Master Thesis Report

Bayesian inference methods for parameter estimation

Implementation and benchmarking

Sebastian Johansson

May 24, 2023

Student
Spring or Fall 2023
Master Thesis, 30 ECTS
Master of Science in Computer Science, 300 ECTS

Abstract

In this study, three implementations of the nested sampling algorithm have been implemented, in the programming language Rust, compared and benchmarked. These variations consist of a classic version, closely resembling the first appearance of the algorithm, a version that produces samples from a single ellipsoid, and a version that uses multiple ellipsoids to generate its samples. These versions where compared to each other and counterparts from two Python libraries, Nestle and pymultinest. Testing the variations of the algorithms found that the multi ellipsoids sampler is the most versatile alternative and when comparing wall clock time, The Rust implementation of the multi ellipsoid sampler ran up to 79 times faster than its Nestle counterpart and up to 51 times faster than pymultinest. Running the rust implementations in parallel with eight threads proved to be slower in most examples, but in computationally difficult problems, the single ellipsoid sampler received a speedup of up to 3.50 while the multi ellipsoid sampler got a speedup of up to 1.86 when running benchmarks with eight threads rather than one.

Contents

1 Introduction **1**

2 Background **2**
2.1 Bayesian statistics . . . 2
2.2 Bayesian Inference . . . 3
2.2.1 Parameter estimation . . . 3
2.3 Markov Chain Monte Carlo . . . 3
2.3.1 Metropolis Hastings . . . 4
2.4 Ellipsoid sampling . . . 4
2.5 K-means clustering . . . 4
2.6 Rust . . . 5
2.6.1 The linear algebra crate nalgebra . . . 5
2.6.2 Rayon . . . 5

3 Previous work **6**
3.1 Nested sampling . . . 6
3.2 A Nested Sampling Algorithm for Cosmological Model Selection . . . 7
3.3 Efficient Bayesian inference for multimodal problems in cosmology . . . 8
3.4 Multinest . . . 8

4 Problem formulation **10**

5 Method **11**
5.1 Implementations . . . 11
5.1.1 Stopping criterion . . . 11
5.1.2 Noteworthy libraries . . . 11
5.2 Prior transform . . . 12
5.3 System specifications . . . 12
5.4 Benchmarks . . . 12
5.4.1 Eggbox . . . 12
5.4.2 Gaussian shells . . . 14
5.4.3 Lotka-Volterra . . . 14
5.4.4 Rosenbrock . . . 15

6 Results and discussion **17**
6.1 Eggbox . . . 17
6.2 Gaussian shells . . . 21
6.3 Lotka-Volterra . . . 25
6.4 Rosenbrock . . . 29

7 Conclusion **34**

References

List of Figures

1 Visualization of Z intergral 7
2 Visualization of an ellipsoid around 10 live points 8
3 Visualization of two ellipsoids forming around 10 live points 8
4 The true surface of the eggbox likelihood function 13
5 The true surface of the Rosenbrock function 16
6 Rust corner plot of eggbox 17
7 Nestle corner plot of eggbox 18
8 Eggbox times 19
9 Eggbox parallel times 19
10 Eggbox volumes 20
11 Eggbox likelihood calls 20
12 Rust corner plot of Gaussian shells 21
13 Nestle corner plot of Gaussian shells 22
14 Gaussian shell times 23
15 Gaussian shell parallel times 23
16 Gaussian shell volumes 24
17 Gaussian shell likelihood calls 24
18 Rust corner plot of Lotka-Volterra (single ellipsoid sampler) 25
19 Rust corner plot of Lotka-Volterra (multi ellipsoid sampler) 25
20 Neslte corner plot of Lotka-Volterra (multi ellipsoid sampler) 26
21 Lotka-Volterra times 26
22 Lotka-Volterra parallel times 27
23 Lotka-Volterra volumes 28
24 Lotka-Volterra likelihood calls 28
25 Rust corner plot of Rosenbrock 29
26 Nestle corner plot of Rosenbrock 30
27 Rosenbrock times 31
28 Rosenbrock parallel times 31
29 Rosenbrock volumes 32
30 The number of likelihood calls at each iteration 32
31 Rosenbrock likelihood calls 32

1 Introduction

Bayesian inference is a statistical method that involves updating our prior beliefs or knowledge about a parameter or hypothesis based on new observed data. It allows us to incorporate both prior information and new data to make more accurate predictions or decisions.

An example of Bayesian inference being used is to find the rate of transmission in a pandemic. In this case, Bayesian inference is used by starting with a prior belief about the distribution which informs the rate of transmission, based on historical data or other information. Then, as new data on the number of cases is collected, the prior belief is updated using Bayes' theorem to form a posterior distribution that better reflects the rate of transmission in the pandemic.

There are several libraries that are used to perform Bayesian inference, a lot of which are for the programming language Python such as Nestle [1]. These use variations of the nested sampling algorithm to find the evidence of a model, a numerical value representing how well the model performs on the data, with posterior samples as a byproduct. There are other ways of performing Bayesian inference such as Markov Chain Monte Carlo (MCMC) methods. These, however, do not calculate the evidence of a model, which makes them less suitable for model selection.

Performing Bayesian inference on model that are hard to compute, such as ordinary differential equations (ODE) and partial differential equations (PDE), can be difficult in a language like Python since there is no guarantee that the libraries are compiled at the same level. ODE solvers might actually implement its functions in another low level language like C which is why it appears to be fast when used for a single problem. The same goes for MCMC sampling. The sampling can appear to be fast when the routine generating that sample is trivial. However, when we combine these parts, the ODE solver and MCMC sampler, such environments suffer from high Python overhead due to frequent calls to these two separate, individually fast, libraries from Python workspace. The reason for this is because of differences in the languages, such as Python being interpreted while C is compiled, Python being dynamically typed while C is statically typed and differences in data structures and memory layout. Due to these differences, frequent type checks and data conversions are performed at runtime which results in overhead that can be quite significant. This problem is hard to avoid in a language like Python as the likelihood function, necessary to perform Bayesian inference, is highly dependent on the problem at hand and is therefore left to be user defined. When the runtime is a concern, ODE systems and Bayesian samplers must be implemented at the same level before compilation. In this project, different nested sampling algorithms will be investigated, implemented in the Rust programming language, and then compared to each other and their Python counterparts in hopes of finding which algorithms and implementations perform best in various metrics.

2 Background

This section provides the necessary concepts to understand previous research and the problem at hand.

2.1 Bayesian statistics

Bayesian statistical methods utilize Bayes' theorem to update the probabilities of events after new data or evidence is obtained. In contrast to classical or frequentist statistics, which assumes probabilities represent the frequency of seemingly random events over a long run of repeated trials, Bayesian methods take into account all available information when calculating probabilities [3].

Bayes' theorem can be derived from the definition of conditional probability:

$$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}$$

where $P(X|Y)$ is the conditional probability of the event X given Y. $P(X \cap Y)$ is the joint probability of events X and Y, and $P(Y)$ is the probability of event Y. We can multiply both sides with $P(Y)$, and since $P(X \cap Y)$ is the same as $P(Y \cap X)$, we also know that

$$P(X|Y) \times P(Y) = P(X \cap Y) = P(Y|X) \times P(X)$$

By then dividing the sides by $P(Y)$ again we get Bayes' theorem:

$$P(X|Y) = \frac{P(Y|X) \times P(X)}{P(Y)} \tag{1}$$

This formula states that the probability of event X given event Y is equal to the probability of event Y given X, multiplied by the ratio of the probabilities of event X and Y.

In a simple example, imagine that 5% of the patients in a hospital are infected with a certain disease and that 80% of infected patients have a cough. However, in the population 10% of patients have a cough. We can use this information to calculate the risk of a patient being infected given that they have a cough. If we define

- X as the event that the patient has the disease, and
- Y as the event that the patient has a cough.

Then we have the probabilities

- $P(X)$ = the probability that the patient has the disease,
- $P(Y)$ = the probability that the patient has a cough, and
- $P(Y|X)$ = the probability of a patient with the disease having a cough.
- $P(X|Y)$ = the probability of a patient having a cough being infected with the disease.

With this information we can calculate the probability of a patient being infected with the disease given that they have a cough with Bayes' theorem from Equation (1)

$$P(X|Y) = \frac{P(Y|X) \times P(X)}{P(Y)} = \frac{0.8 \times 0.05}{0.1} = 0.4$$

We get that there is a 40% risk of a patient with a cough carrying the disease.

2.2 Bayesian Inference

Statistical inference is the process of using samples to infer properties of a larger population. It is typically used when studying phenomenon such as the effects of a new medication or public opinion.

Bayesian inference is a method of statistical inference which makes use of Bayes' theorem to update the probabilities of events. In Bayesian inference, we give names to the different probabilities in Bayes' theorem and often use different symbols to represent the events. Based on Bayes' theorem in Equation (1), we can state:

$$P(H|E) = \frac{P(E|H) \times P(H)}{P(E)}$$

where

- H represents a hypothesis we want to test.
- E represents new data which was not taken into consideration when calculating the posterior probability.
- $P(H)$ represents the posterior probability. This is the previous probability of the hypothesis H before any new data has been taken into consideration.
- $P(H|E)$ is called the posterior probability and represents the updated probability in light of the new data.
- $P(E|H)$ is called the likelihood and is the probability of our new data E given our hypothesis H. Intuitively, this can be viewed as how well our new data correlate with the hypothesis.
- $P(E)$ is called the evidence or the marginal likelihood.

This states that the posterior can be derived in light of new data with a prior probability and some likelihood function which in some way needs to return a numerical value representing how well data fits the current hypothesis.

2.2.1 Parameter estimation

Parameter estimation is the process of determining the values of unknown parameters in a statistical model based on observed data. The goal of parameter estimation is to find the values of these unknown parameters that best fit the available data. This can be done using Bayesian inference, where the parameters θ of a model M are treated as the hypothesis H that needs to be tested.

In Bayesian inference, the likelihood function $P(E|\theta)$ measures how well the model with parameters θ fits the observed data E. The prior probability distribution $P(\theta)$ represents our initial beliefs about the possible values of the parameters before observing any new data. It can be based on domain knowledge or previous data. The prior can be updated by incorporating the likelihood function using Bayes' theorem, which leads to the posterior probability distribution $P(\theta|E)$.

In practice, it may not always be possible to find an analytical solution for the posterior distribution. In such cases, numerical methods like MCMC can be used to sample from the posterior distribution.

2.3 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a computational method used to generate random samples from a probability distribution when the distribution is difficult to sample from directly. It works by simulating a Markov chain, which is a sequence of random variables

where the distribution of each variable depends only on the previous variable in the sequence. The goal of MCMC is to construct a Markov chain that has the desired probability distribution as its equilibrium distribution, so that samples from the chain can be used to estimate properties of the distribution.

2.3.1 Metropolis Hastings

Metropolis Hastings is an MCMC method that works by constructing a Markov chain whose stationary distribution is the desired target distribution. At each step of the chain, a candidate sample is generated by proposing a move from the current sample. The proposed move is accepted or rejected based on a probability ratio, which is a function of the proposal distribution and the target distribution [6].

2.4 Ellipsoid sampling

In mathematics, an ellipsoid can be represented the quadratic equation

$$(x - c)^T A(x - c) = 1$$

where

- x is a point on the ellipsoid surface in n dimensions,
- c is the center of the ellipsoid such that the vector (x-c) represents the displacement of the ellipsoid from the center and
- A is a positive-definite matrix whose eigenvectors represent the directions of the ellipsoid's principal axes, and the square roots of the inverse of the eigenvalues represent the semi-axis lengths of the ellipsoid

With this information, sampling from the ellipsoid surface can be done by generating points uniformly between -1 and 1 on a hypercube, then rejecting the samples that would not be within the hyper sphere contained in the cube by checking if the euclidean distance to the center is more than 1. Then transforming the points with the axes, gathered from the eigenvectors of A, and positioning the points correctly by adding the center vector c.

This can be helpful as it is a much faster way of sampling from a surface rather than using an MCMC method. An ellipsoid can be constructed around a set of points and then be used to generate samples.

2.5 K-means clustering

K-Means Clustering is an algorithm that clusters a set of points in n dimensions into k different clusters where the points closest and seemingly belonging with each other end up in the same cluster. There are 5 steps to the clustering algorithm.

1. Choose the number of clusters k. The value of k might already be known, but if it is not, the algorithm can be run multiple times until a certain convergence criteria has been met.
2. Select k random points as the cluster centroids. These center positions could be completely randomized or informed by some previous knowledge.
3. Assign each point to its closes cluster centroid. In this step, the distance from each point to each cluster centroid is calculated in order to assign each point to its closest centroid.
4. Recalculate the centroids. In this step, the centroids are recalculated according to the points in its cluster. The average of the points is calculated and then assigned to the centroid.

