# Chapter 4

# Bayesian optimization implementation

## 4.1 Gaussian Process

To describe the library and the bayesian optimization algorithm we first need to briefly cover the Gaussian process approach.
Gaussian processes are a probabilistic way to estimate and learn a correlation from a given set of data (variables) to an unknow function using multivariate Gaussian distribution over the mean and variance of those variables, thus defining the Gaussian process as a subset of variables having their joint distribution as its basis.
They are used in Bayesian optimization to solve a regression problem that ultimately aims to finds the function projected by the observed data.

The input of Gaussian processes is some dataset $X$ mapped to their respective evaluations given by an unknow function $f(\mathbf{x})$. This defines a collection of points of size $D$ that can be expressed as: $Data = \{(\mathbf{x_1}, f(\mathbf{x_1})), (\mathbf{x_2}, f(\mathbf{x_2})), \ldots, (\mathbf{x}_D, f(\mathbf{x}_D))\}$.
The next step is to define all the function evaluations as a multivariate Gaussian distribution that uses the $X$ dataset as prior information. Given a mean function $m(\mathbf{x})$ and a covariance function $K(\mathbf{x}, \mathbf{x}')$ we can express the posterior joint distribution as follows.

$$\begin{bmatrix} p(f(\mathbf{x_1})) \\ p(f(\mathbf{x_2})) \\ \vdots \\ p(f(\mathbf{x}_D)) \end{bmatrix} = \mathcal{N}\left( \begin{bmatrix} m(\mathbf{x_1}) \\ m(\mathbf{x_2}) \\ \vdots \\ m(\mathbf{x}_D) \end{bmatrix}, \begin{bmatrix} K(\mathbf{x_1}, \mathbf{x_1}) & K(\mathbf{x_1}, \mathbf{x_2}) & \ldots & K(\mathbf{x_1}, \mathbf{x}_D) \\ K(\mathbf{x_2}, \mathbf{x_1}) & K(\mathbf{x_2}, \mathbf{x_2}) & \ldots & K(\mathbf{x_2}, \mathbf{x}_D) \\ \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_D, \mathbf{x_1}) & K(\mathbf{x}_D, \mathbf{x_2}) & \ldots & K(\mathbf{x}_D, \mathbf{x}_D) \end{bmatrix} \right) \tag{4.1}$$

that can be simplified to

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}')) \tag{4.2}$$

where

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \tag{4.3}$$

$$K(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))^T] \tag{4.4}$$

The covariance function (or kernel) is the core of the Gaussian process. It hold all the information about the unknow function we want to learn and provide the definition of similarity applied on the domain of supervised learning (e.g. closer $\mathbf{x}$ have closer $f(\mathbf{x})$). These functions are non-trivial and present many hyperparameters which are further optimized to learn the unknown function better.

Gaussian processes are able through the joint distribution of the considered variables to sample a function $\mathbf{f_i}$ over a continuous domain.
By sampling many functions in this way we obtain the following graph.

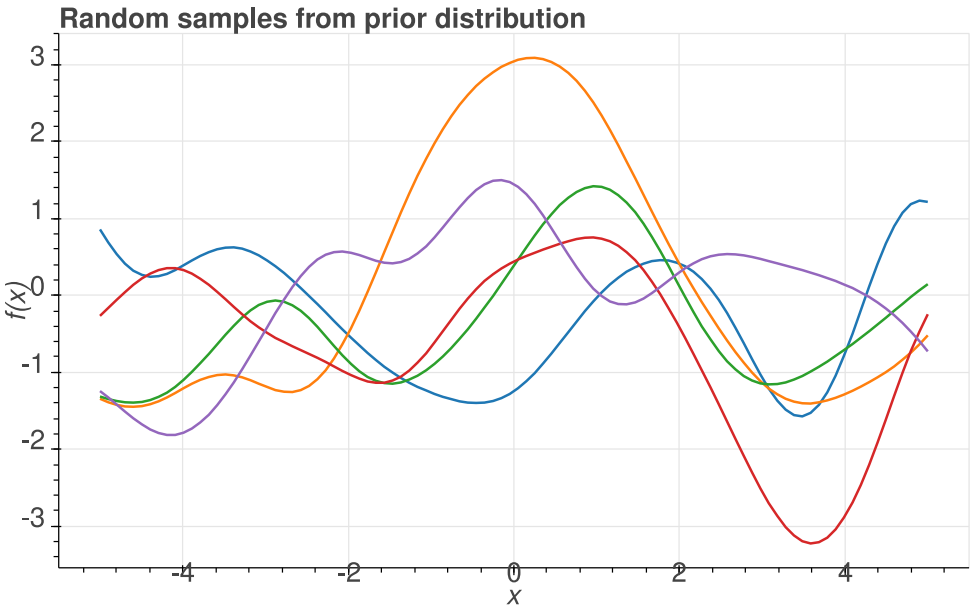



Figure 4.1: Sampling of functions without prior data

By adding the data that we collected before as points and fitting the model through the calculation of the covariance matrix the graph gets 'smoother'.

