Development and Benchmarking of a Bayesian Inference Pipeline for LHC Physics

Scientific Thesis for the procurance of the degree

Bachelor of Science

from the Physics Department at the Technical University of Munich.

Supervised by *Prof. Dr. Lukas Heinrich*
ORIGINS Data Science Lab

Dr. Oliver Schulz
Max Planck Institute for Physics

Submitted by *Christian Gajek*

Submitted on September 22, 2022

Abstract

In High Energy Physics statistical models are an important component to verify the compatibility of collision events with theoretical predictions. A large portion of these models can be sufficiently described by statistical frameworks like HistFactory where parameter inference is mainly carried out in a frequentist manner. In contrast, this thesis aims to perform Bayesian inference on the model parameters and derives a formalism to obtain prior distributions from auxiliary measurements. In addition a complete Bayesian inference pipeline is implemented in Python which uses HistFactory models and the Julia implementation of BAT (Bayesian Analysis Toolkit) to carry out parameter inference. The posterior distributions are generated by two MCMC methods – Metropolis-Hastings and Hamiltonian Monte Carlo sampling. All required steps for interfacing Julia code in Python (and vice versa) are elaborated in this work for both sampling methods. Furthermore theoretical foundations of MCMC sampling, Bayesian inference and HistFactory are presented. Finally the Python-Julia pipeline is benchmarked against a pure-Julia Bayesian inference implementation for different scenarios and the results are discussed.

Contents

Abbreviations

Autograd	auto differentiation.
BAT	Bayesian Analysis Toolkit.
GC	garbage collector.
HEP	High Energy Physics.
HF	HistFactory.
HMC	Hamiltonian Monte Carlo.
LHC	Large Hadron Collider.
llh	log likelihood.
MC	Monte Carlo.
MCMC	Markov chain Monte Carlo.
MH	Metropolis-Hastings.
NP	nuisance parameter.
pdf	probability density function.
POI	parameter of interest.
XLA	accelerated linear algebra.

Chapter 1

Introduction

Large Hadron Collider (LHC)
Luminosity LHC $2.1 \cdot 10^{3}4cm^{-2}s^{-}1$

Chapter 2

Mathematical Preliminaries

2.1 Frequentist versus Bayesian Inference

Given observed data, the likelihood $\mathcal{L}(\boldsymbol{\theta})$ then serves as the basis to test hypotheses on the parameters $\boldsymbol{\theta}$.

$$p\,(\theta\,|\,x) = \frac{p\,(x\,|\,\theta)\,p(\theta)}{p(x)} \tag{2.1}$$

where $p\,(x\,|\,\theta)$ called *likelihood*, $p(\theta)$ the *prior*, $p(x)$ the *evidence* and $p\,(\theta\,|\,x)$ the *posterior*. i.e. the evidence that the data x was generated by this model.

2.2 MCMC Sampling

The key difficulty for computing the posterior $p(\theta|x)$ in Bayes' rule (2.1) is dividing by the evidence $p(x)$. There are only a few special cases where the posterior density can be computed analytically. In most cases $p(x)$ is a high-dimensional, complex density where sampling with standard Monte Carlo (MC) methods is intractable [1].

Instead Markov chain Monte Carlo (MCMC) methods are used to sample from $p(x)$. These methods create a reversible Markov chain that has as an equilibrium distribution which matches the posterior distribution.

2.2.1 Markov Chains

A Markov chain generates a *correlated* sequence of states $x_1, \ldots, x_n$. Each step in the sequence is drawn from a transition operator $T(x' \leftarrow x)$, which represents the probability of moving from state x to state x'. According to the *Markov property*, the transition probability[1] only depends on the current state x. In order to achieve an equilibrium distribution, the Markov chain must fullfil the property of *detailed balance*

$$T(x' \leftarrow x)p(x) = T(x \leftarrow x')p(x') \qquad \forall x, x' \tag{2.2}$$

This means that transitions under T at equilibrium have the same probability "forwards" $x \rightarrow x'$ and "backwards" $x' \rightarrow x$. When a Markov chain satisfies detailed balance, it is called *reversible*, as it is statistically indistinguishable whether the chain runs forwards or backwards in time.

2.2.2 Metropolis-Hastings

The Metropolis-Hastings (MH) algorithm [2] is the original MCMC algorithm to generate a set of random numbers that have the properties of a Markov chain and converge towards a target distribution. The MH procedure is summarized in Algorithm 1.

Metropolis-Hastings is initialized by an initial state x and the number of iterations *nsteps*. For each step we sample proposals for the next state x' from the proposal distribution $q(x' \leftarrow x)$. Next, we calculate the acceptance probability a (line 4) which is used to decide whether to accept or reject the proposal. We accept the proposal, if a is greater than a uniformly sampled random number r and update the value of x. Otherwise the proposal is rejected.

[1] and therefore the next state x'

It is easy to show that MH satisfies detailed balance in (2.2) [1].

Algorithm 1 Metropolis-Hastings [1]

```
1: Input: initial state $x$, iterations $nsteps$
2: for $n = 1 \ldots nsteps$ do
3:     Propose  $x' \sim q(x' \leftarrow x)$
4:     Compute the acceptance probability  $a = \min\left(1,\ \frac{p(x')}{p(x)}\frac{q(x' \leftarrow x)}{q(x \leftarrow x')}\right)$
5:     Accept or reject
           (a) Draw  $r \sim \text{Uniform}(0, 1)$
           (b) if($r \leq a$) then  accept the new state, set $x \leftarrow x'$
                              else  reject the proposal
6: end for
```

Setting x at the end of each iteration is considered as sample from the density $p(x)$.

In general, MH is valid for any proposal distribution q. “Ideal” proposals would ensure a rapid exploration of the distribution of interest p. There are sophisticated approaches that are based on the observed data [1]. However, q is often chosen to be symmetric and centered around x, e.g. a Normal distribution. Then the acceptance probability (line 4) simplifies to

$$a = \min\left(1,\ \frac{p(x')}{p(x)}\right) \tag{2.3}$$

MH has a low *acceptance ratio*[2] as the proposals basically perform a *random walk* in the parameter space.

2.2.3 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) sampling produces distant proposals for the Metropolis algorithm while avoiding the slow exploration of the parameter space by random-walk proposals [3]. It employs the *Hamiltonian dynamics* of classical mechanics and the likelihood gradient for an enhanced exploration of the parameter space. However, HMC is restricted to differential likelihoods.

[2] the ratio of proposed samples that are accepted

Hamiltonian dynamics

Hamiltonian dynamics operates on a $2d$ dimensional phase space with *position* vector $\boldsymbol{q} \in \mathbb{R}^d$ and *momentum* vector $\boldsymbol{p} \in \mathbb{R}^d$. The system is fully described by the Hamiltonian

$$\mathcal{H}(\boldsymbol{q}, \boldsymbol{p}) = U(\boldsymbol{q}) + V(\boldsymbol{p}) \tag{2.4}$$

where $U(\boldsymbol{q})$ is the *potential energy* and $V(\boldsymbol{p})$ the *kinetic energy* of the system. The change of the generalized coordinates $\boldsymbol{q}$ and $\boldsymbol{p}$ over time is determined by the partial derivatives of the Hamiltonian

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial \mathcal{H}}{\partial p_i} \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial \mathcal{H}}{\partial q_i} \qquad \text{for } i = 1, ..., d \tag{2.5}$$

The kinetic energy in (2.4) is defined as

$$V(\boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^T \boldsymbol{M}^{-1} \boldsymbol{p} \tag{2.6}$$

where $\boldsymbol{M}$ is a symmetric, positive-definite mass matrix that is typically diagonal. Applying Equations (2.5) to the kinetic energy in (2.6) and $U(\boldsymbol{q})$ one obtains

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \big[\boldsymbol{M}^{-1}\boldsymbol{p}\big]_i \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial U(\boldsymbol{q})}{\partial q_i} \tag{2.7}$$

Probability and the Hamiltonian – Canonical distributions

When using Hamiltonian dynamics to sample from a distribution, the density function needs to be translated to a potential energy function. A solution for this requirement is the *canonical distribution* from statistical mechanics. Given a energy function $E(x)$ for a state x, the canonical distribution has a probability density function (pdf) of [3]

$$p(x) = \frac{1}{Z}\exp\big(-E(x)/T\big) \tag{2.8}$$

where T is the temperature of the system and Z a normalization constant. The Hamiltonian in (2.4) describes the energy for the joint state of position $\boldsymbol{q}$ and momentum $\boldsymbol{p}$. With (2.8) we obtain the *joint* distribution for $\boldsymbol{q}$ and $\boldsymbol{p}$ as

$$p(\boldsymbol{q}, \boldsymbol{p}) = \frac{1}{Z}\exp\big(-\mathcal{H}(\boldsymbol{q}, \boldsymbol{p})/T\big) = \frac{1}{Z}\exp\big(-U(\boldsymbol{q})/T\big)\exp\big(-K(\boldsymbol{p})/T\big) \tag{2.9}$$

For the definition of $\mathcal{H}(\boldsymbol{q}, \boldsymbol{p})$ in (2.4), the densities for $\boldsymbol{q}$ and $\boldsymbol{p}$ are distributed by *independent* canonical distributions with energies $U(\boldsymbol{q})$ and $K(\boldsymbol{p})$.

HMC Algorithm

In Bayesian statistics, we define the potential energy function as [3]

$$U(\boldsymbol{q}) = -\log\Big[p\left(\boldsymbol{q}\,|\,\boldsymbol{x}\right)\, p(\boldsymbol{q})\Big] \tag{2.10}$$

where $p\left(\boldsymbol{q}\,|\,\boldsymbol{x}\right)$ is the likelihood and $p(\boldsymbol{q})$ the prior density. As the canonical distributions of $\boldsymbol{q}$ and $\boldsymbol{p}$ in (2.9) are independent, we can choose the distribution of the momentum variables $\boldsymbol{p}$ independent of $\boldsymbol{q}$. A practical choice for the momentum variables

$$\boldsymbol{p} \sim \text{Normal}(0, \boldsymbol{\Sigma}) \tag{2.11}$$

is a multivariate zero-mean Gaussian distribution [3].

After the preliminary work the HMC algorithm can be stated.

Algorithm 2 Hamiltonian Monte Carlo

```
Input: initial state $q_0$, $nsteps$, $\boldsymbol{\Sigma}$, $L$, $\epsilon$
for $i = 1 \dots nsteps$ do
    Draw   $p \sim \text{Normal}(0, \boldsymbol{\Sigma})$
    $(q^{(0)}, p^{(0)}) = (q_{i-1}, p)$
    for $j = 1 \dots L$ do                                          ▷ apply $L$ leapfrog steps
        $p^{(j-\frac{1}{2})} = p^{(j-1)} - \frac{\epsilon}{2}\nabla U(q^{(j-1)})$
        $q^{(j)} = q^{(j-1)} + \epsilon\boldsymbol{\Sigma}p^{(j-\frac{1}{2})}$
        $p^{(j)} = p^{(j-\frac{1}{2})} - \frac{\epsilon}{2}\nabla U(q^{(j)})$
    end for
    $(q^*, p^*) = (q^{(L)}, p^{(L)})$
    Draw   $r \sim \text{Uniform}(0, 1)$                            ▷ Metropolis acceptance step
    $a = \min\Big(1,\ \exp\big(-\mathcal{H}(q^*, p^*) + \mathcal{H}(q, p)\big)\Big)$
    if $(r \leq a)$ then
        $(q_i, p_i) = (q^*, p^*)$                                    ▷ *accept*
    else
        $(q_i, p_i) = (q_{i-1}, p_{i-1})$                            ▷ *reject*
    end if
end for
return $\{q_i, p_i\}_{i=0}^{nsteps}$
```

HMC is initialized with an initial value q_0, the number of *nsteps*, a covariance matrix $\mathbf{\Sigma}$, the number of leapfrog steps L as well as the leapfrog step size ϵ. In every step the momentum variable p is drawn from the multivariate Normal distribution. Next, the position and momentum variables q and p are integrated in time by a leapfrog integrator for L steps with step size ϵ. The differentials for the numerical integration are derived from (2.7). The integrated values at step L represent the proposals (q^*, p^*) which are evaluated in a Metropolis acceptance step similar to Algorithm 1. Here the majority of proposals is accepted, since the energy in the system can only be dissipated due to numerical integration errors.

Therefore, HMC has a much higher acceptance rate than MH.

Chapter 3

HistFactory

HistFactory (HF) is a tool for binned statistical analysis which is widely used in LHC physics to measure the consistency of collision events with theoretical predictions. It has been employed for the discovery of the Higgs Boson [4] and is used in searches for new physics [5] by research groups around the planet. The relationship between theoretical predictions and collision events is formalized as *statistical model* $p\,(\boldsymbol{x}\,|\,\boldsymbol{\theta})$. It describes the probability of observing data $\boldsymbol{x}$ given the model parameters $\boldsymbol{\theta}$. Typically, statistical models in High Energy Physics (HEP) are complex with many parameters. HF enables a standardized way to build parametrized probability density functions and infer parameter properties from it.