5. Repeat step 2-3 until a stopping criteria has been met. This criteria could be that a maximum number of iterations has been reached, the position of the cluster centroids stop moving, or that the points no longer change what cluster it is assigned to.

This is useful when identifying groups without having to know why any individual point belong there. When sampling from a set of points, working with clusters gets easier as it has the potential to eliminate any dead space between these clusters by concentrating on the populated areas and ignoring the empty ones. K-means clustering has applications in problems such as document clustering, recommendation engines, and image segmentation [12].

2.6 Rust

Rust is a high level multi paradigm programming language which emphasizes good performance and type safety. It is able to enforce memory safety without the use of a garbage collector, used by other programming languages like Java. Rust is able to do this by using a system of ownership and borrowing to ensure that the memory is managed correctly already at compile time. This system is called the "borrow checker". Being a compiled language, Rust benefits from the performance of other languages that can be compiled to assembly such as C and C++. Combining the performance of these languages with the memory safety offered by the borrow checker and other modern features, such as pattern matching and a functional paradigm, Rust has become an appealing alternative for programmers in many different fields and as of 2022 it is currently on its seventh year of being ranked the most loved programming language in Stack overflows developer survey [17]. Rust is however still in early development and there are problems such as a steep learning curve due to its complexities, and long compile times.

2.6.1 The linear algebra crate nalgebra

`nalgebra` is a linear algebra crate, a library under the cargo package manager, for Rust. It targets general purpose linear algebra, with most of the basic vector and matrix operations covered, as well as real-time computer graphics and real-time computer physics [9].

2.6.2 Rayon

Rayon is another Rust crate which aims to simplify the conversion of sequential computations into parallel. This is done via the parallel iterator. In Rust, an iterator is a type that provides a sequence of variables. There are functions that can be applied to this type such as **map** which operates on each variable in the iterator. In many cases, the existent sequential iterator can be replaced with the parallel iterator from Rayon to parallelize parts of a Rust program [10].

3 Previous work

This section provide the previous work on the subject and forms a sort of timeline over the chosen algorithms development.

3.1 Nested sampling

In 2004, John Skilling introduced an algorithm for calculating the marginal likelihood with posterior samples as a byproduct in the paper Nested sampling [15]. This algorithm uses the likelihood function $\mathcal{L}$ with the prior π to calculate the evidence Z and the posterior P. This is done by evolving a collection of N objects, where N can be chosen to be small for speed or large for accuracy. These objects shape the prior mass according to the shape of the likelihood contours. Each object gets evaluated by a user defined likelihood function and at each iteration, Nested sampling finds the "worst" object, according to the likelihood function, and replaces it with a newly generated one, constrained by the likelihood of the worst object.

Skilling defines x(L) as the prior mass with likelihoods greater than L,

$$x(L) = \int_{\mathcal{L}(\theta)>L} \pi(\theta) d\theta$$

This constructs a decreasing function where dx is the prior mass associated with the likelihoods in the range $[L, L + dL]$. Using $L(x)$, the inverse of $x(L)$, the evidence can be constructed as

$$Z = \int_0^1 L(x) dx$$

The task is to calculate this value of Z. This is done by constructing a decreasing sequence of x values, starting at 1 and converging upon 0 like

$$1 = x_0 > x_1 > x_2 > ... > x_\infty = 0$$

Then evaluating the likelihood at each value,

$$L_i = L(x_i), \quad i = 1, 2, 3, ...$$

And then associating each L_i with its outward interval $[x_i, x_{i-1})$ to arrive at an estimate

$$Z \approx \sum_{i=1}^{\infty} L_i h_i, \quad h_i = x_{i-1} - x_i$$

where the symbol "$\approx$" denotes an estimate, in this case via the trapezoid rule. It is this method of peeling off layers of the likelihood contour with the likelihood constraint that prompts the name **Nested sampling**. A visualization of the integral that is being approximated can be seen in Figure 1. The steps of the algorithm are

1. Generate N objects θ from the prior.

2. Record the lowest likelihood value L at this iteration.
3. Replace this object with a new one with the constraint $\mathcal{L}(\theta) > L$.
4. Accumulate the evidence Z with the trapezoidal rule.
5. Repeat step 2-4.

This can not go on forever though, so after say k steps, the procedure contains N objects drawn from $\mathcal{L}(\theta) > L$, which Skilling suggests averaging, weighting by the remaining mass x_k, and adding to the evidence Z. This leads to the complete estimate of the evidence Z as

$$Z \approx \sum_{i=1}^{k} L_i h_i + \overline{L} x_k$$

The termination criteria could be to either set the number of steps k in advance. Another termination criteria is to set a pre-specified tolerance level and terminate the algorithm when the evidence error fell below this level. The evidence error can be calculated as the difference between the maximum and minimum values of the evidence estimates obtained during the nested sampling run.

Skilling suggests using MCMC to generate the new objects from the prior. In later material, Skilling shows code examples in C using a metropolis hasting method to generate the new samples where the initial point used in the metropolis hasting algorithm is picked randomly from the N objects in the prior [16] [14].

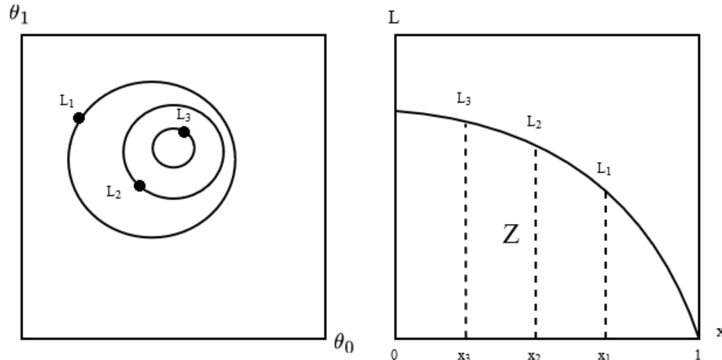


Figure 1: Visualization of Z intergral

3.2 A Nested Sampling Algorithm for Cosmological Model Selection

In a paper by Pia Mukherjee, David Parkinson and Andrew R. Liddle [8], another method of generating objects from the prior when calculating the evidence Z is proposed. The method is based on Skillings Nested sampling, but samples new objects directly from the prior instead of using the metropolis hasting algorithm. This is done by calculating the covariance of the live points N, rotating the coordinates to the principal axes and creating an ellipsoid which touches the maximum coordinate values of the points. To allow for the likelihood contours not being exactly elliptical, the ellipsoid is then expanded by a constant enlargement factor, chosen by the user. New points are then sampled from the ellipsoid when generating new objects for the nested sampling algorithm. A simple visualization of this process can be seen in Figure 2.

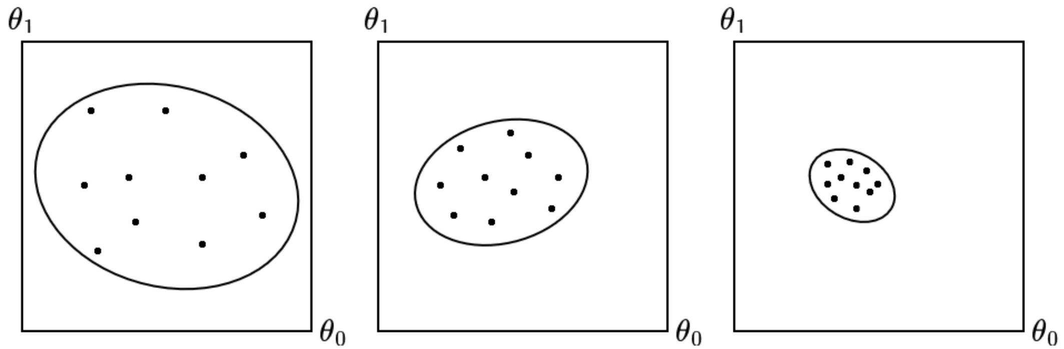


Figure 2: Visualization of an ellipsoid around 10 live points

3.3 Efficient Bayesian inference for multimodal problems in cosmology

In this paper by J.R. Shaw, M. Bridges and M.P. Hobson [13], The methods used by Skilling and Mukherjee et al. are further developed on to be more efficient for multimodal likelihood surfaces. When placing an ellipsoid around the live points, if there are multiple peaks in the likelihood surface, generating new objects becomes inefficient due to the difference in the sampling area and the likelihood contour. That is, there will be a lot of dead space between the peaks where objects can not be generated from due to the likelihood constraint. This problem gets worse in later iterations of the algoritm as he likelihood constraint focus in on the local maximum points while the single ellipsoid has to cover multiple of these points. The method proposed in this paper solves this by using multiple ellipsoids to draw new points from.

When updating the ellipsoids, Shaw et al. suggests recursively running the K-means algorithm with a k value of 2 to subdivide the live points into clusters that each will have their own ellipsoid that new objects can be generated from. The clusters will subdivide recursively until one of two cases have been met. For a subdivide to be accepted, the total volume of the new ellipsoids must be less than some fraction of the pre-clustered ellipsoid. They also have to be sufficiently separated by some distance, in order to avoid overlapping regions. A simple visualization of this process can be seen in Figure 3.

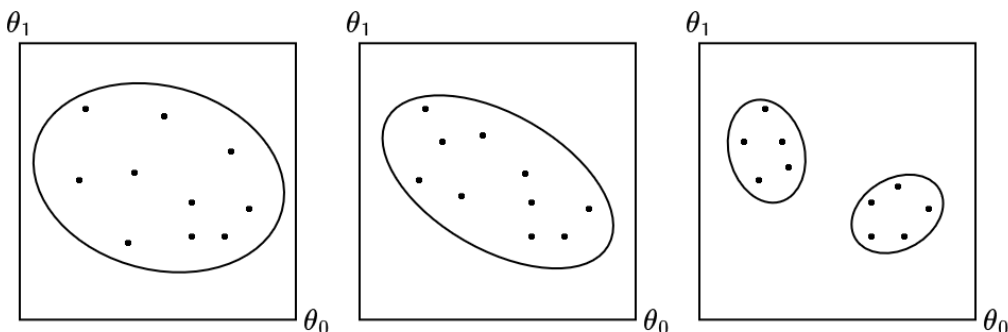


Figure 3: Visualization of two ellipsoids forming around 10 live points

3.4 Multinest

Multinest by F. Feroz, M.P. Hobson and M. Bridges [5] is a public release of a multimodal nested sampling algorithm **Multinest** which works in the same way as the algorithm de-

scribed in Section 3.3. It is mainly implemented in the programming languages C and FORTRAN but bindings exist for other languages such as Python with the library `pymultinest` [4]. It also supports parallelization with MPI which allows it to generate a list of objects to draw from when replacing the worst object, out of the N live points, according to the likelihood function. This means that it can generate and calculate the likelihood for several points in parallel and then simply draw from this list.

4 Problem formulation

We want to know how well different implementations of the nested samlping algorithms preforms in the Rust programming language, how they compare to eachother and their Python counterparts. The variations of the algorithm that we are interested in testing are:

- a classic version of the algorithm using metropolis hasting to generate new samples,
- a single ellipsoid version that samples points from an elipsoid containing the prior samples,
- and a multi ellipsoid version that samples points from a list of ellipsoids centered around clusters in the prior generated by the kmeans algorithm.

These are commonly implemented in Python libraries and can be found in packages like Nestle and pymultinest. This allows us to directly compare these algortihms to one or more of their Python counterparts which makes it easy to measure the performance improvement of moving to Rust.

The nested sampling algorithm has been chosen for its ability to perform model selection as it calculates the evidence Z. With this value, we can run the algorithm with several models and compare the evidence to determine which model best fits the observed data.

When comparing the Rust implementations to their Python counterparts, the metric we are interested in comparing is wall clock time. Other metrics should not be impacted as the speed of the languages and the implementation techniques utalized are the changing factor. When comparing Rust implementations of the algorithms against eachother we meassure:

- **Wall clock time**, how long it takes the algorithm to reach its stopping criteria.
- **Number of likelihood calls**, how many likelihood calls the sampler has to make.
- **Prior volume**, how large the volume is that the points are generated from.

The goal is to find out which implementations perform the best in a selection of problems and what metrics might affect the perfomance.

5 Method

To measure the desired metrics discussed in Section 4, the classic, single ellipsoid and multi ellipsoid variations of the algorithms have been implemented in Rust and will be tested on a few cases that were also implemented in Python libraries with the same algorithms.

