# Bayesian Optimisation Code (***bayesOpt***) User Notes (Release 1.0)

The MATLAB *package* ***bayesOpt*** has been written to implement a Bayesian Optimisation (BO) [[[1]](#endnote-2), [[2]](#endnote-3), [[3]](#endnote-4), [[4]](#endnote-5)] 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-6)].

Although we refer to ***bayesOpt*** 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 ***bayesOpt*** 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 (Release 1.0) 1](#_Toc119060392)

[1 Preamble 2](#_Toc119060393)

[1.1 Supported Surrogate Models 3](#_Toc119060394)

[1.1.1 Gaussian Process Models 3](#_Toc119060395)

[1.1.2 Random Forest 4](#_Toc119060396)

[1.2 Supported Acquisition Functions 4](#_Toc119060397)

[1.2.1 Expected Improvement (EI) 5](#_Toc119060398)

[1.2.2 Upper Confidence Bound (UCB) 5](#_Toc119060399)

[1.3 Primary Limitations of the Algorithm and Comments on Performance 6](#_Toc119060400)

[2 Note on Abstract Classes and Composition 6](#_Toc119060401)

[2.1 Abstract Interfaces 6](#_Toc119060402)

[2.2 Composition and Aggregation 7](#_Toc119060403)

[3 Installation Instructions & MATLAB Toolbox Dependencies 7](#_Toc119060404)

[4 Package Architecture and the ***bayesOpt*** Class Description 8](#_Toc119060405)

[4.1 Creating and using a ***bayesOpt*** Object 10](#_Toc119060406)

[5 The Surrogate Model Interface (***surrogateModel***) 11](#_Toc119060407)

[5.1 The Gaussian Process Regression (***gpr***) Class 13](#_Toc119060408)

[5.2 The Random Forest (***rf***) Class 15](#_Toc119060409)

[6 The Acquisition Function Interface (***acqFcn***) 15](#_Toc119060410)

[6.1 The Expected Improvement (***ei***) Class 16](#_Toc119060411)

[6.2 The Adaptive Expected Improvement (***aei***) Class 17](#_Toc119060412)

[6.3 The Upper Confidence Bound (***ucb***) Class 17](#_Toc119060413)

[7 Worked Example and the “BayesOptExample” Live Script 18](#_Toc119060414)

[8 MATLAB Version and Required MATLAB Toolboxes 22](#_Toc119060415)

[9 Future Enhancements 23](#_Toc119060416)

[10 Glossary 24](#_Toc119060417)

[11 References 25](#_Toc119060418)

# 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-7)]. 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-8)]. 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-9)] 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 commonly encountered kernel in the literature. For compactness, we denote the combined vector of length scales and noise variance as .

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. Here the exact kernel function is replaced by a smaller approximation given an active set. The concept is to approximate the span of the original high dimensional basis with the lower dimensional alternative. The MATLAB ***RegressionGP*** class finds the best approximation by minimising the measure:

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

Where is the reproducing kernel Hilbert spaces (RKHS) associated with the kernel function [viii]. Details of these and other large-scale approaches can be found in the online documentation for the MATLAB Statistics and Machine Learning toolbox.

### Random Forest

The random forest is an alternative surrogate model approach. This is based on random multiple decision trees. Unlike the Gaussian process regression model, it is not restricted to continuous inputs. Consequently, a mixture of discrete and continuous inputs may be supported. Discrete factors, *e.g.,* the number of pilot injections, arise relatively frequently in power train applications. However, at this juncture this modelling method is not supported.

## 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-10), [[10]](#endnote-11)]. 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-12), [[12]](#endnote-13)] 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 wildly through the space and not offer any significant performance improvement 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 again the hyper-parameter, , controls the trade-off between exploitation and exploration. Note the hyper-parameter . Unlike the EI algorithm of section 1.2.1, we implement an algorithm due to Berk *et al* [[[13]](#endnote-14)]. This is builds on the work of Srinivas *et al* [[[14]](#endnote-15)], whose approach is founded on the *Bayesian regret* principle. In stochastic game theory, Bayesian regret is the expected difference (or regret) between the utility of a Bayesian strategy and that of the optimal strategy with the highest expected payoff. Although Srinivas’ approach is intuitive, their presented algorithm overestimates . Indeed, in their paper they recognise and suggest dividing the result from the algorithm by a factor of 5 in practice. The primary contribution of Berk et al is to provide an alternative sampling method to selecta smaller and more viable , while simultaneously maintaining a regret bound. Their proposed algorithm is termed *randomised Gaussian process upper confidence bound* (RGP-UCB). RGP-UCB pseudo code is presented in Figure 1.

