pybitup:
PYthon Bayesian Inference Toolbox and Uncertainty Propagation

Joffrey Coheur
Developpers: Joffrey Coheur, Martin Lacroix

Last update: October 16, 2020

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

1 Introduction

There exists many computational software for Bayesian inference and uncertainty propagation. For Bayesian inference, the BUGS (Bayesian inference Using Gibbs Sampling) program was first developed in the early 1980s (see [1] for a review and history of the program) with the aime of providing a statistical toolbox for sampling from the posterior distribution using a convenient programming language. A related package of BUGS is JAGS (Just Another Gibbs Sampling) which is more UNIX-oriented. More recently was released the sofwate Stan ([2]) written in `C++` which takes the advantage of the development of more efficient sampling algorithms for high dimensions and computational efficiency. The main simulation algorithms feature the Hamiltonian Monte Carlo with the most advanced improvements. The python toolbox PyMC ([3]) features random-walk metropolis and adaptive improvements and several versions of slice samplers, and more advanced methods in PyMC3 [4]. Worth mentionning is the work of [5] for the development of hIPPYlib.

The first objective of `pybitup` is to implement a toolbox that is readily usable and that is suitable to many engineering problems. Problems that are sought are the calibration of time observations to a computational model representing the complex physics of the process and the propagation of the calibrated parameters. Several classical algorithms to solve those problem are implemented in the code (adaptive random-walk Metropolis-Hastings, polynomial chaos methods). The second objective is to allow the users to have the flexibility to implement easily state-of-the-art algorithms for research purposes, both for Bayesian inference and uncertainty propagation. Examples are Hamiltonian Monte Carlo, Ito-SDE (see later) for sampling, or quadrature rules for correlated parameters. The use of `python` as programming language is therefore very suitable. Moreover, more and more programmers are developing their routine tailored to their problem in `python`, making the use of a python software convenient and allowing to use this routine as a black-box without much effort.

Finally, software for calibration are usually developed apart from software for propagation. Results from the model fitting procedure using Bayesian inference are not reused for uncertainty propagation. Uncertainty propagation in the engineering community is usually build based on standard parameter distributions (uniform, Gaussian, gamma) and for which it is easy to build a surrogate, but most often that does not represents the real parameters distribution. One of the challenge is the difficulty to build surrogate models from unknown and correlated distributions ([6]). Pybitup allows to use the samples from the Markov chain resulting from the calibration to be propagated through a computational model and performing sensitivity analysis. Thus performing uncertainty quantification from the inference of parameter distribution to their uncertainty propagation and sensitivity analysis. There are a few examples allowing to do that. Dakota [7] is an open-source software written in C++ developed by Sandia National Laboratories and is developed in a more general framework than uncertainty quantification. It was initially designed for optimization but was then extended to uncertainty propagation and Bayesian inference. In `Matlab`), there is `UQLab`, but to the authors' knowledge, none in `python`. Propagation can be performed directly from a priori generated samples.

The development of a simulation code for model fitting, Bayesian inference and uncertainty propagation is further motivated by the following arguments:

- Research: allow advanced users to have an easy and modulable way of implementing new algorithms in an oriented-object framework using popular and accessible computational language (python). Python language is not a language dedicated for statistical purposes, and model implemented within python can be made more complicated and more general. Full deterministic software can be developed.

- Applications: flexibility to implemented new model just by the knowledge of python. During my thesis, deterministic numerical solver was developed for pyrolysis and later coupled to pybitup for performing Bayesian inference and propagation. Applications in inference for heat capacity simulations was also performed.

1.1 Content of the document

We described here the general structure and some practical implementations of pybitup, the solver for Bayesian inference that is developed within this work. The python Bayesian inference toolbox, or pybitup, 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 pybitup and propagation of these samples using methods such as polynomial chaos expansion (PCE), or other propagation methods.

pybitup 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 Installation

`Pybitup` requires a few packages to be installed on the system to be able to run. It was developed on `Python` version 3.7.0. Packages can be installed using pip, the package installer for `Python` and is already installed for `Python` version >= 3.4.0. Packages can be installed using the command `pip install PackageName`. Here is the list of packages used by the software.

