Michael Hadwiger, BSc

Bayesian Global Optimization with Derivative Observation for Stellarator Design

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Technical Physics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.rer.nat. Christopher Albert

Institute of Theoretical and Computational Physics

Graz, August 2024

Abstract

This thesis develops Bayesian methods for parameter optimization, with both general usage and specific application on stellarator fusion reactor configurations. In contrast to conventional approaches not only values, but also derivatives of the target function are incorporated. The impact of including derivative information on the performance of Bayesian global optimizers is investigated for the Expected Improvement (EI) and Knowledge Gradient (KG) acquisition function. In addition, the convergence of the Bayesian optimizer is compared to local methods that rely on gradient information. The derivative enabled Bayesian Optimization was implemented in Python using the Python package *BoTorch*. Apart from two toy models the developed framework was used to optimize the coil currents of a stellarator with *Simsopt*. The obtained results of the Bayesian Optimization were slightly better than those of the L-BFGS-B method, until the Gaussian process regression reached its maximum accuracy. Moreover, an interactive tool for exploratory data and model analysis was further developed using *plotly/dash*.

Kurzfassung

In dieser Arbeit werden Bayes'sche Methoden zur Parameteroptimierung für den allgemeinen Gebrauch und die spezielle Anwendung auf Stellarator-Fusionsreaktor-Konfigurationen entwickelt. Im Gegensatz zu herkömmlichen Ansätzen werden nicht nur Funktionswerte, sondern auch Ableitungen der Zielfunktion einbezogen. Die Auswirkung der Einbeziehung von Ableitungsinformationen auf die Leistung von Bayes'schen globalen Optimierern wird für die Akquisitionsfunktionen Expected Improvement (EI) und Knowledge Gradient (KG) untersucht. Des weiteren wird die Konvergenz des Bayes'schen Optimierers mit lokalen Methoden verglichen, die auf Gradienteninformationen beruhen. Die abgeleitete Bayes'sche Optimierung wurde in Python mit dem Python-Paket *BoTorch* implementiert. Über zwei Testmodelle hinaus wurde das entwickelte Framework verwendet um die Spulenströme eines Stellarators mit *Simsopt* zu optimieren. Bis der Gaußprozess seine maximale Genauigkeit erreicht, sind die Ergebnisse der Bayes'schen Optimierung etwas besser sind als jene mit der L-BFGS-B-Methode. Zusätzlich wurde ein interaktives Tool für die explorative Daten- und Modellanalyse mit *plotly/dash* weiterentwickelt.

Acknowledgements

I would like to express my gratitude to Professor Christopher Albert for inspiring me to pursue this project and advising me along the way. As well as for the insightful discussions, his personal support and guidance. Special thanks to NAWI Graz for providing me the platform to conclude my studies and the EUROfusion Consortium for granting resources to this thesis. Finally, I am grateful to my family and friends, who have always supported me during my studies.

Michael Hadwiger, August 2024

This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

Contents

1 Introduction

On the road to a positive energy gain of fusion reactors and the enablement for later commercial use many challenges lie ahead. One of the major challenges in a stellarator type reactor is the optimization of the coil parameters and currents. The problem at hand is to optimize the output through systematically varying the model parameters, a so-called parameter study. The nature of this problem is not exclusive to plasma physics, but encountered in many fields of physics and engineering. There are different approaches to find the best set of parameters (the global optimum) including grid search, random sampling, Monte-Carlo sampling, and response surface modelling.

In the frame of this thesis the latter one is investigated in more detail. In order to achieve the goal Bayesian optimization is used. Through the means of Gaussian process regression a model to describe the relationship between the input and output dimensions is established. However, in literature, this approach is also known as response surface or response model.

The aim of this thesis is to utilize derivative observations for the optimization of the target. This information has an increased availability due to the rise of auto-differentiable functionalities in various software packages. Therefore, it was explored how the gradient information can be embedded into the model (the Gaussian process), how the performance changes and what the limitations of the method are. A second major part of parameter studies is the selection of the input parameter set for the next iteration. This was done using different acquisition functions, primarily the Expected Improvement (EI) and the Knowledge Gradient (KG).

The project was implemented in Python using the libraries *BoTorch* with *GPyTorch* and *SciPy* as a backend. In the process of this thesis, the developed algorithm was used to optimize the coil currents of a stellarator in order to generate a previously calculated magnetic field (a Stage-II optimization) using the Python package *Simsopt*.

2 Gaussian Progress Regression

The goal in various kind of parameter studies is to find an input-output mapping. One example is the combination of gasoline mixture and air intake for the optimization of the efficiency of a combustion engine. There are different approaches to get to this mapping, one is to make assumption about the underlying function. For example to model the problem with a summation of polynomial functions with different coefficients and solve the least-squares problem. This can be useful if the underlying structure or theory is known. On the other hand, if this is not the case, the assumption will restrict the potential outcome significantly if no information is available. The second approach is to allow all kinds of functions, but assign a prior probability and condition it on observations. To tackle this infinite number of possibilities one can use Gaussian Processes (GP).

2.1 Gaussian Process

> *A Gaussian process is a generalization of the Gaussian probability distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a stochastic process governs the properties of functions.* [30]

One can approach a Gaussian process from a weight space or a function space view. The following sections briefly explain both approaches, primarily based on the state-of-the-art book on Gaussian Processes by Carl Edward Rasmussen and Christopher K. I. Williams [30], with some additional insights from [16].

Weight space view

For the weight space view, a common linear regression problem (see Eq. 2.1.1) with a normal distributed error term ε can be started with. However, instead of the standard least-squares approach, a Bayesian approach will be used. The observed noisy output y of f evaluated at input x is

$$y = f(x) + \varepsilon, \tag{2.1.1}$$

$$\varepsilon \sim \mathcal{N}(0, \sigma_n^2). \tag{2.1.2}$$

For the linear problem stated in Eq. 2.1.1 a weight space approach can be made by describing the unknown function $f(x)$ with a multiplication of a weights vector w and input vector x,

$$y = \vec{x}^T \cdot \vec{w}, \tag{2.1.3}$$

$$\vec{x} = \begin{bmatrix} 1 \\ x_i \end{bmatrix}, \tag{2.1.4}$$

$$\vec{w} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}. \tag{2.1.5}$$

In a Bayesian framework one now assumes a prior distribution for the weights w (Eq. 2.1.6) as a Gaussian Prior with zero mean and a covariance matrix Σ according to

$$p(\mathbf{w}) = \mathcal{N}(0, \Sigma). \tag{2.1.6}$$

Starting from the model in combination with the noise assumption (Eq. 2.1.3) leads to the expression for the likelihood (probability density of the observations given the parameters) which is factored over all training points,

$$\begin{aligned} p(\mathbf{y} \mid X, \mathbf{w}) &= \prod_{i=1}^{n} p\left(y_i \mid x_i, \mathbf{w}\right) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\left(-\frac{\left(y_i - x_i^\top \mathbf{w}\right)^2}{2\sigma_n^2}\right) \\ &= \frac{1}{\left(2\pi\sigma_n^2\right)^{n/2}} \exp\left(-\frac{1}{2\sigma_n^2}\left|\mathbf{y} - X^\top \mathbf{w}\right|^2\right) \\ &= \mathcal{N}\left(X^\top \mathbf{w}, \sigma_n^2 I\right). \end{aligned} \tag{2.1.7}$$

In order to calculate the posterior distribution over the weights $p(\mathbf{w} \mid \mathbf{y}, X)$ the Bayes Theorem is used. As always in Bayes' Theorem the normalization is the marginalization over the weights,

$$p(\mathbf{w} \mid \mathbf{y}, X) = \frac{p(\mathbf{y} \mid X, \mathbf{w}) p(\mathbf{w})}{p(\mathbf{y} \mid X)}. \tag{2.1.8}$$

This expression can be rewritten by completing the square to achieve a more readable expression, where $A = \sigma_n^{-2} X X^T + \Sigma^{-1}$.

$$p(\mathbf{w} \mid y, X) \sim \mathcal{N}\left(\overline{\mathbf{w}} = \frac{1}{\sigma_n^2} A^{-1} X \mathbf{y}, A^{-1}\right) \tag{2.1.9}$$

If a prediction y_* is to be made at a test point x_*, the error term needs to be averaged out in order to get the prediction $f_* = y_* - \varepsilon_* = f(x_*)$. The predictive distribution is therefore the posterior probability weighted averaged over all possible parameter values

$$\begin{aligned} p\left(f_* \mid x_*, \mathbf{y}, X\right) &= \int p\left(f_* \mid x_*, \mathbf{w}\right) p(\mathbf{w} \mid y, X) d\mathbf{w} \\ &= \mathcal{N}\left(\frac{1}{\sigma_n^2} x_*^\top A^{-1} X y, x_*^\top A^{-1} x_*\right). \end{aligned} \tag{2.1.10}$$

If the base model is more complex than this simple linear model, the approach needs to be extended. One possible idea is to project or map the inputs into some high dimensional space and then do a linear regression in this space. This is done using a non-linear mapping function $\phi(X)$. The input vector/matrix X is then replaced by the mapping $\phi(X)$. The problem is that there are endless possibilities for this kind of mapping with little guidance what mapping to choose. Often there is a model comparison needed within a set of potential mappings.

Function space view

In the previous section a distribution over weights was used. Starting from each weight a specific function is implied which leads to a distribution over functions. This distribution over functions is called a Gaussian Process (GP) and is formally defined as follows:

> *A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.* [30]

A Gaussian Process is completely defined by its mean function ($m(x)$ or $\mu(x)$) and covariance function or kernel ($cov(x, x\prime)$ or $k(x, x\prime)$). For a process $f(x)$ one can specify

$$f(x) \sim \mathcal{GP}\left(\mu(x), k\left(x, x'\right)\right) \tag{2.1.11}$$

with the mean function $\mu(x)$ and the kernel $k\left(x, x'\right)$

$$\mu(x) = \mathbb{E}[f(x)], \tag{2.1.12}$$

$$k\left(x, x'\right) = \mathbb{E}\left[\left(f(x) - \mu(x)\right)\left(f\left(x'\right) - \mu\left(x'\right)\right)\right]. \tag{2.1.13}$$

The mean function is the average over all functions in the distribution. In reality the prior mean function is often set to be zero for practical reasons. This means that the inference is only done via the kernel (covariance function). The kernel models the correlation between different locations. There is a vast variety of useful kernels. The selection is done based on assumptions such as smoothness or correlations in the data. A very common assumption for the correlation between points is that closer points are more similar than distant points, in other words the correlation between the points decays with distance.

A very commonly used kernel is the radial basis function (RBF) or squared exponential kernel (see. Eq. 2.1.14). In this case the ℓ represents the length scale which specifies the smoothness of the function. If is defined as

$$k\left(x_p, x_q\right) = \exp\left(-\frac{|x_p - x_q|^2}{2\ell}\right). \tag{2.1.14}$$

There are a vast variety of different kernels which each have their advantages and disadvantages. A discussion on different kernels and the effect of lengthscales can be found at [30] and [16].

2.2 Prediction with Observations

Up to now, no observations or training data are incorporated into the GP. In a simple case, the situation of noise-free observations is considered. The joint distribution of of the training outputs $\mathbf{f}$ and a set of test outputs $\mathbf{f}_*$ according to the prior is shown in Eq. 2.2.1. In this notation the training data points or observations are the result of actual simulations or measurements whereas the test data points are evaluation of the model. Here, the kernel $K\left(X_*, X_*\right)$ denotes the $n \times n^*$ covariance matrix between training point matrix X and test point matrix X_*

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) & K\left(X, X_*\right) \\ K\left(X_*, X\right) & K\left(X_*, X_*\right) \end{bmatrix}\right). \tag{2.2.1}$$

To obtain the posterior distribution, all function of the prior that do not fit with the observations need to be rejected. The conditioned GP can be written as

$$\begin{aligned} \mathbf{f}_* \mid X_*, X, \mathbf{f} \sim \mathcal{N}(&K(X_*, X)K(X,X)^{-1}\mathbf{f}, \\ &K(X_*, X_*) - K(X_*, X)K(X,X)^{-1}K(X, X_*)). \end{aligned} \tag{2.2.2}$$

The upper part of Fig. 2.1 shows a prior distribution with zero mean. To illustrate this, a few example functions are drawn as dashed lines; however, an infinite number of such functions exists. In the bottom half of the figure, the prediction of the posterior mean, along with the confidence interval and some illustrative functions, is shown. The conditioning of the GP on the observations is clearly visible in the posterior. It is also possible to include both homoscedastic and heteroscedastic noise observations into the Gaussian Process. For further details, the reader is refereed to [30].

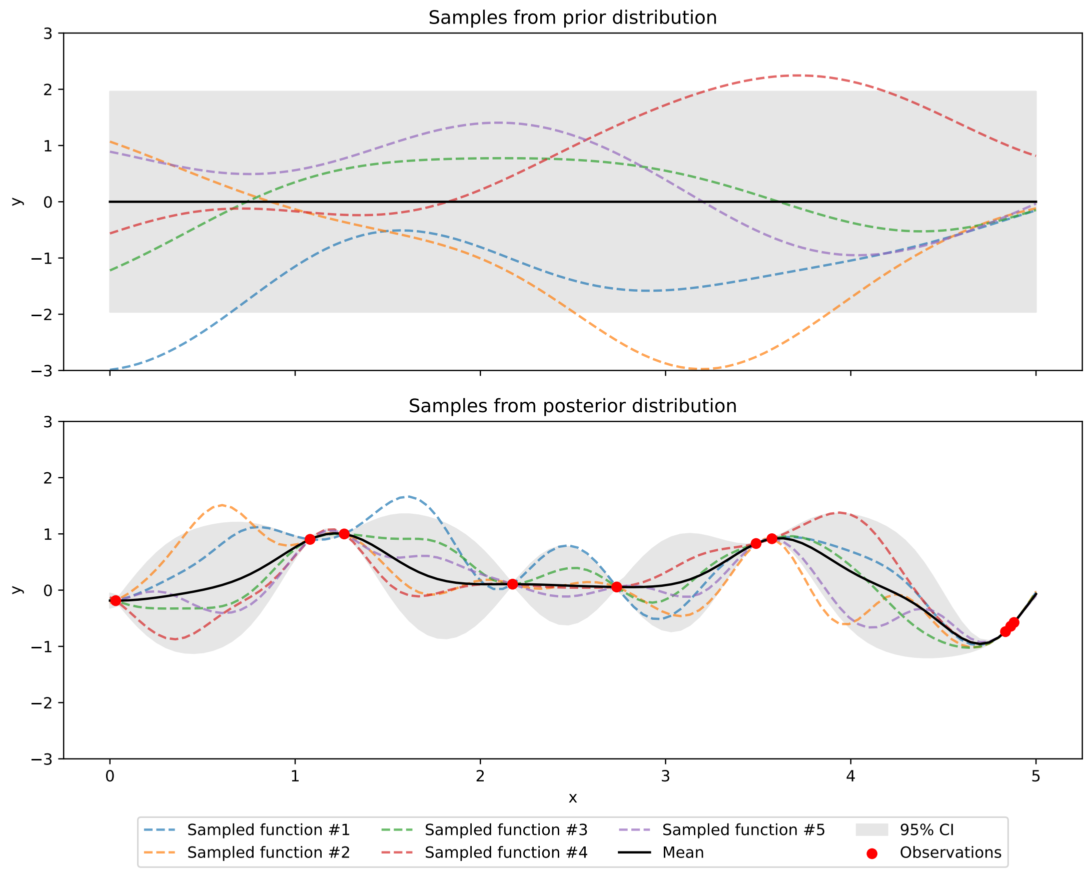


Figure 2.1: Conditioning of the Gaussian Processes with a RBF kernel on observations from the prior distribution with zero mean (top) to the posterior distribution (bottom). In both sub-figures, the mean and the corresponding confidence interval are shown, along with five of the infinitely many possible functions, which are depicted as dashed lines. Adapted from [26].

2.3 Gradient Information

The incorporation of derivative information into Gaussian Processes is discussed in [30] in Chapter 9.4. The main message is, that due to the linearity of differentiation the derivative of an GP is another GP. The properties of linearity are also treated in [2]. The inference is done by means of a joint Gaussian distribution of function values and their partial derivatives. The covariance between these two can be formulated as shown in

Eq. 2.3.1 and 2.3.2.

$$\operatorname{cov}\left(f_i, \frac{\partial f_j}{\partial x_{dj}}\right) = \frac{\partial k(x_i, x_j)}{\partial x_{dj}} \tag{2.3.1}$$

$$\operatorname{cov}\left(\frac{\partial f_i}{\partial x_{di}}, \frac{\partial f_j}{\partial x_{ej}}\right) = \frac{\partial^2 k(x_i, x_j)}{\partial x_{di} \partial x_{ej}} \tag{2.3.2}$$

With n observations in D dimensions this leads to $n \times (D+1)$ values. Not in all applications, however, all partial derivatives are used either due to insufficient availability or to save computational resources. In that case the corresponding rows and columns of the matrix are removed.

In addition to the presented approach by [30] a more detailed discussion can be found in [34]. Moreover, in [16] an approach to derivatives via differentiation is described. The effect of the incorporation of gradient information is discussed in Sec. 6.

3 Bayesian Optimization

The following chapter is largely based on the review paper by Peter I. Frazier [14].

Bayesian Optimization, also known as BayesOpt, BayOpt, or just BO rallies around the question of maximizing (or minimizing) an objective function. The problem can be formulated as

$$\max_{x \in A} f(x) \tag{3.0.1}$$

where A is the parameter space. In most cases the objective function has some common properties. One is that the objective is "expensive to evaluate", which implies that the optimization should be conducted with the minimal number of function evaluations. In addition, the objective is an so-called "black box" function, which means that no structure like linearity or convexity can be leveraged. Typically the input space $A \subset \mathbb{R}^d$ is limited to a few dimensions (typically $d < 20$) and the output is continuous. In the end it is the goal to find the global optimum of the objective not just a local one.

3.1 Core Idea

For the workflow of a BO two main components are needed, a Bayesian statistical model (e.g. a Gaussian Process) to model the objective function and an acquisition function to decide at what input combination to sample next.

The basic workflow is depicted in Fig. 3.1. At the beginning a Gaussian Process is initialized with a prior (typically a zero mean) on the objective function f. As a second step the observation of f on N_0 initial locations needs to be conducted. This can either be a simulation run or a measurement. This observation locations should be chosen space filling (e.g. by means of a Halton sequence or a Sobol sampling) in the parameter space. As a third step of the initialisation the model should be updated (conditioned) with all the available information. This includes the optimization of the lengthscales of the model and calculating the posterior probability distribution. It is worth noting, that the first two steps can be interchanged.