3.1 Formalism

Statistical models in HistFactory describe simultaneous measurements of disjoint *channels* c (binned distributions as subspace of all collision events), where we observe the event counts $\boldsymbol{n}$. In a particle detector several physical processes produce events in the selected channels. Hence, the total number of expected events[1] is the sum of all involved processes, the so called *samples*. This sample rates underlie variations and can be modified by the parameters $\boldsymbol{\theta}$. It is distinguished between *free* parameters $\boldsymbol{\eta}$ (e.g. the luminosity) and *constrained* parameters $\boldsymbol{\chi}$ that account for systematic uncertainties. In a frequentist setting, these constrained terms can be viewed as *auxiliary measurements* $\boldsymbol{a}$ which result together with the channel events $\boldsymbol{n}$ in the observations $\boldsymbol{x} = (\boldsymbol{n}, \boldsymbol{a})$. Equation (3.1) illustrates this parametrization

$$p\,(\boldsymbol{x}\,|\,\boldsymbol{\theta}) = p\,(\boldsymbol{n}, \boldsymbol{a}\,|\,\boldsymbol{\eta}, \boldsymbol{\chi}) \tag{3.1}$$

[1] this is often denoted as *event rate* since it used as input parameter to a Poisson distribution

The statistical model in HF consists of two parts – the *main model* of simultaneous measurements over multiple channels, and a *constrained term* that takes into account auxiliary measurements [6].

$$p\left(\boldsymbol{n},\boldsymbol{a}\,|\,\boldsymbol{\eta},\boldsymbol{\chi}\right) = \underbrace{\prod_{c\,\in\,\text{channels}}\prod_{b\,\in\,\text{bins}_c}\text{Pois}\left(n_{cb}\,|\,\nu_{cb}(\boldsymbol{\eta},\boldsymbol{\chi})\right)}_{\text{main model}} \underbrace{\prod_{\chi\,\in\,\boldsymbol{\chi}} c_\chi\left(a_\chi\,|\,\boldsymbol{\chi}\right)}_{\substack{\text{constraint terms}\\ \text{for “auxiliary measurements”}}} \tag{3.2}$$

For each bin b and channel c, the total event rate ν_{cb} is is the sum over all involved sample rates ν_{scb}.

$$\nu_{cb}(\boldsymbol{\eta},\boldsymbol{\chi}) = \sum_{s\,\in\,\text{samples}} \nu_{scb}(\boldsymbol{\eta},\boldsymbol{\chi})$$

The sample event rates ν_{scb} are determined by a *nominal rate* ν_{scb}^{0} and a set of multiplicative and additive *rate modifiers* $\boldsymbol{\kappa}(\boldsymbol{\theta})$ and $\boldsymbol{\Delta}(\boldsymbol{\theta})$, which are controlled by the model parameters $\boldsymbol{\theta} = (\boldsymbol{\eta},\boldsymbol{\chi})$ [6].

$$\nu_{cb}(\boldsymbol{\eta},\boldsymbol{\chi}) = \sum_{s\,\in\,\text{samples}} \underbrace{\left(\prod_{\kappa\in\boldsymbol{\kappa}} \kappa_{scb}(\boldsymbol{\eta},\boldsymbol{\chi})\right)}_{\text{multiplicative modifiers}} \left(\nu_{scb}^{0}(\boldsymbol{\eta},\boldsymbol{\chi}) + \underbrace{\sum_{\Delta\in\boldsymbol{\Delta}} \Delta_{scb}(\boldsymbol{\eta},\boldsymbol{\chi})}_{\text{additive modifiers}}\right) \tag{3.3}$$

The available modifiers in HistFactory are summarized in Table 3.1. Each modifier is defined for bin b, sample s and channel c and is controlled by at least one parameter $\theta \in \{\gamma,\alpha,\lambda,\mu\}$. By convention, bin-wise modifiers are denoted by γ and interpolation parameters with α. In contrast, luminosity λ and scale factors μ affect all bins equally.

Table 3.1: HistFactory modifiers and constraints [6].

Description	**Modification**	**Constraint Term c_χ**	**Input**
Uncorrelated Shape `shapesys`	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \text{Pois}\left(r_b = \sigma_b^{-2}\,|\,\rho_b = \sigma_b^{-2}\gamma_b\right)$	σ_b
Correlated Shape `histosys`	$\Delta_{scb}(\alpha) = f_p\left(\alpha\,|\,\Delta_{scb,\alpha=\pm1}\right)$	$\text{Normal}\left(a = 0\,|\,\alpha,\sigma = 1\right)$	$\Delta_{scb,\alpha=\pm1}$
Normalization Uncert. `normsys`	$\kappa_{scb}(\alpha) = g_p\left(\alpha\,|\,\kappa_{scb,\alpha\pm1}\right)$	$\text{Normal}\left(a = 0\,|\,\alpha,\sigma = 1\right)$	$\kappa_{scb,\alpha\pm1}$
MC Stat. Uncertainty `staterror`	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \text{Normal}\left(a_{\gamma_b} = 1\,|\,\gamma_b,\delta_b\right)$	$\delta_b^2 = \sum_s \delta_{sb}^2$
Luminosity `lumi`	$\kappa_{scb}(\lambda) = \lambda$	$\text{Normal}\left(l = \lambda_0\,|\,\lambda,\sigma_\lambda\right)$	λ_0, σ_λ
Normalization `normfactor`	$\kappa_{scb}(\mu_b) = \mu_b$		
Data-driven Shape `shapefactor`	$\kappa_{scb}(\gamma_b) = \gamma_b$		

The first five entries in Table 3.1 are constrained modifiers, defined by a constraint term $c_{\boldsymbol{\chi}}$ (right part in (3.2)) and an input parameter.

- **Uncorrelated Shape** modifiers affect each bin individually and hence the sample *shape*. They are applied to model uncorrelated background, with rate uncertainties δ_b for each bin. $\sigma_b = \delta_b / \nu_b$ is the *relative* uncertainty of the expected total event rate ν_b, and serves as input for the constraint term.

- **Correlated Shape** modifiers are additive to the nominal sampling rate $\nu^0_{scb}(\boldsymbol{\eta}, \boldsymbol{\chi})$ and are controlled by one nuisance parameter (NP). They use the interpolation function f_p to interpolate between sample distribution shapes represented by a *downward* variation `lo` $(\alpha = -1)$ and an *upward* variation `hi` $(\alpha = +1)$.

The remaining modifiers directly affect the sample rates and are therefore *shape-invariant*.

- The **Normalization Uncertainty** modifies the sample rate by a factor $\kappa_{scb}(\alpha)$ which is constructed as interpolation g_p between an upward `"hi"` and downward value `"lo"`, for instance `"data": {"hi":1.1, "lo":0.9}`

- **MC Statistical Uncertainty**. Many sample counts are derived from Monte Carlo (MC) datasets which necessarily carry uncertainties. These uncertainties are modelled by a set of bin-wise scale factors γ_b while each bin is constraint by a Normal distribution centered at *one*.

- **Luminosity.** Sample rates that are derived from *theory predictions* are scaled to the integrated luminosity of the observed data. As this value is derived from measurements that itself carry uncertainties, the luminosity uncertainty σ_λ needs to be specified.

- **Unconstrained Normalization** modifiers scale the sample event rate by a free parameter μ. Such parameter are frequently the parameter of interest (POI) of a given measurement.

- **Data-driven Shape** modifiers are unconstrained, bin-wise multiplicative parameters. They are introduced to support data-driven estimation of sample rates (e.g. multijet backgrounds).

3.2 Workspaces

Statistical models in HF are described in plain-text `JSON` format. This scheme fully specifies the model structure as well as necessary constrained data in a single document and hence is implementation independent. This `JSON` files represent a *workspace* which consists of *channels*, *measurements* and *observations.*

- **Channels** are certain regions in the space of collision events, each described by a statistical model. The regions are chosen to be disjoint and typically contain signal regions (SR) and control regions (CR) for a certain particle decay of interest.

  The statistical model for a channel is constructed by a list of *samples* (models of involved physical processes) which consists of the predicted event rates and a set of modifiers (see Table 3.1).

- **Measurements** are a small subset of all model parameters – the parameter of interest (POI). The measurement scheme in the `JSON` file can be configured with the initial value, interval bounds or `auxdata` for the parameter.

- **Observations** are the actual observed events for each bin in all channels.

For a detailed description of the HF `JSON` format, the reader is referred to the `pyhf` documentation [6].

Models in this thesis. Since real world examples are too complex to benchmark in acceptable time, this work mainly uses "toy" workspaces with a single channel. All model specifications utilized in this work are listed in Appendix A.

3.3 HistFactory Implementations

Currently, HistFactory is available in three different programming languages

- a C++ implementation,
- a Python version `pyhf` [7], and
- a Julia implementation `LiteHF` [8]

In chapter 5, this work employs `pyhf` and `LiteHF` for run-time benchmarks. To verify the equivalence of `pyhf` and `LiteHF`, the log posterior measure is evaluated for both implementations. The log likelihood in Python for a parameter vector θ is computed by the `logpdf` of the `main_model`, i.e.

```
def llh(param: np.ndarray) -> float
    """pyhf log likelihood from the main_model."""
    return model.main_model.logpdf(main_data, param)
```

In LiteHF one can access the log likelihood function by

```
llh = pyhf_loglikelihoodof(pyhfmodel.expected, pyhfmodel.observed)
```

For visualization purposes, the verification step is illustrated for the 2D 2_bin_corr model in section A.2. The prior vector is chosen to be

$$p(\boldsymbol{\theta}) = \begin{bmatrix} \text{Uniform}(0,\,5) \\ \text{Normal}(0,\,1) \end{bmatrix} \tag{3.4}$$

and the log posterior measure is evaluated for 10^5 points $\boldsymbol{x} \sim p(\boldsymbol{\theta})$ with equal initial seed. The samples coincidence for both implementations up to numerical precision. In Figure 3.1 1000 randomly chosen data points are picked out and illustrated for pyhf (blue) and LiteHF (orange).

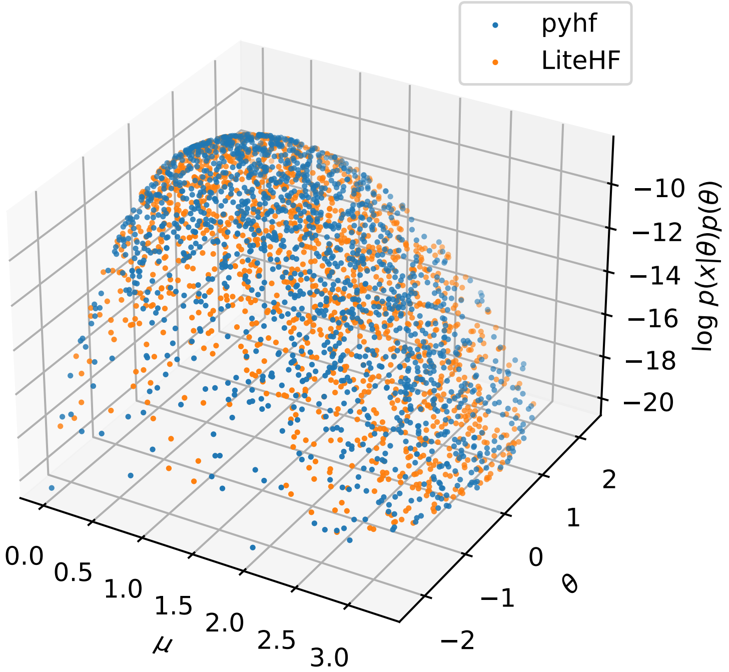


Figure 3.1: Verify the equivalence of the pyhf and LiteHF implementation by evaluating the log posterior measure $\log\big(p\,(\boldsymbol{x}\,|\,\boldsymbol{\theta})\,p(\boldsymbol{\theta})\big)$ at 1000 random points $\boldsymbol{x} \sim p(\boldsymbol{\theta})$ for the 2_bin_corr model (see section A.2). The 2D prior vector is set to $p(\mu, \theta) = \big[\text{Uniform}(0, 5),\, \text{Normal}(0, 1)\big]^T$.

Chapter 4

Bayesian Inference with HistFactory

While the majority of statistical results in LHC physics are obtained by a frequentist setting this chapter formalizes a Bayesian approach for parameter inference with HistFactory. First, it introduces the Bayesian Analysis Toolkit (BAT) which is used for MCMC sampling. Second, it derives a formalism to obtain prior distributions from HF models. Third, it discusses implementation details of `batty` – a Python-to-Julia interface to execute BAT functions in Python. The chapter ends by a review of the implemented Bayesian inference pipeline for the HF models in Appendix A.

4.1 BAT

The Bayesian Analysis Toolkit (BAT) [9] is a multi-purpose software for Bayesian statistical inference, written in Julia [10]. The software design of BAT is modular and code can be written in a clear and easy fashion. At the same time, it achieves a high runtime performance, which is comparable to `C` or `C++` implementations. Likelihoods, priors and posteriors are all expressed as densities, represented by the type `AbstractDensity`[1]. To operate on densities, BAT offers function like `bat_sample()`, `bat_findmode()` or `bat_integrate()`.

