The Probabilistic Design and OPTimization Framework (PDOPT)

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

October 13, 2023

* 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

1.1 General Description

PDOPT, standing for Probabilistic Design and OPTimisation, is a Python library developed at Cranfield University for the design space exploration of complex systems. It adopts a set-based design approach for exploring the design space, aiding in rapidly identifying the configurations best suited for the requirements set by the user. The mapping process does not require additional expertise as it relies on a probabilistic surrogate model in identifying the areas of the design space with the highest probability of satisfying the requirements. As a consequence, the uncertainty of the design is reduced, and exploration is significantly sped up, up to 80%. The individual design points are then recovered using local optimisation problems.

PDOPT requires the user to provide a numerical model of the system under design, which describes the relationship between the input parameters and the quantities of interest (QoI), subject to constraints or optimisation objectives. The framework uses this model for training the probabilistic surrogate and carrying out local optimisation in the identified pockets of the design space. With the model ready, the user specifies the constraints and objectives over the QOIs and the number of levels for each design parameter.

To better explain the workflow, let's analyse the problem of designing a wooden pencil. The design parameters of a wooden pencil are its length, the shell's thickness, the lead's thickness and the mixture of graphite/kaolin. QoIs could be weight, cost, and average use life. Requirements defined over these QoIs can be as minimum cost and weight as possible, with a use life greater than a specified amount. While minimisation/maximisation objectives don't have any bounds, top-level requirements often 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 find a design that can match the market's "state of the art" and possibly improve it.

With the design problem framed this way, PDOPT allows to explore of the different combinations of input parameters, defined as areas of the design space or set and assigns 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 with a high probability of satisfying the requirements. The user can also visualise this step before the optimisation step and understand 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 identify areas of the design space that would be less sensitive to operational or market changes down the conceptual design phase.

The theoretical background behind PDOPT can be found in the author's dissertation [11]. Details on the implementation can be found in published literature [13].

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`
- `joblib`
- `tqdm`

An install script is provided to simply this process. Use the following command in the source folder: `python setup.py`. Optionally, the library `plotly` can be installed for using the prottypical interactive HTML visualisation enviroment provided in the `visualisation.py` script.

1.3 Methodology and Flowchart of the Programme

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 [12]. Figure 1.1 presents the general process of the improved methodology.

Figure 1.2 presents the flow chart of the code, with the section references. The PDOPT library is composed of three objects: the `DesignSpace()` data structure which encapsulates the design space divided into sets; the `ProbabilisticExploration()` object which implements the Exploration step; and the `Optimisation()` object implementing the Search step. Two input files define the input parameters and the QoIs affected by the constraints and objectives. The information flows from the `DesignSpace()` to first the Exploration phase and then to the Search phase, returning the results as Pandas data frames back to the `DesignSpace()` object. These results are then exported to the output files using the Pandas export interface.

The design of the PDOPT framework as a Python library grants a high level of flexibility in its application. The two steps can be run asynchronously and independently of each other. For instance, multiple exploration phases can be run under different requirements, for example, to study their effect on the design space. This chapter will present a canonical example, but the user can experiment with the tools provided.

In addition, `ProbabilisticExploration()` supports reading sampled data from other sources using the `samples.csv` file. This usually is automatically generated from the provided evaluation function. The user can edit it to introduce data from experiments or higher-fidelity simulations. Similarly, `Optimisation()` supports two forms of surrogate optimisation: Gaussian Process Regression and Neural Networks. The user can activate these if the original evaluation function is computationally too expensive or presents instabilities in some areas of the design space.

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.

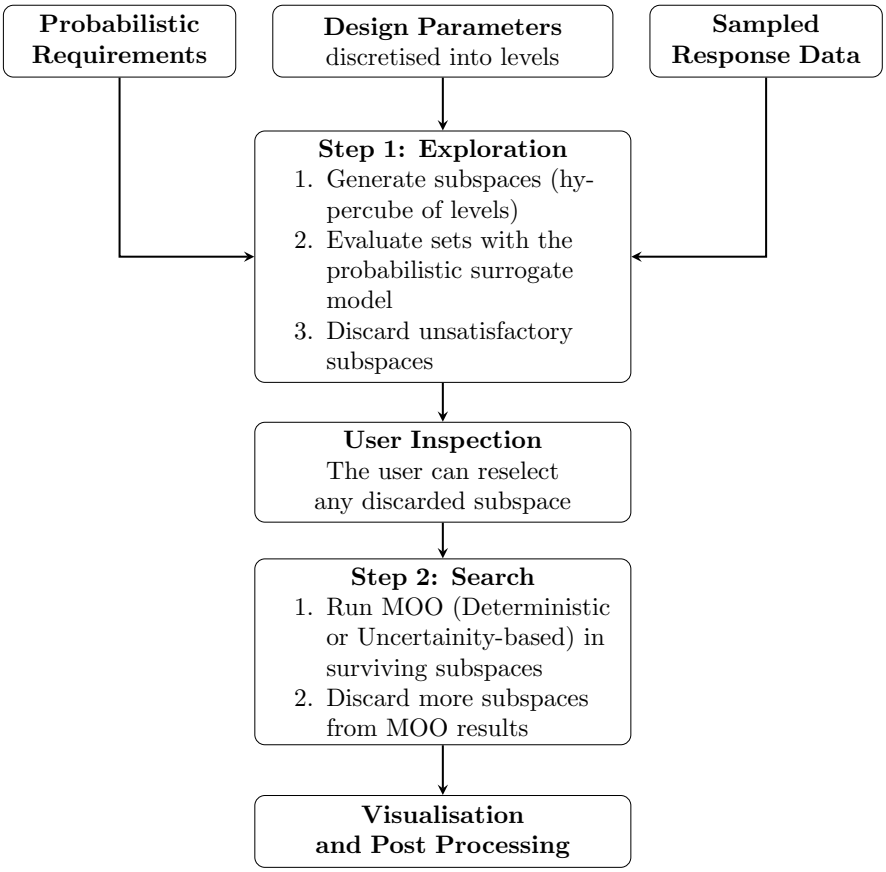

Figure 1.1: Methodology Flowchart.

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.