The initialization sequence is followed by the optimization loop. This loop consists of the following tasks: First an acquisition function (see Sec. 3.2) based on the current posterior probability distribution is initialized. Secondly, the maximum x_n of the acquisition

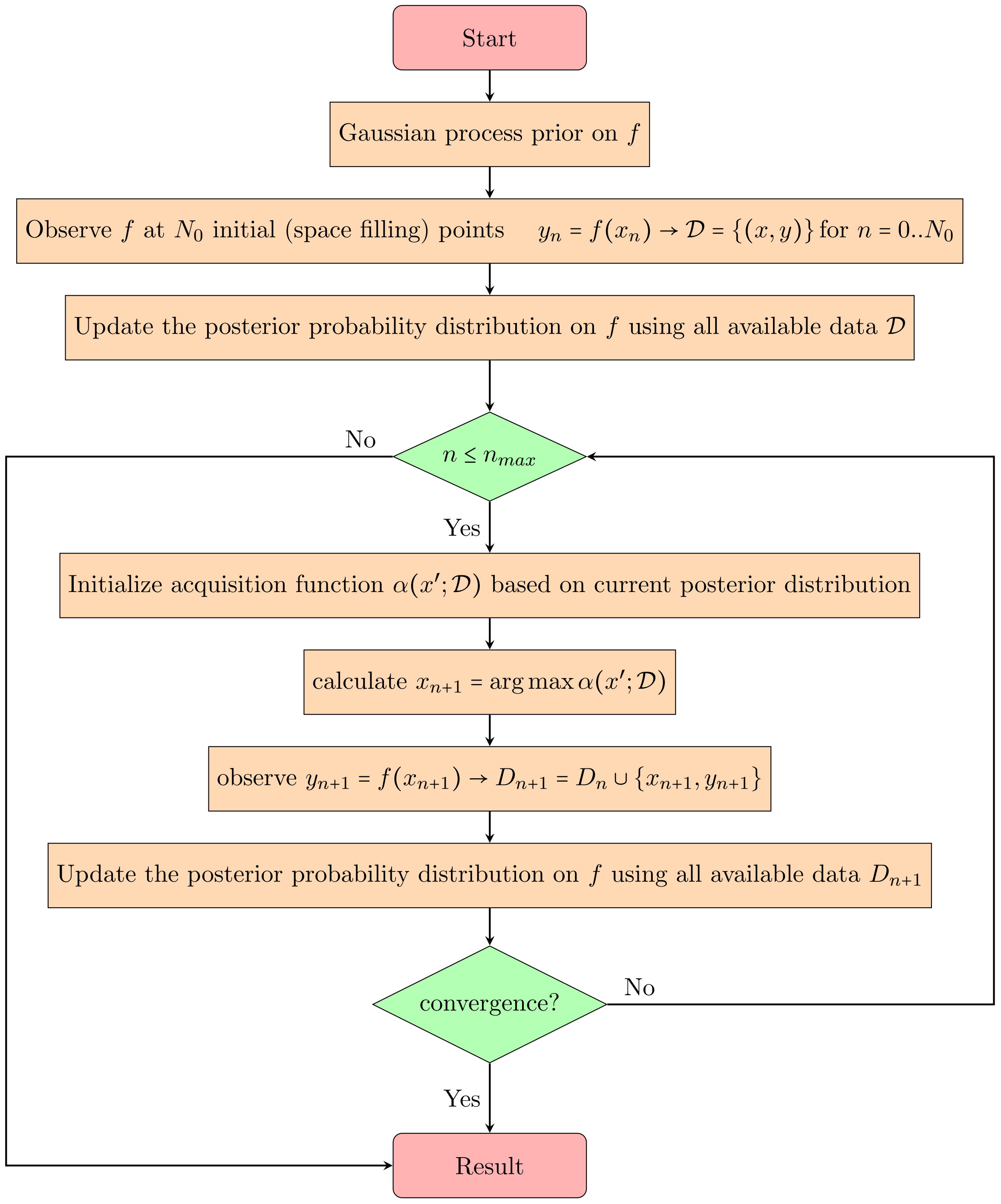

Start
Gaussian process prior on f
Observe f at N_0 initial (space filling) points $y_n = f(x_n) \rightarrow \mathcal{D} = \{(x, y)\}$ for $n = 0..N_0$
Update the posterior probability distribution on f using all available data $\mathcal{D}$
$n \leq n_{max}$
No
Yes
Initialize acquisition function $\alpha(x'; \mathcal{D})$ based on current posterior distribution
calculate $x_{n+1} = \arg\max \alpha(x'; \mathcal{D})$
observe $y_{n+1} = f(x_{n+1}) \rightarrow D_{n+1} = D_n \cup \{x_{n+1}, y_{n+1}\}$
Update the posterior probability distribution on f using all available data D_{n+1}
convergence?
No
Yes
Result

Figure 3.1: Basic workflow of a Bayesian Optimization.

function over x using some sort of optimizer (e.g. gradient descent mixed with a random multi-start theme) needs to be observed. Then observe $y_n = f(x_n)$ at the previously calculated location (this involves an actual measurement or a simulation run). As a last step of the optimization loop, the model is conditioned on f using all available data including the new observation. This is methodically the same process as in the last step of the initialization.

At the end of each optimization loop there are two possible termination conditions. The first is that some kind of convergence criterion is fulfilled. For this a wide variety of choices are available. The other is that a maximum number of iterations is reached. In either case, the optimization loop ends and a result is obtained.

This result of the optimization is either the point x^* where the largest value f^* was observed or the location of the maximum posterior mean μ^*, depending on the risk taken. In the latter case, the actual value of the input parameters can deviate from the maximum posterior mean, since the value was not actually observed and the model is always accompanied with some quantified uncertainty.

Simple Example

A simple toy model optimization is shown in Fig. 3.2. For a toy model no real simulation or measurement was conducted, instead a testfunction $f(x) = \sin(10x) + \cos(5x) + 0.5x$ was used as ground-truth. At the beginning the initial steps of the flow chart (Fig. 3.1) were conducted. For the initial training points the testfunction was evaluated (observed) at three different locations. The GP with a zero mean prior was then conditioned with initial observations. In the top left sub-figure this initial situation is shown. It is visible, that the posterior mean is no longer zero but has its indicated maximum near the leftmost observation. In addition, the 95% confidence interval and the testfunction is shown. The sometimes casually called "sausage like" shape of the confidence interval discussed in the previous chapter is also clearly visible.

In the bottom left sub-figure the acquisition function $\mathrm{EI}(x)$ (in this case Expected Improvement) is shown over the domain of x (see Sec. 3.2.3). The maximum of the acquisition function $x_n = \arg\max_{x\in A} \mathrm{EI}(x)$ is indicated in both sub-figures with the vertical black dashed line. In the following the testfunction was observed at the new location $y_n = f(x_n)$. The new observation is included in the training dataset and the posterior is conditioned on the training dataset. With this new point added, the posterior mean is now resembling the testfunction already a lot better and the uncertainty interval is also reduced. This optimization loop is now done 6 times. It is visible that the Expected Improvement decreases to nearly zero after an observation is conducted (e.g. $\mathrm{EI}(x = 2.2)$ in the last sub-figure after sampling there in the previous iteration).

The effect of different acquisition functions is discussed in more detail in Sec. 3.2 and 6.

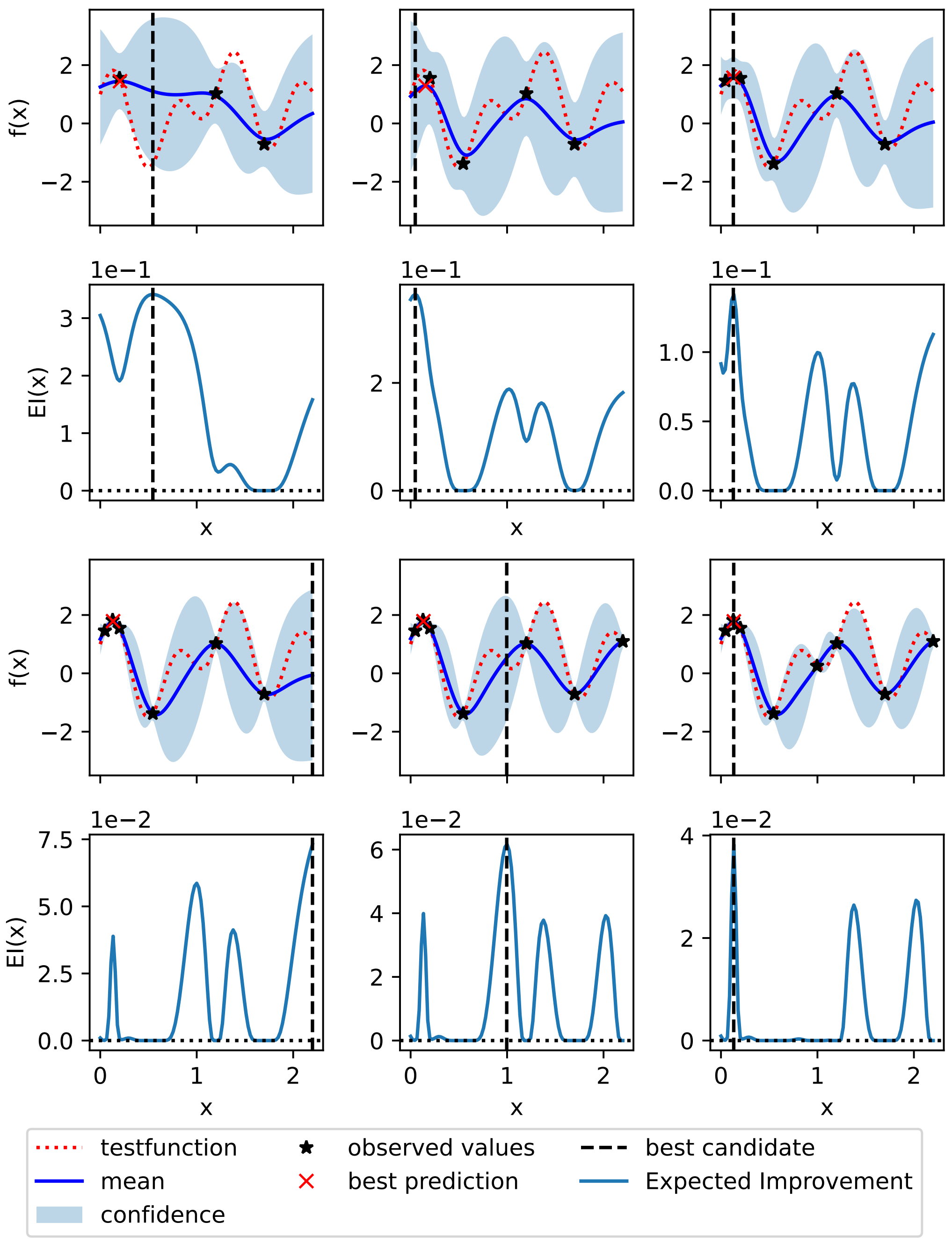


Figure 3.2: BO algorithm with $N_0 = 3$ initial observations and Expected Improvement (EI) as acquisition function.

3.2 Acquisition Functions

The following chapter on acquisition functions is mostly based on [16] accompanied by some interpretations by [14].

One important question in Bayesian Optimization is how to select the input combination for the next iteration. The idea is to have an easy-to-evaluate function that quantifies the information gained by a new observation based on the currently available information. Once such an acquisition function is established the determination of the maximum of this function can be done with state-of-the-art algorithms like BFGS or multi-start gradient ascent. In this way the easy-to-evaluate (”cheap”) acquisition function is evaluated multiple times instead of the expensive-to-evaluate (”expensive”) original physical simulation.

For the acquisition function several different approaches could be used. These can be classified into two main groups, a one-step look-ahead approach for some utility function or adaptations of multi-armed bandits algorithms for optimization. In the following the focus lies on the one-step look-ahead functions Expected Improvement (EI) and Knowledge Gradient (KG). Other examples for acquisition functions would be the probability of improvement, mutual information, or upper confidence bound.

3.2.1 Exploration vs. Exploitation

One of the main questions selecting an acquisition scheme is always the balance between exploration and exploitation. In the following, the different aspects will be described based on the example shown in Fig. 3.2.

Exploitation

Exploitation describes the sampling in proximity to an already known (good) location. The goal is to find an even better location near the current local maximum. This approach may lead to a larger immediate reward (improvement). One example is the new sampling location found in the second and third iteration of the optimization in Fig. 3.2. In both cases the new point is located in the proximity of the current maximum of the posterior mean.

Exploration

On the other hand, exploration in this context refers to sampling in yet not so well known locations. This scheme will not lead to a high immediate reward but may provide valuable information for the following iterations. Ideally, the exploration finds a new local maximum which was not known before. In the example show in Fig. 3.2 the locations in the first and fourth step could be seen as exploration. In both cases the new point is located in a region of high uncertainty where within the uncertainty a new local maximum could arise. A new observation in a region of low observation density reduces the uncertainty in the region significantly (depending on the model lengthscale).

3.2.2 One-Step Look-Ahead

Expected Improvement (EI), Knowledge Gradient (KG) and other one-step look-ahead acquisition functions all have a similar structure. The acquisition function is the expected marginal gain in utility. It is built upon an utility function u that acts as some kind of metric on a given dataset D. The acquisition value

$$\alpha(x;\mathcal{D}) = \mathbb{E}\left[u(\mathcal{D}') \mid x, \mathcal{D}\right] - u(\mathcal{D}) \tag{3.2.1}$$

is the difference between expectation value of the utility function based on the updated dataset $\mathcal{D}'$ and the utility function of the current data set $\mathcal{D}$. The updated dataset includes the presumed new observation y' on the proposed location x' in the form $\mathcal{D}' = (\mathbf{X}', \mathbf{Y}') = \mathcal{D} \cup \{(x', y')\}$.

The next sampling location x is calculated from the acquisition function as the location of the maximum according to Eq. 3.2.2. In this context A is the search domain (parameter space) for the new location.

$$x = \arg\max_{x' \in A} \alpha(x';\mathcal{D}) \tag{3.2.2}$$

3.2.3 Expected Improvement (EI)

Expected Improvement is a very common acquisition scheme or as [16] puts it:

> *Sequential maximization of Expected Improvement is perhaps the most widespread policy in all of Bayesian optimization.*

The key ingredient for Expected Improvement is to use a one-step look-ahead approach (see Sec. 3.2.2) combined with the simple reward utility function (Eq. 3.2.3). The simple reward utility function returns the maximal observed value in the data set $\mathcal{D}$ and is defined as

$$u(\mathcal{D}) = \max \mu_{\mathcal{D}}(x). \tag{3.2.3}$$

The idea of Expected Improvement is to maximize the expected marginal gain in utility $u(\mathcal{D}') - u(\mathcal{D})$. This is also called instantaneous improvement and is realised through sampling the next observation at x. Starting from the one-step look-ahead acquisition function in Eq. 3.2.1 and the simple reward utility function combined with the marginalisation rule the Expected Improvement acquisition function reads

$$\alpha_{\mathrm{EI}}(x;\mathcal{D}) = \int [\max \mu_{\mathcal{D}'}(x')]\ p(y \mid x, \mathcal{D}) dy - \max \mu_{\mathcal{D}}(x) = \mathbb{E}\left[\left[f(x') - f^*\right]^+\right], \quad (3.2.4)$$

where $a^+ = \max(a, 0)$. To get the location for the next iteration the maximum of this acquisition function must be calculated according to Eq. 3.2.2.

The Expected Improvement scheme tends to sample near "good" locations as the simple reward favours immediate improvement which is an exploitation behaviour. However, after a number of observations close to each other, the acquisition value collapses near this location and an exploration behaviour is immanent. This for example can be seen in Fig. 3.2 where the acquisition value near the three observations slightly above $x = 0$ drop compared to the upper end after the third iteration. Expected Improvement balances between exploration and exploitation, but leans a bit towards exploitation. One possible solution is the Log Expected Improvement.

This explanation is based on reference [16]. For a slightly different approach that includes hints for the numerical implementation, see reference [14].

3.2.4 Knowledge Gradient (KG)

The Knowledge Gradient (KG) was first proposed by Frazier and Powell [12]. The idea is similar to Expected Improvement (see Sec. 3.2.3), to create a one-step look-ahead acquisition function. However, in this case the global reward utility function

$$u(\mathcal{D}) = \max_{x \in A} \mu_{\mathcal{D}}(x) = \mu^*_{\mathcal{D}} \quad (3.2.5)$$

is used instead of simple reward utility function (see Eq. 3.2.3). Here $\mu_{\mathcal{D}}(x)$ is the posterior mean and $\mu^*_{\mathcal{D}}$ is the maximum posterior mean.

In Expected Improvement the assumption was that one only wants to return a previously evaluated point as final result. This is reasonable for a noise free-setting when one is highly risk adverse. However, if some risk is tolerated and the observations are not noise-free it is reasonable to return a not yet evaluated point with some attached uncertainty. Nevertheless, risk neutrality is still present and leads to the following acquisition function for the acquisition function:

$$\alpha_{\mathrm{KG}}(x;\mathcal{D}) = \int \left[\max_{x' \in A} \mu_{\mathcal{D}'}(x')\right]\ p(y \mid x, \mathcal{D}) dy - \max_{x' \in A} \mu_{\mathcal{D}}(x') = \mathbb{E}\left[\mu^*_{\mathcal{D}'} - \mu^*_{\mathcal{D}}\right]. \quad (3.2.6)$$

Since the global reward honors the increase of the maximum of the posterior mean somewhere in the domain it is not important that the posterior mean at the new observation is better, but the overall maximum of the posterior mean increases. However, due to the need of calculating the new maximum posterior mean the method is much more computationally expensive. Possible solutions to mitigate this involve Monte Carlo approaches (see Sec. 4.1.3).

One possible interpretation is, that the global reward utility function is in some sense the "knowledge" of the global maximum of the posterior mean offered by the available data $\mathcal{D}$. The acquisition function is consequently the expected change in the knowledge due to the addition of an observation at x'.

The Knowledge Gradient scheme often suggests a new point near a yet best seen point. Near this location there will likely be a local optimum, but the actual location is yet unknown. A new point in the proximity, regardless if it is better or worse than the current best seen, will add valuable information about the location of the optimum. If one somehow learns the "derivative" at the point by finite difference, one can point to which side the optimum is. That the point is potentially worse than the previously sampled, would be a problem in a simple reward scheme. However, this is not the case in a global reward scheme due to the importance of the change in the maximum posterior mean over the whole domain. The consideration about the derivative at the location leads to the idea of incorporating gradient information in the first place if available (see Sec. 2.3 and Sec. 3.2.5).

As shown in the examples in Sec. 6 KG tends to do more exploration than EI and finds the global optimum more efficient. This is due to the fact, that in EI a high reward is only granted if it guessed the right side of a good point for a local optimum. KG is only interested in the increase of maximum of the posterior mean and the sampling on both sides is a success.

In the review paper of Frazier [14] also some pseudo-code implementations are listed.

3.2.5 Derivative-Enabled Knowledge Gradient (dKG)

For the Knowledge Gradient acquisition function (see Sec. 3.2.4) a derivative-enabled version was proposed in [34]. The algorithm is based on the Knowledge Gradient approach in [13] using batch evaluation [32]. For details about the batch evaluation the reader is referred to the mentioned literature.

The formulation of the derivative-enabled Knowledge Gradient function (dKG) can be found in [34]. There however, the goal is to find a global minimum while our goal is to find a global maximum. In an essence, the paper states that the difference between the the derivative-enabled Knowledge Gradient function (dKG) and a batch KG proposed in [32] is that the posterior mean now also depends on the derivative observations due to the incorporation of the gradient information into the GP (see Sec. 2.3). Furthermore, this requires the calculation and marginalization of the distribution of these gradient observations for the posterior at each step.

> *Thus, the d-KG algorithm differs from KG not just in that gradient observations change the posterior, but also in that the prospect of future gradient observations changes the acquisition function.* [34]

A comparison between EI and KG with their derivative-enabled counterparts can be found in Sec. 6.

Due to the bad scaling of the GP inference in higher dimensions $\mathcal{O}(n^3(d+1)^3)$, it can be useful to not include all partial derivatives to reduce the computational cost. In [34], an approach with only one directional derivative from each iteration is proposed. However, this approach is not realised in the scope of this thesis.

The efficient computation of the dKG is quite difficult to implement. A discretization free approach with stochastic gradient ascent is proposed in [34]. In the following, the Python library *BoTorch* is used, which has an Monte Carlo implementation of the KG (see Sec. 4.1.3 and [3]).

4 Implementation

Several software packages are available for the implementation of Bayesian optimization. The most popular libraries include *Cornell-MOE*[1] ([32] and [34]), *Dragonfly*[2] ([24]), *Spearmint*[3] and *BoTorch*[4] ([3]) to list a few. A more extensive list and a discussion can be found in [3].

Cornell-MOE is a proof-of-concept code written in `C++` by the authors of [34]. However, the adaption of it seems to be quite challenging. On the other hand *BoTorch* is an actively maintained package written in Python. Therefore *BoTorch* was used for this thesis. In the following the approach of *BoTorch* will be described.

4.1 BoTorch

BoTorch is an Python package for Bayesian optimization build upon *PyTorch* [5] and *GPyTorch*[6] ([15]) and mainly developed by META. As described in the paper [3], the package combines Monte-Carlo (MC) acquisition functions with a novel sample average approximation optimization approach, auto-differentiation, and variance reduction techniques. For this work especially the novel "one-shot" formulation of the Knowledge Gradient is used.

In the following the idea of *BoTorch* is sketched in a simplified way. This is a summary of Chapter 3 and 4 of [3].

[1]see https://github.com/wujian16/Cornell-MOE
[2]see https://github.com/dragonfly/dragonfly
[3]see https://github.com/HIPS/Spearmint
[4]see https://github.com/pytorch/botorch
[5]see https://pytorch.org/
[6]see https://gpytorch.ai/

4.1.1 Monte-Carlo Acquisition Function

The first part of the one-step look-ahead acquisition functions discussed in Sec. 3.2 can generally be written as

$$\alpha(x; \Phi, \mathcal{D}) = \mathbb{E}[a(g(f(x)), \Phi) \mid \mathcal{D}] \tag{4.1.1}$$

where g is an objective function, a is an utility function and Φ are parameters independent of x (see Eq. 3.2.1). In some cases this can be solved analytically (e.g. for Expected Improvement). However, in general this is not possible. Therefore *BoTorch* uses Monte Carlo (MC) integration to approximate the expectation using samples from the posterior. The MC approximation $\alpha_N(x; \phi, \mathcal{D})$ using N samples result to

$$\alpha_N(x; \Phi, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} a(g(\xi_{\mathcal{D}}^i(x)), \Phi) \tag{4.1.2}$$

where $\xi_{\mathcal{D}}^i(x) \sim f_{\mathcal{D}}(x)$ and is drawn i.i.d. (independent and identically distributed) from the model.

4.1.2 Sample Average Approximation (SAA)

According to the paper [3], the new methodological approach of *BoTorch* is to use the Sample Average Approximation (SAA) within Bayesian Optimization. Instead of re-drawing samples for each optimization of the MC acquisition function, a fixed set of locations is used for the evaluation of the acquisition function. Once drawn, the base sample set is fixed and used for the whole optimization of the acquisition function (e.g. Stochastic Gradient Descent (SGD) or other first order methods). The authors state that this can be seen as a specific incarnation of the method of common random numbers. The candidate set x' can be obtained by

$$x' = \arg\max_{x \in A} \alpha_N(x; \Phi, \mathcal{D}). \tag{4.1.3}$$

The gradient $\nabla_x \hat{\alpha}_N(x; \boldsymbol{\Phi}, \mathcal{D})$ can be computed through auto-differentiation as the average of the sample-level gradients.

