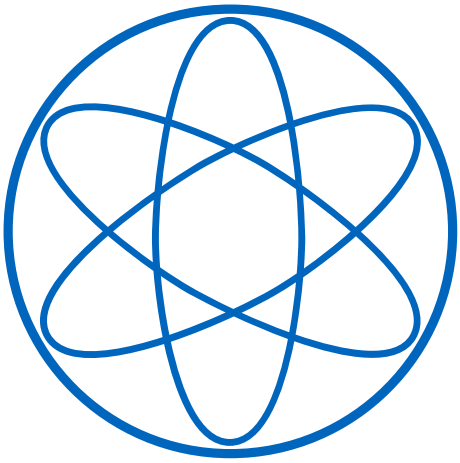
Development and Benchmarking of a Bayesian Inference Pipeline for LHC Physics



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

TODO

Contents

Abbreviations

BAT Bayesian Analysis Toolkit.

GC garbage collector.

HEP High Energy Physics.
HF HistFactory.
HMC Hamiltonian Monte Carlo.

LHC Large Hadron Collider.

MC Monte Carlo.
MH Metropolis-Hastings.

pdfs probability density functions.
POI parameter of interest.

Chapter 1

Introduction

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

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 $\boldsymbol{\theta}$.

2.2 MCMC Sampling

Monte Carlo (MC)
Markov chain Monte Carlo (MCMC)

2.2.1 Markov Chains

2.2.2 Metropolis Hastings

Metropolis-Hastings (MH)

2.2.3 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC)

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 [1] and is used in searches for new physics [2] by research groups around the planet. The relationship between theoretical predictions and collision events is formalized as *statistical model* $p(\boldsymbol{x} \mid \boldsymbol{\theta})$. It describes the probability of observing data $\boldsymbol{x}$ given the model parameters $\boldsymbol{\theta}$. Typically, models in High Energy Physics (HEP) are complex with hundreds of parameters. HF enables a standardized way to build parametrized probability density functions (pdfs) 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 particles 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

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

[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.

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

For each bin b and channel c, the total event rate ν_{cb} is is the sum over 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^0_{scb} 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 $\boldsymbol{\theta} = (\boldsymbol{\eta}, \boldsymbol{\chi})$.

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

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 [3]

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`	$\Delta_{scb}(\alpha) = f_p\left(\alpha \,|\, \Delta_{scb,\alpha=\pm1}\right)$	$\text{Normal}\left(a = 0 \,|\, \alpha, \sigma = 1\right)$	$\Delta_{scb,\alpha=\pm1}$
Normalisation 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
Normalisation `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 that are defined by a constraint term $c_{\boldsymbol{\chi}}$ (right part in (3.2)) and an input parameter (`auxdata`).

- **Uncorrelated shape** modifiers affect each bin individually and are constrained by a Poisson. 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 modifiers and controlled by a single parameter.

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 [3].

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, (cite?)
- a Python version `pyhf` [4], and
- a Julia implementation `LiteHF` [5]

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).

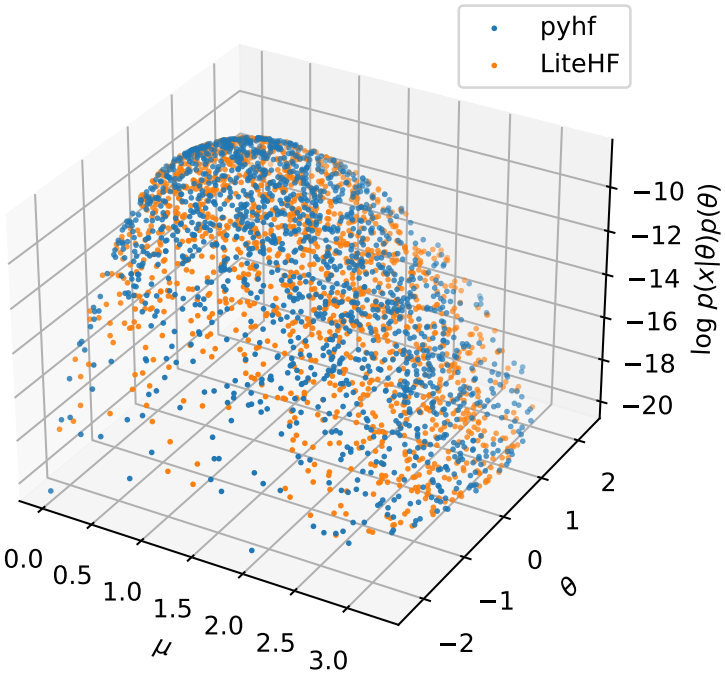


Figure 3.1: Verify the equivalence of the `pyhf` and `LiteHF` implementation by evaluating the log posterior measure $\log\left(p\left(\boldsymbol{x}\,|\,\boldsymbol{\theta}\right)p(\boldsymbol{\theta})\right)$ at 1000 random points $\boldsymbol{x} \sim p(\boldsymbol{\theta})$ for the `2_bin_corr` model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

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-Julia interface to execute BAT code (written in Julia) in Python in order to perform Bayesian inference on `pyhf` models. The chapter closes with ... TODO