- `numpy`: fundamental scientific programming package that provides the tool for manipulating data arrays and several useful functions for data analysis.
- `matplotlib`: library for producing graphs.
- `SciPy`: library containing algorithms for mathematics (integration methods, linear algebra), statistics (distributions).
- `pandas`: provides the data manipulation tool, such as reading and writing csv files for input and output communication.
- `pickle`: package for manipulating large data sets efficiently.
- `json`: is used for the input file management (reading the inputs) and having a structured input file.
- `jsmin`: gives the possibility to add comment to .json input files.
- `seaborn`: (optional) used for heatmap (that was used for graphical representation of correlation matrix)
- `mpi4py`: (optional) provides bindings of the Message Passing Interface (MPI) standard for the Python programming language, allowing any `Python` program to exploit multiple processors. Requires a working `MPI` implementation to be install on the system (e.g. microsoft `MPI`)

3 Tutorial: spring model

We consider the example of the spring model described in [8]. The displacement z (adimensional) as a function of time of the spring without any external excitation is provided by the equations

$$\begin{cases} \ddot{z} + C\dot{z} + Kz = 0 \\ z(0) = 2, \dot{z} = -C \end{cases} \tag{1}$$

with $C = c/m$, $K = k/m$, $m \geq 0$ (kg) is the mass, $c \geq 0$ (N·s m^{-1}) is the damping coefficient and $k \geq 0$ (N m^{-1}) is the stiffness coefficient. For $C^2 - 4K < 0$, the solution is

$$z(t) = 2\exp\left(-\frac{Ct}{2}\right)\cos\left(t\sqrt{K - \frac{C^2}{4}}\right) \tag{2}$$

For this problem, synthetic data z_i are generated for $0 \leq i \leq n$ with $n = 51$ the number of points in the time interval $[0, 5]$ s from a random Gaussian distribution $\epsilon_i \sim N(0, \sigma_0^2)$ with $z_i = z(t_i, p_0) + \epsilon_i$ where $p_0 = [K_0, C_0]$ the nominal parameter values with $K_0 = 20.5$, $C_0 = 1.5$ and $\sigma_0 = 0.1$ (as will be seen later, data will be generated also with uncertainty on the parameters $\sigma_p = [0.1, 1]$).

3.1 Model file

The model, the parameters and its solution are specified in the `spring_model.py` file that we describe below.

```python
import numpy as np
from pybitup import bayesian_inference as bi

class SpringModel(bi.Model):
    """ Class for the spring model """

    def __init__(self, x=[], param=[], name=""):
        # Initialize parent object ModelInference
        bi.Model.__init__(self, name=name)

    def set_param_values(self):
        # Parameters
        self.C = self.param[0]
        self.K = self.param[1]

        # Variable
        self.time = self.x

    def fun_x(self):
        # Get the parameters
        self.set_param_values()

        # Return the function evaluation
        return 2 * np.exp(-self.C * self.time / 2) * np.cos(np.
            sqrt(self.K - self.C**2 / 4) * self.time)
```

Exponential, cosine and square root functions are imported from the numpy package. The bayesian_inference.py file, where the Model class is defined, needs to be imported from pybitup. Our model for the spring that we call here SpringModel is based on the general Model class and inherit from two (aside from __init__) essential methods, namely set_param_values() and fun_x(). On the last line is the solution from Eq. 2 of the spring model. The parameters C and K are assigned in the method set_param_values(). Because the values of C and K will change during the calibration as we will see later, this method is run every time the model is evaluated.

3.2 Input files

Now let's have a look at the general input file, that can be identified by the .json extension. We first have a look at the first keyword Sampling which contains the inputs for the sampling part. The minimal syntax is the following:

```
"Sampling": {
    "BayesianPosterior" : {
        "Data": [],
        "Model": [],
        "Prior": {},
        "Likelihood": {}
    },
    "Algorithm": {}
},
```

The BayesianPosterior keywords means that we will sample from a posterior distribution computed from Bayes' formula. The Algorithm section will define how we will sample from it.