The computational process comprises three main steps: training the required surrogate models, sampling the sets, evaluating the points, and calculating the total probability of the set. Figure 1.4 shows this process in a flowchart.

Training of the surrogate processes begins with sampling the entire design space to generate the training dataset. The quasi-random Sobol sequence is used, with 128 points, to ensure optimal filling [7]. For the validation dataset, Latin Hypercube Sampling (LHS) [9] is used instead, with 32 points. The `joblib` [6] parallelisation library is used to speed up the process,

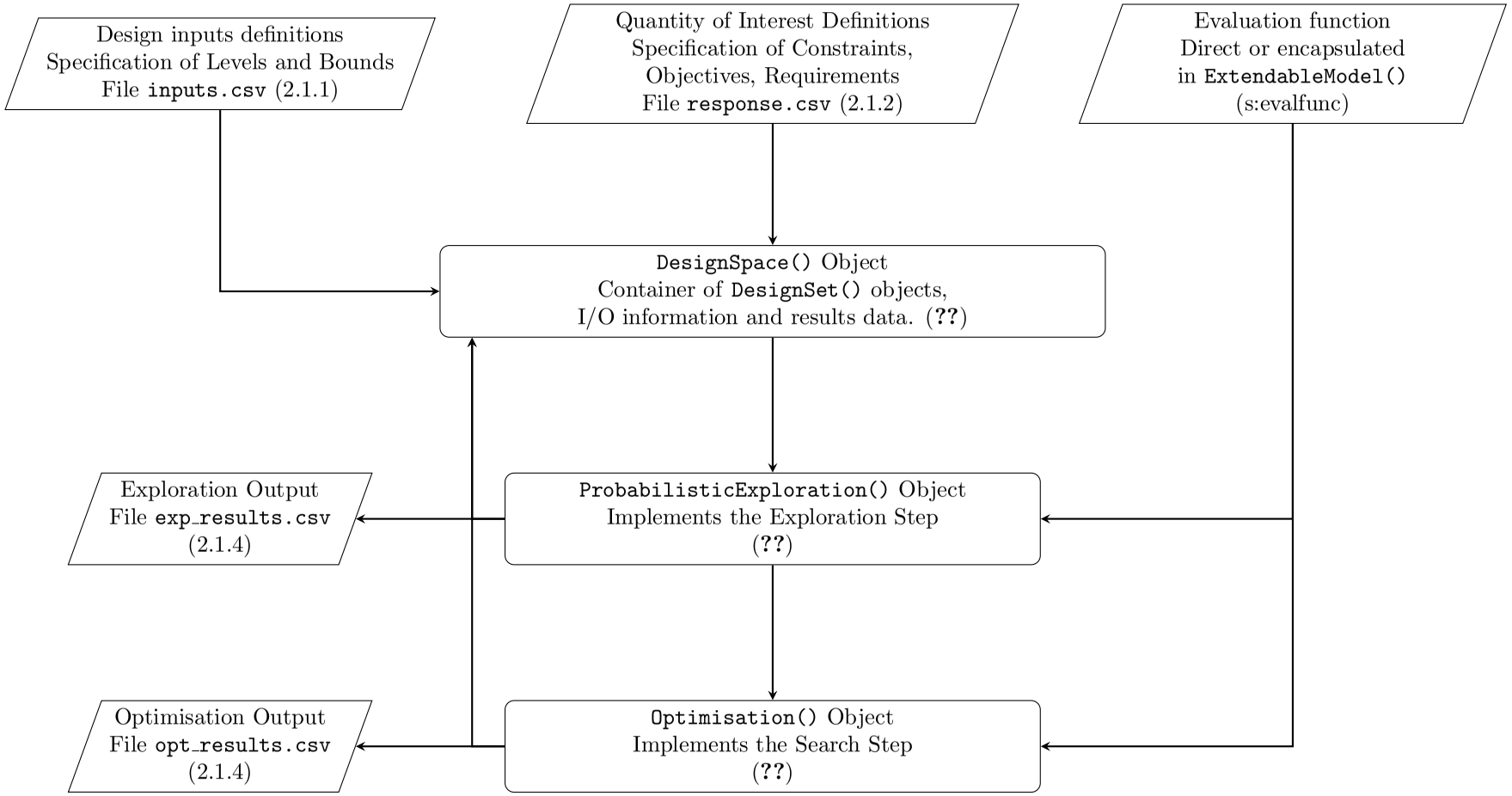

Figure 1.2: Computational flowchart of the PDOPT framework.

and the training dataset is stored in the `samples.csv` file for future use. The data is scaled to the $[0, 1]$ range to improve the convergence of the training algorithm. The inputs and outputs are automatically scaled when calling the surrogate model for prediction.

The Gaussian Process implementation is from the `scikit-learn` machine learning library [10], which also contains dataset normalisation and model validation tools. After training, the user can inspect from the console output the Pearson R^2 correlation coefficient to evaluate the quality of the model. GPs are single output functions; hence one is trained per QoI. The kernel adopted is the Matern kernel with the standard coefficients of `scikit-learn`. Matern can handle more irregular data than the commonly used radial-basis kernel.

Once the GPs are trained, the probabilistic evaluation can start. Each set j is sampled with LHS, generating N_j of candidate points X_k^j (100 by default). For each constraint i, these are passed through the corresponding GP function, generating the mean value $\mu_{y,i}^k$ and variance $\sigma_{y,i}^k$ at the sample point of the QoI y_i. Given the constraint boundary $\bar{g}_i$, it is possible to calculate the probability $P^k(y_i < \bar{g}_i)$ that it is satisfied at the sampled point using the Gaussian cumulative distribution function Φ:

$$P^k(y_i < \overline{g_i}) = \Phi\left(\frac{\overline{g_i} - \mu_i^k}{\sigma_i^k}\right) \tag{1.1}$$

The equation calculates the area under the Gaussian probability distribution curve from $-\infty$ to the constraint value $\overline{g_i}$. Figure 1.3 shows this process. In case of a "greater than" constraint, the function's argument Φ in Eq. 1.1. The framework automatically performs the necessary algebraic manipulations.

