# Bayesian Optimisation Code (***bayesOpt***) User Notes

The MATLAB *package* has been written to implement a Bayesian Optimisation (BO) [[[1]](#endnote-1), [[2]](#endnote-2), [[3]](#endnote-3), [[4]](#endnote-4)] strategy for the global optimisation of an expensive and multi-extremal, and potentially unknown, objective function under a limited budget - typically a maximum number of function evaluations. BO is a black-box, derivative free method. The fundamental underlying concept is that every observation collected by querying the objective function can add to the knowledge about it. This improved knowledge can be subsequently used to determine the location of the next sample, or query, while dealing with the well-known *exploration-exploitation* dilemma.

*Exploration* refers to selecting a new query location in areas where the uncertainty about the objective function is large, but there is potential to improve on the current best-known solution. In contrast, *exploitation* refers to choosing a location close to the current optimal solution. Thus, exploration and exploitation represent two different types of searches: *global* and *local* respectively. In other settings, such as evolutionary and metaheuristic approaches, exploration and exploitation are also known as *diversification* and *intensification* [[[5]](#endnote-5)].

Although we refer to as a package, we do not mean this in the sense of a ***named space*** as implied in MATLAB documentation, rather as a collection of related classes. The concept is the user need only interface with the master process, the class. All the complexity of the underlying algorithm is hidden from the user. The various options for *acquisition* *functions* and *surrogate* *models* are configured at runtime.

Contents

[Bayesian Optimisation Code (***bayesOpt***) User Notes 1](#_Toc115248916)

[1 Preamble 2](#_Toc115248917)

[1.1 Supported Surrogate Models 3](#_Toc115248918)

[1.1.1 Gaussian Process Models 3](#_Toc115248919)

[1.1.2 Random Forest 4](#_Toc115248920)

[1.2 Supported Acquisition Functions 4](#_Toc115248921)

[1.2.1 Expected Improvement 4](#_Toc115248922)

[1.2.2 Upper Confidence Bound 4](#_Toc115248923)

[1.3 Primary Limitations of the Algorithm and Comments on Performance 4](#_Toc115248924)

[2 Note on Abstract Classes and Composition 5](#_Toc115248925)

[2.1 Abstract Interfaces 5](#_Toc115248926)

[2.2 Composition and Aggregation 6](#_Toc115248927)

[3 Installation Instructions & MATLAB Toolbox Dependencies 6](#_Toc115248928)

[4 Package Architecture 6](#_Toc115248929)

[5 The Surrogate Model Interface (***surrogateModel***) 7](#_Toc115248930)

[5.1 The Gaussian Process Regression (***gpr***) Class 8](#_Toc115248931)

[5.2 The Random Forest (***rf***) Class 9](#_Toc115248932)

[6 The Acquisition Function Interface (***acqFcn***) 9](#_Toc115248933)

[6.1 The Expected Improvement (***ei***) Class 9](#_Toc115248934)

[6.2 The Upper Confidence Bound (***ucb***) Class 9](#_Toc115248935)

[7 MATLAB Version and Required MATLAB Toolboxes 9](#_Toc115248936)

[8 Future Enhancements 10](#_Toc115248937)

[9 Glossary 12](#_Toc115248938)

[10 References 13](#_Toc115248939)

# Preamble

Formally, BO is a class of machine learning techniques designed to solve the problem:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

With , the so-called search space, is a black-box expensive and multi-extremal function. Maximisation is considered without loss of generality since the solution to is the same as the solution to [[[6]](#endnote-6)]. BO is relevant in scenarios where a data budget or maximum number of function evaluations is applicable. In practice, may be unknown and all we may have available is some points, , at which has been queried. Hence, denotes the set of training data at the current iteration. As is costly to evaluate and we have a fixed data budget, we require an intelligent strategy to determine the next point at which to query . BO is one such strategy.

