POLITECNICO
MILANO 1863

Geoinformatics Engineering
Geoinformatics Project

A Multi-Fidelity Bayesian Method for Complex Model Optimization: An Industrial Case Study

Student: **Lorenzo Carlassara**

Student ID: 101724
Supervisor: Prof. Ludovico G. A. Biagi
Co-supervisors: Davide Baroli
Academic Year: 2022-23

Abstract

Computational models are often evaluated via stochastic simulation or numerical approximation. Fitting these models implies a difficult optimization problem over complex, possibly noisy parameter landscapes. The need for multiple realizations, as in uncertainty quantification or optimization, makes surrogate models an attractive option. For expensive high-fidelity models, however, even performing the number of simulations needed for fitting a surrogate may be too expensive. Inexpensive but less accurate low-fidelity data or models are often also available. Multi-fidelity combine high-fidelity and low-fidelity in order to achieve accuracy at a reasonable cost. Here we consider an hybrid method based on Multi-fidelity for implementing a Bayesian optimization algorithm. At the heart of this algorithm is maximizing the information criterion called acquisition function, and a list of the possible available choices is presented. Multi-fidelity Bayesian optimization achieves competitive performance with an affordable computational overhead for the running time of non-optimized models.

Keywords: gaussian process regression, neural net, optimization, uncertainty quantification

The project is available at the following repository: `https://github.com/carls31/MFBO-Method`

Contents

1 | Introduction

The aim of this project is to develop additional features in the design-exploration space, including Multi-fidelity and adaptive techniques, as well as ANOVA-analysis for uncertainty reduction. The project will focus on testing the results of these features on industrial case studies, such as the WindAssistant Transoceanic Cargo and Energy System and offers a remark on the use in Geoinformatics. The project plan involves reading and understanding a reference paper on kernel-based methods, and Multi-fidelity, becoming familiar with Bayesian optimization, surrogate models, and Multi-fidelity algorithms, designing, training, and implementing the studied algorithms, and performing a performance assessment on the datasets.
To this end, we propose an efficient method for reducing uncertainty using multi fidelity and Bayesian optimization with a general overview of possible acquisition functions.

Bayesian optimization (BO) is an approach to solving challenging optimization problems. It uses a machine learning technique (Gaussian process regression [6]) to estimate the objective function based on past evaluations, and then uses an acquisition function [2] to decide where to sample next. It typically takes longer than other optimization methods to decide where to sample, but uses fewer evaluations to find good solutions.
Indeed, BO is a good choice when each evaluation of the objective function takes a long time to evaluate or is expensive. This prevents evaluating the objective too many times.

Multi-fidelity (MF) refers to the use of multiple models of varying levels of fidelity to approximate a complex system or process. In the context of optimization, MF methods can be used to reduce the computational cost of evaluating the objective function, by using low-fidelity models to guide the search and high-fidelity models to refine the results.

Neural networks are a class of machine learning models that can learn to predict a continuous target variable based on a set of input features. These models are particularly powerful because they can learn complex, nonlinear relationships between the inputs and outputs, and can automatically extract relevant features from the data.
At a high level, a neural network for regression consists of an input layer, one or more hidden layers, and an output layer. Each layer contains a set of neurons that perform a

weighted sum of the inputs and apply an activation function to produce the output of the layer.
The goal of training the network is to learn the optimal values for the weights and biases such that the output of the network closely matches the target outputs in the training dataset.

Chapter 2 outlines the software requirements for a project, including the necessary operating system, programming language, and required libraries such as Botorch and PyTorch.

Chapter 3 provides a theoretical review to support the implementation, as well as a notebook tutorial in python to facilitate easy comprehension. We discuss Gaussian Process, Bayesian Optimization, Multi-fidelity and Linear Neural Network.

Chapter 4 includes the following tasks:

- Set up the PyTorch environment
- Data loading and preprocessing from a CSV file
- Define a PyTorch Dataset for the test case
- Initialize the data for a Bayesian optimization problem
- Define the neural network model
- Initialize the surrogate model
- Perform multi-fidelity Bayesian optimization (BO) using different acquisition functions
- Visualize the optimized GP model uncertainty level

Chapter 5 presents the results of simulations conducted using the MFBO method with different acquisition functions.

Chapter 6 concludes that MFBO is a powerful approach for global optimization, but it requires careful attention to the selection and calibration of the surrogate model, acquisition function, and high fidelity model parameters. Additionally, the chapter discusses the potential application of MFBO in managing, processing, and visualizing geo data, and presents an adaptation of the method to the ERA5 sample dataset.

2 | Software requirements

1. Operating System: the software is compatible with the operating system being used by the users. In this case, Windows.
2. Programming Language: the software is developed using Python programming language version `3.9`.
3. Libraries: the core of the tasks are carried out through the following Python packages installed in a proper *mamba* virtual environment:

torch	version `1.12.1`	botorch	version `0.8.3`
gpytorch	version `1.9.1`	numpy	version `1.23.5`
scikit-learn	version `1.2.1`	scipy	version `1.10.0`
matplotlib	version `3.4.2`	pandas	version `1.5.3`
pyDoe	version `0.3.8`	xarray	version `2022.11.0`

4. Jupyter Notebook: The software is developed and executed in Jupyter Notebook for better visualization and presentation of results.
5. Hardware Requirements: The software is able to run on a standard computer system with a minimum of 4 GB of RAM and a 64-bit processor.
6. Development Environment: The software is developed using an appropriate Integrated Development Environment (IDE) such as VS Code or Anaconda.

Brief description of the two main packages for scientific computing:

2.1. Botorch

Botorch is a Bayesian optimization library for PyTorch, a popular deep learning framework. It provides a high-level API for specifying and fitting Bayesian optimization models and running Bayesian optimization loops in a modular, scalable, and efficient way. Botorch leverages the tensor computation capabilities of PyTorch to perform gradient-based optimization and handle complex constraints, making it well-suited for optimizing large, complex, and high-dimensional black-box functions.

2.2. PyTorch

PyTorch is an open-source machine learning library for Python. It provides a tensor computation library for building and training deep learning models, with a focus on flexibility and efficiency. PyTorch is widely used in the machine learning community for tasks such as image classification, natural language processing, and generative models. It also provides a high-level API for defining, training, and evaluating deep learning models, as well as tools for debugging and visualization.

3 | Background

In this section we introduce a theoretical review of the techniques implemented in the method.

3.1. Gaussian Process

A Gaussian process (GP) is a flexible probabilistic model used for supervised learning tasks such as regression, classification, and time series analysis. Unlike other models, a GP defines a probability distribution over functions, rather than a single function. A GP can be thought of as a distribution over the space of functions that we might believe could be generating our data.
In a GP, any finite set of function values have a joint Gaussian distribution. The mean function, $m(\mathbf{x})$, and the covariance function (also known as kernel function), $k(\mathbf{x}, \mathbf{x}')$, of the GP define the properties of the distribution over functions. The mean function specifies the expected value of the function being modeled at any given input, and the covariance function specifies how correlated the function values are at different points in the input space.
To use a GP for regression, we start with a training set of input-output pairs, and compute the posterior distribution over functions that best explain the data. This involves updating our prior beliefs about the distribution of functions using Bayes' formula

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3.1}$$

with the data providing new evidence. Its snippet implementation is shown in algorithm 1.

3.1.1. Mean function and Kernel function

In GP regression, the **mean** function is an important component of the GP regression model as it determines the overall trend of the function being modeled, as well as a prior estimate of the expected value of the target variable at each input point. For these reasons

the mean function can be useful for incorporating prior knowledge or incorporating biases in the model.