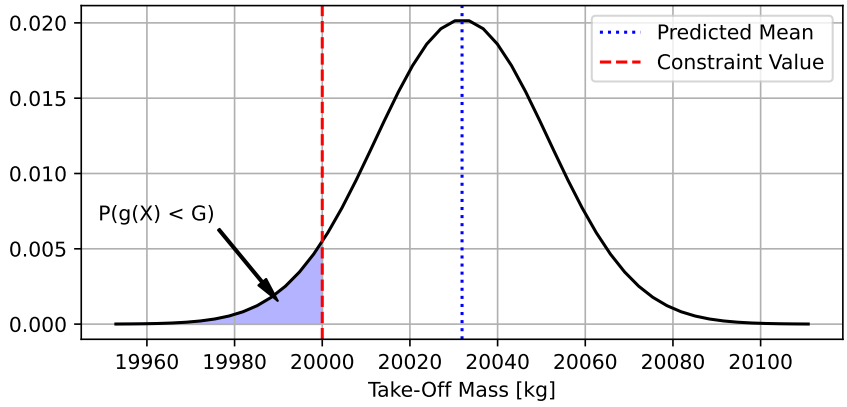

Figure 1.3: Probability of satisfying a constraint $\overline{g_i}$ for a sampled point X_k^j.

The set probability of the requirement $P_{i,j}$ is calculated using a Monte Carlo approximation. In the definition of the problem, each probabilistic constraint is defined with a minimum satisfaction probability $\bar{P}_i$. This quantity then is compared to the probability of each sampled point, using Eq. **??**. $P_{i,j}$ is finally calculated by counting how many of the sampled points satisfy the constraint ($n_{i,sat}$) and dividing it by the total number of sampled points:

$$P_{i,j} = \frac{n_{i,sat}}{N_{samples}} \tag{1.2}$$

Finally, the total set probability is calculated by multiplying all the requirement probabilities. The value is normalised by a factor k such that the probabilities of all sets are rescaled between 0 and 1:

$$P_j = \frac{1}{k} \prod_{i=0}^{N_{constr}} P_{i,j} \tag{1.3}$$

Sets are discarded if the total probability falls below the required minimum. The user specifies this value at the time of execution and is independent of $\bar{P}_i$ of the requirements. Instead, it acts as a filter for the fuzziness of the constraint boundary. Setting it closer to 1 would produce a more crisp division but might eliminate sets partially crossed by the constraint boundary, potentially losing feasible data points.

After the exploration phase, the user can retrieve the results in a `pandas.DataFrame` using the method `DesignSpace.get_exploration_results()`.

$$P(A < k_A) \geq \overline{P_A} \tag{1.4}$$

$$P^k(A < k_A) = \Phi\left(\frac{k_A - \mu_A^k}{\sigma_A^k}\right) \tag{1.5}$$

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 1.4 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.

1.5 Search Phase

The search phase is tasked with exploring the surviving sets with an optimisation algorithm to extract the individual design points. In this context, optimality is used for identifying the trade-off between the objectives and understand the impact of the input paramters on them.

The Python library `pymoo` [**pymoo**] is adopted for this process. This library implements the interface for defining a multiobjective optimisation problem and popular and robust algorithms to solve them. For the scopes of PDOPT, the algorithm should be population-based but capable of handling problems spanning from single-objective to many-objective. Additionally, it should be gradient-free, as the evaluation function may not be differentiable, which is the typical case for multidisciplinary models. Hence the Universal Non-dominated Sorting Genetic Algorithm III (U-NSGA-III) was selected [**seada2016unsga3**].

The drawback of using population-based methods is the large number of evaluations of the objective functions, which grows significantly with the number of input parameters. For this reason, PDOPT presents the option to use a locally trained surrogate model when performing the set optimisation. Two types of models are supported: Gaussian Processes (Kriging) and Neural Networks. It can be activated by passing the parameter `use_surrogate=True` to the `Optimisation()` object. The algorithm recovers the actual output from the found Pareto set after the optimisation cycle when using the surrogate models.

Both are non-parametric for guaranteeing generality. The training process follows the same procedure for training GPs in the Exploration phase (sec. **??**): the set is sampled using the Sobol sequence for the training data and LHS for the validation data. Scaling is also introduced for better convergence.

The kernel of the Kriging model is fixed, a combination of Radial Basis Function (RBF) kernel with two constants:

