The Probabilistic Design and OPTimization Framework (P-DOPT)

Andrea Spinelli[*]
Timoleon Kipouros[†]
Centre for Propulsion and Thermal Power Engineering
School of Aerospace, Transport and Manufacturing
Cranfield University
Vers. 0.4

August 14, 2022

[*] email: andrea.spinelli@cranfield.ac.uk
[†] email: t.kipouros@cranfield.ac.uk

Abstract

This document describes the capabilities of P-DOPT and provides the user guide for the software. P-DOPT is a design exploration tool developed at Cranfield University. Its goal is to allow the designer to identify a family of design points with the minimum number of assumptions over the system(s) under design.

Contents

1 Introduction **3**

1.1 General Description . . . 3

1.2 Installation Requirements . . . 3

1.3 Programme Flowchart . . . 4

1.4 Exploration Phase . . . 4

1.5 Search Phase . . . 6

2 Definition of a P-DOPT Test Case **7**

2.1 The `input.csv` file . . . 7

2.2 The `responses.csv` file . . . 8

2.3 The evaluation function . . . 8

2.4 Layout of the runfile . . . 9

3 Example Tutorial **12**

3.1 Problem Definition . . . 12

3.2 Breakdown of the problem into P-DOPT components . . . 13

3.3 Definition of the .csv Files . . . 13

3.4 Definition of the Evaluation Function . . . 14

3.5 Definition of the Runfile . . . 15

3.6 Running the Case and Post-processing the results . . . 17

4 API Reference **21**

4.1 Data Structures (`pdopt.data`) . . . 21

4.2 Exploration Phase (`pdopt.exploration`) . . . 31

4.3 Search phase `pdopt.optimisation` . . . 35

4.4 Miscellanous tools `pdopt.tools` . . . 36

A Example Input Files **37**

A.1 `input.csv` . . . 37

A.2 `response.csv` . . . 37

A.3 `energy_management_experiment.py` . . . 37

1 Introduction

1.1 General Description

The Probabilistic Design and OPTimisation (P-DOPT) programme is a design exploration tool developed at Cranfield University. Its goals it to allow the designer to identify a family of design points with the minimum number of assumptions over the system under design.

The user provides a numerical model of its design, with design parameters are inputs and quantities of interests (QoI) as output. These QoI represent requirements the user desires to obtain from its design, either as constrained quantities or performance responses to be minimised/maximised.

For instance, let's analyse the problem of designing a wooden pencil. The design parameters of a wooden pencil are its length, the thickness of the shell, the thickness of the lead and the mixture of graphite/kaolin. QoIs are its weight, its cost, and its average use life. Requirements defined over these QoIs can be minimum cost and weight as possible, with a use life greater than a specified amount. While minimisation/maximisation objectives don't have any bounds, often top-level requirements specify a "minimum expectation" from these objectives. In the case of the pencil design problem, if the average price of a pencil is 0.10 USD per item, it is desired to at least find a design which can match the market "state of the art" and possibly improve it.

With the design problem framed this way, PDOPT allows to explore the different combinations of input parameters, defined as areas of the design space or set, and assign a probability of how capable it is to satisfy that set of requirements. This step is called **exploration phase** within PDOPT.

Then, the code automatically proceeds to find the optimal design points within the sets that have an high probability of satisfying the requirements. The user can also visualise this step before going through the optimisation step and get a sense of the interaction between parameters and QoI (**search phase**). An additional functionality is to check the response of the design space and its probability to different requirement levels. This would help the user to identify areas of the design space that would be less sensible to operational or market changes down the conceptual design phase.

More details on the functionality of this aspect of PDOPT can be found in literature [9].

1.2 Installation Requirements

PDOPT is a Python3 library which can be downloaded from the Github repository. As such it requires the following packages to be installed (through `pip`):

- `numpy`
- `scipy`
- `pandas`
- `scikit-learn`
- `pymoo`
- `multiprocess - (N.B. To be replaced with a more stable library)`

- `tqdm`

An install script is provided to simply this process. Use the following command in the source folder: `python setup.py`

1.3 Programme Flowchart

The Set-Based Design space exploration approach is based on previous work by Georgiades [4], which employs a two-step process to explore and refine the design space with a convergent approach.

This method is improved by replacing the feasibility and desirability rules with a statistical response approach, which seeks to estimate the probability to satisfy the optimisation constraints and, if known, to bound the objectives. This second type of constraint is a "soft constraint" defined as a minimum desired outcome from the optimisation. For instance, if the designer desires the model to have no more than a value f for the objective to be minimised F, the framework allows us to introduce the constraint $F(x) < f$ to discard the areas of the design space that, even if minimised, will not meet this desired objective. This approach allows us to replace the desirability rules used previously [8]. Figure 1.1 presents the flowchart of the improved methodology.

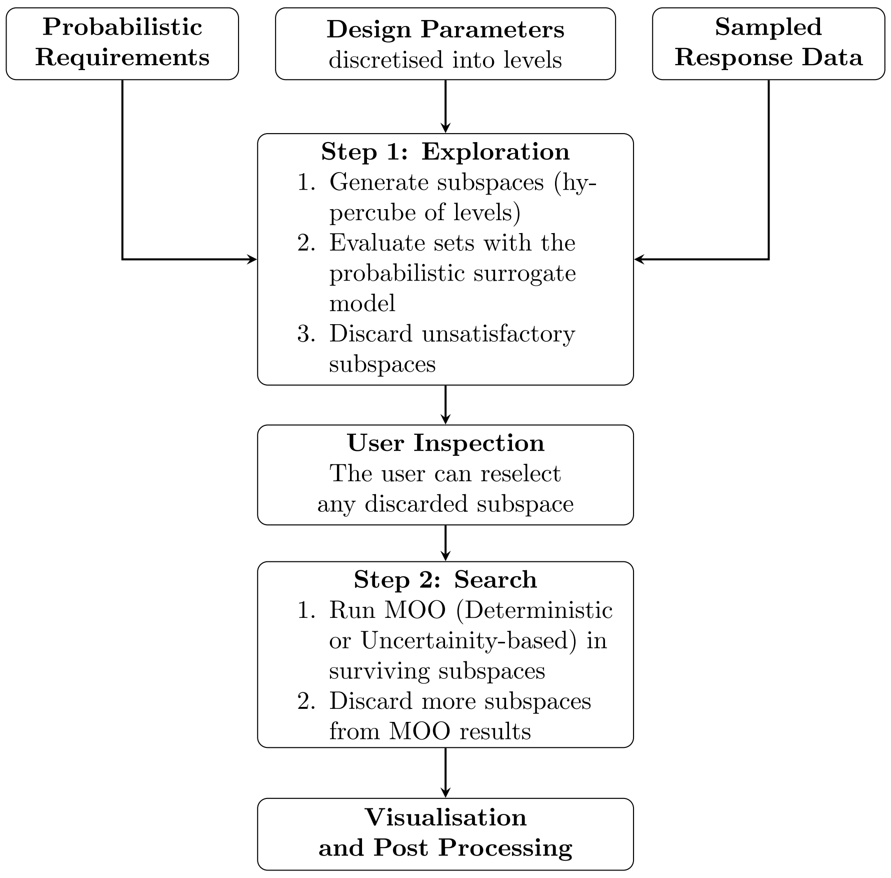

Probabilistic Requirements
Design Parameters discretised into levels
Sampled Response Data
Step 1: Exploration
1. Generate subspaces (hypercube of levels)
2. Evaluate sets with the probabilistic surrogate model
3. Discard unsatisfactory subspaces
User Inspection
The user can reselect any discarded subspace
Step 2: Search
1. Run MOO (Deterministic or Uncertainity-based) in surviving subspaces
2. Discard more subspaces from MOO results
Visualisation and Post Processing

Figure 1.1: Methodology Flowchart.

1.4 Exploration Phase

The goal of the Exploration phase is to survey the entire design space and eliminate those areas that are not suitable for the requirements set by the user. This allows to strike a balance between computational cost of optimisation with the principles of Set-Based Design.

Unlike previous implementations of this methodology [4], the requirements that determine feasibility and desirability of a design set are expressed as inequalities over a model response with a probability attached to them. Therefore the user does not have to assume any rules on the parameters in order to evaluate which sets are desirable or not. Furthermore, these constraints are evaluated not in a deterministic sense, but in a probabilistic way to account for the fuzzyness of the constraint boundary. The PDOPT exploration phase has two types of constraints: hard constraints and soft constraints. The former corresponds to the optimisation constraints, while the latter are additional constraints applied to the model responses selected as objectives. For instance, say the output A of the model has to be minimised. Then if the designer knows which value they want at least to achieve, that is finding those areas of the design space where $A \leq k_A$ where k_A is an arbitrary value, a soft constraint can be added in the exploration phase to at least remove those sets which, despite being *feasible*, are not *desirable*. Indeed the desirability filtering which in ADOPT was implemented with heuristic rules, here is replaced with soft constraints.

Probability theory is used for evaluating these inequalities by first training a probabilistic surrogate model of the design space, and then querying each set with a sufficient number of samples to establish how likely is each requirement satisfied in that set. The user can control the tightness of the probabilistic constraints by adjusting the minimum satisfaction probability per sample ($\overline{P_A}$ in Eq. 1.1), which is used to count how many samples satisfy the constraint. The left hand side of the inequality is obtained using the mean and standard deviation calculated at the sampled point using the corresponding Gaussian Process and the Gaussian cumulative distribution function as shown in Eq. 1.2.

$$P(A < k_A) \geq \overline{P_A} \tag{1.1}$$

$$P^k(A < k_A) = \Phi\left(\frac{k_A - \mu_A^k}{\sigma_A^k}\right) \tag{1.2}$$

The samples that satisfy the constraint are counted and divided by the total number of samples, giving an empirical conditional probability of how likely is the set to satisfy that constraint. If there are multiple constraints, then these probabilities are combined by assuming conditional independence. Figure 1.2 shows this process in detail. The data used for training the Gaussian Processes (i.e. Statistical Surrogate Model) is taken either by sampling the full model or from experimental data provided by the user.

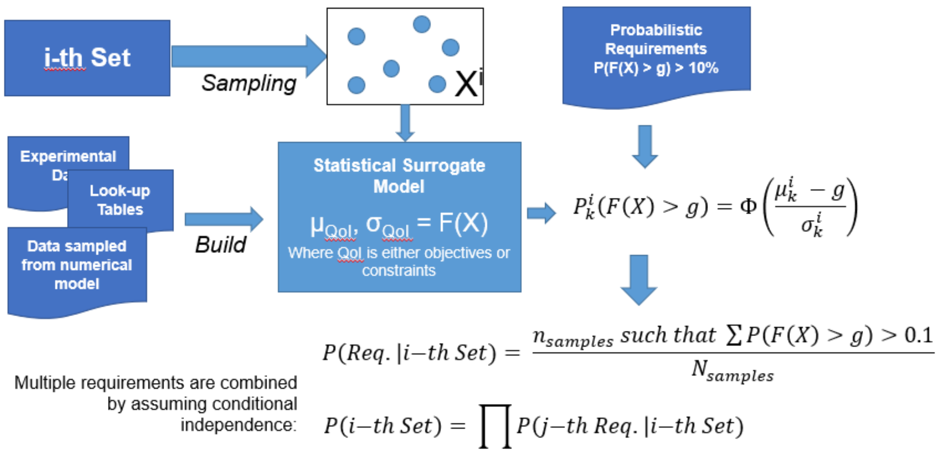

i-th Set
Sampling
X^i
Probabilistic Requirements P(F(X) > g) > 10%
Experimental Da
Look-up Tables
Data sampled from numerical model
Build
Statistical Surrogate Model
$\mu_{QoI}, \sigma_{QoI} = F(X)$
Where QoI is either objectives or constraints
$P_k^i(F(X) > g) = \Phi\left(\frac{\mu_k^i - g}{\sigma_k^i}\right)$
$P(Req.|i\text{–}th\ Set) = \frac{n_{samples}\ such\ that\ \sum P(F(X) > g) > 0.1}{N_{samples}}$
Multiple requirements are combined by assuming conditional independence:
$P(i\text{–}th\ Set) = \prod P(j\text{–}th\ Req.|i\text{–}th\ Set)$

Figure 1.2: Detail of the Exploration phase process

1.5 Search Phase

The Search phase employs the Python library `pymoo` [2], which implements several optimisation algorithms based on heuristic methods. In the context of design exploration, optimality is sought as a mechanism to understand the impact of the parameters over the objectives and its trade-offs, rather than identifying the single best design. With this goal in mind, the population-based Universal Non-dominated Sorting Genetic Algorithm III (U-NSGA-III) was selected, which is efficient in both single, multi- and many-objective problems [7]. It is chosen as it is a gradient-free method, granting flexibility in the evaluation function. The drawback of population methods, however, is the large amount of evaluations of the objective function; hence, modeling should be as computationally cheap as possible.