Due to the use of fixed base samples, the authors conclude, that the SAA approach benefits from the applicability of the various methods of deterministic optimization. This includes the possibility to use quasi-Newton methods, which converge faster and are less sensitive than stochastic first-order methods. *BoTorch* uses a multi-start optimization via L-BFGS-B as a default (for details see [3] Appendix F.1). The bias introduced by the SAA only has a minor effect on the performance compared to the analytic ground truth (if the sample size is large enough) and is even often better than stochastic approaches.

For further details regarding the convergence properties the interested reader is refereed to the original paper [3] (Sec. 4.1 and Appendix D).

4.1.3 One-Shot Formulation of the Knowledge Gradient using SAA

As described in Sec. 3.2.4, the Knowledge gradient (KG) acquisition function quantifies the expected increase of the maximum of the posterior mean $\mu^*_{\mathcal{D}}$ due to adding an additional observation. Compared to the Expected Improvement (EI) acquisition function (see Sec. 3.2.3), it yields better performance, however, it is also computationally more expensive due to the nested optimization problem.

The acquisition function (Eq. 3.2.6) can be rewritten as

$$\alpha_{\mathrm{KG}}(x;\mathcal{D}) = \mathbb{E}\left[\mu^*_{\mathcal{D}'} \mid \mathcal{D}\right] - \mu^*_{\mathcal{D}}, \tag{4.1.4}$$

where $\mu^*_{\mathcal{D}} = \max_{x' \in A} \mathbb{E}\left[g(f(x')) \mid \mathcal{D}\right]$ and $\mu^*_{\mathcal{D}'} = \max_{x' \in A} \mathbb{E}\left[g(f(x')) \mid \mathcal{D}'\right]$ with $\mathcal{D}' := \mathcal{D} \cup \{x', y_{\mathcal{D}}(x')\}$.

This maximization of $\alpha_{\mathrm{KG}}(x;\mathcal{D})$ is now a nested optimization problem. This involves finding the maximum over all x and in each evaluation finding the maximum posterior mean. For details the reader is refereed to [14]. Depending on the implementation, some kind of nested loop (see Algorithm 3 in [14]) is needed in order to solve it.

However, in [3] the nested problem is treated via the SAA approach, which makes it a deterministic problem. Using the reparameterization trick and using fixed samples as described in Sec. 4.1.2 the resulting MC approximation of KG is

$$\alpha_{\mathrm{KG},N}(x;\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \max_{x_i \in A} \mathbb{E}\left[g(f(x_i)) \mid \mathcal{D}^i_x\right] - \mu^*_{\mathcal{D}}. \tag{4.1.5}$$

According to the authors, due to the fixed base samples when calculating the maximum of Eq. 4.1.5, the sum and the max can be exchanged, which leads to the following expression

$$\max_{x \in A} \alpha_{\mathrm{KG},N}(x,\mathcal{D}) \equiv \max_{x,x'} \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left[g(f(x_i)) \mid \mathcal{D}^i_x\right], \tag{4.1.6}$$

where x' are the "next stage" or "fantasy" points. Here the problem is solved jointly over x and x' (fantasy points) instead of solving the inner problem to completion for every fantasy point for each gradient step w.r.t. x. This leads to an increased dimension of the problem of $(q+N) \times d$ instead of $q \times d$, however methods for deterministic optimization can be used. The authors also note that the problem increases linearly with N, but observe good performance for moderate N.

4.2 Python Implementation

Following the theory discussed in the previous sections, the actual implementation of the Bayesian Optimization using *BoTorch* will be discussed. The Python package developed in the course of the thesis is named *bayoptder* and available at https://github.com/itpplasma/bayoptder. It provides a command line interface (CLI) (see Sec. 4.2.3).

In the following, the implementation of the core workflow depicted in Fig. 3.1 will be discussed. First, the noise free case, as shown in the tutorials of *BoTorch* and subsequently the inclusion of the derivative observations to the model. Afterwards the package usage is described.

4.2.1 BO Implementation in BoTorch

Throughout the workflow of Fig. 3.1, from an programming point of view four main ingredients are needed. These are the following:

- initialization of the GP
- update the posterior probability of the GP with new observations
- initialization of the acquisition function
- selecting the next iteration location through maximizing the acquisition function

In the following these parts will briefly be described based on the tutorial of *BoTorch* [11].

GP Initialization

At the beginning, one needs to initialise a GP model. *BoTorch* provides a wide range of models and also supports the *GPyTorch* model class. An overview over all the models can be found at [10]. The built-in *BoTorch* models can be classified into Single-Task GPs (homo- and heteroscedastic noise, mixed discrete and continuous features and full Bayesian with SAAS prior), Multi-Task GPs (homoscedastic noise, Kronecker structure and full Bayesian with SAAS prior) and Model List of Single-Task GPs. In addition, it supports custom models which will be discussed in Sec. 4.2.2.

In Listing. 4.1 a `SingleTaskGP` is initialised. The code consists of three parts. First, a set of training data is generated based on the testfunction $y(x) = \sin(2\pi x_1) \cdot \cos(2\pi x_2)$ for $x \in [0,1]^2$. In line 16 an stochastic error with $\sigma^2 = 0.05$ is added to the observations and then the noisy observations are standardized. In the second section the model is initialised as `SingleTaskGP` from the standard *BoTorch* models with the training data $\mathcal{D} = \{x_n, y_n\}$ with n data points. In the last section the lengthscales of the model are optimized through the build in optimization function `fit_gpytorch_mll` which relies on the `scipy.optimize` library.

```python
import os
import math
import torch

from botorch.fit import fit_gpytorch_mll
from botorch.models import SingleTaskGP
from botorch.utils import standardize
from gpytorch.mlls import ExactMarginalLogLikelihood

# Generate Training Data
bounds = torch.stack([torch.zeros(2), torch.ones(2)])

train_X = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(20, 2)
train_Y = torch.sin(2 * math.pi * train_X[:, [0]]) * torch.cos(2 *
    math.pi * train_X[:, [1]])

train_Y = standardize(train_Y + 0.05 * torch.randn_like(train_Y))

# Initialise Model
model = SingleTaskGP(train_X, train_Y)

# Optimize the lengthscales of the model
mll = ExactMarginalLogLikelihood(model.likelihood, model)
fit_gpytorch_mll(mll);
```

Listing 4.1: Example of a model initialization (adapted from [11]).

Update Posterior

After a new observation is conducted, this observation needs to be incorporated into the model and the lengthscales of the model need to be updated. In general, there are two different methods to do this. One is to initialize a new model with the new extended training data set $\mathcal{D}_{n+1} = \mathcal{D}_n \cup \{x_{n+1}, y_{n+1}\}$ as shown in Listing 4.2.

```python
# Extend Training Data with new observation
train_X = [train_X, new_X]
train_Y = standardize([train_Y, new_Y])

# Initialize model
model = SingleTaskGP(train_X, train_Y)

# Optimize the lengthscales of the model
mll = ExactMarginalLogLikelihood(model.likelihood, model)
fit_gpytorch_mll(mll);
```

Listing 4.2: Example of including a new observation via initialization of a new model and optimizing the lengthscales.

The other method is to use the build-in method `condition_on_observation`, as shown in Listing 4.3. This method should be used, if the model supports fantasizing (e.g. needed for MC acquisition functions such as the Knowledge Gradient acquisition function).

```python
# Condition model on new observation
model = model.condition_on_observations(new_x, new_Y)

# Optimize the lengthscales of the model
mll = ExactMarginalLogLikelihood(model.likelihood, model)
fit_gpytorch_mll(mll);
```

Listing 4.3: Example of including a new observation via `condition_on_observation` and optimizing the lengthscales.

The conditioning has the benefit that no new model with default lengthscales is initialised, but just the new data is added to the model. In both cases it is needed to optimize the lengthscales again. This is done in the same way as in Listing 4.1 line 21 to 23.

The problem concerning the default lengthscales after a new initialization can be solved through starting the optimization with the previous values. Therefore the "old" lengthscales are saved and then loaded to the model before starting the lengthscale optimization.

Acquisition Function Initialization

In Sec. 3.2 different acquisition function were discussed. Both the Expected Improvement (EI) and Knowledge Gradient (KG) are implemented in *BoTorch*. The initialization can be done according to Listing 4.4 for both the EI and KG acquisition function.

```python
from botorch.acquisition import ExpectedImprovement,
    qKnowledgeGradient

# Initialization of the Expected Improvement (EI) function
EI = ExpectedImprovement(model, best_f=best_f)

# Initialization of the Knowledge Gradient (KG) function
qKG = qKnowledgeGradient(model, num_fantasies=NUM_FANTASIES)
```

Listing 4.4: Example of initialising the EI and KG acquisition function for a given model.

In this case `best_f` is the current posterior mean maximum. The constant `NUM_FANTASIES` describes the number of fantasy points during the MC process. An increased number of fantasy points leads to a more accurate approximation of KG. However, this leads to the need of both more memory as well as wall time. The q in the qKG originates from the possibility to calculate a batch of size q points at once. The idea is to calculate q new observations parallel before updating the model again (see [33]). In this case q defaults to 1.

Acquisition Function Maximum

After the initialization of the acquisition function the maximum needs to be determined. This optimization is usually done with the `optimze_acqf` function. This is a wrapper for the `scipy.optimize.minimize` function, but first some checks and rescaling are done if needed.

```python
from botorch.optim import optimize_acqf

candidates, acq_value = optimize_acqf(
    acq_function=qKG,
    bounds=bounds,
    q=1,
    num_restarts=NUM_RESTARTS,
    raw_samples=RAW_SAMPLES,
)
```

Listing 4.5: Example of optimizing an acquisition function using the `optimze_acqf` function of *BoTorch* within specified bounds and a batch size of $q = 1$.

As shown in Listing 4.5, the boundaries of the optimization as well as the number of restarts `NUM_RESTARTS` and raw samples `RAW_SAMPLES` need to be provided. In most cases the boundaries for the optimization are the ones of the parameter space.

The number of raw samples is used in generating the initial conditions for the optimization in *BoTorch* internally. A higher number of raw samples yields better initial conditions. Starting for the initial conditions the optimization is done `NUM_RESTARTS` times. In the end an increase of both constants leads to better results. [4]

4.2.2 Inclusion of Gradient Observation

Based on the implementation seen in the previous subsection the question arises, how to incorporate the gradient observations into the system. The goal is to construct a model (a GP) that includes the derivative observation in a way discussed in Sec. 2.3 and

[30] (Chapter 9.4). On top of the custom derivative enabled GP model the standard acquisition function provided by *BoTorch* is used, as proposed by M. Balandat in [6]. However, *BoTorch* and specially the *SciPy* backbone converge quite badly if the inputs are not (nearly) normalized and the outputs are not standardized. Therefore, a scaling scheme was developed (see Sec. 4.2.2).

Custom derivative enabled GP

For the custom derivative enabled GP, the idea is to have inputs `train_X` with dimensions $(n \times d)$ and `train_Y` with dimensions $(n \times (1+d))$, where n is the number of samples/observations and d is the is the number of dimensions of the problem. The first columns of `train_Y` are the function values, the columns 2 to $d+1$ are the derivatives of the function with respect to the input dimensions. The GP therefore needs a multitask Gaussian likelihood with $d+1$ dimensions, as well as a kernel that supports $d+1$ dimensions.

Based on the discussion in the GitHub Issue 636 of *BoTorch* [6], the best way is to start with the *GPyTorch* tutorials on regression with derivative information in 1d and 2d ([18] and [19]). In the tutorial for the 1d case [18] the following GP, as shown in Listing 4.6, was proposed.

```
class GPModelWithDerivatives(gpytorch.models.ExactGP):
     def __init__(self, train_x, train_y, likelihood):
         super(GPModelWithDerivatives, self).__init__(train_x, train_y, likelihood)
         self.mean_module = gpytorch.means.ConstantMeanGrad()
         self.base_kernel = gpytorch.kernels.RBFKernelGrad()
         self.covar_module = gpytorch.kernels.ScaleKernel(self.base_kernel)

     def forward(self, x):
         mean_x = self.mean_module(x)
         covar_x = self.covar_module(x)
         return gpytorch.distributions.MultitaskMultivariateNormal(mean_x, covar_x)

likelihood = gpytorch.likelihoods.MultitaskGaussianLikelihood(num_tasks=2)  # Value + Derivative
model = GPModelWithDerivatives(train_x, train_y, likelihood)
```

Listing 4.6: GP model proposed in the *GPyTorch* tutorial [18] to incorporate derivative observations in the model for a 1d case.

Based on that, some adaptions where made as shown in Listing 4.7. First, the likelihood was included into the `__init__` function as attribute `self.likelihood`. Secondly, a new variable `dim` was introduced defined as `train_X.shape[-1]`. The dimensionality of the likelihood is now dynamically controlled via the `dim` variable. In addition, in the setting of the base kernel a individual lengthscale for each dimension is set via the flag `ard_num_dims=dim`.

```python
class GPWithDerivatives(GPyTorchModel, ExactGP, FantasizeMixin):
    def __init__(
        self,
        train_X,
        train_Y,
    ):
        # Dimension of model
        dim = train_X.shape[-1]
        # 1+d dimensional likelihood
        likelihood = MultitaskGaussianLikelihood(num_tasks=1 + dim)
        super().__init__(train_X, train_Y, likelihood)
        # Gradient-enabled mean
        self.mean_module = gpytorch.means.ConstantMeanGrad()
        # Gradient-enabled kernel
        self.base_kernel = gpytorch.kernels.RBFKernelGrad(
            ard_num_dims=dim,  # Separate lengthscale for each input dimension
        )

        self.covar_module = gpytorch.kernels.ScaleKernel(self.base_kernel)
        # Output dimension is 1 (function value) + dim (number of derivatives)
        self._num_outputs = train_Y.shape[0]  # 1 + dim
        # Used to extract function value and not gradients during optimization
        self.scale_tensor = torch.tensor([1.0] + [0.0] * dim, dtype=torch.double)

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return MultitaskMultivariateNormal(mean_x, covar_x)
```

Listing 4.7: GP model incorporating gradient observations adapted from the GP model proposed in the 1d example by *GPyTorch* [18] and cleaned up / restructured with insights from [35].

In the first trials the evaluation of the posterior mean of the model failed. Since the model has a multitask structure, the posterior also returns multiple values. However, since only the function value is of interest, the several outputs need to be transformed via a posterior_transform. The weights in this case are 1 for the function value and 0 for all the derivatives. The core idea for the transformation can be found for example at [9] or the tutorial shared by M. Balandat in [6]. This weighting is done since the goal is to optimize the function value and not the derivative for now. The weights are stored in a new attribute, the scale tensor. This is a tensor (vector) of size $(d+1)\times 1$ with a 1 in the first entry and zeros for all others. After achieving a working prototype, a tutorial by A. Yermakov [35] was found which rallies around the same challenge. The advantages of the implementation (cleaner and more straight forward) were subsequently included in this work.

Scaling

In order to achieve better numerical stability and performance, *BoTorch* and the *SciPy* backend recommend for the inputs for the models to be normalized and the outputs to be standardized. This is especially relevant, if the values a very large. Based on some trails the effect is neglectable for values of order one.

BoTorch provided an special input_transform and outcome_transform for all the standard models. For example in the SingleTaskGP. [8]. These built-in transforms enable the user to specify once how the data should be transformed. Afterwards, when evaluating the posterior the user does not have to think about the scaling. One potential usage of the SingleTaskGP with both transforms can be seen in the Listing 4.8 below.

```
model = SingleTaskGP(
    train_X,
    train_Y,
    train_Yvar,
    input_transform=Normalize(d=train_X.shape[-1]),
    outcome_transform=Standardize(m=train_Y.shape[-1]),
    )
```

Listing 4.8: Example for the SingleTaskGP with normalited inputs and standardized outputs using the built-in transforms input_transform and outcome_transform. Adapted from [8].

This works well, but this is not possible for the custom derivative enabled GP. In theory the input_transform would work that way, but the outcome_transform would not work due to the connection between the function values and their derivatives. For the normalization of the input ($x \in [0,1]^d$) the standard approach in Eq. 4.2.1 was used. Depending on the availability of boundaries x_{min} and x_{max} or the lower (lb) and upper bound (ub) were used for the scaling. For the output, the idea was to just manually standardize the whole train_Y tensor individually for each column. The problem, however, is that this can not be done individually for each column. For the function values itself (first column of the `train_Y` tensor) the ordinary standardization with Eq. 4.2.2 can be done.

$$x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} = \frac{x - lb}{ub - lb} \tag{4.2.1}$$

$$y_{\text{scaled}} = \frac{y - \mu_y}{\sigma_y} \tag{4.2.2}$$

For the derivatives one can not simply shift the values by the mean and divide by the standard deviation. However, one must only rescale the derivatives in the same way as the function values were scaled. The shift by the mean is not needed due to the chain rule of differentiation. In addition, since the input values where also scaled (normalized), this scaling must also be taken into account according to the chain rule. In the end, this yields the scaling of the derivatives like

$$y'_{\text{scaled}} = \frac{y'}{\sigma_y} \cdot \frac{1}{x_{\text{max}} - x_{\text{min}}} = \frac{y'}{\sigma_y} \cdot \frac{1}{ub - lb}. \tag{4.2.3}$$

After the training data is rescaled in the above mentioned fashion, it is plugged into the model. Due to the fact that now a model without transformations is just fed with rescaled data, the output of the model is also scaled. Therefore, the result (evaluation of the posterior) needs to be unscaled in the opposite fashion according to Eq. 4.2.4. For the derivatives this must be done following Eq. 4.2.5 which also includes the input scaling factors.

$$y = y_{\text{scaled}} \cdot \sigma_y + \mu_y \tag{4.2.4}$$

$$y' = y'_{\text{scaled}} \cdot \sigma_y \cdot (x_{\text{max}} - x_{\text{min}}) + \mu_y \tag{4.2.5}$$

In the context of scaling some additional aspects need to be considered. The most prominent issue is the maximum amplitude of the derivatives. Since in the current approach the output data is standardized (mean zero and standard deviation one), the scaling factor is the standard deviation of the function values. As already mentioned, the derivatives are only scaled, not shifted, so the values will not have a zero mean. But they will also not have a standard deviation of one, but can be scattered with a substantially larger standard deviation. Up to now, this has not been a problem, however, if the maximum standard deviation is to large some rescaling could be needed. For example, the standard deviation of each column could be calculated and the whole tensor could be divided by the largest standard deviation. This would ensure that all columns fulfill the recommendation of *BoTorch* at least with respect to the standard deviation (range).

Another question is, how often should the data be rescaled. On possibility is to rescale the data each time a new observation is added to the set. This is needed, if one wants to ensure that the training data is standardized in each iteration. However, one could argue that the purpose of rescaling is to enhance the numerical stability and performance/convergence of the parameter optimization. Therefore, it can be sufficient if the mean and standard deviation are reasonably near the expected values. In this case a rescaling could take place after a certain number of iterations or after some threshold is reached. The benefit of this would be that there is no need to determine the mean and standard deviation in each iteration which will enhance the performance.

At this point the data is rescaled in every iteration. The scaling parameters are saved in each iteration along with all the other data. This method enables the possibility to load the (scaled) model later on and unscale the data. This includes the scaling of the test inputs and the unscaling of the model outputs. The scaling method can be controlled in the `config.yaml` file.

However, in the future the implemented scaling capabilities may not be enough and some of the above described approaches or another, maybe more sophisticated approach will be needed. An implementation of an additional scaling method should be feasible without too much work.

4.2.3 Package Usage

The package provides a command line interface for *bayoptder*. The argparser supports a number of different arguments listed below alongside a short description.

- `--path`: The path to the `config.yaml` file.
- `--output`: The path to the output file.
- `--action`: The action to perform. It can be one of the following:
  - `all`: Calls `main.calc_data()` to conduct all tasks (default).
  - `calc`: Calls `main.calc_data()` to calculate the data.
  - `plot`: Calls `plot_data.plot_data()` to generate a 1D or 2D plot of the result.
  - `conv`: Calls `plot_data.plot_convergence()` to generate a convergence plot.
- `--start`: The number of samples to start plotting from. The default is 0.
- `--initial_data`: The path to the initial data. The default is `None`.

The most important ones are the first three. Starting with the `--path` argument, which is the relative path to the `config.yaml` file. In this file all relevant parameters for the calculations are defined. Some examples can be found in the GitHub repository of the package. The next argument is the output path, which defines where the results will be saved to. The third important argument is the `--action`. This controls which of the implemented processes are started. The processes can be split into the calculation/optimization according to Fig. 3.1 and two plotting processes. The plotting is on the one hand the generation of static plots as described in Sec. 5.1 and on the other the generation of convergence plots according to Sec. 5.2. By default all three processes are triggered. However, if the calculation was already done and the iterations where more than 10 samples were present should be done, the arguments `--action="plot"` and `--start=10` would be used. The last argument is the `initial_data` option, where the location to already sampled training data can be provided.