$$k(x_i, x_j) = a \cdot exp\left(\frac{d(x_i, x_j)}{l^2}\right) + b \tag{1.6}$$

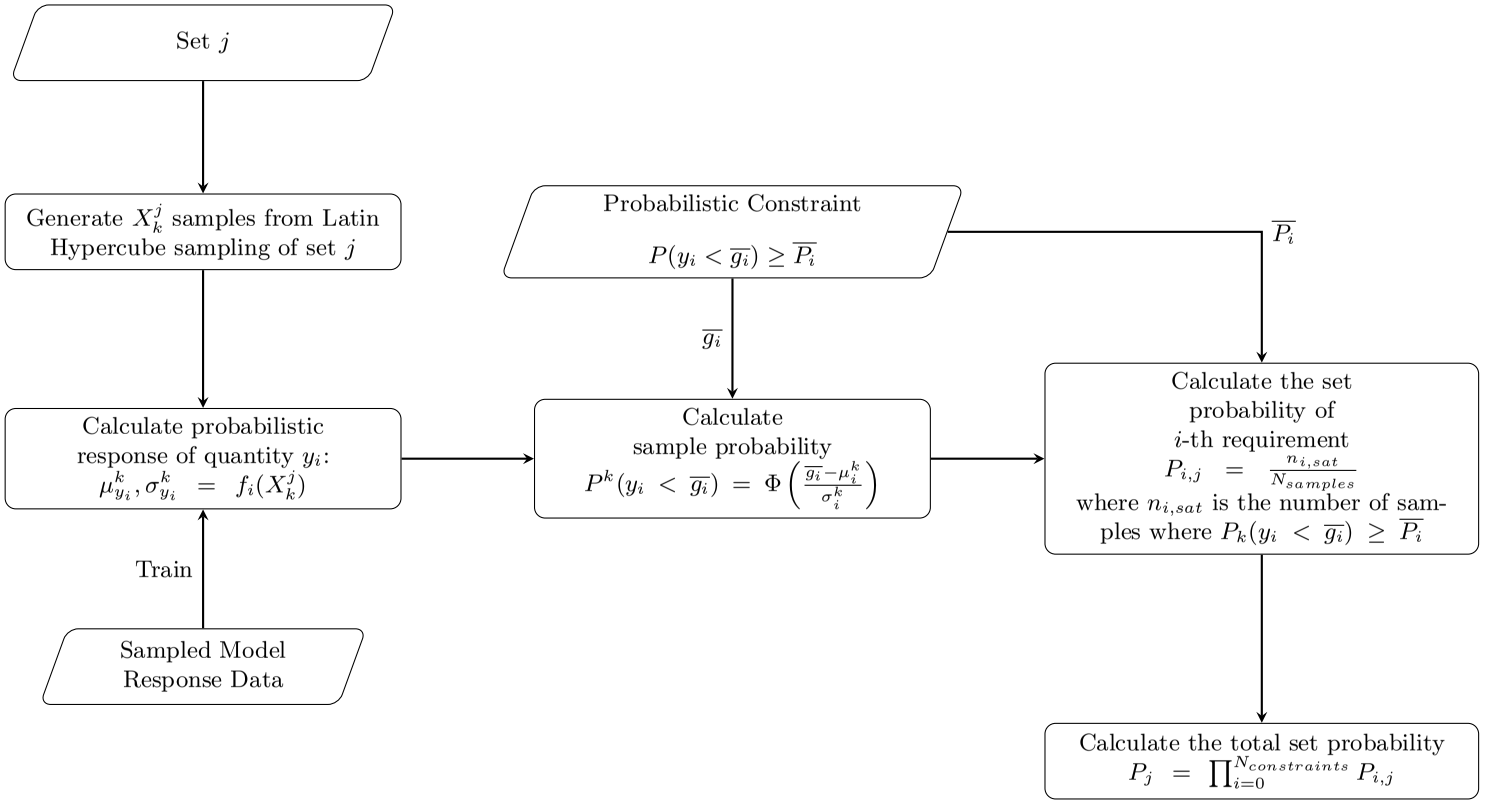

Figure 1.4: Detail of the Exploration phase process

where a, b are constants, l is the RBF length scale and $d(x_i, x_j)$ is the Euclidean distance of the two points. During training, an optimiser finds the best combination of hyperparameters minimising the regression error. RBF has been preferred in this application as the function response is locally smooth for most problems.

On the other hand, the structure of the NN model is parametrised and tuned using a randomised search method. The algorithm generates 100 random candidates with hidden layers ranging from 1 to 4 and a number of neurons ranging from 50 to 200. The search finds the combination that minimises the mean squared error of the validation dataset.

It has been observed that the Kriging model generally has the lowest mean square error, and it should be preferred in most applications. It is the default model when `use_surrogate=True` is passed. The Neural Network model may be helpful if the response surface presents strong discontinuities that cannot be captured with an RBF kernel. If the user prefers to use the NN model, the parameter `use_NN` should be set to `True` along with the parameter activating the surrogate modelling.

The general process of the search phase is as follows. First, in each surviving set, the optimisation problem is set up. Set bounds are converted into bounds for the continuous parameters, while the discrete ones are held constant, as the value is uniform over the whole set. Constraints are converted into the standard form $G(x) < 0$, since `pymoo` evaluates constraint violations if the constrained value is greater than 0. After the setup, each problem is run sequentially. The optimiser uses the `joblib` library to parallelise the function evaluations at each iteration. The results of each problem are stored inside the `DesignSpace()` object in case the programme is interrupted before completion. As mentioned above, if a surrogate model is selected, the code automatically samples and trains the surrogate before running the optimisation problem. Finally, at the end of the process, the results are extracted from each set and aggregated into a single `pandas.DataFrame`, accessible with the method `DesignSpace.get_optimum_data()`.

Name	Description	Default Value
pop_size	Number of population points	10 + Number of Reference Directions (calculated with the Das-Dennis method)
n_partitions	Number of partitions of Pareto front	12
x_tol	Input parameter convergence tolerance	$1 \cdot 10^{-16}$
cv_tol	Constraint violation convergence tolerance	$1 \cdot 10^{-16}$
f_tol	Functional evaluation convergence tolerance	$1 \cdot 10^{-16}$
n_max_gen	Maximum number of generations	1000
n_max_evals	Maximum number of function evaluations	$1 \cdot 10^{6}$

Table 1.1: Optimisation `kwarg` parameters.

It is possible to control the UNSGA-III parameters by passing Python `kwarg` arguments to the `Optimisation()` object. These are shown in Tab. 1.1 along with their default value. The convergence tolerances are set to a very small number to disable them effectively. The user should instead use a maximum number of generations or function evaluations to set the stopping criterion to guarantee a fair comparison between the set results.

2 Definition of a PDOPT run: files and code structure.

This chapter describes the components of the Python library and the required files to set up a basic design space exploration analysis.

The framework was developed with generic I/O to integrate it with decision-making or visualisation tools. However, a bare-bones visualisation web tool is included in the software distribution in the `visualisation.py` script. It uses the `plotly` and `dash` Python libraries to interactively visualise any.csv file outputted by PDOPT. A screenshot of the tool is shown in Fig. 2.1.

The user is presented with a Parallel Coordinates plot and five scatter plots. It is possible to select which dimension of the dataset to use for the XY axes or the colour scheme. Furthermore, selecting a set of points within the Parallel Coordinates and propagating them to the scatter plot is possible. Interactivity is vital to effective decision-making when dealing with complex datasets. Although this tool is helpful in this context, the library `matplotib` was used instead to generate the static images used in this dissertation.

2.1 Problem Definition: Files and Code Structure

PDOPT is primarily a Python library. It is designed to be stand-alone inside a test case script or incorporated into larger frameworks. The minimum amount of files required to perform a PDOPT analysis successfully is:

- A Python script to import the PDOPT library, define the evaluation function, and set up the workflow.
- A .csv input file describing the input parameters (`inputs.csv`).
- A .csv response file, describing the QoI (`response.csv`).