This is fundamental as even with the filtering performed in the exploration phase, a substantial number of optimisation problems must be performed, taking a lot of computational time. The bounds of the input variables are those of the set, while objectives and constraints are the same defined in the previous phase.

2 Definition of a P-DOPT Test Case

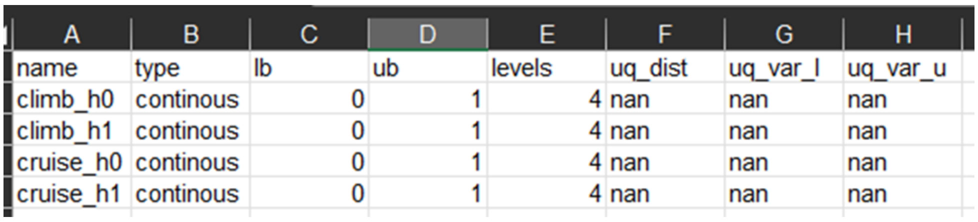
A PDOPT run is composed of:

- A python script containing the PDOPT library and the model evaluation function
- A .csv input file describing the input parameters
- A .csv responses file, describing the QoI

The script file must include the following components from the `pdopt` library:

```python
from pdopt.data import DesignSpace
from pdopt.exploration import  ProbabilisticExploration
from pdopt.optimisation import Optimisation
```

2.1 The `input.csv` file



	A	B	C	D	E	F	G	H
	name	type	lb	ub	levels	uq_dist	uq_var_l	uq_var_u
	climb_h0	continous	0	1	4	nan	nan	nan
	climb_h1	continous	0	1	4	nan	nan	nan
	cruise_h0	continous	0	1	4	nan	nan	nan
	cruise_h1	continous	0	1	4	nan	nan	nan

Figure 2.1: Content of the input.csv file

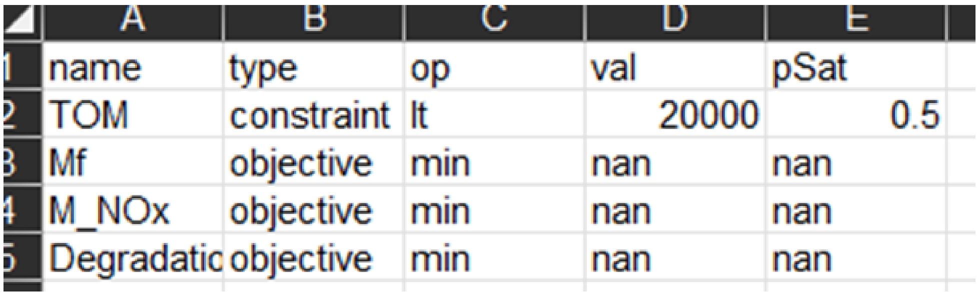
Input parameters are distinguished in continuous and discrete. Discrete parameters are defined by N integers, which are mapped in the evaluation function to discrete possibilities, while continuous parameters represent a range of possible variables.

The input.csv file is structured as such:

- **name** – Name of the parameter. It must not contain spaces, use underscore for it.
- **type** – If the variable is continuous or discrete. If discrete, lb and ub parameters will be ignored, as only integer levels will be provided.
- **lb** – Lower bound of the continuous parameter.
- **ub** - Upper bound of the continuous parameter.
- **levels** – Number of levels of the parameter, must be an integer.
- **uq_dist** – Type of UQ distribution for this parameter for the uncertainty-based optimisation. Can be either "uniform", "triang" (triangular), "norm" (gaussian). Set it to "nan" for ignoring. Uniform and Triangular distributions support asymmetric distributions, while Gaussian is only symmetric.
- **uq_var_l** – Lower variation bound for the distribution, expressed as a decimal percentile variation (i.e. 0.05 for 5% variation).
- **uq_var_u** – Upper variation bound for the distribution, expressed as decimal percentile variation. Set it to "nan" if the distribution is symmetric.

Each row in the .csv file must be filled. If a variable is unused, it must be set to "nan". It is important that **every** input parameter of the evaluation function that the exploration and optimisation phases will expect as input.

2.2 The `responses.csv` file



	A	B	C	D	E
1	name	type	op	val	pSat
2	TOM	constraint	lt	20000	0.5
3	Mf	objective	min	nan	nan
4	M_NOx	objective	min	nan	nan
5	Degradatic	objective	min	nan	nan

Figure 2.2: Content of the response.csv file

The response.csv file is structured as such:

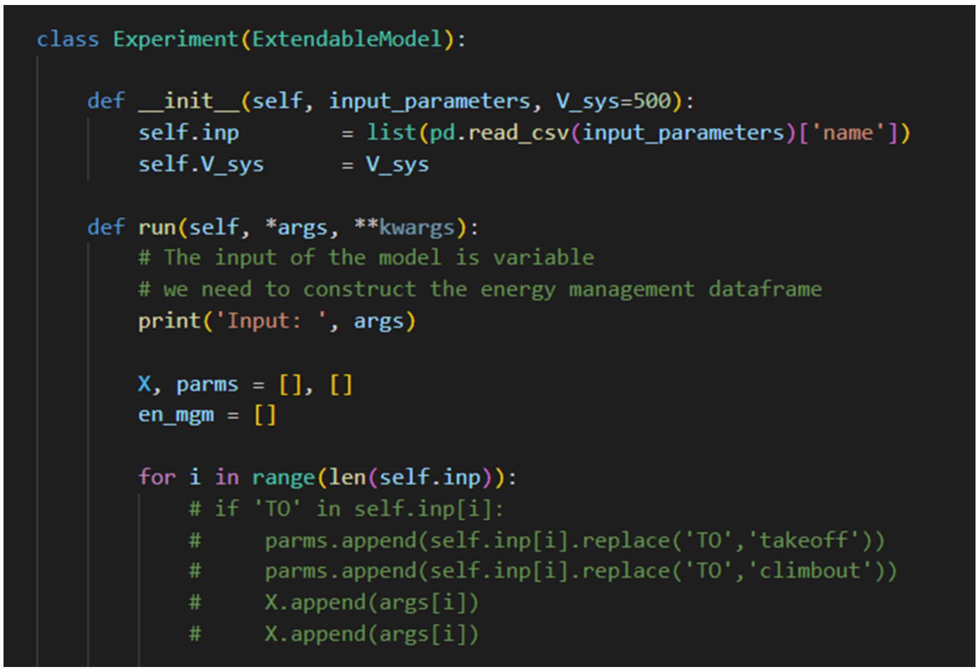
- **name** – Name of the response from the evaluation function. These names must be used in the dict() object that is returned from the evaluation function.
- **type** – Type of response. It can be either “constraint” or “objective”. These are the responses how will be handled by the optimisation step. In the exploration step, constraints are always active while objectives can be set up as constraints or not (see next point).
- **op** – Operator on the response. If type is set to “constraint” the operator is going to be either “lt” (less than) or “gt” (greater than), which represent the inequality of the response with respect to the quantity in the value column (i.e. “TOM, constraint, lt, 20000” corresponds to TOM ¡ 20000). If type is set to “objective” then the operator is either “min” (minimize) or “max” (maximise).
- **val** – Value which is going to be used by the operator. In case of an objective, this value is used to set up a constraint in the exploration phase with the following criteria: if “min” is set as operator, then it’s a less than constraint, if “max” is set as operator then it’s a greater than constraint. This is done as a way to drive the exploration phase and remove areas of the design space that might not satisfy minimum requirements. Set it to “nan” to disable the constraint.
- **pSat** – The minimum satisfaction probability for the constraint/objective. When evaluating the sets, this is the minimum probability for which samples are tested to. Samples that go under pSat are counted as non-satisfactory.

At least one entry has to be defined, otherwise the code will not run.

2.3 The evaluation function

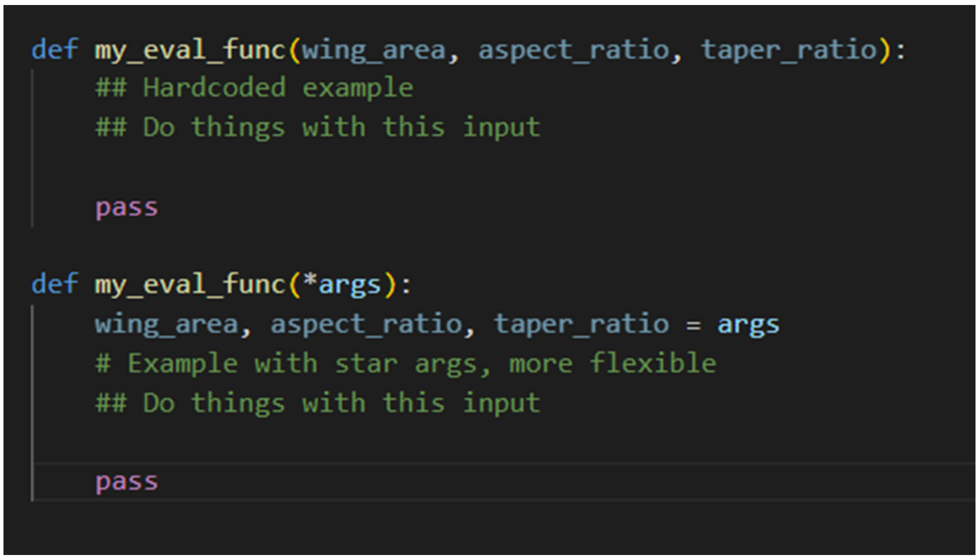
The evaluation function can be implemented in two ways. If the model does not require any setup or state, it can be implemented by wrapping the actual evaluation function in a `Model(model_fun)` object, and passing the function itself as argument. Otherwise, it is possible to define a python class which extends the `ExtendableModel` class as shown in Figure 2.3.

The method `run()` is expected by the P-DOPT framework and it must be defined according to the same evaluation function specification when passing it to the `Model()` object. The function takes as argument the parameters specified in the `input.csv` file. This can be hardcoded in the function or can be encapsulated in the `*args` object. For instance if the parameters are `wing_area`, `aspect_ratio`, `taper_ratio` the function can be either in both ways as shown in Figure 2.4.



```
class Experiment(ExtendableModel):

    def __init__(self, input_parameters, V_sys=500):
        self.inp        = list(pd.read_csv(input_parameters)['name'])
        self.V_sys      = V_sys

    def run(self, *args, **kwargs):
        # The input of the model is variable
        # we need to construct the energy management dataframe
        print('Input: ', args)

        X, parms = [], []
        en_mgm = []

        for i in range(len(self.inp)):
            # if 'TO' in self.inp[i]:
            #     parms.append(self.inp[i].replace('TO','takeoff'))
            #     parms.append(self.inp[i].replace('TO','climbout'))
            #     X.append(args[i])
            #     X.append(args[i])
```

Figure 2.3: An implementation of the `ExtendableModel` class



```
def my_eval_func(wing_area, aspect_ratio, taper_ratio):
    ## Hardcoded example
    ## Do things with this input

    pass

def my_eval_func(*args):
    wing_area, aspect_ratio, taper_ratio = args
    # Example with star args, more flexible
    ## Do things with this input

    pass
```

Figure 2.4: Examples of inputs to the runnable evaluation function

The function must return a python dictionary whose keys are the names defined in the response.csv file. For example if the responses in the `response.csv` file are `mass` and `CD`, the returned object of the evaluation function has to be the dictionary shown in Figure 2.5.