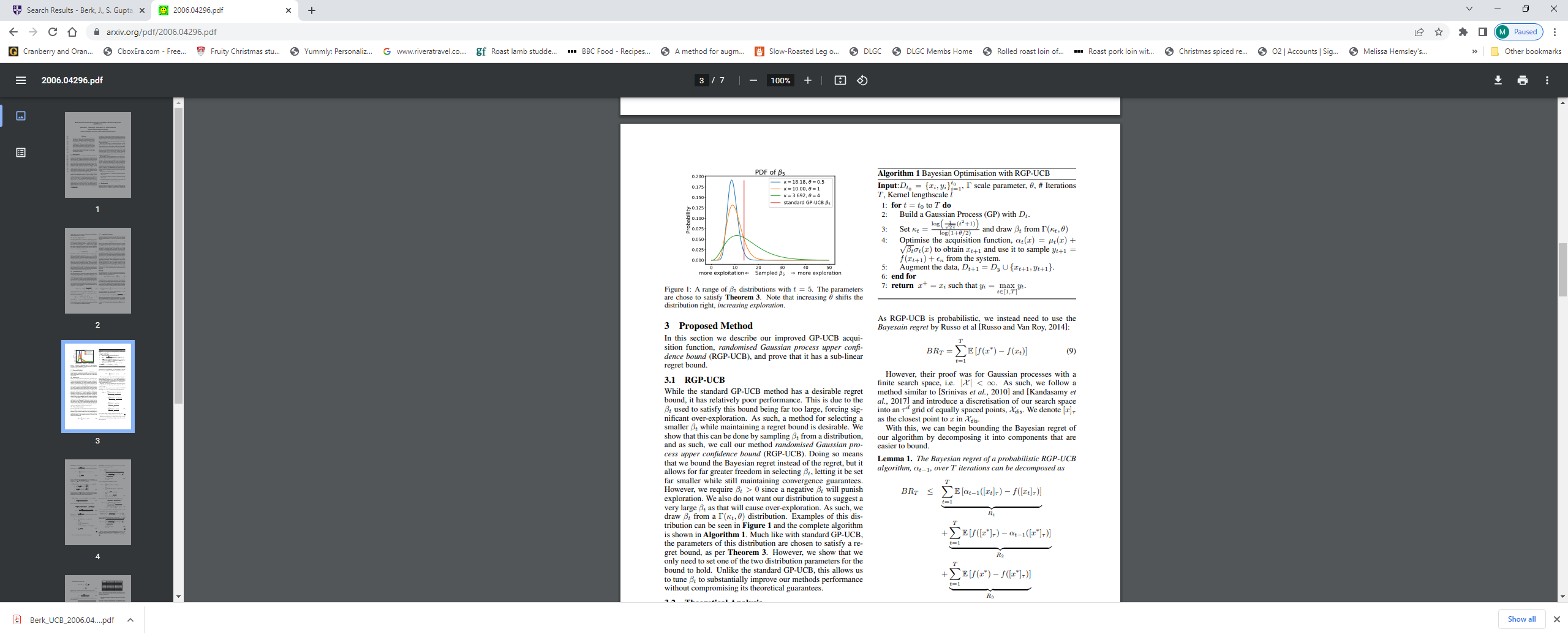


Figure 1: The RGB-UCB algorithm in pseudo-code. Note the hyper-parameter is resampled at each iteration. Reproduced from [xiii].

The significance of the RGP-UCB approach is that it provides an automatic method for hyper-parameter. Our experience is the algorithm performs well in many situations with a Gaussian process as the surrogate model. Consequently, it is highly recommended.

## 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 [[[15]](#endnote-16), [[16]](#endnote-17), [[17]](#endnote-18), [[18]](#endnote-19)]. 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 private GitHub repository:

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

# Package Architecture and the ***bayesOpt*** Class Description

The Bayesian Optimisation package architecture is presented in Figure 2. 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* [[[19]](#endnote-20)], 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.

The user need only understand the ***bayesOpt*** class to implement the entire process. The chosen surrogate model and acquisition function objects are created by the class constructor method, which creates an instance of the class, i.e., a ***bayesOpt*** object. In this first release, the only surrogate model supported is Gaussian process regression. Therefore, this initial release is restricted to applications with continuous inputs. However, the user may currently select either the EI or UCB acquisition function forms.

Table 1 presents a list of the accessible ***bayesOpt*** class properties and their definition. These data are displayable in the MATLAB command window. The ***bayesOpt*** class is designed to implement steps 3 through 4 of the basic algorithm presented in section 1. Thus, it utilises prior queries to train a surrogate model and subsequently optimise the acquisition function.

