Metropolis Markov Chain Monte Carlo: a C++ Implementation

Dawid Hryniuk, 10197579

May 23, 2021

Abstract

Probability distributions can be approximated through sampling. A powerful class of algorithms to efficiently sample from probability distributions is Markov Chain Monte Carlo (MCMC). A program to sample one-dimensional probability distributions by means of Metropolis MCMC with plotting, multithreading, and posterior distribution sampling functionalities is presented and discussed. A number of example applications are considered.

1 Introduction

Many physical phenomena are characterised by randomness and, to be appropriately modeled, must be described in terms of random variables. The probability distribution for a random variable represents how the probabilities are distributed over the values of the random variable. The probability distributions one comes across can be intractable, meaning that they can not be represented analytically in a closed form expression. This is very often the case in the context of Bayesian inverse problems, where the *posterior* distribution $\pi_Y(\theta)$ that one wants to determine depends non-trivially on the set of observations Y and hyperparameters θ that are present in the *prior* $\pi_0(\theta)$ and *likelihood* $\pi(Y|\theta)$ distributions,

$$\pi_Y(\theta) \propto \pi(Y|\theta)\pi_0(\theta). \tag{1}$$

In such cases, numerical approximations must be sought. One popular method is *sampling*, i.e. generating a set of points $\{x_n\}_{n=1}^N$ distributed according to the sampled distribution $\pi(x)$ [1] and which therefore approximate $\pi(x)$. A powerful class of algorithms to sample probability distributions is called Markov Chain Monte Carlo (MCMC). In MCMC, sampling of the distribution occurs by constructing a Markov Chain in the sample space that approximates the probability distribution being sampled [2, 3]. In contrast to simpler Monte Carlo methods (direct sampling, rejection sampling), MCMC benefits from more efficient exploration of the sample space, which becomes especially crucial in higher dimensions where the sample space grows exponentially, a phenomenon referred to as the curse of dimensionality [4].

A popular MCMC algorithm that provides a systematic way of constructing the Markov chain, and of which many other can be considered special cases of, is the Metropolis-Hastings algorithm [5, 6]. It can be stated in a few steps:

0. Initialize state $X_0 \sim \pi_0(X)$.

1. Generate candidate state $X' \sim p(X', X_{i-1})$.

2. Generate random number $r \sim U[0, 1]$.

3. If $r \leq A(X', X_{i-1})$ set $X_i \leftarrow X'$, else set $X_i \leftarrow X_{i-1}$.

4. Go to 1.

The map $A(\cdot, \cdot)$ in step 3 is called the Metropolis-Hastings *acceptance probability*,

$$A(x, y) = \min\left\{1, \frac{\pi(y)p(y, x)}{\pi(x)p(x, y)}\right\}, \tag{2}$$

where $p(y, x)$ is called the *proposal distribution* and gives the probability of transition from state x to state y. The Metropolis-Hastings algorithm generates a Markov chain of states $X_1, X_2, ...$ with stationary distribution π. If the proposal distribution p is symmetric eq. (2) simplifies to

$$A(x, y) = \min\left\{1, \frac{\pi(y)}{\pi(x)}\right\}. \tag{3}$$

In this project, I designed a C++ program to plot and sample simple probability distributions by means of the Metropolis MCMC method. Because the accuracy and precision of such calculations is generally limited by their sheer speed, the program offers multithreading capabilities to fully utilise the available computational resources.

The program makes use of the graphing utility **gnuplot** available for Windows under http://www.gnuplot.info/download.html. On Linux, it can be installed via the commands:

```
$ sudo yum check-update
$ sudo yum install gnuplot
```

2 Code design and implementation

The advanced code features implemented in the program include:

- Class hierarchy
- Class polymorphism
- Smart pointers (`unique_ptr`, `shared_ptr`)
- Dynamic pointer casting
- Static data types
- Function templates
- Lambda expresions
- Containers (`unordered_map`)
- Exception handling
- Multithreading

- Pseudorandom number generation via the Mersenne Twister
- Plotting functionality via gnuplot

The program contains a **class hierarchy** of probability distributions with an abstract base class `distribution` which has two derived classes, `continuous_distribution` and `discrete_distribution`. Both derived classes have several derived classes of their own, corresponding to a distinct probability distribution. Each derived probability distribution contains their parameters as private variables, as well as a probability density function `pdf` and sampling function `sample` that generates numbers distributed according to the given distribution.