The algorithm is comprised of two components: a probabilistic computationally efficient surrogate model, , and an acquisition function, , [i, ii, [[7]](#endnote-7)]. The function is simply the estimate of . As is probabilistic in nature we may also compute its standard error, . Thus, is a measure of , whereas provides a measure of the uncertainty of the estimate. The acquisition function evaluates a trade-off between exploitation and exploration. That is building upon the best-known solution to date (exploitation) and interrogating new parts of the search space, where is large, but we may find a superior solution to that which is already known (exploration).

Note the acquisition function depends on the surrogate model and is consequently numerically cheap to evaluate. Thus, subsequent numerical optimisation procedures which utilise as the cost function are cheap to undertake, compared to optimising directly, assuming it is known. The solution to the following optimisation problem is the next point at which to query the function.

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

In this discourse, we will assume that is unknown and queries are contaminated by gaussian noise. That is:

|  |  |  |
| --- | --- | --- |
|  | , | (3) |

As implied by equation (3), is a Gaussian distributed stochastic variable, . The general variance function can incorporate both heteroscedastic and serially correlated data. Commonly, it is assumed that is independently and identically distributed. That is, . At the iteration, we will write the current training data as the set of ordered pairs . For continuous inputs, will be assumed to be a Gaussian process regression model, say. When the input is comprised of a mixture of discrete and continuous components, we will assume to be a random forest, say. Hence the generic BO-algorithm is:

1. Initially sample locations in and observe .
2. Set: and organise the data as .
3. While , do:
   1. Train on to obtain the functions and .
   2. Maximise the acquisition function:
   3. Observe .
   4. Update the training data pool:
4. Result:

As querying is numerically expensive to evaluate, steps 3a and 3c represent the major computational burden. In contrast, the optimisation of the acquisition function, step 3b, is trivially cheap by comparison. These aspects are the very essence of the algorithm. Note step 3b is the intelligent method by which we select a new point at which to query .

## Supported Surrogate Models

Our intent is to implement both Gaussian process regression and random forests as supported surrogate models. In this section we provide a brief introduction to both forms.

### Gaussian Process Models

Currently, only the Gaussian process regression model is implemented as a surrogate. This automatically limits current applications to systems with continuous inputs and assumes is also smooth. The text by Rasmussen and Williams [[[8]](#endnote-8)] is an excellent reference for users unfamiliar with this technique. As presented in Rasmussen and Williams, Gaussian processes is a generalisation of the Gaussian probability distribution. Whereas a probability distribution describes scalar or vector random variables, is a stochastic process that governs the properties of functions. is completely characterised by a fit model, usually a constant, and a covariance function , . Popular choices for are given below:

|  |  |  |
| --- | --- | --- |
|  | ARD squared exponential | (4) |
|  | ARD exponential | (5) |
|  | ARD Matern 3/2 | (6) |
|  | ARD Matern 5/2 | (7) |

Where denotes automatic relevance determination and:

|  |  |  |
| --- | --- | --- |
|  |  | (8) |

The parameters are often referred to as length scales and allow for the level of smoothing to vary with dimension. Identification of the model essentially amounts to estimating from the training data. By far, the ARD squared exponential covariance function is the

Assuming, additive independent identically distributed gaussian noise, with variance , then it can be shown that:

|  |  |  |
| --- | --- | --- |
|  |  | (9) |
|  |  | (10) |

Where is the prediction, is the corresponding posterior density variance, is the training samples of is the variance-covariance matrix for the training data, the corresponding kernel matrix for and the variance-covariance matrix for . It is this inherent ability to provide and from a single surrogate model which makes the Gaussian process ideal for this application.

#### Large Scale Options

Training with the basic algorithm requires the inversion of an Matrix, where is the number of data points. This requires of order operations where is the number of function evaluations required for estimating and . For large *N*, this becomes computationally prohibitive. One simple mitigating method to reduce the computational burden is to select a subset of the available data, say, where the number of observations in is dramatically lower. The model is subsequently trained on , while ignoring the remaining points. This smaller subset is known as the *active set* or *inducing input set*. Clearly, the computational complexity of the matrix inversion problem is now . This simple approach is termed the *subset of data* (SD) method.