Currently, BAT provides two MCMC sampling algorithms, Metropolis-Hastings (MH) and Hamiltonian Monte Carlo (HMC) which are both applied in this work. Both algorithms have been discussed in section 2.2. For further details the reader is referred to [9].

[1] Even density-like objects like log likelihood (llh) functions or histograms are automatically converted to subtypes of `AbstractDensity`

4.2 Priors for HistFactory Models

This section derives the procedure of generating priors for HF models. As discussed in section 3.1, statistical models in HistFactory consist of a *main model* and *constrained terms*. Bayesian inference for the parameters $\boldsymbol{\theta} = (\boldsymbol{\eta}, \boldsymbol{\chi})$ only uses the likelihood of the main model in (3.2)

$$p_{\text{main}}\left(\boldsymbol{n} \,|\, \boldsymbol{\theta}\right) \;\equiv\; p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right) \;=\; \prod_{c \,\in\, \text{channels}} \; \prod_{b \,\in\, \text{bins}_c} \text{Pois}\left(n_{cb} \,|\, \nu_{cb}(\boldsymbol{\eta}, \boldsymbol{\chi})\right) \tag{4.1}$$

while the event rate $\nu_{cb}(\boldsymbol{\eta}, \boldsymbol{\chi})$ is still computed according to (3.3). Hence, auxiliary measurements $\boldsymbol{a}$ (see section 3.1) does not appear in the main model $p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right)$ and $\boldsymbol{x}$ corresponds to the observed events $\boldsymbol{n}$.

In contrast, priors make use of the auxiliary measurements $\boldsymbol{a}$ and are derived from the constraint terms $c_\chi\left(a_\chi \,|\, \boldsymbol{\chi}\right)$ in Table 3.1. To formalize this step further, we rewrite Bayes' rule:

$$\begin{aligned}
p\left(\boldsymbol{\theta} \,|\, \boldsymbol{x}\right) \quad &= \quad \frac{p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right) p(\boldsymbol{\theta})}{p(\boldsymbol{x})} \\
&\equiv \quad \frac{p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right) p\left(\boldsymbol{\theta} \,|\, \boldsymbol{a}\right)}{p(\boldsymbol{x})} \\
&= \quad \frac{p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right)}{p(\boldsymbol{x})} \frac{p\left(\boldsymbol{a} \,|\, \boldsymbol{\theta}\right) p_{\text{ur}}(\boldsymbol{\theta})}{p(\boldsymbol{a})} \\
&= \quad \frac{p_{\text{main}}\left(\boldsymbol{x} \,|\, \boldsymbol{\theta}\right)}{p(\boldsymbol{x})} \frac{p\left(\boldsymbol{a} \,|\, \boldsymbol{\theta}\right) p_{\text{ur}}(\boldsymbol{\theta})}{p(\boldsymbol{a})}
\end{aligned}$$

The prior $p\left(\boldsymbol{\theta} \,|\, \boldsymbol{a}\right)$ can be interpreted as *posterior* computed from the likelihood of auxiliary measurements $p\left(\boldsymbol{a} \,|\, \boldsymbol{\theta}\right)$, an ur-prior $p_{\text{ur}}(\boldsymbol{\theta})$ and a normalization constant. The last line illustrates the frequentist approach of incorporating auxiliary data – the *green* highlighted part is a product of likelihoods and corresponds to the statistical model in (3.2).

However, as discussed in section 2.2, computing posteriors

$$p\left(\boldsymbol{\theta} \,|\, \boldsymbol{a}\right) = \frac{p\left(\boldsymbol{a} \,|\, \boldsymbol{\theta}\right) p_{\text{ur}}(\boldsymbol{\theta})}{p(\boldsymbol{a})} \tag{4.2}$$

analytically is only possible for special choices of likelihood and prior. One of these special cases are *conjugate priors*: If the posterior distribution $p\left(\boldsymbol{\theta} \,|\, \boldsymbol{a}\right)$ is in the same distribution family as the prior distribution $p_{\text{ur}}(\boldsymbol{\theta})$, then prior and posterior are called *conjugate distributions* and $p_{\text{ur}}(\boldsymbol{\theta})$ is called *conjugate prior* for the likelihood $p(\boldsymbol{a}|\theta)$. This method is applied in this work and elaborated in the next section.

4.2.1 Conjugate Priors

When using conjugate priors, we necessarily incorporate information about the parameter space due to the ur-prior. Nevertheless, to justify this approach we choose a *vague* ur-prior so that the updated prior $p\left(\boldsymbol{\theta} \,|\, \boldsymbol{a}\right)$ is mainly determined by data likelihood $p\left(\boldsymbol{a} \,|\, \boldsymbol{\theta}\right)$.

HistFactory basically uses two types of likelihoods in the constraint terms: On the one hand Poisson constrains for `shapesys` modifiers, on the other hand Normal distributions for the remaining constraint terms (see Table 3.1). Below, conjugate priors for both likelihood types are derived.

Poisson distributed Likelihoods

Conjugate priors for Poisson likelihoods are *gamma*-distributed $p(\theta) \equiv \text{Gamma}(\alpha, \beta)$. The pdf for a random variable $X \sim \text{Gamma}(\alpha, \beta)$ in *shape-rate* parameterization is given as

$$p(x;\, \alpha, \beta) = \frac{x^{\alpha-1} e^{-\beta x} \beta^{\alpha}}{\Gamma(\alpha)} \qquad \text{for } x > 0 \tag{4.3}$$

where $\alpha, \beta > 0$ are the *shape* and *rate* parameter, and $\Gamma(\alpha)$ the *gamma function.* The expected value and variance of $X \sim \text{Gamma}(\alpha, \beta)$ is computed to

$$\mathbb{E}[X] = \frac{\alpha}{\beta} \qquad \text{and} \qquad \text{Var}[X] = \frac{\alpha}{\beta^2} \tag{4.4}$$

The Poisson likelihood of observing N *independent* and *identically distributed* events is

$$p\left(\boldsymbol{x} \,|\, \theta\right) = \prod_{i=1}^{N} \text{Pois}\left(x_i \,|\, \theta\right) \qquad \text{with} \qquad \text{Pois}\left(x_i \,|\, \theta\right) = \frac{\theta^{x_i} e^{-\theta}}{x_i!} \tag{4.5}$$

According to Bayes' theorem (2.1) the posterior is calculated to be

$$p\left(\theta \,|\, \boldsymbol{x}\right) = \frac{p\left(\boldsymbol{x} \,|\, \theta\right) p(\theta)}{p(\boldsymbol{x})} \propto p\left(\boldsymbol{x} \,|\, \theta\right) p(\theta) \tag{4.6}$$

while we neglect the constant evidence term $p(\boldsymbol{x})$. Explicitly computing the posterior using (4.5) and (4.3) for priors $p(\theta) = \text{Gamma}(\alpha, \beta)$ yields

$$\begin{aligned} p\left(\theta \,|\, \boldsymbol{x}\right) &\propto \prod_{i=1}^{n} \frac{\theta^{x_i} e^{-\theta}}{x_i!} \cdot \frac{\theta^{\alpha-1} e^{-\beta\theta} \beta^{\alpha}}{\Gamma(\alpha)} \\ &\propto \theta^{x_1 + x_2 + \ldots + x_N} e^{-\theta N} \cdot \theta^{\alpha-1} e^{-\beta\theta} \\ &= \theta^{N\bar{x} + \alpha - 1} e^{-(\beta + N)\theta} \end{aligned} \tag{4.7}$$

In line 2 we skipped the constant terms $\Gamma(\alpha)$, $x_i!$, β^α and in line 3 the sample mean

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.8}$$

was introduced to simplify notation. Comparing (4.7) with the Gamma pdf in (4.3) and using the argument that the total area under a probability density function must evaluate to *one*, we can conclude that the posterior $p\,(\theta\,|\,\boldsymbol{x})$ is again a Gamma distribution

$$p\,(\theta\,|\,\boldsymbol{x}) = \text{Gamma}\big(N\bar{x}+\alpha,\, \beta+N\big) \tag{4.9}$$

with the updated parameters

$$\begin{aligned} \alpha' &= N\bar{x}+\alpha \\ \beta' &= \beta+N \end{aligned} \tag{4.10}$$

Normally distributed Likelihoods

Conjugate priors for normally distributed likelihoods are *normally* distributed likewise. To proof this, we consider the likelihood

$$p\,(\boldsymbol{x}\,|\,\theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_x^2}} \exp\left(-\frac{(x_i-\theta)^2}{2\sigma_x^2}\right) \propto \exp\left(-\frac{\sum_{i=1}^{N}(x_i-\theta)^2}{2\sigma_x^2}\right) \tag{4.11}$$

of N independent observations $\boldsymbol{x} = \big\{x_i\big\}_{i=1}^{N}$ with same mean θ and variance σ_x^2. During the derivation it is assumed that the likelihood variance σ_x^2 is known.
For normally distributed priors

$$p(\theta) = \text{Normal}(\theta_0,\, \sigma_\theta^2) \tag{4.12}$$

with mean θ_0 and variance σ_θ^2, the posterior is again computed according to (4.6). Once more we neglect constant terms and after elementary calculus we obtain the expression

$$\begin{aligned} p\,(\theta\,|\,\boldsymbol{x}) &\propto \exp\left(-\frac{\sum_{i=1}^{N}(x_i-\theta)^2}{2\sigma_x^2}\right)\cdot\exp\left(-\frac{(\theta-\theta_0)^2}{2\sigma_\theta^2}\right) \\ &= \exp\left[\frac{-\sum_{i=1}^{N}x_i^2+2\theta N\bar{x}-N\theta^2}{2\sigma_x^2} - \frac{\theta^2-2\theta\theta_0+\theta_0^2}{2\sigma_\theta^2}\right] \\ &\propto \exp\left[\frac{2\theta N\bar{x}-N\theta^2}{2\sigma_x^2} - \frac{\theta^2-2\theta\theta_0}{2\sigma_\theta^2}\right] \end{aligned} \tag{4.13}$$

while we applied the sample mean $\bar{x}$ (Equation (4.8)) in line 2 to simplify notation. Rewriting (4.13) in powers of θ yields

$$p\left(\theta \,|\, \boldsymbol{x}\right) \quad \propto \quad \exp\left[-\frac{\theta^2}{2}\underbrace{\left(\frac{1}{\sigma_\theta^2}+\frac{N}{\sigma_x^2}\right)}_{1/\sigma_\theta'^2}+\theta\left(\frac{\theta_0}{\sigma_\theta^2}+\frac{N\bar{x}}{\sigma_x^2}\right)\right] \tag{4.14}$$

Equation (4.14) is proportional to a Normal distribution which can be verified by introducing the variables

$$\sigma_\theta'^2=\left(\frac{1}{\sigma_\theta^2}+\frac{N}{\sigma_x^2}\right)^{-1} \qquad \text{and} \qquad \theta'=\sigma_\theta'^2\left(\frac{\theta_0}{\sigma_\theta^2}+\frac{N\bar{x}}{\sigma_x^2}\right) \tag{4.15}$$

By inserting these variables and multiplying (4.14) by a normalization constant

$$p\left(\theta \,|\, \boldsymbol{x}\right) \propto \exp\left[-\frac{\theta^2}{2\sigma_\theta'^2}+\frac{\theta\theta'}{\sigma_\theta'^2}-\frac{\theta'^2}{2\sigma_\theta'^2}\right]=\exp\left[-\frac{(\theta-\theta')^2}{\sigma_\theta'^2}\right]$$

we can conclude[2] that the posterior is normally distributed with the parameters in (4.15)

$$p\left(\theta \,|\, \boldsymbol{x}\right)=\text{Normal}(\theta',\sigma_\theta'^2)$$

4.2.2 Implementation

After deriving the parameter updates for the conjugate priors, this section describes implementation details. Generating priors from HF models is implemented in the Python package `priorhf` which was developed during this work.

For all NP, auxiliary data in HF is stored in a member variable `auxdata`. The parameter order for `pyhf` models is determined by the *parameter map*

```
    par_map = model.config.par_map.values()
```

which contains all required parameter information (`auxdata`, parameter `name`, standard deviation `sigma` (only for `staterror` and `lumi` modifiers)) as well as the `slice` in which the parameter is located.