2.4 Layout of the runfile

The definition of the runfile follows the two step process outlined. First the input file is read and the `DesignSpace()` object is constructed. Then the exploration phase is performed by first training the Gaussian Processes (by constructing the `ProbabilisticExploration()` object) and then running it. The output of the first step is saved in the design space object and can be retrieved with the `get_exploration_results()` method. Then optimisation is performed

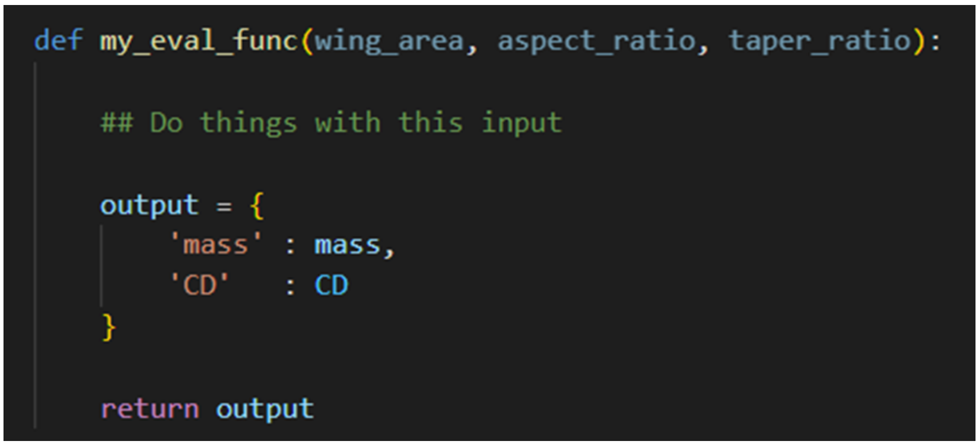
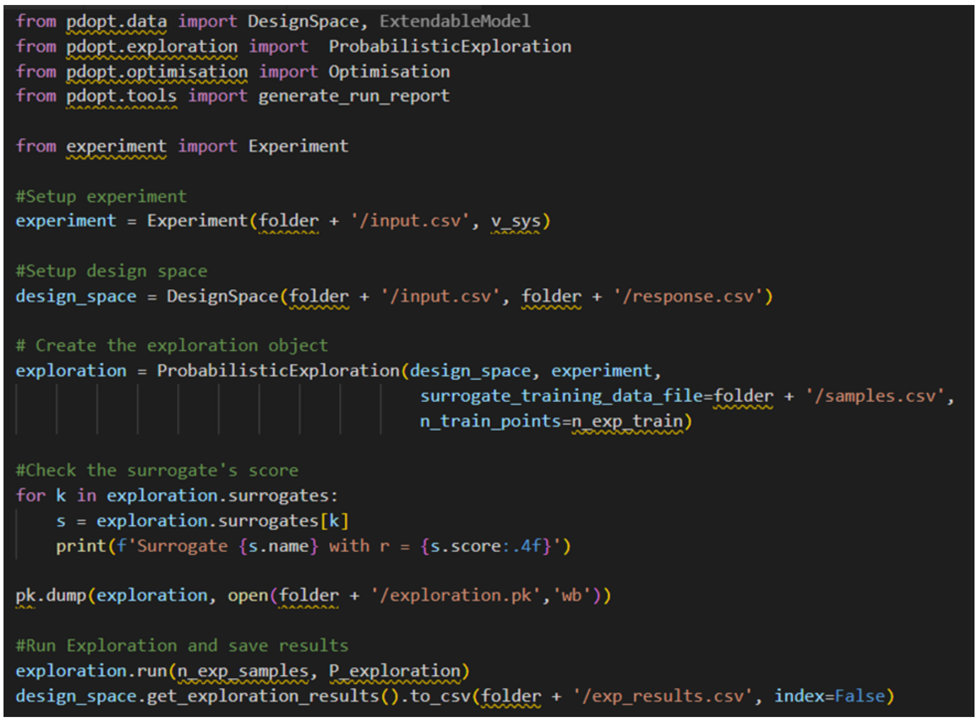

```
def my_eval_func(wing_area, aspect_ratio, taper_ratio):

    ## Do things with this input

    output = {
        'mass' : mass,
        'CD'   : CD
    }

    return output
```

Figure 2.5: Examples of inputs to the runnable evaluation function

by first setting up the `Optimisation()` object and then running it. Results are available from the `get_optimum_results()` method of the design space. A minimum commented example is shown in Figure 2.6. The output provided by the two dataframes are structured as follows:

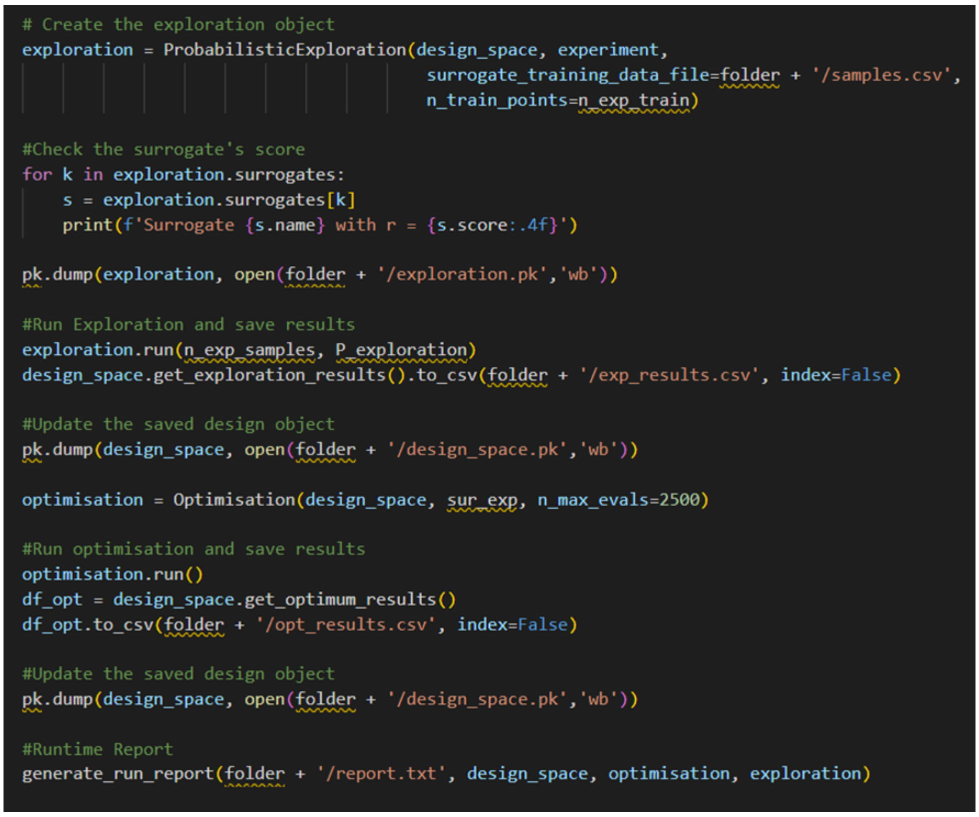
- The exploration dataframe contains the id of the set, the levels of each parameter (going from zero to n_levels), and the associated probabilities for the different constraints plus the total probability.
- The optimisation dataframe contains all the pareto front points for each set. Columns are set_id, input parameters and the response outputs.

The function `generate_run_report()` can be used to produce an output that contains the time of run for each step.



```python
from pdopt.data import DesignSpace, ExtendableModel
from pdopt.exploration import  ProbabilisticExploration
from pdopt.optimisation import Optimisation
from pdopt.tools import generate_run_report

from experiment import Experiment

#Setup experiment
experiment = Experiment(folder + '/input.csv', v_sys)

#Setup design space
design_space = DesignSpace(folder + '/input.csv', folder + '/response.csv')

# Create the exploration object
exploration = ProbabilisticExploration(design_space, experiment,
                                       surrogate_training_data_file=folder + '/samples.csv',
                                       n_train_points=n_exp_train)

#Check the surrogate's score
for k in exploration.surrogates:
    s = exploration.surrogates[k]
    print(f'Surrogate {s.name} with r = {s.score:.4f}')

pk.dump(exploration, open(folder + '/exploration.pk','wb'))

#Run Exploration and save results
exploration.run(n_exp_samples, P_exploration)
design_space.get_exploration_results().to_csv(folder + '/exp_results.csv', index=False)
```

(a)



```python
# Create the exploration object
exploration = ProbabilisticExploration(design_space, experiment,
                                       surrogate_training_data_file=folder + '/samples.csv',
                                       n_train_points=n_exp_train)

#Check the surrogate's score
for k in exploration.surrogates:
    s = exploration.surrogates[k]
    print(f'Surrogate {s.name} with r = {s.score:.4f}')

pk.dump(exploration, open(folder + '/exploration.pk','wb'))

#Run Exploration and save results
exploration.run(n_exp_samples, P_exploration)
design_space.get_exploration_results().to_csv(folder + '/exp_results.csv', index=False)

#Update the saved design object
pk.dump(design_space, open(folder + '/design_space.pk','wb'))

optimisation = Optimisation(design_space, sur_exp, n_max_evals=2500)

#Run optimisation and save results
optimisation.run()
df_opt = design_space.get_optimum_results()
df_opt.to_csv(folder + '/opt_results.csv', index=False)

#Update the saved design object
pk.dump(design_space, open(folder + '/design_space.pk','wb'))

#Runtime Report
generate_run_report(folder + '/report.txt', design_space, optimisation, exploration)
```

(b)

Figure 2.6: Runfile example

3 Example Tutorial

This section presents the step by step process to perform a P-DOPT analysis. For the purpose of this tutorial, the specific details of the simulation function will be overlooked, focusing more on the process of developing a full runnable test case.

Generally it is composed of an understanding and definition of the problem in terms of input parameters and output quantities, and the layout of the runfile to perform this analysis. The example presented here can be found in the GitHub example folder.

3.1 Problem Definition

The example problem is one of the test cases that have been presented in [9]: find the optimal energy management strategies which minimise fuel burn and NOx emissions, with a maximum take-off mass constraint of 20000 kg. The airplane is an hybrid-electric 50 seater regional turboprop, similar to an ATR-42. The propulsion system is a mechanically-integrated hybrid system composed of a gas turbine and an electric motor connected to the propeller with a planetary gearbox (Figure 3.1). The mission profile is shown in Figure 3.2, and it is composed of a main and alternate portion.

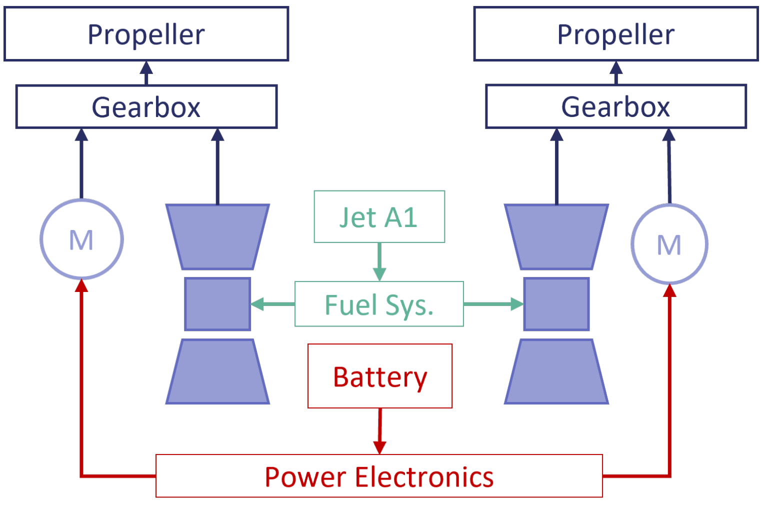


Figure 3.1: Propulsion Architecture

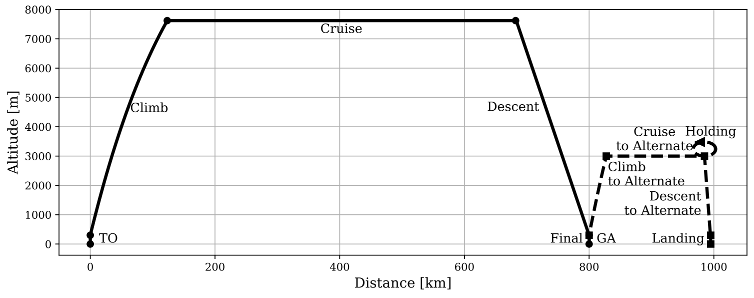


Figure 3.2: Mission profile

The energy management strategies (EMS) are parameterised as a piece-wise linear function of Degree of Hybridisation (DOH). In this specific test problem, the EMS are limited to linear functions defined by the extreme points, over the climb and cruise mission segments of the main phase (See the example in Fig. 3.3, which shows a family of linear EMS).

