SMCPy Documentation

Release 1.0

Patrick Leser

Feb 11, 2019

CONTENTS:

1 Introduction **1**

2 Example - Spring Mass System **3**
- 2.1 Problem Specification . . . 3
- 2.2 Step 1: Initialization; define the model and generate the data . . . 4
- 2.3 Step 2: Perform Parameter Estimation using SMCPy . . . 5
- 2.4 Step 3: Perform Parameter Estimation using MCMCPy . . . 5

3 Source Code Documentation **7**
- 3.1 SMC Module Documentation . . . 7
- 3.2 Particle Module Documentation . . . 7
- 3.3 MCMC Module Documentation . . . 7

4 Indices and tables **9**

Python Module Index **11**

Index **13**

CHAPTER

ONE

INTRODUCTION

Uncertainty quantification (UQ) is necessary to provide meaningful and reliable predictions of real-world system performance. One major obstacle for the implementation of statistical methods for UQ is the use of expensive computational models. Classical UQ methods such as Markov chain Monte Carlo (MCMC) generally require thousands to millions of model evaluations, and, when coupled with an expensive model, result in excessive solve times that can render the analysis intractable. These methods are also inherently serial, which prohibits speedup by high performance computing. Recently, Sequential Monte Carlo (SMC) has emerged as a alternative to MCMC. In contrast, this method uses parallel model evaluations to realize significant speedup.

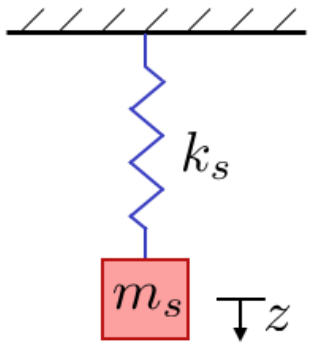
This software is an implementation of SMC that uses the Message Passing Interface (MPI) to provide users general access to parallel UQ methods in Python 2.7. The algorithm used is based on the work by Nguyen et al. ["Efficient Sequential Monte-Carlo Samplers for Bayesian Inference" IEEE Transactions on Signal Processing, Vol. 64, No. 5 (2016)]. To operate the code, the user supplies a computational model built in Python 2.7, defines prior distributions for each of the model parameters to be estimated, and provides data to be used for calibration. SMC sampling can then be conducted with ease through instantiation of the SMC class and a call to the sample() method. The output of this process is an approximation of the parameter posterior probability distribution conditioned on the data provided.

CHAPTER

TWO

EXAMPLE - SPRING MASS SYSTEM

This example provides a simple demonstration of SMCPy functionality. The goal is to inversely determine the uncertainty in the model parameters given a set of noisy observations of the spring mass system using Sequential Monte Carlo (SMC) and to compare the results to Markov chain Monte Carlo (MCMC). As SMC relies on a MCMC kernel, a fully functional MCMC sampler is included in the SMCPy package. In this example, it is used for verification of the SMC sampler. The example covers all steps for implementing SMC and MCMC samplers using SMCPy, including the creation of a user-defined computational model (spring mass numerical integrator) that uses the standardized SMCPy interface, defining prior distributions, initializing the samplers, computing statistical moments with the resulting estimators, and plotting the results. The full source code for this example can be found in the SMCPy repository: `/SMCPy/examples/spring_mass/spring_mass_example.py` and `/SMCPy/examples/spring_mass/mcmc_verify/spring_mass_mcmc.py` for the SMC and MCMC samplers, respectively. Data generation was conducted using `/SMCPy/examples/spring_mass/generate_noisy_data.py`.



2.1 Problem Specification

The governing equation of motion for the system is given by

$$m_s \ddot{z} = -k_s z + m_s g \tag{2.1}$$

where m_s is the mass, k_s is the spring stiffness, g is the acceleration due to gravity, z is the vertical displacement of the mass, and $\ddot{z}$ is the acceleration of the mass. The true stiffness and gravitational constant are unknown. The goal of this example is to, by observing the motion of the mass over time, z_t, estimate both K_s and G, which are random variables representing the uncertainty in the value of spring stiffness and the gravitational constant, respectively. For reasons outside the scope of this example, the problem becomes ill-posed unless the stiffness is normalized by mass such that realizations are now k_s^* of random variable of interest K_s^* and the equation of motion given by Equation (1) is now

$$\ddot{z} = -k_s^* z + g. \tag{2.2}$$

Given observations z_t which, assuming measurement noise exists, are realizations of the random variable Z_t, the relationship between the computational model, measurement noise, and observations is

$$Z_t = f_t(K_s^*, G) + \epsilon_t \tag{2.3}$$

Here, ϵ_t is the measurement noise associated with each observation of displacement taken over time (also a random variable), and f_t is the computational model response at time t, which involves numerical integration of Equation (2).