4.1 BAT

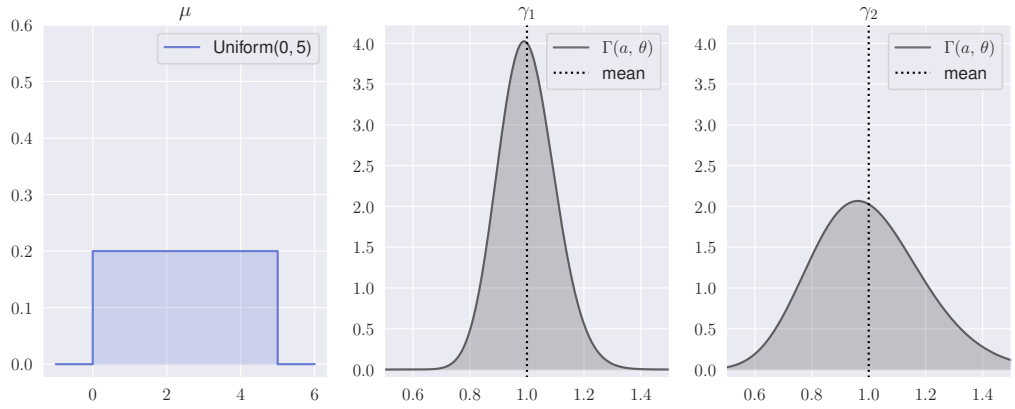
Bayesian Analysis Toolkit (BAT) [6]

4.2 Priors from `auxdata`

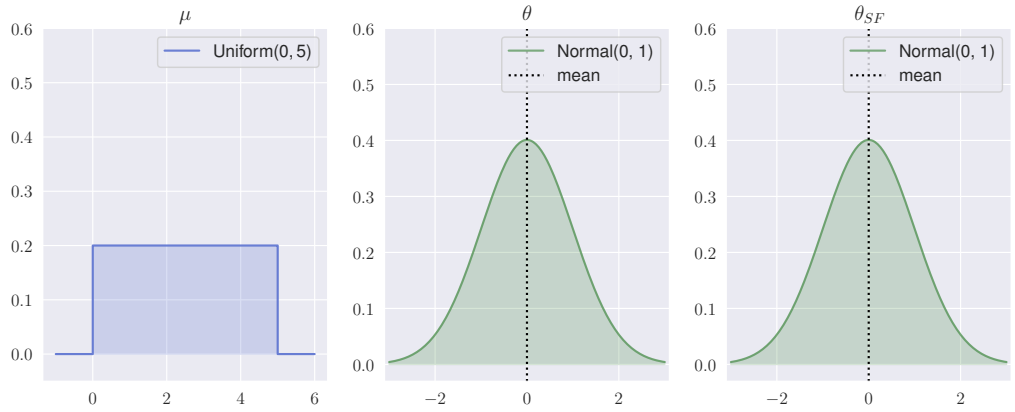
4.2.1 Conjugate Priors

4.2.2 Implementation

Generating priors $p(\boldsymbol{\theta})$ from `auxdata` in HF models is summarized in the Python package `priorhf` which was implemented during this work.
`model.config.par_map` is a ordered mapping from the parameter name to the slice in the parameter vector.
[7]



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

4.3 batty – BAT to Python Interface

The complete pipeline for calling BAT code from Python is fully implemented in `batty`, the BAT to Python Interface, developed here [8]. During this work, `batty` was steadily tested an updated with several contributions. An old version of `batty` used `PyCall.jl` [9] to access Julia code 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` [10]

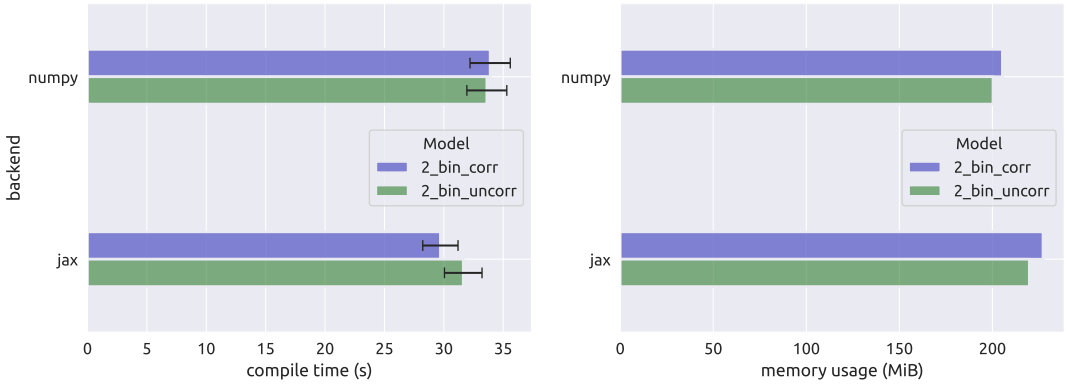
4.3.1 Gradients for HMC

In order to run HMC in

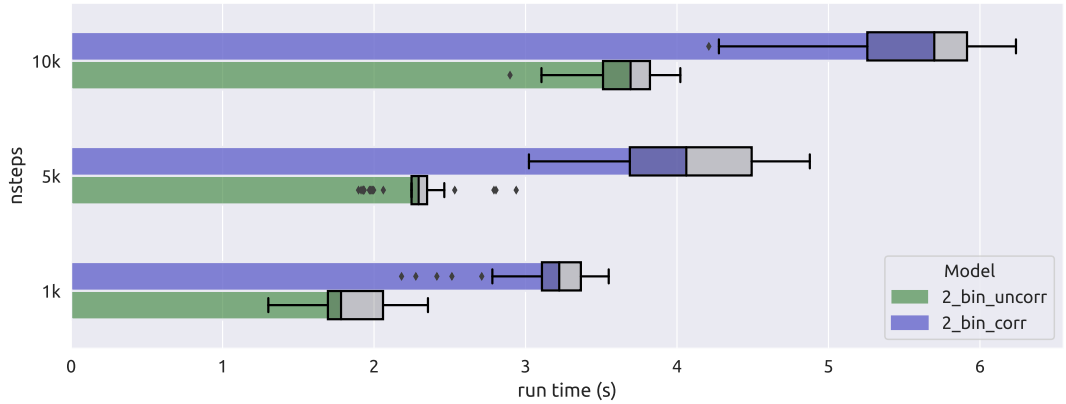
- use median instead mean
- use bar + box plot