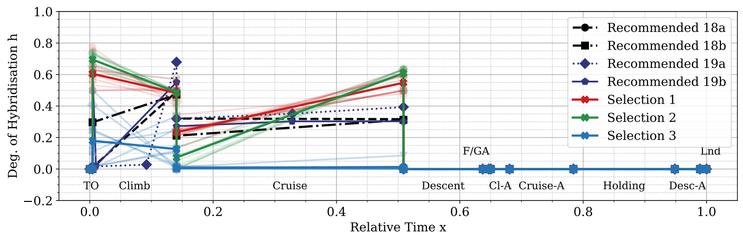


Figure 3.3: Different linear EMS over Climb and Cruise.

3.2 Breakdown of the problem into P-DOPT components

After understanding the problem to analyse, the user should identify the input parameters of the model and the responses he desires to optimise or impose constraints on. For this example, the following input parameters have been identified:

Name	Type	Range	Description
climb_h0	continuous	[0,1]	The DOH at the beginning of the climb segment
climb_h1	continuous	[0,1]	The DOH at the end of the climb segment
cruise_h0	continuous	[0,1]	The DOH at the beginning of the cruise segment
cruise_h1	continuous	[0,1]	The DOH at the end of the cruise segment

Table 3.1: Identified input parameters for the problem under analysis.

The use should decide if these parameters are continuous or discrete, i.e. describe a range of possible values or a set of discrete choices. In the first case, the user should also fix the boundaries of this range. This information will be used in defining the `input.csv` file and the arguments of the evaluation function. Then the user should list the responses that they wish to analyse, and define them as objectives or constraints. Special care should be given to objectives, as they can be set as ”soft constraints” for the exploration phase. This is useful if the user wishes to understand which areas of the design space are guaranteed to provide at most (or least) a certain value. For instance, the user wishes to see which areas of the design space are more likely to have a minimum value of NO_x emissions under a certain figure. For the example problem, the following requirements are identified:

Name	Type	Operator	Value	Description
Mf	objective	min (minimise)	nan	Fuel Burnt during the mission
M_NOx	objective	min (minimise)	nan	NO_x Emission during the mission
TOM	constraint	lt (less than)	20000	Take-off mass, constrainted to 20000 kg

Table 3.2: Identified response parameters for the problem under analysis.

These response parameters will then define the output of the evaluation function and the `responses.csv` file.

3.3 Definition of the .csv Files

With the problem broken down into its P-DOPT components, it is possible to define the input file, with the format described in 2.1. In this case the input file is as follows:

```
    name,type,lb,ub,levels,uq_dist,uq_var_l,uq_var_u
    climb_h0,continous,0,1,4,nan,nan,nan
    climb_h1,continous,0,1,4,nan,nan,nan
    cruise_h0,continous,0,1,4,nan,nan,nan
    cruise_h1,continous,0,1,4,nan,nan,nan
```

The number of levels for each parameter should be selected with caution. Few levels make a faster analysis, especially during the search phase, however it is highly likely the exploration phase would average the probability, leading to a potential loss of resolution of the decision boundary.

On the other hand, an high number of levels leads to an unnecessary number of surviving sets, which would slow down the search process considerably. It is advised to use 3 or 4 levels when starting, and adjusting it by performing a few exploration steps.

Along the input file, the responses csv file must be defined as described in 2.2. Using the information from the problem breakdown the `response.csv` file is as follows:

```
    name,type,op,val,pSat
    Mf,objective,min,nan,nan
    M_NOx,objective,min,nan,nan
    TOM,constraint,lt,20000,0.5
```

As discussed previously, it is possible to include "soft constraints" on the objectives for the exploration phase. It is suggested to not do so at the first run of the problem, unless the user is already aware the range of response values. Furthermore, soft constraints present a phenomenon of "leakage", whereas some sub-optimal sets are kept as they might have a few points satisfying the constraint.

3.4 Definition of the Evaluation Function

In this case, the evaluation function was developed by extending the `ExtendableModel()` class. This was preferred over defining just a Python function (as described in 2.3) for two reasons. First, it is possible to load in the `ExtendableModel()` object data regarding what is being modeled (for instance, model parameters that are fixed during each run of P-DOPT). Second, it allows to package with the model data other routines, such as a post-processing function. This is the case for this example problem. It must contain the `run()` method, which must output the data required from the `response.csv` file as a Python dictionary.

The model class is therefore defined as:

```python
class Experiment(ExtendableModel):

    def __init__(self, input_parameters, architecture, mission_file):

        ## This constructor method allows to store in the Experiment
        ## object some information regarding the aircraft and mission
        self.inp         = list(pd.read_csv(input_parameters)['name'])
        self.arch        = architecture
        self.arch0       = architecture
        self.mission     = mission_file

    def run(self, *args, **kwargs):

        ## Omitted code for performing the analysis

        output = {
                'TOM'       : analysis.iloc[-1].mass,
                'Mf'        : analysis.iloc[-1].m_fl,
                'M_NOx'     : analysis.iloc[-1].m_NOx,
                 }

        return output

    def postprocess_analysis(self, *args, **kwargs):

        ## Omitted post-processing code, runs the same analysis
        ## but returns more information

        return results
```

3.5 Definition of the Runfile

Now that all the ingredients required to perform the analysis are present, it is possible to construct the Python script to run it. First a `DesignSpace()` object has to be created, which will store all the information regarding the problem. It contains a list of all the sets (which are instances of the `DesignSet()` object) and the Pandas DataFrames containing the exploration and search phases results. The directory of the `'input.csv'` and `'response.csv'` are required as arguments. To avoid loss of data, the `pickle` library is used to store each P-DOPT object after each step is performed.

```python
folder = 'run'
architecture = json.load(open('data/architecture.json', 'r'))
mission = 'data/mission_original.csv'
experiment = Experiment(folder + '/input.csv',
                        architecture, mission)

# Check if a design space is already present otherwise create it
if exists(folder + '/design_space.pk') and restart:
    design_space = pk.load(open(folder + '/design_space.pk','rb'))
else:
    design_space = DesignSpace(folder + '/input.csv',
                               folder + '/response.csv')
    pk.dump(design_space, open(folder + '/design_space.pk','wb'))
```

The next step is defining the exploration phase. The `ProbabilisticExploration()` object is used. It requires as arguments the design space object, the model object. Optional arguments include the definition of a surrogate data file, useful to expedite multiple analyses, and the number of samples to use. By default it will use 100 sampled points in the entire design space (using a Latin Hypercube scheme) and 30 points to validate its regression. These points are sampled from the provided model. Once the exploration object is trained, it is run with the `.run()` method, by passing the number of samples for evaluating each set and the minimum satisfaction probability (as discussed in 1.4). The results from the exploration phase are stored in the `DesignSpace()` object, they can be retrived with the `.get_exploration_results()` method.

```python
n_exp_train = 100

# Check if there is already a trained exploration object
if exists(folder + '/exploration.pk') and restart:
    exploration = pk.load(open(folder + '/exploration.pk','rb'))
else:
    exploration = ProbabilisticExploration(design_space,
                    experiment,
                    surrogate_training_data_file=folder + '/samples.csv',
                    n_train_points=n_exp_train)

    for k in exploration.surrogates:
        s = exploration.surrogates[k]
        #Print the R score of each surrogate, to check
        #their training. For smooth problems it should be
        #above 0.8.
        print(f'Surrogate {s.name} with r = {s.score:.4f}')

    pk.dump(exploration, open(folder + '/exploration.pk','wb'))

n_exp_samples = 100
P_exploration = 0.5

# Check if exploration has been done already
```

```
if exists(folder + '/exp_results.csv') and restart:
    pass
else:
    exploration.run(n_exp_samples, P_exploration)
    design_space.get_exploration_results().to_csv(folder \
    + '/exp_results.csv', index=False)

    #Update the saved design object
    pk.dump(design_space, open(folder + '/design_space.pk','wb'))
```

It is possible to perform different exploration analysis by using copies of the `DesignSpace()` object and changing the satisfaction probability. This allows to study how restrictive the requirements are over the design space. Once complete, the exploration step marks some areas of the design space as "Discarded" by setting the `.isDiscarded` boolean of each set as true. The following phase the search step. A multi-objective optimisation algorithm is introduced in the surviving sets to identify the local Pareto front. The `Optimisation()` object is the standard deterministic multi-objective optimiser. The arguments are the design space itself, the model and options regarding the stopping criteria or the genetic algorithm population size. For the purpose of the P-DOPT analysis it is reccomended to use a fixed number of evaluation functions (`n_max_evals`) or generations (`n_max_gen`). This would allow a fair search in every set, as it would ignore the topology of the evaluation function. Instad, using criteria based on convergence would introduce distortions, as some areas of the design space might be shallow (slow convergence) or have multiple local minima/maxima.

Once set up is complete, optimisation is started with the `Optimisation.run()` method. Once complete, results can be retrieved from the `.get_optimum_results()` method of the design space object. It returns a Pandas DataFrame containing the optimal points of each set and the return value of the constrained quantities. The optimisation step code is:

```
optimisation = Optimisation(design_space, experiment, n_max_evals=2000)


# Check if optimisation has been done already
if exists(folder + '/opt_results_raw.csv') and restart:
    pass
else:
    optimisation.run()
    df_opt = design_space.get_optimum_results()
    df_opt.to_csv(folder + '/opt_results_raw.csv', index=False)

    #Update the saved design object
    pk.dump(design_space, open(folder + '/design_space.pk','wb'))


## Post-processing code to recover other output quantities from the model
if exists(folder + '/opt_results.csv') and restart:
    pass
else:
    inp_pars = design_space.par_names

    df_opt['M_batt']  = 0
    df_opt['eff']     = 0
    df_opt['M_CO']    = 0
    df_opt['M_CO2']   = 0
    df_opt['eff_GT_cl'] = 0
    df_opt['eff_GT_cr'] = 0
    df_opt['eff_cl'] = 0
    df_opt['eff_cr'] = 0

    for index in tqdm(range(len(df_opt))):
        t_out = experiment.postprocess_analysis(*df_opt.loc[index, inp_pars].
    tolist())

        df_opt.loc[index, 'M_batt'] = t_out.iloc[-1].m_bat
        df_opt.loc[index, 'eff'] = t_out.iloc[-1].eff
        df_opt.loc[index, 'M_CO'] = t_out.iloc[-1].m_CO
        df_opt.loc[index, 'M_CO2'] = t_out.iloc[-1].m_fl * 3
        df_opt.loc[index, 'eff_GT_cl'] = t_out.loc[t_out['tag'] == 'climb'].
    P_GT_out.sum() / t_out.loc[t_out['tag'] == 'climb'].P_GT_in.sum()
        df_opt.loc[index, 'eff_GT_cr'] = t_out.loc[t_out['tag'] == 'cruise'].
    P_GT_out.sum() / t_out.loc[t_out['tag'] == 'cruise'].P_GT_in.sum()
        df_opt.loc[index, 'eff_cl'] = t_out.loc[t_out['tag'] == 'climb'].P_req.
    sum() / t_out.loc[t_out['tag'] == 'climb'].P_tot.sum()
        df_opt.loc[index, 'eff_cr'] = t_out.loc[t_out['tag'] == 'cruise'].P_req
    .sum() / t_out.loc[t_out['tag'] == 'cruise'].P_tot.sum()

        t_out.to_csv(folder + f'/missions/{index}_mission_output.csv')

    df_opt.to_csv(folder + '/opt_results.csv', index=False)

#Runtime Report
generate_run_report(folder + '/report.txt',
        design_space, optimisation, exploration)
```

By design, the evaluation function returns only the quantities of interest that have been specified in `response.csv`. In order to recover other information after the optimisation run, a post-processing evaluation function can be used by taking as input the optimal points found. Finally the `generate_run_report()` function is used to produce a text output containing information on the total time of run and number of discarded sets.