5 Visualization

After the first initial runs of *BoTorch* in a *Jupyter Notebook* [1] the question arose about the best way to visualize the data. In the long run, the goal is to create publication-quality plots. Additionally, a tool of much help to analyze the data and the model exploratory and to visually interpret whether the model is functioning correctly.

There are four main parts that are of interest for the basic visualization:

- training data
- posterior mean
- confidence intervals
- acquisition function

Later on more quantities, like the observed gradients or the ground truth of the toy models, are of interest. These need to be added subsequently.

In the following, the different ideas and tools used for the visualization are discussed. First starting with a simple implementation of 1D and 2D plots, followed by a more sophisticated interactive approach that also allows exploratory data analysis based on *plotly/dash*[2]. At the end of the optimization the data is saved as `netCDF` files using *xarray*[3] ([21] and [22]) which are later on used for the visualization.

5.1 Static 1D/2D Plot

At the beginning, the first step was to identify the simplest version of a plot. Therefore the starting point was to visualize the first three points of the list above for a 1D case. The most common way to plot data in python is the plotting library *matplotlib* [4].

[1]see `https://jupyter.org/`
[2]see `https://dash.plotly.com/`
[3]see `https://xarray.dev/`
[4]see `https://matplotlib.org/`

Training Data

The plotting of the first point, the training data, is quite straight forward and shown in Listing 5.1. Since the training data is stored in the `train_data.nc` file, the first step is to load the `netCDF` file with `xr.open_dataset()` from the *xarray* library. The training data is then just plotted with the `plot()` function of *matplotlib*. If gradient information is also included in the training data than, it will be plotted in 1D in form of linear extrapolation as a tangent at the data point.

```python
train_data = xr.open_dataset(f"{path}/train_data.nc")
train_x = train_data.train_x.values
train_y = train_data.train_y.values
train_dy = train_data.train_dy.values

# Plot training data as black stars
ax.plot(
    train_x
    train_y,
    "k*",
    label="observed values",
)

# Plot training data gradients
if train_data.attrs.get("do_grad"):
    delta = 0.05
    l = train_x - delta
    r = train_x + delta
    yl = train_y - delta * train_dy
    yr = train_y + delta * train_dy

    x_coords = np.stack([l, r])
    y_coords = np.stack([yl, yr])

    ax.plot(
        x_coords,
        y_coords,
        "k",
        label="observed derivative",
        )
```

Listing 5.1: Minimal example of loading and plotting the training data stored in the `train_data.nc` file as black stars into a existing figure using the *xarray* `open_dataset()` function to load the data and the *matplotlib* `plot` function to plot the data.

Note that the `.values` is needed to access the data within the *xarray* `DataSet` or `DataFrame`.

Posterior mean and confidence region

The next step is to plot the posterior mean along with its confidence interval. To get the posterior mean one needs to evaluate the model posterior. This evaluation is done on so-called test points, which are loaded from the `test_data.nc` `netCDF` file in the same way as the training data before. Getting the model is a bit trickier. The model parameters are save in a `model.pth` file. In order to have the correct model at hand, first a fresh model needs to be initialized with the training data in a similar fashion as described in Sec. 4.2. However, the training data needs to be scaled in the same way as in the calculation (specified in the `config.yaml` and the attributes of `train_data.nc`). Since now the model has the default lengthscales, the trained lengthscales of the optimization are needed. Therefore, the saved model parameters are loadaed form the `model.pth` file into the model through `model.load_state_dict()`. To get now the posterior mean, the model is evaluated on the test points using the method `model.posterior(test_x)` method and then the mean is calculated. To determine the confidence bands, the bulit-in *PyTorch* method `confidence_region()` is applied. The plotting is then again quite straight forward. The mean is just plotted as a line with the `plot()` function. The confidence region is shaded using the `fill_between()` function. The whole procedure is sketched as a minimal example in Listing 5.2.

```
#load test data
test_data = xr.open_dataset(f"{path}/test_data.nc")

# initialize model
model = SingleTaskGP(
        train_X,
        train_Y,
    )
# load correct lengthscales
model.load_state_dict(torch.load(f"{path}/model.pth"))

# evaluate model
posterior = model.posterior(test_data.x.values)
mean = posterior.mean
lower, upper = posterior.confidence_region()

# plot posterior mean
ax.plot(test_x, mean, "b", label="mean")

# shade confidance region
ax.fill_between(
    test_x,
    lower,
    upper,
    alpha=0.3,
    label="confidence",
)
```

Listing 5.2: Minimal example of calculating and plotting the posterior mean along with the confidence interval including the initialization, loading and evaluation of the model on given test points.

If the data is scaled as described in Sec. 4.2.2, the test data need to be normalized according to Eq. 4.2.1 and the posterior mean and confidence bands need to be unscaled according to Eq. 4.2.4. If some different scaling scheme is used, the corresponding unscaling needs to be done at this point.

In more than one dimension the test data needs to be a meshgrid instead of just a linear vector which yields matrices for the posterior and the confidence bound. If the problem has more dimensions than the figure, a planar cut must be done for each non-axis dimensions. Typically, this value will be the midpoint of the dimension's range.

Acquisition values

Especially with the discussion about the different acquisition function in mind (see Sec. 3.2) it would be interesting to visualize them. In the course of the optimization loop the acquisition function is initialized (see Sec. 4.2.1) and then the maximum of the acquisition function is calculated (see Sec. 4.2.1) and the location is taken as next sampling location. In order to display the acquisition values, first the acquisition function needs to be evaluated at all test points. The approach is similar to the evaluation of the posterior (see Listing 5.2) including the considerations concerning the scaling. A minimal example of the evaluation and plotting of the acquisition function is displayed in Listing 5.3.

```python
#load test data
test_data = xr.open_dataset(f"{path}/test_data.nc")

# evalute acquisition function
acq_func_values = acq_func(test_data.x.values)

# plot acquisition values
ax.plot(
    acq_func_values,
    label="acq func",
    )
```

Listing 5.3: Minimal example of the evaluation of an analytic (non MC) acquisition function and plotting of the acquisition values for given test points.

As previously mentioned for the evaluation of the posterior, if the model is trained with scaled (normalized) inputs the test data `test_data.x.values` also need to be scaled in the same way. The acquisition values do not need to be unscaled like the posterior mean, as they are not tied to the absolute function values. Additionally, since the next iteration location is selected at the argmax, the absolute values are irrelevant, only the relative values across the domain matter. If the model operates in more than one dimension, the data handling must be adjusted accordingly.

It is important to emphasise that a continuous evaluation of the acquisition function is only possible for analytic acquisition functions. For MC acquisition functions, evaluating different test points requires some recalculation. Due to performance reasons, the results of the multi-start runs of the acquisition function maximization are saved and can be plotted as data points. For both analytic and MC cases, the maximum of the acquisition function is also indicated. Although there are methods to address this issue, they come with difficulties and numerical instabilities. An extended discussion about the can be found at [5] and [7].

1D examples

In the following, two examples of such a plot for the one dimensional case are shown. In Fig. 5.1 in the upper subplot the training data (observed values), the posterior mean (mean), its maximum (best prediction), the confidence interval (confidence) and the ground truth (true values) are shown. In the lower subplot the evaluation of the Expected Improvement acquisition function is shown with its maximum indicated. In both plots, the vertical dashed line marks the location of the next iteration.

In Fig. 5.2 the same setup is shown, but with the use of gradient information (which is indicated by the tangents at the observations). In this case, the Knowledge Gradient acquisition function is used. Therefore, in the lower subplot shows the results of the multi-start gradient ascent to find the best acquisition value. As in the previous figure, the location of the next sample points is indicated with a dashed vertical line.

5.2 Convergence

To compare the performance of different methods, a metric is needed to quantify the quality of the results. In general, two different ways to quantify the convergence of the result for the toy models with known analytical solutions were thought of. One method measures the absolute difference between the value of the best prediction (maximum posterior mean) and the known analytical maximum. The other method measures the distance in the parameter space between the location of the best prediction and the analytical maximum.

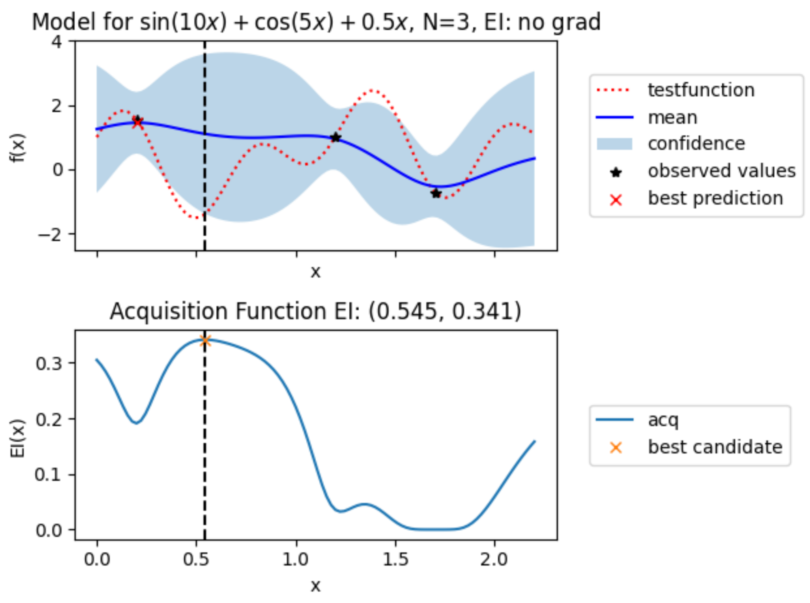


Figure 5.1: Plot of the training data (observations) alongside the posterior mean, its maximum and confidence interval, as well as the ground truth and the Expected Improvement acquisition values with the next sampling point indicated with a vertical dashed black line.

The first one is straightforward, the absolute difference is calculated at each iteration and plotted over the number of iterations. The second one is a bit more tricky. The question is how to calculate the distance between to points in the parameter space. For a one dimensional system this is straightforward. However, in in a multidimensional case this is a bit challenging. The initial approach is to use the $\mathcal{L}^2$-norm. This is reasonable as long as the lengthscales are similar. In case of broadly ranging lengthscales some kind of weighting factor of the different dimensions that correlate with the lengthscale could be useful. However, this was not implemented yet, but could be done in future work.

For real world applications where the ground truth is unknown, it is not possible to plot the differences to the ground truth as described above. Instead, some modification to the convergence approaches need to be done. First, the maximum posterior mean can be plotted over the iterations, and its behavior can be observed. Alternatively, the relative change of the maximum can be plotted, possibly over a few iterations, as improvements may not occur in every iteration. For the second method, where the distance in the parameter space is measured, the distance from the origin or the current best location can be plotted over the iterations. This approach allows the movement of the location of the maximum to be quantified.

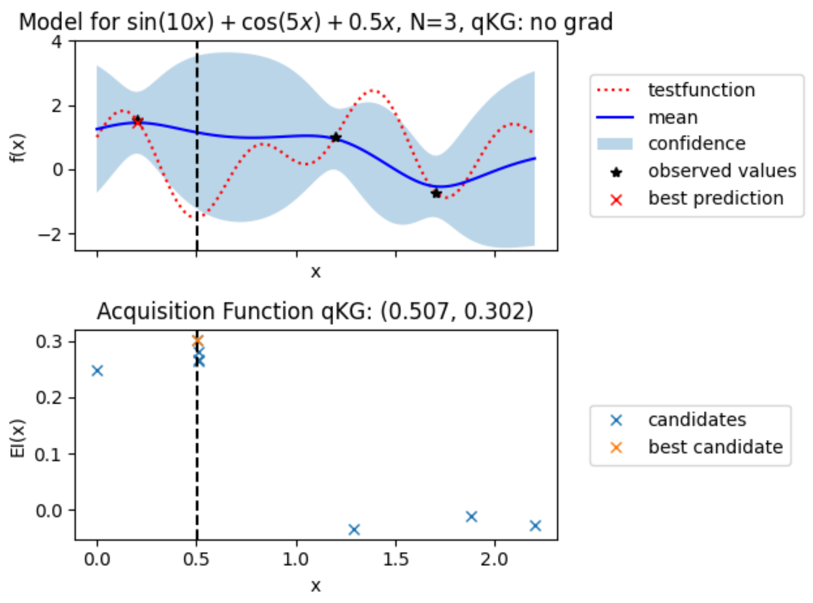


Figure 5.2: Plot of the training data (observations) with the gradient alongside the posterior mean, its maximum and confidence interval, as well as the ground truth and the Knowledge Gradient acquisition values with the next sampling point indicated with a vertical dashed black line.

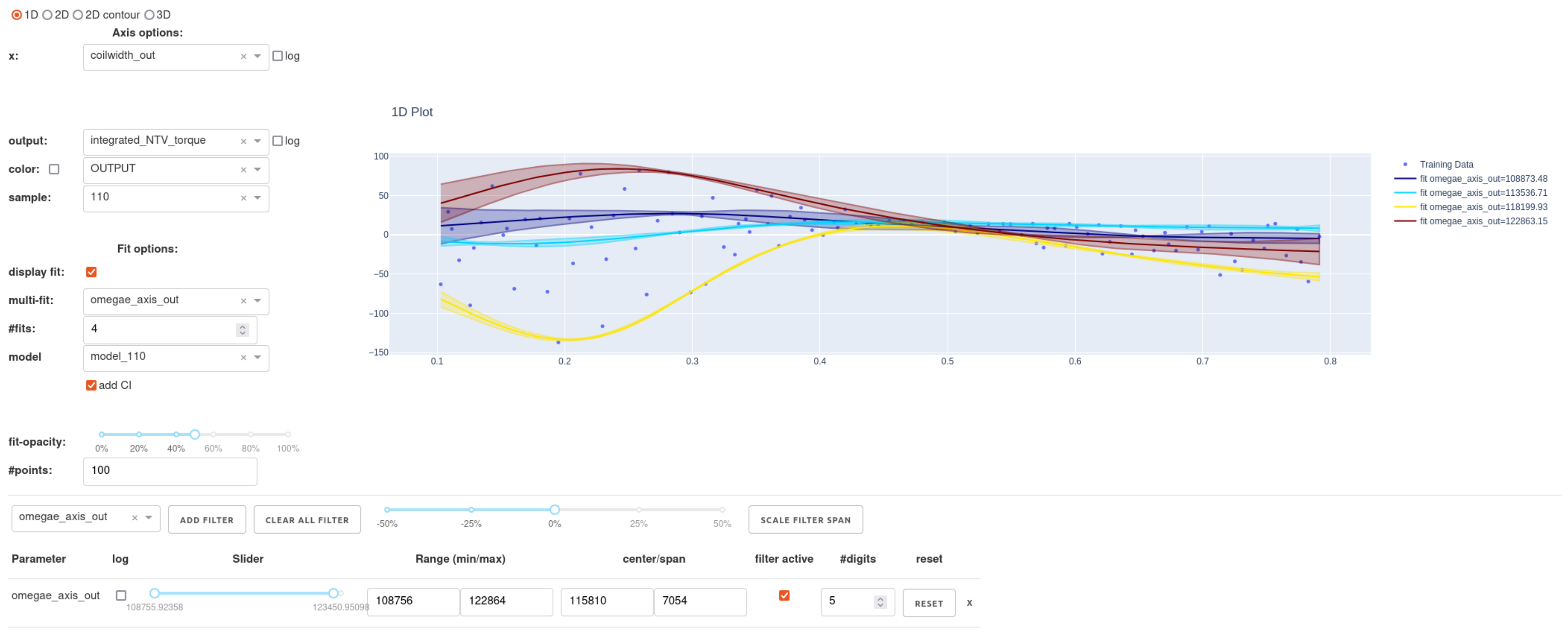
5.3 Interactive UI

When dealing with more than two dimensional parameter spaces, the visualization of the data through 1D or 2D projections and plotting via *matplotlib* becomes more and more challenging. In more than two dimensions, new questions arise when evaluating the posterior. In one and two dimensions this was quite straightforward, just a vector or a meshgrid was needed. However, in a three or more dimensional case where two of this dimensions are along the axis of a 2D plot, there are two possibilities to visualize the posterior. Either one takes the mean over all the other dimensions, or selects a certain value of the other dimensions to evaluate the posterior. In order to conduct exploratory data analysis the latter approach seems more useful. In addition, it would be beneficial if one could quickly change the input dimensions to investigate different correlations. Especially for debugging in the development and exploratory data analysis later on, an interactive tool would be of much help.

In the bachelor thesis of the author, already a similar problem namely the *Visualization of response data from models with high dimensional input spaces* (see [20]) was treated. In the course of the thesis the author developed the interactive graphical analysis tool within *proFit* [1]. The details of the implementation along with a user guide of the original tool can be found in [20] and [29]. In the course of this thesis the tool was extracted from *proFit* and adapted/enhanced to work as a standalone solution. The tool is called *proFitUI* and the code is available at `https://github.com/itpplasma/profitUI`.

The front-end of the tool has remained nearly unchanged. However, in the back-end some substantial changes were made. Within *proFit* the data was mostly stored in `hdf5` binary files (see [28]), while the models where implemented through a universal `surrogate` class with specified methods to evaluate the posterior (see [27]). Due to the use of `netCDF` as file format in the *bayoptder* and the usage of *GPyTorch* models some adaptions were needed. Especially the different ways to access the data caused a lot of work. Previously the dropdown values were used for indexing via a mapping on a linear index. Due to the use of `xarray.DataSets` the indexing is based on variable names and slicing without handmade masks is possible. In addition, for the new structure the calculation and visualization is clearly separated. The data is saved with *bayoptder* and then loaded with *proFitUI*. In *proFitUI* the training data (`train_data.nc`), the model (`model.pth`) and, if available, the test data (`test_data.nc`) are loaded and preprocessed. The evaluation of the model posterior as well as the unscaling (see Sec. 4.2.2) is done within *proFitUI*.

In Fig. 5.3, 5.4, and 5.5 the *proFitUI* interface with all the different filter and model options is shown. The data depicted in the figures is taken from the paper of Grassler et al [17] where the tool was already used.



1D Plot
100
50
0
−50
−100
−150
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Training Data
fit omegae_axis_out=108873.48
fit omegae_axis_out=113536.71
fit omegae_axis_out=118199.93
fit omegae_axis_out=122863.15
ADD FILTER
CLEAR ALL FILTER
SCALE FILTER SPAN
RESET

[✓ 1D] 2D 2D contour 3D

Axis options:

x: coilwidth_out ☐ log

output: integrated_NTV_torque ☐ log

color: ☐ OUTPUT

sample: 110

Fit options:

display fit: ✓

multi-fit: omegae_axis_out

#fits: 4

model: model_110

✓ add CI

fit-opacity: 0% 20% 40% 60% 80% 100%

#points: 100

omegae_axis_out

-50% -25% 0% 25% 50%

Parameter	log	Slider	Range (min/max)		center/span		filter active	#digits	reset
omegae_axis_out	☐	108755.92358 123450.95098	108756	122864	115810	7054	✓	5	RESET

Figure 5.3: Example of a scatter plot with *proFitUI* where one input on the x-axis and the output on the z-axis along with the posterior mean of the model for different values of a third variable including the confidence intervals. The data shown was taken from [17].

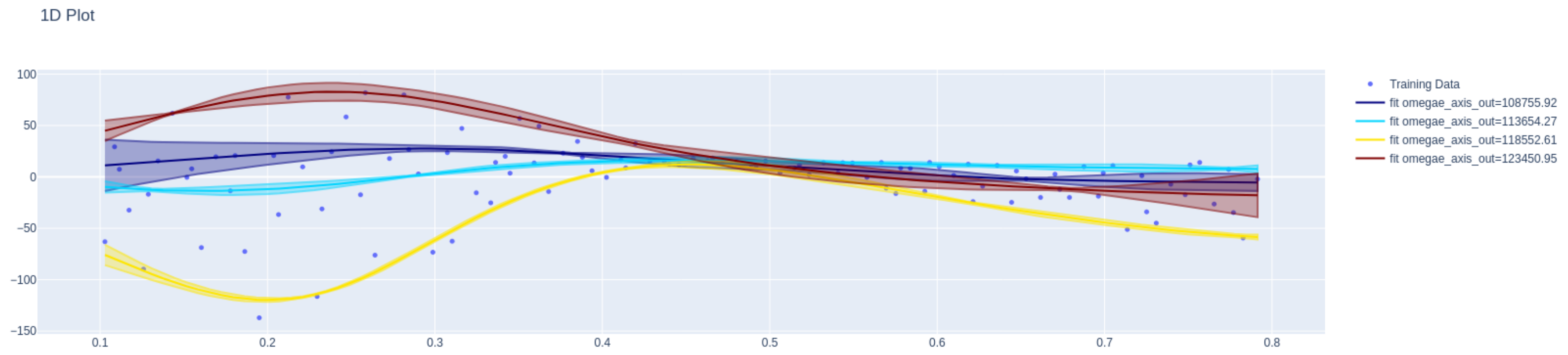


Figure 5.4: Enlarged image of the actual plot in Fig. 5.3 without all the control panels. The data shown was taken from [17].