4.3.2 `pyhf` Backends

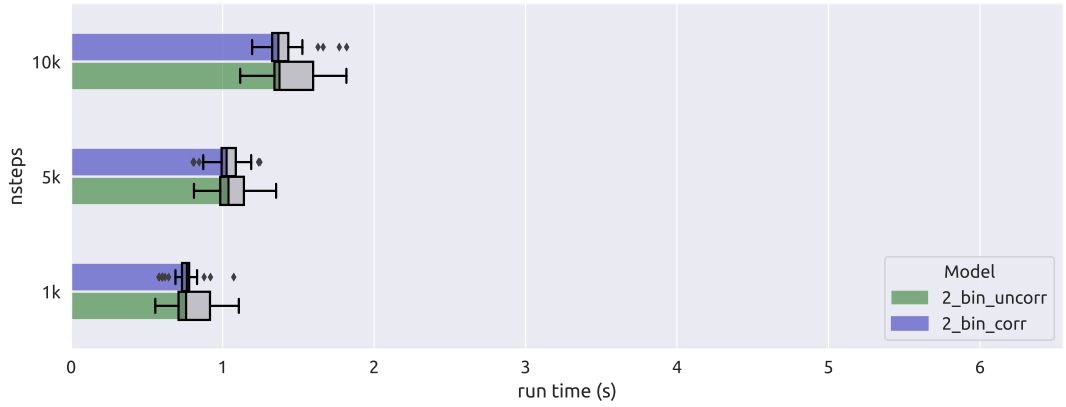
- `jax` jitted log likelihood is 4 times faster with `numpy` backend



(a) Compile time and memory usage during compilation



(b) `pyhf` with `numpy` backend



(c) `pyhf` with `jax` backend

Figure 4.2: Compile time and run-time performance of `bat_sample()` once using the `numpy` backend and once with `jax` backend of `pyhf`. The run-time is evaluated for two different models `2_bin_corr` and `2_bin_uncorr`.

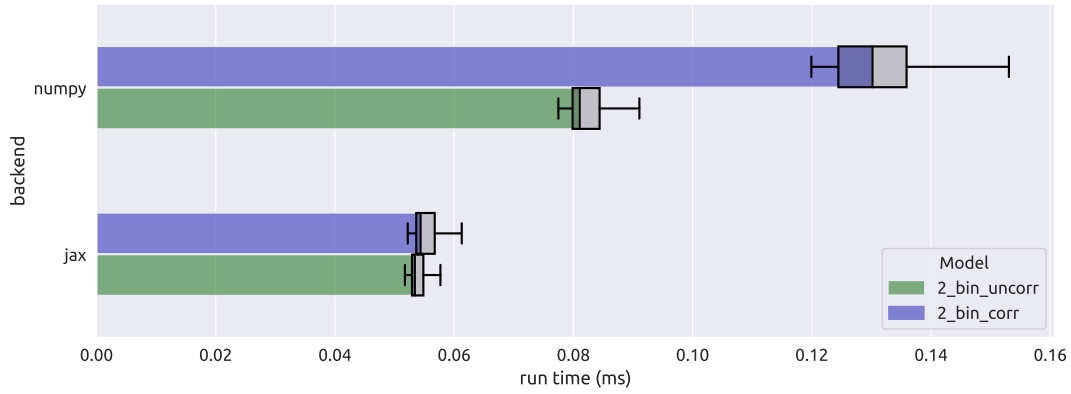


Figure 4.3: Benchmarking the likelihood for the `numpy` and `jax` `pyhf` backend based on two HF models.