5.1 Implementations

The Rust implementations mirror the Nestle ones which are also based on the different variations discussed in Section 3. The classic version is based on the first version Skilling proposed, details on this can be seen in Section 3.1. Instead of choosing a random element from the prior for the start position of the metropolis hasting algorithm, the Rust implementation draws this initial point from an ellipsoid similar to how the single ellipsoid sampler works. The number of steps it takes in the metropolis hasting step is set to 20 which should result in 20 likelihood each iteration unless the point gets rejected.

The single ellipsoid sampler works as the one described in Section 3.2 with an enlargement factor set to 1.2. The update interval was set to 20, meaning that the ellipsoid that new points are drawn from will update every 20 iterations. This is the case for both the Rust and Nestle implementation.

The multi ellipsoid sampler works as the one described in Section 3.3. However, the distance of the ellipsoids are not considered in the Rust implementation when subdividing the ellipsoids. This version will accept subdivisions if they result in a 50% volume reduction, but if the volume were to expand by 100% instead, this might indicate that there are more than 2 clusters, so it tries to subdivide further. These ellipsoids are also expanded by a factor of 1.2 and to avoid biased sampling in regions where multiple ellipsoids overlap, the number of ellipsoids that a generated point is contained in is counted and then the point is only accepted with a probability of $P(1/N)$, where N is the number of ellipsoids the suggested point is contained in. The update interval for this implementation was also chosen as 20 like the classic and single ellipsoid variations.

5.1.1 Stopping criterion

Since the sampling cannot go on forever, the main loop will stop when the estimated contribution of the remaining prior volume to the total evidence Z falls bellow a specified decline factor. This is calculated as

$$\log(Z + Z_{est}) - \log(Z) < d \log Z$$

where Z is the current evidence, Z_{est} is an estimate of the remaining evidence which we get by finding the highest likelihood value from the live points and subtracting the current iteration divided by the number of live points from it, and $d \log Z$ is the decline factor which have been set to 0.5 in the Rust and Nestle implementations.

5.1.2 Noteworthy libraries

To handle the vector and matrix operations and to hold the generated points, the library `nalgebra` was used in the Rust implementations, this is also used in the likelihood and prior transformation functions. In Nestle and pymultinest, the library used for these purposes is NumPy.

The single and multi ellipsoid samplers support parallelization in all implementations. In Rust, this is done by generating and evaluating the likelihood of new points in parallel and placing them in a queue. If this queue is empty when drawing a new point, it will be filled, otherwise a point will be picked from the queue.

5.2 Prior transform

All points are generated from a unit cube where the values in each dimension goes from 0 to 1. When sampling from a prior than is not in this space, a prior transformation has to be applied before evaluating the points likelihood. Consider a one dimensional problem where we are searching for the value 5. We know that the value might be between the range 0 to 10, but points are sampled between 0 to 1. In the prior transformation function, the following transformation would be applied

$$\theta_P = 20\theta$$

This assures that the newly generated point is in a valid range before evaluating its likelihood. Our previous knowledge about the problem informs this transformation. If we instead would know that the value we are looking for is between 4 to 7, finding the real value becomes faster.

5.3 System specifications

The system specifications of the machine the tests were performed on are:

- Intel(R) Core(TM) i7-6700k, 4 Cores, 8 logical processors CPU
- Kingston 16GB, 2400Mhz DDR4 RAM
- ASUS Z170-A motherboard

The machine is running Debian 11. The Cargo version is 1.69.0 and the Python version is 3.9.2.

5.4 Benchmarks

A few test cases were constructed to measure the time each implementation took to reach a stopping criteria, as well as the sample volume and number of likelihood calls in the Rust implementations of the single and multi ellipsoid samplers. All tests ran 11 times for each benchmark and the median time was picked for for the comparisons of wall clock time. The number of likelihood calls and prior volumes were also picked from these runs. The implementations described in Section 4 are tested in Rust, the Python library Nestle and the Python binding for multinest, pymultinest. These benchmark tests have been selected to highlight some of the theoretical benefits and drawbacks of the different algorithms, and due to the inherent performance benefit of Rust over Python, all problems should in theory perform better on the Rust variations of the algorithms. The creator of Nestle, Kyle Barbary, mentions in the FAQ section of the Nestle documentation that the multi ellipsoid version of Nestle is based on an early version of multinest and that it does not include later improvements to the algorithm [2]. Because of this and that pymultinest is a Python binding for multinest, which is mainly implemented in FORTRAN and C, it is hard to tell how these implementations will compare before running the tests. All tests ran with 400 live points to stay consistent.

5.4.1 Eggbox

In this benchmark, the likelihood function creates a surface with multiple modes of equal height creating an appearance which prompts its name. The surface can be seen in Figure 4. The likelihood function is defined as:

$$L(x, y) = (2 + \cos(tx/2) * \cos(ty/2))^5$$

$$tx = 10x\pi * (-5\pi)$$

$$ty = 10y\pi * (-5\pi)$$

The likelihood surface we are interested in is already in the correct space (0 to 1) so no prior transformation has to be applied.

This runs on all implementations in both Rust and Nestle as well as pymultinest. Considering the multimodal nature if this problem, it will most likely perform best on the multi ellipsoid implementations, and because of where these modes are located on the surface, the single ellipsoid version should almost cover the entire likelihood surface would lead to it suffering in terms of efficiency.

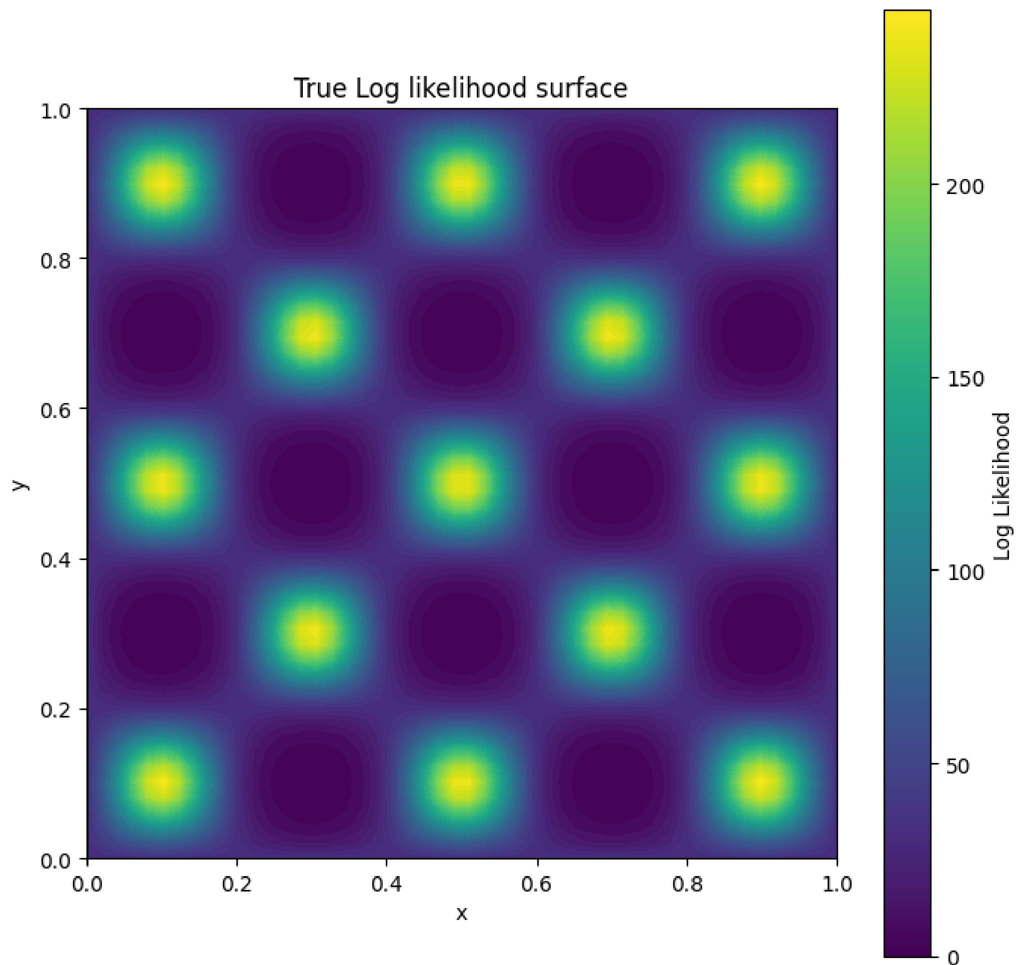


Figure 4: The true surface of the eggbox likelihood function

5.4.2 Gaussian shells

The likelihood function in this test creates peaks along the circumference of two circles on a two-dimensional surface. The likelihood function is defined as:

$$L(\theta, c_1, r_1, w_1, c_2, r_2, w_2) = \mathrm{circ}(c_1, r_1, w_1) + \mathrm{circ}(c_2, r_2, w_2)$$

$$\mathrm{circ}(\theta, c, r, w) = \frac{1}{\sqrt{2\pi w^2}} \exp\left[-\frac{|\theta - c| - r^2}{2w^2}\right]$$

where $c_1 = (-3.5, 0.0), r_1 = 2.0, w_1 = 0.1, c_2 = (3.5, 0.0), r_2 = 2.0$ and $w_2 = 0.1$. The prior transformation being applied here assures all values are in the range -6 to 6 with

$$\theta_P = 12\theta - 6$$

This will most likely perform similarly to the eggbox problem as its surface should favor the multi ellipsoid implementations.

5.4.3 Lotka-Volterra

The Lotka-Volterra equations are a pair of first order differential equations that describes the dynamics of predators and prey. The equations describe the change in their respective populations through time and are defined as:

$$\frac{dx}{dt} = \alpha x - \beta xy,$$

$$\frac{dy}{dt} = \delta xy - \gamma y$$

Given population data through a time period, we can estimate the parameters α, β, δ and γ as well as the initial values x_0 and y_0. To do this, we sample from parameter space, solve the differential equations and compare the solution with observed data in the likelihood function. We want to measure how far our model is to the data points available, or more precisely, how often the parameters would arise under the given parameters. This can be done with the following likelihood function:

$$X^2 = \sum \left(\frac{m_i - y_i}{\sigma}\right)^2$$

$$\log L = -X^2/2$$

where m_i is the model data we get from solving the differential equation with our sampled parameters, y_i is the available data (the real value), and σ is the known error in the data. The data used in this benchmark is available from D. R. Hundley at Whitman College [7]. This data features the population of snowshoe-hares and Canadian lynxes between the years 1900 and 1920. In the prior transformation function, the first four dimensions, representing α, β, δ and γ, are transformed to be in the range 0.01 to 2 with

$$\theta_P = \theta * (2.0 - 0.01)) + 0.01$$

and the last two dimensions, representing the initial values x_0 and y_0, are transformed to be in the range 1 to 50 with

$$\theta_P = \theta * (50.0 - 1.0)) + 1.0$$

Since an ODE has to be solved every time the likelihood function is called, efficiency will be valued highly and the multi ellipsoid variations will most likely perform the best. The ODEs are solved with RK4, a member of the RungeKutta family of differential equation solvers, and a step size of 0.01. This was implemented in both Rust and Python for this problem.

5.4.4 Rosenbrock

The Rosenbrock function was introduced by Howard H. Rosenbrock in 1960 [11]. It is a non-convex function commonly used as a performance test for optimization algorithms. The likelihood function is defined as:

$$L(x, y) = (a - x)^2 + b(y - x^2)^2$$

where the parameters usually are set as $a = 1$ and $b = 100$. The prior transformation being applied before likelihood evaluation here makes sure all values are in the range -4 to 4 with

$$\theta_P = 8\theta - 4$$

This likelihood function is easy to calculate and considering the surface appearance, that can be seen in Figure 5, this problem might be faster to solve with the single ellipsoid samplers. The extra efficiency that the multi ellipsoid samplers grants us might not be enough in this example due to the extra cost of running the K-means clustering and ellipsoid bounding algorithms multiple times.