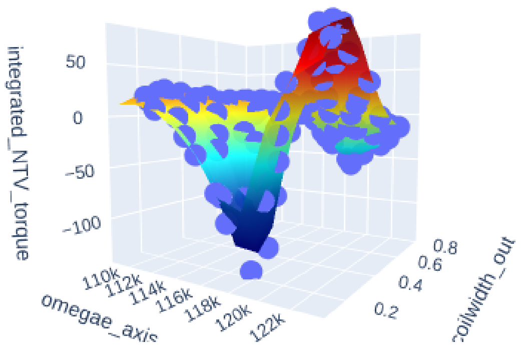


Figure 5.5: Example of a scatter plot with *proFitUI* where two input are on the x- and y-axis and the output on the z-axis along with the posterior mean of the model. The data shown was taken from [17].

6 Comparison of Optimizers

Following the discussion on the implementation, a look on actual results will be shown. Therefore, some benchmarks were done on several toy models. The goal was to compare the Expected Improvement (EI) and the Knowledge Gradient (KG) acquisition function with and without the use of gradient information and also to each other.

6.1 1D Toy Model

As a simple and intuitive mock-up case the function $y(x) = \sin(10x) \cdot \cos(5x) + 0.5x$ was used in the interval $x \in [0, 2.2]$. This lead to a ground truth (testfunction) with four maxima, three local and one global (within the search space). The idea was to sample the function at $N_0 = 3$ random locations and then run the optimization for four iterations to end up with seven observations.

Expected Improvement (EI)

The first case, the Expected Improvement (EI) acquisition function with and without gradient information was investigated. The iterative process is shown in Fig. 6.1. In the upper row the observations (black stars) are shown as sampled from the testfunction (ground truth in dotted red) alongside the posterior mean with the according 95% confidence interval (blue). The current maximum of the posterior mean, the best prediction is shown as red star. In the lower row, the acquisition function value is plotted. The dashed vertical line indicates the location of the maximum of the acquisition which is the proposed location for the next sample.

In the second run, the gradient was also observed and included into the model as described in Sec. 2.3. In Fig. 6.2 the same three starting points where used as before in Fig. 6.1. The gradient observation is visualised by the black line around the data point. This represents a linear extrapolation of the slope at the data point as a tangent.

When comparing the two figures (Fig. 6.1 and Fig. 6.2) it is visible that even with only four points the posterior mean resembles the ground truth a lot better when the gradient information is included in the model. Especially near the observations the slope matches

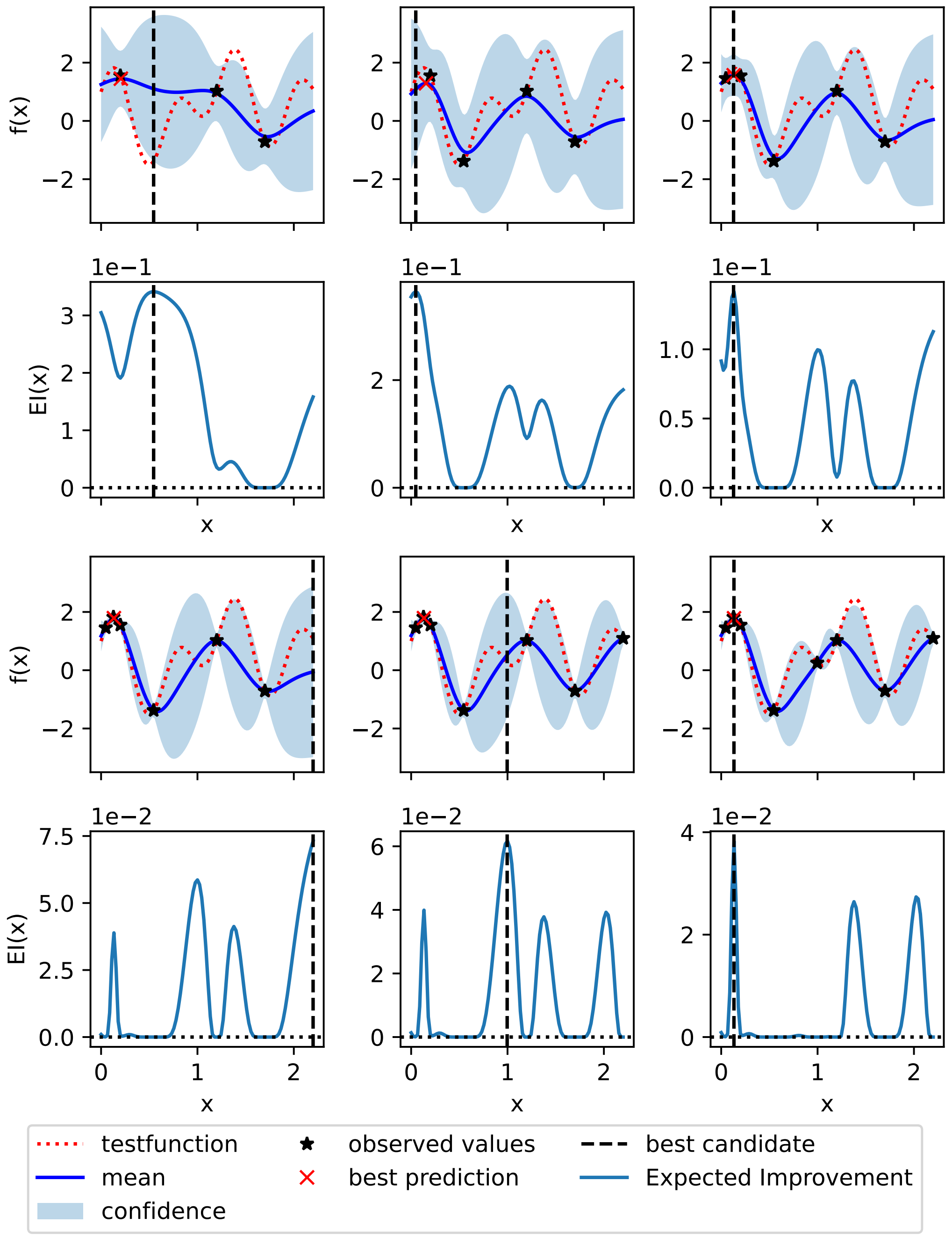


Figure 6.1: Four iterations of the Bayesian Optimization of the testfunction $y(x) = \sin(10x) \cdot \cos(5x) + 0.5x$ for the parameter space of $x \in [0, 2.3]$ with zero noise, $N_0 = 3$ and the Expected Improvement (EI) acquisition function without derivative information.

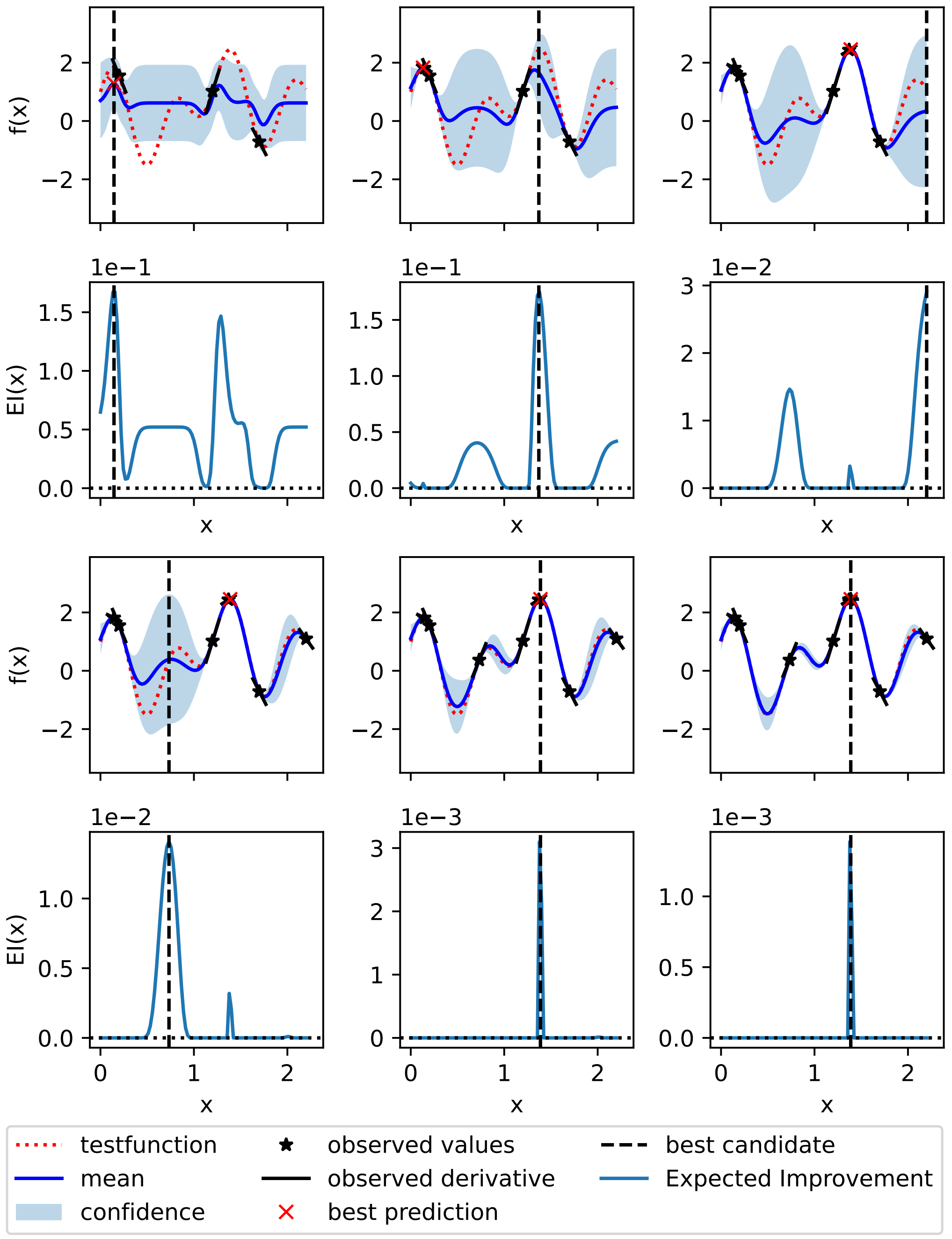


Figure 6.2: Four iterations of the Bayesian Optimization of the testfunction $y(x) = \sin(10x) \cdot \cos(5x) + 0.5x$ for the parameter space of $x \in [0, 2.3]$ with zero noise, $N_0 = 3$ and the Expected Improvement (EI) acquisition function with derivative information.

very well. In addition, the uncertainty near the observations is a lot smaller. After sampling only five new points (eight in total) the testfunction is very well described by the model (posterior mean) and the objective (to find the maximum) is already fulfilled with only five points. In contrast, in the case without the derivative information after eight points the model does not really resemble the testfunction. The lengthscale is too long and the current best prediction is at the highest local maximum, but not the global maximum of the search space. For this comparison it can be concluded that the inclusion of the gradient information increases the quality of the model dramatically.

Knowledge Gradient (KG)

For the toy model, now instead of the Expected Improvement (EI) acquisition function the Knowledge Gradient (KG) acquisition function is used. The case without gradient information is show in Fig. 6.3. In this case, due to the Monte Carlo implementation of the KG in *BoTorch* it is not possible to display the acquisition function over x. The maximum of the acquisition function is found using a multi-start gradient ascent (see Sec. 5.1), therefore it is only possible to show the result of each of the gradient ascents. In the end, the location with the largest acquisition value is used as location for the sampling in the next iteration.

In comparison in Fig. 6.4 the derivative observations are included into the model. Again the derivative observation are shown by the linear extrapolation of the slope at the observed points with a tangent.

Similar to the discussion for the Expected Improvement (EI) it is clearly visible that the inclusion of the gradient information visibly yields significantly better results. The performance improvement will be discussed in detail in the following section on convergence.

Convergence

As discussed in the previous section the result appears to be significantly better when using gradient information in addition. Now the interesting question is, how much does the optimization improve when using the gradient information. Therefore, the convergence calculation described in Sec. 5.2 is used to quantify the difference in the location, whereas for illustrative reasons a slightly different approach was chosen for the values.

In this case, the analytical ground truth for the toy model is known. In the following, the known ground truth for the global maximum is slightly above $y = 2.4$ and indicated with a black dashed line. In addition, the best prediction (maximum posterior mean in Fig. 6.1 to 6.4) and the best training point (observation) is plotted for each iteration. This is shown for both acquisition function and with and without gradient information

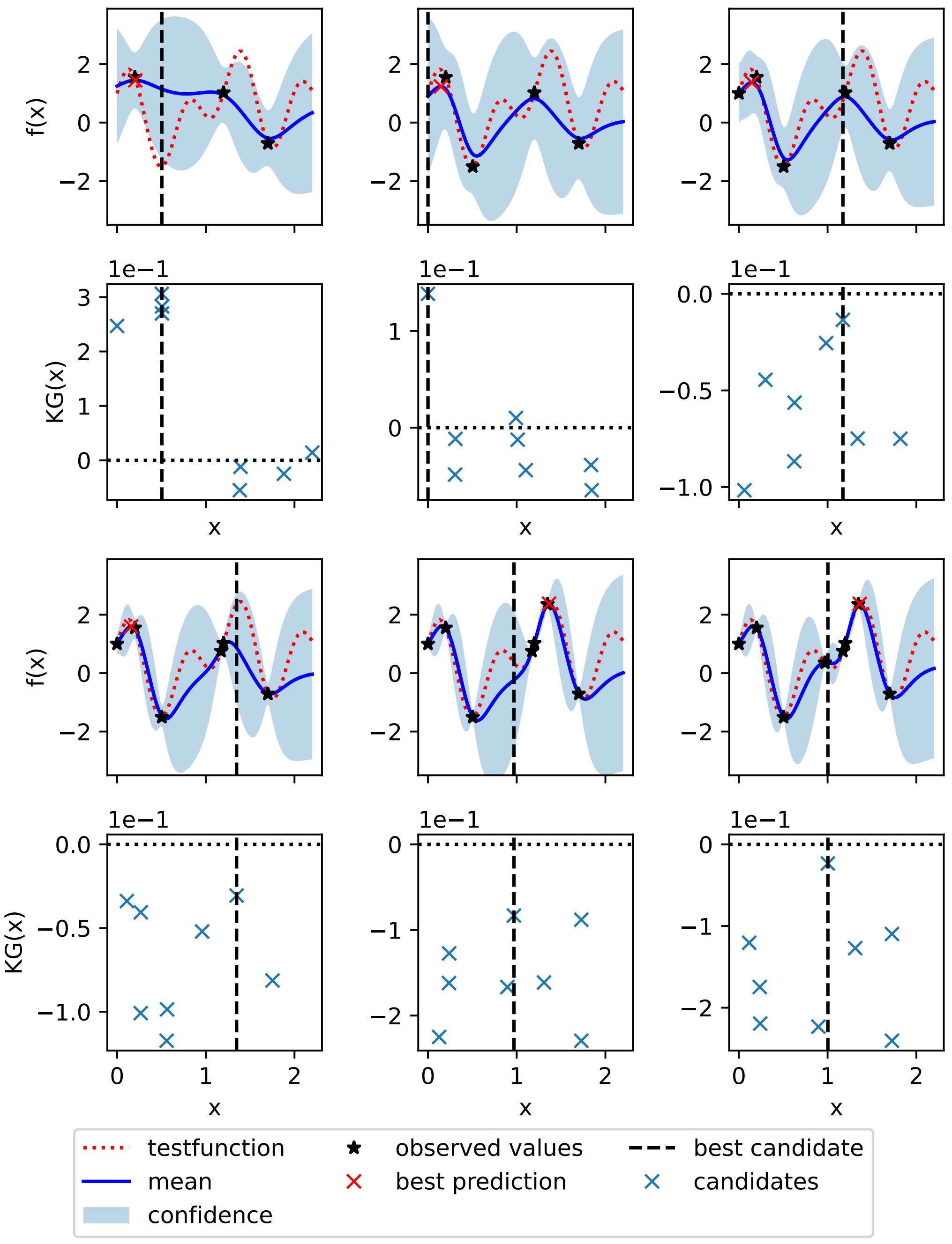


Figure 6.3: Four iterations of the Bayesian Optimization of the testfunction $y(x) = \sin(10x) \cdot \cos(5x) + 0.5x$ for the parameter space of $x \in [0, 2.3]$ with zero noise, $N_0 = 3$ and the Knowledge Gradient (KG) acquisition function without derivative information and a batch size of $q = 1$.

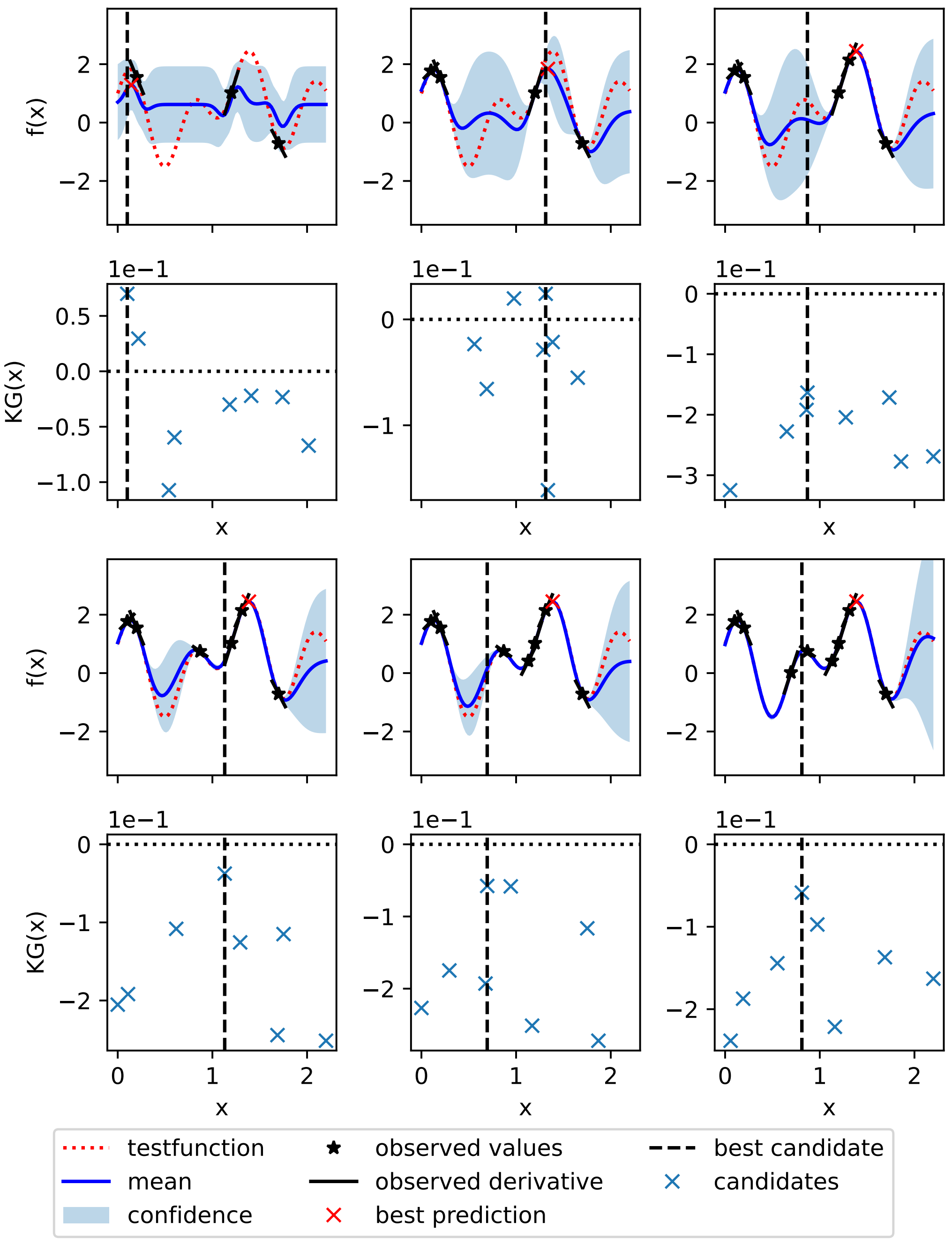
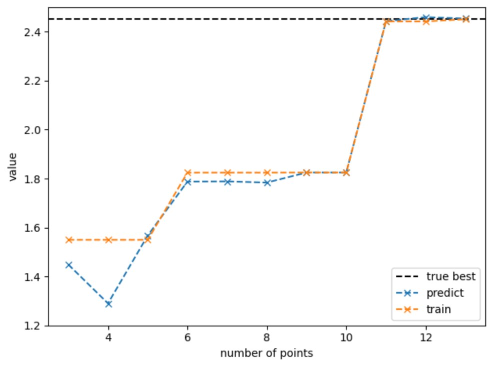
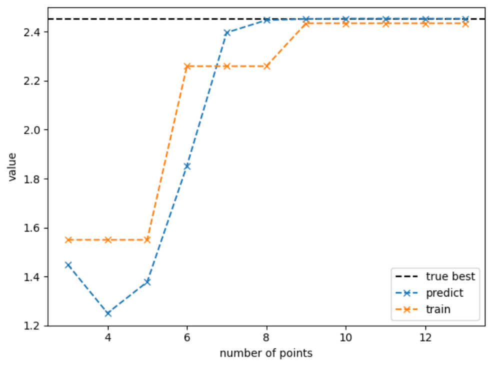


Figure 6.4: Four iterations of the Bayesian Optimization of the testfunction $y(x) = \sin(10x) \cdot \cos(5x) + 0.5x$ for the parameter space of $x \in [0, 2.3]$ with zero noise, $N_0 = 3$ and the Knowledge Gradient (KG) acquisition function with derivative information and a batch size of $q = 1$.

