot='svg'*, *image_export_format='svg'*, *data_export_format='csv'*, *dev_mode=False*, *pop_up=True*, *port='http://127.0.0.1:8050/'*)

Create a scatter plot matrix to visualize the results of a MOOP.

Parameters

- **moop** (*MOOP*) – The MOOP results that you would like to visualize.
- **db** (*str*) – Either 'pf' to plot just the Pareto front, or 'obj' to plot the complete objective database. Defaults to 'pf'.
- **output** (*str*) – Either 'dash' to generate an interactive plot running in your browser using the dash app, or anything else to save a static plot to the desktop. Defaults to 'dash'.
- **points** (*str*) – Plot only constraint satisfying points ('constraint_satisfying'), constraint violating points ('constraint_violating'), all points ('all'), or no points ('none').
- **height** (*str*) – The height in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **width** (*str*) – The width in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **font** (*str*) – The font that will be used for axis labels and legends. These values are automatically inferred from the name fields of your MOOP object. Any specified font must be available on your computer and available in the appropriate path. Defaults 'auto', which is times new roman on most machines.
- **fontsize** (*str*) – The font size (in points). Defaults to 'auto', which infers the size based on the plot dimensions.
- **background_color** (*str*) – Set the background color for this plot. Defaults to 'auto', which is white with grey axis lines on most systems.
- **screenshot** (*str*) – Set the download mode when saving a screenshot using the "screenshot" button. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **image_export_format** (*str*) – Set the export format when exporting a plot directly image file. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **data_export_format** (*str*) – Set the format for exporting selected data to a file. Defaults to 'csv'. The other option is 'json'.
- **dev_mode** (*bool*) – Run in developer mode, which allows changes to the code to automatically render in the browser. Activating this mode will interfere with some functionalities (such as checkpointing) since it results in multiple calls to the script. This value defaults to False, and should only be adjusted by developers.
- **pop_up** (*bool*) – Automatically pop-up the dash app when called. Defaults to True. The only reason one might want to adjust, is if the environment prevents pop-ups or a non default browser is desired.
- **port** (*str*) – The port through which the Dash app is hosted. Defaults to 'http://127.0.0.1:8050/'.

viz.plot.**parallel_coordinates**(*moop*, *db='pf'*, *output='dash'*, *points='constraint_satisfying'*, *height='auto'*, *width='auto'*, *font='auto'*, *fontsize='auto'*, *background_color='auto'*, *screenshot='svg'*, *image_export_format='svg'*, *data_export_format='csv'*, *dev_mode=False*, *pop_up=True*, *port='http://127.0.0.1:8050/'*)

Create a parallel coordinates plot to visualize the results of a MOOP.

Parameters

- **moop** (*MOOP*) – The MOOP results that you would like to visualize.
- **db** (*str*) – Either 'pf' to plot just the Pareto front, or 'obj' to plot the complete objective database. Defaults to 'pf'.
- **output** (*str*) – Either 'dash' to generate an interactive plot running in your browser using the dash app, or anything else to save a static plot to the desktop. Defaults to 'dash'.
- **points** (*str*) – Plot only constraint satisfying points ('constraint_satisfying'), constraint violating points ('constraint_violating'), all points ('all'), or no points ('none').
- **height** (*str*) – The height in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **width** (*str*) – The width in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **font** (*str*) – The font that will be used for axis labels and legends. These values are automatically inferred from the name fields of your MOOP object. Any specified font must be available on your computer and available in the appropriate path. Defaults 'auto', which is times new roman on most machines.
- **fontsize** (*str*) – The font size (in points). Defaults to 'auto', which infers the size based on the plot dimensions.
- **background_color** (*str*) – Set the background color for this plot. Defaults to 'auto', which is white with grey axis lines on most systems.
- **screenshot** (*str*) – Set the download mode when saving a screenshot using the "screenshot" button. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **image_export_format** (*str*) – Set the export format when exporting a plot directly image file. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **data_export_format** (*str*) – Set the format for exporting selected data to a file. Defaults to 'csv'. The other option is 'json'.
- **dev_mode** (*bool*) – Run in developer mode, which allows changes to the code to automatically render in the browser. Activating this mode will interfere with some functionalities (such as checkpointing) since it results in multiple calls to the script. This value defaults to False, and should only be adjusted by developers.
- **pop_up** (*bool*) – Automatically pop-up the dash app when called. Defaults to True. The only reason one might want to adjust, is if the environment prevents pop-ups or a non default browser is desired.
- **port** (*str*) – The port through which the Dash app is hosted. Defaults to 'http://127.0.0.1:8050/'.

`viz.plot.radar`(*moop*, *db='pf'*, *output='dash'*, *points='constraint_satisfying'*, *height='auto'*, *width='auto'*, *font='auto'*, *fontsize='auto'*, *background_color='auto'*, *screenshot='svg'*, *image_export_format='svg'*, *data_export_format='csv'*, *dev_mode=False*, *pop_up=True*, *port='http://127.0.0.1:8050/'*)

Create a radar plot to visualize the results of a MOOP.

Parameters

- **moop** (*MOOP*) – The MOOP results that you would like to visualize.
- **db** (*str*) – Either 'pf' to plot just the Pareto front, or 'obj' to plot the complete objective database. Defaults to 'pf'.
- **output** (*str*) – Either 'dash' to generate an interactive plot running in your browser using the dash app, or anything else to save a static plot to the desktop. Defaults to 'dash'.
- **points** (*str*) – Plot only constraint satisfying points ('constraint_satisfying'), constraint violating points ('constraint_violating'), all points ('all'), or no points ('none').
- **height** (*str*) – The height in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **width** (*str*) – The width in pixels of the resulting figure. Defaults to 'auto', which matches your screen size.
- **font** (*str*) – The font that will be used for axis labels and legends. These values are automatically inferred from the name fields of your MOOP object. Any specified font must be available on your computer and available in the appropriate path. Defaults 'auto', which is times new roman on most machines.
- **fontsize** (*str*) – The font size (in points). Defaults to 'auto', which infers the size based on the plot dimensions.
- **background_color** (*str*) – Set the background color for this plot. Defaults to 'auto', which is white with grey axis lines on most systems.
- **screenshot** (*str*) – Set the download mode when saving a screenshot using the "screenshot" button. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **image_export_format** (*str*) – Set the export format when exporting a plot directly image file. Defaults to 'svg'. Other available options include: 'html', 'webp', 'jpeg', 'png', 'svg', 'eps', and 'pdf'. Note that the 'eps' option requires the poppler library, which is not included in any of ParMOO's dependency lists.
- **data_export_format** (*str*) – Set the format for exporting selected data to a file. Defaults to 'csv'. The other option is 'json'.
- **dev_mode** (*bool*) – Run in developer mode, which allows changes to the code to automatically render in the browser. Activating this mode will interfere with some functionalities (such as checkpointing) since it results in multiple calls to the script. This value defaults to False, and should only be adjusted by developers.
- **pop_up** (*bool*) – Automatically pop-up the dash app when called. Defaults to True. The only reason one might want to adjust, is if the environment prevents pop-ups or a non default browser is desired.
- **port** (*str*) – The port through which the Dash app is hosted. Defaults to 'http://127.0.0.1:8050/'.