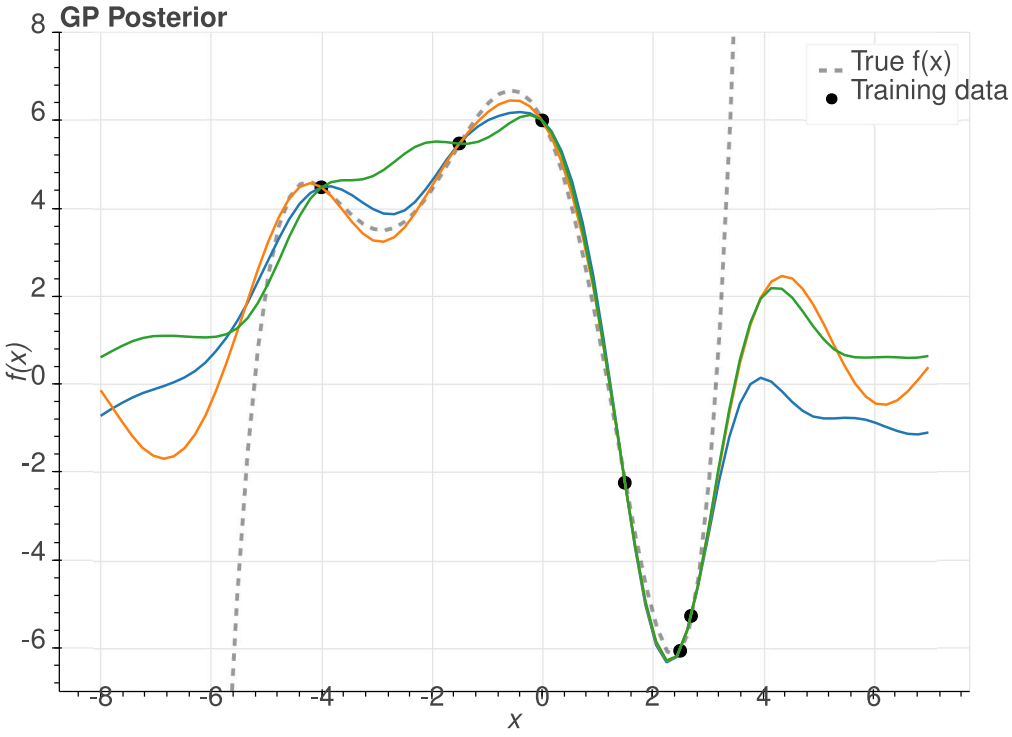



Figure 4.2: Sampling of functions from the GP posterior

We can observe that now every function crosses the specified data points and in the proximity of them there is a very low variance.
The mean and the variance between these functions are the modelled guess of the Gaussian process of the unknow function, where the variance is seen as the uncertainty of the model.

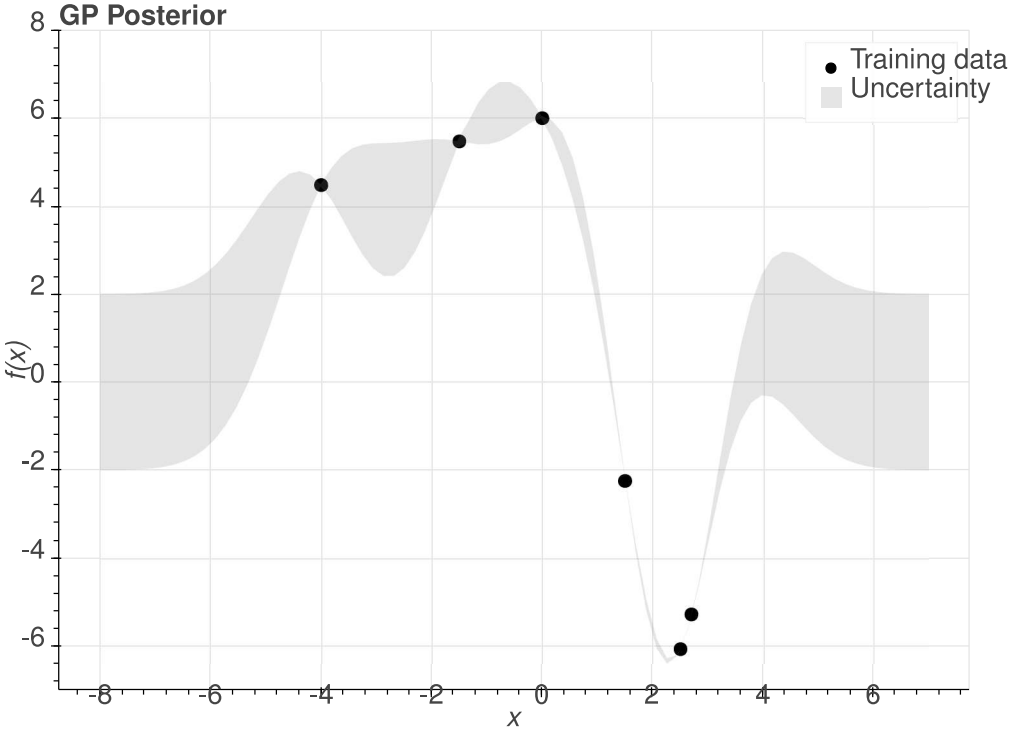



Figure 4.3: Gaussian process model function

### 4.1.1 Noiseless predictions

With the basics explained until now we are ready to make predictions using the Gaussian process model.