The **kernel** function takes two input points, $\mathbf{x}$ and $\mathbf{x}'$, and computes the covariance between the function values at these two points. The covariance determines how similar the function values are at these two points, with higher covariance indicating that the function values are more strongly correlated.

There are many different types of kernel functions that can be used in GP regression, each with their own properties and assumptions about the function being modeled. The kernel function implemented in the code is the **squared exponential kernel**, also known as the **radial basis function** (RBF), this kernel assumes that the function is smooth and infinitely differentiable. It is defined as:

$$k(\mathbf{x}, \mathbf{x}') = exp(\frac{-||\mathbf{x} - \mathbf{x}'||^2}{2l^2}) \tag{3.2}$$

where l is a length scale parameter that determines the scale of the smoothness.

In practice, the kernel function is typically learned by optimizing a measure of model fit, such as the log marginal likelihood, using an optimization algorithm.

Together, the mean and covariance functions define the properties of the distribution over functions that we believe could be generating our data.

One advantage of GP is that they allow for uncertainty quantification in predictions. In addition to a point estimate of the function value, a GP also provides a measure of the uncertainty or confidence in the prediction at each point in the input space. This can be useful in many applications, such as when making decisions based on the predicted values, or, like in this case, when the model is used to optimize a function.

Algorithm 1 Gaussian Process

Input: Training data $\mathcal{D} = (\mathbf{x}_n, y_n) : n = 1, \ldots, N$, test input $\mathbf{x}^*$
Output: Predictive distribution $p(y|\mathbf{x}, \mathcal{D})$
while $i < iteration_{max}$ **do**
    Form a covariance matrix $K = k(X, X) + \sigma^2 I_N$;
    Compute a vector a such that $Ka = y$;
    Compute the log-likelihood $\log p(y|X) = -\frac{1}{2}y^T a - \frac{1}{2}log|K - \frac{N}{2}log(2\pi)$;
    Compute the gradient of the likelihood w.r.t. to the hyperparameters;
    Estimate the optimal hyperparameters θ_{opt} by maximizing the likelihood
    Increment i
end while
Set $K = K(\theta_{opt})$ and $a = a(\theta_{opt})$ for the optimal hyperparameters;
Form correlation matrices $K^{**} = k(X^*, X^*)$ and $K^* = k(X^*, X)$ for the optimal hyperparameters
Compute a matrix A^* such that $KA^* = K^*$
Form the conditioning mean value vector $m^* = K^{*T}a$ and the corresponding covariance matrix $C^* = K^{**} - K^{*T}A^*$
Define f^*

3.2. Bayesian optimization

BO is a class of machine-learning-based optimization methods focused on finding the maximum of an unknown, potentially noisy, objective function $f(x)$ [2]:

$$\max_{x \in A} f(x), \tag{3.3}$$

It uses a Bayesian framework to model the underlying function and update the model based on the observed data to guide the search towards the optimum. This approach is particularly useful for expensive to evaluate black-box functions, such as simulations or experiments, as it tries to maximize the information gained at each iteration. The method uses a combination of prior knowledge and observed data to balance exploration and exploitation in the search process, making it a popular method for hyperparameter tuning and other optimization problems.

Algorithm 2 Bayesian optimization

1: Define the search space $\mathcal{S}$ and the objective function f
2: Initialize the dataset $\mathcal{D}$ with a small number of points
3: Define a prior distribution over the objective function f using a GP model
4: **while** $n \leq N$ **do**
5: Update the posterior probability distribution on f using all available data.
6: Let x_n be a maximizer of the acquisition function over x, where the acquisition function is computed using the current posterior distribution.
7: Observe $y_n = f(x_n)$
8: Increment n
9: **end while**
10: Return the best observed point as the solution

3.2.1. Gaussian Process Regression in BO

BO involves the iterative construction of a probabilistic surrogate model of the expensive function being optimized, which is used to predict the function values at unexplored points in the search space. A surrogate is usually viewed as a function informative of, but easier to optimise than, objective itself [3]. The surrogate model is typically constructed using a GP [6], which allows us to model the distribution over functions that could potentially be the unknown expensive function.

At each iteration of BO, the surrogate model is used to guide the selection of the next point to evaluate in the search space. This is done by defining an acquisition function that balances exploration (sampling points where the model is uncertain) and exploitation (sampling points where the model predicts high values). The acquisition function is optimized using the GP posterior distribution to find the point in the search space with the highest expected improvement in the objective function.

Once a new point is selected, it is evaluated using the expensive objective function, and the GP surrogate model is updated with the new data. The updated model is then used to select the next point to evaluate, and the process continues until a stopping criterion is met.

By iteratively improving the GP surrogate model and selecting the next point to evaluate based on the acquisition function, BO is able to efficiently search for the optimal point in the search space. The GP allows for the uncertainty in the predictions to be quantified and updated with each new evaluation, and the acquisition function balances exploration and exploitation to ensure efficient search. The overall procedure is shown in algorithm 2.

In summary, GP regression is used in BO to construct a probabilistic surrogate model of the objective function being optimized, which is used to guide the search for the optimal point in the search space.

3.2.2. Objective Function

The objective function (3.3) in BO is the function that we want to optimize. The objective function maps the input domain (search space) to a scalar output, which represents the value of the function at that point in the search space.

The choice of the objective function depends on the specific optimization problem and the goal of the optimization. The objective function can be any function that can be evaluated at any point in the search space. The function can be deterministic or stochastic, noisy or noise-free, and may have multiple optima or a single global optima.

In BO, the objective function is typically expensive to evaluate, meaning that each function evaluation can be time-consuming, computationally intensive, or require a physical experiment. Therefore, the number of function evaluations should be minimized while still finding the optimal solution.

BO tries to minimize the number of function evaluations required to find the optimal solution by iteratively fitting a probabilistic model to the observed data and selecting the next point to evaluate based on the acquisition function, which balances exploration and exploitation.

Overall, the choice of the objective function is a crucial aspect of BO, as it defines the optimization problem and the search space, and impacts the optimization algorithm's performance and convergence.

3.2.3. Acquisition Functions

Acquisition functions in BO are used to decide the next point to sample the objective function. The acquisition function measures the utility of each point in the search space and guides the optimization algorithm to sample the most promising points, i.e., those that are most likely to improve the objective function's value.

The acquisition function is a trade-off between exploration and exploitation. The exploration term encourages the optimization algorithm to sample points that are uncertain, i.e., with high variance in the probabilistic model. The exploitation term, on the other hand, directs the algorithm to sample points that are expected to have a high function

value based on the probabilistic model.

The choice of the acquisition function depends on the specific optimization problem and the characteristics of the objective function. Some popular acquisition functions include:

- Upper Confidence Bound (UCB):

$$UCB(x, \lambda) = \mu(x) + \lambda\sigma^2(x), \tag{3.4}$$

  UCB (3.4) balances the exploration and exploitation of the search space by selecting points with high expected values and high uncertainty. It is useful for problems with a high-dimensional search space and when the objective function has a complex structure.

- Probability of Improvement (PI):

$$PI(x) = \max(f(x) - f(x^*), 0), \tag{3.5}$$

  PI (3.5) selects points based on the probability that the function will improve over the current best solution. It is useful for problems where the objective function has a high signal-to-noise ratio.