Other Private Classes and Functions

There are several other private submodules, classes, and functions contained in the viz module that are used for creating and hosting the dashboard. These do not need to be referenced by the user, but are detailed below for developers.

Please note that documentation for some of these functions may be incomplete.

This module contains private methods for hosting and receiving callbacks from an interactiv dashboard. This module is intended only for developer use.

Note that some docstrings may be incomplete.

`class viz.dashboard.`**`Dash_App`**(*moop*, *plot_type*, *db*, *points*, *height*, *width*, *font*, *fontsize*, *background_color*, *screenshot*, *image_export_format*, *data_export_format*, *dev_mode*, *pop_up*, *port*)

A class for hosting the dashboard app.

`__init__`(*moop*, *plot_type*, *db*, *points*, *height*, *width*, *font*, *fontsize*, *background_color*, *screenshot*, *image_export_format*, *data_export_format*, *dev_mode*, *pop_up*, *port*)

Constructor for dashboard app.

`generate_graph`()

Documentation incomplete.

`configure`()

Documentation incomplete.

`update_height`()

Documentation incomplete.

`update_width`()

Documentation incomplete.

`update_font`()

Documentation incomplete.

`update_font_size`()

Documentation incomplete.

`update_plot_name`()

Documentation incomplete.

`update_background_color`()

Documentation incomplete.

`update_plot_type`()

Documentation incomplete.

`update_database`()

Documentation incomplete.

`evaluate_height`(*height_value*)

Documentation incomplete.

`evaluate_width`(*width_value*)

Documentation incomplete.

`evaluate_font`(*font_value*)

Documentation incomplete.

evaluate_font_size(*font_size_value*)
  Documentation incomplete.

evaluate_background_color(*background_color_value*)
  Documentation incomplete.

evaluate_plot_name(*plot_name_value*)
  Documentation incomplete.

evaluate_plot_type(*plot_type_value*)
  Documentation incomplete.

evaluate_database(*database_value*)
  Documentation incomplete.

evaluate_dataset_download(*n_clicks*)
  Documentation incomplete.

evaluate_selected_data(*data*, *type*)
  Documentation incomplete.

set_constraint_range(*restyleData*)
  Documentation incomplete.

update_constraint_range(*restyleData*)
  Documentation incomplete.

evaluate_selection_download(*n_clicks*)
  Documentation incomplete.

evaluate_image_export_format(*image_export_format_value*)
  Documentation incomplete.

evaluate_data_export_format(*data_export_format_value*)
  Documentation incomplete.

evaluate_image_download(*n_clicks*)
  Documentation incomplete.

evaluate_customization_options(*action*, *n_clicks*)
  Documentation incomplete.

evaluate_export_options(*action*, *n_clicks*)
  Documentation incomplete.

evaluate_constraint_showr(*value*)
  Documentation incomplete.

generate_graph()
  Documentation incomplete.

configure()
  Documentation incomplete.

update_height()
  Documentation incomplete.

update_width()
    Documentation incomplete.

update_font()
    Documentation incomplete.

update_font_size()
    Documentation incomplete.

update_plot_name()
    Documentation incomplete.

update_background_color()
    Documentation incomplete.

update_plot_type()
    Documentation incomplete.

update_database()
    Documentation incomplete.

evaluate_height(*height_value*)
    Documentation incomplete.

evaluate_width(*width_value*)
    Documentation incomplete.

evaluate_font(*font_value*)
    Documentation incomplete.

evaluate_font_size(*font_size_value*)
    Documentation incomplete.

evaluate_background_color(*background_color_value*)
    Documentation incomplete.

evaluate_plot_name(*plot_name_value*)
    Documentation incomplete.

evaluate_plot_type(*plot_type_value*)
    Documentation incomplete.

evaluate_database(*database_value*)
    Documentation incomplete.

evaluate_dataset_download(*n_clicks*)
    Documentation incomplete.

evaluate_selected_data(*data*, *type*)
    Documentation incomplete.

set_constraint_range(*restyleData*)
    Documentation incomplete.

update_constraint_range(*restyleData*)
    Documentation incomplete.

evaluate_selection_download(*n_clicks*)

Documentation incomplete.

evaluate_image_export_format(*image_export_format_value*)

Documentation incomplete.

evaluate_data_export_format(*data_export_format_value*)

Documentation incomplete.

evaluate_image_download(*n_clicks*)

Documentation incomplete.

evaluate_customization_options(*action*, *n_clicks*)

Documentation incomplete.

evaluate_export_options(*action*, *n_clicks*)

Documentation incomplete.

evaluate_constraint_showr(*value*)

Documentation incomplete.

This module contains tools to generate interactive Plotly-based graphs.

The functions are:

Display data in interactive browser plot

- `generate_scatter(moop)` – Generate interactive scatterplot matrix
- `generate_parallel(moop)` – Generate interactive parallel plot
- `generate_radar(moop)` – Generate interactive radar plot

`viz.graph.`**generate_scatter**(*moop*, *db*, *points*)

Generate a scatterplot or scatterplot matrix.

Parameters

- **moop** (*MOOP*) – An object containing the data to be visualized.
- **db** (*string*) – Filter traces by dataset. * 'pf' - Plot the Pareto Front. * 'obj' - Plot objective data.
- **points** (*string*) – Filter traces by constraint score. * 'constraint_satisfying' - Show only points that

  satisfy every constraint.

  - 'constraint_violating' - Show only points that violate any constraint.
  - 'all' - Plot all points.
  - 'none' - Plot no points.

Returns

A scatterplot or scatterplot matrix displaying traces that fit the filtering criteria.

Return type

plotly.graph_objects.Figure

`viz.graph.`**`generate_parallel`**`(`*moop*, *db*, *points*`)`

Generate a parallel coordinates plot.

Parameters

- **`moop`** (*`MOOP`*) – An object containing the data to be visualized.
- **`db`** (*`string`*) – Filter traces by dataset. * 'pf' - Plot the Pareto Front. * 'obj' - Plot objective data.
- **`points`** (*`string`*) – Filter traces by constraint score. * 'constraint_satisfying' - Show only points that

  satisfy every constraint.

  - 'constraint_violating' - Show only points that violate any constraint.
  - 'all' - Plot all points.
  - 'none' - Plot no points.

Returns
A parallel coordinates plot displaying traces that fit the filtering criteria.

Return type
plotly.graph_objects.Figure

`viz.graph.`**`generate_radar`**`(`*moop*, *db*, *points*`)`

Generate a radar plot.

Parameters

- **`moop`** (*`MOOP`*) – An object containing the data to be visualized.
- **`db`** (*`string`*) – Filter traces by dataset. * 'pf' - Plot the Pareto Front. * 'obj' - Plot objective data.
- **`points`** (*`string`*) – Filter traces by constraint score. * 'constraint_satisfying' - Show only points that

  satisfy every constraint.

