LA-UR-18-20185
Approved for public release; distribution is unlimited.

Title: Metis: A Pure Metropolis Markov Chain Monte Carlo Bayesian Inference Library

Author(s): Bates, Cameron Russell
Mckigney, Edward Allen

Intended for: Report

Issued: 2018-01-09

Metis: A Pure Metropolis Markov Chain Monte Carlo Bayesian Inference Library

Cameron R. Bates and Edward A. McKigney

January 3, 2018

Abstract

The use of Bayesian inference in data analysis has become the standard for large scientific experiments [1, 2]. The Monte Carlo Codes Group(XCP-3) at Los Alamos has developed a simple set of algorithms currently implemented in C++ and Python to easily perform flat-prior Markov Chain Monte Carlo Bayesian inference with pure Metropolis sampling. These implementations are designed to be user friendly and extensible for customization based on specific application requirements. This document describes the algorithmic choices made and presents two use cases.

1 Introduction

Pure Metropolis Markov Chain Monte Carlo is a method of calculating a posterior distribution of how well a continuous set of models represent data with uncertainties. Our implementation currently assumes that the covariance matrix of our data points is diagonal. In addition, it assumes that a Gaussian jump function for the Markov Chain Monte Carlo is appropriate for all variables. Finally, it assumes a uniform prior across the bounds provided by the user. While this combination of assumptions is not appropriate for all problems, we have found it to have broad applicability to many problems of interest to us.

The core Metropolis routine has the following structure:

1. Define starting point for model parameters and bounds
2. Calculate observables for x_0
3. Calculate the χ^2 of the observables given the data and uncertainties
4. Perform a Monte Carlo jump in all parameter dimensions
5. The standard deviation of the jump size is specified by the user for each dimension independently
6. If the jump is outside a bound try again until the new x lies within all bounds
7. Calculate observables for x_n
8. Calculate the χ^2 of the observables given the data and uncertainties
9. Calculate if the log likelihood ratio is greater than a random number on [0,1]
10. If it is jump to x_n
11. If it is not remain at x_{n-1}
12. If $n >$ burn in and n%skip== 0 then save x_n
13. Go back to step four until n=total samples

As long as n is large enough that the distribution has converged, the distribution of x values will be representative of the posterior distribution of model parameters. In both C++ and Python the implementation has been designed such that the core evaluation function used to calculate the observables for the model can be provided by the user.

Two applications of this technique are evaluation of nuclear data and unfolding of neutron spectra from a liquid scintillator.

The data used by neutron transport codes like PARTISN and MCNP is based on evaluations of nuclear data. These evaluations attempt to combine nuclear theory with measured data to produce a best estimate of nuclear data at all energies. Bayesian inference is a powerful tool for this type of work. One example of this is the evaluation of the total kinetic energy of fission fragments. For this evaluation, we used data for ^{235}U ^{238}U, and ^{239}Pu. Theoretical models that use an empirically fit set of constant for all three isotopes have been developed separately [3]. In order to determine an uncertainty in the evaluation it is necessary to jointly infer parameters across all three models and calculate a posterior distribution of valid model parameters. Metis makes this task trivial. The results of this analysis are shown in Figure 1 and will be included in ENDF-VIII.