3.6 Running the Case and Post-processing the results

When the full script is ready with the required input files, the file is run. While the code is running, it will output to the console the current progress (Listing 3.1). In particular, after training the Gaussian processes, it will output the R^2 score (Coefficient of Determination) of each trained surrogate model. This gives a good indication of the reliability of the exploration process. Usually if the response of the model is smooth, it is expected to be at least above 0.8. Increasing the number of training points may be necessary for non-smooth models. The ouput during the search phase, is the underlying U-LSGA-III output from the `pymoo` library [2]. It informs the user about the progress of each generation (`n_gen`) with the number of function evaluations made so far (`n_eval`), the minimum and average constraint violations (`cv (min)` and `cv (avg)`), the number of non-dominated solutions found (`n_nds`) and the change of the Pareto convergence indicator (columns `epsindicator`). The `pymoo` documentation [3] provides more detail about this output in the Display section, futhermore points to an article by Blank and Deb on the multiobjective convergence indicator [1].

```
Generating Training Data:  100%|**********************|100/100 [01:23<00:00,  1.20it/s]
Generating Testing Data: 100%|*****************************| 30/30 [00:28<00:00, 1.06it/s]
Training Surrogate Responses: 100%|*************************| 3/3 [00:00<00:00, 36.59it/s]
Surrogate M_NOx with r = 0.9931
Surrogate TOM with r = 1.0000
Surrogate Mf with r = 0.9999
Exploring the Design Space: 100%|*********************| 256/256 [00:00<00:00, 639.79it/s]
Searching in the Design Space:   0%|                           | 0/25 [00:00<?, ?it/s]
==========================================================================================
n_gen |  n_eval |    cv (min)    |    cv (avg)    |  n_nds  |      eps       |  indicator
==========================================================================================
    1 |     101 |  0.00000E+00 |  0.00000E+00 |      14 |              - |              -
    2 |     202 |  0.00000E+00 |  0.00000E+00 |      17 |  0.012666136 |          ideal
    3 |     303 |  0.00000E+00 |  0.00000E+00 |      19 |  0.009879243 |          ideal
    4 |     404 |  0.00000E+00 |  0.00000E+00 |      19 |  0.017813004 |          ideal
    5 |     505 |  0.00000E+00 |  0.00000E+00 |      21 |  0.046526161 |          ideal
    6 |     606 |  0.00000E+00 |  0.00000E+00 |      21 |  0.017590293 |          ideal
    7 |     707 |  0.00000E+00 |  0.00000E+00 |      20 |  0.058322279 |          ideal
    8 |     808 |  0.00000E+00 |  0.00000E+00 |      20 |  0.007842796 |              f
    9 |     909 |  0.00000E+00 |  0.00000E+00 |      20 |  0.001397298 |              f
   10 |    1010 |  0.00000E+00 |  0.00000E+00 |      20 |  0.003076413 |              f
   11 |    1111 |  0.00000E+00 |  0.00000E+00 |      20 |  0.005812082 |              f
   12 |    1212 |  0.00000E+00 |  0.00000E+00 |      20 |  0.046733100 |          ideal
   13 |    1313 |  0.00000E+00 |  0.00000E+00 |      21 |  0.007279993 |              f
   14 |    1414 |  0.00000E+00 |  0.00000E+00 |      21 |  0.003993163 |              f
   15 |    1515 |  0.00000E+00 |  0.00000E+00 |      21 |  0.005201088 |              f
   16 |    1616 |  0.00000E+00 |  0.00000E+00 |      21 |  0.001729105 |              f
   17 |    1717 |  0.00000E+00 |  0.00000E+00 |      21 |  0.000351185 |              f
   18 |    1818 |  0.00000E+00 |  0.00000E+00 |      21 |  0.002476596 |              f
   19 |     1919 |  0.00000E+00 |  0.00000E+00 |       21 |  0.005252863 |            f
   20 |     2020 |  0.00000E+00 |  0.00000E+00 |       21 |  0.003703162 |            f
Searching in the Design Space:   4%|*                  | 1/25 [20:00<8:00:08, 1200.34s/it]
================================================================================
n_gen |  n_eval |   cv (min)    |   cv (avg)    |  n_nds  |      eps      |  indicator
================================================================================
    1 |      101 |  0.00000E+00 |  0.00000E+00 |       13 |            - |            -
    2 |      202 |  0.00000E+00 |  0.00000E+00 |       18 |  0.066779342 |        ideal
    3 |      303 |  0.00000E+00 |  0.00000E+00 |       19 |  0.013040164 |            f
    4 |      404 |  0.00000E+00 |  0.00000E+00 |       19 |  0.003240776 |        ideal
    5 |      505 |  0.00000E+00 |  0.00000E+00 |       19 |  0.006436898 |            f
    6 |      606 |  0.00000E+00 |  0.00000E+00 |       18 |  0.013058896 |        ideal
    7 |      707 |  0.00000E+00 |  0.00000E+00 |       18 |  0.008253434 |            f
    8 |      808 |  0.00000E+00 |  0.00000E+00 |       17 |  0.023029914 |        ideal
    9 |      909 |  0.00000E+00 |  0.00000E+00 |       17 |  0.002872064 |            f
   10 |     1010 |  0.00000E+00 |  0.00000E+00 |       18 |  0.008945702 |            f
   11 |     1111 |  0.00000E+00 |  0.00000E+00 |       18 |  0.002731946 |            f
   12 |     1212 |  0.00000E+00 |  0.00000E+00 |       18 |  0.006083088 |        nadir
   13 |     1313 |  0.00000E+00 |  0.00000E+00 |       18 |  0.006646305 |            f
   14 |     1414 |  0.00000E+00 |  0.00000E+00 |       18 |  0.004031693 |            f
   15 |     1515 |  0.00000E+00 |  0.00000E+00 |       18 |  0.002041885 |            f
   16 |     1616 |  0.00000E+00 |  0.00000E+00 |       19 |  0.004039912 |            f
   17 |     1717 |  0.00000E+00 |  0.00000E+00 |       19 |  0.001856166 |            f
   18 |     1818 |  0.00000E+00 |  0.00000E+00 |       19 |  0.004233351 |            f
   19 |     1919 |  0.00000E+00 |  0.00000E+00 |       19 |  0.004437145 |            f
   20 |     2020 |  0.00000E+00 |  0.00000E+00 |       19 |  0.028033354 |        nadir
Searching in the Design Space:   8%|**                 | 2/25 [39:41<7:35:48, 1189.07s/it]
================================================================================
n_gen |  n_eval |   cv (min)    |   cv (avg)    |  n_nds  |      eps      |  indicator
================================================================================
    1 |      101 |  0.00000E+00 |  0.148803098 |       15 |            - |            -
    2 |      202 |  0.00000E+00 |  0.00000E+00 |       17 |  0.012496447 |            f
    3 |      303 |  0.00000E+00 |  0.00000E+00 |       18 |  0.017854883 |            f
    4 |      404 |  0.00000E+00 |  0.00000E+00 |       18 |  0.037275050 |        ideal
```

Listing 3.1: Console Output

The analysis ouputs are saved in the `.csv` files, as specified in the script, alongside the pickled Python object for inspection and debugging. First, let's inspect the report file which provides information about the computational time and the number of discarded sets (Listing 3.2). This information is useful for tuning the number of levels for each set and the number of samples, both for training and set evaluation.

```
Total Number of Sets      : 256
Number of Surviving Sets  : 26

Total Surrogate Train Time :        0.021 s
Total Exploration Time     :        0.861 s
Total Search Time          :     5358.473 s
Number of Cores Used       :           16

Train time and score of each Surrogate:
       TOM    0.0211(s)     0.9998

Search time and f_evals of each Set:
         0  205.0636(s)
         1  206.2956(s)
         4  204.2401(s)
         8  203.9787(s)
        16  203.4907(s)
        17  206.1708(s)
        20  203.5766(s)
        32  204.3334(s)
        33  206.6882(s)
        36  207.6279(s)
        48  206.3984(s)
        64  203.8703(s)
        65  206.5425(s)
        68  206.0475(s)
        80  205.4995(s)
        81  208.0759(s)
        84  209.0785(s)
        96  206.8746(s)
       112  206.2059(s)
       128  206.7444(s)
```

```
        129   207.0894(s)
        132   207.0260(s)
        144   207.7437(s)
        160   205.8206(s)
        192   206.0405(s)
        208   207.9498(s)
```

Listing 3.2: Report Output

A printout of the Pandas dataframe of the exploration results is shown in Listing 3.3. This data is stored in a `.csv` file. The format of the columns is as follows:

- `set_id` : The number identifying the set.
- `is_discarded` : A boolean value of 0/1 indicating if the set has been discarded.
- The input parameters specified in `input.csv`, each column contains the level selected for each set.
- `P` : The total calculated probability for that set.
- The probabilities for each requirement that has been specified in the `response.csv` file such that it plays a role in the exploration phase.

```
     set_id  is_discarded  climb_h0  climb_h1  cruise_h0  cruise_h1     P  P_TOM
0         0             0         0         0          0          0  1.00   1.00
1         1             0         0         0          0          1  1.00   1.00
2         2             1         0         0          0          2  0.44   0.44
3         3             1         0         0          0          3  0.00   0.00
4         4             0         0         0          1          0  0.98   0.98
..      ...           ...       ...       ...        ...        ...   ...    ...
251     251             1         3         3          2          3  0.00   0.00
252     252             1         3         3          3          0  0.00   0.00
253     253             1         3         3          3          1  0.00   0.00
254     254             1         3         3          3          2  0.00   0.00
255     255             1         3         3          3          3  0.00   0.00

[256 rows x 8 columns]
```

Listing 3.3: Exploration Results Dataframe

A printout of the Pandas dataframe of the search results is shown in Listing 3.4. This information is also stored in a `.csv` file for easy interchange with other data analysis softwares. Unlike the exploration datafraeme, here each row represents a single design point: filtering by `set_id` is necessary for recovering the Pareto front of each surviving set. The structure of the table of data is as follows:

- `set_id` : The number identifying the set that this data point belongs to.
- Columns of the input parameters as defined in `input.csv`. These values are the actual number and not a level, it's the found optimum value. For this example they are the degrees of hybridisation at the beginning and end of each climb and cruise mission phase.
- Columns of the response quantities, objective and constraints, as defined in `response.csv`. In this example `TOM` is the take-off mass constraint value, while `Mf` and `M_NOx` are the two objectives to be minimised.

```
     set_id  climb_h0  climb_h1  cruise_h0  cruise_h1           TOM           Mf     M_NOx
0       0.0  0.019019  0.042392   0.176987   0.136356  18488.123494  1053.114840  6.390926
1       0.0  0.248721  0.247232   0.248985   0.246306  19570.910964  1019.602066  6.012993
2       0.0  0.247531  0.247817   0.228041   0.242325  19497.310632  1021.829023  6.047204
3       0.0  0.061144  0.103122   0.119441   0.056398  18250.701614  1060.478028  6.395150
4       0.0  0.242335  0.001371   0.010946   0.010773  17969.991130  1070.045548  6.416366
..      ...       ...       ...        ...        ...           ...          ...       ...
590   208.0  0.771168  0.282262   0.089347   0.093017  19435.377825  1025.854261  5.904732
591   208.0  0.843885  0.322598   0.078514   0.163723  19780.236214  1015.011526  5.812730
592   208.0  0.975473  0.255174   0.126832   0.076693  19750.269531  1015.474685  5.853289
593   208.0  0.773123  0.411460   0.108772   0.104280  19701.794994  1017.944584  5.820675
594   208.0  0.752590  0.252407   0.005897   0.001813  18873.928896  1044.221176  6.062478

[595 rows x 8 columns]
```

Listing 3.4: Search Results Dataframe

The `.csv` format allows for easy interchange between the PDOPT results and other visualisation programs. On the other hand, the `pandas` and `matplotlib` Python libraries can be used for visualising the output. While outside of the scope of this manual, it is suggested to use a combination of Parallel Coordinates [5] and Scatter plots as shown in Figures 3.4 and 3.5. The example shown here is the result of the selection of the Pareto points from the scatter plot: the parallel coordinates plot allows to correlate each objective with the input and other quantities of interest [6].

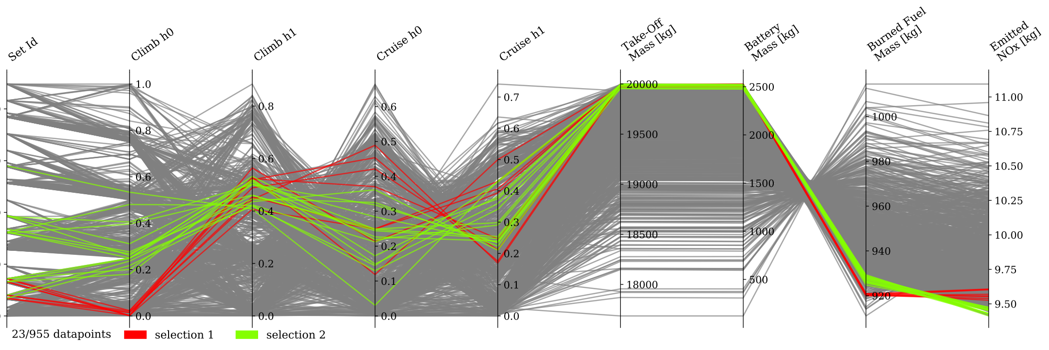


Figure 3.4: Parallel Coordinates with selection.