Table 1: List of **bayesOpt** properties and their definition

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| AcqObj | Acquisition function object | {ucb} or ei | SetAccess = protected |
| AcqFcn | Acquisition function type | string | SetAccess = protected  Dependent = true |
| HyperPar | Hyper-parameter value | double | SetAccess = protected  Dependent = true |
| Model | Surrogate model name | string | SetAccess = protected  Dependent = true |
| ModelObj | Surrogate model object | {gpr} | SetAccess = protected  Dependent = true |
| X | Current training input matrix. | double | SetAccess = protected  Dependent = true |
| Y | Corresponding training response data vector | double | SetAccess = protected  Dependent = true |
| Xnext | Next input configuration to query. | double | SetAccess = protected  Dependent = true |

Table 2: List of **bayesOpt** methods and their function

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| bayesOpt | Class constructor | Ordinary method |
| setTrainingData | Define the initial training data and train the surrogate model | Ordinary method |
| predict | Generate model prediction, , calculate the corresponding uncertainty measure, , and provide upper and lower 95% prediction intervals for | Ordinary method |
| setHyperPar | Set the hyperparameter for the acquisition function. Note this initial value may be overwritten by the hyperparameter selection algorithm implemented in the acquisition function object. | Ordinary method |
| acqFcnMaxTemplate | Optimise the acquisition function on the current training data. That is solve: | Ordinary method |
| addNewQuery | Add a new function query, to the training data and retrain the surrogate model. | Ordinary method |
| conDataCoding | Configure the data coding; that is, define the mapping: | Ordinary method |

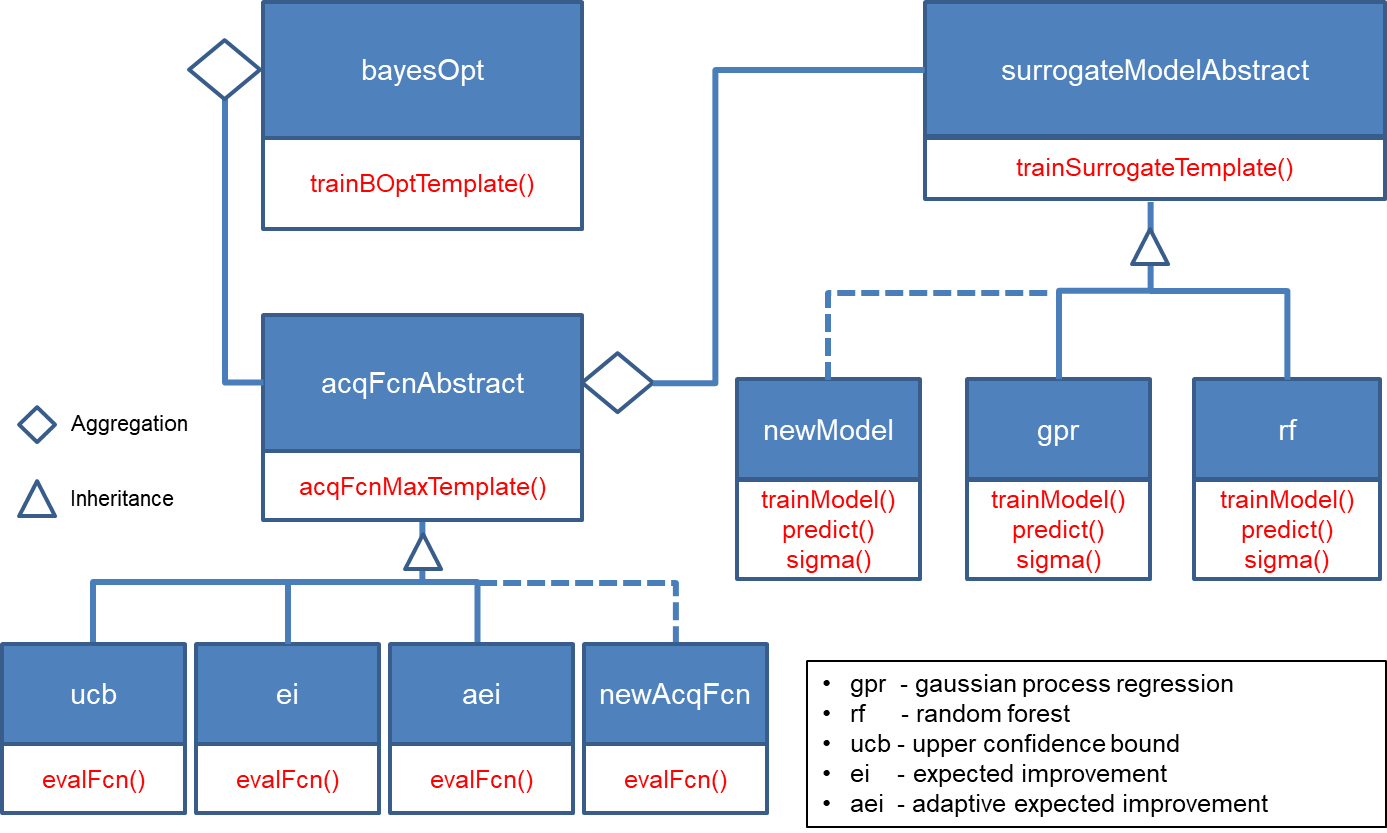


Figure 2: Bayesian Optimisation Package Architecture. The bayesOpt class aggregates the concrete surrogate model and acquisition function implementations. To extend the choice of surrogate models or acquisition functions the user need only add in new concrete implementations to the corresponding abstract interface.