Let's first have a look at the BayesianPosterior keyword. Mandatory keywords for BayesianPosterior are Data, Model, Prior and Likelihood. The experimental data are first defined under the Data keyword.

```
"Data": [
    {
        "Type": "ReadFromFile", // ReadFromFile,
           GenerateSynthetic
        "FileName": "spring_model_data",
        "xField": ["time"],
        "yField": ["d"],
        "sigmaField": ["std_d"],
        "n_runs": 1
    }
],
```

The data are read from files with general name spring_model_data. The n_runs keyword denotes the number of experimental data files that we have, which is only one here. Thus, the input file is named spring_model_data_0.csv and must appear in the case directory. Below is the structure of the dataset of the spring_model_data_0.csv file and for compactness, we only show here only the 10 first lines.

```
,time,d,std_d
0,0.0,1.9320285551921579,0.1
1,0.1,1.7158980306494984,0.1
```

```
2,0.2,1.0116134181643508,0.1
3,0.30000000000000004,0.43030863754766857,0.1
4,0.4,-0.1967505502959701,0.1
5,0.5,-0.8165563476714192,0.1
6,0.6000000000000001,-1.179212035121202,0.1
7,0.7000000000000001,-1.1151949721017202,0.1
8,0.8,-0.9116380936755248,0.1
```

The first column are the indices of the data. Following columns are the fields whose names appear in the input file, namely the time, the displacement of the spring `d` and the experimental standard deviation on `d`, `std_d`. The entries inside the input file and the data file must corresponds. In `pybitup`, we provide a function that allows to generate synthetic data. We just keep in mind here that the data for the spring model encoded in the `.csv` file were generated synthetically using the `generateDataFile.py` file and we will see later how to generate them.

Going back to the input file, the `Model` keyword specifies all the parameters of the model, their nominal values and if the model will be parameterized during the calibration, which is not the case here.

```
"Model": [
    {
        "param_names": ["C", "K"],
        "param_values": [1.5, 20.5],
        "parametrization" : "no"
    }
],
```

There are only two parameters here that are C and K. The nominal values of the model parameters will be replaced if those parameters are considered to be unknown in the calibration process.

Next, `Prior` defines the prior distribution.

```
"Prior": {
    "Distribution": "Mixture",
    "Param": {
        "C": {"initial_val" : 1.5,    "prior_name" : "Uniform",
           "prior_param" : [0.0, 100]},
        "K": {"initial_val" : 20.5,   "prior_name" : "Uniform",
           "prior_param" : [0.0, 100]}
    }
},
```

The parameters that are unknown in the calibration process appear under the `Param` keyword. We consider the two unknown parameters to be $p = [C, K]$. An initial value must be specified to start the calibration process. The `Mixture` distribution means that we consider the product distribution of the two marginal distributions assumed to be independent. The marginal distributions are supposed to be uniform pdf with support [0, 100] each.

The `Likelihood` keywords defines the likelihood function.

```
"Likelihood": {
    "function": "Gaussian",
    "distance": "L2"
}
```

It is assumed to be a `Gaussian` function with the weighted `L2` distance between the model and the data in the argument of the exponential term. So far, there is no other option for the likelihood (will be modified soon).

The Bayesian posterior distribution will be sampled using a Metropolis-Hastings algorithm that is specified under the `Algorithm` keyword.

```
"Algorithm": {
    "name": "AMH",
    "AMH": {
        "starting_it": 1e2,
        "updating_it": 1e1,
        "eps_v": 0.0
    },
    "n_iterations": 1e4,
    "proposal": {
        "name": "Gaussian",
        "covariance": {
            "type": "diag",
            "value": [0.0345, 0.7071]
        }
    }
}
```

The algorithm selected here is the adaptive random-walk Metropolis-Hastings (AMH) that will adapt the covariance matrix of the proposal during the iterations. In particular, the AMH algorithm requires additional parameters that are specified under the `AMH` keyword, namely the number of iterations at which the adaptation starts (`starting_it`), the frequency at which we update the covariance (`updating_it`) and a last parameter that controls the covariance adaptation (`eps_v`) which is set here to zero. The algorithm will perform during 10^4 iterations. Finally, we need to defined the proposal covariance matrix. Here, it is a Gaussian function with a diagonal covariance matrix with values $[0.0345, 0.7071]$ on the main diagonal. The covariance matrix is a tuning parameter that is not known a priori and may require several trial-and-error tests before finding the adequate covariance. Having an initial covariance matrix that is diagonal is is easier to set and can be guessed based on the typical scales of the problem that can be known by performing a local sensitivity analysis. Note that because here we selected the AMH algorithm, this covariance matrix will be modified with the iterations in the present case and will most probably not remain diagonal.