The probability distributions are instantiated by means of **smart pointers** to the class, either with `std::unique_ptr` or, more frequently, `std::shared_ptr`. The reason for the choice of `std::shared_ptr` over `std::unique_ptr` becomes clear if we examine the function `make_distribution` in utils.h, which contains an `std::unordered_map` (one of two **containers** used in the program, excluding `std::arrays` and `std::vectors`) that maps the names of the distributions, as given by the user in the terminal, to their associated class constructors, defined via an `std::initializer_list`, beginning at line 80. `std::unordered_map` must call a move or copy constructor to store the keys and values. For the case of `std::unique_ptr`, the copy constructor is, naturally, deleted, while the move constructor attempts to move the values in the initializer list, which are `const` and can't therefore be moved. As such, unless one intends to instantiate a specific distribution at compile-time (as is the case with the `random_uniform` pointer used in all Metropolis calculations), `std::shared_ptr` must be used over `std::unique_ptr`.

To leverage **polymorphism**, the distributions are declared as pointers to the `distribution` base class in a first instance, and subsequently specialised to a specific desired distribution through **dynamic pointer casting**. To avoid overloading functions to suit both types of probability distributions, **function templates** were employed, as demonstrated in `utils.h`.

There is also a separate `markov_chain` class, which describes the Markov chain constructed in a MCMC simulation. The class contains a **static data type** `number_of_chains` which counts the number of present instances of the class, which is a useful quantity to have access to for debugging purposes, particularly in case of **multithreading**. The sampling of the distributions can occur in parallel by instantiating a number of `markov_chain` objects and `std::thread` objects and performing a separate simulation on each thread simultaneously. This is achieved in part by the use of a **lambda expression** in line 260 of file `interface.cpp`.

In order for any Monte Carlo method to work, one needs to be able to generate random numbers. In this program, this is achieved via the state-of-the-art pseudorandom number generator, the **Mersenne Twister**, available in the C++ standard library via the class `std::mt19937`.

The plotting of the probability density functions and produced histograms is done with **gnuplot**. The plotting instructions are piped to gnuplot via the `gnu_pipe.cpp` file, which can be modified easily by the user to include additional gnuplot commands to customise the resultant graphs. To minimise potential gnuplot stability and compatibility issues, the current implementation of the program executes the given gnuplot commands when exiting the program. If the reader wishes to generate the plots dynamically, they

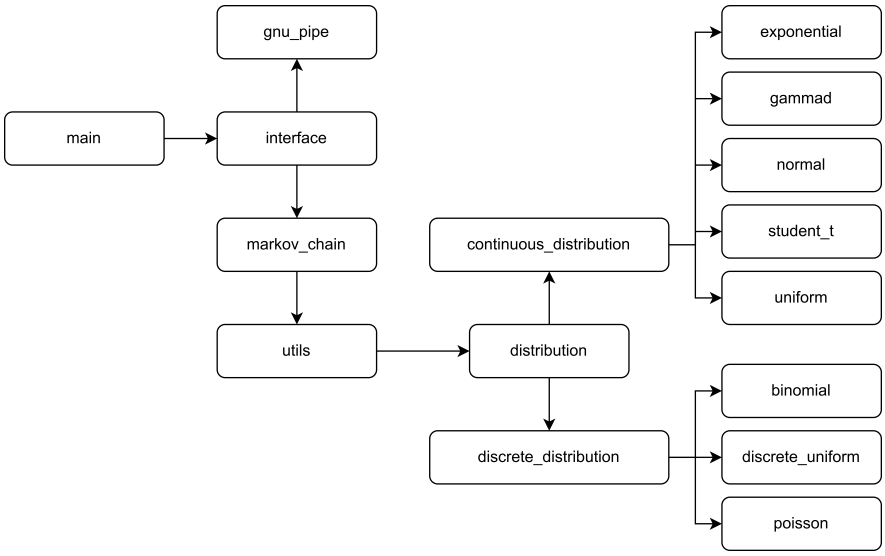

Figure 1: Flowchart of the file and class structure of the program. This flowchart was produced by means of the free software available under `https://app.diagrams.net/`.

are invited to add the command "`fflush(gnupipe);`" to the end of both functions in the `gnu_pipe.cpp` file and recompile the program.