- For `shapesys` modifiers (uncorrelated shape), the `auxdata` entry corresponds to the rate parameter $r_b = \sigma_b^{-2}$ (Table 3.1, row 1), where $\sigma_b = \delta_b/\nu_b$ is the relative uncertainty of the expected total event rate ν_b [6]. This uncertainties are modeled by a Poisson $\tilde{\gamma}_b \sim \text{Pois}(r_b)$ with expectation $\mathbb{E}[\tilde{\gamma}_b] = r_b$. As *initial* prior (ur-prior)

[2] Again, we use the argument that the area under a pdf must be equal to *one*.

we choose $p_{\mathrm{ur}}(\tilde{\gamma}_b) = \mathrm{Gamma}(1, \beta)$, with $\alpha = 1$ and compute the rate parameter β of the initial prior so that the expectation of likelihood and initial prior match, i.e.

$$\beta = \frac{\alpha}{r_b} \tag{4.16}$$

Then, we perform the parameter update in (4.10) for $N = 1$, $\bar{x} = r_b$ and obtain an up-scaled prior

$$p\,(\tilde{\gamma}_b \,|\, \boldsymbol{a}) = \mathrm{Gamma}(\alpha', \beta') \quad \text{with} \quad \mathbb{E}[\tilde{\gamma}_b] = r_b$$

In a final step, we re-scale the density $p\,(\tilde{\gamma}_b \,|\, \boldsymbol{a})$ so that it has an expected value of 1 by multiplying β' with r_b. To summarize, the prior distribution for a `shapesys` parameter γ_b is given by

$$\gamma_b \sim \mathrm{Gamma}(r_b + 1, r_b + 1) \quad \text{with} \quad \mathbb{E}[\gamma_b] = 1 \tag{4.17}$$

which is derived from an ur-prior $p_{\mathrm{ur}}(\tilde{\gamma}_b) = \mathrm{Gamma}(\alpha = 1,\ \beta = 1/r_b)$.

The priors for the `2_bin_uncorr` model[3] are illustrated in Figure 4.1a). The relative uncertainties σ_b for bin 1 and 2 are 10 % and 20 % (see section A.1). This results in a tighter prior distribution for γ_1 and a broader spread for the parameter γ_2.

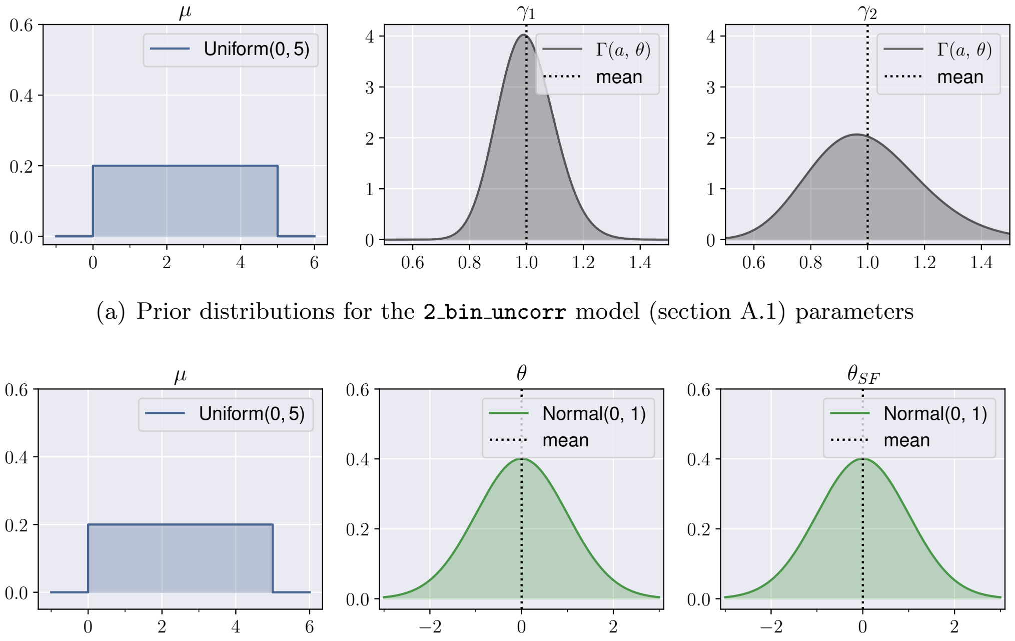
- The modifiers `histosys` and `normsys` are constrained by a zero-mean Normal distribution with variance 1 (see Table 3.1). The corresponding values in the `auxdata` vector are zeros. The initial prior is chosen as zero-mean Normal distribution with variance $\sigma_\theta^2 = 100$. After the parameter update in (4.15) (with $N = 1$, $\bar{x} = \theta_0 = 0$ and $\sigma_x^2 = 1$) we obtain the prior distribution

$$p\,(\theta \,|\, \boldsymbol{a}) = \mathrm{Normal}\Big(0,\ \sigma_\theta'^2 = \frac{100}{101}\Big) \tag{4.18}$$

Priors for the `4_bin` model[4] are illustrated in Figure 4.1b) as standard normal distribution for the parameters θ and θ_{SF}.

[3] a 2-bin model with uncorrelated background

[4] 3D parameter space with `normsys` and `histosys` modifiers



(a) Prior distributions for the `2_bin_uncorr` model (section A.1) parameters

(b) Prior distributions for the `4_bin` model (section A.3) parameters

Figure 4.1: Conjugate priors for the `2_bin_uncorr` and `4_bin` model. The signal strength parameter μ (*left*) is uniformly distributed for both models. The dashed line represents the mean of the Gamma and Normal distribution. Priors for the 2 dimensional `2_bin_corr` model (section A.2) are equal to the first two distributions of the `4_bin` model.

- `staterror` and `lumi` modifiers are also constrained by a Normal distribution (see Table 3.1). Their mean values are 1 or λ_0 and are stored in `auxdata`. The ur-prior variance $\sigma_\theta^2 = 100$ is again set to 100 and the mean θ_0 corresponds to the value in `auxdata`. The likelihood variance σ_x^2 is computed from the standard deviation `sigma` in the parameter map. Once more the parameter update is calculated according to Equation (4.15) with $N = 1$.

Parameter of interest (POI) (like the signal strength μ) are unconstrained and modeled as *uniform* distribution from 0 to 5. All prior distributions are implemented as `NamedTupleDist` [11] of densities from the `Distributions.jl` package [12]. This step is necessary since priors directly interface to BAT which requires this type. Before converting priors to a `NamedTupleDist`, priors need to be represented as `namedtuple`[5] to preserve the parameter order[6].

[5] `from collections import namedtuple`

[6] Passing Python dictionaries to the `NamedTupleDist` constructor interchanges the parameter order even for Python versions $\geq$ `3.7`. This is because when a Python `dict` is converted to a Julia `Dict`, the order is not maintained.

This step is illustrated in the snipped below, where `names` is a `list` of the parameter names and `param` a `list` of `Distributions.jl` densities.

```
# Makro to convert keys of a dict as parameter string for namedtuple
key_str = lambda d: ' '.join(list(d.keys()))

prior_specs = dict(zip(names, param))
p = namedtuple('Prior', key_str(prior_specs))(**prior_specs)
priors = jl.unshaped(jl.NamedTupleDist(p))
```

In the last line the `unshaped` [11] function is applied to obtain a *flat* prior vector.

4.3 batty – BAT to Python Interface

The complete pipeline for calling BAT functions from Python is implemented in `batty`, the BAT to Python interface which is developed here [13]. During this work, `batty` was steadily tested an improved with several contributions. An old version of `batty` used `PyCall.jl` [14] to execute Julia functions from Python which had one major drawback: The complete Python environment needed to be compiled each time `batty` was imported which takes about 2 minutes. The current version of `batty` employs `PythonCall.jl` [15] that reduces the total import time to about 10 seconds and speed up inference in `batty` by a factor of 4.

`PythonCall.jl` is suited to call Python functions in Julia and vice versa [15]. The corresponding Python package is called `juliacall` and is used in Python as

```
from juliacall import Main as jl
jl.seval('using BAT, Distributions')
```

The second line imports the Julia modules `BAT` [9] and `Distributions` [12] which enables the access to Julia types or functions using the prefix `jl.<type/function>`.

4.3.1 Likelihoods for pyhf Models

All functionalities for computing `pyhf` likelihoods are implemented in the Python package `pyhf_llh`. For a fast computation of the log likelihood (llh) this work makes use of the `jax`-backend of `pyhf`[7]. `jax` [16] exploits auto differentiation (Autograd) and accelerated linear algebra (XLA) to speed up computations and automatically compute gradients. It is developed by Google and is widely used in Machine Learning frameworks.

[7] `pyhf` can be configured to use different *tensor backends* (NumPy, PyTorch, TensorFlow and JAX) for all numerical computations.

This work aims to implement MCMC sampling in Python for Metropolis-Hastings (MH) and Hamiltonian Monte Carlo (HMC) sampling. Since HMC requires the likelihood gradient, these values need to be passed from Python to Julia. This is accomplished by using a `PyCallDensityWithGrad` wrapper that contains pointers to the log likelihood (llh) function `logf` as well as a pointer to the llh function with gradient `valgradlogf`, i.e.

```julia
struct PyCallDensityWithGrad <: BAT.BATDensity
    logf::PythonCall.Py
    valgradlogf::PythonCall.Py
end
```

In addition, several steps have to be considered to pass gradient from Python to Julia. For details see `pybat.jl` in [13]. The `pyhf` likelihood is implemented as functor that returns the `llh` function for a given HF workspace

```python
def pyhf_llh(ws: str) -> Callable:
    """Returns the log likelihood function from the pyhf workspace 'ws'."""
    model, main_data, _ = load_pyhf(ws)
    logpdf = model.main_model.logpdf
    def llh(param: np.ndarray) -> jnp.DeviceArray:
        """pyhf log likelihood from the main_model."""
        return logpdf(main_data, param)
    return jit(llh)
```

To enhance the computation time, we employ the just-in-time compilation (`jit`) by `jax` which returns highly optimized machine code for computing the log likelihood efficiently. The second entry for the `PyCallDensityWithGrad` wrapper is the `llh_with_grad` function

```python
def pyhf_llh_with_grad(ws: str) -> Tuple[Callable, Callable]:
    """Returns the Tuple (llh, llh_with_grad) for the pyhf workspace 'ws'"""
    # jit llh and compute grad
    llh = pyhf_llh(ws)
    llh_grad = jit(grad(llh))
    # convert to float and catch out 'nan' values of pyhf
    llh_float = lambda x: -inf if jnp.isnan(llh(x)) else float(llh(x))

    def llh_with_grad(param: np.ndarray) -> Tuple[float, np.ndarray]:
        """Convert jax.DeviceArray to (float, np.float64)"""
        return (llh_float(param), np.array(llh_grad(param), dtype=np.float64))

    return (llh_float, llh_with_grad)
```

which returns a tuple `(llh_float, llh_with_grad)` required by `PyCallDensityWithGrad`. The gradient of the likelihood is computed by the `grad()` function from `jax`. To ensure that the return types can be converted with `PythonCall.jl`, `jax.DeviceArray` types need to be converted to `float` types. In addition it is necessary to catch `nan` values of `pyhf` and replace them by `-inf` since `bat_sample()` does not work for `nan` values.

4.3.2 pyhf Benchmarks

The effects of using the `jax` backend instead `numpy` are discussed below for the HF models `2_bin_corr` and `2_bin_uncorr` (Appendix A). The compile time of `bat_sample()` for both cases is shown in Figure 4.2 (*left*). Using the `jax` backend slightly reduces the compile time for both models. In contrast the memory usage during compilation increases by about 10 % (Figure 4.2 *right*) for the `jax` backend.