  - 'constraint_violating' - Show only points that violate any constraint.
  - 'all' - Plot all points.
  - 'none' - Plot no points.

Returns
A radar plot displaying traces that fit the filtering criteria.

Return type
plotly.graph_objects.Figure

This module contains utilities (helper functions) that are used throughout the viz tool.

`viz.utilities.`**`export_file`**`(`*fig*, *plot_name*, *file_type*`)`

Export image of figure to working directory.

Parameters

- **`fig`** (*`plotly.graph_objects.Figure`*) – The figure to export.
- **`plot_name`** (*`string`*) – Set the filename of the image file.

- **file_type** (*string*) – Set the image file type. - 'html' - Export as .html file. - 'pdf' - Export as .pdf file. - 'svg' - Export as .svg file. - 'eps' - Export as .eps file

  if the poppler dependency is installed.

  - 'jpeg' - Export as .jpeg file.
  - 'png' - Export as .png file.
  - 'webp' - Export as .webp file.

`viz.utilities.`**`set_plot_name`**`(`*db*`)`

Provide a default graph title.

Parameters

db (*string*) – Graph contents inform title. - 'pf' - Set plot name to "Pareto Front" - 'obj' - Set plot name to "Objective Data"

Returns

The default plot name.

Return type

plot_name (string)

`viz.utilities.`**`set_database`**`(`*moop*, *db*, *points*`)`

Choose which points from MOOP object to plot.

Parameters

- **db** (*string*) – Set dataset. - 'pf' - Set Pareto Front as dataset. - 'obj' - Set objective data as dataset.
- **points** (*string*) – Filter traces from dataset by constraint score. - 'constraint_satisfying' - Include only points that

  satisfy every constraint.

  - 'constraint_violating' - Include only points that violate any constraint.
  - 'all' - Include all points in dataset.
  - 'none' - Include no points in dataset.

Returns

A 2D dataframe containing post-filter data from the MOOP.

Return type

df (Pandas dataframe)

`viz.utilities.`**`set_hover_info`**`(`*database*, *i*`)`

Customize information in hover label for trace i.

Parameters

- **database** (*Pandas dataframe*) – A 2D dataframe containing the traces to be graphed.
- **i** (*int*) – An index indicating the row where the trace we're labeling is located.

Returns

An HTML-format string to display when users hover over trace i.

Return type

hover_info (string)

`viz.utilities.`**`check_inputs`**(*db*, *output*, *points*, *height*, *width*, *font*, *fontsize*, *background_color*, *screenshot*, *image_export_format*, *data_export_format*, *dev_mode*, *pop_up*, *port*)

Check keyword inputs to user-facing functions for validity

Parameters

- **db** – The item passed to the 'db' keyword in a user-facing function. If db cannot be cast to a string valued 'pf' or 'obj', a ValueError is raised.
- **output** – The item passed to the 'output' keyword in a user-facing function. If output cannot be cast to a string corresponding to one of the supported output filetypes, a ValueError is raised.
- **points** – The item passed to the 'points' keyword in a user-facing function. If points cannot be cast to a string corresponding to one of the supported contraint filters, a ValueError is raised.
- **height** – The item passed to the 'height' keyword in a user-facing function. If height is not the default string 'auto' or cannot be cast to an int of value greater than one, a ValueError is raised.
- **width** – The item passed to the 'width' keyword in a user-facing function. If width is not the default string 'auto' or cannot be cast to an int of value greater than one, a ValueError is raised.
- **font** – The item passed to the 'font' keyword in a user-facing function. If font cannot be cast to a string, a ValueError is raised.
- **fontsize** – The item passed to the 'fontsize' keyword in a user-facing function. If fontsize is not the default value 'auto' or cannot be cast to an int of value between 1 and 100 inclusive, a ValueError is raised.
- **background_color** – The item passed to the 'background_color' keyword in a user-facing function. If background_color cannot be cast to a string, a ValueError is raised.
- **screenshot** – The item passed to the 'screenshot' keyword in a user-facing function. If screenshot cannot be cast to a string corresponding to one of the supported screenshot filetypes, a ValueError is raised.
- **image_export_format** – The item passed to the 'image_export_format' keyword in a user-facing function. If image_export_format cannot be cast to a string corresponding to one of the supported image_export_format filetypes, a ValueError is raised.
- **data_export_format** – The item passed to the 'data_export_format' keyword in a user-facing function. If data_export_format cannot be cast to a string corresponding to one of the supported data_export_format filetypes, a ValueError is raised.
- **data_export_format** – The item passed to the 'data_export_format' keyword in a user-facing function. If data_export_format cannot be cast to a string corresponding to one of the supported data_export_format filetypes, a ValueError is raised.
- **dev_mode** – The item passed to the 'dev_mode' keyword in a user-facing function. If dev_mode cannot be cast to one of the boolean values True and False, a ValueError is raised.
- **pop_up** – The item passed to the 'pop_up' keyword in a user-facing function. If pop_up cannot be cast to one of the boolean values True and False, a ValueError is raised.
- **port** – The item passed to the 'port' keyword in a user-facing function. If port cannot be cast to a string beginning with 'http', a ValueError is raised.

Raises

- A ValueError if any of the values passed by a user to a keyword in –
- a user-facing function are judged invalid. –

Chapter 3

Tutorials

3.1 Basic Tutorials

This is a collection of all tutorials demonstrating basic ParMOO functionality (collected from throughout the ParMOO User Guide).

3.1.1 Quickstart Demo

This is a basic example (see quickstart.py) of how to build and solve a MOOP with ParMOO, taken from the `Quickstart` guide.

```
import numpy as np
import pandas as pd
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import RandomConstraint
from parmoo.optimizers import LocalGPS

# Fix the random seed for reproducibility
np.random.seed(0)

my_moop = MOOP(LocalGPS)

my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': ["good", "bad"]})

def sim_func(x):
   if x["x2"] == "good":
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
```

(continues on next page)