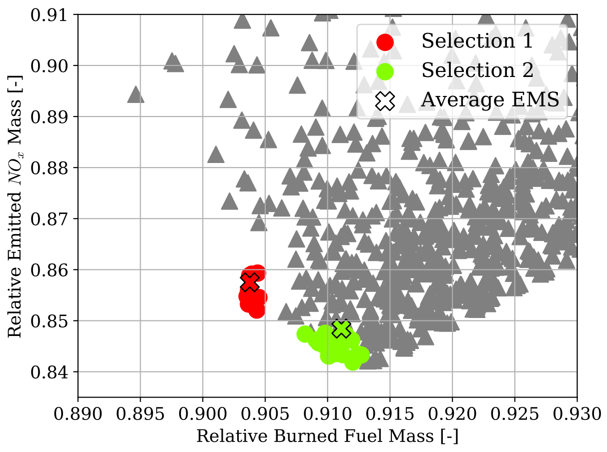


Figure 3.5: Scatter plot showing the selection of Fig. 3.4

4 API Reference

The API reference is structured in seactions corresponding to each file in the PDOPT library.

4.1 Data Structures (`pdopt.data`)

This module contains all the data structures utilised within PDOPT.

`class` `pdopt.data.`**`ExtendableModel`**()

Abstract class for wrapping evaluation functions if they require some state that has to be maintained. See the example in 2.3.

Parameters:

None

Returns:

None

`method` `pdopt.data.ExtendableModel.`**`run`**()

The `run()` method has to be overloaded with the evaluation function required to run the analysis. The input parameters must match the ones of the `input.csv` file, while it must return a Python `dict` containing the responses outlined in the `response.csv` file.

Parameters:

None

Returns:

None

`class` `pdopt.data.`**`ContinousParameter`**(*name, lb, ub, n_levels, uq_dist, uq_var_l, uq_var_u*)

This class reprents a continous parameter as defined in the input.csv file of the PDOPT case.

Parameters:

name [**str**]	The name of the continous parameter.
lb [**float**]	Lower bound value of the continous parameter.
ub [**float**]	Upper bound value of the continous parameter.
n_levels [**int**]	Number of levels to discretise the continouus range.
uq_dist [**str**]	Type of uncertainty distribution to be applied to this parameter. Options are ”norm”, ”uniform” and ”triang”.
uq_var_l [**float**]	Percentile lower variation of the quantity from the expected mean.
uq_var_u [**float**]	Percentile upper variation of the quantity from the expected mean. If symmetric, this has to be set to None.

Returns:

None

method `pdopt.data.ContinousParameter.`**`get_bounds`**()

Returns a tuple with the continous parameter bounds

Parameters:

None

Returns:

bounds [**(float, float)**]	The lower and upper bounds of the continouus parameter.

method `pdopt.data.ContinousParameter.`**`get_level_bounds`**(*level*)

Returns a tuple containing the bounds of the selected level.

Parameters:

level [**int**]	N-th selected level.

Returns:

bounds [**(float, float)**]	The lower and upper bounds of the selected level.

method `pdopt.data.ContinousParameter.`**`sample`**(*n_samples, level=None*)

Sample within the entire continuous parameter or in a level.

Parameters:

n_samples [**int**]	Number of samples.
level [**int, optional**]	N-th level to sample in. If None, sample in the entire range.

Returns:

bounds [**numpy.ndarray**]	Array of random samples of length *n_samples*.

method `pdopt.data.ContinousParameter.`**`ppf`**(*quantile, x0*)

Inverse cumulative function for obtaining random values around a reference point, given a quantile.

Parameters:

quantile [**float or numpy.ndarray**]	Probability quantile(s)
x0 [**float**]	Mean value of the uncertainty distribution.

Returns:

bounds [**numpy.ndarray**]	Array of samples from the distribution matching the quantiles. *n_samples*.

class pdopt.data.**DiscreteParameter**(*name, n_levels*)

This class reprents a discrete parameter as defined in the input.csv file of the PDOPT case. In essence it acts as a C `enum` as it returns only integers ranging from 0 to *n_levels*.

Parameters:

name [**str**]	The name of the discrete parameter.
n_levels [**int**]	Number of levels of the discrete parameter.

Returns:

None

method pdopt.data.DiscreteParameter.**get_n_levels**()

Returns the number of levels in this parameter.

Parameters:

None

Returns:

n_levels [**int**]	The total number of levels of this parameter.

class pdopt.data.**Objective**(*name, operand, min_requirement=None, p_sat=0.5*)

This class represents an objective as defined in the response.csv file of the PDOPT case.

Parameters:

name [**str**]	The name of the discrete parameter.
operand [**str**]	The type of objective. It can be either "min" for minimise or "max" for maximise.
min_requirement [**float, optional**]	Optional soft constraint. If present, it will affect the exploration phase by setting a maximum value constraint (if objective set to minimise), viceversa minimum value constraint (if objective set to maximise).
p_sat [**float, optional**]	If the soft constraint is set, the satisfaction probability

Returns:

None

method pdopt.data.Objective.**get_requirement**()

Get the disequation that defines the soft constraint, if present.

Parameters:

None

Returns:

requirement [**(str, float)**]	Tuple containing the type of constraint ('lt' for < and 'gt' for >) and the value of the constraint. None if no soft constraint is present.

method pdopt.data.Objective.**get_operand**()

Because the `pymoo` optimiser is by default set to minimise, this method converts the objective to a minimisation objective by flipping the sign.

Parameters:

None

Returns:

sign [**int**]	Returns -1 if objective is set to maximise, 1 otherwise.

class pdopt.data.**Constraint**(*name, operand, value, p_sat=0.5*)

This class represents a constraint as defined in the response.csv file of the PDOPT case.

Parameters:

name [**str**]	The name of the discrete parameter.
operand [**str**]	The type of constraint. It can be either "lt" for < or "gt" for >.
value [**float**]	Value of the constraint
p_sat [**float**]	If the soft constraint is set, the satisfaction probability

Returns:

None

method pdopt.data.Constraint.**get_requirement**()

Get the disequation that defines the soft constraint, if present.

Parameters:

None

Returns:

requirement [**(str, float)**]	Tuple containing the type of constraint ('lt' for < and 'gt' for >) and the value of the constraint. None if no soft constraint is present.

class pdopt.data.**DesignSet**(*input_parameter_levels, response_parameters*)

Class that represents a single design set. It contains also the optimisation problem used in the search phase of the framework.

Parameters:

input_parameter_levels [dict]	Python dictionary containing for each parameter, whose name is used as keyword, the level for this DesignSet.
response_parameters [list(str)]	List of strings containing the names of the responses (objectives and constraints).

Returns:

None

Attributes:

id [int]	Unique identifier of this design set.
parameter_levels_dict [dict]	Dictionary that mirrors the *input_parameter_levels* parameter.
parameter_levels_list [list]	Ordered Python list containing the information of *input_parameter_levels*
response_parameters [list]	Python list containing the information of *response_parameters*
is_discarded [bool]	Flag if the set has been discarded.
P [float]	Overall probability of this set, calculated in the exploration phase.
P_responses [dict]	Python dictionary containing the satisfaction probability for each constraint (whose names are used as keywords).
optimisation_problem [pdopt.data.OptimisationProblem]	Deterministic optimisation problem for this Design set.
optimisation_results [pandas.DataFrame]	DataFrame contining the optimisation results after the search phase.

method pdopt.data.DesignSet.**get_discarded_status**()

Return the discarded status of the set.

Parameters:

None

Returns:

is_discarded [bool]	Boolean if this set has been marked as discarded

method pdopt.data.DesignSet.**get_P**()

Return the total probability of the set.

Parameters:

	None

Returns:

P [**float**]	Overall probability of this set, calculated in the exploration phase.

method pdopt.data.DesignSet.**get_response_P**(*response_id=None*)

Return the probability for one of the responses of this set.

Parameters:

response_id [**str**]	Name of the response to find the calculated probability. It set to None, returns the whole list.

Returns:

response_P [**floatlist**]	Value of the selected response, or a list containing all of them.

method pdopt.data.DesignSet.**set_responses_P**(*response_name, P_response*)

Add the new response result, and updates the total probability.

Parameters:

response_name [str]	Name of the response.
P_response [**float**]	Calculated probability of the response.

Returns:

	None.

method pdopt.data.DesignSet.**set_as_discarded**()

Updates the is_discarded flag to False.

Parameters:

	None.

Returns:

	None.

method `pdopt.data.DesignSet.`**`set_optimisation_problem`**(*model, parameters, objectives, constraints, pool*)

Sets up the optimisation problem within this set.

Parameters:

model [**pdopt.data.Model**]	Evaluation function model.
parameters [**list**]	List of parameter objects.
objectives [**list**]	List of objectives objects.
constraints [**list**]	List of constraint objects.
pool [**multiprocessing.Pool**]	Python Pool for multicore support.

Returns:

None.

method `pdopt.data.DesignSet.`**`sample`**(*n_samples, parameters_list*)

Sample a n-th amount of points within the set using Latin Hypercube sampling.

Parameters:

n_samples [**int**]	Number of samples.
parameters_list [**list**]	List of parameters (Continous or Discrete).

Returns:

None.

method `pdopt.data.DesignSet.`**`get_optimum`**()

Return the pandas dataframe with the optimisation results.

Parameters:

None.

Returns:

optimisation_results [**pandas.DataFrame**]	Optimisation results in dataframe format.

class `pdopt.data.`**`OptimisationProblem`**(*model, parameters, objectives, constraints, set_levels*)

Class that represents a deterministic optimisation problem, which is contained in a DesignSet. This class wraps the `pymoo.core.problem.ElementwiseProblem` class.

Parameters:

model [**pdopt.data.Model**]	Evaluation function model.
parameters [**list**]	List of parameter objects.
objectives [**list**]	List of objectives objects.
constraints [**list**]	List of constraint objects.
set_levels [**list**]	List of levels for each parameter.

Returns:

None

Attributes:

model [**pdopt.data.Model**]	Evaluation function model.
var [**list**]	List of parameter objects.
obj [**list**]	List of objectives objects.
cst [**list**]	List of constraint objects.
l [**list**]	List of lower bound values of each parameter.
u [**list**]	List of upper bound values of each parameter.

class `pdopt.data.`**`Model`**(*model_fun*)

Class used to contain the evaluation function. It is passed and wrapped within the `run()` method.

Parameters:

model_fun [**pdopt.data.Model**]	Evaluation function model.

Returns:

None

Attributes:

None

method `pdopt.data.Model.`**`run`**`()`

This method is overloaded with the evaluation function that has been passed in the `Model()` object. Hence the parameters are the same as the evaluation function.

Parameters:

***parameters** [**list**]	List of parameters that are input of the evaluation function. These are unpacked into arguments using the Python star operator.

Returns:

response [**dict**]	Dictionary containing the value of the responses as outlined in the `response.csv` file.

class `pdopt.data.`**DesignSpace**(*csv_parameters, csv_responses*)

Class that contains the entire design space. It automatically reads the input files and constructs the sets.

Parameters:

csv_parameters [**str**]	Directory and name of the `input.csv` file.
csv_responses [**str**]	Directory and name of the `response.csv`.

Returns:

None

Attributes:

parameters [**list**]	List containing the parameter objects.
n_par [**int**]	Number of parameters.
par_names [**list**]	List containing the parameter names.
objectives [**list**]	List containing the objective objects.
constraints [**list**]	List containing the constraint objects.
obj_names [**list**]	List containing the objectives names.
con_names [**list**]	List containing the constraints names.
sets [**list**]	List containing the generated DesignSet objects.

method `pdopt.data.DesignSpace.`**get_exploration_results**`()`

Returns the results from the exploration phase.

Parameters:

None

Returns:

exploration_results [**pandas.DataFrame**]	A dataframe containing the results of the exploration phase as described in 3.6.

method `pdopt.data.DesignSpace.`**get_optimum_results**`()`

Returns the results from the search phase.

Parameters:

csv_parameters [**str**]	Directory and name of the `input.csv` file.
csv_responses [**str**]	Directory and name of the `response.csv`.

