# Bayesian Optimisation Package (***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)].

Contents

[Bayesian Optimisation Package (***bayesOpt***) User Notes 1](#_Toc110865467)

[1 Preamble 1](#_Toc110865468)

[2 Note on Abstract Classes and Composition 2](#_Toc110865469)

[2.1 Abstract Interfaces 2](#_Toc110865470)

[2.2 Composition and Aggregation 3](#_Toc110865471)

[3 Package Architecture 3](#_Toc110865472)

[4 The Surrogate Model Interface (***surrogateModel***) 4](#_Toc110865473)

[4.1 The Gaussian Process Regression (***gpr***) Class 4](#_Toc110865474)

[4.2 The Random Forest (***rf***) Class 4](#_Toc110865475)

[5 The Acquisition Function Interface (***acqFcn***) 4](#_Toc110865476)

[5.1 The Expected Improvement (***ei***) Class 4](#_Toc110865477)

[5.2 The Upper Confidence Bound (***ucb***) Class 5](#_Toc110865478)

[6 Glossary 6](#_Toc110865479)

[7 References 7](#_Toc110865480)

# 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. In essence, 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 . 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 we may find a better solution over what is already known.

We will utilise the following notation to distinguish between the noise-free, *e.g.,* samples from a complex physical simulation code, and the measurement case, in which the data will be contaminated with random noise. In the noise-free case we will use to denote function queries. However, in the case where data is contaminated by random noise, we will use , where 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 or as appropriate. However, since the class was originally conceived for noisy data, we will utilise to denote the training data throughout.

Further, we utilise to represent the surrogate model. 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. Finally, the acquisition function will be written . We are now able to define the generic BO-algorithm as:

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:

# 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-a*** or ***is-an*** 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.

# 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 [[[8]](#endnote-8)], implying that the specific concrete implementation is selected at run-time.

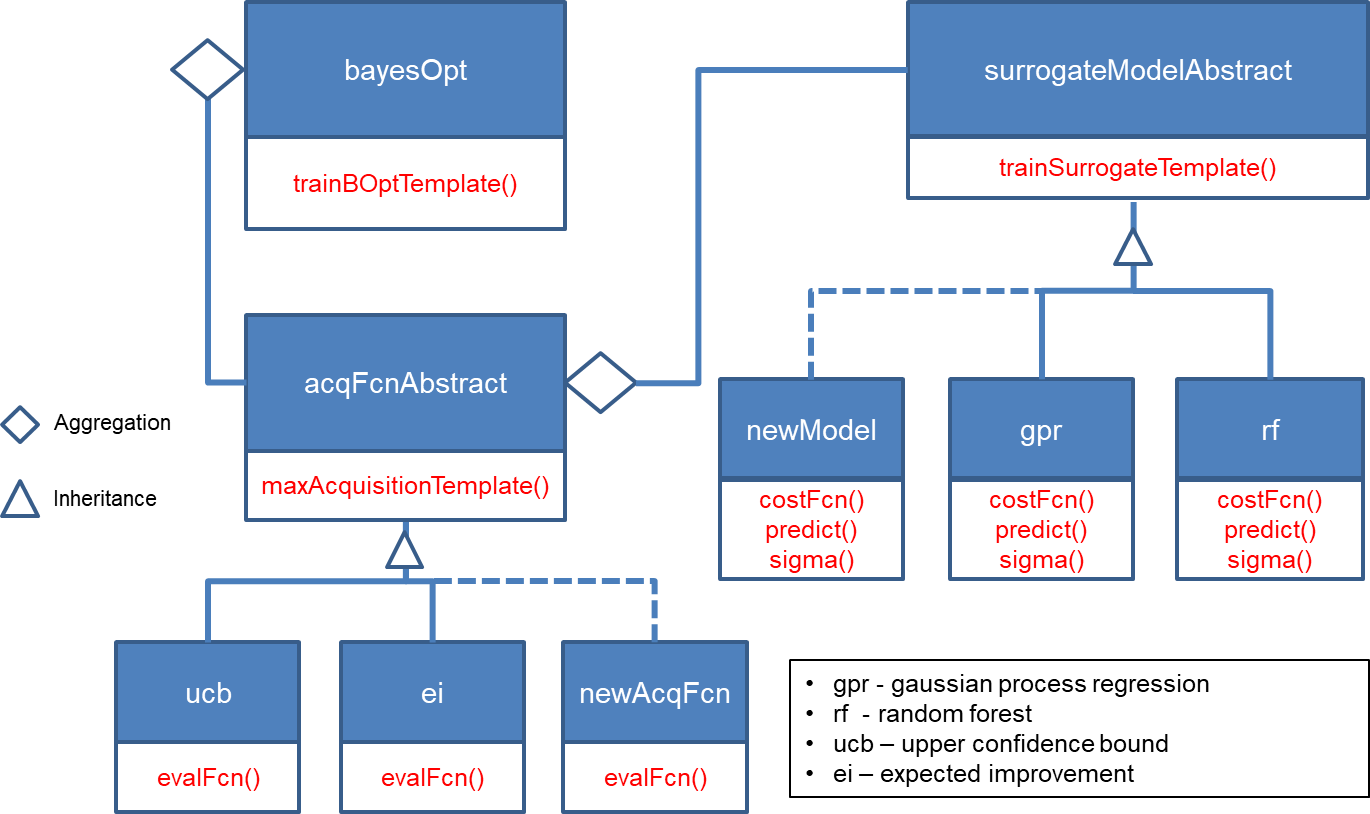


Figure : Bayesian Optimisation Package Architecture. The bayesOpt class aggregates the concrete surrogate model and acquisition function implementations.

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

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

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

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

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

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

# Glossary

|  |  |
| --- | --- |
| Symbol/Abbreviation | Definition |
| BO | Bayesian optimisation |
|  |  |
| OOP | Object-oriented programming |

# 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. 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-8)