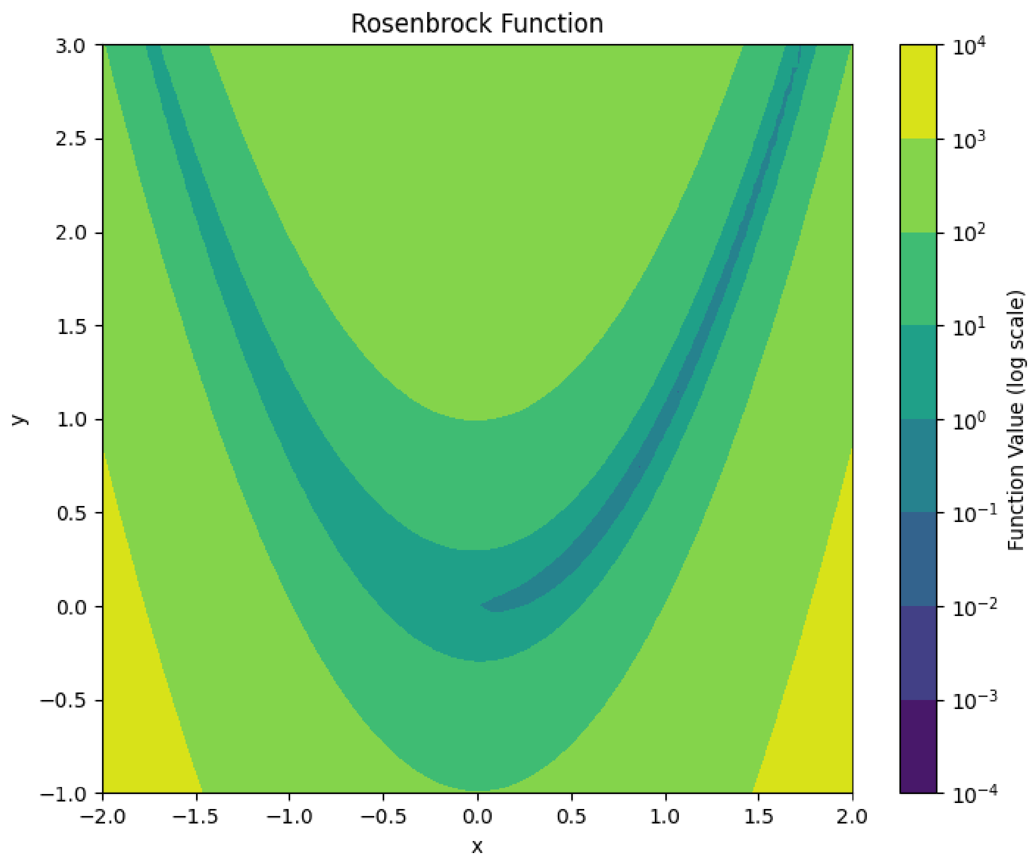
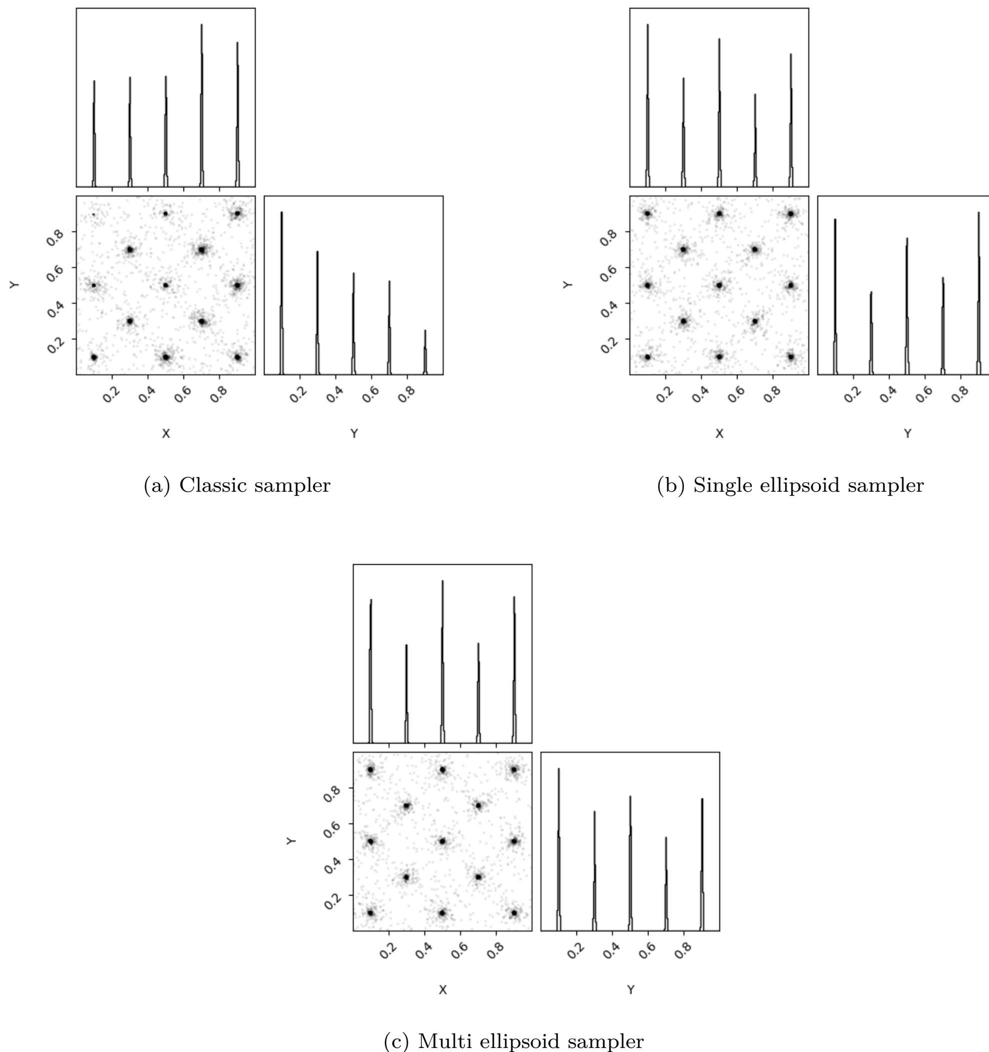


Figure 5: The true surface of the Rosenbrock function

6 Results and discussion

In this section, the results can be seen from the benchmark examples described in Section 5.4. The graphs were generated with Python and matplotlib, as well as corner.py. Corner plots have been provided to show that the algorithms produce results comparable to each other. In these corner plots, the frequency of values of samples in each dimension is shown as well as their relationship with other dimensions. For problems like eggbox, this relation can be directly compared to the true likelihood surface. The transparency of each point in the corner plots represents the weight of that sample. In this case, the higher weights appear later in the sampling process, which means that these are the later samples.

6.1 Eggbox



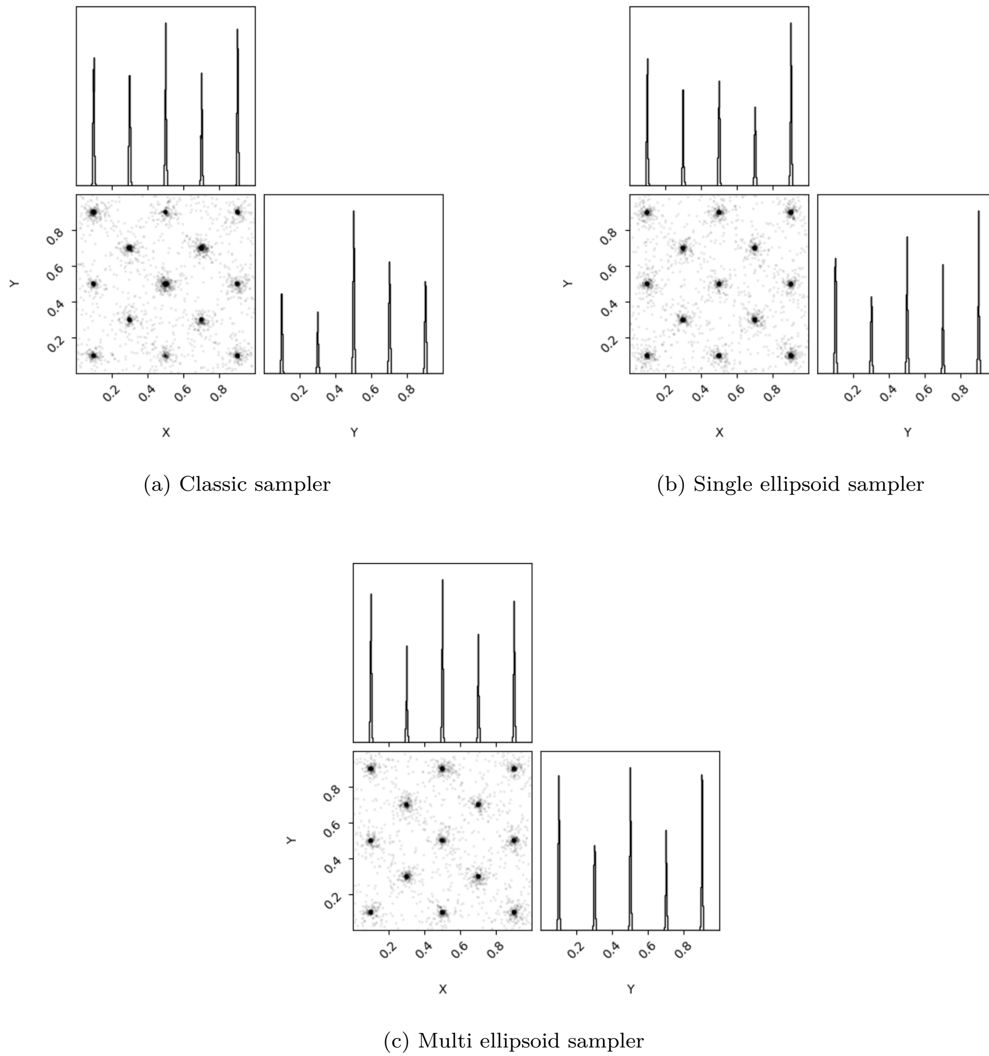
(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 6: Corner plot showing the eggbox samples generated by the Rust versions of the algorithms

The corner plots in Figure 6 show the samples explored by the Rust variations in the eggbox example. These variables are just x and y coordinates in the likelihood surface and the results looks like expected.



(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 7: Corner plot showing the eggbox samples generated by the Nestle versions of the algorithms

The corner plots in Figure 7 show the samples explored by Nestle in the eggbox example. Like the Rust samples, these are to be expected.

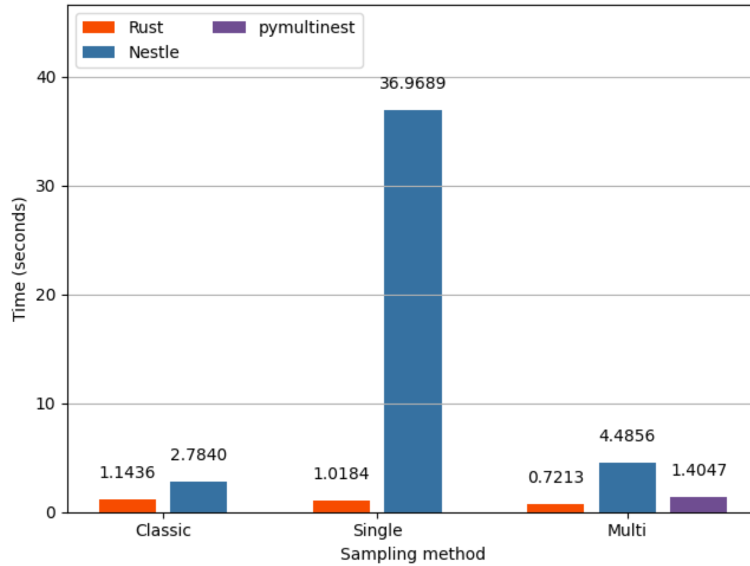


Figure 8: Time results for the eggbox benchmark in seconds

The bar plot in Figure 8 shows the wall clock times of each implementation when ran on the eggbox example. Lower results are better and as can be seen, the multi ellipsoid sampler performs well in both the Nestle and Rust implementations, it is however beat by the classic version among the Nestle implementations. The Rust implementations perform significantly better than their Neslte counterparts which is to be expected as Rust is a compiled language.