We want to predict the values $f(\mathbf{x}_\star)$ of a new test dataset $X_\star$ of size $N$.

This is done by computing the joint probability of all the previous $f(\mathbf{x})$ with the constraint on $X$.

$$\begin{bmatrix} p(f(\mathbf{x_1})) \\ p(f(\mathbf{x_2})) \\ \vdots \\ p(f(\mathbf{x}_D)) \\ p(f(\mathbf{x}_{\star\mathbf{1}})) \\ p(f(\mathbf{x}_{\star\mathbf{2}})) \\ \vdots \\ p(f(\mathbf{x}_{\star N})) \end{bmatrix} = \mathcal{N}\left( \begin{bmatrix} m(\mathbf{x_1}) \\ m(\mathbf{x_2}) \\ \vdots \\ m(\mathbf{x}_D) \\ m(\mathbf{x}_{\star\mathbf{1}}) \\ m(\mathbf{x}_{\star\mathbf{2}}) \\ \vdots \\ m(\mathbf{x}_{\star N}) \end{bmatrix}, K_{ext} \right) \tag{4.5}$$

Where the matrix $K_{ext}$ is an extension of the previous $K(X, X)$ matrix with the new dataset:

$$K_{ext} = \begin{pmatrix} K(\mathbf{x_1},\mathbf{x_1}) & K(\mathbf{x_1},\mathbf{x_2}) & \dots & K(\mathbf{x_1},\mathbf{x}_D) & K(\mathbf{x_1},\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x_1},\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x_1},\mathbf{x}_{\star N}) \\ K(\mathbf{x_2},\mathbf{x_1}) & K(\mathbf{x_2},\mathbf{x_2}) & \dots & K(\mathbf{x_2},\mathbf{x}_D) & K(\mathbf{x_2},\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x_2},\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x_2},\mathbf{x}_{\star N}) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_D,\mathbf{x_1}) & K(\mathbf{x}_D,\mathbf{x_2}) & \dots & K(\mathbf{x}_D,\mathbf{x}_D) & K(\mathbf{x}_D,\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x}_D,\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x}_D,\mathbf{x}_{\star N}) \\ K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x_1}) & K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x_2}) & \dots & K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x}_D) & K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x}_{\star\mathbf{1}},\mathbf{x}_{\star N}) \\ K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x_1}) & K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x_2}) & \dots & K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x}_D) & K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x}_{\star\mathbf{2}},\mathbf{x}_{\star N}) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_{\star N},\mathbf{x_1}) & K(\mathbf{x}_{\star N},\mathbf{x_2}) & \dots & K(\mathbf{x}_{\star N},\mathbf{x}_D) & K(\mathbf{x}_{\star N},\mathbf{x}_{\star\mathbf{1}}) & K(\mathbf{x}_{\star N},\mathbf{x}_{\star\mathbf{2}}) & \dots & K(\mathbf{x}_{\star N},\mathbf{x}_{\star N}) \end{pmatrix} \tag{4.6}$$

Which can be further summarized as follows.

$$\begin{bmatrix} f(X) \\ f(X_\star) \end{bmatrix} \sim \mathcal{N}\left( \begin{bmatrix} m(X) \\ m(X_\star) \end{bmatrix}, \begin{bmatrix} K(X,X) & K(X,X_\star) \\ K(X_\star,X) & K(X_\star,X_\star) \end{bmatrix} \right)$$
$$\sim \mathcal{N}\left( \begin{bmatrix} m(X) \\ m(X_\star) \end{bmatrix}, \begin{bmatrix} K & K_\star \\ K_\star^T & K_{\star\star} \end{bmatrix} \right)$$

The last step is to compute the probability distribution of $p(f(X_\star)|X_\star, X, f(X))$ as we are not in need of the prediction of $f(X)$ given by the normal model, which is a known dataset.

$$\begin{aligned} p(f(X_\star)|X_\star, X, f(X)) &= \mathcal{N}(\mu_\star, \Sigma_\star) \\ \mu_\star &= m(X_\star) + K_\star^T K^{-1}(f(X) - m(X)) \\ \Sigma_\star &= K_{\star\star} - K_\star^T K^{-1} K_\star \end{aligned}$$

### 4.1.2 Noisy predictions

Most of the time, the function we are trying to model will have a noise: $y = f(\mathbf{x}) + \epsilon$.
We assume that the noise $\epsilon$ is indipendent and is derivated from another Gaussian distribution with zero mean and a variance of $\sigma^2$.