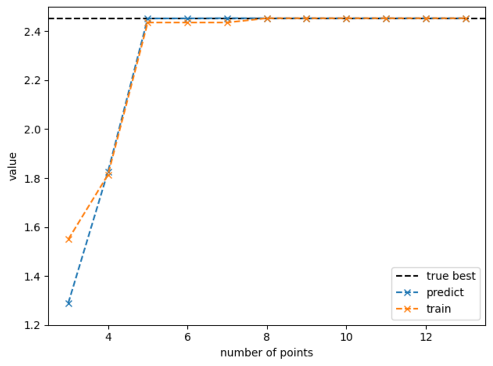
(Fig. 6.1 to 6.4) in Fig. 6.5.



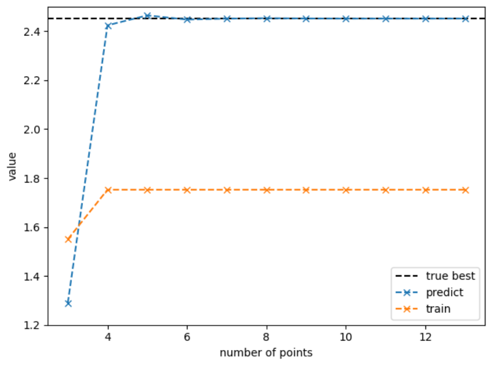
(a) Expected Improvement (EI) without gradient information (see Fig. 6.1).



(b) Knowledge Gradient (KG) without gradient information (see Fig. 6.3).



(c) Expected Improvement (EI) with gradient information (see Fig. 6.2).

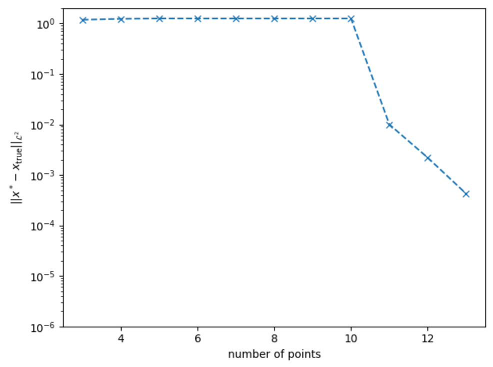


(d) Knowledge Gradient (KG) with gradient information (see Fig. 6.4).

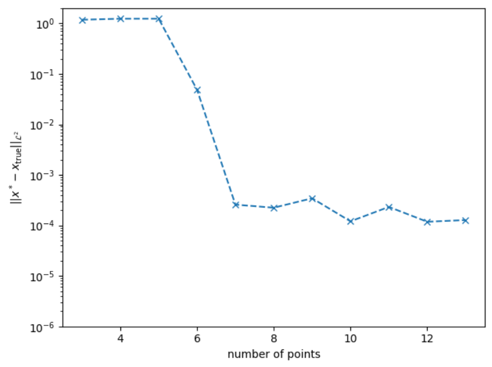
Figure 6.5: Convergence of the current best prediction (maximum posterior mean - red stars), the best training points (observation - black stars) in Fig. 6.1 to 6.4 compared to the analytical maximum for all four variants of the 1D toy model.

The visual assessment in the previous chapter is now shown quantified in a sense that due to the inclusion of the gradient information the maximum values are found with 5 observations (data points) for both acquisition functions. The two models without derivative information need 8 (Knowledge Gradient) and 11 observations (Expected Improvement) to locate the maximum. An interesting observation is that the model with the Knowledge Gradient acquisition function does not sample a training point near the maximum. Nevertheless, the model predicts the maximum values very good, but has small deviations from the analytical solution for the iterations with 5 and 6 observations. The prediction approaches the analytical value in the end, even though no data point was sampled near the maximum.

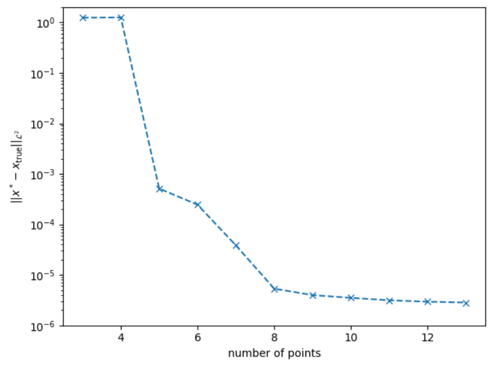
As second way to inspect the convergence, the distances of the location of the analytical maximum and the current best prediction where determined. The resulting convergence curves are displayed in Fig. 6.6.



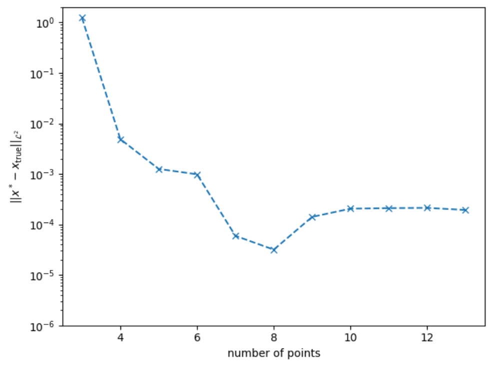
(a) Expected Improvement (EI) without gradient information (see Fig. 6.1).



(b) Knowledge Gradient (KG) without gradient information (see Fig. 6.3).



(c) Expected Improvement (EI) with gradient information (see Fig. 6.2).



(d) Knowledge Gradient (KG) with gradient information (see Fig. 6.4).

Figure 6.6: Convergence of the absolute distance between the location of the current best prediction (maximum posterior mean - red stars in Fig. 6.1 to 6.4) and the location of the analytical maximum for all four variants of the 1D toy model.

In the investigation of the convergence of the distance between the best prediction x^* and the analytical maximum x_{true} two different aspects are visible. For the cases where no derivative information was used the global maximum was not found within the iterations investigated (see Fig.6.1 and 6.3). Both needed more iterations and get close to the analytical maximum after 7 (Knowledge Gradient) and 11 observations (Expected Improvement). Interestingly the distance in the location for KG without gradient is close after 7 iterations, whereas the values is quite a bit of still after 7 iterations. However, in the cases with derivative information the global optimum was found and approached value for the distance in the order of 10^{-5} (EI with gradients) and 10^{-4} (KG with gradients).

One can see that the distance decreases until some threshold is reached. This result is to be expected if the figures Fig. 6.6c and 6.6d are compared with Fig. 6.5c and 6.5d.

6.2 2D Toy Model

Following the 1D toy model, the next more complex, but still easy to visualize case is a 2D toy model. The idea arose upon studying the 2D regression example from *GPyTorch* [19]. Instead of the Franke function a multiplication of sinus and cosine was used. The boundaries for the parameter space of the testfunction $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$ where chosen in a way that along the x_0 direction on local maximum and one minimum (one wavelength) and along the x_1 direction two local maxima and one local minimum (1.5 wavelength) fit. In Fig. 6.7 the ground truth for the 2D toy model is shown.

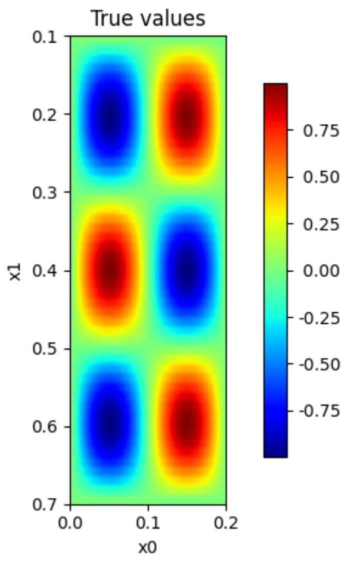


Figure 6.7: Analytic solution (ground truth) for the 2D toy model $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$.

The optimization is done in a similar way to the 1D case described in Sec. 6.1. In this case at the start $N_0 = 5$ observations where selected in a space filling manner as initial condition. For the visualization some adaptions needed compared to the approach described in Sec. 5.1 since now the problem is of higher dimension. Therefore, the *matplotlib* `colormaps` are used.

Expected Improvement (EI)

At first, the Expected Improvement (EI) acquisition function without and with gradient information was investigated. In Fig. 6.8 the optimization is shown for eight iterations without gradient information and in Fig. 6.9 with gradient information. In the left frame the maximum posterior mean is shown as a colormap along with the observations (black stars). The current best prediction is indicated as blue cross while the best candidate for the next sampling is indicated as a black cross. On the right panel the acquisition values a shown alongside the best candidate (black cross).

In Fig. 6.8 it is visible that the Expected Improvement acquisition function tends to sample in the proximity of good observations as described in Sec. 3.2.3. This effect is in this 2D case even more pronounced than in the 1D case in Fig. 6.1. The strong single island like behaviour of the acquisition function in the last four iterations indicates that the usage of a LogEI acquisition function could be useful, but is beyond the scope of the thesis.

In case of the derivative enabled variant, it is clear in Fig. 6.9 that the result is significantly better than in the non-gradient information case. The result even after a few iterations resembles the ground truth a lot better. Already then two of the three maxima were found and the third is at least identified as a local optimum.

Knowledge Gradient (KG)

The same 2D toy model was also optimized for the Knowledge Grdient (KG) acquisition function. Again this was done without (Fig. 6.10) and with gradient information (Fig. 6.11). As discussed before (see Sec. 5.1 and 6.1) the acquisition values can only be plotted for the results of the multi-start gradient ascents. In each plot on the right side there are eight acquisition values shown. However often there are fewer points visible, this is due to the fact that several gradient ascents yield the same location as candidate point. The best candidate is indicated with a black cross in both plots.

One can see that the model with the gradient information outperforms the gradient-free model significantly. In the non-gradient case, one local optimum is found and predicted quite well, whereas in the gradient case, similar to the EI case, two maximum are well described and the third one is slightly visible. A more detailed discussion about the performance can be found in the following section section on convergence.

In Fig. 6.12 a 3D view of the model with 12 observations (Model of Fig. 6.11 with additional candidate point included) is shown using the *proFitUI* (see Sec. 5.3).

Convergence

Apart from the visual interpretation of the figures (Fig. 6.8 to Fig. 6.11) a look on the convergence can be insightful. However, since the model has three equal maxima the evaluation of the convergence is not trivial. There are two approaches with different goals. One is to look at the best prediction and not care about which of the maxima this is at how good the other maxima are described. The other method is to split the parameter space in three subspaces (around the maxima) and look at the best prediction in each of the subspaces.

In the following the model was divided into three subspaces around the maxima, indicated in Fig. 6.7 with the black lines. In Fig. 6.14 the convergence of the best prediction and the best observation is depicted for each of the subspaces. All maxima have a values of $y = 1$, which is indicated with the dashed black line. For each subspace the best prediction (dashed line) and best observation (dotted line) is shown in the colors blue, orange and green.

As expected from Fig. 6.8 the maximum of subspace 2 approaches the true maximum well in Fig. 6.13a, while the other two maxima are not found at all. In addition, since no observation is made in subspace 3 there are no observations (dotted line) for this subspace in the convergence graph. The distance between the analytical maximum and the best prediction and the best observation ($\mathcal{L}^2$-norm) shows similar behaviour. Especially when looking at the distance, some interesting effects are visible. First, in the EI method, the best prediction tends to improve parallel to the best observations. In contrast, for KG, the best prediction improves even when the best observations stays constant. This behavior occurs because KG does not sample at the best location but rather at a location where the maximum posterior mean increases the most. Another noteworthy effect is observed in the KG case without derivative information. Here the optimization of the lengthscales appears have been gotten stuck in a local minimum between sample 7 and 11. As seen in Fig. 6.10 the posterior mean values are comparably close to zero, indicating that the lengthscale was way too long. However, the lengthscale optimization escaped the local optimum after 11 samples. Since the model parameters of the previous iteration are used as starting value for the next (as described in Sec. 4.2.2) it could be beneficial to try a different set (perhaps the default parameters) after a certain number of iterations to avoid such local problems. This adjustment is something that could be done in the future work.

Another illustration of the optimization can be see in Fig. 6.15. Here the movement of the best prediction and best observation on the 2D plane is shown. In the case with gradient information (Fig. 6.15c and 6.15d) the location is found nearly at initialization which is visible through the high precision around the maximum.

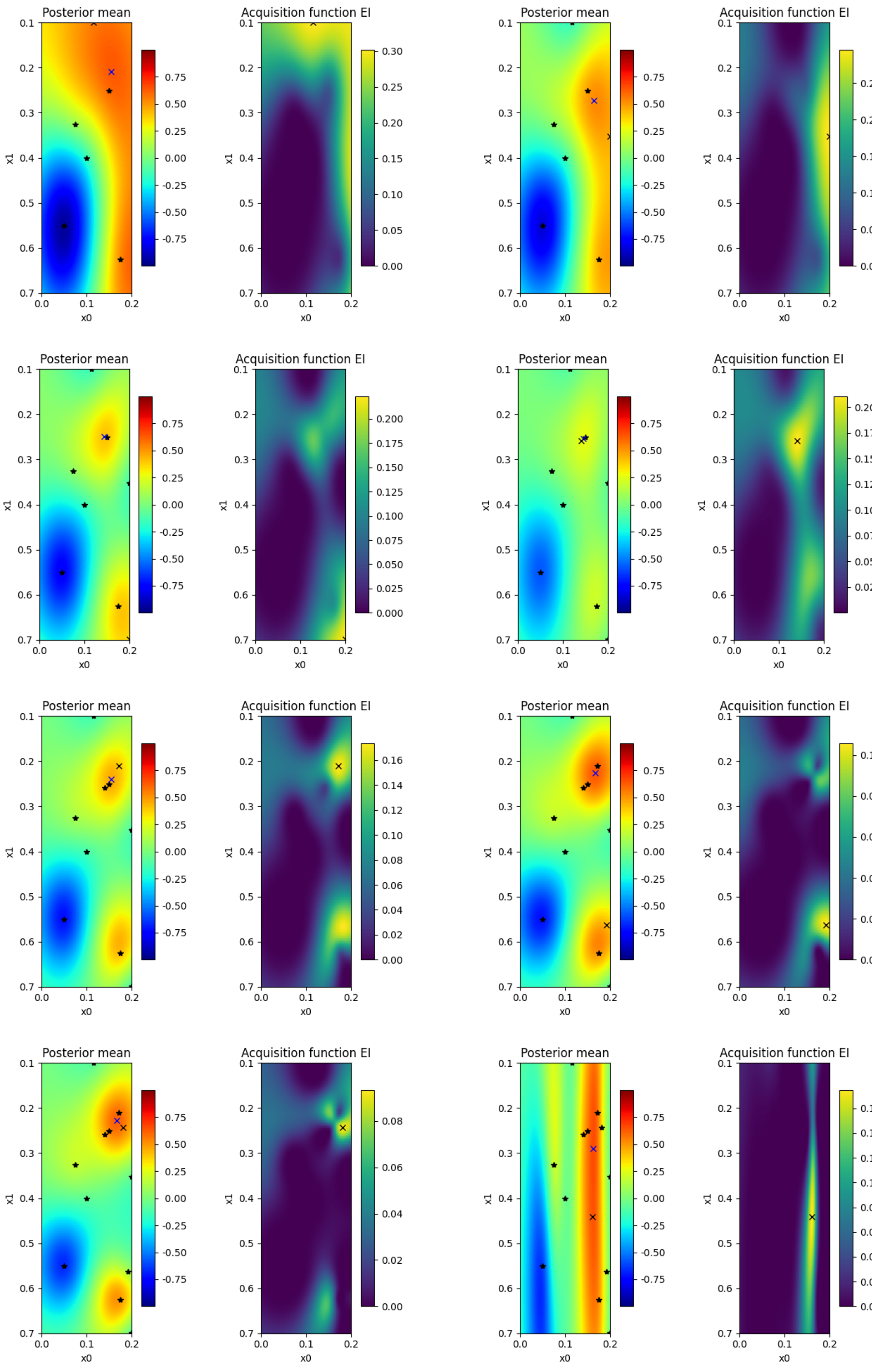


Figure 6.8: Eight iterations of the Bayesian Optimization for the testfunction $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$ with zero noise, $N_0 = 5$ and the Expected Improvement (EI) acquisition function without derivative information.

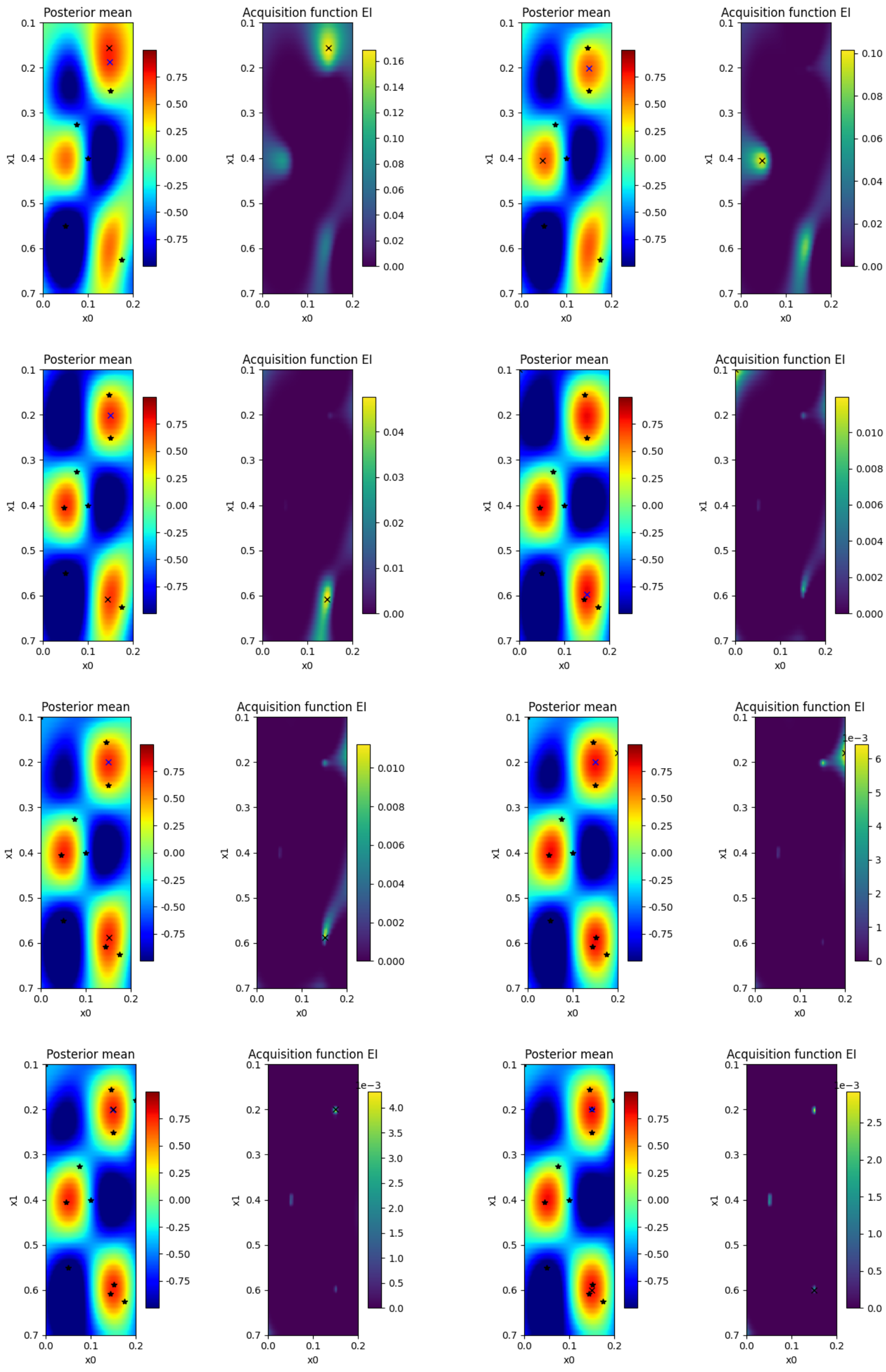


Figure 6.9: Eight iterations of the Bayesian Optimization for the testfunction $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$ with zero noise, $N_0 = 5$ and the Expected Improvement (EI) acquisition function with derivative information.