- Expected Improvement (EI):

$$EI(x) = \int_{-\infty}^{+\infty} \max(f(x) - f(x^*), 0)\varphi(z)dz, \tag{3.6}$$

  EI (3.6) encourages the exploration of the search space by selecting points that have a high probability of improving over the current best solution. It is well-suited for problems where the objective function is expensive to evaluate and has a low-dimensional search space. If the measurement error is small relative to the effect size, Bayesian optimization using a heuristic EI can be successful. However, when the measurement noise is high we can substantially improve performance by properly integrating out the uncertainty [4].

Other acquisition functions suitable for problems where multiple points can be sampled simultaneously and can be evaluated in parallel are for example:

- q-Expected Improvement (qEI): qEI selects a set of points with high expected improvement over the current best solution. It is useful for problems where multiple points can be sampled simultaneously and evaluated in parallel.

- q-Noisy Expected Improvement (qNEI): qNEI is an extension of EI that accounts for the noise in the objective function. It selects a set of points that maximizes the expected improvement while taking into account the noise in the function evaluations.
- q-Knowledge Gradient (qKG): qKG selects the next sampling decision by maximizing the expected incremental value of a measurement, without assuming (as expected improvement does) that the point returned as the optimum must be a previously sampled point [7]. This acquisition function, regretfully, leads to computationally too expensive calculations with the actual machine and will be only partially implemented in the code.

3.3. Multi-fidelity

Multi-fidelity (MF) refers to a concept in simulation and optimization where multiple models or simulations of varying accuracy and computational cost are used to represent a system. In a MF framework, a hierarchy of models is constructed, with the most accurate and computationally expensive model at the top, and the lower-fidelity and less expensive models at the bottom. The idea is to use the lower-fidelity models to quickly identify the best regions of the solution space, and then use the higher-fidelity models to more accurately evaluate these candidate solutions. This approach can significantly reduce the uncertainty area compared to using only the highest-fidelity model for the entire optimization process as shown in the example 3.1. The goal of MF optimization is to find the optimal solution by balancing the trade-off between accuracy and computational cost. BO, on the other hand, is a probabilistic approach to global optimization that involves iteratively constructing a probabilistic model of the objective function and using it to make informed decisions about where to evaluate the function next. The main advantage of BO is its ability to handle high-dimensional and noisy objectives and to effectively balance exploration and exploitation.

3.3.1. Multi-fidelity Bayesian optimization

Multi-fidelity Bayesian optimization (MFBO) combines the advantages of both MF and BO techniques by incorporating multiple fidelities of the objective function into the probabilistic model used in BO. This can lead to more efficient optimization by allowing for faster convergence to the optimal solution, as well as improved robustness to noise and uncertainty in the objective function.
Many applications allow multi-fidelity queries of the objective function, $f(x)$, where the

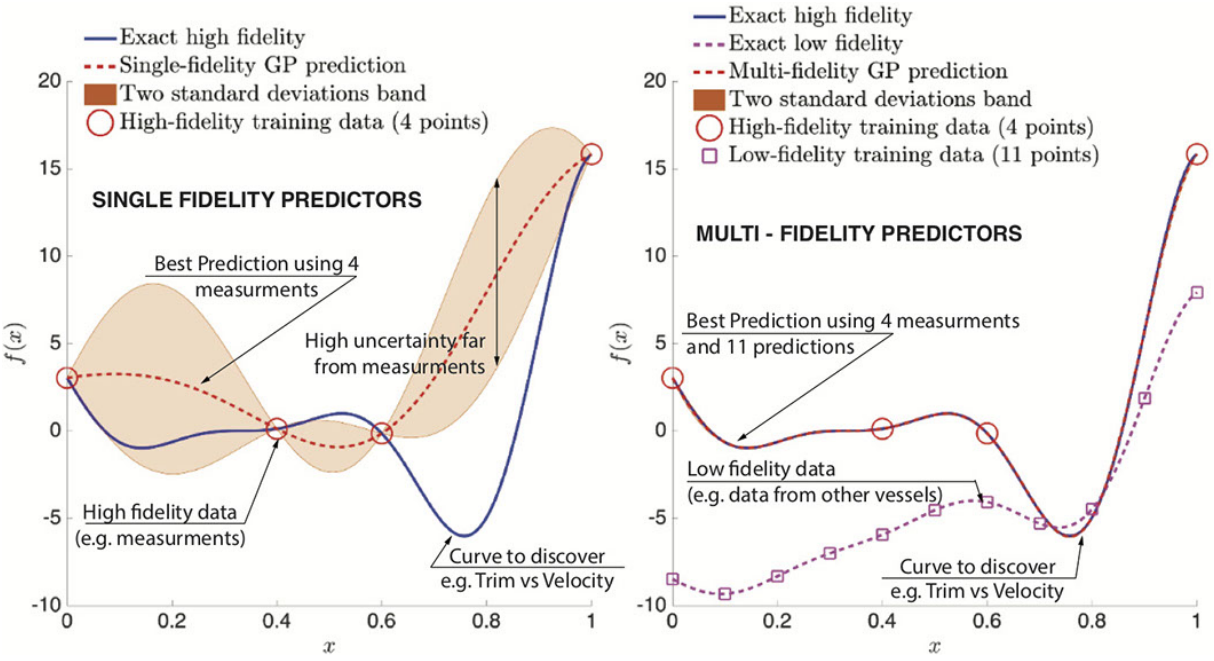

Figure 3.1: GP regression on Single-fidelity data (left) vs GP regression on Multi-fidelity data (right) as shown in [1]

higher (larger) the fidelity, the more accurate yet costly the query of $f_m(x)$. Many studies have extended BO for multi-fidelity settings [5].

MFBO is a powerful extension of BO that incorporates information from multiple models of different fidelity levels to improve the efficiency and accuracy of the optimization process. The key idea behind MFBO is to use a hierarchy of surrogate models that vary in fidelity and computational cost, to approximate the objective function at different levels of detail.

At the lowest fidelity level, a cheap and coarse model is used to explore the search space and identify promising regions. At higher fidelity levels, more expensive and accurate models are used to refine the results and converge to the global optimum. The models at each level are typically linked through a set of calibration parameters that allow them to be combined and integrated into a coherent probabilistic model.

MFBO has been shown to be particularly effective for optimizing complex systems with expensive-to-evaluate objective functions, such as aerospace design, chemical engineering, and climate modeling. By using a hierarchy of surrogate models, MFBO can significantly reduce the computational cost and accelerate the optimization process, while maintaining high accuracy and robustness.

3.4. Linear Neural Network for Regression

A neural network for regression works by learning a mapping function from input features to continuous output values. This mapping function is represented as a series of interconnected processing nodes, or artificial neurons, arranged in multiple layers.
The **first layer** of the network, called the input layer, takes the input features and passes them through to the next layer.
Subsequent layers, called **hidden layers**, contain artificial neurons that apply a series of transformations to the inputs and pass the results to the next layer.
The final layer of the network, called the **output layer**, produces the final prediction for the target output.
Each artificial neuron in a hidden layer applies a linear transformation to its inputs using a weight matrix and adds a bias term. The result is then passed through an activation function, which non-linearly transforms the inputs and introduces non-linearity into the model.
The **activation function** is applied to the output of each neuron and can have a significant impact on the performance of the model. Common activation functions include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function, which is implemented in our method.
Another important consideration is the choice of **loss function**, which measures the difference between the predicted output of the network and the true target value. Common loss functions for regression are supervised learning algorithm such as the mean squared error (MSE) optimization, where the difference between the network's predictions and the target outputs is minimized. The weights and biases of the neurons are learned during training using an optimization algorithm such as stochastic gradient descent or Adam.
The training process typically involves iteratively adjusting the weights and biases to minimize the loss function on the training data, while also monitoring the performance on a validation set to avoid overfitting.
Overall, neural networks for regression are a powerful and flexible class of machine learning models that can be used to learn complex, nonlinear relationships between inputs and outputs. With careful design and training, these models can achieve high accuracy and generalization performance on a wide range of regression tasks.
While linear models are not sufficiently rich to express the many complicated neural networks, neural networks are rich enough to subsume linear models as neural networks in which every feature is represented by an input neuron, all of which are connected directly to the output [8].