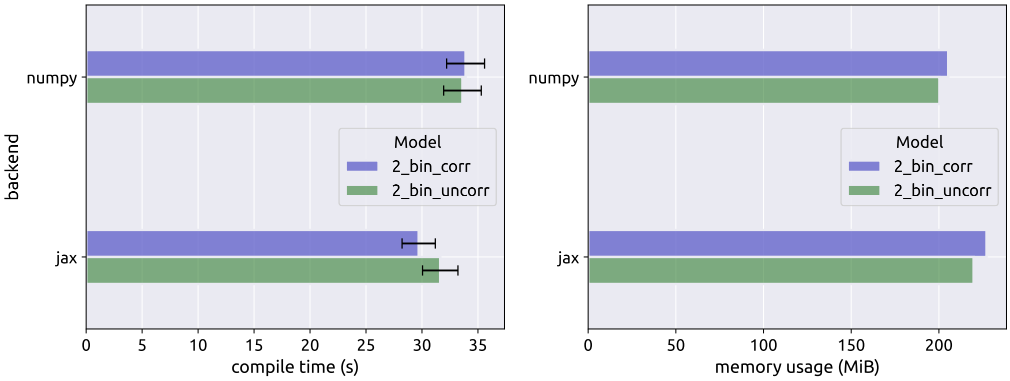
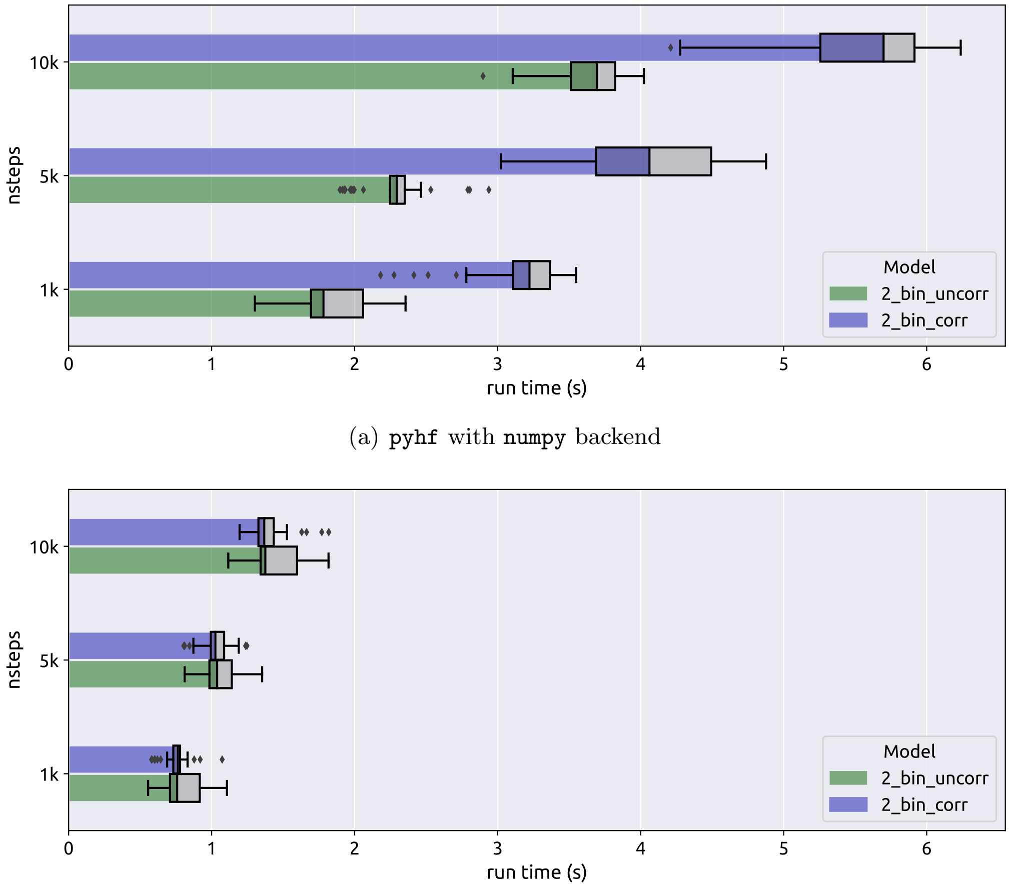


Figure 4.2: Compile time and memory usage while compiling `bat_sample()` with `jax` and `numpy` backend. Compile times typically fluctuate between ±5 %.

However, more important is the run-time performance. Figure 4.3a) shows the run-time with `numpy` backend for different MCMC steps in `bat_sample()`. The run-time statistics is visualized as box-plot as described in Appendix B. For the `numpy` setting, the `2_bin_corr` is about two times *slower* than the `2_bin_uncorr` model by performing `1k`, `5k` or `10k` MCMC `nsteps`. With `jax` backend, the total run time can be *reduced* by a factor of 3 to 5 (see Figure 4.3b). In addition the run-time of both models is roughly the same using the `jax` backend.

Benchmarks for directly evaluating the llh for `jax` and `numpy` are given in Figure B.2.

Unless stated differently this work uses the `jax` backend for evaluating the log likelihood.



(a) `pyhf` with `numpy` backend

(b) `pyhf` with `jax` backend

Figure 4.3: Run-time performance of `bat_sample()` once using the `numpy` backend and once with `jax` backend. The run-time is evaluated for the models `2_bin_corr` and `2_bin_uncorr` for different MCMC `nsteps`.

4.4 Bayesian Inference Examples

After setting up the priors and likelihood in section 4.2 and section 4.3, the complete Bayesian inference pipeline is verified for the HF models in Appendix A. As an example, the sampled posterior distribution for the `4-bin` model is illustrated in Figure 4.4 as corner plot [17] for `100k` MCMC steps.

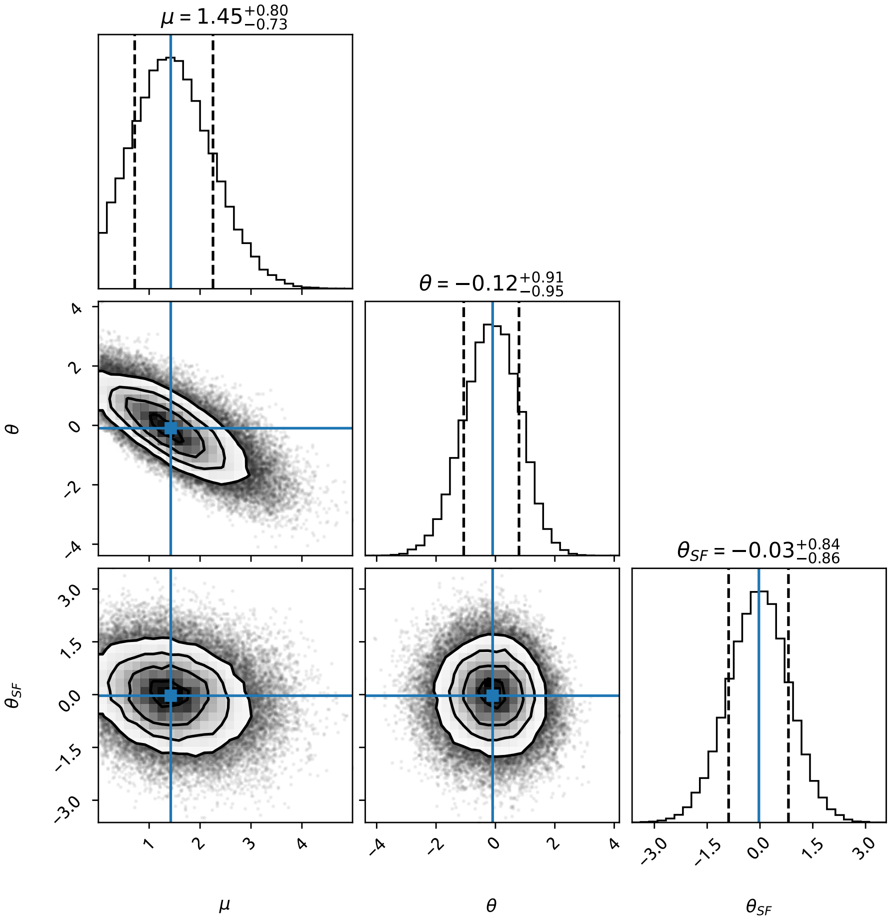


Figure 4.4: Corner plot [17] for the `4_bin` Model (section A.3) using `100k` HMC steps. The parameter mode is indicated by *blue* lines and the *dashed* lines represent the 1σ quantile of the sampled distributions.

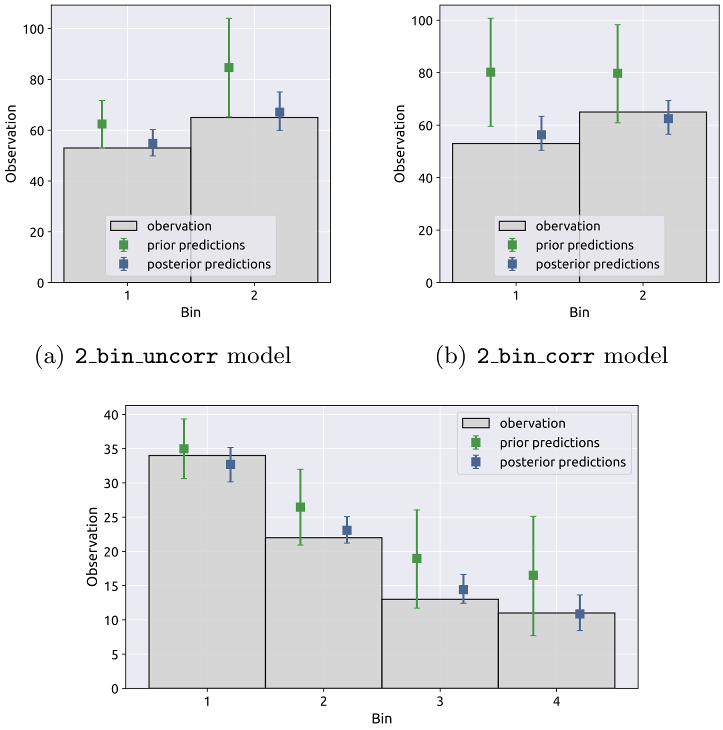
A snipped for sampling this posterior in Python is given below

```
likelihood = jl.PyCallDensityWithGrad(llh, llh_grad)
prior = make_prior('path/to/model')
posterior = jl.BAT.PosteriorMeasure(likelihood, prior)

samples = jl.bat_sample(posterior, method).result
```

where `method` is the MCMC sampling algorithm – Metropolis-Hastings or HMC.

The sampled posteriors are verified by a *posterior predictive check*, which is depicted in Figure 4.5 for all three models. All models sufficiently represent the observations with a reduced variance compared to the prior predictions.



(a) `2_bin_uncorr` model

(b) `2_bin_corr` model

(c) `4_bin` model

Figure 4.5: Prior and posterior predictive checks all models in Appendix A. The dot represents the median of the predicted output and the error bars indicate the 1σ quantile.

Chapter 5

Benchmarking the Python-Julia Pipeline against the Julia Implementation

This chapter compares the run-time performance of `batty` (section 4.3) with a pure-Julia implementation of Bayesian inference with `LiteHF` [8]. More specific, the following three implementations are evaluated

- **Python + BAT** (visualized in *blue*)

  Uses the Python-Julia bridge `batty` for Bayesian inference on HF models in Python.

- **Julia +** `LiteHF` (visualized in *green*)

  The complete inference chain is written and executed in Julia by using `LiteHF` [8], the Julia implementation of HF.

- **Julia +** `pyhf` (visualized in *red*)

  The whole code runs in Julia but uses the Python likelihood from `pyhf` for inference. Executing Python code from Julia is accomplished by the `PythonCall.jl` module [15]. The `pyhf` likelihood is implemented in the `pyhf_llh` package that can be embedded in Julia by

  ```
  llh = pyimport("pyhf_llh")
  llh_func, llh_with_grad = llh.pyhf_llh_with_grad("path/to/model")
  ```

Run-times benchmarks are measured for both, MH and HMC based on two HF models.

5.1 Benchmark Setup

For each implementation the run-time of `bat_sample()` is measured. Therefore, `bat_sample()` takes two arguments, the `posterior` to be sampled and the MCMC `method` to be used.

```
# Metropolis Hastings
method = BAT.MCMCSampling(mcalg=BAT.MetropolisHastings(), nsteps=nsteps, nchains=2)
# or Hamiltonian MC
method = BAT.MCMCSampling(mcalg=BAT.HamiltonianMC(), nsteps=nsteps, nchains=2)

samples = bat_sample(posterior, method).result
```

The number of chains `nchains` is set to 2 for all benchmarks. The *complexity* of the MCMC sampling is controlled by the number of `nsteps`. It is *not* considered whether all chains converge or not.

In order to ensure the reproducibility of benchmarks, the following steps are taken into account

- Code that is executed by `bat_sample()` – like computing the log likelihood – does not use global variables. Accessing global variables inside a function increases the run-time in Julia and Python.
- Since Julia code is pre-compiled before execution, a distinction is made between the *compile time* and *run-time* of `bat_sample()`. The proportion of both times is illustrated in Figure 5.1 for the `2_bin_corr` model for sampling 10k MCMC steps. The compile time accounts for more than 90 % of the total execution time in all implementations and has a uncertainty of ±4 %. Compiling `bat_sample()` in Python takes about twice as long as in Julia (see Figure 5.1 *left*).
- Benchmarks in Julia are obtained using the `BenchmarkTools.jl` module [18]. To measure "long" run-times sufficiently, the `@benchmarkable` makro is configured as

  ```
  b = @benchmarkable bat_sample(posterior, mcalg) setup=(mcalg=$method)
  sec = bootstrap_sec()
  res = run(b, samples=samples, seconds=maximum([sec, 40]))
  ```

  where the required time `sec` is bootstrapped from previous run-time measurements.
- All run-time benchmarks are written in Python or Julia scripts, since Jupyter Notebooks have lots of overhead.
- The benchmarks are carried out on a `Linux` computer with a `Intel(R) Core(TM) i7-7700HQ` CPU.