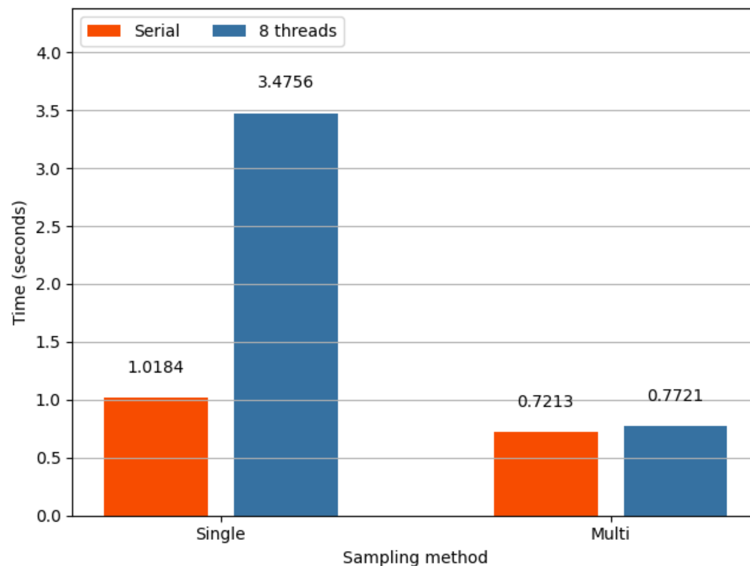


Figure 9: Time results comparing the serial and parallel Rust times for the eggbox benchmark in seconds

The bar plot in Figure 9 shows the parallel and serial wall clock times of the single and multi ellipsoid samplers in Rust when ran on the eggbox example. Lower results are better, and in this case it is apparent that there is no benefit of running this example in parallel with eight threads. It is probably due to the overhead of syncing and dividing work among threads that it takes longer to run this benchmark with eight threads rather than one.

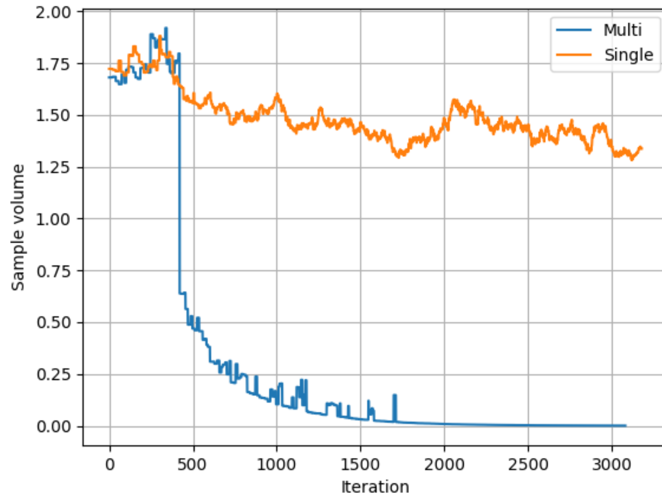


Figure 10: The sampling volumes for eggbox in the Rust implementations of the ellipsoid samplers

In Figure 10 the total volume of the ellipsoids, in the Rust implementation of the single and multi ellipsoid samplers, at each iteration can be seen. The volumes get progressively smaller as the ellipsoids focus in on the prior when it gets updated with the likelihood constraint. Lower volumes are better as it means that the algorithm is able to accept samples from the prior at a higher efficiency, as long as the prior is well defined. If, for example, there are too few live points making up the prior, it could be the case that sections of the likelihood surface are missing, which would make the volume smaller. Imagine that only 5 live points were used, since there are 13 peaks in this example, the ellipsoid approaches might not be able to cover all of them at once which results in a smaller volume.

The volumes here behave as expected. To cover all peaks, the single ellipsoid sampler needs to cover a much larger area than the multi ellipsoid sampler, that can create multiple ellipsoids covering the peaks without the surrounding area.

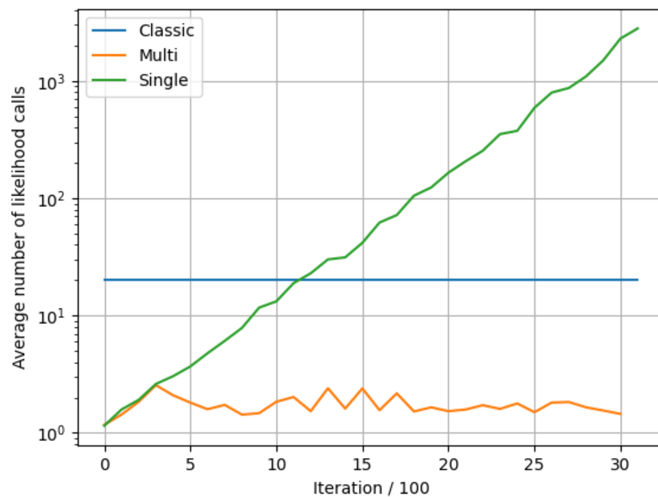
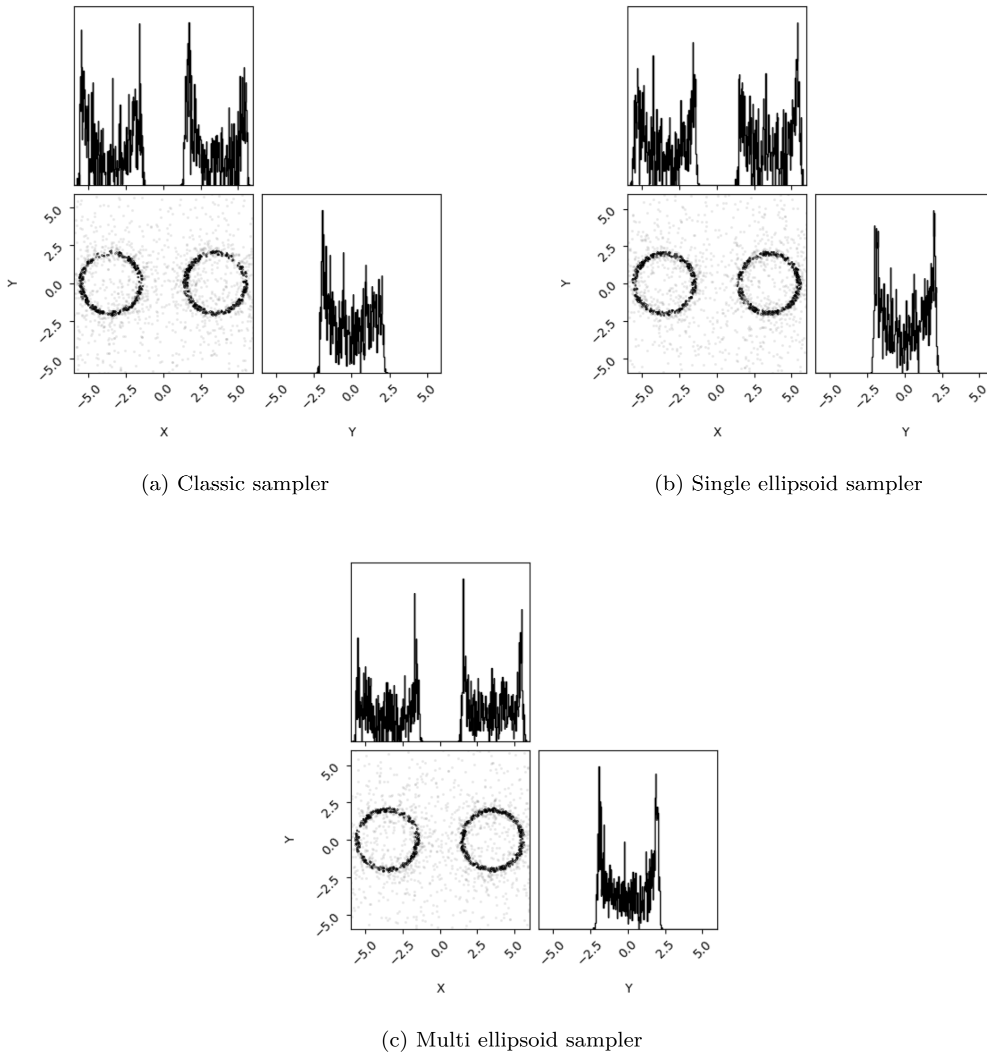


Figure 11: The number of likelihood calls at each iteration for eggbox

Figure 11 shows the number of likelihood calls, averaged in buckets of 100 iterations, for each sampler. Lower values here on the y-axis means that there were, on average, fewer likelihood calls made that iteration in the nested sampling loop. Here the y-axis is log scaled in order to accurately show the number of calls each sampler does. Ideally, only one sample would have to be generated per iteration, resulting in only one likelihood call. However, since there is a likelihood constraint, multiple samples might have to be generated before one is accepted.

In this example, it makes sense that the single ellipsoid sampler has to make more likelihood calls as the likelihood constraint gets more strict. This is because the single ellipsoid sampler covers a much greater volume, as can be seen in Figure 10, than the multi ellipsoid sampler. As a result of how the Metropolis Hastings algorithm work, it will most times only require 20 likelihood calls.

6.2 Gaussian shells



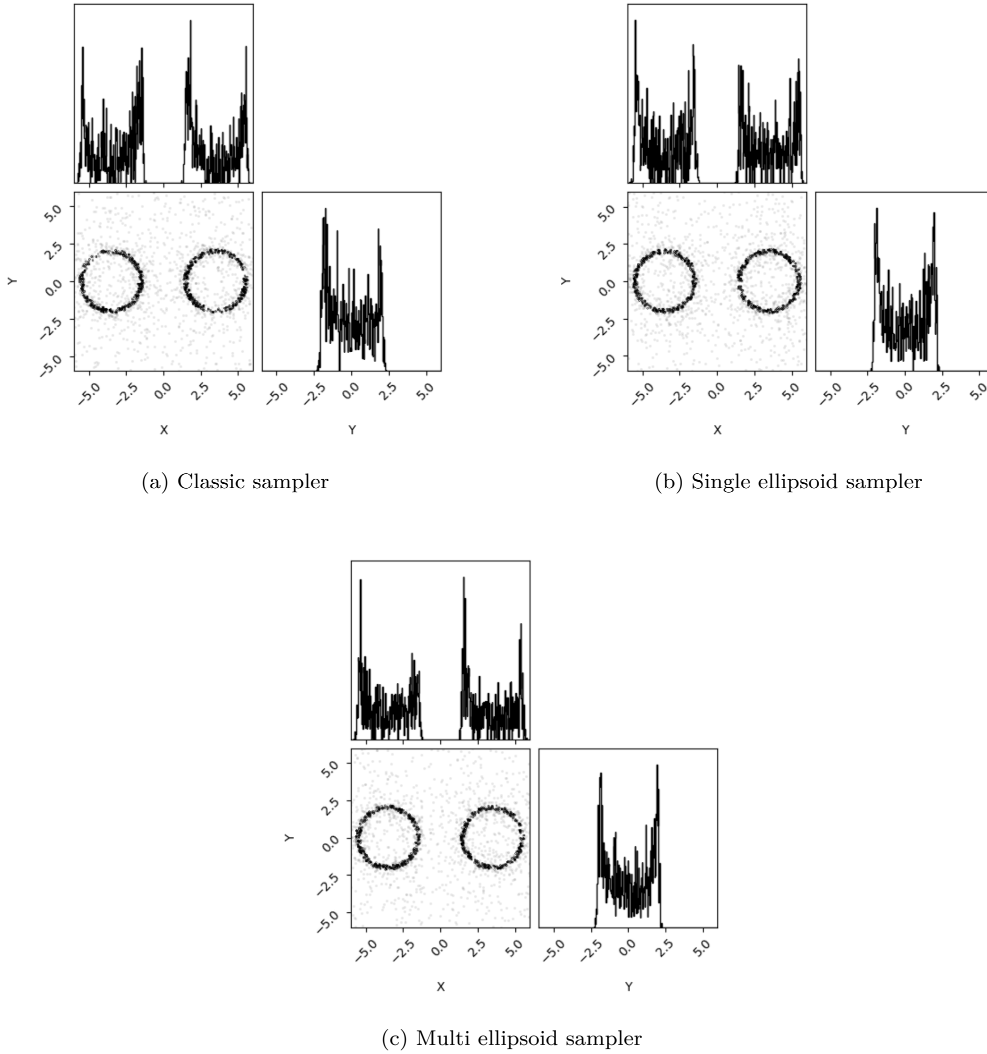
(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 12: Corner plot showing the Gaussian shells samples generated by the Rust versions of the algorithms

The corner plots in Figure 12 show the samples generated by the Rust samplers on the Gaussian shells example. The dimensions here represent x and y coordinates in the likelihood surface and look as expected.



(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 13: Corner plot showing the Gaussian shells samples generated by the Nestle versions of the algorithms

The corner plots in Figure 12 show the samples generated by the Nestle samplers on the Gaussian shells example. The dimensions here represent x and y coordinates in the likelihood surface and look like expected.