4.4 Bayesian Inference Examples

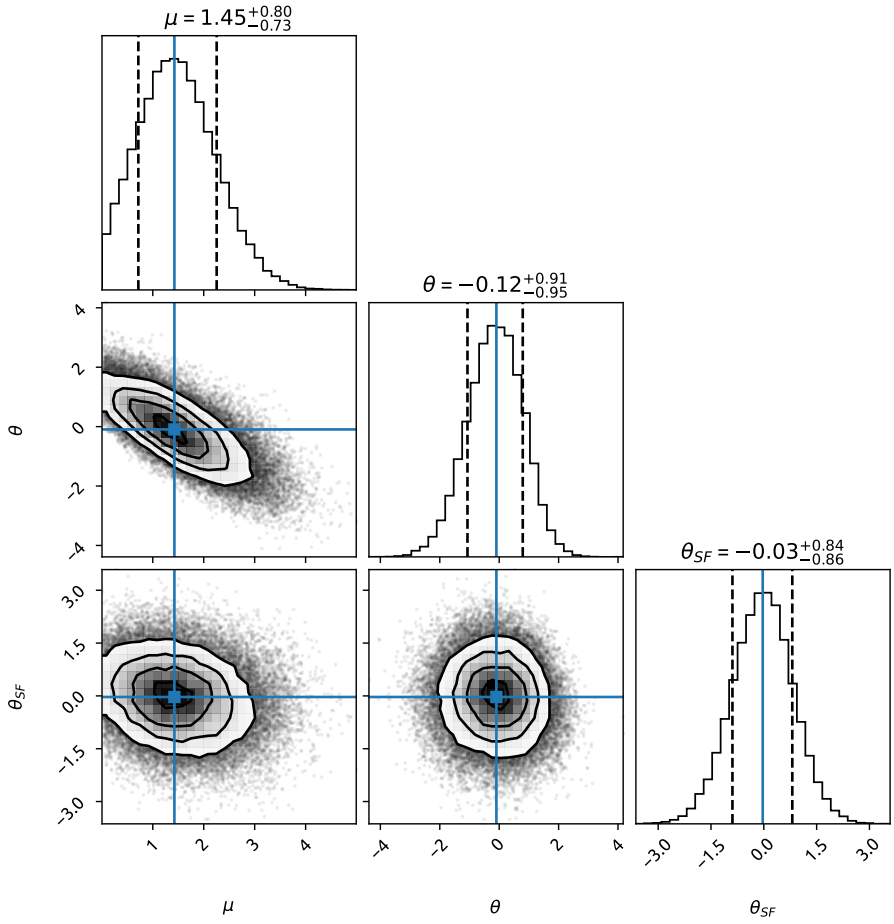
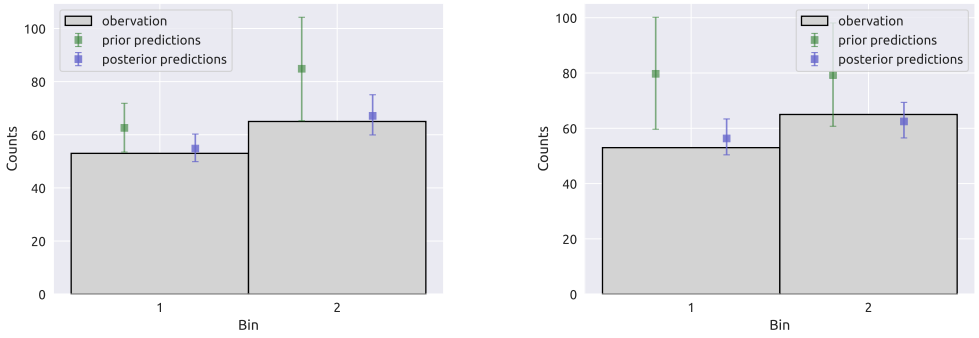
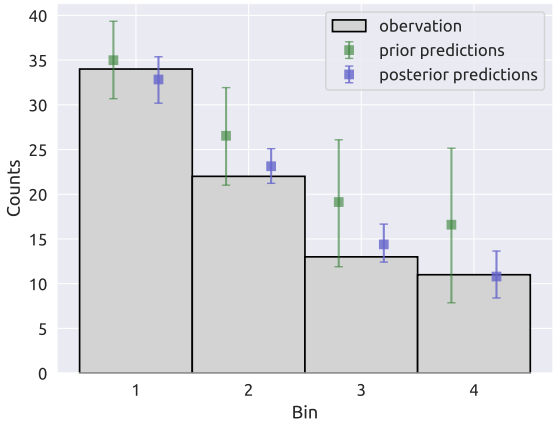


Figure 4.4: Corner plot for the `4_bin` Model (section A.3) for 100k samples. (dashed line 0.16, 0.84 quantile, blue lines mode) XX



(a) `2_bin_uncorr` model

(b) `2_bin_corr` model



(c) `4_bin` model

Figure 4.5: Prior and posterior predictive checks for all used models. The dot represents the median for expected 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` [5]. 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` [5], 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 [10]. The `pyhf` likelihood are outsourced in a module `pyhf_llh` 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 different HF models.