3.3 Run file

The file that needs to be run by python to perform the calibration is named `run.py`. It gathers the information from the input file, the model and it is where we select what function from `pybitup` we want to perform. In this case, in this the `Sampling` of the posterior distribution. We need first to import the `spring_model` file as well as `pybitup`.

```
import spring_model
import pybitup

case_name = "spring_model"
```

```
input_file_name = "{}.json".format(case_name)

# Define the model
my_spring_model = {}
my_spring_model[case_name] = spring_model.SpringModel(name=
    case_name)

# Sample
post_dist = pybitup.solve_problem.Sampling(input_file_name)
post_dist.sample(my_spring_model)
post_dist.__del__()

# Post process
pybitup.post_process.post_process_data(input_file_name)
```

The model in `pybitup` is specified in a list where the models we want to run are provided. The only model here in the `SpringModel` from the `spring_model.py` file.

All the main functions of `pybitup` (sampling, propagation, sensitivity analysis) are provided in the `solve_problem` file. We thus create an object `post_dist` for the sampling and provide as input the `.json` file. Next, we use the method `.sample` and provide the model list `my_spring_model` as input. At the end of the sampling, the output file are written in an `/output` folder and we delete the object `post_dist`. The last line is for the post process that will be described later.

4 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.

5 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

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

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 pybitup 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.

6 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**).

6.1 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.

6.2 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

$$\log(r) = \log(\pi(\mathbf{p}^*|\mathbf{d})) + \log(J(\cdot|\mathbf{p}_{i-1})) - \log(\pi(\mathbf{p}_{i-1}|\mathbf{d})) - \log(J(\mathbf{p}_{i-1}|\cdot)) \tag{3}$$

or by specifying the target distribution in terms of Bayesian posterior

$$\begin{aligned}\log(r) = \log(\pi(\mathbf{d}|\mathbf{p}^*)) + \log(\pi_0(\mathbf{p}^*)) + \log(J(\cdot|\mathbf{p}_{i-1})) \\ - \log(\pi(\mathbf{d}|\mathbf{p}_{i-1})) - \log(\pi_0(\mathbf{p}_{i-1})) - \log(J(\mathbf{p}_{i-1}|\cdot))\end{aligned} \tag{4}$$

Thus, for Gaussian likelihood, the log-likelihood function is the L_2 distance between the data and the model. In pybitup, 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. 4 is performed.

6.3 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.

7 Uncertainty propagation

8 Sensitivity analysis

References

[1] D. Lunn, D. Spiegelhalter, A. Thomas, N. Best, The BUGS project: Evolution, critique and future directions, Statistics in Medicine 28 (25) (2009) 3049–3067. doi:10.1002/sim.3680.

[2] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, A. Riddell, Stan: A probabilistic programming language, Journal of Statistical Software 76 (1) (2017). doi:10.18637/jss.v076.i01.

[3] A. Patil, D. Huard, C. Fonnesbeck, PyMC: Bayesian stochastic modelling inPython, Journal of Statistical Software 35 (4) (2010). doi:10.18637/jss.v035.i04.

[4] J. Salvatier, T. V. Wiecki, C. Fonnesbeck, Probabilistic programming in python using PyMC3, PeerJ Computer Science 2 (2016) e55. doi:10.7717/peerj-cs.55.

[5] U. Villa, N. Petra, O. Ghattas, hIPPYlib: An Extensible Software Framework for Large-Scale Inverse Problems Governed by PDEs; Part I: Deterministic Inversion and Linearized Bayesian Inference, arXiv e-prints (2019). arXiv:1909.03948.

[6] J. D. Jakeman, F. Franzelin, A. Narayan, M. Eldred, D. Plfüger, Polynomial chaos expansions for dependent random variables, Computer Methods in Applied Mechanics and Engineering 351 (2019) 643–666. doi:10.1016/j.cma.2019.03.049.

[7] B. Adams, L. Bauman, W. Bohnhoff, K. Dalbey, M. Ebeida, J. Eddy, M. Eldred, P. Hough, K. Hu, J. Jakeman, J. Stephens, L. Swiler, D. Vigil, , T. Wildey, Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.10 User's Manual, sandia technical report sand2014-4633, july 2014. updated may 2019 (version 6.10) Edition.

[8] R. C. Smith, Uncertainty Quantification: Theory, Implementation, and Applications, Society for Industrial and Applied Mathematics, Philadelphia, 2014.