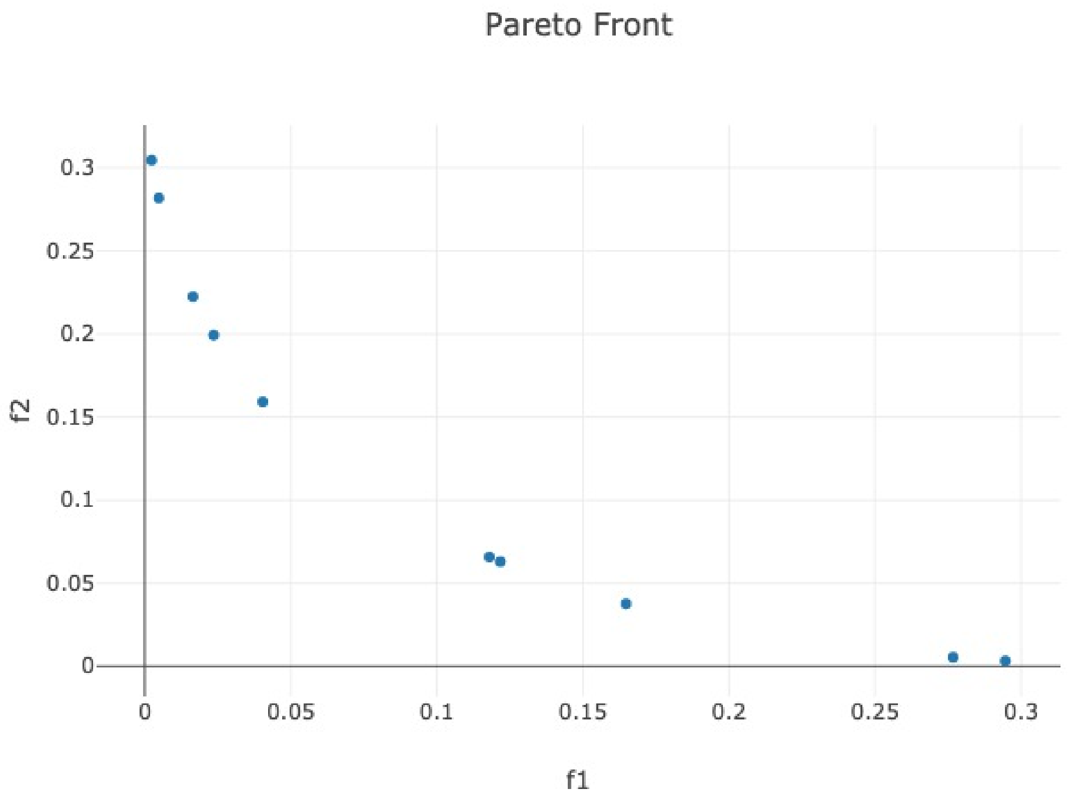
(continued from previous page)

```
    return np.array([99.9, 99.9])

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

def f1(x, s): return s["MySim"][0]
def f2(x, s): return s["MySim"][1]
my_moop.addObjective({'name': "f1", 'obj_func': f1})
my_moop.addObjective({'name': "f2", 'obj_func': f2})

def c1(x, s): return 0.1 - x["x1"]
my_moop.addConstraint({'name': "c1", 'constraint': c1})

for i in range(3):
   my_moop.addAcquisition({'acquisition': RandomConstraint,
                           'hyperparams': {}})

my_moop.solve(5)
results = my_moop.getPF(format="pandas")

# Display solution
print(results)

# Plot results -- must have extra viz dependencies installed
from parmoo.viz import scatter
# The optional arg `output` exports directly to jpg instead of interactive mode
scatter(my_moop, output="jpeg")
```

The above code saves all (approximate) Pareto optimal solutions in the `results` dataframe, and prints the `results` dataframe to the standard output:

```
         x1    x2        f1        f2        c1
0  0.742840  good  0.294675  0.003267 -0.642840
1  0.726092  good  0.276773  0.005462 -0.626092
2  0.605914  good  0.164766  0.037669 -0.505914
3  0.548931  good  0.121753  0.063036 -0.448931
4  0.543499  good  0.117991  0.065793 -0.443499
5  0.401011  good  0.040405  0.159192 -0.301011
6  0.353552  good  0.023578  0.199316 -0.253552
7  0.328402  good  0.016487  0.222404 -0.228402
8  0.269175  good  0.004785  0.281775 -0.169175
9  0.248183  good  0.002322  0.304502 -0.148183
```

And produces the following figure of the Pareto points:


Pareto Front
f2
0.3
0.25
0.2
0.15
0.1
0.05
0
f1
0
0.05
0.1
0.15
0.2
0.25
0.3

3.1.2 Named Output Types

The following named_var_ex.py code demonstrates ParMOO's output datatype when the `MOOP` object is defined using *named* variables.

```python
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

my_moop = MOOP(LocalGPS)

# Define a simulation to use below
def sim_func(x):
    if x["MyCat"] == 0:
        return np.array([(x["MyDes"]) ** 2, (x["MyDes"] - 1.0) ** 2])
    else:
        return np.array([99.9, 99.9])

# Add a design variable, simulation, objective, and constraint.
# Note the 'name' keys for each
my_moop.addDesign({'name': "MyDes",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "MyCat",
                   'des_type': "categorical",
                   'levels': 2})

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'name': "MyObj",
                      'obj_func': lambda x, s: sum(s["MySim"])})

my_moop.addConstraint({'name': "MyCon",
                       'constraint': lambda x, s: 0.1 - x["MyDes"]})

# Extract numpy dtypes for all of this MOOP's inputs/outputs
des_dtype = my_moop.getDesignType()
obj_dtype = my_moop.getObjectiveType()
sim_dtype = my_moop.getSimulationType()

# Display the dtypes as strings
print("Design variable type:   " + str(des_dtype))
print("Simulation output type: " + str(sim_dtype))
print("Objective type:         " + str(obj_dtype))
```

The above code produces the following output.

```
Design variable type:   [('MyDes', '<f8'), ('MyCat', '<i4')]
Simulation output type: [('MySim', '<f8', (2,))]
Objective type:         [('MyObj', '<f8')]
```

3.1.3 Unnamed Output Types

The following unnamed_var_ex.py code demonstrates ParMOO's output datatype when the `MOOP` object is defined using *unnamed* variables.

```
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

# Fix the random seed for reproducibility
np.random.seed(0)

my_moop = MOOP(LocalGPS)

# Define a simulation to use below
def sim_func(x):
    return np.array([(x[0]) ** 2, (x[0] - 1.0) ** 2])

# Add a design variable, simulation, objective, and constraint, w/o name key
my_moop.addDesign({'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})

my_moop.addSimulation({'m': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'obj_func': lambda x, s: sum(s)})

my_moop.addConstraint({'constraint': lambda x, s: 0.1 - x[0]})

# Extract numpy dtypes for all of this MOOP's inputs/outputs
des_dtype = my_moop.getDesignType()
sim_dtype = my_moop.getSimulationType()
obj_dtype = my_moop.getObjectiveType()
const_dtype = my_moop.getConstraintType()

# Display the dtypes as strings
print("Design variable type:   " + str(des_dtype))
print("Simulation output type: " + str(sim_dtype))
print("Objective type:         " + str(obj_dtype))
```

(continues on next page)

(continued from previous page)

```
print("Constraint type:        " + str(const_dtype))
print()

# Add one acquisition and solve with 0 iterations to initialize databases
my_moop.addAcquisition({'acquisition': UniformWeights})
my_moop.solve(0)

# Extract final objective and simulation databases
obj_db = my_moop.getObjectiveData()
sim_db = my_moop.getSimulationData()

# Print the objective database dtypes
print("Objective database keys: " + str([key for key in obj_db.keys()]))
for key in obj_db.keys():
    print("\t'" + key + "'" + " dtype: " + str(obj_db[key].dtype))
    print("\t'" + key + "'" + " shape: " + str(obj_db[key].shape))
print()

# Print the simulation database dtypes
print("Simulation database type: " + str(type(sim_db)))
print("Simulation database length: " + str(len(sim_db)))
for i, dbi in enumerate(sim_db):
    print("\tsim_db[" + str(i) + "] database keys: " +
          str([key for key in dbi.keys()]))
    for key in dbi.keys():
        print("\t\t'" + key + "'" + " dtype: " + str(dbi[key].dtype))
        print("\t\t'" + key + "'" + " shape: " + str(dbi[key].shape))
```

