UNIVERSITY OF CAMBRIDGE

Algorithm & Errors

Presented
By
Matteo Mancini

Department
Department of Physics

Degree
MPhil Data Intensive Science

Project Supervisor
Dr. Christopher J. Moore

GitHub:
https://github.com/MatteoMancini01/Gaia_EDR3

Easter Term 2025

Contents

1 Introduction

This document is intended as supplementary material and should not be considered for marking. Its purpose is to provide a comprehensive overview of the mathematical and computational methodologies employed in this project, targeting readers beyond the examination context. Here we detail the formalism for error propagation, as well as the full implementation and theoretical background of the sampling algorithms used, in particular Hamiltonian Monte Carlo (HMC). By consolidating this technical information, we aim to ensure the transparency and reproducibility of the analysis presented in the main report.

2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo method; this algorithm is used to determine a sequence of random samples whose distribution over iterations will converge to the target probability distribution. HMC is a variant of the Metropolis-Hastings algorithm that uses simulated Hamiltonian dynamics to propose new states in the sampling process. Instead of relying on a simple Gaussian random walk, HMC integrates the system using a time-reversible, volume-preserving symplectic integration (in our case leapfrog method, implemented in the package NumPyro) to generate proposals. These proposals often reach distant regions of the parameter space while maintaining a high acceptance probability due to the approximate conservation of energy. As a result, HMC produces less correlated samples, improving sampling efficiency and requiring fewer iterations to achieve a given level of accuracy, Wikipedia contributors (2023).

Algorithm 1 Hamiltonian Monte Carlo (HMC)

Input: Initial position θ, step size ϵ, number of steps L, potential energy $U(\theta)$, and mass matrix M
Sample momentum $p \sim \mathcal{N}(0, M)$
Set $(\theta_0, p_0) \leftarrow (\theta, p)$
Compute (θ', p') by simulating Hamiltonian dynamics with L leapfrog steps:
for $i = 1$ to L **do**
  $p \leftarrow p - \frac{\epsilon}{2}\nabla U(\theta)$
  $\theta \leftarrow \theta + \epsilon M^{-1} p$
  $p \leftarrow p - \frac{\epsilon}{2}\nabla U(\theta)$
end for
Negate momentum: $p' \leftarrow -p$ ▷ Ensures reversibility
Compute acceptance probability:

$$\alpha = \min\left(1, \exp\left[H(\theta_0, p_0) - H(\theta', p')\right]\right)$$

Accept θ' with probability α, else retain θ_0

No U-Turn Sampler

The No U-Turn Sampler (NUTS) is an extension of HMC that automatically determines the optimal number of leapfrog steps L, eliminating the need for manual tuning. Choosing L appropriately is critical: in simple examples such as 1D Gaussian distribution $\mathcal{N}(0, 1/\sqrt{k})$, the potential energy $U(x) = \frac{1}{2}kx^2$ corresponding to that of a harmonic oscillator. If L is too large, the trajectory loops back. causing wasted computation. if L is too small, the sampler behaves like a random walk reducing efficiency.

NUTS overcomes this by simulating Hamiltonian dynamics both forward and backward in time, continuing until a U-turn condition is met, i.e. when the trajectory starts to retrace itself. In that instance, a point along the trajectory is chosen at random and accepted as the next step. This adaptivity improves mixing and sampling efficiency.

More specifically, NUTS builds a binary tree to track leapfrog steps. At each iteration, slice variable $U_n \sim \mathcal{U}(0, \exp\left[-H(x_n(0), p_n(0)\right]$is sampled, where H is the Hamiltonian. The sampler maintains two trajectories: one moving forward (x_n^+, p_n^+) and one moving backward (x_n^-, p_n^-). In each iteration, the algorithm randomly chooses to extend either end of the trajectory by doubling the number of leapfrog steps, continuing until the U-turn condition is satisfied, Wikipedia contributors (2023).