To avoid problems stemming from setting incorrect values for the parameters in the simulations, **exception handling** was employed extensively, as demonstrated in the `interface.cpp` file.

The file and class structures of the program are illustrated by the flowchart in fig. 1.

3 Results

We shall now demonstrate three example calculations using the program. During a MCMC simulation or plotting of the probability density function, the data is saved to a file in the `data` folder and can be accessed by the user at any time, unless overwritten by a subsequent calculation.

3.1 Sampling of a continuous distribution

To demonstrate the principal features of the program, let us type the command `sample` into the terminal. We are immediately asked to specify the target distribution, i.e. the distribution we want to sample. In this example, we shall attempt to sample the `gamma` distribution, with `2` set as the value of the shape and rate parameters of the distribution.

We also need to specify the proposal distribution p discussed earlier. In the current version of the program, the proposal distribution must be symmetric to satisfy detailed balance and ensure that the correct distribution is sampled. Good choices include the `normal` (with mean 0) and `student_t` distribution (the Student's t-distribution has longer tails than the normal distribution, so will propose states farther away from a current state more often than a normal distribution).

We are then asked to specify the number of steps in a simulation, followed by a value for `decorrelation`, which specifies the period between recorded states of the Markov chain. MCMC calculations routinely suffer from a high degree of autocorrelation, which can be mitigated by setting a high value for `decorrelation`. It is important to note that, during histogram construction, the program needs to sort all concatenated Markov chain states at once. As such, one needs to ensure that sufficient memory is available, otherwise the program will be unable to plot the sampled distribution.

We are also asked to set a value for the number of threads we wish the program to make use of to perform the calculation. Finally, we need to choose a name for the file in which the samples will be recorded. After completing the calculation, the program calculates the acceptance ratio across all parallel computations. If the acceptance rate is too low, this means that the Markov chains doesn't explore the sample space uniformly as it rarely changes states. If the acceptance rate is too high, the jumps between states are too conservative and the sample space isn't explored efficiently. For one-dimensional problems, the optimal Metropolis acceptance rate is 44%.

To visualise the evolution of the recorded Markov chain states, we shall also type in the command `plot_mc`, followed by the same name we previously set for the data file. To see the actual sampled distribution, we then type in the command `histogram`, again giving directions to the file name. We are then asked to set the number of bins in the histogram we are about to produce. The reader can choose to manually set the x-range over which the plot is to be made by typing in the command `custom` in the next step. Alternatively, the interval can be set automatically by typing in `auto`.

To gauge the quality of the histogram, we shall compare it to the exact probability density function of the gamma distribution. To plot it, we type in the command `plot`, followed by specifying the distributions and its associated parameters, as well as the file name in which the data shall be stored. We also need to manually set the interval over which we want the distribution to be plotted.

A complete sample terminal input and output for the example is shown in fig. 2. Upon typing in the command `exit`, gnuplot should now plot all the requested diagrams, similar to figs. 3-5.

3.2 Sampling of a discrete distribution

The program can be used to also sample from discrete distributions. Because only discrete values can be attained, we must use the `discrete_uniform` distribution as the proposal distribution. We can repeat the procedure from the previous example for a chosen discrete distribution, e.g. a Poisson distribution with $\lambda = 3$. This yields the histogram shown in fig. 6. The probability density function for a discrete distribution can also be plotted as a step function. To do this, the user can type in `plot` as in the previous example. For the Poisson distribution with $\lambda = 3$ this results in the plot shown in fig. 7.