$$\epsilon \sim \mathcal{N}\left(\mathbf{0}, \sigma^2\right) \tag{4.7}$$

The joint distribution thus becomes:

$$\begin{bmatrix} y \\ f(X_\star) \end{bmatrix} \sim \mathcal{N}\left( \begin{bmatrix} m(X) \\ m(X_\star) \end{bmatrix}, \begin{bmatrix} K + \sigma^2 I & K_\star \\ K_\star^T & K_{\star\star} \end{bmatrix} \right) \tag{4.8}$$

Which further changes the predictive conditional probability distribution $p(f(X_\star)|X_\star, X, y)$ as follows:

$$\begin{aligned} p(f(X_\star)|X_\star, X, y) &= \mathcal{N}(\mu_\star, \Sigma_\star) \\ \mu_\star &= m(X_\star) + K_\star^T (K + \sigma^2 I)^{-1}(y - m(X)) \\ \Sigma_\star &= K_{\star\star} - K_\star^T (K + \sigma^2 I)^{-1} K_\star \end{aligned}$$

## 4.2 Bayesian optimization

Bayesian optimization is the application of a Gaussian process in a step-wise optimization of unknow functions.
The objective of a Bayesian optimization is to find a point $\mathbf{x}$ for which its unknow function evaluation $f(\mathbf{x})$ is minimum (or analogically maximum).

$$\begin{aligned} \mathbf{x}_{min} &= \arg\min_{\mathbf{x}\in X} f(\mathbf{x}) \\ f &: X \to \mathbb{R} \\ X &\subseteq \mathbb{R}^n \end{aligned}$$

This point could also be obtained through the means of standard optimization techniques, but Bayesian optimization excels when the target function is unknown, noisy and/or very expensive to evaluate.

The algorithm starts with a given set of points $X$ (at least one) and their evaluations $Y$. We use the Guassian process to model the posterior distribution over functions and then, given an acquisition function, we evaluate all the possible next points $X_\star$ by optimizing it. The acquisition function is the focus of the assumptions taken by a Bayesian optimization instance, given a point it takes as input the mean and the variance predicted by the current Gaussian process model on said point and returns an evaluation for it that combines exploitation (mean) and exploration (variance). The best estimated new point(s) will then be added to the set $X$ and their "manual" evaluation (obtained through the unknow real function, which context may vary depending on the application of the Bayesian optimization) will be added to $Y$. All of these steps are then repeated in a loop until an optimal new point is reached.