Algorithm 2 No-U-Turn Sampler (Termination & Sample Selection)

Input: Leapfrog path $\{\mathbf{x}_n^-, \ldots, \mathbf{x}_n(0), \ldots, \mathbf{x}_n^+\}$, momenta $\mathbf{p}_n^\pm$, Hamiltonian $H(\cdot)$, slice variable U_n, threshold δ
while not U-Turn and Hamiltonian is accurate **do**
  if $(\mathbf{x}_n^+ - \mathbf{x}_n^-) \cdot \mathbf{p}_n^- < 0$ **or** $(\mathbf{x}_n^+ - \mathbf{x}_n^-) \cdot \mathbf{p}_n^+ < 0$ **then**
    break ▷ U-Turn condition met
  end if
  if $\exp[-H(\mathbf{x}_n^+, \mathbf{p}_n^+) + \delta] < U_n$ **or** $\exp[-H(\mathbf{x}_n^-, \mathbf{p}_n^-) + \delta] < U_n$ **then**
    break ▷ Energy error too large
  end if
end while
Sample $\mathbf{x}_{n+1}$ uniformly from the tree path where

$$U_n < \exp[-H(\mathbf{x}_{n+1}, \mathbf{p}_{n+1})]$$

Python Implementation

To perform HMC sampling with Python, one can either code Algorithms 1 and 2 from scratch or use existing optimised packages such as NumPyro. NumPyro is a lightweight, flexible probabilistic programming library built on JAX that enables scalable Bayesian inference using modern MCMC methods like Hamiltonian Monte Carlo (HMC) and the No-U-Turn Sampler (NUTS). It combines NumPy-like syntax with JAX's automatic differentiation and GPU/TPU acceleration for efficient, high-performance inference. Here is an example of how to perform HMC sampling with NumPyro:

```python
import numpyro
from numpyro.infer import NUTS, MCMC, Predictive
from numpyro import handlers
from numpyro.diagnostics import summary, autocorrelation
import numpyro.distributions as dist
#%%
def chi2_jit(angles, obs, error, theta, lmax):
    return least_square(angles, obs, error, theta, lmax=lmax, grid=False)
chi2_jit = jit(chi2_jit, static_argnames=['lmax'])


def model_for_HMC(angles, obs, error, lmax):
    total_params = count_vsh_coeffs(lmax)

    # Prior on all VSH coefficients (both toroidal and spheroidal)
    theta = numpyro.sample("theta", dist.Normal(0.0, 1.0).expand([total_params]))
    # Least-squares residuals: we assume Gaussian-distributed residuals
    chi2_val = chi2_jit(angles, obs, error, theta, lmax=lmax)

    # The log-likelihood is proportional to -0.5*chi^2
    numpyro.factor("likelihood", -0.5*chi2_val)

n_s = 5000 # number of samples
n_warmup = 2000 #  number of warmups
n_chains = 8 # numbe of chains

rng_key = jax.random.key(0)
kernel = NUTS(model_for_HMC, target_accept_prob=0.75) # this is to make sure
                                                      #acceptance does not exceed 90%

angles, obs, error = config_data(df)
mcmc = MCMC(kernel, num_warmup=n_warmup, num_samples=n_s,
num_chains=n_chains, chain_method='sequential', progress_bar=True)
mcmc.run(rng_key, angles = angles, obs = obs, error = error, lmax=3)
ps = mcmc2.get_samples()
```

```
36 mcmc.print_summary()
```

See https://github.com/MatteoMancini01/Gaia_EDR3.

3 Statistical Checks

To validate our posterior samples, we performed few powerful convergence checks as well as dealing with potential autocorrelation in each sample.

3.1 Integrated Autocorrelation Time (IAT)

Integrated Autocorrelation Time (IAT) is a diagnostic used to assess the efficiency of Markov Chain Monte Carlo (MCMC) sampling. It measures how many iterations are effectively independent by quantifying the correlation between successive samples in a chain. A lower IAT indicates that the chain mixes well, and fewer samples are needed to approximate the posterior distribution. Conversely, a high IAT suggests strong correlation between samples, requiring more samples for accurate estimation. In our analysis, IAT was computed for each VSH coefficient in the posterior distribution. The estimated IAT values were found to be consistently low (IAT = 2), indicating rapid mixing and justifying the use of thinning to reduce sample redundancy. Specifically, we thinned the samples by keeping only one every 2 steps to obtain effectively independent samples for posterior summaries.

Function In Python

We designed a function in Python `estimate_iat()` that estimates the (IAT) frp, effective sample size (`n_eff`) values obtained during the MCMC sampling.

```python
def estimate_iat(n_samples, n_chains, n_eff_arr, index = None):

    if index:
        subset_n_eff = n_eff_arr[index]
    else:
        subset_n_eff = n_eff_arr

    total_samples = n_samples*n_chains

    iat = jnp.ceil(total_samples/jnp.min(subset_n_eff))

    return int(iat)
```

Inputs

- `n_samples`: Number of samples per MCMC chain (after burn-in).
- `n_chains`: Total number of MCMC chains used.
- `n_eff_arr`: Array of effective samples sizes (one per parameter)

How it computes the IAT estimate:

1. Calculates the total number of samples across all chains
2. Uses the ruel:
$$\text{IAT} = \left\lceil \frac{\text{total samples}}{\min(n_{\text{eff}})} \right\rceil \tag{1}$$
3. This returns the estimated IAT values as an integer, which can be used for thinning the chain (i.e. keeping only every IAT-th sample).