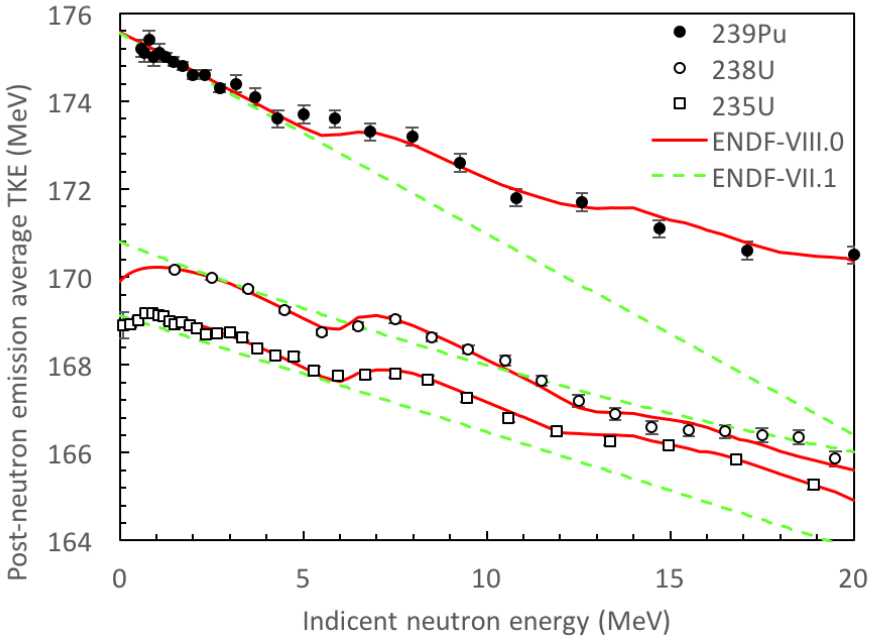

Figure 1: Experimental data for the post-neutron emission average TKE as a function of incident neutron energy, En, for the three targets ^{235}U [4], ^{239}Pu [5], and ^{238}U [6]. The red and green curves show the ENDF-VIII.0 and VII evaluations, respectively. The ^{238}U data and evaluations curves have been shifted upwards by 1 MeV, to avoid clutter.

A second use case is unfolding of neutron spectroscopy data. In a simple model example we have a Watt spectrum of neutrons incident on a liquid scintillator. These neutrons produce proton recoils that can be up to the total neutron energy but are most often much less. This makes the inference of the incident energy non-trivial because the uncertainties in high energy bins can dominate the uncertainty in the inferred spectrum. In order to make this problem tractable it is often reduced to just a few bins and then the incident spectrum is calculated using simple minimization techniques. This does not allow for any uncertainty quantification and limits the fidelity of the results. With Metis it is possible to reconstruct the full Watt fission spectrum with uncertainties based on a set of measured data with uncertainties. Figure 2 shows the reconstruction of an incident neutron spectrum in an idealized model case. We have used this technique for inference of measured data with similar results.

As a check on the accuracy of the MCMC method we also developed an analytic integral method and

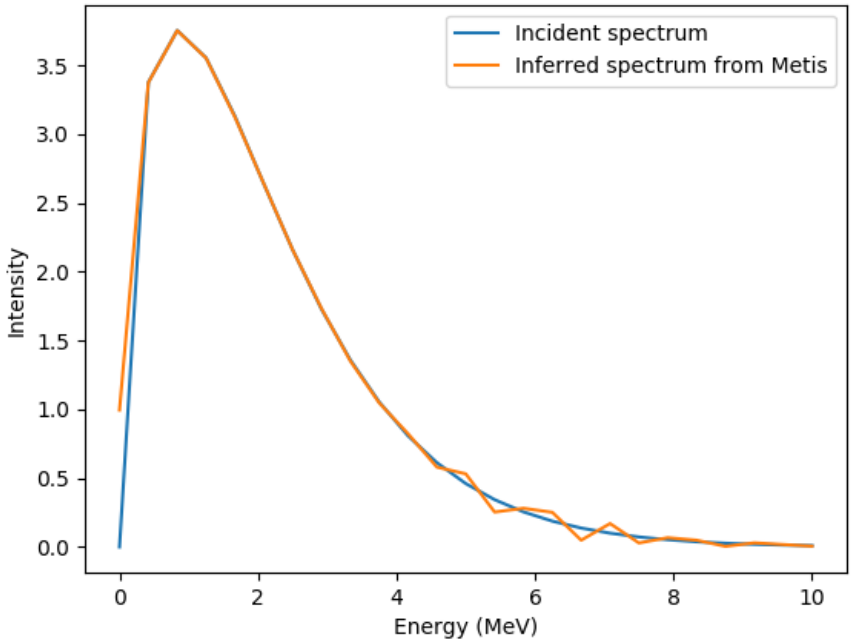

Figure 2: Neutron spectrum unfolding of a ^{252}Cf source from a model of a liquid scintillator detector with the incident source in the model plotted.

verified that the MCMC method converges to it. This method is computationally expensive as it has to perform a calculation for every voxel in parameter space and has numerical stability issues in areas with large gradients in observables so it has limited usefulness for realistic applications.