The structure of the script run file follows the two-step procedure outlined in Fig. 1.1. The components required to be imported from the `pdopt` library are:

```
from pdopt.data import DesignSpace, Model
from pdopt.exploration import ProbabilisticExploration
from pdopt.optimisation import Optimisation
```

As described before, `DesignSpace()` is the data object containing all the information about the problem (parameters, QoI, sets). In contrast, the other two objects implement the exploration and search phases. Given the required input files, a minimal workable example is shown in Listing 2.1, assuming the evaluation function is defined as `my_model()`.

```
# Wrapping the evaluatio function into a model object
my_model = Model(my_eval_fun)

# Definition of the design space
design_space = DesignSpace("input.csv", "responses.csv")

# Creation of the exploration phase object
exploration  = ProbabilisticExploration(design_space, my_model)

# Run exploration phase
n_exploration_samples = 100
P_exploration_sat = 0.5
exploration.run(n_exploration_samples, P_exploration)

# Get exploration results
```

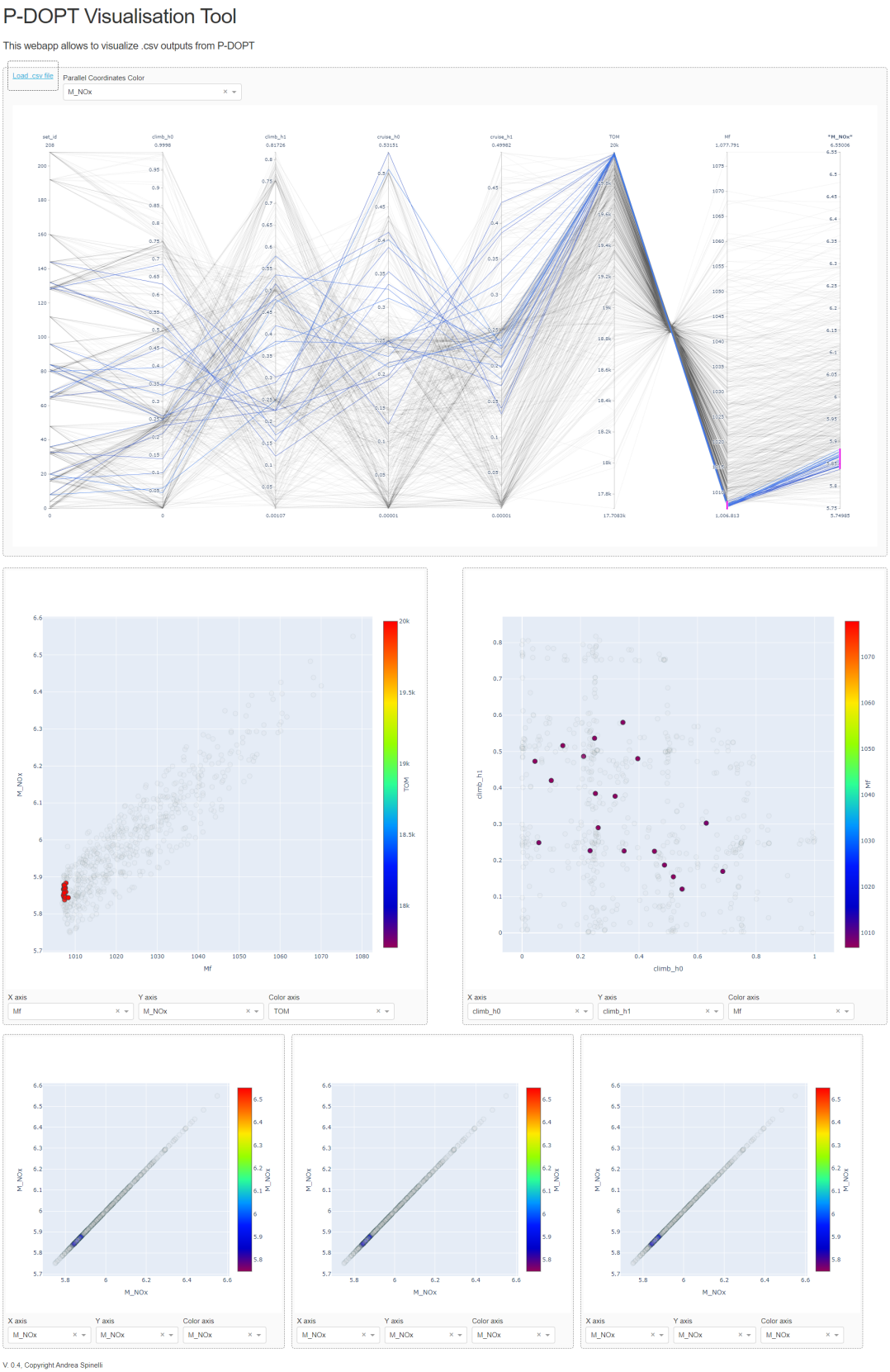

Figure 2.1: Visualisation Tool Example

```
df_exp_results = design_space.get_exploration_results()

# Creation of search phase object
optimisation = Optimisation(design_space, my_model)

# Run search phase
optimisation.run()

# Get search phase results
df_opt_results = design_space.get_optimum_results()
```

2.1.1 Definition of Input Parameters

The input parameters are defined in an ASCII .csv file (`input.csv`), one row per parameter and heading as shown in Listing 2.1. While any text editor can edit the file, a spreadsheet programme can be used to better visualise the tabular layout, as shown in Fig. 2.2.

```
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
```

Listing 2.1: Input 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 2.2: Content of the input.csv file

The input parameters are distinguished as continuous and discrete. Discrete parameters are defined by N integers, mapped in the evaluation function to discrete possibilities, while continuous parameters represent a range of possible variables. The file is structured as follows:

- **name** – Name of the parameter. It must not contain spaces; use an underscore for it.
- **type** – If the variable is continuous or discrete. If discrete, the 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** – The number of parameter levels: it must be an integer.
- **uq_dist** – Type of UQ distribution for this parameter for uncertainty-based optimisation. It can be 'uniform', 'triangular' (triangular), or '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 the decimal percentile variation. Set it to 'nan' if the distribution is symmetric.

Each row in the .csv file must be filled in. If a variable is unused, it must be set to 'nan'. Due to the implementation of the framework, it is fundamental that the input parameters in the `input.csv` file are ordered in the same order as the evaluation function arguments. During

any functional evaluation, PDOPT passes an array of inputs in the specified order assuming that it matches the expected order of the evaluation function arguments.