Another option is to approximate the kernel function. One approach is the *subset of regressors* (SR) method.

### Random Forest

Not implemented in this release.

## Supported Acquisition Functions

A very large number of acquisition functions have been proposed in the literature, each offering a different trade-off between exploration and exploitation [[[9]](#endnote-9), [[10]](#endnote-10)]. Candelieri *et al* [i] proposes a partition of acquisition functions into two domains: *mean-variance* and *sampling-based*. In this package we implement only mean-variance acquisition functions and specifically implement two general approaches: the expected improvement and upper confidence bound functions. We discuss these in the next two sections.

### Expected Improvement (EI)

Like all mean-variance approaches, the expected improvement function [[[11]](#endnote-11), [[12]](#endnote-12)] considers only the predictive mean and the standard deviation of . In this context the expectation is taken under the posterior distribution given evaluation of at . As with all members of this family, the fundamental idea is to add an exploitive uncertainty bonus to the exploitive choice based on the predictive mean alone. As the name suggests, the expected improvement function measures the expectation of the improvement over the current best observed value to date, , depending on the predictive distribution of . That is:

|  |  |  |
| --- | --- | --- |
|  |  | (11) |

If and only if , otherwise , where and are the cumulative probability density function and probability density function of the standard normal distribution respectively, and is defined as:

|  |  |  |
| --- | --- | --- |
|  |  | (12) |

If and only if , otherwise . The term is a hyperparameter and controls the balance between exploration and exploitation. The main problem with the approach is in selecting , although is often a good choice. If is too small, the algorithm may converge prematurely to a local optimum. If too large, the algorithm may oscillate around the space and not converge within the data budget.

### Upper Confidence Bound (UCB)

The upper confidence bound method, UCB, utilises the following acquisition function relation:

|  |  |  |
| --- | --- | --- |
|  |  | (13) |

This is intuitively straightforward to understand and the hyper-parameter, , controls the trade-off between exploitation and exploration. Note the hyper-parameter .

## Primary Limitations of the Algorithm and Comments on Performance

Whereas the algorithm is very general in terms of its applications, it also has several limitations in this basic form. In this section we review the limitations so that the user can decide if this approach is appropriate. Where appropriate, we provide guidance on mitigating some of these issues.

Firstly, the limited data budget implies in practice the true global maximum may not be found. Indeed, step 4 of the algorithm simply selects the best result observed so far! In addition, unlike gradient-based optimisation methods, BO does not guarantee improvement across consecutive iterations. This is due to:

* The need to balance exploration and exploitation (global and local search). In addition, the result at each step depends on both the choice of acquisition function and the surrogate model.
* At best, only approximates .

To mitigate this issue, in practice it is essential to monitor the progress of the algorithm as it proceeds. If oscillatory behaviour is observed in terms of the behaviour of across iterations this suggests the acquisition function hyperparameters may favour exploration too much. Under these circumstances it may be necessary to change the hyperparameter values as the algorithm proceeds. How to accomplish this is not always clear or obvious, particularly in the case of the EI algorithm.

The solution is sensitive to the choice of the initial training data, the choice of surrogate model and the acquisition function. Both the solution and the rate of convergence depend on these. Typically, the initial query sites are based on space-filling design of experiment principles [[[13]](#endnote-13), [[14]](#endnote-14), [[15]](#endnote-15), [[16]](#endnote-16)]. Heuristic ideas, such as including known points of interest are also germane. When the inputs to the model and the function to be maximised are known to be continuous, then the surrogate model is often a Gaussian process, . When some or all the inputs to are discrete then the random forest, , is more appropriate as a choice for .

# Note on Abstract Classes and Composition

The code architecture makes considerable use of *abstract interfaces* and *aggregation*. Consequently, we include a brief introduction to these concepts in the following two sub-sections.

## Abstract Interfaces

Abstract classes are extremely useful for describing functionality common to a group of subclasses but requires unique implementations within each subclass. An abstract class cannot be *instantiated*. That is, you cannot create an abstract class object in the workspace. Instead, an abstract class defines the components used by its subclasses. The terminology *abstract member* is used to refer to properties or methods declared in the abstract parent but implemented in a child subclass.