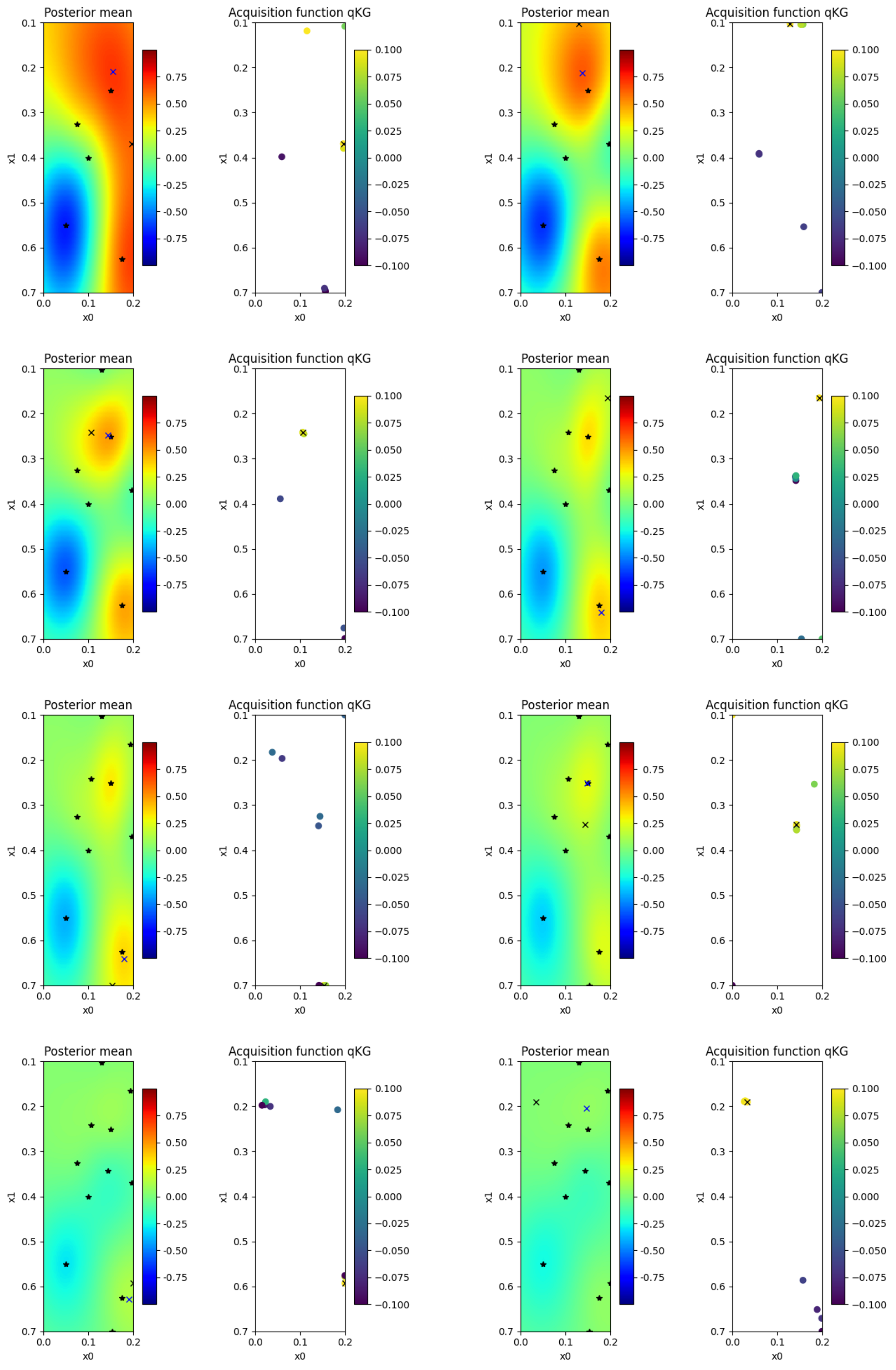


Figure 6.10: Eight iterations of the Bayesian Optimization for the testfunction $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$ with zero noise, $N_0 = 5$ and the Knowledge Gradient (KG) acquisition function without derivative information.

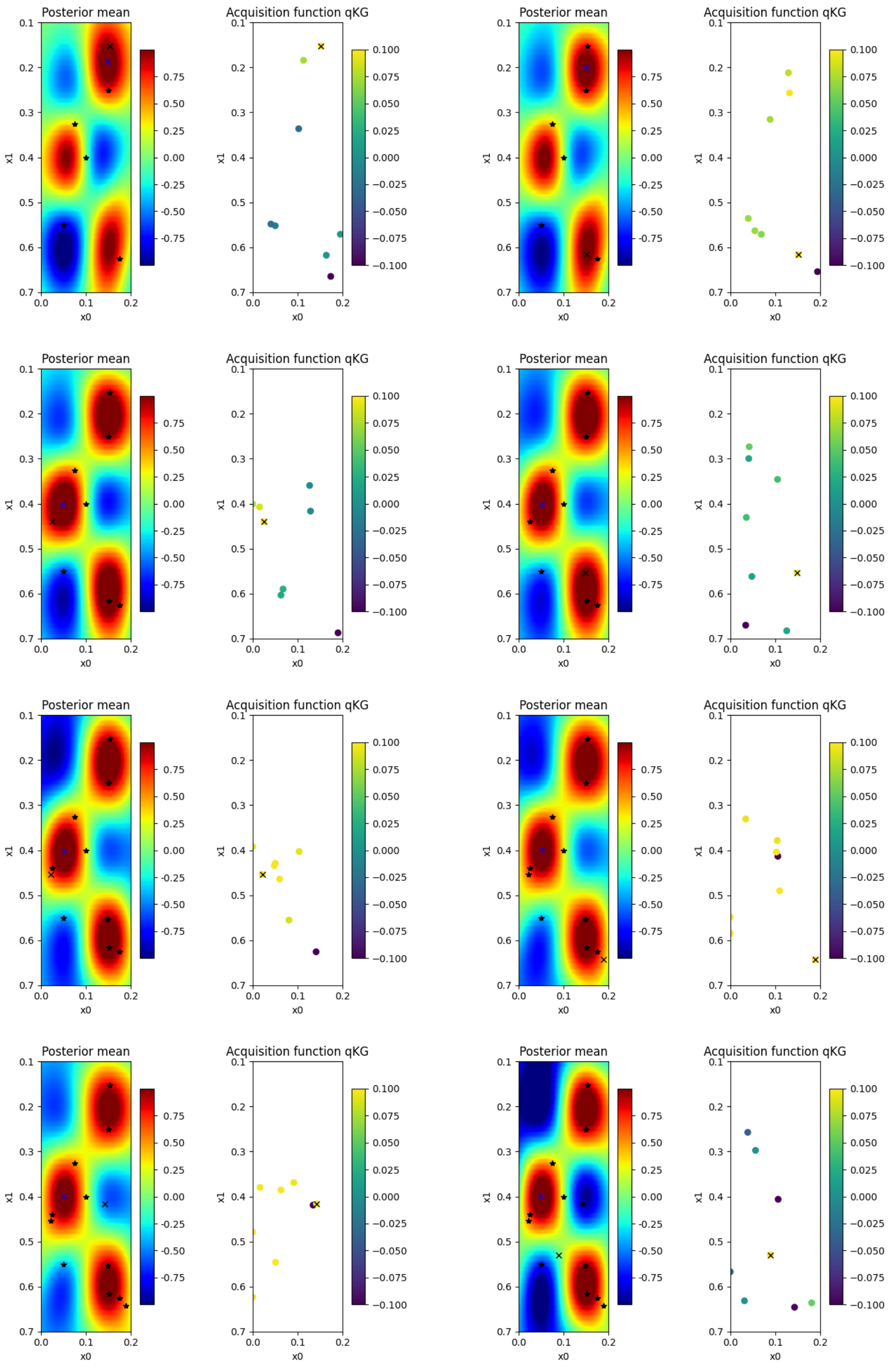


Figure 6.11: Eight iterations of the Bayesian Optimization for the testfunction $y(x_0, x_1) = \sin(10\pi x_0) \cdot \cos(5\pi x_1)$ with zero noise, $N_0 = 5$ and the Knowledge Gradient (KG) acquisition function with derivative information.

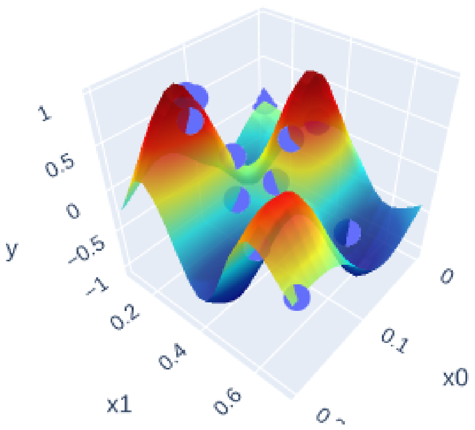
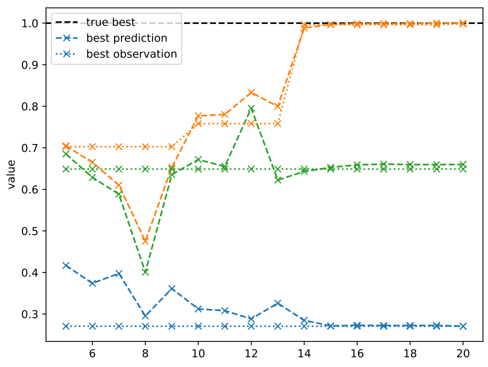
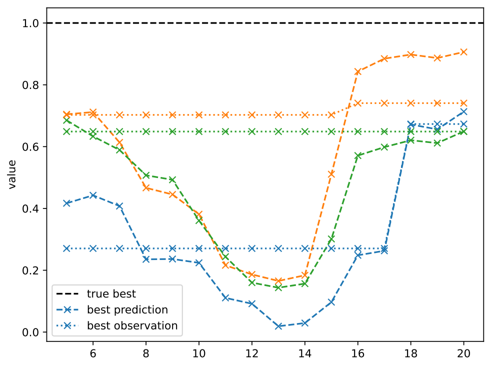


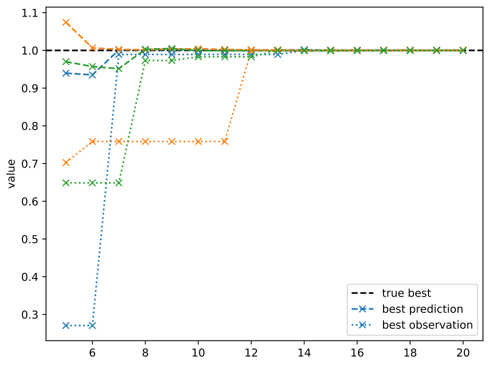
Figure 6.12: 3D scatter plot and responce surface of the model with the Knowledge Gradient (KG) acquisition function with gradients information with $N = 12$ observations (Fig. 6.11 with the additional iterations included) visualized with *proFitUI*.



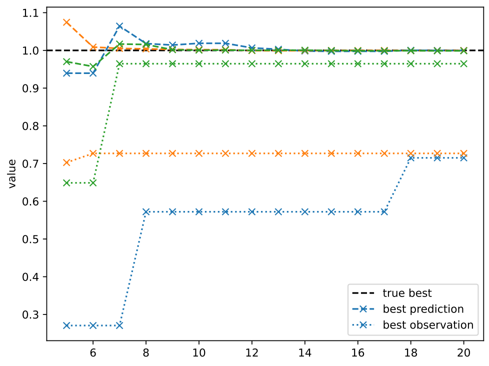
(a) Expected Improvement (EI) without gradient information (see Fig. 6.8).



(b) Knowledge Gradient (KG) without gradient information (see Fig. 6.10).

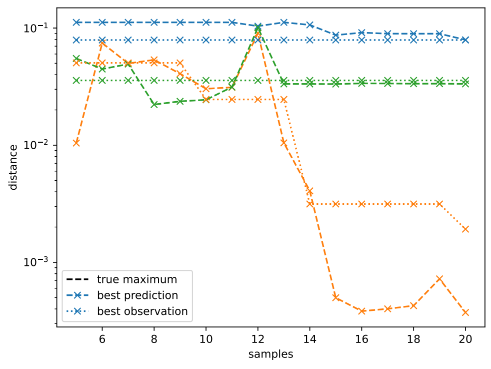


(c) Expected Improvement (EI) with gradient information (see Fig. 6.9).

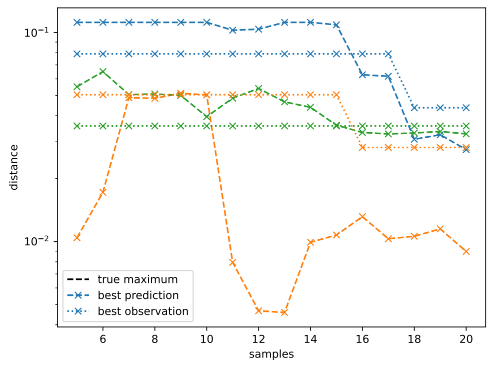


(d) Knowledge Gradient (KG) with gradient information (see Fig. 6.11).

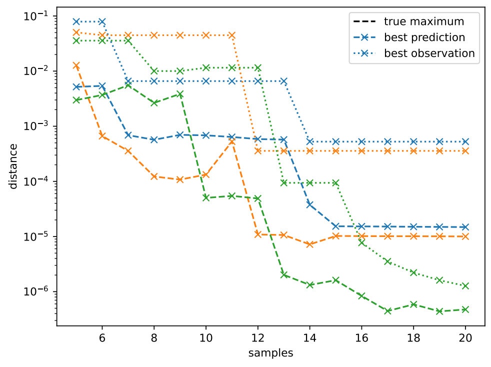
Figure 6.13: Convergence of the current best prediction (maximum posterior mean - dashed line), the best training points (observation - dotted line) in Fig. 6.8 to 6.11) for each of the different subspaces (1: blue, 2: orange, 3: green) compared to the analytical maximum for all four variants of the 2D toy model.



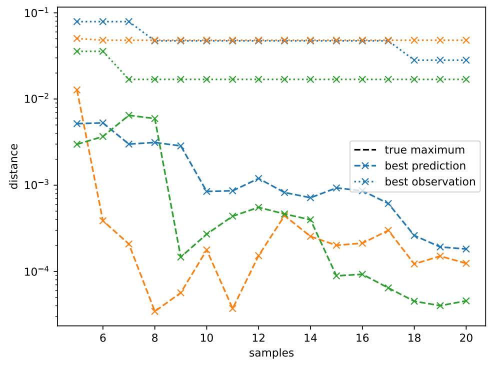
(a) Expected Improvement (EI) without gradient information (see Fig. 6.8).



(b) Knowledge Gradient (KG) without gradient information (see Fig. 6.10).

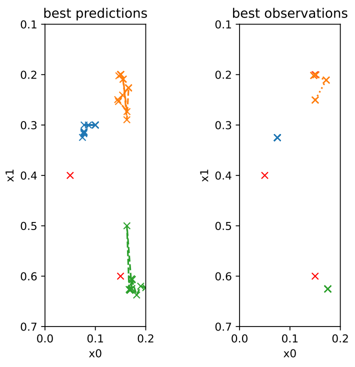


(c) Expected Improvement (EI) with gradient information (see Fig. 6.9).

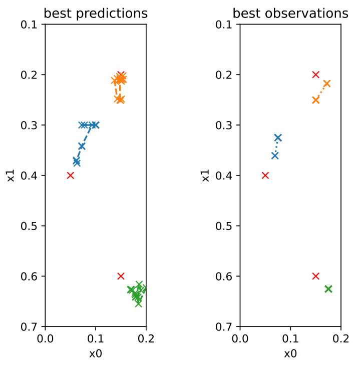


(d) Knowledge Gradient (KG) with gradient information (see Fig. 6.11).

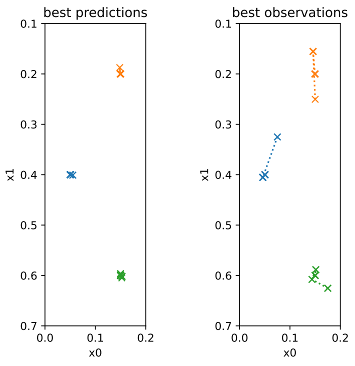
Figure 6.14: Convergence of the euclidean norm of the location of the best prediction (maximum posterior mean - dashed line), the location of the best training points (observation - dotted line) in Fig. 6.8 to 6.11) for each of the different subspaces (1: blue, 2: orange, 3: green) compared to the analytical location of the maximum for all four variants of the 2D toy model.



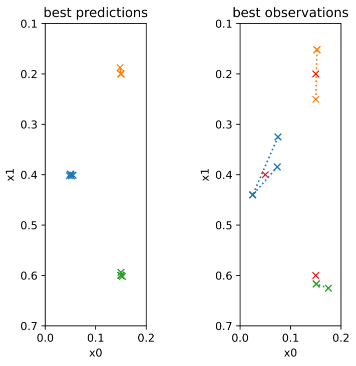
(a) Expected Improvement (EI) without gradient information (see Fig. 6.8).



(b) Knowledge Gradient (KG) without gradient information (see Fig. 6.10).



(c) Expected Improvement (EI) with gradient information (see Fig. 6.9).



(d) Knowledge Gradient (KG) with gradient information (see Fig. 6.11).

Figure 6.15: Movement of the location of the best prediction (maximum posterior mean - left side), the location of the best training points (observation - right side) in Fig. 6.8 to 6.11) in a 2D plane for each of the different subspaces (1: blue, 2: orange, 3: green) compared to the analytical location of the maximum (red crosses) for all four variants of the 2D toy model.

7 Simsopt

In the previous section the performance of the developed package was assessed on two toy models. As a next step it is interesting to apply the methods to an actual problem in plasma physics. Therefore the optimization of coils for a fusion device, in this case a stellarator, was chosen. In the following the optimization of the coils and its challenges are described. The description is largely based on the review paper [23].

The design of a stellarator and the needed optimizations can be split in two major parts:

- Stage-I: magnetic field based on equilibrium models,
- Stage-II: coil shape and current designed to generate the desired magnetic field.

In the remainder of this section the main area of interest will be the above mentioned Stage-II. The focus lies on the optimization of the coils, not the magnetic equilibrium. For details on the determination of the magnetic field the reader is refered to [23].

For the optimization of the coils the Python package *Simsopt*[1] (see [25]) is used. *Simsopt* has high-level Python routines that call `C++` and `Fortran` backends if needed for performance.

The magnetic equilibria used for the coil optimization are taken from QUASR[2]. QUASR (A QUAsi-symmetric Stellarator Repository) is a database of over 320,000 curl-free stellarators and the coil sets that generate them, optimized for volume quasi-symmetry. In general, the configurations can be classified by the number of coils per half period. This is equivalent to the number of different coil shapes.

7.1 Stage-II Optimization

In a Stage-II optimization there are two different parts to be optimized. On the one hand it is the coil shape and on the other the coil currents in order to represent the desired magnetic field. This is done via the minimization of the target normal field on a given surface.

[1]see `https://github.com/hiddenSymmetries/simsopt`
[2]see `https://quasr.flatironinstitute.org/`

Since this thesis focuses on the optimization and not on the plasma physics itself, the approach is mainly based on the *Simsopt* tutorial on Stage-II optimization found at [31]. The first idea was to optimize the Fourier modes of the coils (the coil shape). However, if one wants to include 3 or 4 orders of the Fourier coefficients for a configuration with one coil per half period, the problem has a dimension of order 10^2. It soon turned out, that this is not possible with the current state of the *bayoptder* package. To make this work, a better scaleable kernel and physical informed priors for the lengthscales could help.

Based on this observation, the next approach was to take a configuration from QUASR that has 5 coils per field period, keep the Fourier coefficients fixed and just varys the currents in the coils. This results in a much lower dimensionality. Therefore, the configuration `0124275` was selected (see `https://quasr.flatironinstitute.org/model/0124275`). A picture taken from the site is shown in Fig. 7.1.

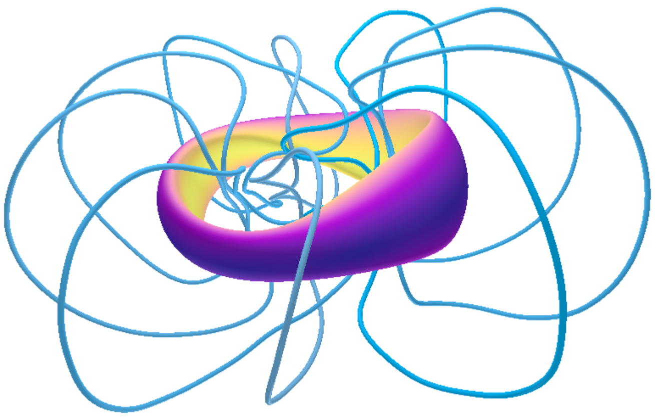
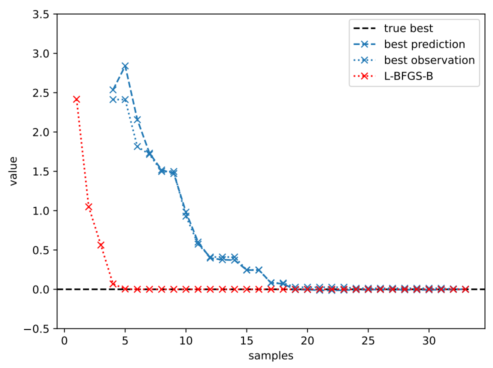


Figure 7.1: Picture of the configuration `0124275` with five coils per field period taken from `https://quasr.flatironinstitute.org/model/0124275`.