4 | Model setup

In this section we introduce the procedure followed to implement the script.

Set up the PyTorch environment for use with the Botorch library for Bayesian optimization research. The code imports several libraries including `os`, `torch`, `botorch`, and `gpytorch`. We set the default data type to `double` for the `torch` library. Then, we check if an environment variable called `SMOKE_TEST` is set and determines whether the deployed build is stable or not. After that, we check if a GPU device is available, and if it is, set the `device` variable to `"cuda"`. If not, we set device to `"cpu"`. Finally, we display the versions of PyTorch and Botorch libraries being used, as well as the device being used.

Data loading and preprocessing from a CSV file for use in a machine learning model. The code first imports `numpy`, `pandas`, `StandardScaler` and `MinMaxScaler`. Then, we define a variable `data` as the path to a CSV file called `Case3_finer.csv`. Next, we defines a function called `load_data` that loads the data from the CSV file into a pandas dataframe using the `pd.read_csv()` function. Then, we extract the "alpha" values and the "SumCx" and "SumCy" values from the dataframe, and standardize them using the `StandardScaler()` function. Finally, the function returns the standardized "alpha" values and the standardized "SumCx" and "SumCy" values.

Performing a PCA transformation on a given dataset. The code imports the `PCA` function from the `sklearn.decomposition` module. We define `PCA_transformation`, a function which takes in input a variable `pts_original`. Inside the function, we create a PCA object with `n_components=4` and `svd_solver='full'` using the `PCA` function, and fit it to the `pts_original` data using the `fit()` method. Then, we transform the `pts_original` data into a new dataset called `pts_transformed` using the `transform()` method of the PCA object. Finally, the transformed dataset `pts_transformed` is returned.

Data resampling using a Latin hypercube sample and a nearest neighbor interpolation. The code imports the `lhs` function from `pyDOE` and the `NearestNDInterpolator` function from `scipy.interpolate`. It then defines a function called `data_resample` that

takes in `pts` and `obs` as inputs. Inside the function, it first determines the number of dimensions `dim` of the `pts` data and sets the number of sample points `N` to 10000. It also sets the lower and upper bounds of the `pts` data to `lb` and `ub`, respectively, and stores them in a dictionary called `bounds`. Next, it generates a Latin hypercube sample of size N within the bounds of `pts` using the `lhs()` function, and stores the result in a new array called `new_pts`. After that, it creates a `NearestNDInterpolator` object called `r` with `pts` and `obs` as inputs. Then, it uses the `r()` method to interpolate the `obs` data at the locations of the `new_pts` data, and stores the result in an array called `valuesTrasf`. Finally, the function returns both the `new_pts` and `valuesTrasf` arrays.

Define the PyTorch Dataset for the test case, which is used for training and testing the machine learning model. The `load_data()` function loads data from a CSV file and returns two arrays `pts` and `obs`. Then, `PCA_transformation()` is called to perform principal component analysis (PCA) on `pts`, which reduces the dimensionality of the input data. `data_resample()` is then called to sample the data for use in training and testing the model. The `TestcaseDataset` class defines a custom PyTorch Dataset that takes the sampled data and splits it into high-fidelity, low-fidelity, and test datasets using the `train_test_split()` function from scikit-learn. The constructor of the class creates three instances of `TensorDataset` representing the high-fidelity, low-fidelity, and test datasets. The `__call__()` method returns a tuple containing the three instances of `TensorDataset`, as well as the original high-fidelity, low-fidelity, and test data as PyTorch tensors. This tuple can be unpacked and used to train and test the machine learning model.

Initialize the data for the Bayesian optimization problem. The `TestcaseDataset` class is defined to create a dataset from the resampled data points and observations. This class splits the data into high fidelity (`X_hifi`, `Y_hifi`) and low fidelity (`X_lofi`, `Y_lofi`) sets, as well as a test set (`X_test`, `Y_test`). These datasets are then normalized using normalize and stored as tensors. The `hifi_dataset` and `test_dataset` are converted to PyTorch `DataLoader` objects, with batch size 32. Finally, noise is added to the low fidelity observations (`Y_lofi`) and a noise level is set using the `NOISE_LEVEL` variable.

Create 3D scatter plot using the matplotlib library. The plot shows the distribution of points in the high fidelity, test, and low fidelity datasets. The X, Y, and Z coordinates of each dataset are obtained from `X_hifi`, `X_test`, and `X_lofi`, respectively. The scatter plot is created using ax.scatter(), with different colors for each dataset and a legend to indicate which color corresponds to which dataset. Finally, the plot is displayed using `plt.show()`.

Define the neural network model using the PyTorch library. It specifies the number of input features, the size of the hidden layer, and the number of output features. The neural network consists of two linear layers with a LeakyReLU activation function applied between them. The `forward` method specifies the flow of information through the network. Specifically, it takes an input tensor `x`, applies the linear layers with the activation function, and returns the output tensor.

Train the neural network model using PyTorch. The neural network is defined using the nn.Module class and includes an input layer, a hidden layer, and an output layer. The model is trained using the Mean Square Error (MSE) loss and the Adam optimizer. The training process involves forward passes of the input data in batches, computation of prediction errors, back propagation of errors to update the model's parameters, and optimization of the weights. The trained model is then saved to disk. Finally, the model is evaluated on a test dataset, and the MSE loss is reported as the test error.

Load a saved PyTorch model from a file path. The saved model is of the `NeuralNet` class defined earlier in the code, and it is loaded using the `load_state_dict()` method of the `nn_model` object. The `eval()` method is then called on the `nn_model` object to set the model in evaluation mode.

Visualize the neural network's predictions vs the test data. It first uses the trained neural network model to make predictions on the test data (`X_test`) and stores the results in `Y_pred`. It then loops through each output variable (i) and creates a scatter plot for each input feature (`alpha`). For each input feature, it plots the actual test data (`X_test` and `Y_test`) as well as the neural network's predictions (`X_test` and `Y_pred`). Finally, it adds a legend to the plot to distinguish between the actual data and the predictions.

Initialize the surrogate model. Define a function called `initialize_model` that takes input data `X` and output data `Y` along with the noise level, and optionally a `state_dict` which is the dictionary of the model parameters. The function creates a list of `SingleTaskGP` models, where each model corresponds to a single output feature. Then, it combines all these models into a `ModelListGP` object. This allows to create a multi-task GP model, where each output is modeled independently. Next, it creates a likelihood object which assumes Gaussian noise, and uses it to compute the marginal log-likelihood of the models. If a `state_dict` is provided, the function loads the model state from it. Finally, the function returns the marginal log-likelihood and the GP model.

Define a helper function for performing the essential Bayesian optimization (BO) step. The function takes an acquisition function acqf as input, which is optimized

to obtain a new candidate point. The `optimize_acqf` function from the `botorch` library is used for this purpose. The function also returns the predicted optimal objective value at the new candidate point obtained using a pre-trained neural network model `nn_model`. The constants `NUM_POINTS, NUM_RESTARTS, RAW_SAMPLES`, and `NUM_INITIAL_POINTS` are defined for configuring the optimization process. The function uses a fixed random seed (`manual_seed(1234)`) for reproducibility.