References

[1] B.P. Abbott et al. (LIGO Scientific Collaboration and Virgo Collaboration). Observation of Gravitational Waves from a Binary Black Hole Merger. Phys. Rev. Lett. 116, 061102.

[2] L. Lyons, Bayes and Frequentism: a Particle Physicist's perspective. DOI: 10.1080/00107514.2012.756312

[3] J. P. Lestone and T. T. Strother, Energy Dependence of Plutonium and Uranium Average Fragment Total Kinetic Energies, Nuclear Data Sheets, 118, 208 (2014).

[4] D. L. Duke, Fission Fragment Mass Distributions and Total Kinetic Energy Release of 235-Uranium and 238-Uranium in Neutron-Induced Fission at Intermediate and Fast Neutron Energies, Colorado School of Mines PhD Thesis, Los Alamos National Laboratory Report LA-UR-15-28829.

[5] K. Meierbachtol, F. Tovesson, D. L. Duke, V. Geppert-Kleinrath, B. Manning, R. Meharchand, S. Mosby, and D. Shields, Total kinetic energy release in 239Pu(n, f) post-neutron emission from 0.5 to 50 MeVincident neutron energy, Phys. Rev. C 94, 034611 (2016).

[6] C. M. Zller, Investigation of Neutron-Induced Fission of 238U in the Energy Range from 1 MeV to 500 MeV, Ph. D. thesis. Technische Hochschule Darmstadt, Germany (1995).

A Basic Implementation

This simple implementation has no burn-in period or skips but is meant to illustrate the simplicity of the methodology.

```
def prob_density(value, sigma, model):
    expo = np.sum((value-model)**2/(sigma**2))
    return -expo
def newloc(current,limits,jumps):
    current = np.ones(len(current))*current
    theloc = current + np.random.normal(0,1.0,len(current))*jumps
    mask = np.logical_or(limits[:,1] < theloc, theloc < limits[:,0])
    while np.sum(mask) > 0:
        mi = np.nonzero(mask)
        theloc[mi] = current[mi] + \
                     np.random.normal(0, 1.0, np.sum(mask))*jumps[mi]
        mask = np.logical_or(limits[:,1] < theloc,
                             theloc < limits[:,0])
    current = theloc
    return current
def runmcmc(nsamples, loc, model_spec,
            jumps, limits, spec, sigmas, extended =False):
    n=0
    bloc = loc
    nbins = 1000
    deltas = (limits[:,1] - limits[:,0])/float(nbins)
    vals = model_spec(loc)
    prob = prob_density(spec,sigmas,vals)
    bp = prob
    h = np.zeros((len(loc),nbins))
    indices = range(len(loc))
    loclist = None
    if extended:
        loclist = np.zeros((nsamples,len(limits)+1))
    accepted = 0
    while n < nsamples:
        tmploc = newloc(loc,limits,jumps)
        vals = model_spec(tmploc)
        tmpprob = prob_density(spec,sigmas,vals)
        if tmpprob > prob or np.exp((tmpprob - prob)*0.5) > np.random.rand():
            prob = tmpprob
            loc = tmploc
            accepted += 1
        if tmpprob > bp:
            bp = tmpprob
            bloc = loc
        h[indices, np.array((loc-limits[:,0])/deltas,dtype='int')] += 1
        if loclist is not None:
            loclist[n][:-1] = loc
            loclist[n][-1] = prob
        n += 1
    if extended:
        return h, bloc, float(accepted)/float(nsamples), loclist
    return h, bloc, float(accepted)/float(nsamples)
```