Some rows are used to specify an uncertainty quantification distribution to be used in uncertainty-based optimisation. While the bare bones are present in the current version of the codebase, it is not yet fully implemented. Hence, these rows are not used other than to set up the `ContinousParameter()` objects.

2.1.2 Definition of Requirements, Constraints, and Objectives)

The `responses.csv` file is used for defining the QoIs of the evaluation function that are taken into account by PDOPT. Listing 2.2 shows an example of its content. Much like input.csv, a spreadsheet programme can be used to assist with data entry and visualise the contents in a table format (fig:resp).

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

Listing 2.2: Responses 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 2.3: Content of the response.csv file

Unlike the input definition, the order of the entries does not affect the code's functionality. Instead, the user must use a unique name for each QoI: these have to match the keyword used in the return Python dictionary of the evaluation function (see 2.1.3). At least one entry per type (objective and constraint) must be present for the code to function, and every column must be filled. The file is structured as such:

- **name** – Name of the response from the evaluation function. These names must be used in the dict() object returned from the evaluation function.
- **type** – Type of response. It can be 'constraint' or 'objective'. These are the responses that the optimisation step will handle. 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 the type is set to 'constraint', the operator will be 'lt' (less than) or 'gt' (greater than), which represents the inequality of the response with respect to the quantity in the value column (that is, 'TOM, constraint, lt, 2000' corresponds to TOM ¡ 20000). If the type is set to 'objective', the operator is 'min' (minimise) or 'max' (maximise).
- **val** – Value which is going to be used by the operator. In the case of an objective, this value is used to set a constraint in the exploration phase with the following criteria: if 'min' is set as the operator, then it is less than the constraint; if 'max' is set as the operator, then it is greater than the constraint. This is done to drive the exploration phase and to 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 to which samples are tested. Samples that go under pSat are counted as unsatisfactory.

2.1.3 The evaluation function

Both the `ProbabilisticExploration()` and `Optimisation()` objects expect the simulation model to have the `.run()` method for running the evaluation function. The framework provides two approaches to connect the simulation model to the API: the class `data.Model()` and the `data.ExtendableModel()` abstract class. The first is a wrapper for the actual evaluation function. At the same time, the second is a template for defining a custom model object, useful if the simulation model requires a setup or parameters to be changed at the object creation (see the example in Listing 2.3).

```
class My_Model(ExtendableModel):

def __init__(self, model_parameters):
self.model_parameters = model_parameters

### Do something with the parameters to setup the simulation

def run(self, *args):

### Evaluation function, with variable inputs

```

Listing 2.3: Example of use of `ExtendableModel()`

In both cases, the evaluation function must be crafted so that the input and output match the format PDOPT expects. Input arguments are specified in the `input.csv` file in the order listed. Failure to do this will cause a mismatch of the input parameters. For instance, if the input file contains the parameters A, B, and C in this order, the evaluation function should be:

```
def my_eval_fun(A, B, C):

## Process the input and return the QoIs.
```

Listing 2.4: Evaluation function with hardcoded inputs.

It is possible to make the evaluation function more flexible by using `*args`. This creates a tuple object called `args` that contains the parameters passed, enabling handling a variable number of inputs. The user must hardcode the internal logic to handle the variable inputs. In the case of the previous example (Listing 2.4), the function can be rewritten using the `*args` argument:

```
def my_eval_fun(*args):

## Internal logic
if len(args) == 3:
A, B, C = args
else:
A, B, C, D = args

## Process the input and return the QoIs.
```

Listing 2.5: Evaluation function with variable inputs.

The example unpacks the `args` object depending if it has a length of 3 or 4. The output of the evaluation function expected by the framework is a Python dictionary containing the values of the QoIs indexed by the name used in the `response.csv` file. An example is shown in Listing 2.6 assuming the QoIs are X, Y, and Z.

```
def my_eval_fun(*args):
## Variable input example
```

```
3
4  ## Internal logic
5  if len(args) == 3:
6  A, B, C = args
7  else:
8  A, B, C, D = args
9
10 ## Process the input and gejnerate the QoIs.
11 out_dict = {'X': X,
12  'Y': Y,
13  'Z': Z}
14
15 return out_dict
```

Listing 2.6: Example of return dictionary in the evaluation function.

2.1.4 Structure of the Output

Two Pandas `DataFrame` are generated as output. The first covers the results from the exploration phase, while the second covers the results of the search phase.

The results of the exploration phase cover the probabilities of satisfaction of the requirements and the levels of the input parameters that make up each set. Each entry is a set with the following columns:

- `set_id` : The number identifying the set.
- `is_discarded` : A Boolean value of 0/1 indicating whether 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 specified in the `response.csv` file such that it plays a role in the exploration phase.

For the search phase, the results present the optimal points and their QoI values. Each entry in the table is a design point with these columns:

- `set_id`: The number identifying the set to which this data point belongs.
- Columns of the input parameters as defined in `input.csv`. These values are the actual number and not a level; it is the found optimum value.
- Columns of response quantities, objective and constraints, as defined in `response.csv`.

3 Example Tutorial

This section presents the step by step process to perform a PDOPT 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 [13]: 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.