## Creating and using a ***bayesOpt*** Object

As the ***bayesOpt*** class is the only class with which the user directly interacts with, in this section we detail the how to create an object and define the class method arguments necessary to implement the BO algorithm. To create a ***bayesOpt*** object utilise the following syntax:

**B = bayesOpt( Model, AcqFcn );**

We will use **B** to denote an instance of the ***bayesOpt*** class throughout. At this juncture, **Model** must be the string **“gpr”,** where gpr denotes Gaussian process regression, the only supported surrogate model at this time. Similarly, the **AcqFcn** argumentstring denotes the choice of acquisition function. It must be either **“ucb”** for upper confidence bound or **“ei”** for the expected improvement function. Use of the UCB function is strongly recommended as the hyperparameter can be determined automatically. In our experience, if using the EI approach, setting the hyperparameter in the interval usually affords good results. The default choices for the surrogate model are the Gaussian process regression surrogate model and the UCB acquisition function. To utilise the default choice, use:

**B = bayesOpt();**

To use the default surrogate model, but with the EI acquisition function, use the syntax:

**B = bayesOpt( [], “ei” );**

With the EI acquisition function, to set the hyperparameter use:

**B = B.setHyperPar( Value );**

Where **Value** is number in the interval . Next we may define the data coding. All fitting actions are carried out in coded units, which defines a standardised scale for any problem. This simplifies any subsequent tuning of any hyperparameters and permits the user to build on any previous experience. The coding is the linear vector mapping: , where and denote the respective lower and upper bounds for , and is the corresponding coded input data. The necessary syntax is:

**B = B.conDataCoding( L, U );**

we may now supply training data to the object using the command:

**B = B.setTrainingData( X, Y );**

Where the matrix **X** is the matrix of inputs, **X** , and **Y** the corresponding vector of objective function queries, **Y**. Adding training data to the ***bayesOpt*** object automatically trains the corresponding preferred surrogate model. At this point we may optimise the acquisition function using the command:

**B = B.acqFcnTemplate();**

This solves the problem using the **MATLAB optimisation toolbox** function **fmincon**. As this method utilises **fmincon**, the user may supply any optional arguments supported by **fmincon** to the method as ***name-value*** pairs. For example, to apply the bound to the single input , use:

**B = B.acqFcnTemplate( “lb”, 0, “ub”, 10 );**

For a complete list of options consult the associated MATLAB **fmincon** documentation by executing the following at the command line: **doc fmincon**

With reference to the MATLAB **fmincon** documentation, any field defined in the structure **PROBLEM** for command call **fmincon( PROBLEM )** is a possible name argument for the **acqFcnTemplate** method. This provides the user with complete control over the optimisation process. However, sensible defaults are set by the ***bayesOpt*** code. Note once the **acqFcnTemplate** method is executed the property **Xnew = B.Xnext** is updated. This is, the next point at which to evaluate the objective function or collect data as appropriate is now known. Once the objective function is queried at **Xnew**, to yield the corresponding **Ynew**, then we can add this new ordinate pair to the existing training data and simultaneously retrain the surrogate model using:

**B = B.addNewQuery( Xnext, Ynext );**

This sequence of commands represents steps 3 and 4 in the basic algorithm. In sections 5 and 6, we describe the individual classes which implement the acquisition function and surrogate model elements of the general algorithm.

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

The purpose of the surrogate model interface is to provide a consistent approach to accessing concrete implementations inheriting 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 3 and Table 4 respectively. Any future surrogate model implementations (child class) must implement these abstract members for the interface to function.

Property attributes either define the nature of class data or access to them. The “Constant” attribute implies the variable is a constant. The “SetAccess” attribute defines write access to the property. If “protected”, the value of the property can only be set in a child method. In contrast, all abstract method signatures are defined as ordinary methods. Consequently, these methods can be executed at the command line.

Table 3: Abstract properties defined by the **surrogateModel** class. A concrete implementation must be provided in the child class

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class** | **Attributes** |
| ModelType | Name of surrogate model | string | Constant |
| Cov | Kernel (covariance) matrix | double | SetAccess = protected |

Table 4: 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. The concrete property members represent general, or summary, data pertinent to all child classes. For example, the name of the response variable, or the names of the input variables. The ability to define both abstract and concrete members in an abstract interface class is a very useful feature and maximises code re-use. Similarly, the concrete methods implemented provide ***SetAccess*** to the corresponding concrete properties. For example, the ***setYname*** method permits the user to define the name of the response variable. Again, all these concrete members are ordinary methods and can thus be executed at the command line.