3.3 Sampling of a posterior distribution

In the final example we shall attempt to sample the posterior distribution from a chosen pair of prior and likelihood distributions. The choice for the pairing needs to be sensible—the mixing of continuous and discrete distributions is not recommended. For this example, we shall consider an ex-

ponential distribution with $\lambda = 1$ as the prior and a normal distribution with $\mu = 2, \sigma = 1$ as the likelihood. Typing in the commands `posterior` and `histogram`, along with the values for the requested variables, into the terminal yields the plot shown in fig. 8.

4 Discussion and outlook

While the efficiency and versatility of the program exceeded my personal expectations, during test-running it quickly became evident that the main limitation in obtaining more precise estimates of the sampled distributions lies in the histogram construction, which needs to sort the recorded Markov chain states and which may be difficult to accomplish for more than a few hundred thousand states due to limited memory. This can be mitigated in a future implementation of the program by the use of external sorting algorithms such as merge sort, which makes use of external memory (e.g. a hard drive) to store and sort the data by chunks.

In more challenging statistical problems, the posterior distributions are often multidimensional or depend on an input of experimental or model observations. The program can be extended in functionality to such classes of problems straightforwardly.

Metropolis MCMC is only one of many popular MCMC algorithms. An obvious extension of the program is to introduce the Hastings correction to the acceptance probability, to allow for non-symmetric proposal distributions. Alternative algorithms, such as Hamiltonian MCMC or the preconditioned Crank–Nicolson algorithm can be implemented as well, benefiting from the already developed program structure.

References

[1] A. M. Stuart. Inverse problems: A bayesian perspective. *Acta Numerica*, 19:451–559, 2010.

[2] Jari Kaipio and Erkki Somersalo. *Statistical and Computational Inverse Problems*. Springer, Dordrecht, 2005.

[3] Daniela Calvetti and Erkki Somersalo. *Introduction to Bayesian Scientific Computing: Ten Lectures on Subjective Computing*. Springer Science Business Media, LLC, 2007.

[4] Christian P. Robert. *The Metropolis–Hastings Algorithm*, pages 1–15. American Cancer Society, 2015.

[5] 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.

[6] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

```
sample
Set target distribution: gamma
Set shape parameter: 2
Set rate parameter: 2
Set proposal distribution: student_t
Set number of degrees of freedom: 1
Set number of trials: 1000000
Set decorrelation: 50
Set number of threads: 8
Set file name (w/o extension): gamma
thread 7 started...
thread 8 started...
thread 3 started...
thread 6 started...
thread 2 started...
thread 5 started...
thread 1 started...
thread 4 started...
Metropolis acceptance: 5631762/8000000 [70.397%]
Concatenating data...
Done

plot_mc
Select file name (w/o extension): gamma
Done

histogram
Select file name (w/o extension): gamma
Set number of bins: 200
Set x-range [auto/custom]: auto
Done

plot
Set distribution: gamma
Set shape parameter: 2
Set rate parameter: 2
Set file name (w/o extension): gamma_pdf
Set minimum x: 0
Set maximum x: 30
Done

exit
```

Figure 2: Sample terminal input and output from example 1.

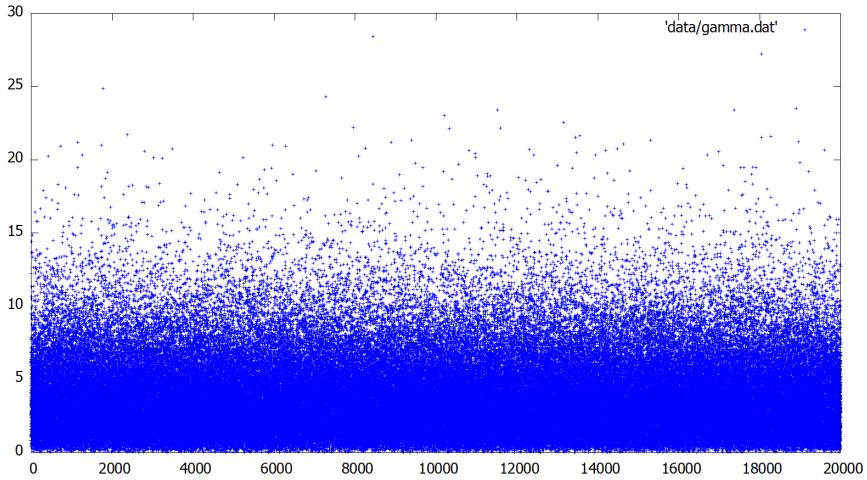

Figure 3: A plot of the Markov chain states as a function of step number of each chain in example 1.

