pybit:
python Bayesian inference toolbox

Joffrey Coheur, Maarten Arnst

October 7, 2019

Université de Liège, Aerospace and Mechanical Engineering, Allée de la Découverte 9, 4000 Liège, Belgium

1 Introduction

We described here the general structure and some practical implementations of pybit, the solver for Bayesian inference that is developed within this work. The python Bayesian inference toolbox, or pybit, is mainly constructed for Bayesian inference but can also be used for sampling from known probability distribution functions. It is therefore developed in a general framework. Future applications should combine sampling from posterior distributions in pybit and propagation of these samples using methods such as polynomial chaos expansion (PCE), or other propagation methods.

Pybit to not intend to compete with current "industrial level" toolbox for uncertainty quantification, such as Dakota, UQLab or ChaosPy, but rather intends to be a research code where state-of-the-art algorithms can be implemented within python.

2 Input/Output

Every case contains an input file that the code will first read and which specifies what the code will run. For Bayesian inference, one needs to specify the experimental data, the model used with its parameter and the method for the inference. For the propagation, (future development), we only need to specify the model with its variable and the method.

3 General description

The code is based on the definition of a `ProbabilityDistribution` class. For known distribution, it is defined by an analytical function, the probability distribution function, and we can access to its value by calling the `compute_value` method[1] at a given sample point $\mathbf{p}$. For Bayesian posteriors, the definition of the likelihood and the prior distributions need to be first specified. In this case the `compute_value` method returns directly the product of the likelihood times the prior distribution.

The Bayesian posterior definition needs to be complemented by the definition of the data and the physical model, for which we have two separated python classes, `Data` and `Model` respectively. The `Data` class contains the abscissa and ordinates (the observations) of the problem. We can specify as much data sets as we want. Those are specified in the input file using the list `Data : [ dataSet1 , dataSet2 , etc, ]`. The different data sets are then concatenated in a single one dimensional `numpy` array in memory. The `Model` class contains the information about the physical model (and not the statistical model). There must be as much model delimiters in the input file as data sets, even if the model is the same to represent all data sets. The `Model` class can incorporate change of variable (used for example for the Ito-SDE algorithm of for reparametrizing the model) and is optional (default is $\mathbf{p} = \tilde{\mathbf{p}}$). There is not pre-implemented models within pybit and the user must specified its model in a separated python file following the `Model` class structure. The `fun_x` method is mandatory and specifies the function that is used to reproduce the observations. This function can be defined directly within the python file or can be provided from an other program (only other python modules so far; **future extension to external program such as Argo or PATO**) in which case the `fun_x` method needs to point towards the execution of the program. In the case where the program is external, the model will write at every iterations a temporary input file (`write_tmp_input_file`) that automatically updates the inputs of the model according to the Markov chain.

[1] A method of a class is a function available for objects of that class.

4 Sampling

Once the probability distribution function to sample has been specified, either Bayesian or not, one need to decide the sampling algorithm that is going to be used. All Metropolis-Hastings mentioned in this report are implemented and are based on a general class `MetropolisHastings`. The Ito-SDE algorithm is also implemented (**so far is herits from `MetropolisHastings` although is not a MH method ; but we should create a general class `IterativeAlgorithms`, on which `MetropolisHastings` and Ito-SDE are based**).

4.1 Metropolis-Hastings algorithms

The `MetropolisHastings` is based on the definition of three main methods that we distinguish: `compute_new_val`, `compute_acceptance_ratio` and `accept_reject`. They all appear sequentially in the `run_algorithm`, which is the main function to run when using Metroplis-Hastings algorithm. For the adaptive Metropolis-Hastings, a fourth method `adapt_covariance` is defined, while for the delayed-rejection Metropolis-Hastings, the `accept_reject` step is redefined by adding the extra delayed-rejection step. The delayed-rejection adaptive algorithm is just a combination of the two previous ones.

4.2 Note on the `compute_acceptance_ratio` step

Because we perform Bayesian inference where the estimation can be sometimes quite bad, the value of the likelihood function can be very low. This can happens at the beginning of a Markov chain from a sample that estimates badly the model or when we don't have any good initial guess. The value is thus close to the epsilon machine and can be erroneously rounded to zero. The acceptance ratio can be estimated wrong and we need to avoid computation of $0/0$. We therefore compute the logarithm of the acceptance ratio to avoid these problems leading to

$$\log(r) = \log(\pi(\mathbf{p}^*|\mathbf{d})) + \log(J(\cdot|\mathbf{p}_{i-1})) - \log(\pi(\mathbf{p}_{i-1}|\mathbf{d})) - \log(J(\mathbf{p}_{i-1}|\cdot)) \tag{1}$$

or by specifying the target distribution in terms of Bayesian posterior

$$\begin{aligned}\log(r) = &\log(\pi(\mathbf{d}|\mathbf{p}^*)) + \log(\pi_0(\mathbf{p}^*)) + \log(J(\cdot|\mathbf{p}_{i-1})) \\ &- \log(\pi(\mathbf{d}|\mathbf{p}_{i-1})) - \log(\pi_0(\mathbf{p}_{i-1})) - \log(J(\mathbf{p}_{i-1}|\cdot))\end{aligned} \tag{2}$$

Thus, for Gaussian likelihood, the log-likelihood function is the L_2 distance between the data and the model. In pybit, the log value of a distribution can be accessed through the `compute_log_value` method that is implemented within every `ProbabilityDistribution` class. The final value of the acceptance ratio r after the sum Eq. 2 is performed.

4.3 Ito-SDE algorithm

For the Ito-SDE algorithm, we do not perform any more a random walk loop and the `run_algorithm` method is different than for the MetropolisHastings classes.