Table 5: 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 |
| Xlo | Lower bound for x-data (used for data coding) | double | SetAccess = protected |
| Xhi | Upper bound for x-data (used for data coding) | double | SetAccess = protected |
| Trained | Set to true if model has been fitted to training data | logical {false} | 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 6: 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 and response variables. | Ordinary method |
| setVarNames | Set the input variable names. | Ordinary method |
| setYname | Set the response variable name. | Ordinary method |
| conDataCoding | Configure the data coding:  The default limits are: | Ordinary method |

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

The ***gpr*** class is essentially a *wrapper* [xiii] around the MATLAB ***RegressionGP*** class provided in the MATLAB ***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 2, the ***gpr*** class inherits from the abstract ***surrogateModel*** class. Table 5 defines the concrete property members for the ***gpr*** class.

Table 7: List of **gpr** class properties

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| ModelType | Surrogate model name | string {“GP”} | Constant |
| Kernel | Covariance matrix kernel function. | kernels  {kernels(“ ARDsquaredExponential”)} | SetAccess = protected |
| PredMethod | Prediction calculation method | gprPredMethod  Algorithm for determining predictions from the model {exact} | SetAccess = protected |
| FitMethod | Representation of the Gaussian process regression model during model training. | gprFitMethod  Algorithm for representing the Gaussian process and active data set during model training {exact} | SetAccess = protected |
| LenScale | Vector of length scales per input variable. | double  Vector of regularisation constants associated with each dimension. A large (small) value denotes smooth (non-smooth) behaviour in a particular dimension. | Dependent = true |
| SigmaF | Process noise standard deviation | double  Estimate of the standard deviation associated with the Gaussian prior . | Dependent = true |
| ModelObj | The Gaussian process regression model. | RegressionGP  This is a class implemented in the **MATLAB statistics and machine learning toolbox**. | SetAccess = protected |

Table 8: List of **gpr** methods and their function

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| setKernel | Set the covariance matrix kernel function.  **G = G.setKernel( Name );**  The string **Name** may be any of the following:  “ARDsquaredExponential”  “ARDexponential”  “ARDmatern32”  “ARDmater52”  *Details of the various kernel types supported can be found in section 1.1.1 on page 3.* | Ordinary method |
| setFitMethod | Set the method of kernel representation and training data subset approach for training.  **G = G.setFitMethod( Method );**  The string Method may be any of the following:  “exact”  “sd”  “sr”  “fic”  The default fit method is “exact”. Details are provided in the MATLAB **RegressionGP** documentation. For large data sets “fic” is highly recommended. | Ordinary method |
| setPredMethod | Set the method of kernel representation and training data subset approach during training.  **G = G.setPredMethod( Method );**  The string Method may be any of the following:  “exact”  “sd”  “sr”  “fic”  “bcd”  The default fit method is “exact”. Details are provided in the MATLAB **RegressionGP** documentation. For large data sets “fic” or “bcd” is recommended. | Ordinary method |
| trainModel | Train the Gaussian process regression model. The default fit method is “exact”.  **G = G.trainModel();** | Ordinary method |

To instantiate a ***gpr*** object, use the following command syntax:

**G = gpr( X, Y );**

Arguments **X** and **Y** are defined in Table 5 on page 12. To train a model, execute the following at the command line.

**G = G.trainModel();**

The default fit method for the ***gpr*** class is “exact”, but any of the specialised methods for large data sets described in the MATLAB documentation for the ***RegressionGP*** class can be specified as ***name- value*** pairs. A brief introduction is provided in section 1.1.1.1 on page 4. Alternatively, again consult the MATLAB documentation for the ***RegressionGP*** class. This can be accessed via the command:

**doc RegressionGP**

The kernel type, fit and prediction methods can all be selected. Changing the fit and prediction methods is highly recommended for large data sets.

## 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***)

Analogous to the ***surrogateModel*** interface class, the purpose of the abstract ***acqFcn*** function class is to provide a clean, consistent and easily extendable interface for defining, configuring and evaluating concrete child implementations. Any child class must define concrete implementations of the abstract members listed in Table 8 and Table 9 respectively.

Table 9: 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 | Abstract  Constant |
| ModelObj | Aggregated surrogate model object. | gpr | SetAccess = protected |
| BestX | Best result present in the training data to date | double | SetAccess = protected |
| Data | Current training input data | double | SetAccess = protected  Dependent |
| Response | Current response training data | double | SetAccess = protected  Dependent |

Table 10: 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 |
| setBeta | Set the hyperparameter method.  **E = E.setBeta( Beta );** | Abstract  Ordinary method |
| setBestX | Update the BestX property  **obj = obj.setBestX( Xmax );** | Ordinary method |
| addFcnSample2Data | Add a new function query to the training data and retrain the surrogate model.  **obj = obj. addFcnSample2Data( Xnew, Ynew);** | 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. Like all acquisition functions in the ***bayesOpt*** package, the primary purpose of this class is to provide a concrete implementation for evaluating the specific acquisition function. The EI concept is described in more detail in section 1.2.1. Unlike the UCB approach, there is no automatic method for selecting implemented in the class. Consequently, the user must specify the value. In our experience, the default of 0.01 seems to work well in a wide variety of settings. Table 11 and Table 12 define the relevant class properties and methods respectively.