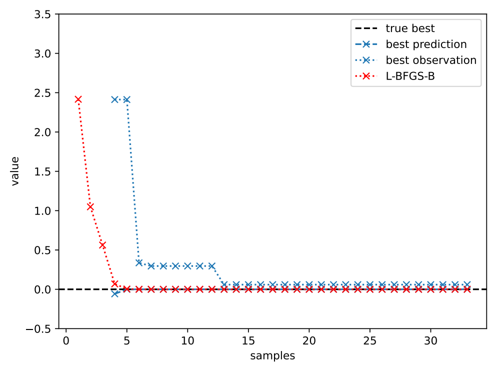
7.2 Implementation

For the optimization, the idea was to use the optimizer developed within *bayoptder* instead of the standard optimizer `scipy.minimize`. Therefore, first the magnetic surfaces and the coil shapes are loaded into *Simsopt*. Since the trivial solution for a vanishing normal magnetic field is a non-existing field, this must be avoided. To achieve this, the current of the first coil was fixed. The other four can be varied which leads to a four-dimensional problem.

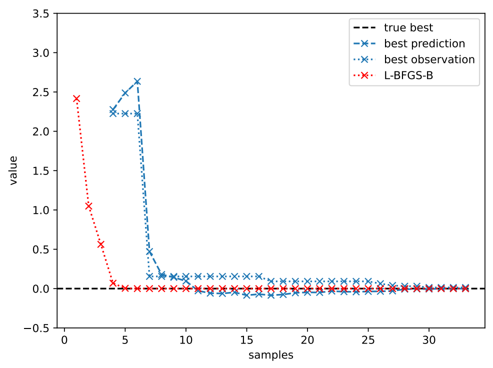
To compare the developed BO optimizer with the `scipy.minimize` three different calculations where made. The first was the optimization of the currents with the standard optimizer using *SciPy* L-BFGS-B till convergence. Then L-BFGS-B runs with a different number of iterations as a reference. To test the BO optimizer the same problem was optimized with both acquisition functions (EI and KG), as well as without and with gradient information. In Fig. 7.2, Fig. 7.3, and Fig. 7.4 the convergence of the methods is compared. For all variants some boundaries and initial values needed to be defined. The *SciPy* optimizer needs one location as initial guess. The values for the currents from QUASR are in the order 10^6. The parameter space for the BO was set to $[-1 \cdot 10^7, 1 \cdot 10^7]^4$ with initial space filling sampling at $N_0 = 4$ locations. As initial guess x_0 for the L-BFGS-B runs the corner of the parameter space with the lowest values was used.



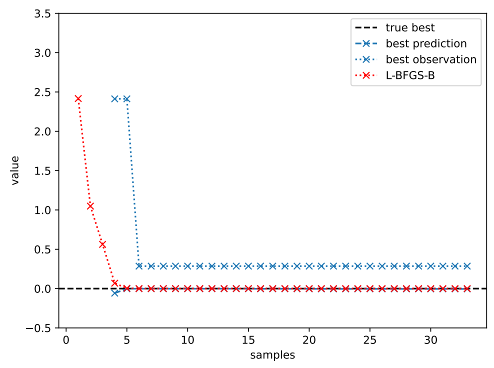
(a) Expected Improvement (EI) without gradient information.



(b) Expected Improvement (EI) with gradient information

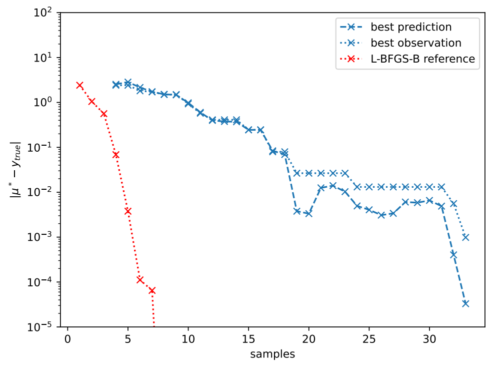


(c) Knowledge Gradient (KG) without gradient information.

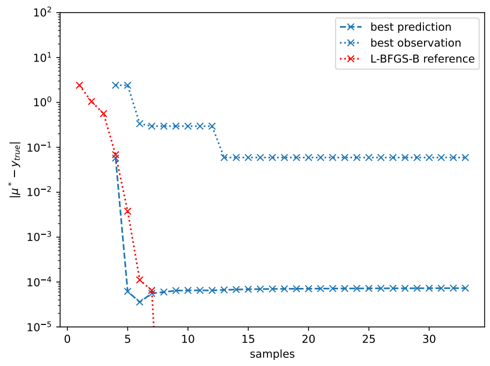


(d) Knowledge Gradient (KG) with gradient information.

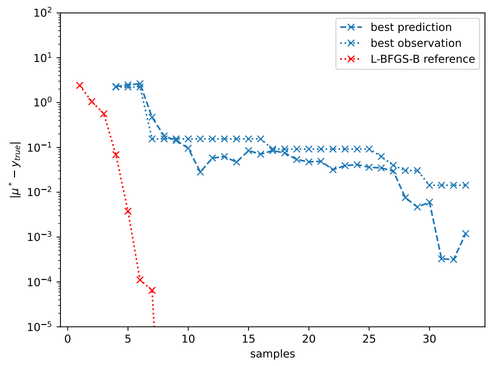
Figure 7.2: Convergence of the best prediction and best observation of the optimization of the currents of the coils 2 to 5 of the configuration `0124275` from QUASR using the *SciPy* `L-BFGS-B` optimizer and the BO optimizer from *bayoptder* with two different acquisition functions and without and with gradients.



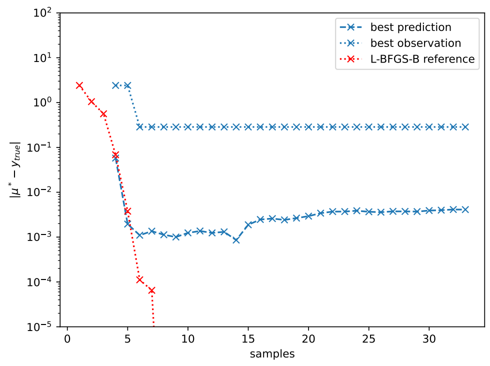
(a) Expected Improvement (EI) without gradient information.



(b) Expected Improvement (EI) with gradient information.



(c) Knowledge Gradient (KG) without gradient information.

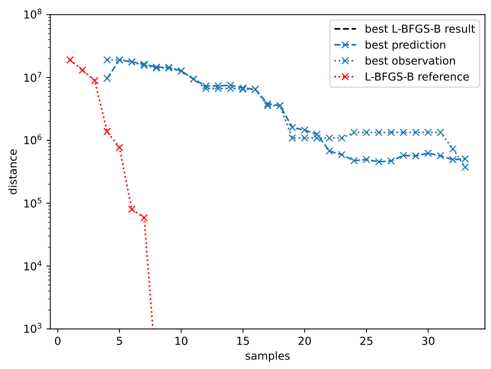


(d) Knowledge Gradient (KG) with gradient information.

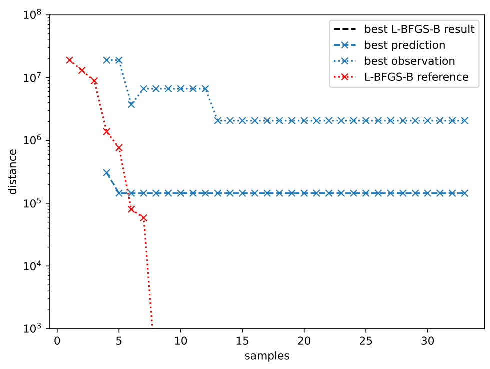
Figure 7.3: Convergence of the best prediction and best observation relative to the best L-BFGS-B result of the optimization of the currents of the coils 2 to 5 of the configuration `0124275` from QUASR using the *SciPy* `L-BFGS-B` optimizer and the BO optimizer from *bayoptder* with two different acquisition functions and without and with gradients.

In Fig. 7.2 it is visible that the cases with gradient observations perform significantly better than the ones without gradient observation as expected from the toy models. It is apparent that the KG case converges faster towards the solution than the EI. In the case with gradient information the best prediction of the model is near the solution right at initialisation, even though the best observations are still far away. This again shows the immense improvement due to the inclusion of the gradient information.

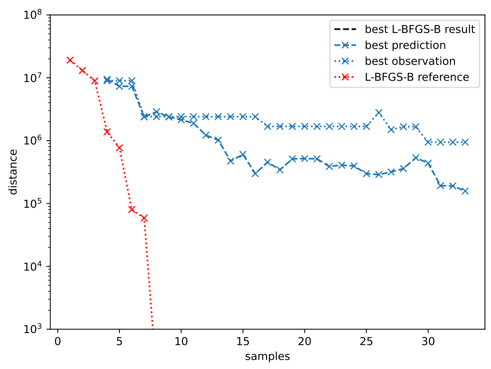
In Fig. 7.3 it is visible that the convergence even for the gradient informed cases is limited to order 10^{-4} for the EI gradient case and order 10^{-3} for the KG gradient case. The slightly worse value for the KG is probably based on the fact that the samples are further away from the best prediction and the model adds some uncertainty. The lower bound



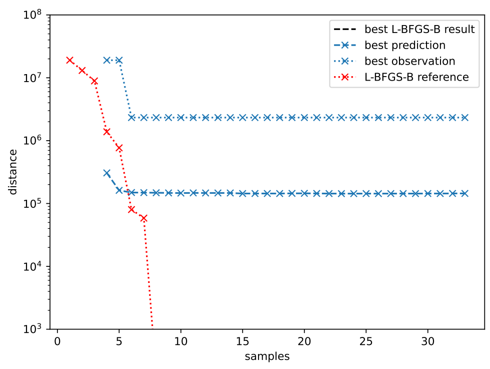
(a) Expected Improvement (EI) without gradient information.



(b) Expected Improvement (EI) with gradient information.



(c) Knowledge Gradient (KG) without gradient information.



(d) Knowledge Gradient (KG) with gradient information.

Figure 7.4: Convergence of the location of the best prediction and best observation relative to the best L-BFGS-B result of the optimization of the currents of the coils 2 to 5 of the configuration `0124275` from QUASR using the *SciPy* `L-BFGS-B` optimizer and the BO optimizer from *bayoptder* with two different acquisition functions and without and with gradients.

of about 10^{-4} is probably due to numerical limitations in the inversion of the matrix in *PyTorch*. However, the origin of this limitation can only be guessed.

Especially in the cases with gradient information it is visible that the location (see Fig. 7.4) is fixed with only a few iterations. This is consistent with the behaviour of the prediction for the values (see Fig. 7.2). However, the distance to the solution of the first run is quite large (order 10^5). This is a strong indication that the problem has a local minimum there and in the course of the 30 iterations no other local optimum is found. This behaviour is similar to the observations made for the 2D toy model, where the non-gradient informed versions did not find the third maximum in the observed number of iterations. Therefore, an acquisition function that has a stronger focus on exploration could help here.

The insights of this example yields the conclusion that the gradient informed method is significantly better at finding optimum compared to the non-gradient case. However, the accuracy is limited to the order 10^{-4}, probably due to the matrix inversion. Therefore it can be useful to use BO to find the local optima and subsequently use a gradient optimizer like L-BFGS-B to find the actual optimum.

8 Conclusion and Outlook

In the course of this thesis the Python package *bayoptder* was developed. It combines the Bayesian Optimization approach of *BoTorch* with the SAA-approximation and the MC acquisition function with the usage of gradient information. The new method shows promising results and significantly increased performance through adding the gradient information.

The effect of the gradient information is vividly shown for the 1D toy model (see Sec. 6.1). The different behaviour of the two acquisition functions Expected Improvement (EI) and Knowledge Gradient (KG) are discussed with respect to exploration vs. exploitation and the computational cost. For the 2D toy model (see Sec. 6.2) the benefit of the gradient information is amplified and the different kinds of convergence criteria with respect to the value or the location are presented.

In Sec. 7 the method is applied to a Stage-II optimization problem of a stellarator. A configuration with results near the optimum was found, but also the limits of the current implementation became apparent, especially with respect to the dimensional scaling.

In addition, the interactive visualization tool originally developed for *proFit* was further enhanced. It now is available as a standalone package named *proFitUI* and can be used for exploratory data analysis and visualization. With regard to the visualization the possibility for further improvements arose, as this was not the main focus of the thesis. This includes, among other things, the possibility to display units, as well as more options to control the non-axis parameters of the fits. In the future, it would be desirable to create an interface between the *plotly/dash* backend and *matplotlib*, in order to generate plots with *matplotlib* after setting the parameters and filters within the *proFitUI* interface.

The *proFitUI* package has already had a use in current research activities. The plasma physics group at the Graz University of Technology applied the package in combination with the *bayoptder* package for the publication of Grassler et al [17].

In the current state of *bayoptder* there is room for improvement. This includes the scalability with respect to dimensions. However, this could be improved through the usage of other kernels, which scale better with the number of dimensions. If the lengthscales of the model spread over several orders of magnitude, the underlying lengthscale optimzer encounters difficulties. Therefore, it could be a solution to assign physical informed priors for the lengthscales. In the current state, the lengthscales from the previous iteration are used as the starting values for the new lengthscale optimization. Furthermore, it could be useful to occasionally employ a different set of starting parameters to avoid getting stuck in a local lengthscale optimum. In addition, the scaling of the observations before conditioning the model could be improved. Instead of scaling in every iteration, there could be a rescale after a fixed number of iterations, or else after a certain threshold is reached.

Regarding the performance of the code, enhancements can be achieved. Since this is more of a proof-of-concept, a lot of diagnostics and intermediate saving options are included that slow down the calculation. Furthermore, with minor adaption to the code, it should be executable on a GPU instead of an CPU. This modification is going to speed up the computation.

In conclusion, the approach shows promising results. After finding the location of the optima by using the Bayesian global optimization with derivative observation, a subsequent application of a gradient optimizer like CG of L-BFGS-B seems to be the most promising use-case. However, some further improvements, as mentioned above, can be incorporated to achieve even better results.

Bibliography

[1] C. Albert et al. *proFit: Probabilistic Response Model Fitting with Interactive Tools*. Version v0.6. 2022. DOI: 10.5281/zenodo.7478488.

[2] C. Albert and K. Rath. “Gaussian Process Regression for Data Fulfilling Linear Differential Equations with Localized Sources”. In: *Entropy* 22.2 (2020). ISSN: 1099-4300. DOI: 10.3390/e22020152.

[3] M. Balandat et al. *BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization*. 2020. arXiv: 1910.06403 [cs.LG].

[4] BoTorch Contributors and M. Balandat. *GitHub Issue 366: What is the role of the 'raw samplers' in optimize_acqf?* https://github.com/pytorch/botorch/issues/366#issuecomment-581951153. Accessed: 2024-06-19.

[5] BoTorch Contributors, M. Balandat, and S. Cakmak. *GitHub Discussion 815: Plotting the acquisition values for knowledgegradient*. https://github.com/pytorch/botorch/discussions/815. Accessed: 2024-06-28.

[6] BoTorch Contributors, M. Balandat, and S. Cakmak. *GitHub Issue 636: [Docs] - Supplying gradients from an external, non-torch function*. https://github.com/pytorch/botorch/issues/636. Accessed: 2024-06-20.

[7] BoTorch Contributors and S. Cakmak. *GitHub Discussion 1470: Question about plotting*. https://github.com/pytorch/botorch/discussions/1470. Accessed: 2024-06-28.

[8] BoTorch Team. *BoTorch: Bayesian Optimization in PyTorch - Source code for botorch.models.gp_regression*. https://botorch.org/api/_modules/botorch/models/gp_regression.html#SingleTaskGP. Accessed: 2024-06-25.

[9] BoTorch Team. *Custom Acquisition Functions*. https://botorch.org/tutorials/custom_acquisition. Accessed: 2024-06-20.

[10] BoTorch Team. *Models*. https://botorch.org/docs/models. Accessed: 2024-06-19.

[11] BoTorch Team. *The one-shot Knowledge Gradient acquisition function*. https://botorch.org/tutorials/one_shot_kg. Accessed: 2024-06-19.

[12] P. Frazier and W. Powell. “The knowledge gradient policy for offline learning with independent normal rewards”. In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE. 2007, pp. 143–150.

[13] P. Frazier, W. Powell, and S. Dayanik. “The Knowledge-Gradient Policy for Correlated Normal Beliefs”. In: *INFORMS Journal on Computing* 21 (2009), pp. 599–613. DOI: 10.1287/ijoc.1080.0314.

[14] P. I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. arXiv: 1807.02811 [stat.ML].

[15] J. R. Gardner et al. *GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration*. 2021. arXiv: 1809.11165 [cs.LG].

[16] R. Garnett. *Bayesian Optimization*. Cambridge University Press, 2023. DOI: 10.1017/9781108348973.

[17] G. Georg et al. “Parameter study of neoclassical toroidal viscous torque for 3D coils in EU-DEMO”. In: *50th EPS Conference Salamanca*. 2024.

[18] GPyTorch Team. *Simple GP Regression with Derivative Information 1d*. https://docs.gpytorch.ai/en/stable/examples/08_Advanced_Usage/Simple_GP_Regression_Derivative_Information_1d.html. Accessed: 2024-06-20.

[19] GPyTorch Team. *Simple GP Regression with Derivative Information 2d*. https://docs.gpytorch.ai/en/stable/examples/08_Advanced_Usage/Simple_GP_Regression_Derivative_Information_2d.html. Accessed: 2024-06-20.

[20] M. Hadwiger. “Visualization of response data from models with high dimensional input spaces”. Bachelor’s thesis. Graz University of Technology, 2021.

[21] S. Hoyer and J. Hamman. “xarray: N-D labeled arrays and datasets in Python”. In: *Journal of Open Research Software* 5.1 (2017). DOI: 10.5334/jors.148.

[22] S. Hoyer et al. *xarray*. Version v2024.03.0. Mar. 2024. DOI: 10.5281/zenodo.10895413.

[23] L.-M. Imbert-Gerard, E. J. Paul, and A. M. Wright. *An Introduction to Stellarators: From magnetic fields to symmetries and optimization*. 2020. arXiv: 1908.05360 [physics.plasm-ph].

[24] K. Kandasamy et al. “Tuning Hyperparameters without Grad Students: Scalable and Robust Bayesian Optimisation with Dragonfly”. In: *Journal of Machine Learning Research* 21.81 (2020), pp. 1–27.

[25] M. Landreman et al. “SIMSOPT: A flexible framework for stellarator optimization”. In: *Journal of Open Source Software* 6.65 (2021), p. 3525. DOI: 10.21105/joss.03525.

[26] J. H. Metzen and G. Lemaitre. *Illustration of prior and posterior Gaussian process for different kernels*. https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_prior_posterior.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-prior-posterior-py. Accessed: 2024-07-27.

[27] Profit Team. *proFit Documentation: Surrogate models*. https://profit.readthedocs.io/en/latest/surrogates.html. Accessed: 2024-07-05.

[28] Profit Team. *proFit Documentation: The Run System.* https://profit.readthedocs.io/en/latest/run_system.html. Accessed: 2024-07-05.

[29] Profit Team. *proFit Documentation: User Interface.* https://profit.readthedocs.io/en/latest/ui.html. Accessed: 2024-07-05.

[30] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning.* The MIT Press, 2006. DOI: 10.7551/mitpress/3206.001.0001.

[31] Simsopt Team. *Coil optimization.* https://simsopt.readthedocs.io/en/latest/example_coils.html#minimal-example. Accessed: 2024-07-28.

[32] J. Wu and P. Frazier. "The parallel knowledge gradient method for batch bayesian optimization". In: *Advances in Neural Information Processing Systems.* 2016, pp. 3126–3134.

[33] J. Wu and P. I. Frazier. *The Parallel Knowledge Gradient Method for Batch Bayesian Optimization.* 2018. arXiv: 1606.04414 [stat.ML].

[34] J. Wu et al. "Bayesian Optimization with Gradients". In: *Advances in Neural Information Processing Systems.* Vol. 30. 2017, pp. 5267–5278.

[35] A. Yermakov. *FOBO Notebook.* https://github.com/pytorch/botorch/blob/a39e6f51fdaaa8b8760717a83b23f6678c61473f/tutorials/fobo.ipynb. Accessed: 2024-06-20.

List of Figures

List of Code Listings