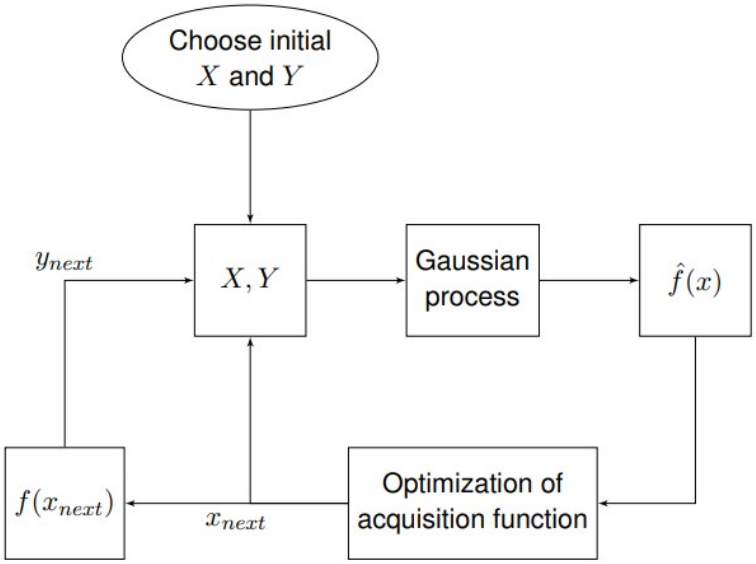



Figure 4.4: Bayesian optimization flow chart

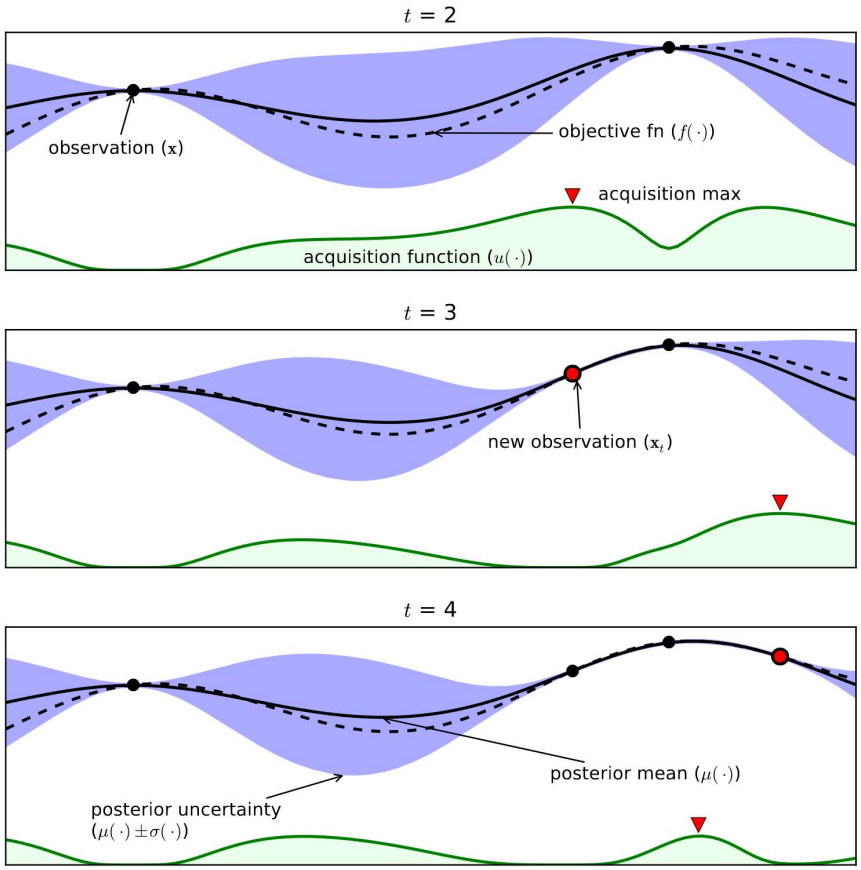



Figure 4.5: Bayesian optimization example

## 4.3 Technical documentation of the BOCore library

### 4.3.1 Creating the optimization instance

A bayesian optimization instance must be instantiated through its constructor with the following parameters:

- **fvars_x:**

  Python list containing all the information regarding the domain of each input structured as the following example:

```
fvarsX =
[{'name':'x1', 'type':'continuous', 'domain': (-5, 10)},
 {'name':'x2', 'type':'discrete', 'domain': np.arange(0., 15, 0.5)},
 {'name':'x3', 'type':'categorical', 'domain': ['yes', 'no']},
 {'name':'x4', 'type':'continuous', 'domain': (2.2, 8.3)}]
```

- **fvars_y:**

  Python list containing all the information regarding the domain of each output structured like *fvars_x*.
  Type and domain are not used at the moment, but the parameter is needed for its cardinality used in optimization setups.

- **g (Optional):**

  Must contain a pointer to a python function that takes the points with their evaluations and the model(s) as parameters and returns a single numerical evaluation for every point.
  This parameter is necessary for the optimization when using more than one evaluation output, as the model must know how those evaluations are combined to reach a general score for every point. An example of general function that combines all the evaluation (in this case two) for each point could be specified as follows.

```
def my_fun(X, Y, Models):
    return Y[:, 0] + Y[:, 1]
```

- **constraints (Optional):**

  Python list containing pointers to functions that take as input an array of points and must return an array of evaluations for each point: a positive value means the point

satisfies the imposed constraint, a zero or negative value means the point violates the constraint.
When performing an optimization step, new points that do not satisfy the constraint will have assigned the worst acquisition value possible, making them most likely not able to be proposed as a possible next iteration point.

- **objectives (Optional):**

  Array of pointers to functions (one for each output evaluation) used to automaically evaluate points generated randomly at the beginning of an optimization instance.
  The input of the function is an array of points and the output must be an array of evaluations of said points.
  If not specified, when starting with random points, the optimization returns an array of said points to be evaluated and added manually to the model when performing the first step.

- **acquisition_type (Optional):**

  The acquisition function used in the bayesian optimization, currently supports: 'EI', 'EIMC', 'MU', 'LCB', and 'UCB'. Defaulted to 'EI'.
  EIMC is currently the only acquisition function that supports multiple evaluation outputs.
  When using 'EI' an additional parameter can be included in ***kwargs*:
  - acquisition_jitter: jitter used to compute the acquisition, defaulted to 0.

- **acquisition_optimizer_type (Optional):**

  The acquisition optimizer used in the bayesian optimization, currently supports: 'lbfgsb', 'de', and 'slsqp'. Defaulted to 'lbfgsb'.

- **model_type (Optional):**

  The model used in the bayesian optimization, currently it only supports 'GP' (Gaussian Process) which is the model developed through the GPy library. Defaulted to 'GP'.
  When using the 'GP' model some additional parameter can be included in ***kwargs*:
  - ARD: boolean kernel setting, defaulted to True.
  - optimize_restarts: restarts of the model fitting, defaulted to 10.
  - model_optimizer_type: optimizer type for model fitting, defaulted to 'lbfgsb'.
  - kernel_variance: starting value of the kernel's variance, defaulted to 1.
  - kernel_variance_bounds: bounds on which the *kernel_variance* parameter can be internally optimized.

- kernel_lengthscale: starting value of the kernel's lengthscale, defaulted to None.
- kernel_lengthscale_bounds: bounds on which the *kernel_lengthscale* parameter can be internally optimized.
- noise_variance: starting value of the gaussian process' noise variance , defaulted to 1.
- noise_variance_bounds: bounds on which the *noise_variance* parameter can be internally optimized.
- max_iters: maximum iterations for fitting the model, defaulted to 1000.