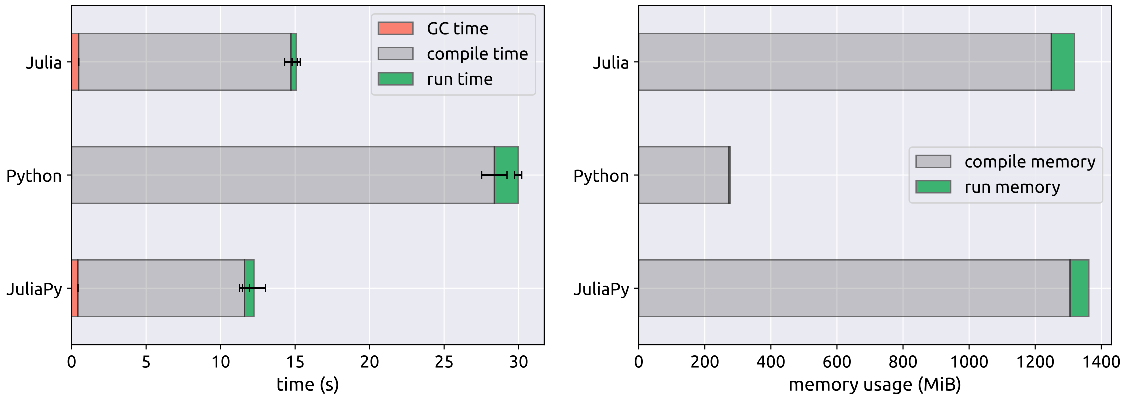


Figure 5.1: Total execution time and memory usage of `bat_sample()` for the `2_bin_corr` model with 10k steps. The *red*, *gray* and *green* portion is the garbage collector (GC) time, the compile time and run time respectively. Compiling `bat_sample()` in Python takes about twice as long as in Julia. In contrast, the memory usage during compilation in Python is only one sixth of the memory usage in Julia. (The run-time memory usage in Python can't be measured sufficiently and hence is neglected in the right plot. Moreover, the GC time can not be evaluated in Python.)

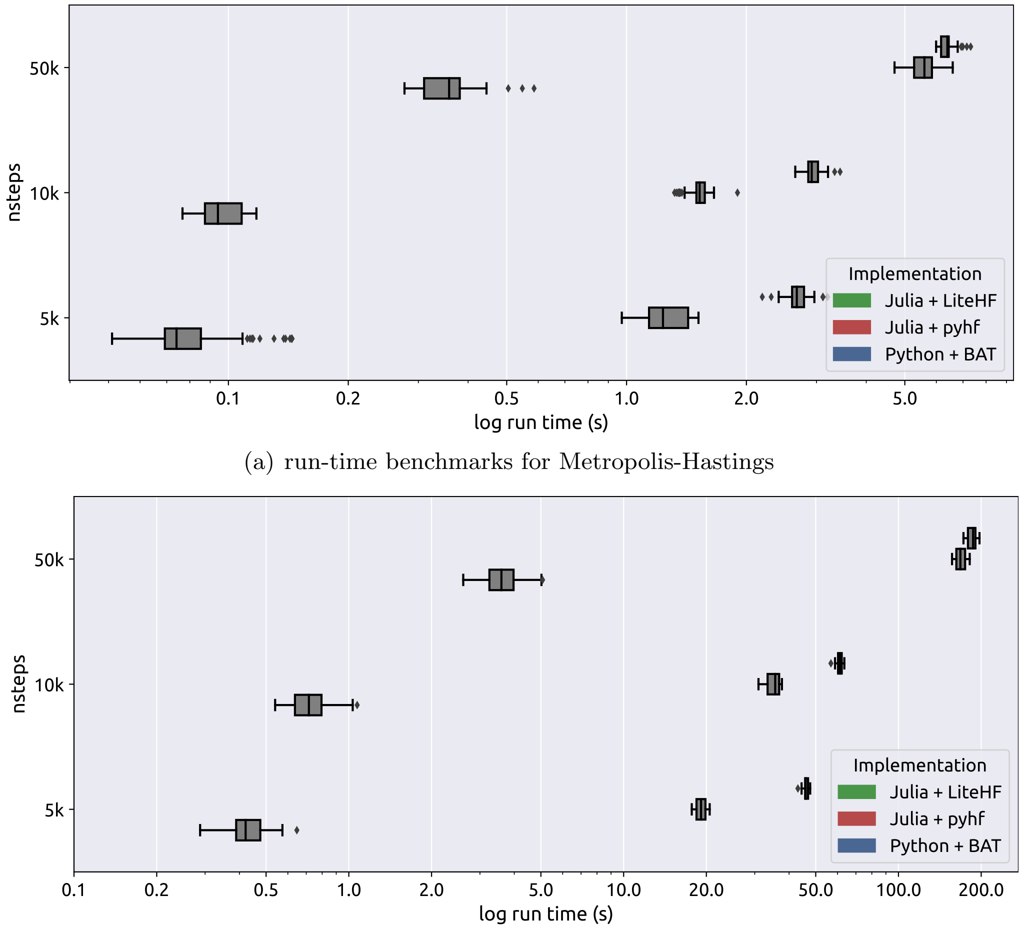
5.2 Benchmarks

5.2.1 `2_bin_corr` Model

First, the run-time of the `2_bin_corr` model (with a 2D parameter space) is evaluated for different MCMC `nsteps`. The results are visualized in Figure 5.2a) for Metropolis Hastings and in Figure 5.2b) for Hamiltonian MC. The run-time on the x-axis is plotted logarithmically and the numerical values are summarized Table 5.1. For confirmation, the run-time benchmarks for the `4_bin` model is additionally given in Appendix C.

Table 5.1: Summary of run-time benchmarks in Figure 5.2. The values represent the median run-time with 1σ standard deviation without outliers for `5k`, `10k` and `50k` `nsteps`.

		5k	**10k**	**50k**
Method	**Implementation**		**run-time** (s)	
MH	Julia + `LiteHF`	0.07 ± 0.01	0.09 ± 0.01	0.35 ± 0.04
	Julia + `pyhf`	1.23 ± 0.16	1.52 ± 0.10	5.59 ± 0.43
	Python + BAT	2.67 ± 0.20	2.91 ± 0.15	6.39 ± 0.30
HMC	Julia + `LiteHF`	0.42 ± 0.07	0.71 ± 0.10	3.59 ± 0.50
	Julia + `pyhf`	19.1 ± 2.0	35.6 ± 2.3	168 ± 9
	Python + BAT	45.9 ± 1.3	61.2 ± 1.9	186 ± 8



(a) run-time benchmarks for Metropolis-Hastings

(b) run-time benchmarks for Hamiltonian MC sampling

Figure 5.2: Run-time benchmark for the `2_bin_corr` model for different MCMC `nsteps` with logarithmic time axis. Uncertainties are visualized as box-plot as illustrated in Figure B.1. The run-time in Julia is averaged over 100 runs, for the other two implementations over 40 runs for MH and 10 runs for HMC sampling.

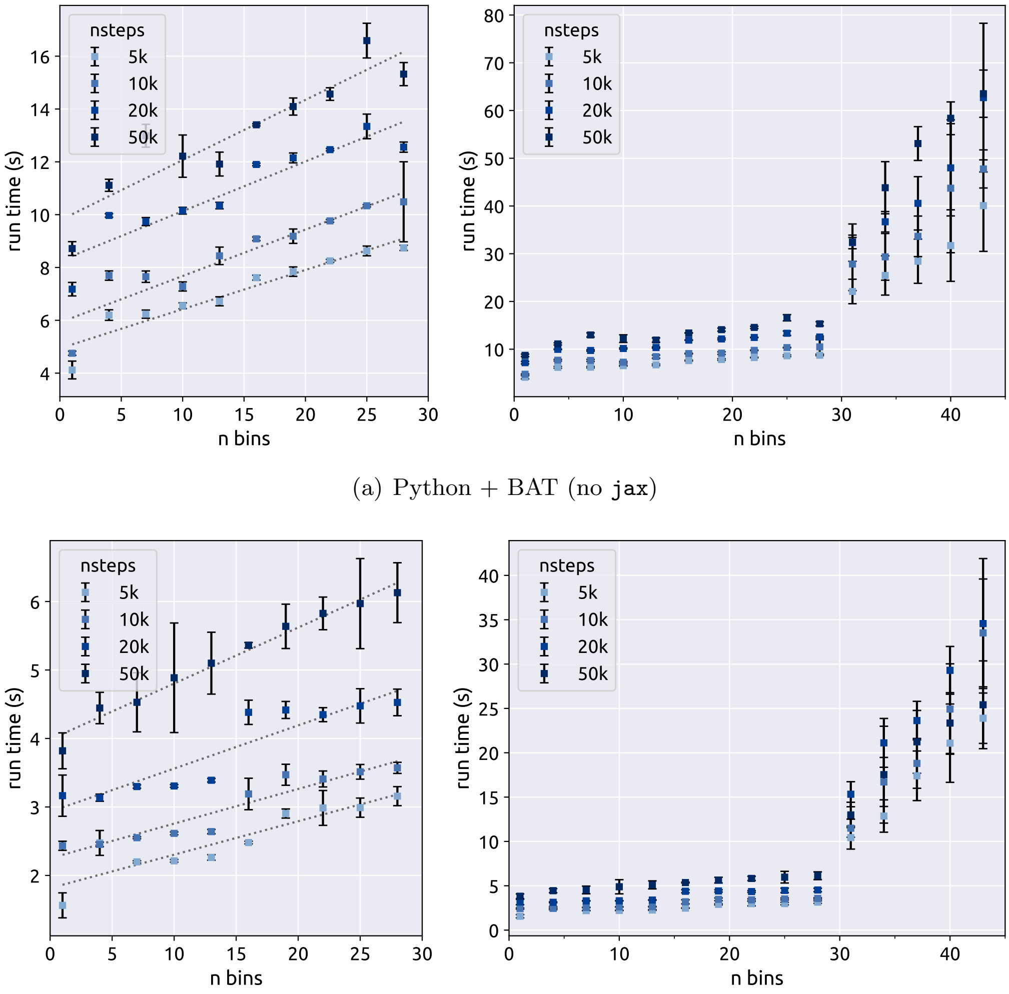
Metropolis Hastings. The run-time of the pure-Julia implementation (*green*) is about 20 times faster than the Python-Julia pipeline (*blue*) for all considered steps. Executing the `pyhf` likelihood from Julia (*red*) is about 12 times slower that the pure-Julia implementation. For an increasing number of samples (`nsteps`) the run-time of Julia + `pyhf` approaches the execution time of batty (Python + `pyhf`).

Hamiltonian MC. Executing HMC in Python (Figure 5.2b) is about 100 times slower than the pure-Julia implementation (*green*) for `5k` and `10k` steps, and about 50 times slower for `50k` steps. Again, calling the `pyhf` likelihood from Julia (*red*) outperforms the `batty` implementation (*blue*). For a small number of `nsteps=[5k, 10k]` the Julia version (Julia + `pyhf`) is about two times faster than `batty`. For `50k` steps both implementations require approximately the same execution time.

5.2.2 n-Bin-Model

The runtime measurements in this section examine how the dimensionality of the parameter space affects the run-time. The n-bin model is implemented as `simplemodel` with uncorrelated background according to the `2_bin_uncorr` model in section A.1. Repeatedly appending bins to the histogram is used as a simple method to increase the dimensionality of the inference problem (n bins corresponds to a $(n+1)$-dimensional parameter vector).

Metropolis-Hastings (MH)



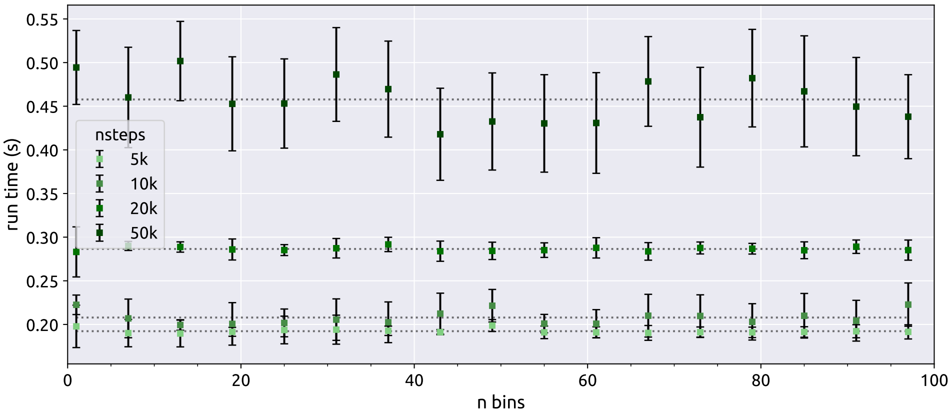
(a) Python + BAT (no `jax`)

(b) Python + BAT (with `jax`)

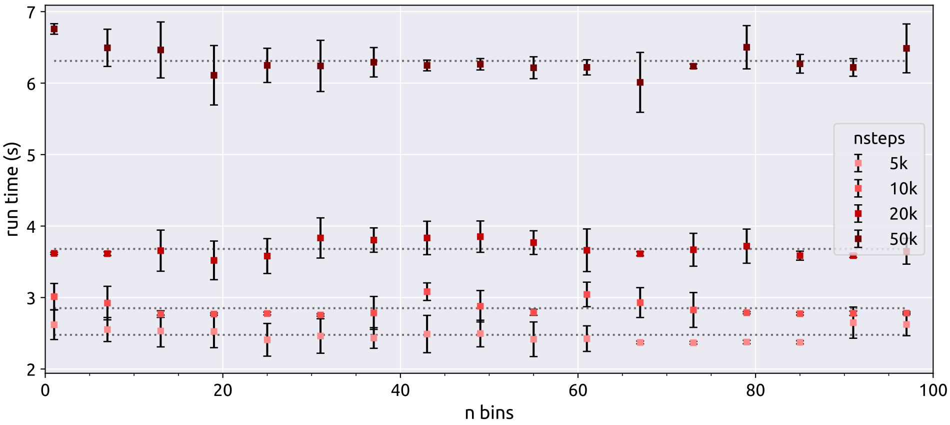
Figure 5.3: Run-time benchmark for the n-bin model in Python using Metropolis Hastings. A model with n bins represents a $(n+1)$-dimensional parameter vector. The upper Figure uses a `Float64` likelihood with no just-in-time compilation by `jax`, the likelihood in the lower Figure is jitted and has `Float32` precision.