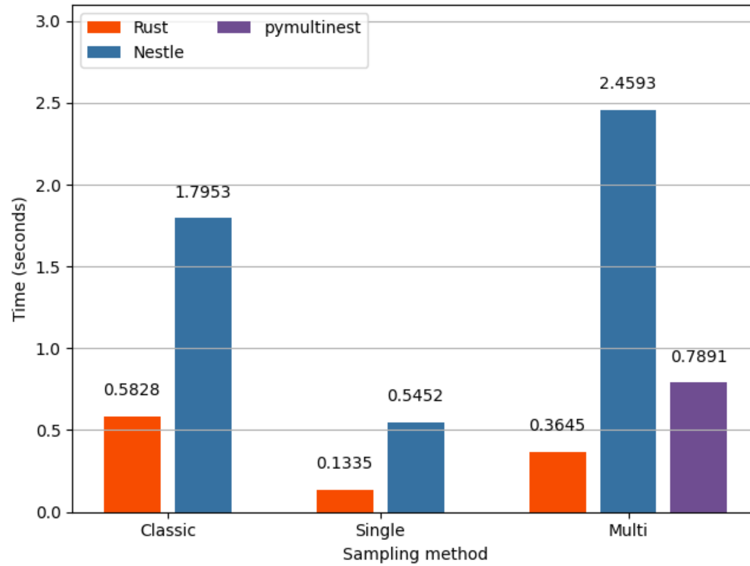


Figure 14: Time results for the Gaussian shells benchmark in seconds

The bar plot in Figure 14 shows the wall clock times of each implementation when ran on the Gaussian shells example. Lower results are better and here the single ellipsoid sampler performs the best. Here, I expected the multi ellipsoid sampler to outperform the single ellipsoid sampler, but according to the results, this is not the case. Further insight can be gathered by looking at the volumes of these samplers in Figure 16. The volumes do not differ that much until somewhere right after iteration 900 where the total volume for the multi ellipsoid sampler drops while the single ellipsoid sampler stays the same. It then peaks a few times and reaches the volume of the single ellipsoid sampler again, and then returns to normal. This is probably because there was too much noise in the prior up until this point for the multi ellipsoid sampler to subdivide its first ellipsoid, effectively making it a single ellipsoid sampler but with extra overhead before.

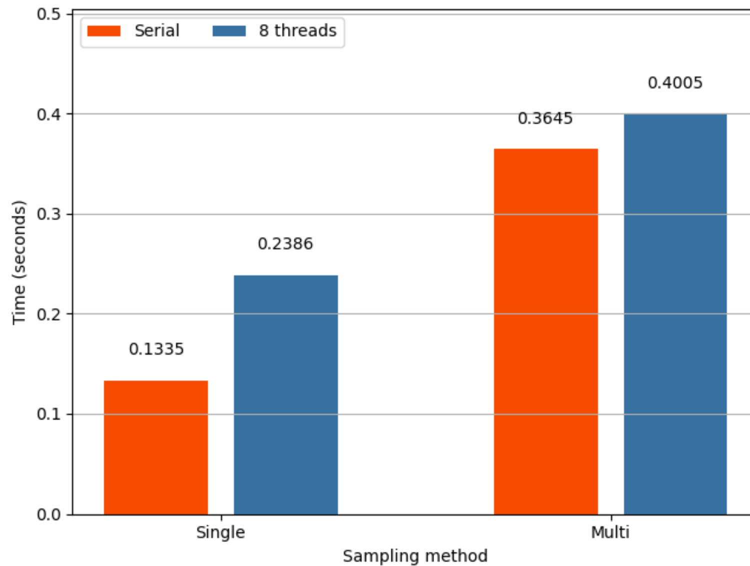


Figure 15: Time results comparing the serial and parallel Rust times for the Gaussian shells benchmark in seconds

The bar plot in Figure 15 shows the parallel and serial wall clock times of the single and

multi ellipsoid samplers in Rust when ran on the Gaussian shells example. Lower results are better, and in this case it is apparent that there is no benefit of running this example in parallel. It is probably due to the overhead of syncing and dividing work among threads that it takes longer to run this benchmark with eight threads rather than one.

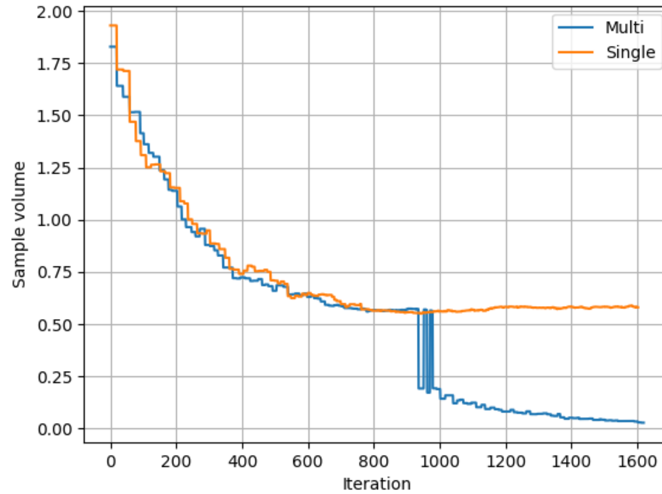


Figure 16: The sampling volumes for Gaussian shells in the Rust implementations of the ellipsoid samplers

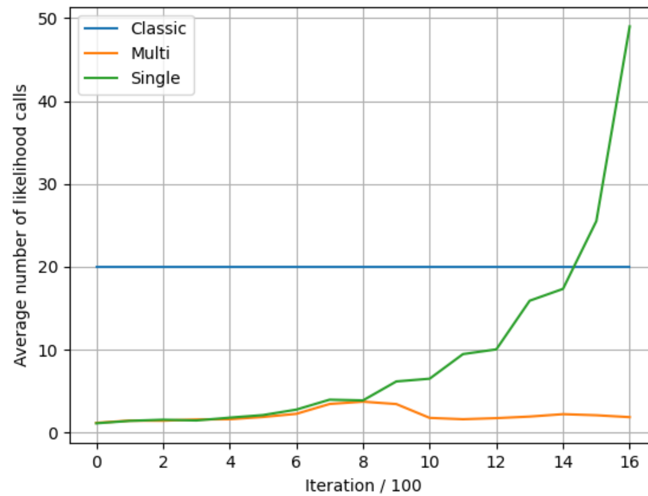


Figure 17: The number of likelihood calls at each iteration for Gaussian shells

Figure 17 shows the number of likelihood calls, averaged in buckets of 100 iterations, for each sampler. Lower values here on the y-axis means that there were, on average, fewer likelihood calls made that iteration in the nested sampling loop which is better. Here we see that the multi and single ellipsoid samplers require a similar amount of likelihood calls up until iteration 800. This makes sense considering their volumes as seen in Figure 16. If the nested sampler would have continued further, it is likely that this would have taken a toll on the single ellipsoid samplers performance relative to the other samplers, but at this

number of iterations, the clustering algorithm of the multi ellipsoid sampler resulted in too much overhead, making it the slowest for this example.

6.3 Lotka-Volterra

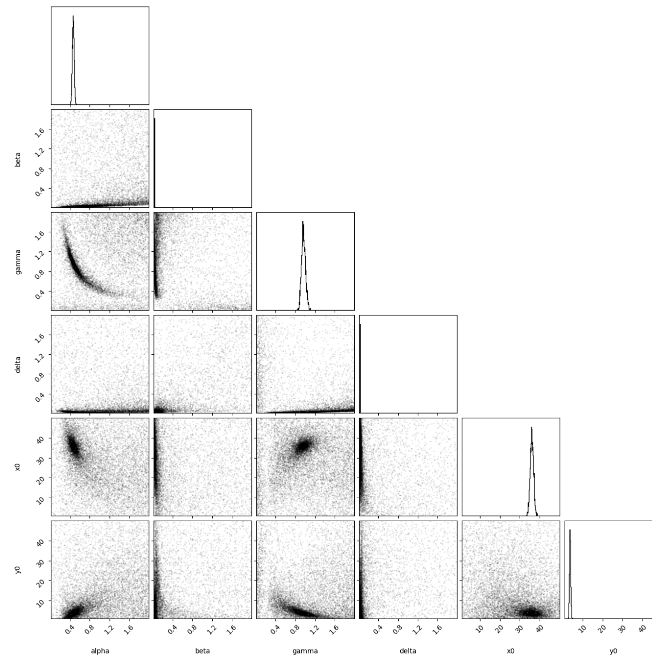


Figure 18: Corner plot showing the Lotka-Volterra samples generated by the Rust version of the single ellipsoid sampler

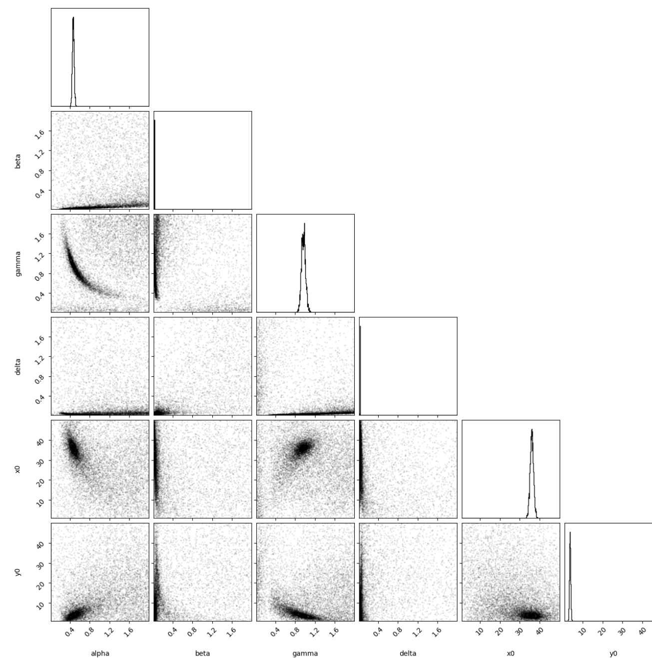


Figure 19: Corner plot showing the Lotka-Volterra samples generated by the Rust version of the multi ellipsoid sampler

Figure 18 and Figure 19 show the samples generated by the Rust samplers on the Lotka-Volterra example. Only the single and multi ellipsoid samplers were tested here due to the time it would take to run the classic sampler. The dimensions represent the different variables in the differential equation, they are alpha, beta, delta and gamma. These samples do not say much by them self, what is important to know is that all algorithms produce very similar results here.

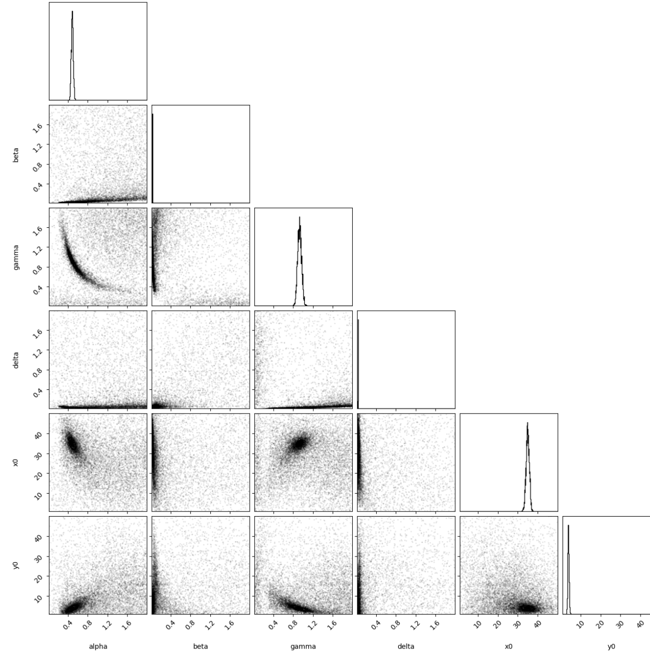


Figure 20: Corner plot showing the Lotka-Volterra samples generated by the Nestle version of the multi ellipsoid sampler

Figure 20 shows the samples generated by the Nestle multi ellipsoid sampler. Only the multi ellipsoid sampler was tested out of the Nestle variations due to the time it would take for the other variations.