- **kernel_type (Optional):**

  The kernel used to fit the model, kernels supported are tied with the respective model implementation. Defaulted to 'Matern52'.

- **models (Optional):**

  Model(s) as dictionaries, if specified it will load them as models to use in the optimization instead of creating new ones.

- **X_init (Optional):**

  Initial points to fit the model(s), if specified they will be used instead of creating new ones with the *init_n_points* parameter.

- **Y_init (Optional):**

  Initial evaluations of the points to fit the model(s), if specified they will be used instead of creating new ones with the *init_n_points* parameter.

- **init_n_samples (Optional):**

  Number of points to generate as an initial sample when none are passed through the parameters *X_init* and *Y_init*, defaulted to 5.

- **samples_generator (Optional):**

  Mode of the random points generator, currently supports: 'random', and 'bruteforce'. Defaulted to 'random'.

- **norm_x (Optional):**

  Boolean flag that indicates if the points used to fit the model should be normalized to perform internal operations, defaulted to True.

- **norm_y (Optional):**

  Boolean flag that indicates if the evalkuations of the points should be normalized to perform internal model operations, defaulted to True.

- **evaluator_type (Optional):**

  Strategy used to optimize the best anchor points obtained in the bayesian optimization, currently supports: 'sequential_batch', and 'thompson'. Defaulted to 'sequential_batch'.

- **batch_size (Optional):**

  Number of points returned after an iteration of the optimization, currently only supports 1 point. Defaulted to 1.

- **anchor_points_samples (Optional):**

  Number of anchor points generated when searching the best points over the domain of the acquisition function, defaulted to 1000.

- **transfer_learning_model (Optional):**

  An instance of another bayesian optimization object used to abilitate all the transfer learning features.

### 4.3.2 Running the optimization

Running the optimization can be performed with the *run_optimization* method.
The method uses the current data present in the fitted model to perform a step of bayesian optimization that returns a new point.
The parameters of the method are the following:

- **n_random_anchor_points (Optional):**

  Integer value used to select the number of 'best' anchor points selected through the acquisition function in the bayesian optimization, defaulted to 5.

- **anchor_points_samples (Optional):**

  List of points that can be used to select the anchor points instead of generating the selection list randomly.

- **selective_percentage (Optional):**

  Double percentage value between 0 and 1 used to separate the domain of the selected anchor points, defaulted to 0.
  For each dimension of the selected anchor points they must have a difference of at least the specified percentage between them, thus spreading the selection over the domain.

- **maxiter (Optional):**

  The maximum number of iterations allowed to optimize the anchor points, defaulted to 1000.

- **epsilon (Optional):**

  Hyperparameter used in the anchor point optimization TODO: describe.

- **exploration (Optional):**

  Boolean flag that specifies if the internal acquisition function should be computed in exploration or exploitation mode, defaulted to True (exploration)

### 4.3.3 Acquiring anchor points

It is possible to acquire anchor points by using the current fitted model through the *generate_anchor_points* method.
The parameters of the method are the following:

- **num_anchor (Optional):**

  Integer value, the number of returned anchor points, defaulted to 5.

- **anchor_points_samples (Optional):**

  List of points that can be used to select the anchor points instead of generating the selection list randomly.

- **selective_percentage (Optional):**

  Double percentage value between 0 and 1 used to separate the domain of the selected anchor points, defaulted to 0.
  For each dimension of the points, they will have a difference of at least the specified percentage between them, thus spreading the selection over the domain.

- **norm (Optional):**

  Boolean flag that specifies if the returned point(s) should be normalized over the domain, defaulted to False.

- **exploration (Optional):**

  Boolean flag that specifies if the internal acquisition function should be computed in exploration or exploitation mode, defaulted to True (exploration)

### 4.3.4 Adding new evaluations

Adding data to the optimization model is done through the *add_evaluations* method that takes a point, or an array of points, and their respective evaluations as input.
For example, to add two points each one evaluated over three outputs, their format should be:
Points: [[x1_1,x1_2,x1_3,x1_4], [x2_1,x2_2,x2_3,x2_4]]
Evaluations: [[y1_1, y1_2], [y2_1, y2_2], [y3_1, y3_2]]

### 4.3.5 Predicting new points

Model's predictions are performed with the *predict* method that takes a point, or an array of points as input and returns an array containing the predicted mean and variance of each point, one for each model currently being optimized.