Figure 5.3 shows the run-time of `bat_sample()` in `batty` (Python + BAT) for an increasing number of bins using a jitted and not jitted likelihood[1]. In both plots the run-time slightly increases for models with less than 30 bins (*left*). For larger models ($n > 30$ bins), the run-time in Python increases more sharply. The run-time for a 40-bin model already takes about 5 times longer than a 30-bin model.

The Julia benchmarks for the n-bin model are illustrated in Figure 5.4. The pure-Julia version (*green*) is more than 10 times faster than the the jitted Python implementation in Figure 5.3b). Moreover, run-times stay *constant* up to a 100 dimensional parameter vector for different number of steps. The same behavior can be observed for calling the `pyhf` likelihood from Julia (Figure 5.4b).



(a) Julia + `LiteHF`



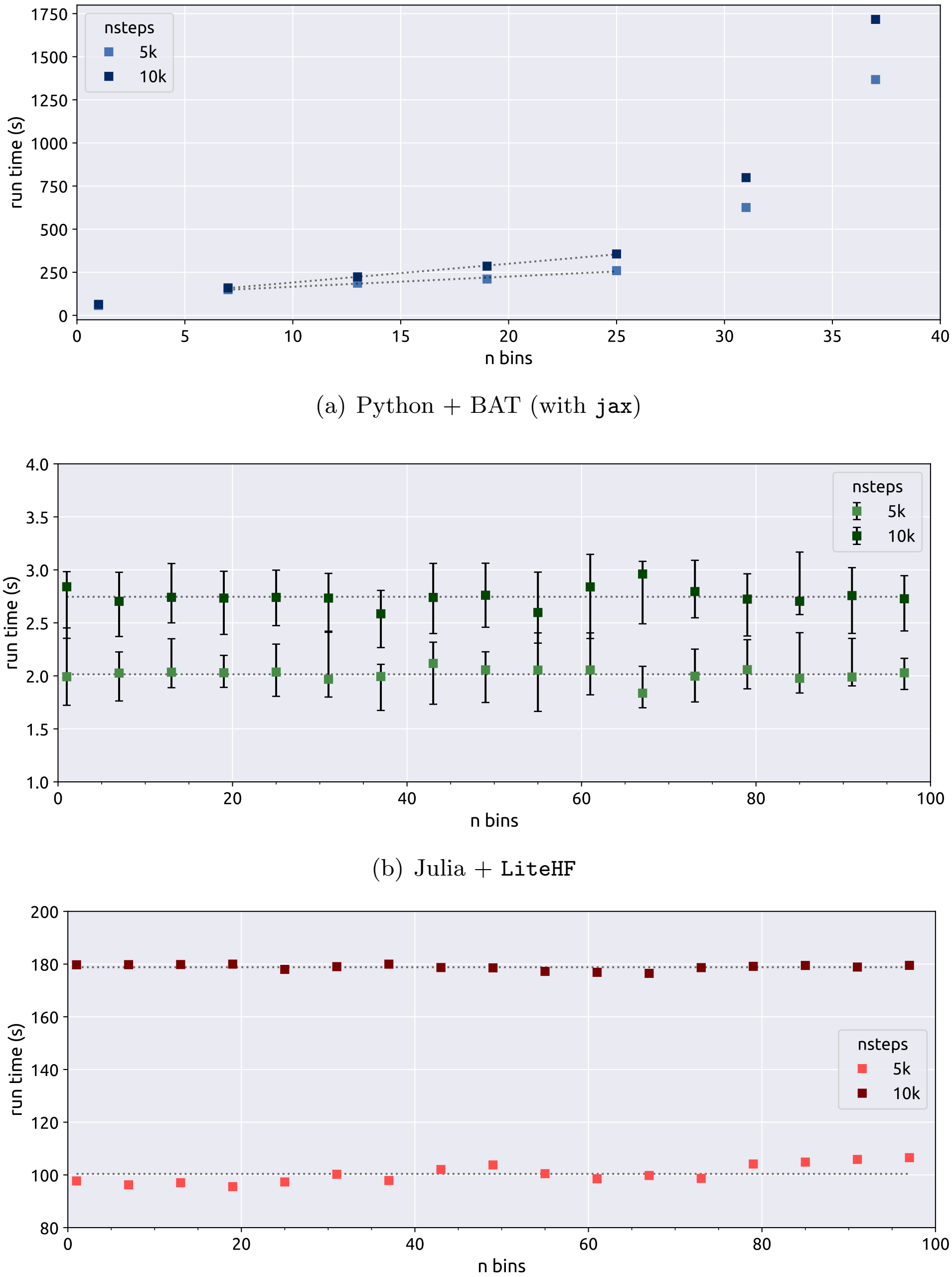
(b) Julia + `pyhf`

Figure 5.4: Run time benchmark for the n-bin model in Julia using Metropolis Hastings. The Julia + `LiteHF` version (*top*) is fully implemented in Julia and outperforms the Julia + `pyhf` implementation (*bottom*) by a factor of 12.

[1] The jitted likelihood is about 2.5 times faster for the n-bin model.

Hamiltonian Monte Carlo (HMC)

The HMC run-times for the n-bin model are illustrated in Figure 5.5 for `5k` and `10k` steps.



(a) Python + BAT (with `jax`)

(b) Julia + `LiteHF`

(c) Julia + `pyhf`

Figure 5.5: Run time benchmark for the n-bin model using HMC for `5k` and `10k` `nsteps`. The Julia run-times (*top*) are averaged over 100 runs, the other two run-times represent a single run.

Similar to MH in Figure 5.3, run-times for the `batty` implementation (Figure 5.5a) slightly

increase for n-bin models with less than 30 parameters. In this range HMC in `batty` is about 100 times slower than in pure Julia (*green*). For $n > 30$ bins the `batty` run-time increases significantly, as already noted for Metropolis Hastings in Figure 5.3.
The run-times for the Julia implementations are shown in Figure 5.5b) and c). Again run-times stay approximately constant up to a 100-bin model.

5.3 Discussion

In all considered benchmarks Julia + `LiteHF` clearly outperforms the other two implementations that call the `pyhf` likelihood. This is, however, to be expected as pure-Julia code is fully compiled to fast machine code while Python code is executed by a high-level interpreter. For the `2_bin_corr` and `4_bin` model (Appendix C) the run-time performance of `batty` is similar to the Julia + `pyhf` implementation for *large* numbers of MCMC steps. On the other hand for `nsteps` $\leq$ `10k`, calling the `pyhf` likelihood from Julia is almost twice as fast as for the `batty` implementation. Again this can be explained by the interpreter overhead of Python which becomes relatively smaller when the run-time of the Julia function `bat_sample()` dominates. For sufficiently small parameter spaces we can summarize that Metropolis-Hastings in `batty` is about 10 times slower than a pure Julia implementation and for HMC `batty` is even slower by a factor of 100.

With the n-bin model the run-time was examined as a function of the parameter dimension. However for both Julia implementations (pure Julia and calling `pyhf` from Julia) the run-time remains *constant* up to a 100 dimensional parameter space for MH and HMC. The log likelihood for the n-bin model appears to be computationally too cheap to measure the influence of the parameter dimension on the run-time in Julia. As a result, other computational load in Julia is predominant which keeps the run-time constant up to 100 bins. In the pure Julia version this is very likely the overhead of the `bat_sample()` method. For Julia + `pyhf` the time for converting Julia types to Python types and vice versa seems to be the time-intensive part.

In Python, on the other hand, the run-time rapidly increases for parameter dimensions greater than 30. Within this thesis task it was not possible to determine the exact reason for this behavior in Python. Still it can be *excluded* that the `Float32` precision of `jax` is an issue, since the `Float64` `numpy` likelihood in Figure 5.3a) shows the same behavior.

In addition, the run-time of the n-bin log likelihood was evaluated for `pyhf` (with and without `jax`) as well as for `LiteHF`. The run-time benchmark is visualized as semi-logarithmic plot in Figure 5.6.

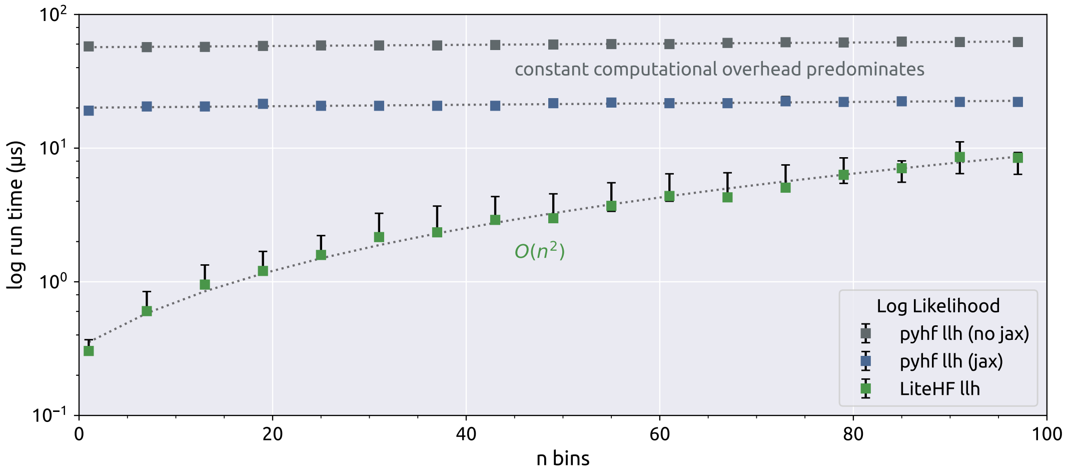


Figure 5.6: Benchmarking the log likelihood of the n-bin model up to 100 bins. The `pyhf` log likelihood is evaluated with and without just-in-time compilation by `jax` and compared to the `LiteHF` log likelihood. The run-time is averaged over 10^4 evaluations.

The `Float64` likelihood (*gray*) has a constant run-time of about 60 µs. Just-in-time compilation (*blue*) reduces the run-time by a factor of 3. However, the `LiteHF` log likelihood (*green*) is still computed more efficiently than the `jax pyhf` llh and shows an $\mathcal{O}(n^2)$ dependency.

The log likelihood can therefore also be excluded as the cause of the run-time increase for $n > 30$. Moreover, `batty` and Julia use the *same* wrappers and conversions to run the `pyhf` likelihood in Julia which makes the Python behavior even more inexplicable. Nevertheless this rapid run-time increase is one of the major drawbacks in the current `batty` implementation which requires further investigation.

Appendix A

HistFactory Models

A.1 2_bin_uncorr Model

- parameter $\boldsymbol{\theta} = [\mu, \gamma_1, \gamma_2]$
- signal model `[5.0, 10.0]` with `normfactor` modifier
- bkg model `[50.0, 60.0]`, `shapesys` modifier with relative uncert. `[10 %, 20 %]`

```
{
  "channels": [
    { "name": "singlechannel",
      "samples": [
        { "name": "signal",
          "data": [5.0, 10.0],
          "modifiers": [ { "name": "mu", "type": "normfactor", "data": null} ]
        },
        { "name": "background",
          "data": [50.0, 60.0],
          "modifiers": [
            { "name": "uncorr_bkguncrt",
              "type": "shapesys",
              "data": [5.0, 12.0] }
          ]
        }
      ]
    }
  ],
  "observations": [
    { "name": "singlechannel", "data": [50.0, 60.0] }
  ],
  "measurements": [
    { "name": "Measurement", "config": {"poi": "mu", "parameters": []} }
  ],
  "version": "1.0.0"
}
```

A.2 2_bin_corr Model

- parameter $\boldsymbol{\theta} = [\mu, \theta]$
- signal model `[12.0, 11.0]` with `normfactor` modifier
- correlated background `[50, 52.0]` with `histosys` modifier
- observations `[53.0, 65.0]`

```
{
    "channels": [
      { "name": "singlechannel",
        "samples": [
          { "name": "signal",
            "data": [12.0,11.0],
            "modifiers": [ { "name": "mu", "type": "normfactor", "data": null } ]
          },
          { "name": "background",
            "data": [ 50.0, 52.0 ],
            "modifiers": [
              { "name": "correlated_bkg_uncertainty",
                "type": "histosys",
                "data": { "hi_data": [45.0, 57.0], "lo_data": [55.0, 47.0] }
              }
            ]
          }
        ]
      }
    ],
    "observations": [
      { "name": "singlechannel", "data": [53.0, 65.0] }
    ],
    "measurements": [
      {
        "name": "Measurement",
        "config": {
        "poi": "mu",
        "parameters": []
        }
      }
    ],
    "version": "1.0.0"
  }
```

A.3 4_bin Model

- parameter $\boldsymbol{\theta} = [\mu, \theta, \theta_{SF}]$
- signal model `[2, 3, 4, 5]` with `normfactor` modifier
- background model `[30, 19, 9, 4]` with `histosys` and `normsys` modifier
- observations `[34, 22, 13, 11]`

```
{
  "channels": [
    { "name": "singlechannel",
      "samples": [
        { "name": "signal MC",
          "data": [2, 3, 4, 5],
          "modifiers": [ { "name": "mu", "type": "normfactor", "data": null } ]
        },
        {
          "name": "bkg MC",
          "data": [30, 19, 9, 4],
          "modifiers": [
            { "name": "theta",
              "type": "histosys",
              "data": { "hi_data": [31, 21, 12, 7], "lo_data": [29, 17, 6, 1] }
            },
            { "name": "SF_theta",
              "type": "normsys",
              "data": {"hi": 1.1,"lo": 0.9}
            }
          ]
        }
      ]
    }
  ],
  "observations": [
    { "name": "singlechannel", "data": [34, 22, 13, 11]}
  ],
  "measurements": [
    { "name": "Measurement", "config": {"poi": "mu","parameters": []} }
  ],
  "version": "1.0.0"
}
```

Appendix B

Run-time Statistics

While executing code on a PC, the running time of a function varies due to other active processes. A typical run-time statistics for 10^4 samples is illustrated in Figure B.1.