The above code produces the following output.

```
Design variable type:   ('<f8', (1,))
Simulation output type: ('<f8', (2,))
Objective type:         ('<f8', (1,))
Constraint type:        ('<f8', (1,))

Objective database keys: ['x_vals', 'f_vals', 'c_vals']
        'x_vals' dtype: float64
        'x_vals' shape: (20, 1)
        'f_vals' dtype: float64
        'f_vals' shape: (20, 1)
        'c_vals' dtype: float64
        'c_vals' shape: (20, 1)

Simulation database type: <class 'list'>
Simulation database length: 1
        sim_db[0] database keys: ['x_vals', 's_vals']
                'x_vals' dtype: float64
                'x_vals' shape: (20, 1)
                's_vals' dtype: float64
                's_vals' shape: (20, 2)
```

3.1.4 Adding Precomputed Simulation Values

The following precomputed_data.py code demonstrates how to add a precomputed simulation output to ParMOO's internal simulation databases.

```
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

my_moop = MOOP(LocalGPS)

my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': 3})

def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

my_moop.addObjective({'name': "f1", 'obj_func': lambda x, s: s["MySim"][0]})
my_moop.addObjective({'name': "f2", 'obj_func': lambda x, s: s["MySim"][1]})

my_moop.addAcquisition({'acquisition': UniformWeights})

# Precompute one simulation value for demo
des_val = np.zeros(1, dtype=[("x1", float), ("x2", int)])[0]
sim_val = sim_func(des_val)

# Add the precomputed simulation value from above
my_moop.update_sim_db(des_val, sim_val, "MySim")

# Get and display initial database
sim_db = my_moop.getSimulationData()
print(sim_db)
```

The above code produces the following output.

```
{'MySim': array([(0., 0, [0.04, 0.64])],
      dtype=[('x1', '<f8'), ('x2', '<i4'), ('out', '<f8', (2,))])}
```

3.1.5 Logging and Checkpointing

When solving a large or expensive problem, logging and checkpointing are recommended. The following code snippet shows how to solve the same problem from the Quickstart example, but with logging and checkpointing turned on. Then, another MOOP is created by reloading from the checkpoint file, and run for 1 extra iteration, in order to demonstrate how to reload from a saved checkpoint file.

Note how the lambda functions from Quickstart example have been explicitly defined below. This is because ParMOO reloads functions by name, which means that it does not support checkpointing for lambda functions and any named function must be defined with the same name in the global scope during reloading.

```
import numpy as np
from parmoo import MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS
import logging

# Fix the random seed for reproducibility
np.random.seed(0)

# Create a new MOOP
my_moop = MOOP(LocalGPS)

# Add 1 continuous and 1 categorical design variable
my_moop.addDesign({'name': "x1",
                   'des_type': "continuous",
                   'lb': 0.0, 'ub': 1.0})
my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                   'levels': 3})

# Create a simulation function
def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])

# Add the simulation function to the MOOP
my_moop.addSimulation({'name': "MySim",
                       'm': 2,
                       'sim_func': sim_func,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 20}})

# Define the 2 objectives as named Python functions
def obj1(x, s): return s["MySim"][0]
def obj2(x, s): return s["MySim"][1]

# Define the constraint as a function
def const(x, s): return 0.1 - x["x1"]
```

(continues on next page)

(continued from previous page)

```
# Add 2 objectives
my_moop.addObjective({'name': "f1", 'obj_func': obj1})
my_moop.addObjective({'name': "f2", 'obj_func': obj2})

# Add 1 constraint
my_moop.addConstraint({'name': "c1", 'constraint': const})

# Add 3 acquisition functions (generates batches of size 3)
for i in range(3):
   my_moop.addAcquisition({'acquisition': UniformWeights,
                           'hyperparams': {}})

# Turn on logging with timestamps
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)-8s %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S')

# Use checkpointing without saving a separate data file (in "parmoo.moop" file)
my_moop.setCheckpoint(True, checkpoint_data=False, filename="parmoo")

# Solve the problem with 4 iterations
my_moop.solve(4)

# Create a new MOOP object and reload the MOOP from parmoo.moop file
new_moop = MOOP(LocalGPS)
new_moop.load("parmoo")

# Do another iteration
new_moop.solve(5)

# Display the solution
results = new_moop.getPF()
print(results, "\n dtype=" + str(results.dtype))
```

The above checkpointing.py code produces the following output.

```
[(0.79013666, 0, 3.48261275e-01, 9.72855201e-05, -0.69013666)
 (0.7895263 , 0, 3.47541259e-01, 1.09698380e-04, -0.6895263 )
 (0.73488156, 0, 2.86098283e-01, 4.24041125e-03, -0.63488156)
 (0.70656124, 0, 2.56604287e-01, 8.73080244e-03, -0.60656124)
 (0.68409101, 0, 2.34344111e-01, 1.34348928e-02, -0.58409101)
 (0.67237225, 0, 2.23135544e-01, 1.62888423e-02, -0.57237225)
 (0.5784217 , 0, 1.43202981e-01, 4.90969442e-02, -0.4784217 )
 (0.53031761, 0, 1.09109723e-01, 7.27285920e-02, -0.43031761)
 (0.51322778, 0, 9.81116425e-02, 8.22383058e-02, -0.41322778)
 (0.49723345, 0, 8.83477213e-02, 9.16675863e-02, -0.39723345)
 (0.44987019, 0, 6.24351129e-02, 1.22590882e-01, -0.34987019)
 (0.40591372, 0, 4.24004606e-02, 1.55303995e-01, -0.30591372)
 (0.39028872, 0, 3.62097978e-02, 1.67863331e-01, -0.29028872)
 (0.38995793, 0, 3.60840145e-02, 1.68134501e-01, -0.28995793)
 (0.38287784, 0, 3.34443041e-02, 1.73990897e-01, -0.28287784)
```

(continues on next page)

(continued from previous page)

```
(0.25435646, 0, 2.95462529e-03, 2.97726867e-01, -0.15435646)
(0.20137796, 0, 1.89878434e-06, 3.58348342e-01, -0.10137796)]
dtype=[('x1', '<f8'), ('x2', '<i4'), ('f1', '<f8'), ('f2', '<f8'), ('c1', '<f8')]
```

3.1.6 Solving a MOOP with Derivative-Based Solvers

This example shows how to minimize two conflicting quadratic functions of three variables (named `x1`, `x2`, and `x3`), under the constraint that an additional categorical variable (`x4`) must be fixed in class `0` *, using the derivative-based solver `LBFGSB`.