In contrast, a *concrete class* can be instantiated. A concrete class has no abstract members. The terminology *concrete members* applied to properties or methods fully implemented within a class. Note an abstract class may contain concrete as well as abstract members. In this scenario, the concrete elements realised in the abstract class would be required by all subclasses. Thus, an abstract class predominantly forms an *interface*, describing functionality common to a group of subclasses. The abstract class defines the interface of every subclass without specifying the concrete implementation, which is contained in the subclass. Any concrete subclass must implement all inherited abstract members to be able to access (or plug into) the parent interface. The primary advantages of this approach are:

1. Any concrete members in the abstract parent are reused repeatedly in each child application.
2. From a user-perspective, all concrete implementations behave very similarly.

*Inheritance* provides the means of associating the abstract parent with the concrete child class. Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as the *parent* class, whereas the class inheriting the parent attributes is the child class. Inheritance permits the implementation of an ***is-an*** or ***is-a*** relationship among objects. For example, an engineer ***is an*** employee. Class hierarchies can be assembled by inheriting from child classes. Again, for example, we may define an engineer class and subsequently a mechanical engineer class. Inheritance is the appropriate association given that “***a mechanical engineer is an engineer***”. Note as we proceed along the class hierarchy, each child represents an increasing level of specialisation.

## Composition and Aggregation

Composition and aggregation are two of the most fundamental concepts in object-oriented programming. It describes a class that references one or more objects of other classes in instance variables. Essentially an object of another class (child) is stored as a property of the parent. This allows you to model a ***has-a, has-an,*** ***have-a*** or ***have-an*** association between objects. Such relationships occur quite naturally in the real world. For example, a car, has an engine and modern coffee machines may have an integrated grinder and a brewing unit. However, there is a distinct difference in behaviour between composition and aggregation when the parent object is deleted. With composition, when the parent object is destroyed, so is the child. A real-world example would be “when we scrap a car, we also scrap the engine”. In contrast, with aggregation when the parent object is deleted, the child persists. A corresponding real-world example is “a car has passengers, but when we scrap the car, the passengers remain unharmed”. Given their broad use in the real world, it is not surprising that composition and aggregation are routinely applied to software component design. The advantages of composition and aggregation are:

1. Code re-use. The child class requires no modification.
2. Implementing clean interfaces.
3. Changing the implementation of a composited or aggregated child class does not require modification of any external clients.

# Installation Instructions & MATLAB Toolbox Dependencies

The software was developed in ***MATLAB version 9.12.0.1956245 (2022a)*** - Update 2. In addition, the code assumes the presence of the following MATLAB toolboxes:

* ***Optimisation Toolbox, version 9.3*** or later
* ***Statistics and Machine Learning Toolbox, version 12.3*** or later.

The software is available from the following public GitHub repository:

<https://github.com/MarkCaryLboro/BayesOpt>

# Package Architecture

The Bayesian Optimisation package architecture is presented in Figure 1. Note, the ***bayesOpt*** class is the master process and the only one with which the user need by familiar with. In principle, the ***bayesOpt*** class aggregates two concrete implementations: one for the surrogate model and another for evaluating and maximising the acquisition function. Both make use of the OOP strategy pattern [[[17]](#endnote-17)], implying that the specific concrete implementation is selected at run-time. Use of the strategy pattern makes it easy to include alternative surrogate models and acquisition functions. To do so only requires the user to implement new appropriate concrete implementations as indicated in the figure.

The architecture makes use of both inheritance and aggregation as class associations. Inheritance is required to link the abstract interfaces to the respective concrete implementations. However, the acquisition function object is aggregated with the master ***bayesOpt*** class. This provides a clean interface for the optimisation process. The surrogate model is aggregated with the acquisition function object, as this class is required to calculate and . Note the master process does not require knowledge of the surrogate model, only the acquisition function.