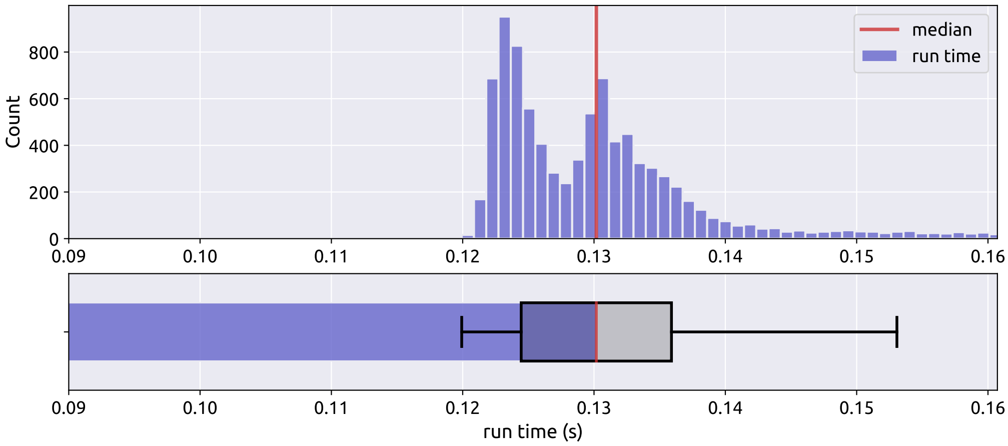


Figure B.1: Run-time statistics for evaluating the log likelihood 10k times.

In order simplify the plot while keeping track of the process statistics, run-times are visualized as bar-plot with an box-plot on top. The final bar value is the median run-time, which corresponds to the red line in the box-plot. The box contains 50 % of the data that is closed to the median and the whiskers mark the minimum and maximum value.

Benchmarking the log likelihood function with numpy and jax backend

A just-in-time compiled likelihood function (by jax) reduces the run-time by a factor of 1.5 to 2.5, compared to the numpy backend.

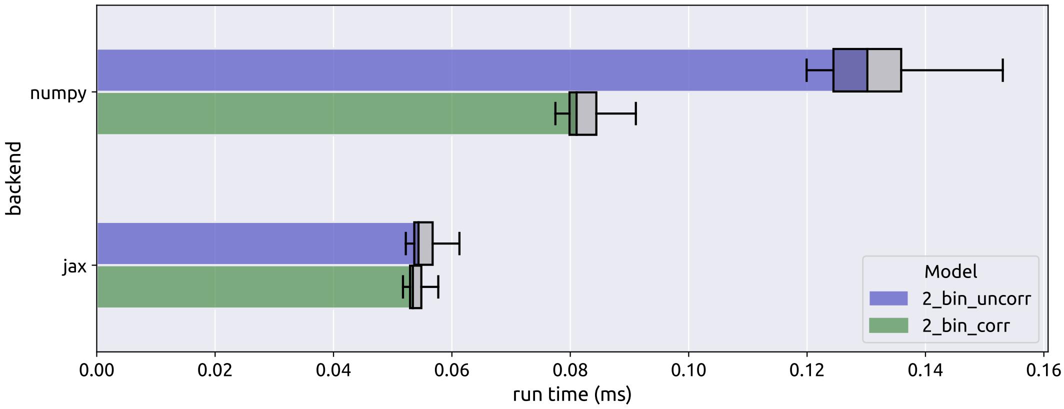


Figure B.2: Benchmarking the log likelihood with numpy and jax backend for two HF models.

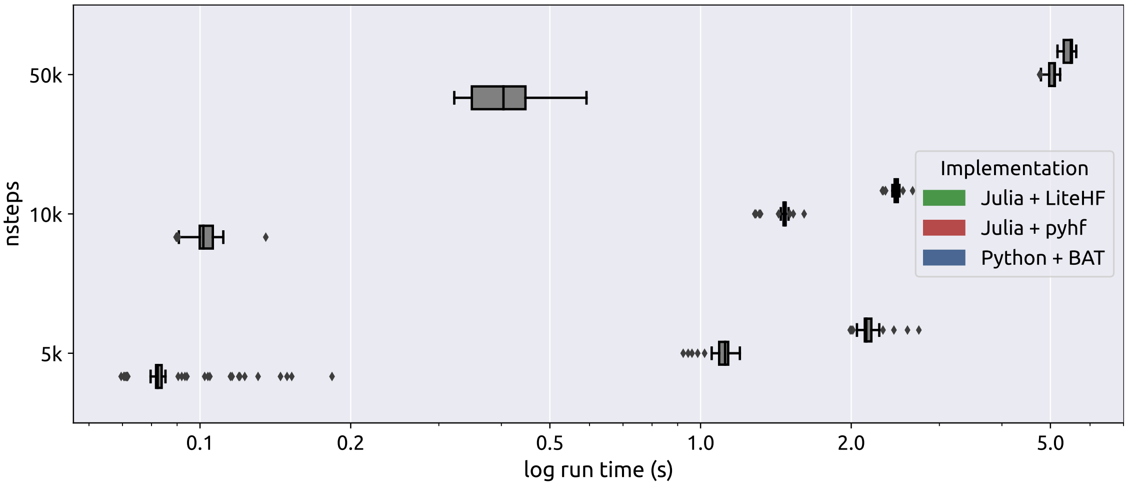
Appendix C

Additional Run-time Benchmarks

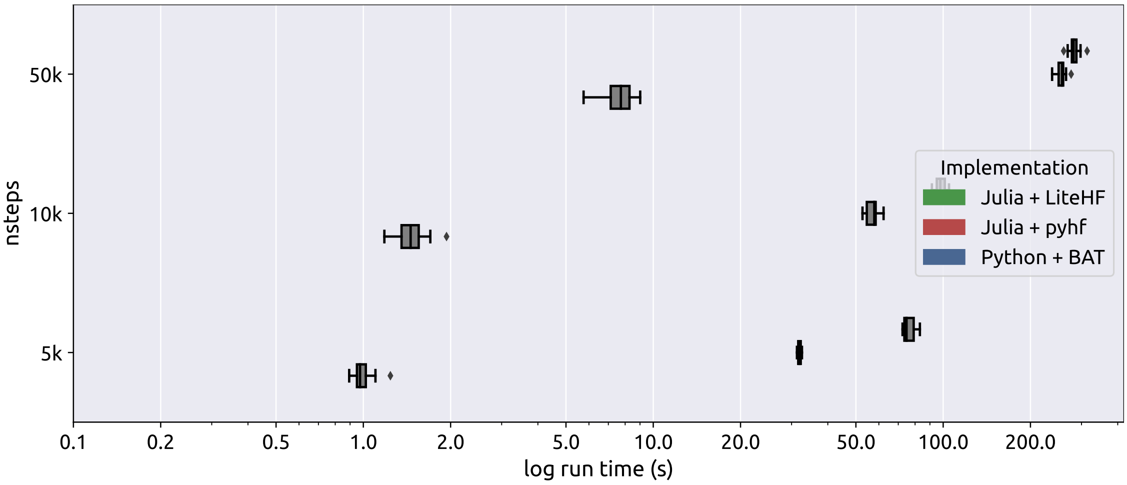
The run-time benchmarks for the `4-bin` model (section A.3) are visualized in Figure C.1 and the numerical values summarized in Table C.1. The run-time performance coincidence with the observations for the `2_bin_corr` model in Figure 5.2, while HMC need almost twice as long for the `4_bin` model. This can be explained by the increased model complexity. The run-time performance for MH is similar for both models.

Table C.1: Summary of the run-time benchmarks in Figure C.1. The values represent the median run-time with 1σ standard deviation without outliers for `5k`, `10k` and `50k` `nsteps`.

		5k	**10k**	**50k**
Method	**Implementation**		**run-time** (s)	
MH	Julia + `LiteHF`	0.08 ± 0.01	0.10 ± 0.00	0.40 ± 0.05
	Julia + `pyhf`	1.11 ± 0.06	1.47 ± 0.05	5.09 ± 0.13
	Python + BAT	2.14 ± 0.12	2.46 ± 0.05	5.49 ± 0.13
HMC	Julia + `LiteHF`	0.97 ± 0.06	1.45 ± 0.14	7.73 ± 0.79
	Julia + `pyhf`	31.9 ± 0.9	57.8 ± 3.0	257 ± 11
	Python + BAT	74.7 ± 4.0	97.7 ± 4.2	281 ± 14



(a) run-time benchmarks for Metropolis-Hastings



(b) run-time benchmarks for Hamiltonian MC sampling

Figure C.1: Run-time benchmark for the `4_bin` model for different MCMC `nsteps` with logarithmic time axis. Uncertainties are visualized as box-plot as illustrated in Figure B.1. The run-time in Julia is averaged over 100 runs, for the other two implementations over 40 runs for MH and 10 runs for HMC.

List of Figures

Bibliography

[1] Iain Murray. *Advances in Markov chain Monte Carlo methods.* University of London, University College London (United Kingdom), 2007.

[2] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[3] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo.* CRC press, 2011.

[4] ATLAS Collaboration. Measurements of Higgs boson production and couplings in diboson final states with the ATLAS detector at the LHC. *Physics Letters B*, 726:88–119, 2013.

[5] Albert M Sirunyan, Armen Tumasyan, Wolfgang Adam, Federico Ambrogi, Thomas Bergauer, Johannes Brandstetter, Marko Dragicevic, Janos Erö, Alberto Escalante Del Valle, Martin Flechl, et al. Search for supersymmetry in proton-proton collisions at 13 TeV in final states with jets and missing transverse momentum. *Journal of High Energy Physics*, 2019(10):1–61, 2019.

[6] Lukas Heinrich, Matthew Feickert, and Giordon Stark. `pyhf` documentation `v0.6.3`. https://pyhf.readthedocs.io/en/v0.6.3/intro.html.

[7] Lukas Heinrich, Matthew Feickert, and Giordon Stark. Python HistFactory `pyhf`. https://github.com/scikit-hep/pyhf, 2021.

[8] Jerry Ling, Oliver Schulz, and Gabriel Rabanal. LiteHF.jl, Light-weight HistFactory in pure Julia. https://github.com/JuliaHEP/LiteHF.jl.

[9] Oliver Schulz, Frederik Beaujean, Allen Caldwell, Cornelius Grunwald, Vasyl Hafych, Kevin Kröninger, Salvatore La Cagnina, Lars Röhrig, and Lolian Shtembari. BAT.jl: A Julia-Based Tool for Bayesian Inference. *SN Computer Science*, 2(3):210, Apr 2021.

[10] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.

[11] Oliver Schulz. ValueShapes.jl. https://github.com/oschulz/ValueShapes.jl.

[12] Dahua Lin, John Myles White, Simon Byrne, Douglas Bates, Andreas Noack, John Pearson, Alex Arslan, Kevin Squire, David Anthoff, Theodore Papamarkou, Mathieu Besancon, Jan Drugowitsch, Moritz Schauer, and other contributors. JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions. https://github.com/JuliaStats/Distributions.jl, July 2019.

[13] Philipp Eller, Oliver Schulz, and Christian Gajek. batty: BAT to Python Interface. https://github.com/bat/batty.

[14] Steven G. Johnson. PyCall.jl. https://github.com/JuliaPy/PyCall.jl.

[15] Christopher Rowley. PythonCall.jl: Python and Julia in harmony. https://github.com/cjdoris/PythonCall.jl, 2022.

[16] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

[17] Daniel Foreman-Mackey. corner.py: Scatterplot matrices in Python. *The Journal of Open Source Software*, 1(2):24, 2016.

[18] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. https://github.com/JuliaCI/BenchmarkTools.jl, 2016.