* No, this constraint does not really affect the solution; it is just here to demonstrate how constraints/categorical variables are handled by derivative-based solvers. ParMOO does not use derivative information associated with any categorical variables, so the derivative w.r.t. `x4` can be set to any value, without affecting the outcome.

```python
import numpy as np
from parmoo import MOOP
from parmoo.acquisitions import RandomConstraint, FixedWeights
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.optimizers import LBFGSB

# Fix the random seed for reproducibility
np.random.seed(0)

# Create a new MOOP with a derivative-based solver
my_moop = MOOP(LBFGSB, hyperparams={})

# Add 3 continuous variables named x1, x2, x3
for i in range(3):
    my_moop.addDesign({'name': "x" + str(i+1),
                       'des_type': "continuous",
                       'lb': 0.0,
                       'ub': 1.0,
                       'des_tol': 1.0e-8})
# Add one categorical variable named x4
my_moop.addDesign({'name': "x4",
                   'des_type': "categorical",
                   'levels': 3})

def quad_sim(x):
    """ A quadratic simulation function with 2 outputs.

    Returns:
        np.ndarray: simulation value (S) with 2 outputs
          * S_1(x) = <x, x>
          * S_2(x) = <x-1, x-1>

    """

    return np.array([x["x1"] ** 2 + x["x2"] ** 2 + x["x3"] ** 2,
```

(continues on next page)

(continued from previous page)

```python
                     (x["x1"] - 1.0) ** 2 + (x["x2"] - 1.0) ** 2 +
                     (x["x3"] - 1.0) ** 2])

# Add the quadratic simulation to the problem
# Use a 10 point LH search for ex design and a Gaussian RBF surrogate model
my_moop.addSimulation({'name': "f_conv",
                       'm': 2,
                       'sim_func': quad_sim,
                       'search': LatinHypercube,
                       'surrogate': GaussRBF,
                       'hyperparams': {'search_budget': 10}})

def obj_f1(x, sim, der=0):
    """ Minimize the first output from 'f_conv' """

    if der == 0:
        return sim['f_conv'][0]
    elif der == 1:
        return np.zeros(1, dtype=x.dtype)[0]
    elif der == 2:
        result = np.zeros(1, dtype=sim.dtype)[0]
        result['f_conv'][0] = 1.0
        return result

def obj_f2(x, sim, der=0):
    """ Minimize the second output from 'f_conv' """

    if der == 0:
        return sim['f_conv'][1]
    elif der == 1:
        return np.zeros(1, dtype=x.dtype)[0]
    elif der == 2:
        result = np.zeros(1, dtype=sim.dtype)[0]
        result['f_conv'][1] = 1.0
        return result

# Minimize each of the 2 outputs from the quadratic simulation
my_moop.addObjective({'name': "f1",
                      'obj_func': obj_f1})
my_moop.addObjective({'name': "f2",
                      'obj_func': obj_f2})

def const_x4(x, sim, der=0):
    """ Constrain x["x4"] = 0 """

    if der == 0:
        return 0.0 if (x["x4"] == 0) else 1.0
    elif der == 1:
        # No derivatives for categorical design var, just return all zeros
        return np.zeros(1, dtype=x.dtype)[0]
    elif der == 2:
        return np.zeros(1, dtype=sim.dtype)[0]
```

(continues on next page)

(continued from previous page)

```
# Add the single constraint to the problem
my_moop.addConstraint({'name': "c_x4",
                       'constraint': const_x4})

# Add 2 different acquisition functions to the problem
my_moop.addAcquisition({'acquisition': RandomConstraint})
my_moop.addAcquisition({'acquisition': FixedWeights,
                        # Fixed weight with equal weight on both objectives
                        'hyperparams': {'weights': np.array([0.5, 0.5])}})

# Turn on checkpointing -- creates the files parmoo.moop and parmoo.surrogate.1
my_moop.setCheckpoint(True, checkpoint_data=False, filename="parmoo")

# Turn on logging
import logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)-8s %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S')

# Solve the problem
my_moop.solve(5)

# Get and print full simulation database
sim_db = my_moop.getSimulationData(format="pandas")
print("Simulation data:")
for key in sim_db.keys():
    print(f"\t{key}:")
    print(sim_db[key])

# Get and print results
soln = my_moop.getPF(format="pandas")
print("\n\n")
print("Solution points:")
print(soln)
```

The above advanced_ex.py code produces the following output.

```
Simulation data:
        f_conv:
          x1        x2        x3  x4     out_0     out_1
0   0.164589  0.071519  0.256804   0  0.098153  2.112328
1   0.279173  0.761210  0.446148   2  0.856425  0.883365
2   0.054881  0.143759  0.645615   1  0.440497  1.751987
3   0.761764  0.514335  0.869763   1  1.601312  0.309588
4   0.563992  0.252889  0.383262   1  0.528930  1.128643
5   0.626456  0.302022  0.761693   2  1.063841  0.683499
6   0.835951  0.921038  0.912893   1  2.380498  0.040735
7   0.308713  0.677423  0.189177   0  0.589994  1.239367
8   0.497862  0.843703  0.060276   0  0.963335  1.159652
9   0.967064  0.479916  0.594467   0  1.518922  0.436029
10  0.198616  0.279706  0.253141   0  0.181764  1.718838
```

(continues on next page)

(continued from previous page)

```
11  0.104311  0.713583  0.360565   0  0.650089  1.293170
12  0.990617  0.496152  0.621227   2  1.613413  0.397420
13  0.322478  0.475506  0.173748   1  0.360287  1.416822
14  0.143465  0.160993  0.217433   0  0.093778  2.049997
15  0.317939  0.476173  0.171654   0  0.357290  1.425761
16  0.711117  0.366636  0.640507   0  1.050358  0.613839
17  0.373639  0.410939  0.465101   0  0.524796  1.025438
18  0.193532  0.456200  0.200312   0  0.285698  1.585610
19  0.448389  0.368552  0.567048   0  0.658426  0.890449



Solution points:
         x1        x2        x3  x4        f1        f2  c_x4
0  0.967064  0.479916  0.594467   0  1.518922  0.436029   0.0
1  0.711117  0.366636  0.640507   0  1.050358  0.613839   0.0
2  0.448389  0.368552  0.567048   0  0.658426  0.890449   0.0
3  0.373639  0.410939  0.465101   0  0.524796  1.025438   0.0
4  0.317939  0.476173  0.171654   0  0.357290  1.425761   0.0
5  0.193532  0.456200  0.200312   0  0.285698  1.585610   0.0
6  0.198616  0.279706  0.253141   0  0.181764  1.718838   0.0
7  0.143465  0.160993  0.217433   0  0.093778  2.049997   0.0
```

Note how in the full simulation database several of the design points violate the constraint (`x4 != 0`). But in the solution, the constraint is always satisfied.