5.1 Benchmark Setup

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

```
# Metropolis Hastings
method = BAT.MCMCSampling(mcalg=BAT.MetropolisHastings(), nsteps=nsteps, nchains=2)
# or HamiltonianMC
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, and the run-time is measured for different numbers of steps `nsteps`.

To ensure reproducible 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 steps. The compile time accounts for more than 90 % of the total execution time in all implementation 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 [11]. 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-times.
- All run-time benchmarks are written in Python or Julia scripts, since Jupyter Notebooks have lots of overhead.
- All benchmarks are carried out on a `Linux` computer with a `Intel(R) Core(TM) i7-7700HQ` CPU.

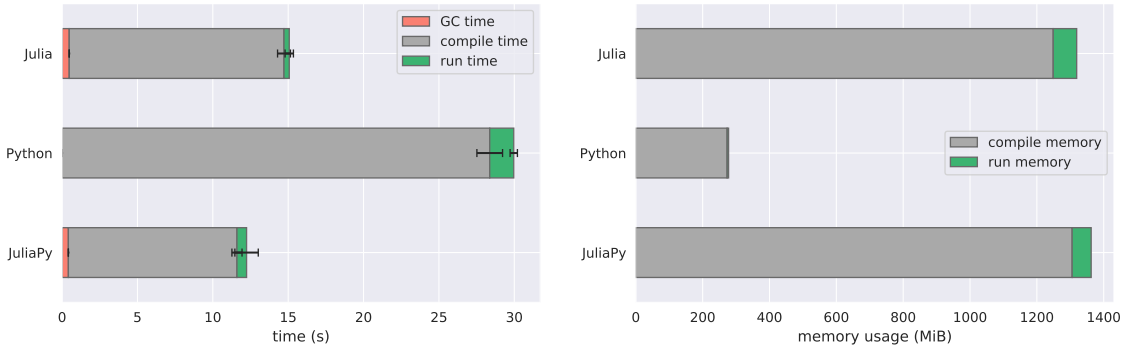


Figure 5.1: Total execution time and memory usage of `bat_sample()` for the `2_bin_corr` model with 10k steps. The red, grey and green portion is the garbage collector (GC) time, the compile time and run time respectively. Compiling `bat_sample()` in Python takes about twice as long as in Julia. In contrast, 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.)

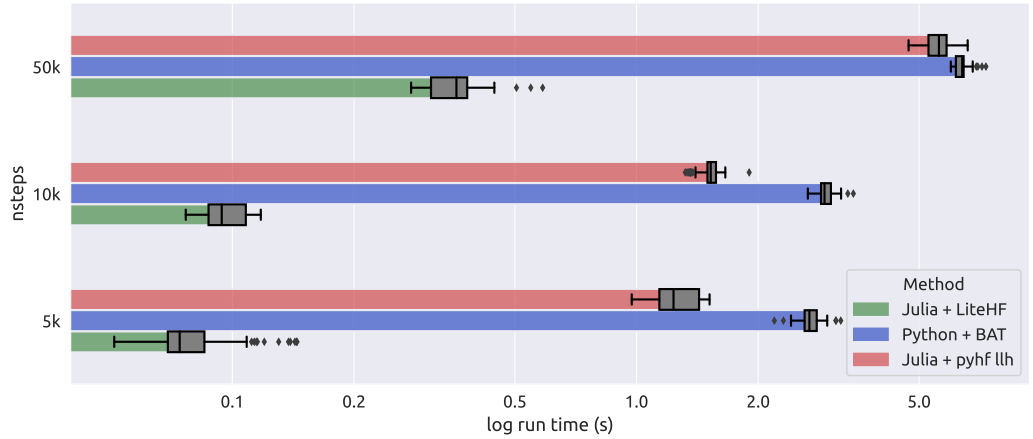
5.2 Benchmarks

5.2.1 `2_bin_corr` Model

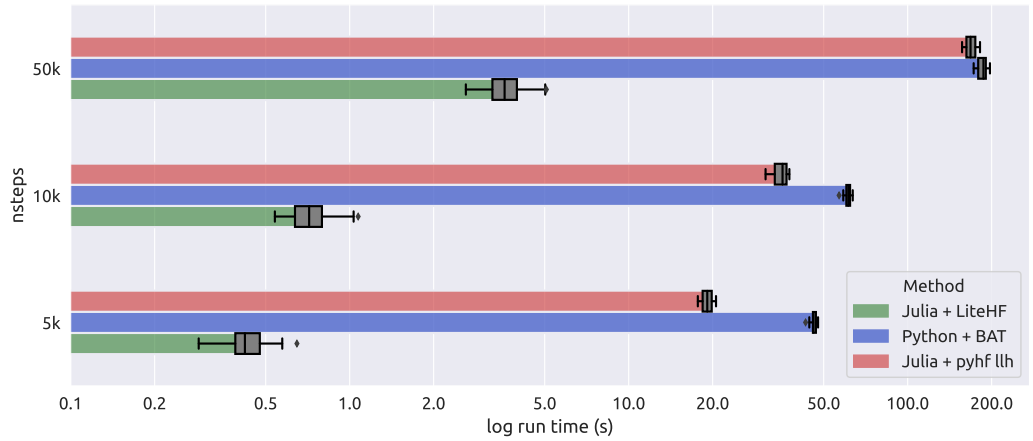
First, the run-time of the `2_bin_corr` model is evaluated for different values of `nsteps`. The results are visualized in Figure 5.2a) for Metropolis Hastings and in Figure 5.2b) for Hamiltonian MC. The run-time on the x-axis is plotted logarithmically and the numerical values are summarized Table 5.1.

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
	Python + `pyhf`	2.67 ± 0.20	2.91 ± 0.15	6.39 ± 0.30
	Julia + `pyhf`	1.23 ± 0.16	1.52 ± 0.10	5.59 ± 0.43
HMC	Julia + `LiteHF`	0.42 ± 0.07	0.71 ± 0.10	3.59 ± 0.50
	Python + `pyhf`	45.9 ± 1.3	61.2 ± 1.9	186 ± 8