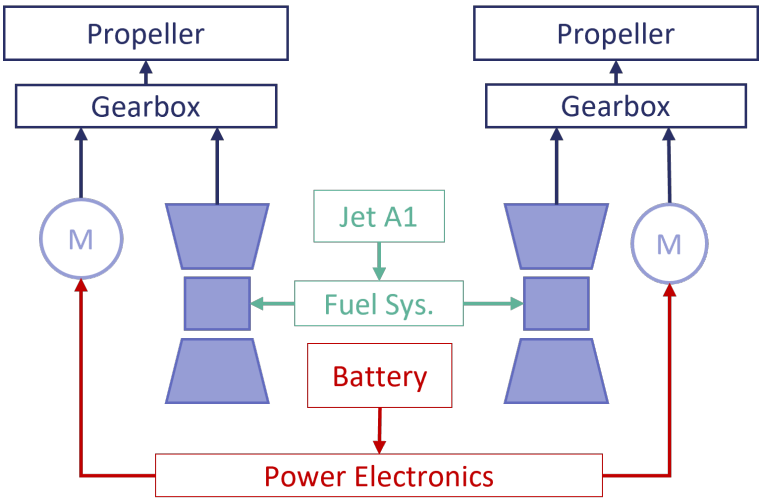

Figure 3.1: Propulsion Architecture

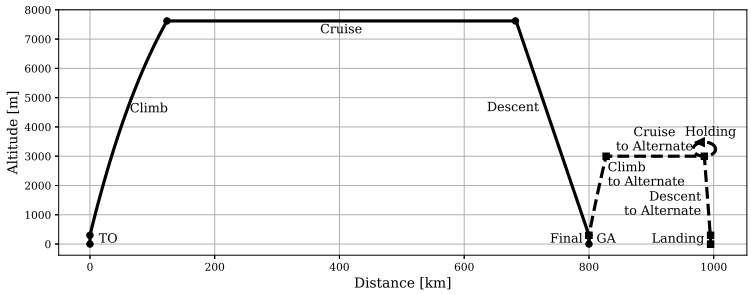

Figure 3.2: Mission profile

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).

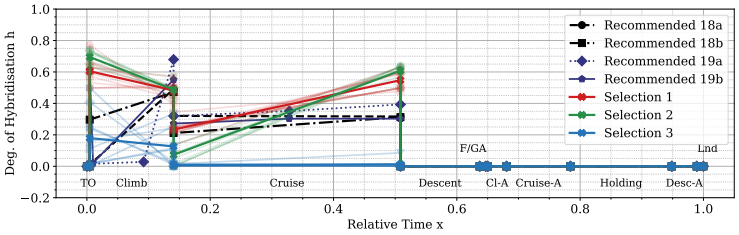

Figure 3.3: Different linear EMS over Climb and Cruise.

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 user 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 2.1.1. 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 2.1.2. 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 2.1.3) 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:

```
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):
```

```
25
26          ## Omitted post-processing code, runs the same analysis
27          ## but returns more information
28
29          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.

```
1  folder = 'run'
2  architecture = json.load(open('data/architecture.json', 'r'))
3  mission = 'data/mission_original.csv'
4  experiment = Experiment(folder + '/input.csv',
5                          architecture, mission)
6
7  # Check if a design space is already present otherwise create it
8  if exists(folder + '/design_space.pk') and restart:
9      design_space = pk.load(open(folder + '/design_space.pk','rb'))
10 else:
11     design_space = DesignSpace(folder + '/input.csv',
12                                folder + '/response.csv')
13     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.

```
1  n_exp_train = 100
2
3  # Check if there is already a trained exploration object
4  if exists(folder + '/exploration.pk') and restart:
5      exploration = pk.load(open(folder + '/exploration.pk','rb'))
6  else:
7      exploration = ProbabilisticExploration(design_space,
8                      experiment,
9                      surrogate_training_data_file=folder + '/samples.csv',
10                     n_train_points=n_exp_train)
11
12     for k in exploration.surrogates:
13         s = exploration.surrogates[k]
14         #Print the R score of each surrogate, to check
15         #their training. For smooth problems it should be
16         #above 0.8.
17         print(f'Surrogate {s.name} with r = {s.score:.4f}')
18
19     pk.dump(exploration, open(folder + '/exploration.pk','wb'))
20
21 n_exp_samples = 100
22 P_exploration = 0.5
23
24 # 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 dataframeme, 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 [8].

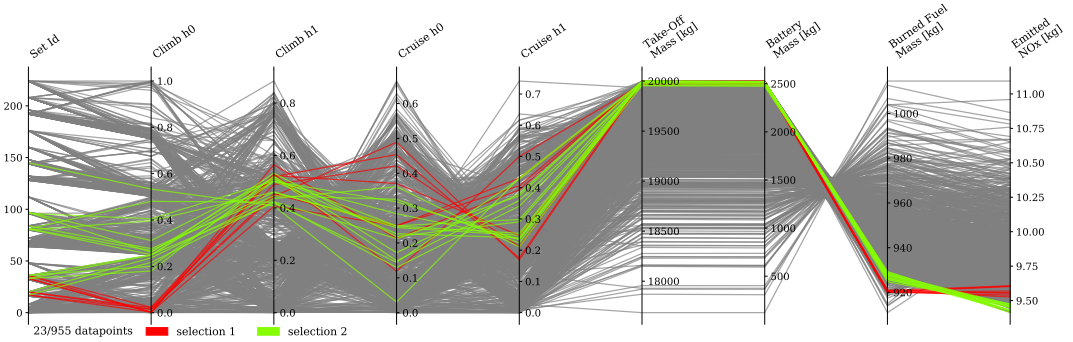

Figure 3.4: Parallel Coordinates with selection.

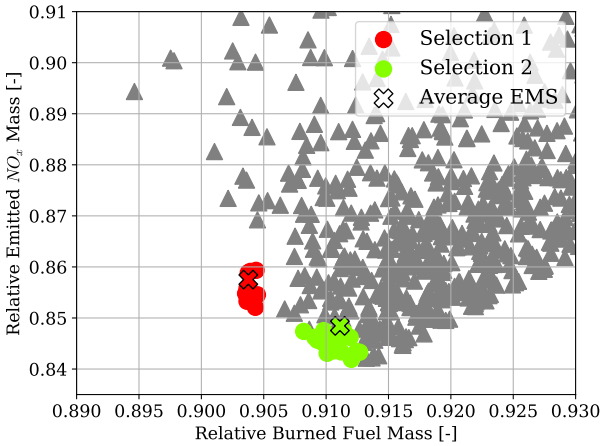

Figure 3.5: Scatter plot showing the selection of Fig. 3.4

3.7 The `visualisation.py` interactive visualisation environment

Included with PDOPT is a simple interactive visualisation tool built on the `plotly-dash` library. Once executed with the command `python visualisation.py`, an HTML link will be available to use the tool in a web browser. It provides a

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 2.1.3.

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.**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.optimisation.**NNSurrogate**(*model, design_space, set_id*)

Class that represents a deterministic optimisation problem, using a neural-network surrogate model for function evaluation. It is locally trained before use on the DesignSet it is part of. This class wraps the `pymoo.core.problem.Problem` class.

Parameters:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	The design space.
set_id [int]	Unique id of the set the optimisation problem is part of.

Returns:

None

Attributes:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	Reference to DesignSpace object.
var [list]	List of parameter objects.
obj [list]	List of objectives objects.
cst [list]	List of constraint objects.
set_id [int]	Unique id of the set the optimisation problem is part of.
l [list]	List of lower bound values of each parameter.
u [list]	List of upper bound values of each parameter.

`class` pdopt.optimisation.**KrigingSurrogate**(*model, design_space, set_id, kernel*)

Class that represents a deterministic optimisation problem, using a Kriging surrogate model for function evaluation. It is locally trained before use on the DesignSet it is part of. This class wraps the `pymoo.core.problem.Problem` class.

Parameters:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	The design space.
set_id [int]	Unique id of the set the optimisation problem is part of.
kernel [str]	Gaussian Process kernel. Default is Matern with `'matern'`. Alternative is Radial Basis Function with `'rbf'`.

Returns:

None

Attributes:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	Reference to DesignSpace object.
var [list]	List of parameter objects.
obj [list]	List of objectives objects.
cst [list]	List of constraint objects.
set_id [int]	Unique id of the set the optimisation problem is part of.
l [list]	List of lower bound values of each parameter.
u [list]	List of upper bound values of each parameter.

class pdopt.optimisation.**KrigingSurrogate**(*model, design_space, set_id*)

Class that represents a deterministic optimisation problem, using the direct model for function evaluation. This class wraps the `pymoo.core.problem.Problem` class.

Parameters:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	The design space.
set_id [int]	Unique id of the set the optimisation problem is part of.

Returns:

None

Attributes:

model [pdopt.data.Model]	Evaluation function model.
design_space [pdopt.data.DesignSpace]	Reference to DesignSpace object.
var [list]	List of parameter objects.
obj [list]	List of objectives objects.
cst [list]	List of constraint objects.
set_id [int]	Unique id of the set the optimisation problem is part of.
l [list]	List of lower bound values of each parameter.
u [list]	List of upper bound values of each parameter.

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.
use_surrogate [bool, default=True]	Flag to use the surrogate optimisation model. Set to false it will use the full model.
use_nn [bool, default=False]	If **use_surrogate** set to true, it will use the NN model. Otherwise it will be using the KrigingModel.
***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.
algorithm [unsga3]	The `pymoo` UNSGA3 algorithm implementation.
termination [DefaultMultiObjectiveTermination]	The `pymoo` MOO termination criteria implementation.

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

Run the search in all the surviving design sets.

Parameters:

folder	The folder where the test case is stored.

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)