3.2 Gelman-Rubin statistic

After removing the burn-in (this is automatically done with NumPyro), we want to check if all chains have converged. A powerful approach is to compute the Gelman-Rubin (GR) statistic (Wikipedia contributors, 2024a), to compute it we require the samples $x_1^{(j)}, \ldots, x_L^{(j)}$ (for the j-th chain), the variance between chains as well as the variance in the chains estimates:

- $\bar{x} = \frac{1}{L}\sum_{i=1}^{L} x_i^{(j)}$ parameter mean value of chain j.
- $\bar{x}_* = \frac{1}{J}\sum_{j=1}^{J} \bar{x}_j$ this the mean of the parameter means of all chains.
- $B = \frac{L}{J-1}\sum_{j=1}^{J}(\bar{x}_j - \bar{x}_*)^2$ where B is the variance of the means from each chain.
- $W = \frac{1}{J}\sum_{j}^{J}\left[\frac{1}{L-1}\sum_{i=1}^{L}(x_i^{(j)} - \bar{x}_j)^2\right]$ where W average within-chain variance, computed by averaging the variances of each chain.

Hence the estimate of the GR statistic $\hat{R}$ is:

$$\hat{R} = \frac{\frac{L-1}{L}W + \frac{1}{L}B}{W} \tag{2}$$

If the Gelman–Rubin statistic is approximately 1, it indicates that the chains have successfully converged. This diagnostic is computed automatically by NumPyro and can be accessed via the `r_hat` attribute using the `numpyro.diagnostics` module.

4 Propagation of Error

The uncertainties on the VSH-derived parameters were obtained from the posterior covariance matrix estimated directly from the HMC samples. By propagating this covariance through analytical Jacobian transformations, we derived uncertainties for Cartesian vector components, the vector norm, and the corresponding right ascension and declination angles. This approach accounts for correlations between parameters and ensures consistent error estimates on all derived quantities. The covariance matrix from the posterior samples is computed as:

$$\Sigma_\theta = \frac{1}{N-1}\sum_{i=1}^{N}(\theta_i - \bar{\theta})(\theta_i - \bar{\theta})^T, \tag{3}$$

where θ_i is a vector of parameters from the i-th posterior sample, and $\bar{\theta}$ is the mean vector. We then transformed the covariance of the VSH coefficients to the vector components g_x, g_y, g_z via the Jacobian:

$$\Sigma_{\mathbf{g}} = J\Sigma_\theta J^T \tag{4}$$

where the Jacobian matrix J is defined as:

$$J = \begin{bmatrix} \frac{\partial g_x}{\partial s_{11}^{\mathfrak{R}}} & \frac{\partial g_x}{\partial s_{11}^{\mathfrak{I}}} & \frac{\partial g_x}{\partial s_{10}} \\ \frac{\partial g_y}{\partial s_{11}^{\mathfrak{R}}} & \frac{\partial g_y}{\partial s_{11}^{\mathfrak{I}}} & \frac{\partial g_y}{\partial s_{10}} \\ \frac{\partial g_z}{\partial s_{11}^{\mathfrak{R}}} & \frac{\partial g_z}{\partial s_{11}^{\mathfrak{I}}} & \frac{\partial g_z}{\partial s_{10}} \end{bmatrix} = \begin{bmatrix} -\sqrt{\frac{3}{4\pi}} & 0 & 0 \\ 0 & \sqrt{\frac{3}{4\pi}} & 0 \\ 0 & 0 & \sqrt{\frac{3}{8\pi}} \end{bmatrix} \tag{5}$$

Then for each individual uncertainties:

$$\begin{aligned} \sigma_{g_x} &= \sqrt{\Sigma_{g,11}}, \\ \sigma_{g_y} &= \sqrt{\Sigma_{g,22}}, \\ \sigma_{g_z} &= \sqrt{\Sigma_{g,33}}. \end{aligned} \tag{6}$$