Table 11: List of **ei** class properties and their definition

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| FcnName | Name of acquisition function. | acqFcnType  {acqFcnType( “ei”) | Constant |
| Beta | Hyperparameter value. | double  {0.01} | SetAccess = protected |

Table 12: List of **ei** class methods and their function

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| evalFcn | Evaluates the EI acquisition function at the input locations specified.  **Fcn = E.evalFcn( X, Beta );**  Where **X** denotes the input data locations where **(X)** need be evaluated, and **Beta** is a hyperparameter value. The **Beta** argument need only be specified if it is desired to change the hyperparameter for a given BO iteration. This provides a convenient way of evaluating different choices for the hyperparameter if required. | Ordinary method |
| setBeta | Set the hyperparameter method.  **E = E.setBeta( Beta );**  Where Beta is the desired hyperparameter value. | Ordinary method |

To create an ***ei*** class object utilise the following syntax:

**E = ei( ModelObj, Beta );**

Where **ModelObj** must be a ***gpr*** object and **Beta** is the specified hyperparameter.

## The Adaptive Expected Improvement (***aei***) Class

Recently Xu *et al* [[[20]](#endnote-21)] published a hybrid BO algorithm, which included a novel adaptive expected improvement acquisition function algorithm.

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

Similarly, the UCB acquisition function class, ***ucb***, is designed primarily to facilitate calculation of at the input configurations specified. Consequently, the ***ucb*** class possesses relatively few accessible properties and methods, which are listed in Table 13 and Table 14 respectively.

Table 13: List of properties and their definitions for the **ucb** class

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Class {default}** | **Attributes** |
| FcnName | Name of acquisition function. | acqFcnType  {acqFcnType( “ucb”) | Constant |
| Beta | Hyperparameter value. | double  {0.02} | SetAccess = protected |
| Scale | Scale parameter for the Gamma distribution | double  {1} | SetAccess = protected |

Table 14: List of **ucb** class methods and their function

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Description** | **Attributes** |
| evalFcn | Evaluates the EI acquisition function at the input locations specified.  **Fcn = E.evalFcn( X, Beta );**  Where **X** denotes the input data locations where **(X)** need be evaluated, and **Beta** is a hyperparameter value. The **Beta** argument need only be specified if it is desired to change the hyperparameter for a given BO iteration. This provides a convenient way of evaluating different choices for the hyperparameter if required. | Ordinary method |
| setBeta | Set the hyperparameter method.  **E = E.setBeta( Beta );**  Where Beta is the desired hyperparameter value. | Ordinary method |
| setScale | Set the Gamma distribution scale parameter [[[21]](#endnote-22)]. | Ordinary method |

To create an ***ucb*** class object utilise the following syntax:

**U = ucb( ModelObj, Beta );**

Where **ModelObj** must be a ***gpr*** object and **Beta** is the specified hyperparameter.

# Worked Example and the “BayesOptExample” Live Script

To demonstrate the use of the package an interactive MATLAB live script has been developed, ***BayesOptExmple.mlx***. This demonstrates the maximisation of the function in the interval using Bayesian optimisation. The objective function is presented in Figure 3, along with the seven initial queries afforded by the initial experimental design. It is instructive to execute the live script a section at a time. As well as demonstrating the use of the ***bayesOpt*** class methods, the code blocks demonstrate code for useful processes such as, for example, how to generate model predictions, and the use of the **AcqObj** and **ModelObj** properties to provide data for plotting.

Chart, line chart, scatter chart

Description automatically generated

Figure 3: Plot of and the seven queried points available from the initial design

The first section of the live script generates a high-resolution data set for plotting the objective function and query the function at 7 initial points defined by an experimental design. These data represent the initial training data. The second section creates a ***bayesOpt*** object and subsequently assigns the training data using the **setTrainingData()** method. The ***bayesOpt*** object utilises the default settings of a Gaussian process regression model as the surrogate and the UCB acquisition function.

The code then creates Figure 4, which demonstrates the fit afforded by the surrogate. The solid blue line is the fit afforded by . Clearly, based on the initial training data, is a relatively poor facsimile for the true object function; especially in the region of where the global maximum is located. However, in this region, the uncertainty, as measured by the difference between the lower (LPI) and upper (UPI) 95 % prediction intervals, is very large. Consequently, this represents an area for potential ***exploration***.

Chart

Description automatically generated

Figure 4: Initial fit afforded by the Gaussian process regression surrogate model and upper and lower 95 [%] prediction intervals

Section 3 optimises the UCB acquisition function using the **acqFcnMaxTemplate()** method. The function , presented in Figure 5 is also plotted and the optimal value highlighted by a blue square. Note, the value of the hyperparameter, , is relatively small at 0.018. Consequently, the next point to query is located quite close to the maximum value observed in the original training date. That is, the result of the first iteration is more exploitive than explorative. Due to the nature of the sampling process the -value will change every time the live script is executed.