## 4.4 Practical example

In this section we will apply our library step-by-step to implement a bayesian optimization of a given problem.
The problem we are considering is a minimization of the following unknown function:

$$f(x_1, x_2) = y = x_1^2 + x_2^2 \tag{4.9}$$

given two input domains $X_1 \in [-10, 10]$ and $X_2 \in [-10, 10]$.
Said domains are specified in the code as following.

```
fvarsX = [{'name': 'x1', 'type': 'continuous', 'domain': (-10, 10)},
          {'name': 'x2', 'type': 'continuous', 'domain': (-10, 10)}]
fvarsY = [{'name': 'y', 'type': 'continuous', 'domain': ()}]
```

We can observe that the optimum is obtained with $x_1 = 0$ and $x_2 = 0$.
The unknown function is implemented as a "black box" function that returns evaluations of

the specified points.

```
def out(X):
    points_eval = []
    for x in X:
        x1 = x[0]
        x2 = x[1]
        r = x1**2 + x2**2
        points_eval.append(r)
    return points_eval
```

Next we instantiate a *BayesianOptimization* object with the following parameters:

```
my_bo = BO.BayesianOptimization(fvars_x=fvarsX, fvars_y=fvarsY,
                                objectives=[out], init_n_samples=1)
```

*out* is used to evaluate the initial randomly generated samples.

To obtain the next point calculated by the Bayesian optimization we use the *run_optimization* method.

```
x_next = my_bo.run_optimization()
```

Once we obtain the first point we can create a loop to automatically perform our iterations.

```
iterations = 10
for i in range(iterations):
    my_bo.add_evaluations(x_next, out1(x_next))
    x_next = my_bo.run_optimization()
```

Given the plot of the original function, we compare it to the plots of the model over the course of the iterations with the associated selected points.