The correlation between each parameter estimate was computed as:

$$\rho_{ij} = \frac{\Sigma_{g,ij}}{\sigma_{g_i} \cdot \sigma_{g_j}} \tag{7}$$

for $i, j = x, y, z$. The uncertainty on the norm of the vector is computed via error propagation,

$$\sigma_{|\mathbf{g}|} = \sqrt{\frac{\mathbf{g}T\Sigma_g\mathbf{g}}{|\mathbf{g}|^2}} \tag{8}$$

where $|\mathbf{g}| = \sqrt{g_x^2 + g_y^2 + g_z^2}$ is the magnitude of the estimated acceleration components. Similarly, we determined the uncertainties on the angles RA and Dec, but first we need to consider how RA and Dec are derived from the acceleration components estimates:

$$\alpha = \arctan 2(g_x, g_y),\ \delta = \arcsin\left(\frac{g_z}{|\mathbf{g}|}\right) \tag{9}$$

where RA=α and Dec=δ. Their uncertainties use a Jacobian transformation J^* from (g_x, g_y, g_z):

$$\Sigma_{\alpha,\delta} = J^*\Sigma_g J^{*T} \tag{10}$$

where J^* is defined as follows:

$$J^* = \begin{bmatrix} -\frac{g_y}{r_1} & \frac{g_x}{r_1} & 0 \\ -\frac{g_x g_z}{r_0\sqrt{r_1}} & -\frac{g_y g_z}{r_0\sqrt{r_1}} & \frac{\sqrt{r_1}}{r_0} \end{bmatrix}, \tag{11}$$

here $r_0 = |\mathbf{g}|^2$ and $r_1 = g_x^2 + g_y^2$. Just like for the acceleration components, the individual uncertainties for RA and Dec are:

$$\sigma_\alpha = \sqrt{\Sigma_{\alpha\alpha}},\ \sigma_\delta = \sqrt{\Sigma_{\delta\delta}}, \tag{12}$$

with correlation:

$$\rho_{\alpha,\delta} = \frac{\Sigma_{\alpha\delta}}{\sigma_\alpha\sigma_\delta} \tag{13}$$

We repeated a similar procedure to convert the equatorial components to galactic components.

5 Power Spectral Density (PSD)

The power spectral distribution (PSD) (Wikipedia contributors, 2024b) provides a quantitative way to examine how signal variance is distributed among different spherical harmonic degrees, allowing us to verify that the inferred acceleration signal is dominated by the expected large-scale modes and to assess the impact of possible systematic effects.

To compute the power spectra from the Vector Spherical Harmonic (VSH) expansion, use the following formulas for each multipole l:

Toroidal Power Spectrum C_l^T

$$C_l^T = \frac{1}{2l+1}\left(t_{l0}^2 + 2\sum_{m=1}^{l}\left[\left(t_{lm}^{\mathfrak{R}}\right)^2 + \left(t_{lm}^{\mathfrak{I}}\right)^2\right]\right) \tag{14}$$

Spheroidal Power Spectrum C_l^S

$$C_l^S = \frac{1}{2l+1}\left(s_{l0}^2 + 2\sum_{m=1}^{l}\left[\left(s_{lm}^{\mathfrak{R}}\right)^2 + \left(s_{lm}^{\mathfrak{I}}\right)^2\right]\right) \tag{15}$$

These give the toroidal and spheroidal angular power spectra, analogous to C_l^B and C_l^E in CMB polarisation analysis.

6 Links to Code

This part of the appendix is dedicated to guide the reader to the source of the results presented in this report.

i All the functions presented in Section 2, of the main report, are coded in the Python file `src.models.vsh_model.py`, see https://github.com/MatteoMancini01/Gaia_EDR3/blob/main/src/models/vsh_model.py.

ii For propagation of error, and output layout see `src.models.configuration.py` at https://github.com/MatteoMancini01/Gaia_EDR3/blob/main/src/models/configuration.py.

iii For all Figures and Tables in this report, please see the three Jupyther notebooks `main.ipynb`, `plots.ipynb` and `extension.ipynb`, see https://github.com/MatteoMancini01/Gaia_EDR3/tree/main.

For more details see `README.md` at https://github.com/MatteoMancini01/Gaia_EDR3/blob/main/README.md

References

Wikipedia contributors. (2023). *Hamiltonian monte carlo* [Accessed: 2025-06-21]. https://en.wikipedia.org/wiki/Hamiltonian_Monte_Carlo

Wikipedia contributors. (2024a). Gelman–rubin statistic — wikipedia, the free encyclopedia [Accessed: 2025-06-27]. https://en.wikipedia.org/wiki/Gelman-Rubin_statistic

Wikipedia contributors. (2024b). Spectral density — wikipedia, the free encyclopedia [[Accessed: 2025-06-28]].