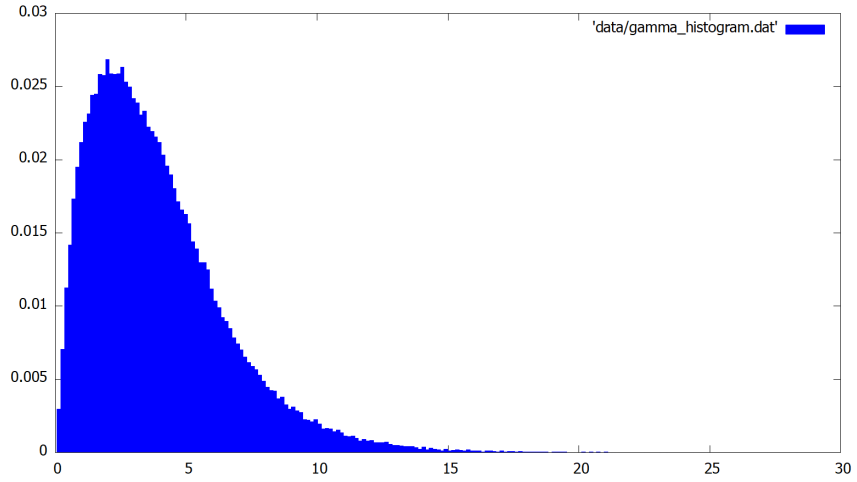

Figure 4: Histogram of the obtained samples in example 1, shown in fig. 3.

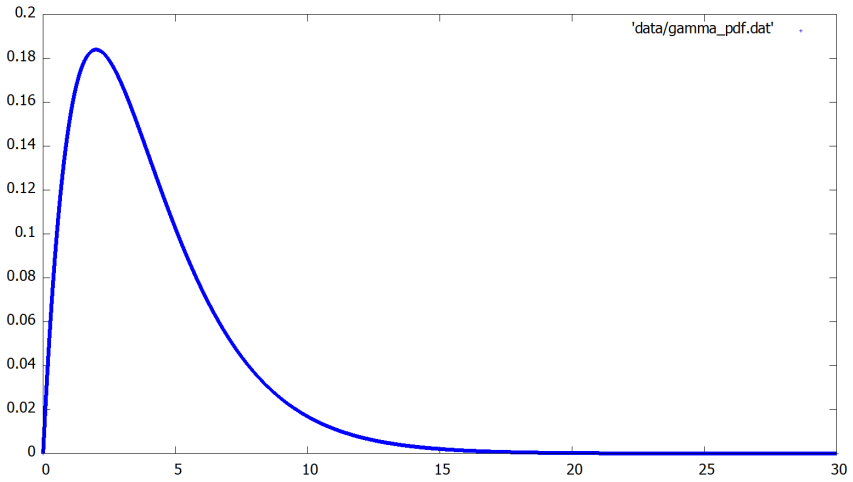

Figure 5: An exact plot of the probability density funtion for the gamma distribution with parameters $\alpha = 2$ and $\beta = 2$ considered in example 1. The plot shows very good agreement with the histogram in fig. 4 up to a constant of proportionality.