Perform multi-fidelity Bayesian optimization (BO) using different acquisition functions: Upper Confidence Bound (UCB), Probability of Improvement (PI), Expected Improvement (EI), qExpected Improvement (qEI), and qNoisy Expected Improvement (qNEI). The goal is to iteratively optimize the unknown objective function by evaluating a low-fidelity and high-fidelity version of the function at each iteration. At each iteration, the code fits a Gaussian process model with the low-fidelity observations and uses it to optimize the acquisition function, which measures the potential utility of evaluating the high-fidelity function at different input points. The code defines different acquisition functions with their respective parameters and posterior transforms, and uses them to query the high-fidelity function at the most promising point. This process is repeated until convergence on the accuracy test with test data.

Define the Gaussian Process model for uncertainty quantification. The model is defined using the `ExactGP` class provided by the `GPyTorch` library. The class takes in the training data, `X_train` and `Y_train`, and a likelihood function (which defaults to `GaussianLikelihood`). Inside the class, the mean and covariance functions are defined. The mean function is set to a constant mean, while the covariance function is set to the `RBF` kernel scaled by a constant value. The forward method is used to calculate the mean and covariance of the Gaussian Process given an input x. The output is a `MultivariateNormal` distribution with mean `mean_x` and covariance `covar_x`.

Visualize the GP model uncertainty level. Through a function which takes as input the training data (`x_train, y_train`), along with other optional parameters such as `alpha`, `iter`, `acqf`. Inside the function, the Gaussian process model is defined using the `ExactGPModel` class, which inherits from `gpytorch.models.ExactGP`. The model uses a constant mean function and a scaled radial basis function (RBF) kernel as the covariance function. Then, the function finds the optimal model hyperparameters using the Adam optimizer and the exact marginal log-likelihood loss function. The optimizer iteratively updates the model parameters using the gradients computed from the loss function. The number of iterations is set by the `iter` parameter. Finally, the function returns the trained Gaussian process model and likelihood function in the evaluation mode. The returned model can be used to make predictions on new test data.

Train multiple Gaussian Process models with different acquisition functions: rand, UCB, PI, EI, qEI, qNEI on the given dataset (`X_lofi, Y_lofi`). The training is done for a fixed number of iterations (`TRAIN_ITER`). At each iteration, the model is trained using the `gp_visualization` function which trains the Gaussian Process and returns the trained model and the likelihood function. For each acquisition function, the trained models and likelihoods (`likelihood_list`, `gp_model_list`, etc.) are stored in separate lists for further use.

Set of loops that creates multiple plots for different acquisition functions used in Bayesian optimization. It uses the `matplotlib` library for plotting, and torch and `gpytorch` for the Gaussian process regression. Each loop initializes a subplot in the figure object using the `add_subplot()` method. Then, it creates a set of points to use as inputs to the Gaussian process regression using `torch.linspace()`. It feeds these inputs to the likelihood object associated with each acquisition function, which returns predicted mean and confidence intervals. The plots show the mean of the predictions as a solid line, and the confidence intervals as a shaded area. The observed data points are also plotted along with the predicted mean to show the accuracy of the model's predictions. The `figsize` parameter in the `plt.figure()` function specifies the size of the figure in inches. The loops use with `torch.no_grad()`: and `gpytorch.settings.fast_pred_var()`: to speed up the calculations by avoiding gradient computations during the forward pass.

Define helper functions that create different types of scatter plots using Plotly. The `import` statements at the top of the code import the necessary libraries for the functions, including NumPy for array manipulation, Plotly for data visualization, and ipywidgets for creating interactive widgets. The first function, `update_layout_of_graph`, updates the layout of a Plotly figure object. It takes in a Plotly `Figure` object, sets some attributes such as width, height, and plot background color, and returns the updated `Figure` object. The next four functions all create different types of scatter plots using the `go.Scatter` class from Plotly. `uncertainty_area_scatter` creates an area plot with shaded regions between two lines. It takes in boolean values for visibility and various NumPy arrays that define the x and y coordinates of the lines. `line_scatter` creates a line plot with specified x and y coordinates. It takes in boolean values for visibility, NumPy arrays for x and y coordinates of the line, and a string for the name of the line. `test_scatter` creates a scatter plot with green markers. It takes in boolean values for visibility and NumPy arrays for x and y coordinates of the points. `dot_scatter` creates a scatter plot with red markers. It takes in boolean values for visibility, NumPy arrays for x and y coordinates of the points, a string for the name of the points, and a boolean value for whether or not to show the plot's legend. All of these functions return objects

that can be added to a Plotly `Figure` object for display. All of these functions return objects that can be added to a Plotly `Figure` object for display.

5 | Results

In this section we report a list of the simulation results.

Industrial test case

Figure (5.1) shows the green uncertainty level described by the standard deviation before the MFBO method is applied.

Figure (5.2) shows the blue uncertainty level described by the standard deviation after applying MFBO with the EI acquisition function, but still the result wanted is not met yet due to not well tuned parameters inside the model.

Figure (5.3) shows the first fulfilled outcome applied to only one of the input (angle of the cargo wing), which will be extended on all the four wings.

Figure (5.8) shows the same result of the previous figure obtained by a different acquisition function, PI, which in the current settings appears to be, at least, equally promising.

Figure (5.5) shows the result provided by another acquisition function, qKG. This is the first test on all the four input wings. As mentioned before, even if the result of qKG were expected to be the most promising, we decided to discard this acquisition function because of the high computational time required.

Figure (5.6) shows the result provided by random choice for selecting the next points, i.e.

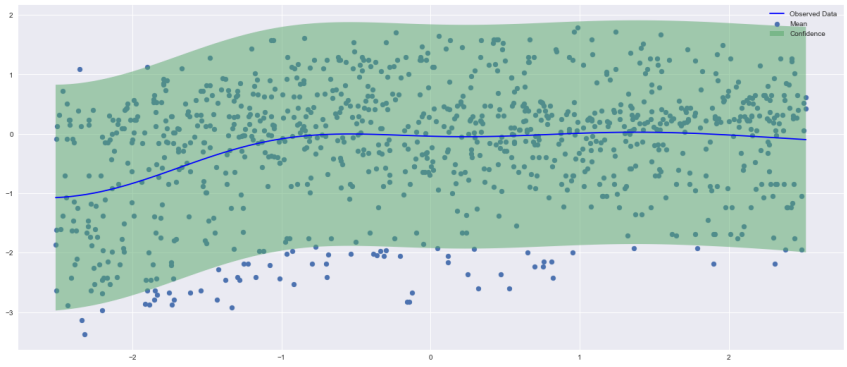

Figure 5.1: GP uncertainty level without applying MFBO (wing 1 only)

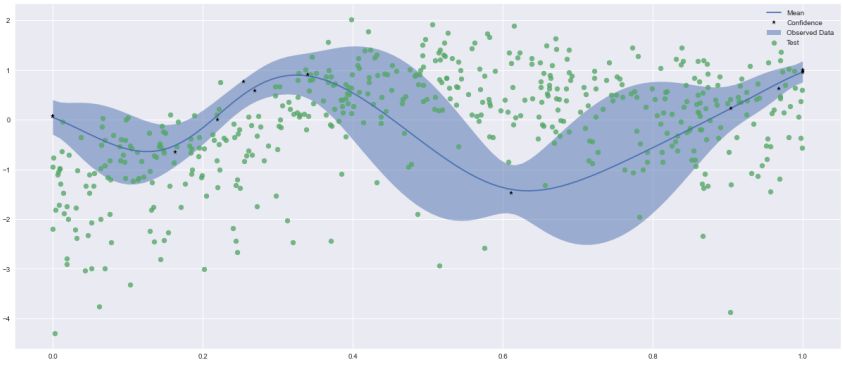

Figure 5.2: EI acquisition function output with not well tuned parameters (wing 1 only)