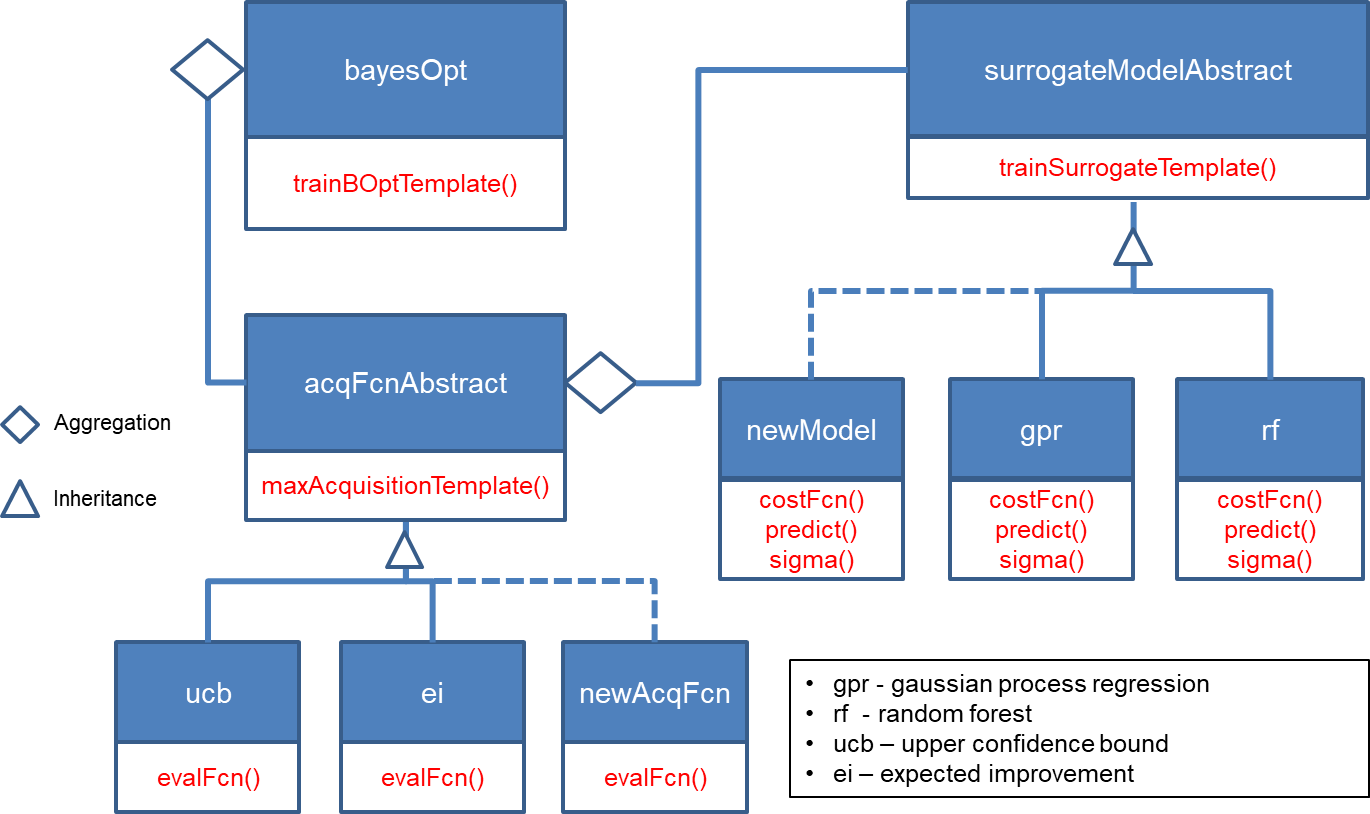


Figure : Bayesian Optimisation Package Architecture. The bayesOpt class aggregates the concrete surrogate model and acquisition function implementations. To extend the choice of surrogate models or acquisition functions the user need only add in new concrete implementations to the corresponding abstract interface.