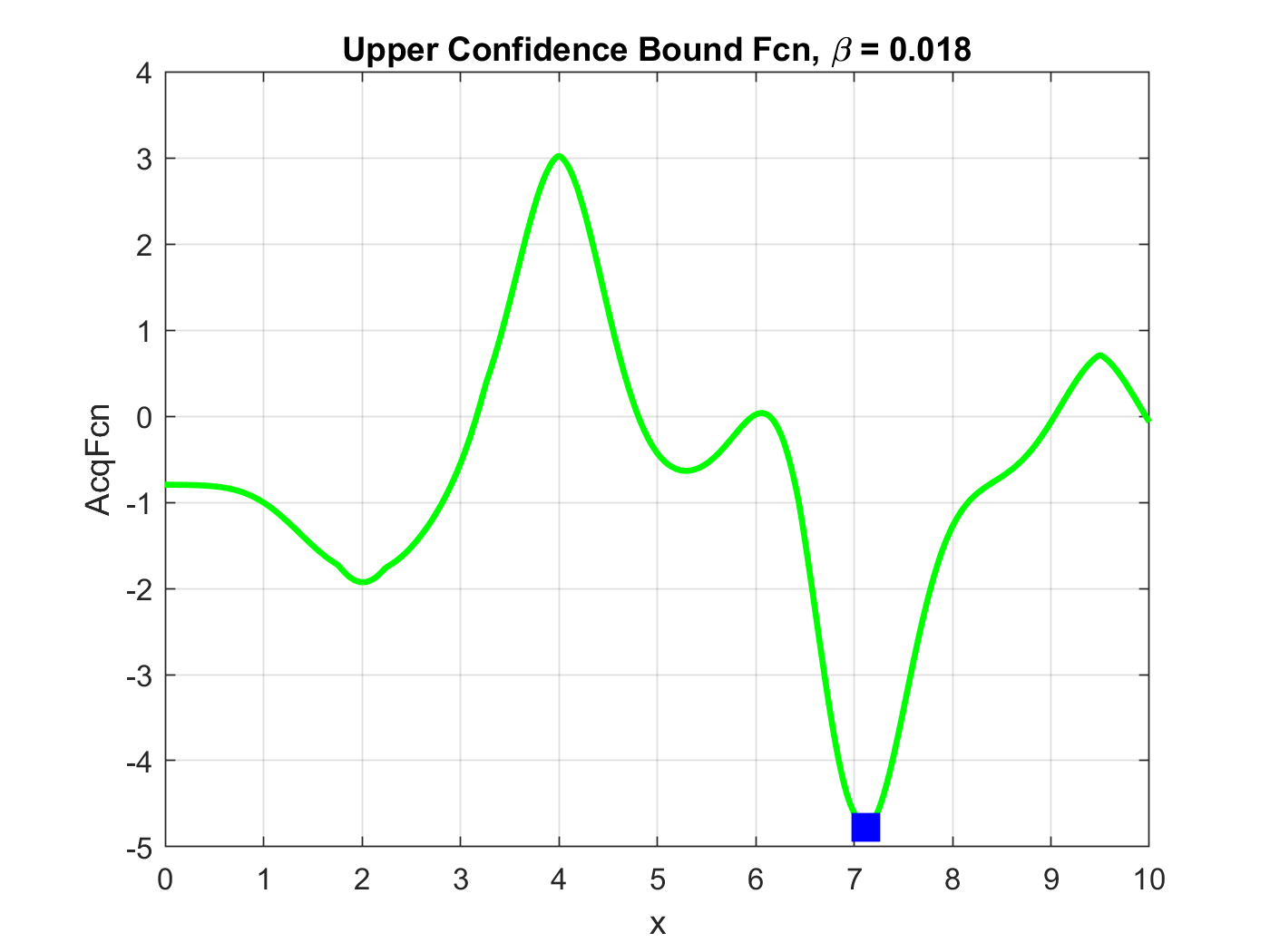


Figure 5: Plot of , with hyperparameter . The blue square at denotes the optimal value and is the next point to query.

In section 4, we augment the current training data with the point using the **addNewQuerry()** method. This method also retrains on the extended training set . The live script also creates Figure 6, which demonstrates the influence of the additional query on the surrogate model. After just one iteration, the predictive performance of the Gaussian process regression model is vastly improved. This is demonstrated not only by the improved prediction afforded by , but by the considerably reduced uncertainty – the LPI and UPI are now much tighter about the prediction in the vicinity of the global maximum.

A picture containing chart

Description automatically generated

Figure 6: Influence of on , which now affords a much better representation of the true objective function.