	Julia + `pyhf`	19.1 ± 2.0	35.6 ± 2.3	168 ± 9



(a) run-time benchmarks for Metropolis-Hastings



(b) run-time benchmarks for Hamiltonian MC

Figure 5.2: Run-time benchmark of the `2_bin_corr` model for different values of `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.

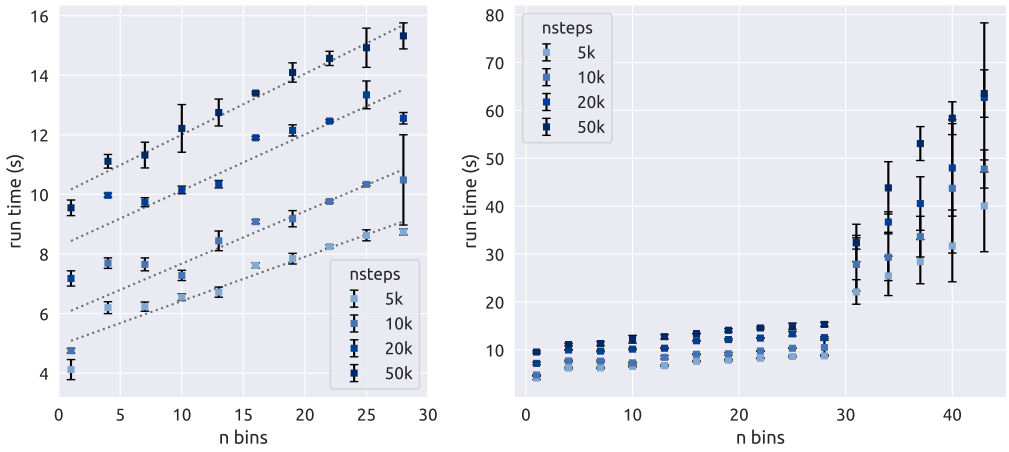
Metropolis Hastings. The run-time of the pure-Julia implementation (green) is about 20 times faster than the Python-Julia pipeline (blue) for all steps. Executing the `pyhf` likelihood from Julia (red) is about 12 times slower that the pure-Julia implementation. For an increasing number of samples (`nsteps`)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

Hamiltonian MC. Executing HMC in Python is about 100 times slower than the pure-Julia implementation 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 almost the same time.

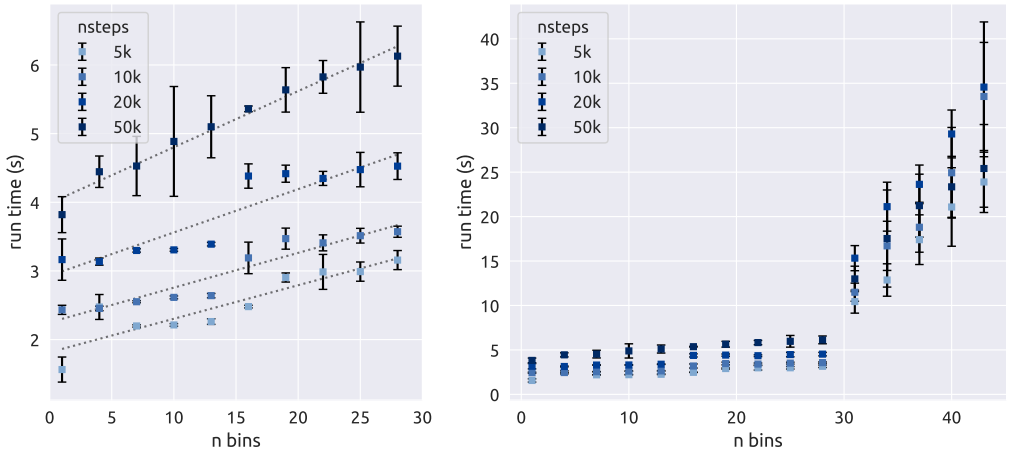
5.2.2 n-Bin-Model

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)

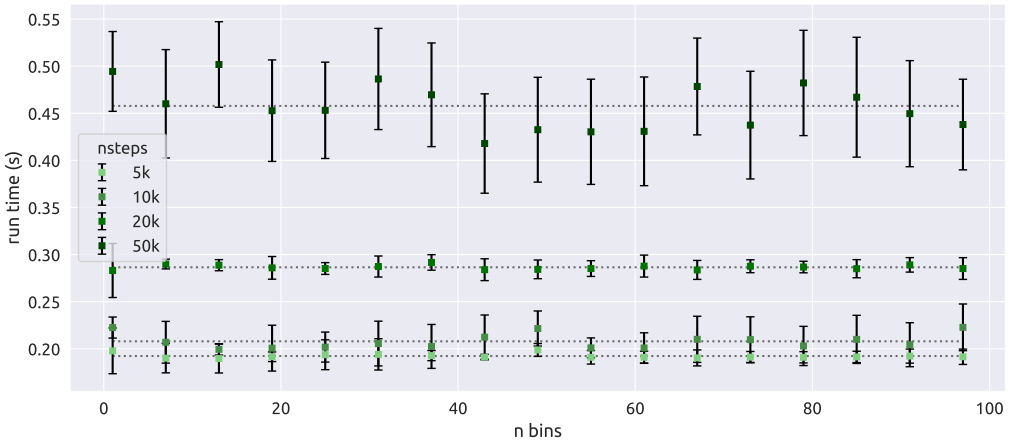


(a) Python + BAT (no `jax`)

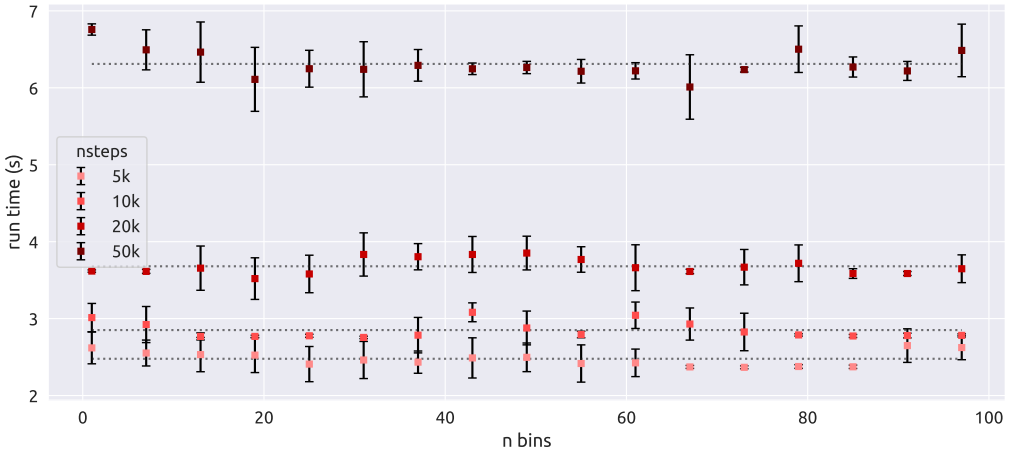


(b) Python + BAT (with `jax`)

Figure 5.3: Run-time benchmark of 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.



(a) Julia + `LiteHF`



(b) Julia + `pyhf`

Figure 5.4: Run time benchmark of the n-bin Model in Julia using Metropolis Hastings.

Figure 5.3 shows the run-time development in Python (`batty`) for a likelihood with and without just-in-time compilation by `jax`[1]. In both plots the run-time slightly increases for models with less than 30 bins (*left*). For larger models with $n > 30$ bins (*right*), 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.
Explain?

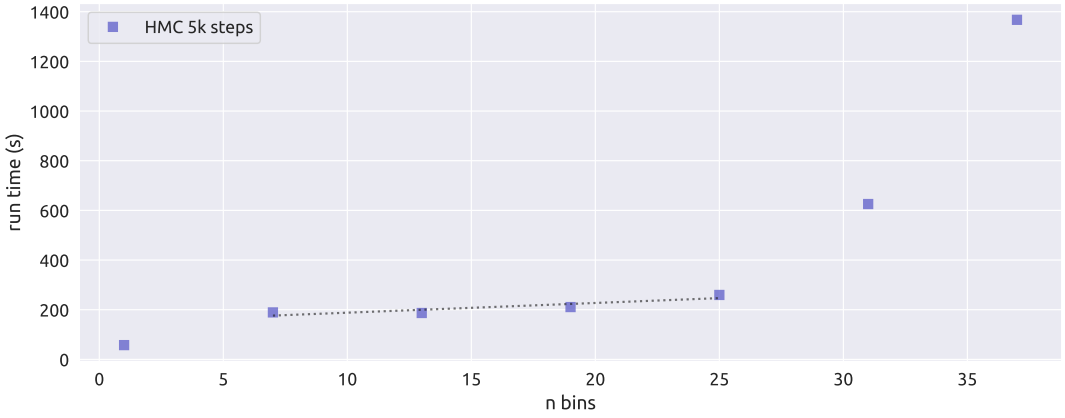
The Julia benchmarks for the n-bin model are visualized in Figure 5.4. The pure-Julia version (green) is more than 10 times faster than the the jitted Python implementation. Moreover, run-times stay *constant* up to a 100 dimensional parameter vector for different