# The Surrogate Model Interface (***surrogateModel***)

The purpose of the surrogate model interface is to provide a consistent approach to accessing concrete implementations, which inherit from this class. As such the class defines several abstract members which any child class, inheriting from the ***surrogateModel*** abstract parent, must provide a concrete implementation for. Abstract property and method definitions are provided in Table 1 and Table 2 respectively. Any future surrogate model implementations (child class) must implement these abstract members for the interface to function.

Table : Abstract properties defined by the **surrogateModel** class. A concrete implementation must be provided in the child class

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| ModelType | Name of surrogate model | string {“GP”} | Constant |
| X | Training inputs | double | SetAccess = protected |
| Y | Training responses | double | SetAccess = protected |
| Yname | Name of response variable | string | SetAccess = protected |
| Xname | Array of input variable names | string | SetAccess = protected |
| Cov | Kernel (covariance) matrix | double | SetAccess = protected |

Table : Abstract methods defined by the **surrogateModel** class. A concrete implementation must be provided in the child class.

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| trainModel() | Train the surrogate model.  **obj = trainModel( obj, X, varargin );** | Ordinary method |
| predict() | Calculate model outputs at input coordinates provided.  **Y = predict( obj, X, varargin );** | Ordinary method |
| sigma() | Calculate the covariance (kernel) matrix for the input data coordinates provided.  **S = sigma( obj, X );** | Ordinary method |

With reference to Figure 1, if the concrete ***newModel*** class implemented all abstract members it would plugin seamlessly into the interface, permitting the master process class, ***bayesOpt***, to subsequently utilise this new surrogate model alternative, without knowing the specific algorithmic details. In this way, new surrogate modelling approaches can be efficiently integrated into the package.

The ***surrogateModel*** class also implements several concrete members. Concrete properties and methods are defined in Table 3 and Table 4 respectively.

Table : Concrete property members for **surrogateModel** class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| X | Training inputs | double | SetAccess = protected |
| Y | Training responses | double | SetAccess = protected |
| Yname | Name of response variable | string | SetAccess = protected |
| Xname | Array of input variable names | string | SetAccess = protected |
| Xunits | Array of unit symbols for input variables | string | SetAccess = protected |
| Yunits | Unit symbol for response variable | string | SetAccess = protected |
| Trained | Set to true if model has been fitted to training data | logical | SetAccess = protected |
| N | Number of input variables | Int8 | Dependent = true |
| NumPoints | Number of data points | Int64 | Dependent = true |
| DataOk | True if rows of X and Y are equal (i.e., data dimensions are consistent) | logical | Dependent = true |
| Fmax | Best function query value in training data | double | Dependent = true |
| Xmax | Input coordinates corresponding to Fmax | double | Dependent = true |

Table : Concrete method members for **surrogateModel** class.

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| updateModel | Add new data point(s) to the current training data. Model is automatically retrained. | Ordinary method |
| setTrainingData | Set the training data, define properties X and Y. | Ordinary method |
| decode | Map the coded inputs interval, [-1,1], onto the natural units. | Ordinary method |
| setVarUnits | Set the unit symbols for the input variables. | Ordinary method |
| setVarNames | Set the input variable names. | Ordinary method |
| setYname | Set the response variable name. | Ordinary method |

## The Gaussian Process Regression (***gpr***) Class

The ***gpr*** class is essentially a *wrapper* [xiii] around the MATLAB ***RegressionGP*** class provided in the ***statistics and machine learning toolbox***. The concept is to simplify the user interface for multi-input, single output applications, as required for Bayesian optimisation. To review the ***RegressionGP*** class properties refer to the MATLAB user documentation by typing ***doc RegressionGP*** at the command line. *Note with the gaussian process regression approach all input variables must be continuous*. As depicted in Figure 1, the ***gpr*** class inherits from the abstract ***surrogateModel*** class. Table 5 defines the concrete property members for the ***gpr*** class.