Returns:

optimisation_results [**pandas.DataFrame**]	A dataframe containing the results of the search phase as described in 3.6.

method pdopt.data.DesignSpace.**set_discard_status**(*set_id, status*)

Change the discarded status of a set within the design space. Useful for manually re-enabling some areas that have been discarded before running the search phase.

Parameters:

set_id [**int**]	Id of the set to change the discarded status.
status [**bool**]	Value to set to the discarded status.

Returns:

None

4.2 Exploration Phase (`pdopt.exploration`)

This module contains the functions and objects required to carry out the exploration analysis.

`func` `pdopt.exploration.`**`generate_surrogate_training_data`**(*parameters_list, model, n_train_points, save_dir=None*)

Generates from the evaluation function the training data for the probabilistic surrogate model, by sampling in the entire design space using latin hybercube sampling.

Parameters:

parameters_list [**list**]	List containing the parameter objects.
model [**pdopt.data.Model pdopt.data.ExtendableModel**]	Model object which contains the evaluation function.
n_train_points [**int**]	Number of training datapoints to generate.
save_dir [**str**]	Directory and filename where to save the generated data to as `.csv`. If set to None, then it is ignored.

Returns:

training_data [**pandas.DataFrame**]	Dataframe with the generated datapoints, with input and output columns as defined in the PDOPT case `.csv` files.

`func` `pdopt.exploration.`**`generate_surrogate_test_data`**(*n_points, parameters_list, model, save_dir=None*)

Generates from the evaluation function the testing data for the probabilistic surrogate model, by sampling in the entire design space using latin hybercube sampling.

Parameters:

n_points [**int**]	Number of testing datapoints to generate.
parameters_list [**list**]	List containing the parameter objects.
model [**pdopt.data.Model pdopt.data.ExtendableModel**]	Model object which contains the evaluation function.
save_dir [**str**]	Directory and filename where to save the generated data to as `.csv`. If set to None, then it is ignored.

Returns:

testing_data [**pandas.DataFrame**]	Dataframe with the generated datapoints, with input and output columns as defined in the PDOPT case `.csv` files.

`func` `pdopt.exploration.`**`generate_input_samples`**(*n_points, parameters_list, rule='lhs'*)

Auxiliary function used by the data generating functions for sampling the full design space. The default rule is Latin Hypercube sampling (lhs), but Sobol and factorial grid sampling are also available.

Parameters:

n_points [**int**]	Number of samples to generate.
parameters_list [**list**]	List containing the parameter objects.
rule [**str, default='lhs'**]	Method used for sampling. Default is Latin Hypercube 'lhs'. Set it to 'sobol' for Sobol sampling and 'grid' for full factorial sampling.

Returns:

input_samples [**numpy.ndarray**]	Numpy array containing the input samples, columns ordered as the parameters in `parameters_list`.

class `pdopt.exploration.`**`SurrogateResponse`**(*response_name, parameters_list, model, train_data=None, test_data=None*)

Class that encapsulates the Gaussian Process Regressor to be used as probabilistic surrogate model.

Parameters:

n_points [**int**]	Number of testing datapoints to generate.
parameters_list [**list**]	List containing the parameter objects.
model [**pdopt.data.Model, pdopt.data.ExtendableModel**]	Model object which contains the evaluation function.
save_dir [**str**]	Directory and filename where to save the generated data to as `.csv`. If set to None, then it is ignored.

Returns:

None

Attributes:

id [**int**]	Unique identifier of the surrogate response.
name [**str**]	Name of the surrogate response.
par_names [**list**]	List containing the parameter names.
score [**float**]	R^2 score of the trained Gaussian Process.
train_time [**float**]	Total training time.
x_scaler [**sklearn.MinMaxScaler**]	Function to normalize the input values between [0,1].
f [**sklearn.GPR**]	Trained Gaussian Process function.

method `pdopt.exploration.SurrogateResponse.`**`predict`**(*x*)

Return the mean and standard deviation of the response for a given input.

Parameters:

X [**list, numpy.ndarray**]	The input parameters where to evaluate the surrogate.

Returns:

mu [**numpy.ndarray**]	Numpy array with the expected response.
sigma [**numpy.ndarray**]	Numpy array containing the standard deviation of the expected response.

class `pdopt.exploration.`**`ProbabilisticExploration`**(*design_space, model, surrogate_training_data_file=None, n_train_points=100*)

Class for the object that performs the probabilistic exploration.

Parameters:

design_space [pdopt.data.DesignSpace]	The design space object.
model [pdopt.data.Model, pdopt.data.ExtendableModel]	Model object which contains the evaluation function.
surrogate_training_data_file [str, default=None]	Directory and filename where the generated training data is present. If not present, then it will generate it at runtime and save it there for next time use. Set it to None to avoid saving/loading any testing data and generate a new batch every run.
n_train_points [int, default=100]	Number of datapoints for training the surrogate models.

Returns:

None

Attributes:

design_space [pdopt.data.DesignSpace]	The design space object.
parameters [list]	List containing the parameter objects.
objectives [list]	List containing the objective objects.
constraints [list]	List containing the constraint objects.
run_time [float]	Total running time.
surrogate_train_data [pandas.DataFrame]	Training data (generated or loaded).
surrogate_test_data [pandas.DataFrame]	Testing data, generated on the fly.
surrogates [dict]	Python dictionary of `pdopt.exploration.SurrogateResponse` objects for each response. Keywords are the response name.

method `pdopt.exploration.ProbabilisticExploration.`**`run`**(*n_samples=100, p_discard=0.5*)

Perform the probabilistic exploration procedure.

Parameters:

n_samples [int, default=100]	The number of samples used to evaluate the probability of a design set.
p_discard [float, default=0.5]	Probability threshold under which a design set is marked as discarded.

Returns:

None.

method `pdopt.exploration.ProbabilisticExploration.`**`run_surrogate`**(*X*)

Run all the surrogate models for a given input.

Parameters:

X [**list, numpy.ndarray**]	The input parameters where to evaluate the surrogate.

Returns:

mu [**dict**]	Python dictionary containing Numpy arrays with the expected response for each surrogate. Keywords are the response name.
sigma [**dict**]	Python dictionary containing Numpy arrays with the standard deviations of the expected response for each surrogate. Keywords are the response name.

4.3 Search phase `pdopt.optimisation`

This module contains classes and functions used for the search phase.

class `pdopt.exploration.`**`Optimisation`**(*design_space, model, save_history=False, **kwargs*)

Class for the object that performs the search within the surviving design sets. Keyword arguments that can be passed are the termination criteria hyperparmeters used in the `pymoo` library, along with the population size argument of the UNSGA3 algorithm.

Parameters:

design_space [pdopt.data.DesignSpace]	The design space object.
model [pdopt.data.Model, pdopt.data.ExtendableModel]	Model object which contains the evaluation function.
save$_history[bool, default = False]$	Flag to save the optimisation history.
***kwargs**	Optional keyword arguments that are used to setup the `pymoo` hyperparameters. Most can be seen from the code itself. The most important ones are: n_max_gen for maximum number of generations, n_max_evals for maximum number of function evaluations, pop_size for GA population size and n_proc for number of processors to be used (default set to three fourths of total number of cores

Returns:

None

Attributes:

design_space [pdopt.data.DesignSpace]	The design space object.
model [pdopt.data.Model, pdopt.data.ExtendableModel]	Model object which contains the evaluation function.
valid_sets_id [list]	List containing the id of sets that have survived the exploration phase.
pool [multiprocess.Pool]	Pool object for performing the .
algorithm [unsga3]	The `pymoo` UNSGA3 algorithm implementation.
termination [MultiObjectiveDefaultTermination]	The `pymoo` MOO termination criteria implementation.

method `pdopt.exploration.Optimisation.`**`run`**`()`

Run the search in all the surviving design sets.

Parameters:

None

Returns:

None

4.4 Miscellanous tools `pdopt.tools`

Auxiliary library for miscellanous functions.

func pdopt.tools.**is_pareto_efficient**(*costs*)

Finds the Pareto front from a table of objectives.

Parameters:

costs [numpy.ndarray]	An (n_points, n_objectives) array

Returns:

is_efficient [numpy.ndarray]	A (n_points,) boolean array, indicating whether each point is Pareto efficient

func pdopt.tools.**generate_run_report**(*file_directory, design_space, optimisation, exploration*)

Generates a text report file with information regarding the time of execution and the number of discarded sets.

Parameters:

file_directory [str]	Directory where to export the `report.txt` file
design_space [pdopt.data.DesignSpace]	The design space object of the runned case.
optimisation [pdopt.optimisation.Optimisation]	The optimisation object of the runned case.
exploration [pdopt.exploration.ProbabilisticExploration]	The exploration object of the runned case.

Returns:

None

Appendix A Example Input Files

This example refers to the one available in the GitHub repository in the example folder.

A.1 input.csv

```
name,type,lb,ub,levels,uq_dist,uq_var_l,uq_var_u
climb_h0,continous,0,1,4,nan,nan,nan
climb_h1,continous,0,1,4,nan,nan,nan
cruise_h0,continous,0,1,4,nan,nan,nan
cruise_h1,continous,0,1,4,nan,nan,nan
```

A.2 response.csv

```
name,type,op,val,pSat
TOM,objective,min,nan,nan
Mf,objective,min,nan,nan
M_NOx,objective,min,nan,nan
TOM,constraint,lt,20000,0.5
```

A.3 energy_management_experiment.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 12 11:23:06 2021

PDOPT Analysis using new HEPS code, an updated version of the code from
 Dec. 2020 with a Gas Turbine Map and Boeing FuelFlow2 method for estimating
 NOx and CO emissions with data from https://doi.org/10.1016/j.trd.2018.01.019.

The mission is the reference design mission from FP50, flight from
 Edimbourgh to Dublin with alternate to Belfast, defined in the
 data/mission.csv file.

The architecture is fixed with parameters defined in the
 data/architecture.json file.

Objectives and Constraints are pulled from the TLARs of FP50, as presented
 in https://www.mdpi.com/2226-4310/8/3/61/htm.

Assumptions for this set of experiments:
    - The aircraft is retrofitted, i.e. we target a MTOM that is no larger
      than the reference aircraft (with some added margin).
    - Empty weight is assumed constant, the GT is the same as reference (PW127)
    .
    - The battery are assumed to be removable, hence we want to minimize TOM.
    - Descent phase runs on prime mover only, it is expected to have some form
      electrical storage charging (not modeled here).
    - Some flight conditions are lumped (TO and Climbout, Landing and Final).
    - Ground movements are ignored.


Architecture: Parallel (FP50 Type 2)

Shared Objective:
    - Minimize Fuel Consumption (CO2)
    - Minimize NOx

Shared Constraints, encoded also as Step 1 Requirements:
    - TOM < 20000 kg (MTOM)

```

```python
@author: Andrea Spinelli

This file contains the Experiment definition and shared data
which is imported in each individual file that runs the tests.
"""

import json
import sys
import pickle as pk
import argparse

from os.path import exists

import pandas as pd
import numpy as np
from tqdm import tqdm

from pdopt.data import DesignSpace, ExtendableModel
from pdopt.exploration import  ProbabilisticExploration
from pdopt.optimisation import Optimisation
from pdopt.tools import generate_run_report
#from pdopt.visualisation import main_inline

from HE_Model import model, postpro_run


class Experiment(ExtendableModel):

    def __init__(self, input_parameters, architecture, mission_file):
        self.inp        = list(pd.read_csv(input_parameters)['name'])
        self.arch       = architecture
        self.arch0      = architecture
        self.mission    = mission_file

    def run(self, *args, **kwargs):
        # The input of the model is variable
        # we need to construct the energy management dataframe

        X, parms = [], []
        en_mgm = []

        # The TO/LND conditions will be single point only
        # logic to convert TO/LND into takeoff, climbout, final, landing
        for i in range(len(self.inp)):

            if 'TO' in self.inp[i]:
                parms.append(self.inp[i].replace('TO','takeoff'))
                parms.append(self.inp[i].replace('TO','climbout'))
                X.append(args[i])
                X.append(args[i])

            elif 'LND' in self.inp[i]:
                parms.append(self.inp[i].replace('LND','final'))
                X.append(args[i])

            elif 'LNDToAlternate' in self.inp[i]:
                parms.append(self.inp[i].replace('LND','final'))
                parms.append(self.inp[i].replace('LNDToAlternate','landing'))
                X.append(args[i])
                X.append(args[i])

            else:
                parms.append(self.inp[i])
                X.append(args[i])