```

```
@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()
```

```
248
249
250         analysis = model(en_management, architecture_data=self.arch,
    mission_file=self.mission)
251
252         results = postpro_run(self.arch, analysis)
253
254         return results
255
256
257 def run_experiment(folder, n_exp_samples, P_exploration, restart, n_exp_train):
258     print('Input Args: ', folder, n_exp_samples, P_exploration, restart,
    n_exp_train)
259
260     architecture = json.load(open('data/architecture.json', 'r'))
261     mission = 'data/mission_original.csv'
262     experiment = Experiment(folder + '/input.csv', architecture, mission)
263
264
265     # Check if a design space is already present otherwise create it
266     if exists(folder + '/design_space.pk') and restart:
267         design_space = pk.load(open(folder + '/design_space.pk','rb'))
268     else:
269         design_space = DesignSpace(folder + '/input.csv', folder + '/response.
    csv')
270         pk.dump(design_space, open(folder + '/design_space.pk','wb'))
271
272     # Check if there is already a trained exploration object
273     if exists(folder + '/exploration.pk') and restart:
274         exploration = pk.load(open(folder + '/exploration.pk','rb'))
275     else:
276         exploration = ProbabilisticExploration(design_space, experiment,
277                                                surrogate_training_data_file=folder
    + '/samples.csv',
278                                                n_train_points=n_exp_train)
279
280         for k in exploration.surrogates:
281             s = exploration.surrogates[k]
282             print(f'Surrogate {s.name} with r = {s.score:.4f}')
283
284         pk.dump(exploration, open(folder + '/exploration.pk','wb'))
285
286
287     # Check if exploration has been done already
288     if exists(folder + '/exp_results.csv') and restart:
289         pass
290     else:
291         exploration.run(n_exp_samples, P_exploration)
292         design_space.get_exploration_results().to_csv(folder + '/exp_results.
    csv', index=False)
293
294         #Update the saved design object
295         pk.dump(design_space, open(folder + '/design_space.pk','wb'))
296
297
298     optimisation = Optimisation(design_space, experiment, n_max_evals=2000)
299
300
301     # Check if optimisation has been done already
302     if exists(folder + '/opt_results_raw.csv') and restart:
303         pass
304     else:
305         optimisation.run()
306         df_opt = design_space.get_optimum_results()
307         df_opt.to_csv(folder + '/opt_results_raw.csv', index=False)
308
309         #Update the saved design object
310         pk.dump(design_space, open(folder + '/design_space.pk','wb'))
311
312     if exists(folder + '/opt_results.csv') and restart:
```

```
            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 multi-dimensional geometry". In: *Proceedings of the First IEEE Conference on Visualization: Visualization '90*. 1990, pp. 361–378. DOI: 10.1109/VISUAL.1990.146402.

[6] *Joblib: running Python functions as pipeline jobs*. 2008. URL: https://joblib.readthedocs.io/ (visited on 06/13/2023).

[7] Stephen Joe and Frances Y Kuo. "Constructing Sobol sequences with better two-dimensional projections". In: *SIAM Journal on Scientific Computing* 30.5 (2008), pp. 2635–2654. DOI: 10.1137/070709359.

[8] 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.

[9] MD McKay, RJ Beckman, and WJ Conover. "Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. DOI: 10.5555/1953048.2078195.

[11] Andrea Spinelli. "A Set-Based Design Space Exploration Framework for Hybrid-Electric Aircraft Design". PhD Thesis. Cranfield, UK: Cranfield University, Aug. 2023.

[12] 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.

[13] 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