Table : List of **gpr** class properties

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| ModelType | Name of surrogate model | string {“GP”} | Constant |
| Kernel | Covariance matrix kernel function. | kernels  kernels{“ ARDsquaredExponential”} | SetAccess = protected |
| PredMethod | Prediction calculation method |  | SetAccess = protected |
| FitMethod |  |  | SetAccess = protected |
| LenScale | Vector of length scales per input variable. |  | Dependent |
| SigmaF | Process noise standard deviation |  | Dependent |
| ModelObj |  |  |  |

## The Random Forest (***rf***) Class

***Not implemented in this version***. However, the random forest model applies in scenarios where the input variables are a mixture of continuous and discrete parameters.

# The Acquisition Function Interface (***acqFcn***)

Table : Abstract acqFcn Class Properties. Abstract members have the "Abstract" attirbute. All other properties are concrete implementations and are inherited by child classes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| FcnName | Name of acquisition function | acqFcnType {“ei”} | Abstract  Constant |
| ModelObj | Aggregated surrogate model object. | gpr | SetAccess = protected |
| BestX |  | double | SetAccess = protected |
| Data | Current training input data | double | SetAccess = protected  Dependent |
| Response | Current response training data | double | SetAccess = protected  Dependent |

Table : Abstract acqFcn Class Methods. Abstract members have the "Abstract" attirbute. All other methods are concrete implementations and are inherited by child classes.

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| evalFcn | Evaluate the acquisition function | Abstract  Ordinary method |
| setBestX | Update the BestX property | Ordinary method |
| addFcnSample2Data | Add a new function query to the training data and retrain the surrogate model. | Ordinary method |

## The Expected Improvement (***ei***) Class

The expected improvement acquisition can be interpreted in the framework of Bayesian decision theory. Essentially, it measures the expected loss associated with evaluating at a point . We choose the point with the lowest expected loss as the next point to query.

## The Upper Confidence Bound (***ucb***) Class

Not implemented in this version.

# MATLAB Version and Required MATLAB Toolboxes

The package was originally developed in MATLAB version 2022a and requires the following MATLAB toolboxes:

1. Statistics and Machine Learning (Version 12.3)
2. Optimisation (Version 9.3)

From a surrogate model perspective, the code currently effectively implements a *wrapper* [xiii] around the MATLAB ***RegressionGP*** class supplied with the statistics and machine learning toolbox. This class implements Gaussian process regression. Future versions will exploit the MATLAB ***TreeBagger*** class and provide a similar wrapper to implement random forest regression. The optimisation toolbox is used for minimising , and consequently determining the next point to query. specifically, the ***bayesOpt*** package utilises the *fmincon* function supplied with the optimisation toolbox.

# Future Enhancements

The code architecture is designed deliberately to make extensions straightforward. Both the acquisition function and surrogate model implementations are based on the OOP-strategy pattern [xvii]. This is relevant in situations when an analyst may have a range of algorithmic options available to solve a specific problem. However, which one to use can only be determined at run time. The *strategy* is an OOP behavioural pattern that enables runtime selection of an algorithm. Thus, instead of implementing a single algorithm directly, code receives run-time instructions as to which, in a family of related algorithms, to use. This permits the algorithm implemented to vary independently from clients that use it. Figure 2 is a schematic depicting the implementation of the *strategy* pattern. Note the *context* class, or *client*, does not implement the strategy directly. Instead, it refers to the strategy interface (***StrategyAbstract***). This makes the context independent of how the algorithm is implemented. The respective concrete strategy classes implement the desired algorithm. The inclusion of new algorithms is very straightforward as it requires only a new concrete strategy class, consistent with the abstract strategy interface. Once implemented, the new concrete algorithm is available to the client via the abstract interface which remains unchanged.

In this initial release, the random forest is not implemented as an alternative surrogate model. Like the ***gpr*** class, the intent is to create a wrapper around an existing MATLAB class implemented in the statistics and machine learning toolbox – the ***TreeGrabber*** class. The advantage of the random forest over the gaussian process is can accept both continuous and discrete inputs.