        # Stuff to introduce constraints over the x_positions
        all_x_vals = []
        x_vals = []
        old_seg = None

        for i in range(len(parms)):
            # separate segment from type
            segment, value = parms[i].split('_')

            if old_seg != segment and len(x_vals) > 0:
                all_x_vals.append(x_vals)

            x_vals = [] if old_seg != segment else x_vals

            if value[0] == 'h':
                if len(en_mgm) == 0 or en_mgm[-1][0] != segment:
                    en_mgm.append([segment, 0, X[i]])
                else:
                    en_mgm.append([segment, 1, X[i]])
            elif value[0] == 'x':
                x_vals.append(X[i])
                en_mgm[-1][1] = X[i]

            old_seg = segment
        all_x_vals.append(x_vals)


        en_management = pd.DataFrame(en_mgm, columns=['segment','x','doh'])

        #For doing UQ on architecture parameters
        checks = [
            'e_bat' in kwargs.keys(),
            'motor' in kwargs.keys(),
            'power_el' in kwargs.keys(),
            'cables' in kwargs.keys(),
            'battery' in kwargs.keys()
            ]

        if any(checks):

            if 'e_bat' in kwargs.keys():
                self.arch['e_bat'] = kwargs['e_bat']

            if 'motor' in kwargs.keys():
                self.arch['eta_e_comp']['motor'] = kwargs['motor']
                self.arch['eta_e'] = np.prod(
                    list(self.arch['eta_e_comp'].values())
                    )

            if 'power_el' in kwargs.keys():
                self.arch['eta_e_comp']['power_el'] = kwargs['power_el']
                self.arch['eta_e'] = np.prod(
                    list(self.arch['eta_e_comp'].values())
                    )

            if 'cables' in kwargs.keys():
                self.arch['eta_e_comp']['cables'] = kwargs['cables']
                self.arch['eta_e'] = np.prod(
                    list(self.arch['eta_e_comp'].values())
                    )

            if 'battery' in kwargs.keys():
                self.arch['eta_e_comp']['battery'] = kwargs['battery']
                self.arch['eta_e'] = np.prod(
                    list(self.arch['eta_e_comp'].values())
                    )

        else:
            self.arch = self.arch0.copy()




        analysis = model(en_management, architecture_data=self.arch,
    mission_file=self.mission)

        output = {
                'TOM'       : analysis.iloc[-1].mass,
                'Mf'        : analysis.iloc[-1].m_fl,
                'M_NOx'     : analysis.iloc[-1].m_NOx,
                 }

        # Add constraints over the position of the segments, x2 - x1 < 0
        if len(all_x_vals) > 0:
            counter = 1
            for x_vals in all_x_vals:
                for i in range(1,len(x_vals)):
                    output.update({f'x{counter}':x_vals[i-1]-x_vals[i]})
                    counter += 1

        return output

    def postprocess_analysis(self, *args, **kwargs):
        # The input of the model is variable
        # we need to construct the energy management dataframe

        X, parms = [], []
        en_mgm = []

        # The TO/LND conditions will be single point only
        # logic to convert TO/LND into takeoff, climbout, final, landing
        for i in range(len(self.inp)):
            if 'TO' in self.inp[i]:
                parms.append(self.inp[i].replace('TO','takeoff'))
                parms.append(self.inp[i].replace('TO','climbout'))
                X.append(args[i])
                X.append(args[i])

            elif 'LND' in self.inp[i]:
                parms.append(self.inp[i].replace('LND','final'))
                X.append(args[i])

            elif 'LNDToAlternate' in self.inp[i]:
                parms.append(self.inp[i].replace('LND','final'))
                parms.append(self.inp[i].replace('LNDToAlternate','landing'))
                X.append(args[i])
                X.append(args[i])

            else:
                parms.append(self.inp[i])
                X.append(args[i])


        for i in range(len(parms)):
            # separate segment from type
            segment, value = parms[i].split('_')

            if value[0] == 'h':
                if len(en_mgm) == 0 or en_mgm[-1][0] != segment:
                    en_mgm.append([segment, 0, X[i]])
                else:
                    en_mgm.append([segment, 1, X[i]])
            elif value[0] == 'x':
                en_mgm[-1][1] = X[i]


        en_management = pd.DataFrame(en_mgm, columns=['segment','x','doh'])

        #For doing UQ on battery energy density
        if 'e_bat' in kwargs.keys():
            self.arch['e_bat'] = kwargs['e_bat']
        else:
            self.arch = self.arch0.copy()


        analysis = model(en_management, architecture_data=self.arch,
    mission_file=self.mission)

        results = postpro_run(self.arch, analysis)

        return results


def run_experiment(folder, n_exp_samples, P_exploration, restart, n_exp_train):
    print('Input Args: ', folder, n_exp_samples, P_exploration, restart,
    n_exp_train)

    architecture = json.load(open('data/architecture.json', 'r'))
    mission = 'data/mission_original.csv'
    experiment = Experiment(folder + '/input.csv', architecture, mission)


    # Check if a design space is already present otherwise create it
    if exists(folder + '/design_space.pk') and restart:
        design_space = pk.load(open(folder + '/design_space.pk','rb'))
    else:
        design_space = DesignSpace(folder + '/input.csv', folder + '/response.
    csv')
        pk.dump(design_space, open(folder + '/design_space.pk','wb'))

    # Check if there is already a trained exploration object
    if exists(folder + '/exploration.pk') and restart:
        exploration = pk.load(open(folder + '/exploration.pk','rb'))
    else:
        exploration = ProbabilisticExploration(design_space, experiment,
                                           surrogate_training_data_file=folder
    + '/samples.csv',
                                           n_train_points=n_exp_train)

        for k in exploration.surrogates:
            s = exploration.surrogates[k]
            print(f'Surrogate {s.name} with r = {s.score:.4f}')

        pk.dump(exploration, open(folder + '/exploration.pk','wb'))


    # Check if exploration has been done already
    if exists(folder + '/exp_results.csv') and restart:
        pass
    else:
        exploration.run(n_exp_samples, P_exploration)
        design_space.get_exploration_results().to_csv(folder + '/exp_results.
    csv', index=False)

        #Update the saved design object
        pk.dump(design_space, open(folder + '/design_space.pk','wb'))


    optimisation = Optimisation(design_space, experiment, n_max_evals=2000)


    # Check if optimisation has been done already
    if exists(folder + '/opt_results_raw.csv') and restart:
        pass
    else:
        optimisation.run()
        df_opt = design_space.get_optimum_results()
        df_opt.to_csv(folder + '/opt_results_raw.csv', index=False)

        #Update the saved design object
        pk.dump(design_space, open(folder + '/design_space.pk','wb'))

    if exists(folder + '/opt_results.csv') and restart:
```

```python
        pass
    else:
        inp_pars = design_space.par_names #pd.read_csv(inp_files[i])['name'].tolist()

        df_opt['M_batt'] = 0
        df_opt['eff']    = 0
        df_opt['M_CO']   = 0
        df_opt['M_CO2']  = 0
        df_opt['eff_GT_cl'] = 0
        df_opt['eff_GT_cr'] = 0
        df_opt['eff_cl'] = 0
        df_opt['eff_cr'] = 0

        for index in tqdm(range(len(df_opt))):
            t_out = experiment.postprocess_analysis(*df_opt.loc[index, inp_pars].tolist())

            df_opt.loc[index, 'M_batt'] = t_out.iloc[-1].m_bat
            df_opt.loc[index, 'eff'] = t_out.iloc[-1].eff
            df_opt.loc[index, 'M_CO'] = t_out.iloc[-1].m_CO
            df_opt.loc[index, 'M_CO2'] = t_out.iloc[-1].m_fl * 3
            df_opt.loc[index, 'eff_GT_cl'] = t_out.loc[t_out['tag'] == 'climb'].P_GT_out.sum() / t_out.loc[t_out['tag'] == 'climb'].P_GT_in.sum()
            df_opt.loc[index, 'eff_GT_cr'] = t_out.loc[t_out['tag'] == 'cruise'].P_GT_out.sum() / t_out.loc[t_out['tag'] == 'cruise'].P_GT_in.sum()
            df_opt.loc[index, 'eff_cl'] = t_out.loc[t_out['tag'] == 'climb'].P_req.sum() / t_out.loc[t_out['tag'] == 'climb'].P_tot.sum()
            df_opt.loc[index, 'eff_cr'] = t_out.loc[t_out['tag'] == 'cruise'].P_req.sum() / t_out.loc[t_out['tag'] == 'cruise'].P_tot.sum()

            t_out.to_csv(folder + f'/missions/{index}_mission_output.csv')

        df_opt.to_csv(folder + '/opt_results.csv', index=False)

    #Runtime Report
    generate_run_report(folder + '/report.txt', design_space, optimisation, exploration)

if __name__ == '__main__':
    # run experiment with the input set inside the file
    case_folder = "test_case_linear"
    P_sat = 0.5
    n_exp_samples   = 100
    n_train_samples = 100
    restart = False

    run_experiment(case_folder, n_exp_samples, P_sat, restart, n_train_samples)
```

Bibliography

[1] Julian Blank and Kalyanmoy Deb. "A Running Performance Metric and Termination Criterion for Evaluating Evolutionary Multi- and Many-objective Optimization Algorithms". In: *2020 IEEE Congress on Evolutionary Computation (CEC)*. 2020, pp. 1–8. DOI: `10.1109/CEC48606.2020.9185546`.

[2] Julian Blank and Kalyanmoy Deb. "Pymoo: Multi-Objective Optimization in Python". In: *IEEE Access* 8 (2020), pp. 89497–89509. DOI: `10.1109/ACCESS.2020.2990567`.

[3] Julian Blank and Kalyanmoy Deb. *Pymoo: Multi-Objective Optimization in Python Documentation.* 2020. URL: `https://pymoo.org/index.html` (visited on 07/15/2022).

[4] Alex Georgiades et al. "ADOPT: An augmented set-based design framework with optimisation". In: *Design Science* 5 (2019). DOI: `10.1017/dsj.2019.1`.

[5] A. Inselberg and B. Dimsdale. "Parallel coordinates: a tool for visualizing multidimensional geometry". In: *Proceedings of the First IEEE Conference on Visualization: Visualization '90.* 1990, pp. 361–378. DOI: `10.1109/VISUAL.1990.146402`.

[6] T. Kipouros et al. "Parallel Coordinates in Computational Engineering Design". In: *54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference.* DOI: `10.2514/6.2013-1750`.

[7] Haitham Seada and Kalyanmoy Deb. "A Unified Evolutionary Optimization Procedure for Single, Multiple, and Many Objectives". In: *IEEE Transactions on Evolutionary Computation* 20.3 (2016), pp. 358–369. DOI: `10.1109/TEVC.2015.2459718`.

[8] Andrea Spinelli et al. "Application of Probabilistic principles to Set-Based Design for the optimisation of a hybrid-electric propulsion system". In: *IOP Conference Series: Materials Science and Engineering.* Vol. 1226. 1. IOP Publishing. 2022, p. 012064. DOI: `10.1088/1757-899X/1226/1/012064`.

[9] Andrea Spinelli et al. "Application of Probabilistic Set-Based Design Exploration on the Energy Management of a Hybrid-Electric Aircraft". In: *Aerospace* 9.3 (2022), p. 147. DOI: `10.3390/aerospace9030147`.

Index

ADOPT, 4

constraint, 5
constraints, 13

evaluation function, 8, 14
exploration, 3, 4, 15

input.csv, 7, 13
installation, 3

objectives, 13
optimisation, 6
output, 10, 18

pdopt.data.**Constraint**, 24
pdopt.data.**ContinousParameter**, 21
pdopt.data.**DesignSet**, 25
pdopt.data.**DesignSpace**, 29
pdopt.data.**DiscreteParameter**, 23
pdopt.data.**Model**, 28
pdopt.data.**Objective**, 23
pdopt.data.**OptimisationProblem**, 28
pdopt.exploration.**generate_input_samples**, 31
pdopt.exploration.**generate_surrogate_test_data**, 31
pdopt.exploration.**generate_surrogate_training_data**, 31
pdopt.exploration.**Optimisation**, 35
pdopt.exploration.**ProbabilisticExploration**, 33
pdopt.exploration.**SurrogateResponse**, 32
problem definition, 12

requirements, 3
response.csv, 8, 14
responses.csv, 13
runfile definition, 15
runfile layout, 9

screen output, 17
search, 3, 6, 16
soft constraint, 4, 5