Section 5.0 performs a second iteration of the BO algorithm. The results are presented in Figure 7. Note at this iteration, the value of the hyperparameter is considerably larger – with . Consequently, more credence is given to exploration and the solution to the acquisition function optimisation problem is located much further away at , compared to from the first iteration. Rerunning section 5.0 increments the algorithm by an additional step. Consider Figure 8, which demonstrates the rapid convergence behaviour of the algorithm in this example.

Chart, line chart

Description automatically generated

Figure 7: Results of iteration 2 of the BO algorithm. The best result (denoted by the blue square) at is now very close to the true maximum.

Chart, line chart

Description automatically generated

Figure 8: Convergence behaviour of the BO algorithm

# 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 is defined 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.

Diagram

Description automatically generated

Figure 9: Schematic of the OOP Strategy Pattern

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 gaussian process regression is can accept both continuous and discrete inputs. Based on random decision trees, and are necessarily computed from sample statistics taken across all decision trees in the random forest.

Finally, the current release does not support optimisation when the number of decision variables is very high. Typically, this release supports scenarios where the number of inputs is in the range 30-50. In cases with distributed parameters, e.g., a system described by partial differential equations, the number of estimable terms may be very large, several hundreds or even thousands. To mitigate this in recent years, several large-scale versions of the BO algorithm have appeared in the literature. All of these are based on projecting the inputs, , into a lower dimensional space, say. As , and , minimisation of the acquisition cost function is much more efficient due to the reduction in parameters. To return to the full coordinate system required for evaluation of the objective function, it is also necessary to learn the inverse mapping . This increases the numerical complexity.

# Glossary

|  |  |
| --- | --- |
| Symbol/Abbreviation | Definition |
|  | Feasible set of input configurations |
|  | Acquisition function |
|  | Surrogate model prediction at point |
|  | Uncertainty (standard deviation) 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 |
| RGP-UCB | Randomised Gaussian Process Upper Confidence Bound |
| SD | Subset of data |
|  | Current training data set |
| UCB | Upper confidence bound |
|  | Input configuration vector: |
|  | Acquisition function hyperparameter |
|  | Observations of contaminated with random noise |
|  | Set of real numbers |

# 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-2)
2. Archetti, F., and A. Candelieri. 2019. Bayesian Optimization and Data Science. *Springer* [↑](#endnote-ref-3)
3. Frazier, P. I. 2018. “Bayesian Optimization”. In Recent Advances in Optimization and Modeling of Contemporary Problems, 255–278. INFORMS. [↑](#endnote-ref-4)
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-5)
5. Glover, F., and M. Samorani. 2019. “Intensification, Diversification and Learning in Metaheuristic Optimization”. Journal of Heuristics 25(4):517–520. [↑](#endnote-ref-6)
6. Gill, P. E., Murray, W., Wright, M. H., Practical Optimization, *Academic Press*, First Edition, 1981. [↑](#endnote-ref-7)
7. Cui, J., and Yang, B., *Journal of Software*, 2018, Vol 29, pp 3068-3090. [↑](#endnote-ref-8)
8. Rasmussen, C. E., Williams, C. K. I., Gaussian Processes for Machine Learning, *The MIT Press*, 2006. [↑](#endnote-ref-9)
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-10)
10. Archetti, F., and A. Candelieri. 2019. Bayesian Optimization and Data Science. *Springer*. [↑](#endnote-ref-11)
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-12)
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-13)
13. Berk, J., S. Gupta, S. Rana, and S. Venkatesh. 2020. “Randomised Gaussian Process Upper Confidence Bound for Bayesian Optimisation”. arXiv preprint arXiv:2006.04296 [↑](#endnote-ref-14)
14. Srinivas, N., Krause, A., Kakade, S. M., Seeger, M. W., Information-Theoretic Regret Bounds for Gaussian Process Optimization in the Bandit Setting, *IEEE Transactions on Information Theory*, 2012, 58(5), pp 3250-3265. [↑](#endnote-ref-15)
15. Cary, M., A model Based Engine Calibration Methodology for a Port Fuel Injection, Spark-Ignition Engine, *Ph.D. Thesis*, Bradford University, 2003. [↑](#endnote-ref-16)
16. Stein, M., Large Sample, Properties of Simulations Using Latin Hypercube Sampling, *Technometrics*, 1987, 29, pp 143-151. [↑](#endnote-ref-17)
17. 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-18)
18. 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-19)
19. 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-20)
20. Xu, Z., Guo, Y., Saleh, H., Efficient hybrid Bayesian optimization algorithm with adaptive expected improvement acquisition function, *Engineering Optimization*, 2021, 53;10, pp 1786-1804. DOI: 10.1080/0305215X.2020.1826467. [↑](#endnote-ref-21)
21. Devore, Jay. *Probability and Statistics for Engineering and the Sciences / Jay L. Devore*. 4th ed. Wadsworth Publishing, 1995. [↑](#endnote-ref-22)