Diagram

Description automatically generated

Figure : Schematic of the OOP Strategy Pattern

# Glossary

|  |  |
| --- | --- |
| Symbol/Abbreviation | Definition |
|  | Feasible set of input configurations |
|  | Acquisition function |
|  | Surrogate model prediction at point |
|  | Uncertainty in prediction at point |
| BO | Bayesian optimisation |
| EI | Expected improvement |
|  | Objective function |
|  | Gaussian process regression model |
|  | Number of data points |
| OOP | Object-oriented programming |
|  | Random forest regression model |
| SD | Subset of data |
|  | Current training data set |
| UCB | Upper confidence bound |
|  | Input configuration vector: |
|  | Hyperparameter for the expected improvement function |
|  | Observations of contaminated with random noise |

# References

1. A. Candelieri, “A Gentle Introduction to Bayesian Optimization,” in *Proceedings - Winter Simulation Conference*, 2021, vol. 2021-December. doi: 10.1109/WSC52266.2021.9715413 [↑](#endnote-ref-1)
2. Archetti, F., and A. Candelieri. 2019. Bayesian Optimization and Data Science. *Springer* [↑](#endnote-ref-2)
3. Frazier, P. I. 2018. “Bayesian Optimization”. In Recent Advances in Optimization and Modeling of Contemporary Problems, 255–278. INFORMS. [↑](#endnote-ref-3)
4. Shahriari, B., K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. 2015. “Taking the Human out of the Loop: A Review of Bayesian Optimization”. *Proceedings of the IEEE* 104(1):148–175. [↑](#endnote-ref-4)
5. Glover, F., and M. Samorani. 2019. “Intensification, Diversification and Learning in Metaheuristic Optimization”. Journal of Heuristics 25(4):517–520. [↑](#endnote-ref-5)
6. Gill, P. E., Murray, W., Wright, M. H., Practical Optimization, *Academic Press*, First Edition, 1981. [↑](#endnote-ref-6)
7. Cui, J., and Yang, B., *Journal of Software*, 2018, Vol 29, pp 3068-3090. [↑](#endnote-ref-7)
8. Rasmussen, C. E., Williams, C. K. I., Gaussian Processes for Machine Learning, *The MIT Press*, 2006. [↑](#endnote-ref-8)
9. Shahriari, B., K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. 2015. “Taking the Human out of the Loop: A Review of Bayesian Optimization”. Proceedings of the IEEE 104(1):148–175. [↑](#endnote-ref-9)
10. Archetti, F., and A. Candelieri. 2019. Bayesian Optimization and Data Science. *Springer*. [↑](#endnote-ref-10)
11. Mockus, J., V. Tiesis, and A. Zilinskas. 1978. “The Application of Bayesian Methods for Seeking the Extremum”. Towards global optimization 2(117-129):2. [↑](#endnote-ref-11)
12. Jones, D. R., M. Schonlau, and W. J. Welch. 1998. “Efficient Global Optimization of Expensive Black-box Functions”. *Journal of Global Optimization* 13(4):455–492. [↑](#endnote-ref-12)
13. Cary, M., A model Based Engine Calibration Methodology for a Port Fuel Injection, Spark-Ignition Engine, *Ph.D. Thesis*, Bradford University, 2003. [↑](#endnote-ref-13)
14. Stein, M., Large Sample, Properties of Simulations Using Latin Hypercube Sampling, *Technometrics*, 1987, 29, pp 143-151. [↑](#endnote-ref-14)
15. I. Friedel and A. Keller. Fast Generation of Randomized Low-Discrepancy Point Sets. In H. Niederreiter, K. Fang, and F. Hickernell, editors, Monte Carlo and Quasi-Monte Carlo Methods 2000, pages 257–273. *Springer*, 2002. [↑](#endnote-ref-15)
16. I. Sobol’. On the Distribution of Points in a Cube and the approximate Evaluation of Integrals. *Zh. vychisl. Mat. mat. Fiz*., 7(4):784–802, 1967 [↑](#endnote-ref-16)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, *Addison-Wesley Professional Computing Series*, 1995, ISBN 0-201-63361-2. [↑](#endnote-ref-17)