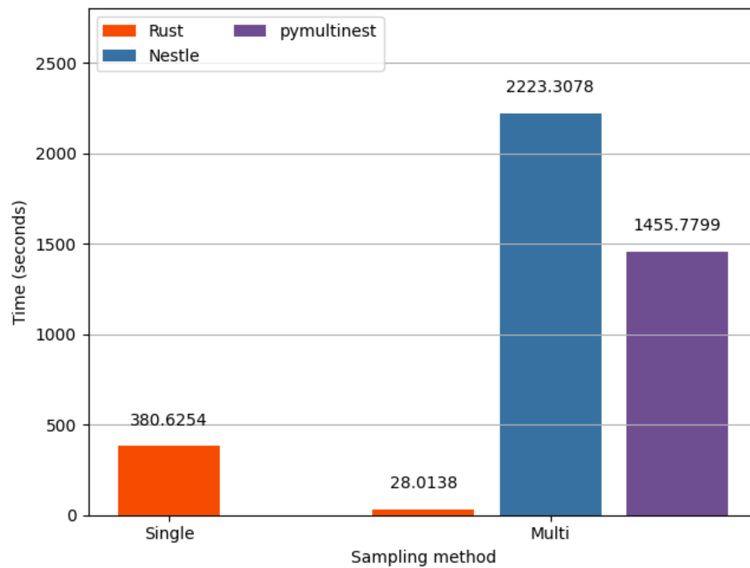


Figure 21: Time results for the Lotka-Volterra benchmark in seconds

Figure 21 shows the wall clock times of each multi ellipsoid sampler, as well as the single ellipsoid sampler implemented in Rust. In this example, efficiency is highly valued as each likelihood call results in the solving of an ODE. Figure 24 shows the number of likelihood calls by the single and multi ellipsoid samplers in Rust and explains why this example runs this slow on the single ellipsoid sampler. The reason for the higher number of likelihood calls on the single ellipsoid sampler can be explained by the results in Figure 23 which shows the sampling volume for each sampler throughout the iterations. The y-axis is log scaled to accurately show the volumes here since the single ellipsoid sampler reaches a much higher volume than the multi ellipsoid sampler.

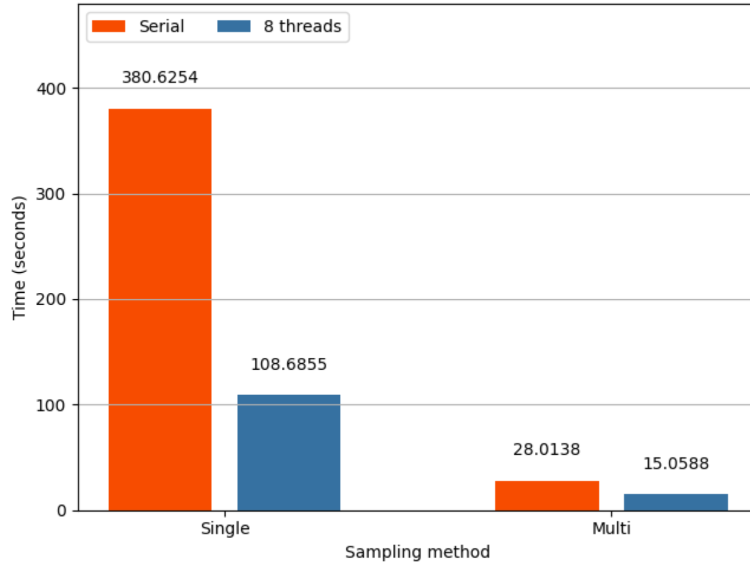


Figure 22: Time results comparing the serial and parallel Rust times for the Lotka-Volterra benchmark in seconds

The bar plot in Figure 22 shows the parallel and serial wall clock times of the single and multi ellipsoid samplers in Rust when ran on the Lotka-Volterra example. Lower results are better, and in this case, running the benchmark with eight threads results in a slight speedup of about 1.86 for the multi ellipsoid sampler and a more significant speedup of about 3.50 for the single ellipsoid sampler. The reason for the parallel version being faster in this example but not in others is probably due to the difficulty of calculating the likelihood here. The likelihood takes longer to compute here since an ODE has to be solved every time the likelihood is evaluated. This makes the runtime longer and the overhead of running the benchmark in parallel worth it compared to easier problems.

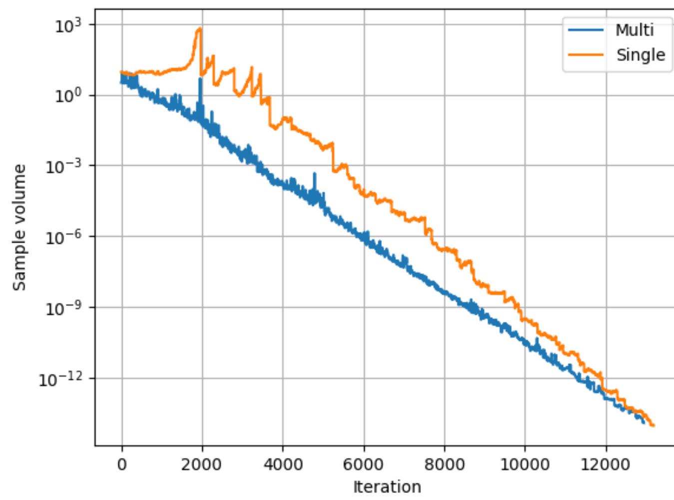


Figure 23: The sampling volumes for Lotka-Volterra in the Rust implementations of the ellipsoid samplers

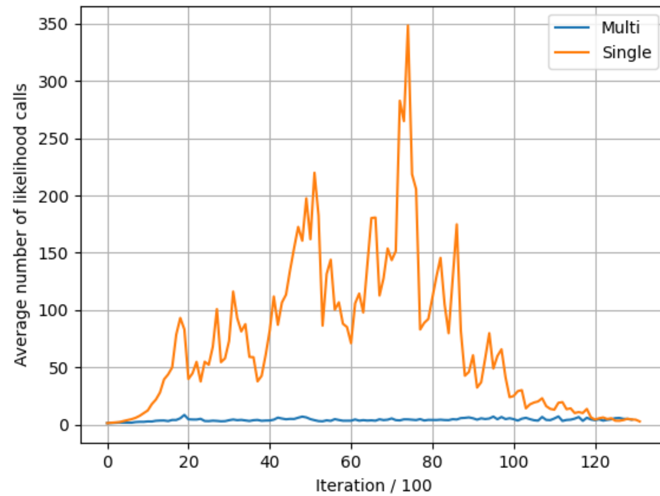
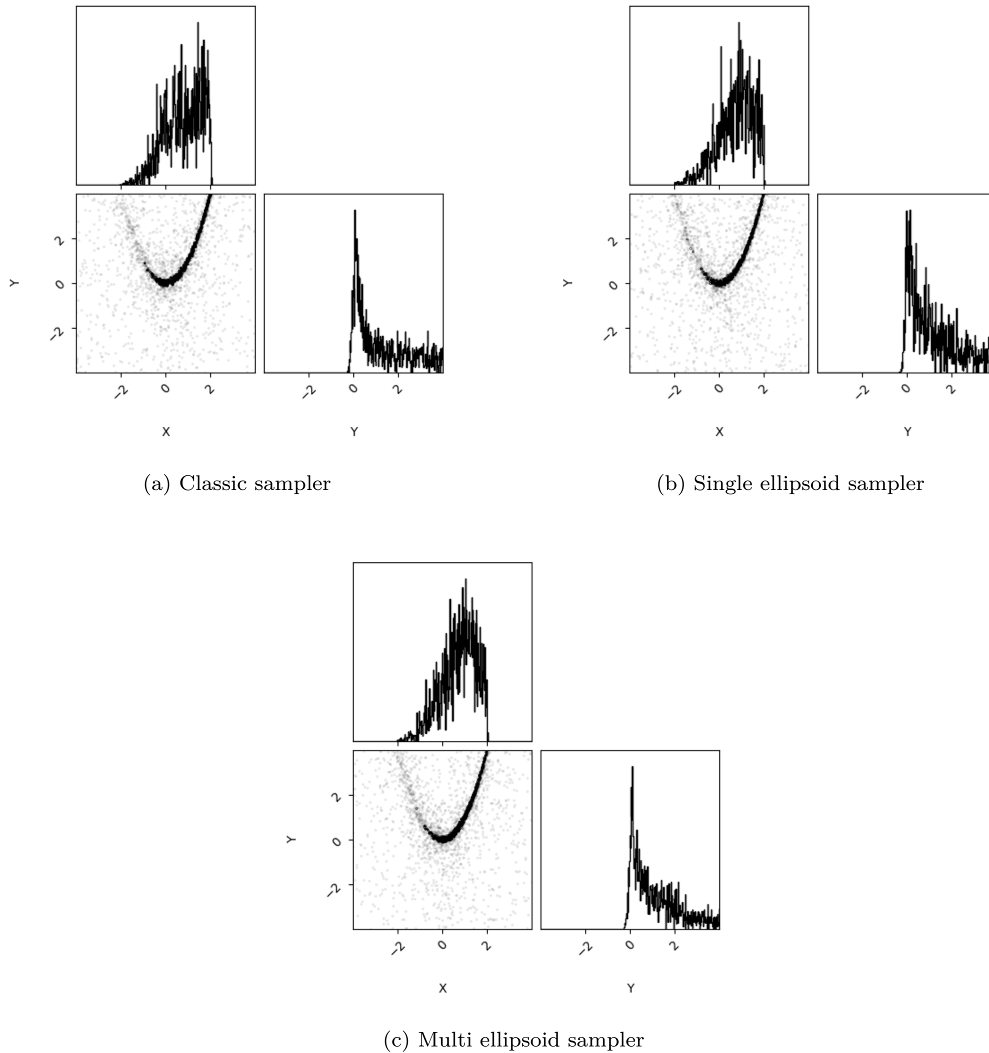


Figure 24: The number of likelihood calls at each iteration for Lotka-Volterra

6.4 Rosenbrock



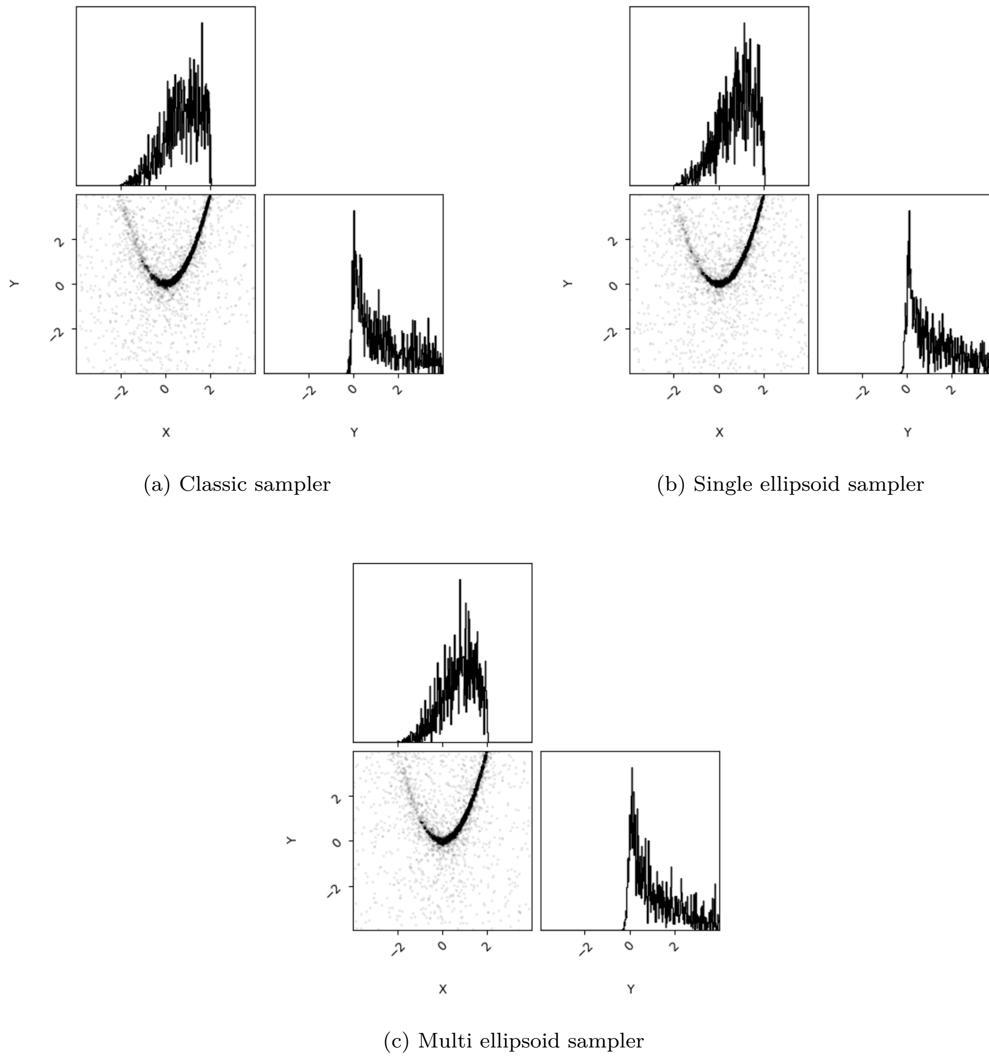
(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 25: Corner plot showing the Rosenbrock samples generated by the Rust versions of the algorithms

The corner plots in Figure 25 show the samples generated by the Rust implementations on the Rosenbrock example. The dimensions here represent the x and y coordinates in the likelihood surface and look as expected.