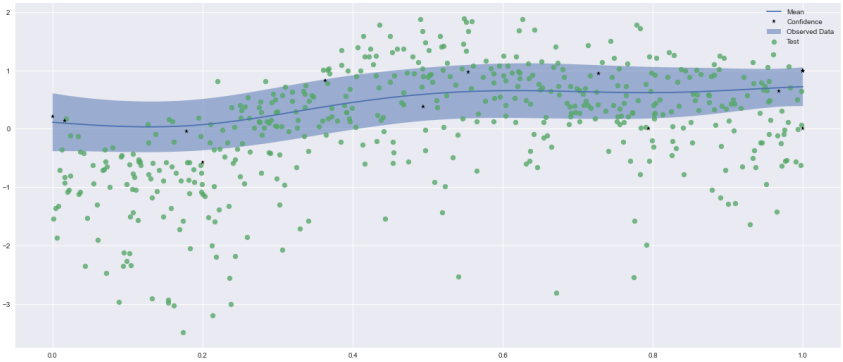

Figure 5.3: EI acquisition function output (wing 0 only)

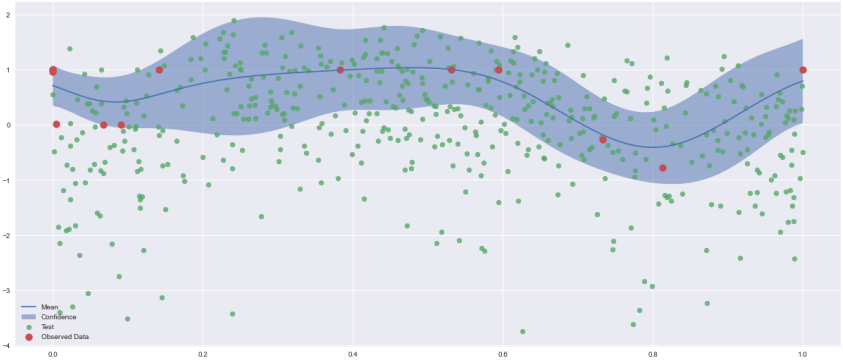

Figure 5.4: PI acquisition function output (wing 0 only)

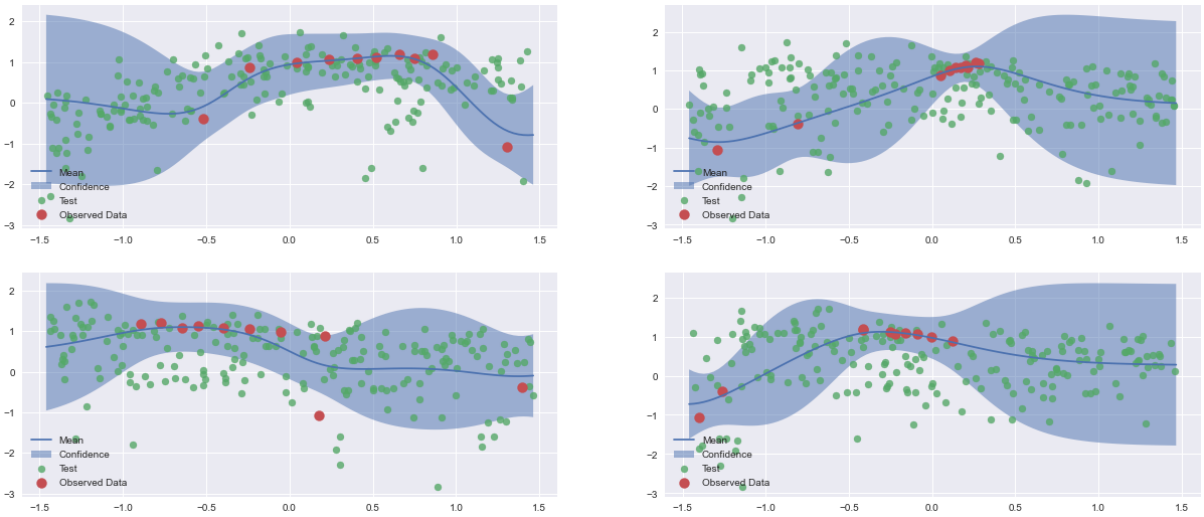

Figure 5.5: qKG acquisition function output interrupt before convergence after reaching maximum time threshold

without using any acquisition function. As we can see the sampled points are homogeneous spread in the definition space and outside the uncertainty region, they not provide any information.

Figure (5.7) shows the result provided by UCB acquisition function which are not quite reliable, even after several attempts to tune the parameter λ.

Figure (5.8) shows the result provided by PI acquisition function, as in the single input, the results are quite promising.

Figure (5.9) shows the result provided by EI acquisition function, as in the single input, the results are quite promising.

Figure (5.10) shows the result provided by qEI acquisition function, and still the results are quite promising, but not as well as the one provided by PI and EI.

Figure (5.11) shows the result provided by qNEI acquisition function, and still the results are quite promising, but not as well as the one provided by PI and EI.

We compare the results created be six acquisition functions, with PI and EI performing best.

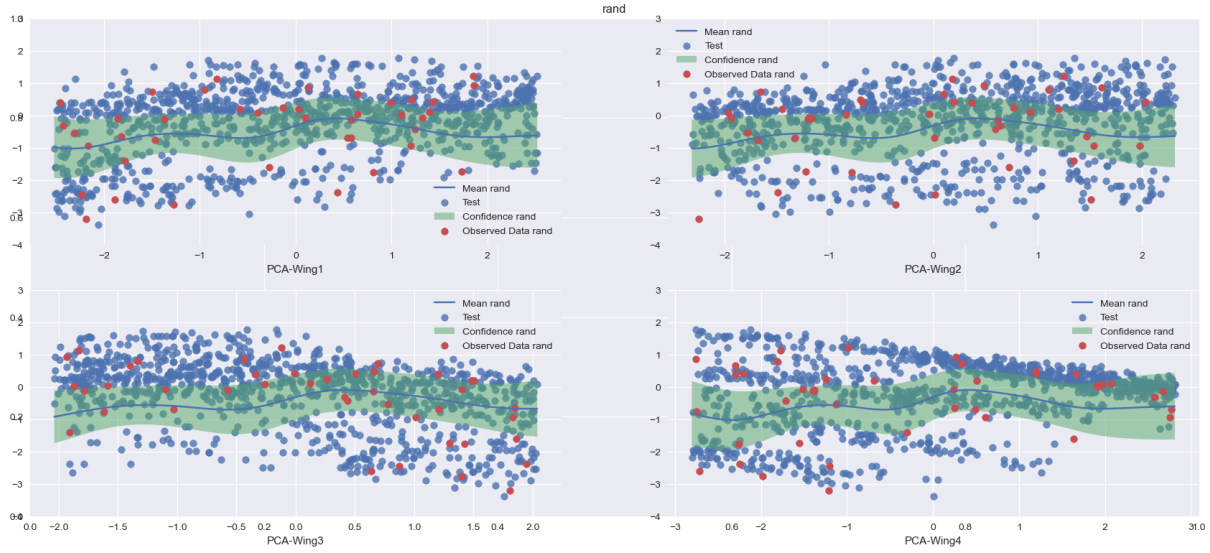

Figure 5.6: random choice for sampling the next point without acquisition function output