3.2 libEnsemble Tutorial

The following libe_basic_ex.py code is an example of basic ParMOO + libEnsemble usage from the `Extras and Plugins` section of the User Guide.

```python
import numpy as np
from parmoo.extras.libe import libE_MOOP
from parmoo.searches import LatinHypercube
from parmoo.surrogates import GaussRBF
from parmoo.acquisitions import UniformWeights
from parmoo.optimizers import LocalGPS

# When running with MPI, we need to keep track of which thread is the manager
# using libensemble.tools.parse_args()
from libensemble.tools import parse_args
_, is_manager, _, _ = parse_args()

# All functions are defined below.

def sim_func(x):
   if x["x2"] == 0:
      return np.array([(x["x1"] - 0.2) ** 2, (x["x1"] - 0.8) ** 2])
   else:
      return np.array([99.9, 99.9])
```

(continues on next page)

(continued from previous page)

```python
def obj_f1(x, s):
    return s["MySim"][0]

def obj_f2(x, s):
    return s["MySim"][1]

def const_c1(x, s):
    return 0.1 - x["x1"]

# When using libEnsemble with Python MP, the "solve" command must be enclosed
# in an "if __name__ == '__main__':" block, as shown below
if __name__ == "__main__":
    # Fix the random seed for reproducibility
    np.random.seed(0)

    # Create a libE_MOOP
    my_moop = libE_MOOP(LocalGPS)

    # Add 2 design variables (one continuous and one categorical)
    my_moop.addDesign({'name': "x1",
                       'des_type': "continuous",
                       'lb': 0.0, 'ub': 1.0})
    my_moop.addDesign({'name': "x2", 'des_type': "categorical",
                       'levels': 3})

    # Add the simulation (note the budget of 20 sim evals during search phase)
    my_moop.addSimulation({'name': "MySim",
                           'm': 2,
                           'sim_func': sim_func,
                           'search': LatinHypercube,
                           'surrogate': GaussRBF,
                           'hyperparams': {'search_budget': 20}})

    # Add the objectives
    my_moop.addObjective({'name': "f1", 'obj_func': obj_f1})
    my_moop.addObjective({'name': "f2", 'obj_func': obj_f2})

    # Add the constraint
    my_moop.addConstraint({'name': "c1", 'constraint': const_c1})

    # Add 3 acquisition functions
    for i in range(3):
       my_moop.addAcquisition({'acquisition': UniformWeights,
                               'hyperparams': {}})

    # Turn on checkpointing -- creates files parmoo.moop & parmoo.surrogate.1
    my_moop.setCheckpoint(True, checkpoint_data=False, filename="parmoo")

    # Use sim_max = 30 to perform just 30 simulations
    my_moop.solve(sim_max=30)
```

(continues on next page)

(continued from previous page)

```
    # Display the solution -- this "if" clause is needed when running with MPI
    if is_manager:
        results = my_moop.getPF(format="pandas")
        print(results)
```

You can run the above script with MPI

```
mpirun -np N python3 libe_basic_ex.py
```

or with Python's built-in multiprocessing module.

```
python3 libe_basic_ex.py --comms local --nworkers N
```

The resulting output is shown below.

```
         x1  x2        f1        f2        c1
0  0.742825   0  0.294659  0.003269 -0.642825
1  0.680283   0  0.230672  0.014332 -0.580283
2  0.616501   0  0.173473  0.033672 -0.516501
3  0.580369   0  0.144680  0.048238 -0.480369
4  0.555222   0  0.126183  0.059916 -0.455222
5  0.518980   0  0.101749  0.078972 -0.418980
6  0.475477   0  0.075888  0.105315 -0.375477
7  0.302503   0  0.010507  0.247503 -0.202503
8  0.201285   0  0.000002  0.358460 -0.101285
```

Chapter 4

Developer's Guide

4.1 Contributing to ParMOO

Contributions of source code, documentations, and fixes are happily accepted via GitHub pull request to

https://github.com/parmoo/parmoo/tree/develop

If you are planning a contribution, reporting bugs, or suggesting features, we encourage you to discuss the concept by opening a github issue at

https://github.com/parmoo/parmoo/issues

or by emailing `parmoo@mcs.anl.gov` and interacting with us to ensure that your effort is well-directed.

4.1.1 Contribution Process

ParMOO uses the Gitflow model. Contributors should typically branch from, and make pull requests to, the `develop` branch. The `main` branch is used only for releases. Pull requests may be made from a fork, for those without repository write access.

Issues can be raised at

https://github.com/parmoo/parmoo/issues

Issues may include reporting bugs or suggested features.

By convention, user branch names should have a `<type>/<name>` format, where example types are `feature`, `bugfix`, `testing`, `docs`, and `experimental`. Administrators may take a `hotfix` branch from the main, which will be merged into `main` (as a patch) and `develop`. Administrators may also take a `release` branch off `develop` and then merge this branch into `main` and `develop` for a release.

When a branch closes a related issue, the pull request message should include the phrase "Closes #N," where N is the issue number.

New features should be accompanied by at least one test case.

All pull requests to `develop` or `main` must be reviewed by at least one administrator.

4.1.2 Developer's Certificate

ParMOO is distributed under a 3-clause BSD license (see LICENSE). The act of submitting a pull request or patch will be understood as an affirmation of the following:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

4.2 Release Process

A release can be undertaken only by a project administrator. A project administrator should have an administrator role on the ParMOO GitHub, PyPI, and readthedocs pages.

4.2.1 Before release

- A release branch should be taken off `develop` (or `develop` pulls controlled).
- Release notes for this version are added to the `CHANGELOG.rst` file.
- Version number is updated wherever it appears and `+dev` suffix is removed (in `parmoo/version.py`, `README.rst`, and `docs/refs.rst`).
- Check `README.rst` *Citing ParMOO* and `docs/refs.rst` for correctness.
- `setup.py` and `parmoo/__init__.py` are checked to ensure all information is up to date.
- `MANIFEST.in` is checked. Locally, try out `python setup.py sdist` and check created tarball contains correct files and directories for PyPI package.
- Check that `parmoo` requirements (in `REQUIREMENTS.txt`) are compatible with `readthedocs.io` (in `.readthedocs.yml`)

- Tests are run with source to be released (this may iterate):
  - On-line CI (GitHub Actions) tests must pass.
  - Documentation must build and display correctly wherever hosted (currently readthedocs.org).
- Pull request from either the develop or release branch to main requesting one or more reviewers (including at least one other administrator).
- Reviewer will check that all tests have passed and will then approve merge.

4.2.2 During release

An administrator will take the following steps.

- Merge the pull request into main.
- Once CI tests have passed on main:
  - A GitHub release will be taken from the main
  - A tarball (source distribution) will be uploaded to PyPI (should be done via `twine` by an admin using PyPI-API-token authentication)
- If the merge was made from a release branch (instead of develop), merge this branch into develop.
- Create a new commit on develop that appends `+dev` to the version number (wherever it appears).

4.2.3 After release

- Ensure all relevant GitHub issues are closed.
- Check that the conda-forge package has tracked latest release and update dependency list if needed – an admin will need to approve the automatically generated PR on https://github.com/conda-forge/parmoo-feedstock

4.3 Release Notes

Below are the release notes for ParMOO.