The inverse solution of equations in the form of (2) is the posterior distribution, or, in the context of this example, the joint distribution of K_s^* and G conditional on the observed data, z_t. While difficult to solve directly, the posterior distribution can be approximated via sampling methods such as SMC and MCMC. This example covers this process using the SMCSampler and MCMCSampler classes in the SMCPy Python module. In particular, the objective is to approximate the expected value of the model parameters given observations z_t.

The MCMC sampler draws samples from the unknown posterior distribution by forming a Markov chain through the parameter space whose stationary distribution is the posterior. The samples forming the chain can then be used to build an estimator of the expected value

$$E[X] = \frac{1}{N}\sum_{i=1}^{N} X_i$$

where X_i is the random variable of interest, and $i = 1, \ldots, N$, with N being the number of equally-weighted samples drawn using the MCMC sampler.

While MCMC is a proven approach to evaluting the quantities of interest, it can be slow if the computational model is expensive. The Markovian nature of MCMC means it is inherently a serial process. SMC, on the other hand, is a parallelizable alternative that uses weighted samples called "particles." In SMC, a sequence of target distributions is defined that gradually transitions from the typically-known prior distribution to the posterior distribution of interest. A particle filtering framework based on importance sampling can then be introduced that allows for recursive estimation of each target in the sequence. Taking the final particle state, the SMC estimator is

$$E[X] = \sum_{j=1}^{M} W_j X_j$$

where W_j is the normalized weight associated with the j^{th} particle and M is the total number of particles.

For this example, synthetic data will be generated by adding noise to a model response for a given set of "true" parameters. Next, MCMC and SMC will be used to estimate the expected value of these parameters with the expectation that these estimates are close to the true parameter values. Note that the code used to generate results in this example is actually split between two files in the SMCPy package. The two will be combined here and redundant code will be skipped.

2.2 Step 1: Initialization; define the model and generate the data

Begin by importing the needed Python modules, including SMCPy classes and the SpringMassModel class that defines the spring mass numerical integrator:

```
import numpy as np
from spring_mass_models import SpringMassModel
from smcpy.smc.smc_sampler import SMCSampler
```

Below is a snippet of the SpringMassModel class; the entire class can be found in the SMCPy repo (`/SMCPy/examples/spring_mass/spring_mass_model.py`):

```
from smcpy.model.base_model import BaseModel

...

class SpringMassModel(BaseModel):
    '''
    Defines Spring Mass model with 2 free params (spring stiffness, k &
    mass, m)
    '''
    def __init__(self, state0=None, time_grid=None):
```

Note that user-defined models in SMCPy must inherit from the SMCPy abstract class `BaseModel` and implement an `evaluate` function that accepts and returns numpy arrays for inputs and outputs, respectively. Here, the `state0` argument defines the initial state of the spring mass system, and `time_grid` defines the times at which to return displacement.

The first step in an analysis is to obtain data from which to make an inference. In this example, this data will come in the form of observations of the z-displacement of the mass made over time. For demonstration purposes, the data will be generated from the spring mass model, and noise will be added by sampling from a zero-mean Gaussian distribution and adding these values to the model output. While not a realistic case, it is typical to generate and use synthetic data in this manner for verification purposes when performing inverse uncertainty quantification.

2.3 Step 2: Perform Parameter Estimation using SMCPy

2.4 Step 3: Perform Parameter Estimation using MCMCPy

CHAPTER

THREE

SOURCE CODE DOCUMENTATION

Documentation for the primary SMCPy classes.

3.1 SMC Module Documentation

3.2 Particle Module Documentation

3.3 MCMC Module Documentation

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

smcpy, 7
smcpy.mcmc, 7
smcpy.particles, 7

INDEX

S

smcpy (module), 7
smcpy.mcmc (module), 7
smcpy.particles (module), 7