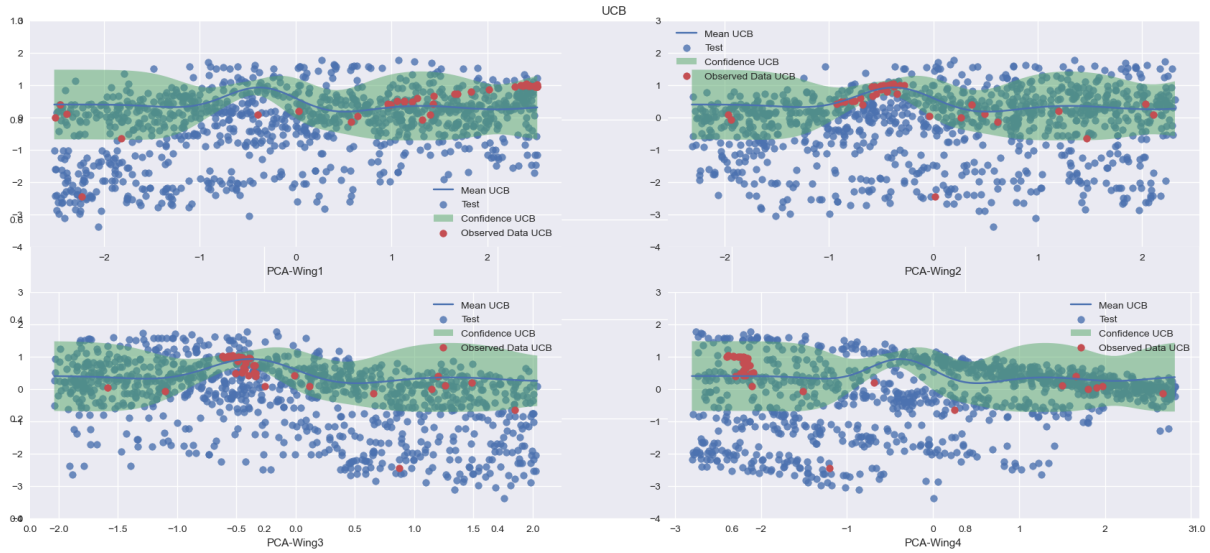

Figure 5.7: UCB acquisition function output

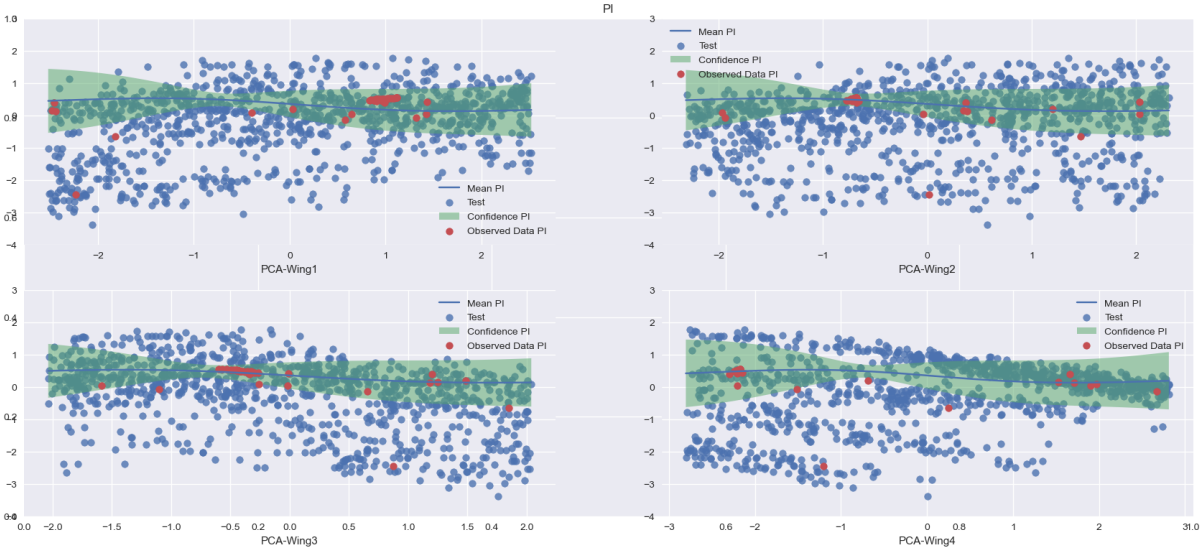

Figure 5.8: PI acquisition function output

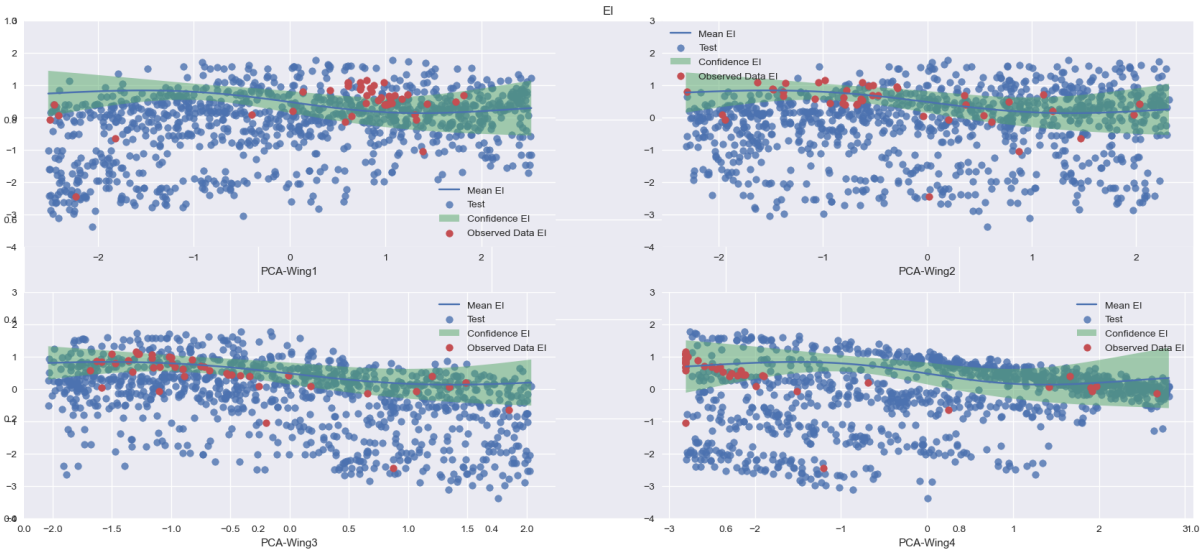

Figure 5.9: EI acquisition function output

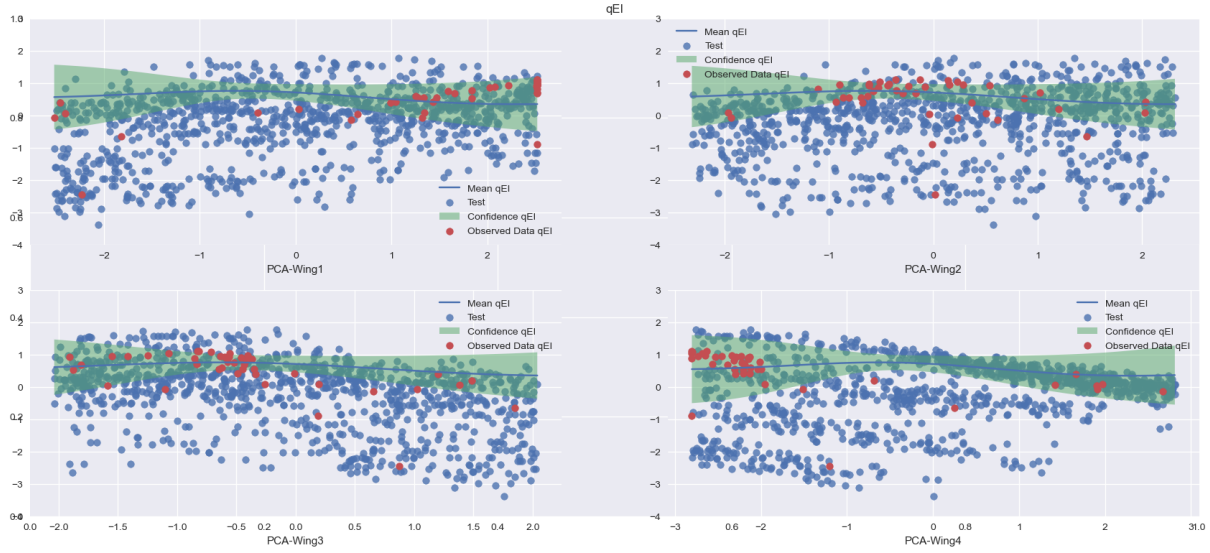

Figure 5.10: qEI acquisition function output

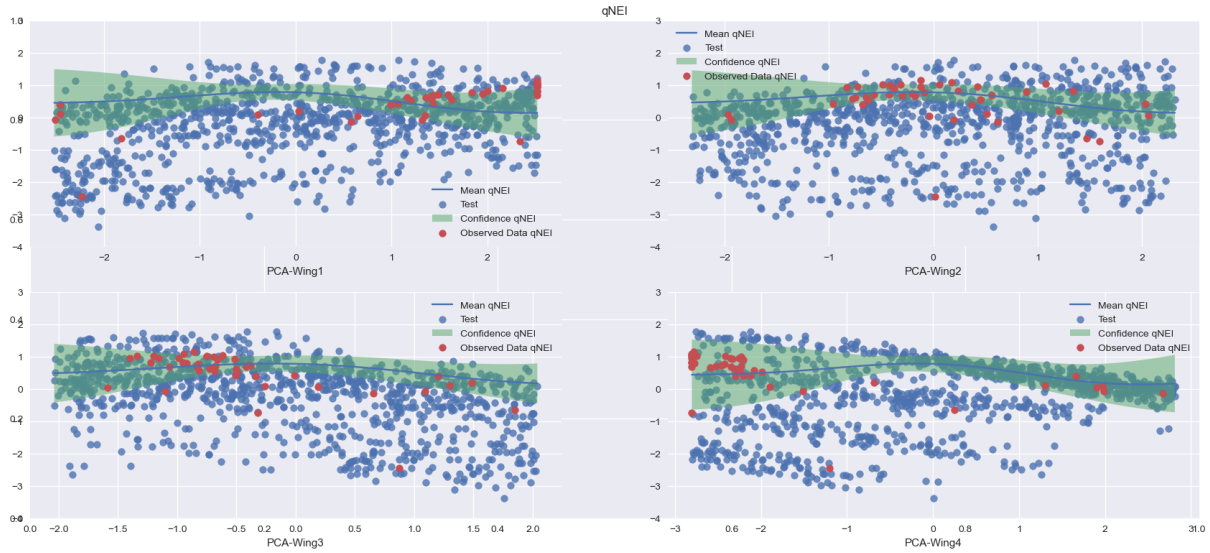

Figure 5.11: qNEI acquisition function output

6 | Conclusions and future developments

Overall, MFBO is a powerful approach to global optimization that can provide significant benefits in a wide range of applications, but requires careful attention to the selection and calibration of the surrogate model, acquisition function, and high fidelity model parameters to ensure good performance:

- In the surrogate model, it's important to note that the choice of mean function is not the only factor that determines the behavior of the GP model. The kernel function is equally important and can have a significant impact on the surrogate model's predictive performance. Together, the mean and covariance functions define the properties of the distribution over functions that we believe could be generating our data. The optimization of hyperparameters is an important step in GP regression as it determines the properties of the distribution over functions that the model will use to make predictions. By tuning the hyperparameters to match our prior beliefs about the function being modeled, we can improve the model's predictive performance and its ability to generalize to new data.
- In the BO, the choice of the acquisition function affects the accuracy of the final model. In the implemented method, a list of different acquisition functions is presented in order to provide an effective overview of the possible performance. It is worth to mention that changing the initial problem, as well as changing the input data, will most likely the performance results of the acquisition functions.
- Last but not least, the linear neural networks has shown to be a powerful and effective high-fidelity model in MFBO, providing a fast and accurate surrogate for the objective function that can significantly reduce the computational cost of the optimization process. However, care must be taken in the design and implementation of the network to ensure good performance and robustness.

6.1. Geo data framework

Since the purpose of the project is to develop the required working script, it was chosen to provide this additional section with simple insights as not to limit it by the tools available up to date.
Geo data are expensive to manage, process and visualize. Their heavy dimensions make them computational prohibitive to work with for the average user machine, this method tries to make them more accessible.