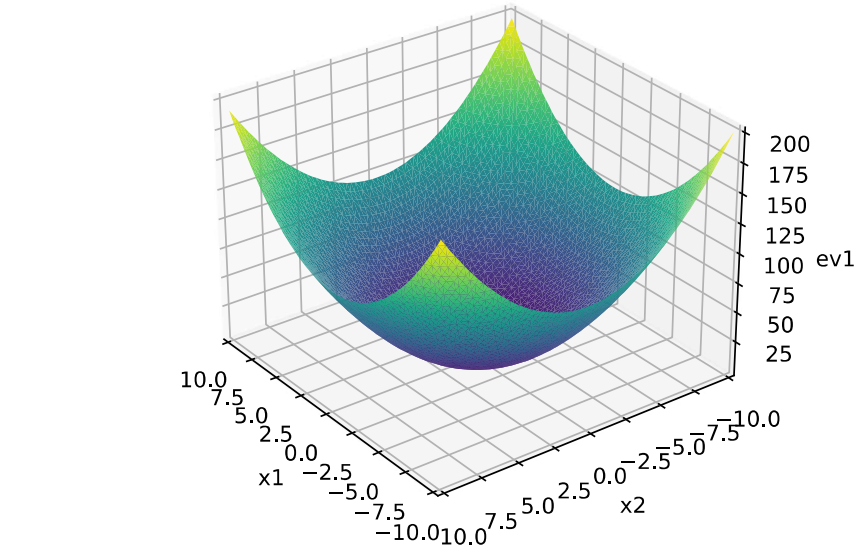



Figure 4.6: Plot of unknow function over the domains

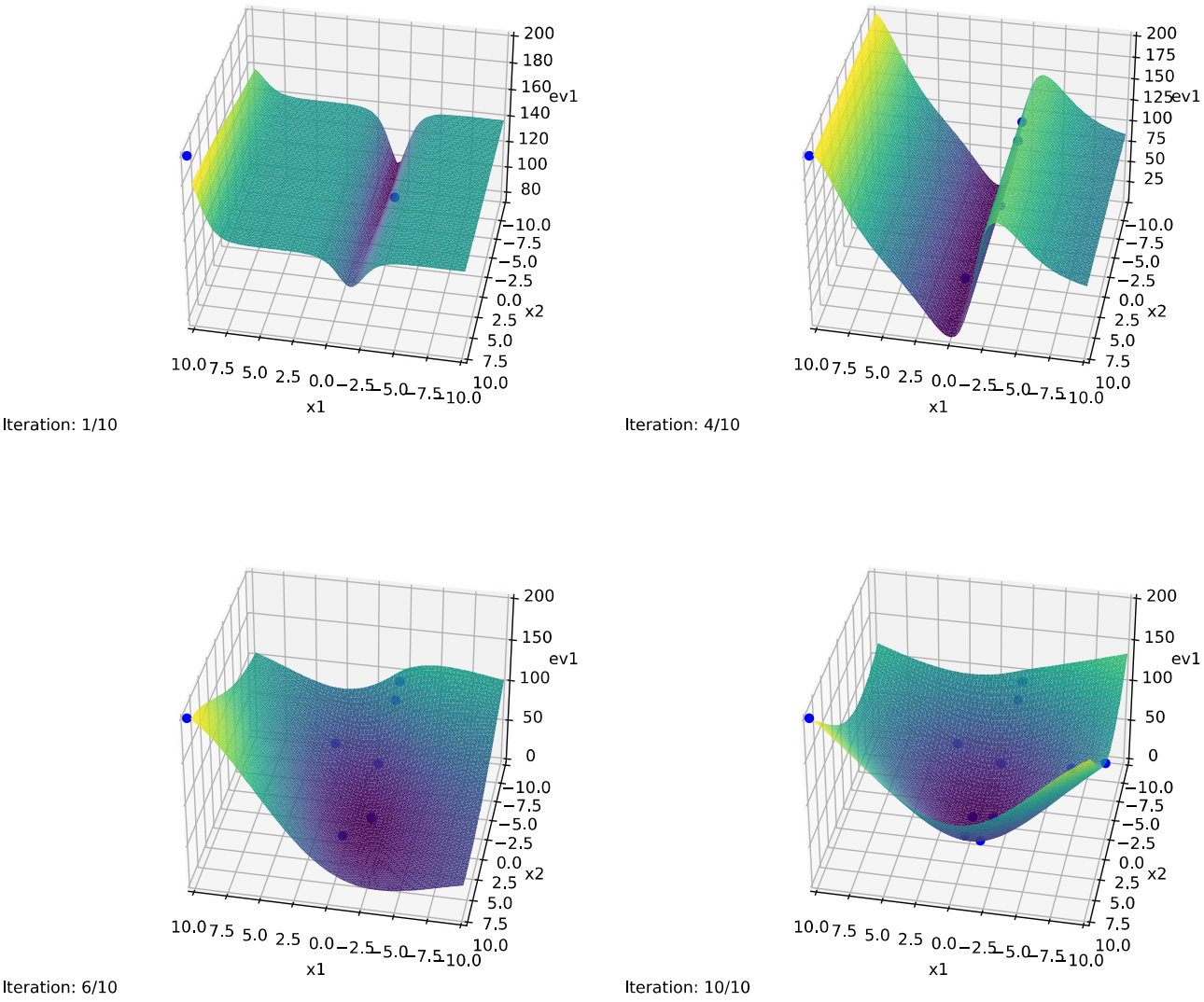
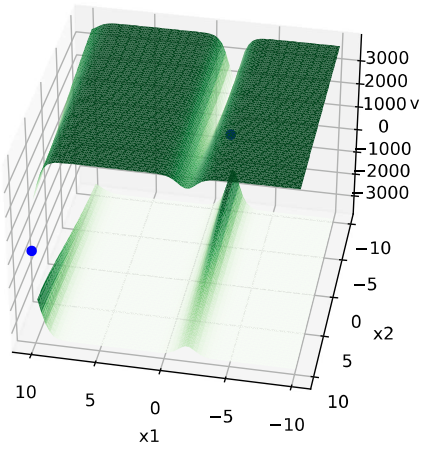
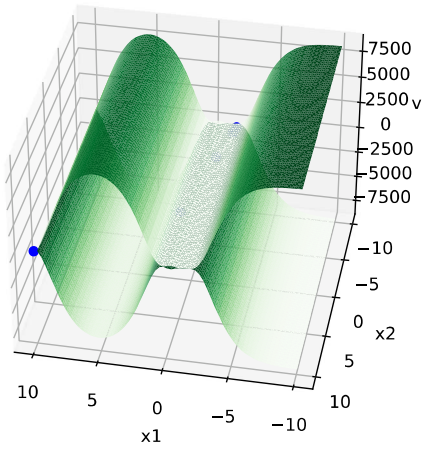


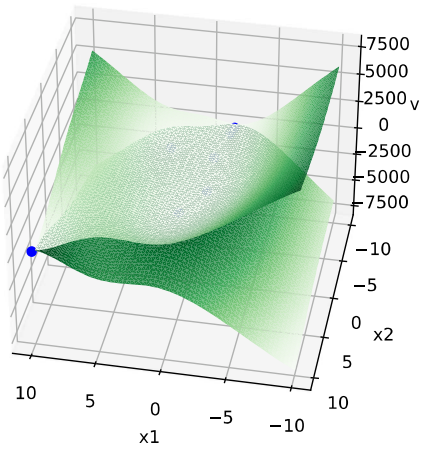

Figure 4.7: Predicted values of the model over the course of the iterations



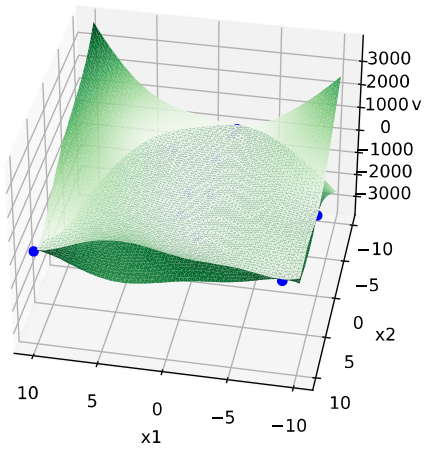

Iteration: 1/10




Iteration: 4/10




Iteration: 6/10




Iteration: 10/10

Figure 4.8: Uncertainty of the model over the course of the iterations

As we can see, after ten iterations the predicted model will strongly resemble the original function and as soon as the model structure is estimated correctly, an optimum point will be selected accordingly by the Bayesian optimization algorithm.