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

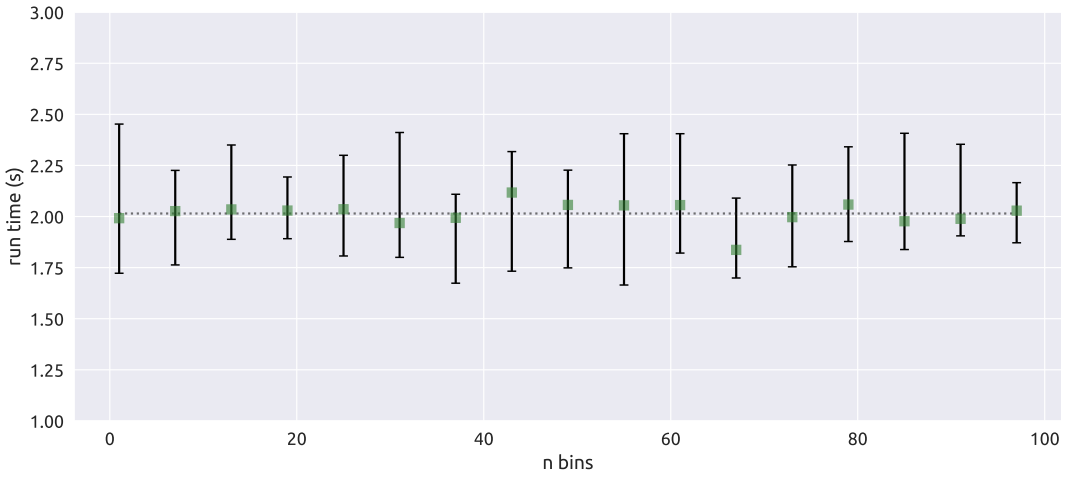
number of steps. The same behaviour can be observed for calling the `pyhf` likelihood from Julia (Figure 5.4b) while this implementation is about 12 times slower than the pure-Julia version.

Hamiltonian Monte Carlo (HMC)

The HMC run-times for the n-bin model are illustrated in Figure 5.5 for 5k `nsteps`. Similar to MH in Figure 5.3, run-times for the `batty` implementation only slightly increase for models with less than 30 parameters. Afterwards, the run-time significantly increases for $n > 30$ bins.



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



(b) Julia + `LiteHF`

Figure 5.5: Run time benchmark of the n-bin Model in Julia using HMC for 5k steps. The Julia run-times are averaged over 100 runs, the Python run-times only represent a single run.

The run-times for the pure-Julia implementation are shown in Figure 5.5b). Again, run-

times stay constant up to a 100 bin model and are about 100 times faster the Python version.

5.3 Discussion

In all considered benchmarks, the pure-Julia runs significantly faster than the other two implementations, where the `pyhf` likelihood is involved. Metropolis Hastings runs about ...
... n-bin model in python jump at 30... However, the executing time of the log likelihood only slightly decreases

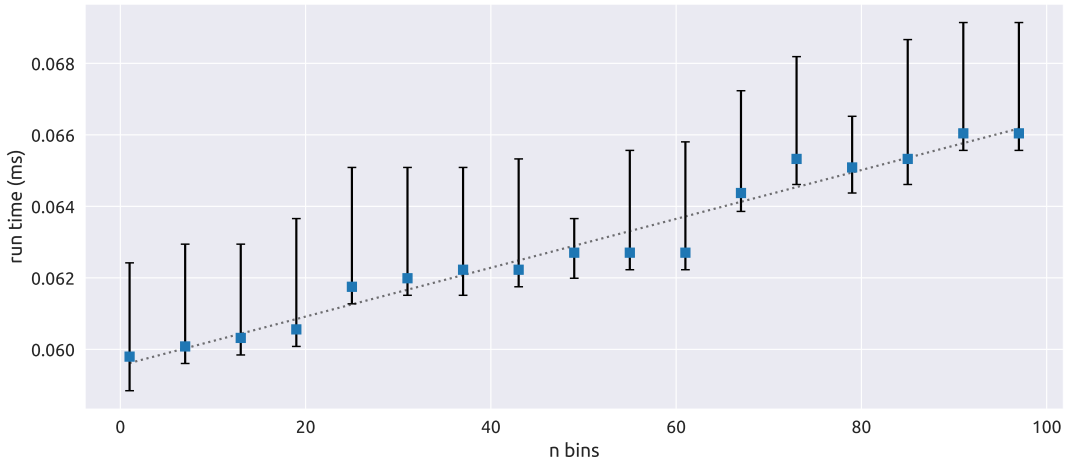


Figure 5.6: Evaluating the log likelihood for the n-bin model up to 100 bins.

Python **Issue** for large dimensions (> 30): sampler chains often do not converge, since `BrooksGelmanConvergence` is not achieved. `MCMCMultiCycleBurnin` needs to be adapted...

- Explain
- Although

Appendix A

HistFactory Models

A.1 2_bin_uncorr Model

- parameter $\boldsymbol{\theta} = [\mu, \gamma_1, \gamma_2]$
- signal model `[5.0, 10.0]`, `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]`, `normfactor` modifier