From the optimization perspective the structure of MFBO makes itself suitable for proposing a solution to this task (from a statistical point of view).
In this last task we adapt the method submitted above to the ERA5 sample dataset, which is the Latest Climate Reanalysis Produced by ECMWF.
It was possible to communicate with the Copernicus dataset through the CDS API, Climate Data Store Application Program Interface made available by the Copernicus Climate Change Service, through a json file request in python.

Moving forward, we plan to explore the use of Google Earth Engine for processing and managing the ERA5 dataset. By utilizing this powerful tool, we aim to streamline the data management process and reduce the time required for processing. With its ability to handle large datasets and perform complex analyses, Google Earth Engine has the potential to greatly enhance our research capabilities.

Our first step will be to conduct a thorough evaluation of Google Earth Engine's capabilities and determine how best to integrate it into our workflow. We will also work to gain a deeper understanding of the ERA5 dataset and how it can be optimized for use with Google Earth Engine. Additionally, we will seek out training and support to ensure that we are utilizing the platform to its full potential.

Stochastic optimization for scientific computing can concretely offer in the climate and environmental context, and Geoinformatics tools provide support for bridging these two fields. Overall, we believe that incorporating Google Earth Engine into our workflow will be a significant step forward in our ability to manage and process large datasets. We look forward to exploring this exciting new technology and the benefits it can bring to our research.

Bibliography

[1] L. Bonfiglio. Optimization and performance predictions using machine learning and stochastic multi-fidelity modeling, 2018. URL `https://web.mit.edu/~bonfi/www/multi-fidelity.html`.

[2] P. I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811v1*, 2018.

[3] P. Hennig, M. A. Osborne, and H. P. Kersting. Probabilist numerics. *Cambridge University Press*, 2022. www.cambridge.org/.

[4] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy. Constrained bayesian optimization with noisy experiments. *arXiv preprint arXiv:1706.07094v2*, 2018.

[5] S. Li, W. Xing, R. M. Kirby, and S. Zhe. Multi-fidelity bayesian optimization via deep neural networks experiments. *arXiv preprint arXiv:2007.03117v4*, 2020.

[6] C. E. Rasmussen and C. K. I. Williams. Gaussian processes for machine learning. *The MIT Press*, pages 79–128, 2006. www.GaussianProcess.org/gpml.

[7] J. Wu and P. I. Frazier. The parallel knowledge gradient method for batch bayesian optimization. *NIPS Workshop on Bayesian Optimization*, 2016.

[8] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.