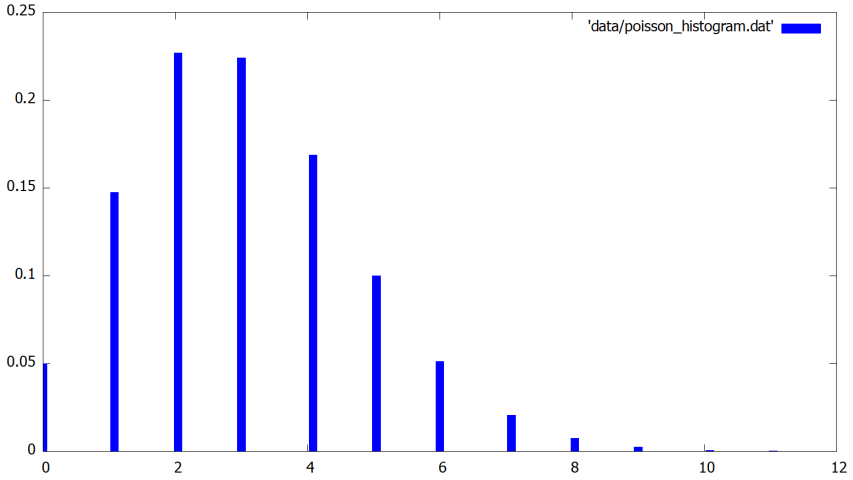

Figure 6: Histogram of the obtained samples of the Poisson distribution considered in example 2.

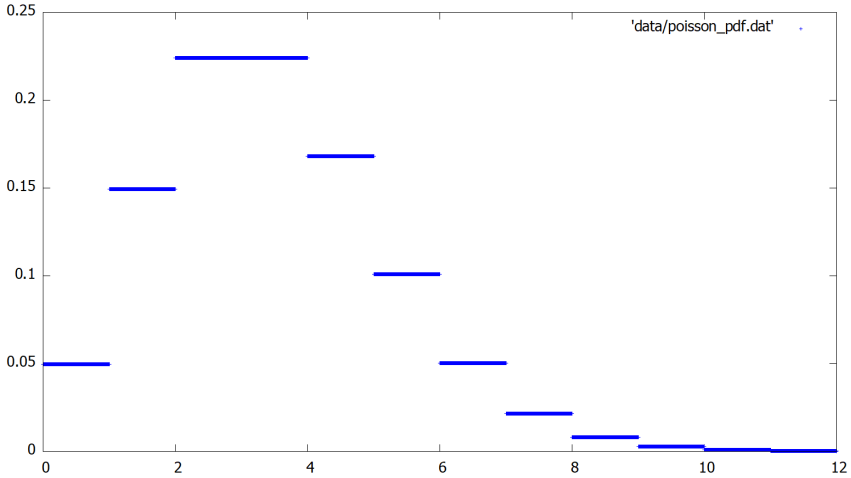

Figure 7: The probability density function of the Poisson distribution considered in example 2, plotted as a step function. The plot is consistent with the histogram in fig. 6.

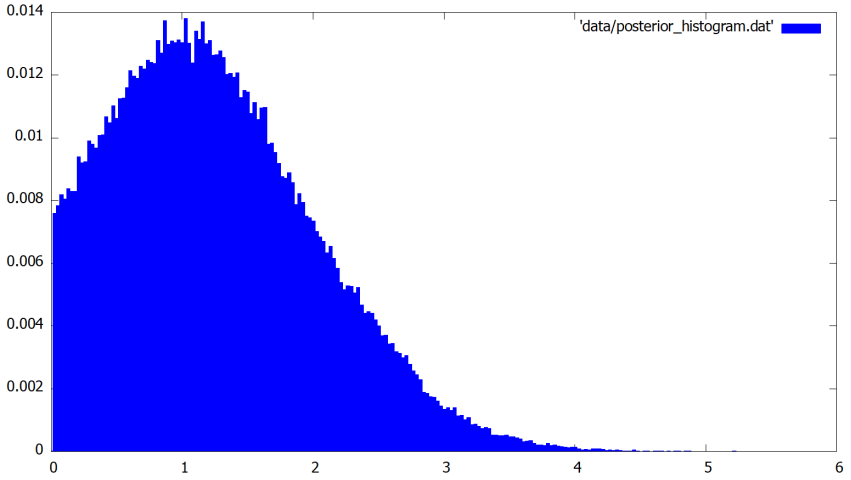

Figure 8: Histogram of the obtained samples of the posterior distribution considered in example 3.