May reference issues on: https://github.com/parmoo/parmoo/issues

4.3.1 Release 0.3.1

Date
Sep 25, 2023

Bug-fixes and minor restructuring for future releases.

Fixed several serious bugs/limitations:

- Introduced in v0.3.0: when generating batches, a bug was introduced into the lines of code that filter out duplicate candidates, resulting in significantly decreased performance but no errors being raised
- Allow for ParMOO to still generate target points for the `AcqusitionFunction`, even when there are no feasible points in the database
- Increase the number of characters allowed in a name when working with libEnsemble from 10 to 40 characters

- Broke the `MOOP.iterate()` method apart into 2 functions (`iterate` and `filterBatch`), which makes the code more maintainable and allows for future improvements to the `libE_MOOP` parallelism
- Updated deprecated keys in `.readthedocs.yml` config file

4.3.2 Release 0.3.0

Date
Jul 6, 2023

Significant structural changes for long-term support of future solvers, bug-fixes, and significant improvements to documentation.

Major Changes:

- `surrogates.GaussRBF` and `surrogates.LocalGaussRBF` now calculate model-form uncertainties
- structural changes to `MOOP` class to support propagation of uncertainty information
- added `EI_RandomConstraint` acquisition, which can be used to implement Bayesian optimization – note that for large budgets, this is not currently recommended due to computational expense of numerical integration
- updated `LocalGPS` to use trust regions, when provided, and perform multiple restarts
- `SurrogateOptimizer` class now has access to more information about the objective, including raw simulation outputs, in order to support more diverse structure-exploiting solvers
- Added additional stopping criteria to both `MOOP.solve()` and `libE_MOOP.solve()` – all stopping criteria are now optional (although at least one must be specified) but they are ordered such that calling `MOOP.solve(k)`, where `k` is a positional input, will pass to the `iter_max` criteria and produce the same behavior as before – closes #18

API Changes:

- In most cases, none. However, it is possible that if users were previously passing arguments to the `MOOP.solve()` method explicitly, then the name of the first positional argument has changed: `budget` -> `max_iters`
- For users implementing their own `searches`, `surrogates`, `optimizers`, or `acquisitions`, several classes in the `structs` module have been updated to support the present restructuring of the `MOOP` class

Docs:

- Updated Quickstart guide and README to demonstrate recommended inputs and settings for ParMOO – this includes no more `lambda` functions, which closes #50
- Added a FAQ page with additional usage details and responses to frequent questions – the answers in which close #61
- Added a new tutorial on how to perform high-dimensional multiobjective optimization on a limited budget with ParMOO
- Changed examples and documentation to use and discuss pandas dataframes, which generally produce more legible outputs
- Updated `libE_MOOP` example to demonstrate how to retrieve data in a way that is threadsafe for both Python MP and MPI usage

Requirements:

- We now require scipy v1.10 or newer, due to usage of qmc integration tools

- At the time of this release, libEnsemble is using a deprecated version of Pydantic – for this release only we have fixed the requirement on libEnsemble to v0.9.2, but we will relax this requirement in the future once they have patched the issue

Bug-fixes:

- Fixed an issue where in rare cases, problems with too many categorical variables could produce unexpected batch sizes
- Errors in definition of test problems: DTLZ5, 6, and 7 (new implementations have been confirmed against `pymoo`)
- Fixed an issue which occasionally caused the `libE_MOOP` class to error out during post-run cleanup when used with MPI
- Patched an issue with `format="pandas"` option for `MOOP.getSimulationData()` class and added a similar option to all `libE_MOOP` "getter" functions

Minor changes:

- Fixed typos in docs/doc-strings
- Updated styles to comply with new `flake8` recommendations
- New unit tests added
- Added warnings when ParMOO is run with bad budget settings

4.3.3 Release 0.2.2

Date
Apr 25, 2023

Hot-fix for a minor issue in the plotting library without workaround.

- Resolves #58

4.3.4 Release 0.2.1

Date
Apr 10, 2023

Minor performance improvements, maintenance, and restructuring of test cases.

- Both Gaussian RBF surrogates in `parmoo/surrogates/gaussian_proc.py` now use the current mean of the response values as the prior instead of the zero function. This greatly improves convergence rates in practice, especially for our structure-exploiting methods.
- Using an old version of `plotly/dash` for now because of a dash issue described in plotly/dash#2460
- Added additional tests to check gradient calculations of `GaussRBF` surrogates.
- Added whitespace to pass new `flake8` standards.
- Added year to JOSS publication in several places
- Added "et al." to our docs configuration file after author names, to credit additional contributors in our documentation.

4.3.5 Release 0.2.0

Date
Feb 2, 2023

Official release corresponding to accepted JOSS article.

- Added support for a wider variety of design variables (including integer types), as well as support for "custom" design variables that use user-provided custom embedders/extractors Documentation on design variables has been expanded accordingly. Although design variables are still specified through dicts not classes, this addresses and therefore closes the primary issue raised in parmoo/parmoo#28
- Updated `extras/libe.py` corresponding to interface changes made in libEnsemble Release 0.8.0. This also addresses the issues on MacOS, referenced in parmoo/parmoo#34
- Added a post-run visualization library and corresponding documentation, closing issue parmoo/parmoo#27
- Allow solvers to start from an initial point that is infeasible, so that problems with relaxable constraints and a very small feasible set can still be solved
- Various style changes and additional usage environments requested by JOSS reviewers openjournals/joss-reviews#4468 including parmoo/parmoo#32
- Added support for multistarting optimization solvers when solving surrogate problems. This is particularly important for the global `GaussRBF` surrogate
- Fixed an issue in how model improvement points are calculated, as implemented in the `surrogate.improve` method for each GaussRBF variation in `surrogates/gaussian_proc.py`, which was created when adding support for custom design variables
- The default design tolerance for continuous variables now depends upon the value of `ub - lb`

Note:

- Dropped support for Python 3.6, due to changes to GitHub Actions documented on actions/setup-python#544

Known issues:

- The visualization library uses advanced plotly/dash features, which may not support the chrome browser, as described in parmoo/parmoo#37

4.3.6 Release 0.1.0

Date
May 10, 2022

Initial release.

Known issues and desired features will be raised on GitHub post-release.

Known issues:

- update unit tests to use sim/obj/const libraries
- restructure test suite, unit tests are currently not usable as additional documentation
- `solve()` method(s) should support additional stopping criteria
- allow for maximizing objectives and constraint lower bounds without "hacky" solution (negating values)
- missing functions from DTLZ libraries
- `README.md` needs a code coverage badge

Desired features:

- update, test, and merge-in MDML interface
- allow user to choose whether or not to use named variables via `useNames` method, or similar
- add a funcx simulation interface, using libEnsemble release 0.9
- add predicter interface and standalone module
- a GUI interface for creating MOOPs
- static visualization tools for plotting results (from `MOOP.getPF()` method)
- a visualization dashboard for viewing progress interactively
- design variable types should be a class, with embed/extract methods that can be called by `MOOP.__embed__()` and `MOOP.__extract__()`

Chapter 5

Appendices

Python Module Index

Index