```
{
    "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 $\boldsymbol{\theta} = [\mu, \theta, \theta_{SF}]$
- ...


```
{
  "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 execution time of a function varies due to other active processes. A typical run time statistics for 10k runs is illustrated in Figure B.1.

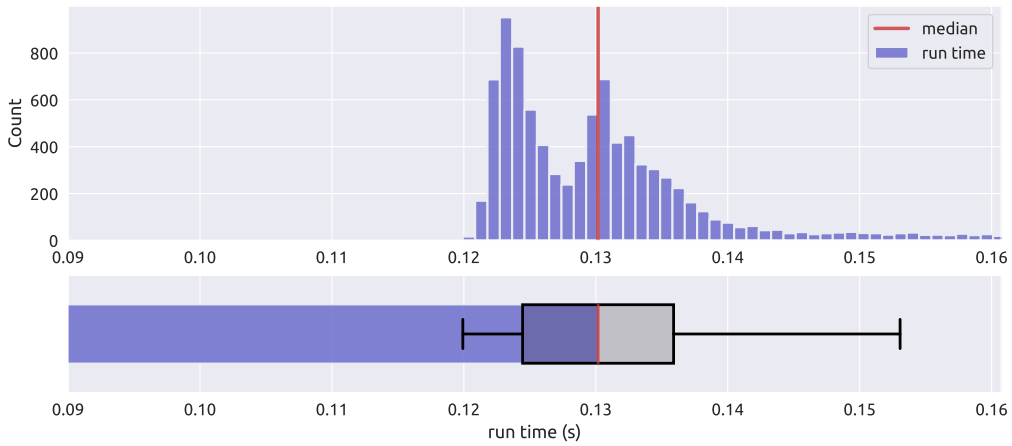


Figure B.1: Run-time statistics for evaluating the log likelihood 10k times.

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 value of the bar is the median run-time, which represents 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.

Appendix C

Additional Run-time statistics

List of Figures

Bibliography

[1] 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.

[2] 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.

[3] `pyhf` documentation `v0.6.3`. https://pyhf.readthedocs.io/en/v0.6.3/intro.html.

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

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

[6] 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.

[7] Dahua Lin, John Myles White, Simon Byrne, Douglas Bates, Andreas Noack, John Pearson, Alex Arslan, Kevin Squire, David Anthoff, Theodore Papamarkou, Mathieu BesanÃ§on, 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.

[8] Philipp Eller. batty: BAT to Python Interface. https://github.com/philippeller/batty.

[9] Pycall.jl. https://github.com/JuliaPy/PyCall.jl.

[10] Christopher Rowley. Pythoncall.jl: Python and julia in harmony. https://github.com/cjdoris/PythonCall.jl, 2022.

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