(a) Classic sampler

(b) Single ellipsoid sampler

(c) Multi ellipsoid sampler

Figure 26: Corner plot showing the Rosenbrock samples generated by the Nestle versions of the algorithms

The corner plots in Figure 26 show the samples generated by the Nestle implementations on the Rosenbrock example. The dimensions here represent the x and y coordinates in the likelihood surface and look as expected.

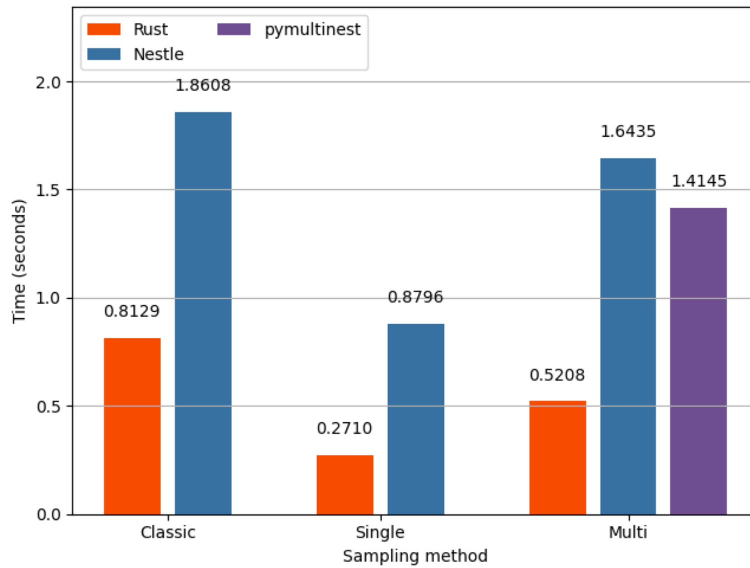


Figure 27: Time results for the Rosenbrock benchmark in seconds

Figure 27 shows the wall clock times of each implementation when ran on the Rosenbrock example. Lower times are better and here we see that the single ellipsoid sampler performed best which was expected. When looking at the Volumes of the ellipsoid samplers in Figure 29, we see that the multi ellipsoid sampler starts to shrinks its volume faster than the single ellipsoid somewhere around iteration 700 while the single ellipsoid volume actually grows during some iterations. This might be because of noise in the prior, but it is expected for the multi ellipsoid volumes to shrink faster. The multi ellipsoid volume peaks a few times between iteration 1000 and 1500 and becomes similar to the single ellipsoid volume again. This might be because the prior becomes too noisy for the ellipsoids to subdivide. The single ellipsoid sampler catches up in later iterations which is probably because the later samples favor X values greater than 0, meaning that there would be less dead space when drawing an ellipsoid around the live points.

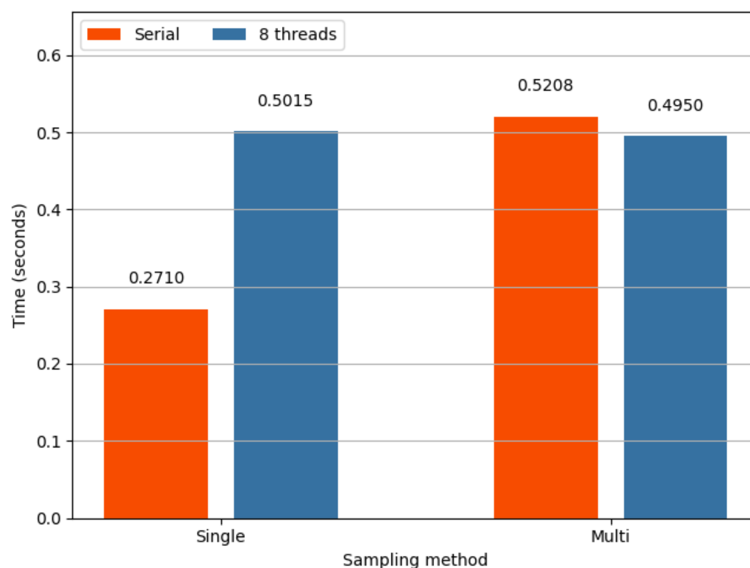


Figure 28: Time results comparing the serial and parallel Rust times for the Rosenbrock benchmark in seconds

The bar plot in Figure 28 shows the parallel and serial wall clock times of the single and multi ellipsoid samplers in Rust when ran on the Rosenbrock example. Lower results are better, and in this case it is apparent that there is little to no benefit of running this example in parallel. The parallel version of the multi ellipsoid sampler is slightly faster than running it with one thread. However, even in this case, the serial version of the single ellipsoid sampler is faster. It is probably due to the overhead of syncing and dividing work among threads that the performance of running this benchmark with eight threads is this bad.

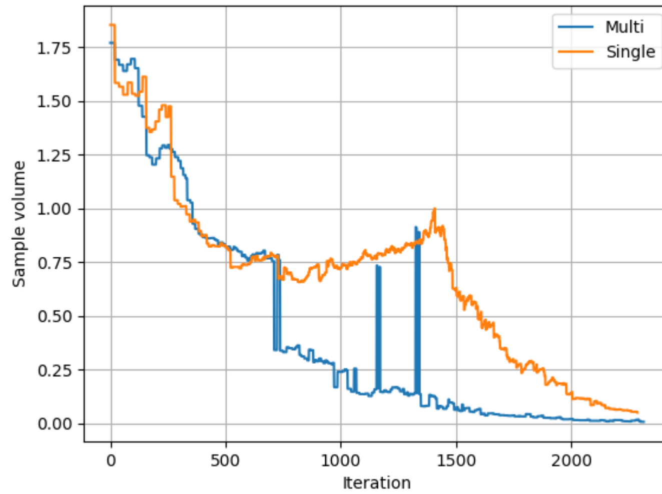


Figure 29: The sampling volumes for Rosenbrock in the Rust implementations of the ellipsoid samplers

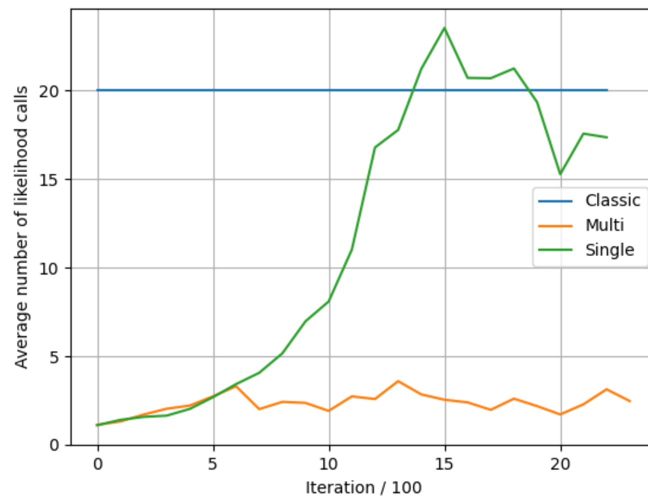


Figure 30: The number of likelihood calls at each iteration

Figure 31: The number of likelihood calls at each iteration for Rosenbrock

Figure 31 shows the number of likelihood calls, averaged in buckets of 100 iterations, for each sampler. Lower values here on the y-axis means that there were, on average, fewer

likelihood calls made that iteration in the nested sampling loop which is better. What can be seen here reflects the volume results seen in Figure 29. The number of calls the single ellipsoid sampler does grows but peaks around 1400 iterations. Then it goes below 20, the number of calls the classic sampler does, and stays stagnant until the nested sampled has finished.

7 Conclusion

The results confirm that these algorithms can be efficiently implemented in Rust and that they outperform their Python counterparts. The Rust implementation of the multi ellipsoid sampler runs up to 79 times faster than its Nestle counterpart, and up to 51 times faster than pymultinest while the Rust implementation of the single ellipsoid sampler runs up to 36 times faster than its Nestle counterpart. While it still performed much slower than the Rust implementation, pymultinest performed better than the Nestle multi ellipsoid sampler in every test case. However, considering how bad it compares to the Rust implementation, even though it is implemented in FORTRAN and C, the python overhead is apparent.

Running the Rust versions of the single and multi ellipsoid implementations in parallel proved to be slower in most examples, with the exception of the Lotka-Volterra benchmark. This is due to the overhead of syncing and dividing the work among the threads when running the benchmarks with eight threads rather than one. However, in more computationally difficult problems, like Lotka-Volterra, running the benchmark with eight threads, compared to one, resulted in a 3.50 times faster runtime for the single ellipsoid version and a 1.86 times faster runtime for the multi ellipsoid version.

Out of the different implementations, the most versatile is the multi ellipsoid variation. This performs good in examples where multiple peaks in the likelihood surface are explored, and in problems with high dimensions. It rarely performs slow and in the examples that were benchmarked, even when it was the slowest compared to the other implementation, it did not even take a second to run on the Rust implementation. If the run time is of concern even in these cases where the likelihood function is easy to compute, the single ellipsoid sampler might be a better alternative. It is often less efficient, but does not suffer from the overhead of running the clustering algorithm like the multi ellipsoid sampler does. This means it is preferable to reject more samples in order to avoid this overhead.

References

[1] Kyle Barbary. *nestle*. original-date: 2013-07-18T19:28:48Z. Apr. 2023. URL: https://github.com/kbarbary/nestle (visited on 05/23/2023).

[2] Kyle Barbary. *Nestle nestle 0.2.0 documentation*. URL: http://kylebarbary.com/nestle/index.html (visited on 05/23/2023).

[3] *Bayesian Statistics: A Beginner's Guide | QuantStart*. URL: https://www.quantstart.com/articles/Bayesian-Statistics-A-Beginners-Guide/ (visited on 03/19/2023).

[4] J. Buchner et al. "X-ray spectral modelling of the AGN obscuring region in the CDFS: Bayesian model selection and catalogue". en. In: *Astronomy & Astrophysics* 564 (Apr. 2014). Publisher: EDP Sciences, A125. ISSN: 0004-6361, 1432-0746. DOI: 10.1051/0004-6361/201322971. URL: https://www.aanda.org/articles/aa/abs/2014/04/aa22971-13/aa22971-13.html (visited on 05/10/2023).

[5] F. Feroz, M. P. Hobson, and M. Bridges. "MultiNest: an efficient and robust Bayesian inference tool for cosmology and particle physics". In: *Monthly Notices of the Royal Astronomical Society* 398.4 (Oct. 2009). arXiv:0809.3437 [astro-ph], pp. 1601–1614. ISSN: 00358711, 13652966. DOI: 10.1111/j.1365-2966.2009.14548.x. URL: http://arxiv.org/abs/0809.3437 (visited on 03/11/2023).

[6] Nicholas Metropolis et al. "Equation of State Calculations by Fast Computing Machines". In: *The Journal of Chemical Physics* 21.6 (June 1953). Publisher: American Institute of Physics, pp. 1087–1092. ISSN: 0021-9606. DOI: 10.1063/1.1699114. URL: https://aip.scitation.org/doi/10.1063/1.1699114 (visited on 04/02/2023).

[7] Gerhard Dangelmayr Micheal J. Kirby. *Introduction to Mathematical Modeling, Whitman College*. URL: http://people.whitman.edu/~hundledr/courses/M250F03/M250.html (visited on 04/20/2023).

[8] Pia Mukherjee, David Parkinson, and Andrew R. Liddle. "A Nested Sampling Algorithm for Cosmological Model Selection". In: *The Astrophysical Journal* 638.2 (Feb. 2006). arXiv:astro-ph/0508461, pp. L51–L54. ISSN: 0004-637X, 1538-4357. DOI: 10.1086/501068. URL: http://arxiv.org/abs/astro-ph/0508461 (visited on 03/11/2023).

[9] *nalgebra - Rust*. URL: https://docs.rs/nalgebra/latest/nalgebra/ (visited on 05/23/2023).

[10] *rayon - Rust*. URL: https://docs.rs/rayon/latest/rayon/ (visited on 05/23/2023).

[11] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: *The Computer Journal* 3.3 (Jan. 1960), pp. 175–184. ISSN: 0010-4620. DOI: 10.1093/comjnl/3.3.175. URL: https://doi.org/10.1093/comjnl/3.3.175 (visited on 04/20/2023).

[12] Pulkit Sharma. *The Ultimate Guide to K-Means Clustering: Definition, Methods and Applications*. en. Aug. 2019. URL: https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/ (visited on 05/04/2023).

[13] J. R. Shaw, M. Bridges, and M. P. Hobson. "Efficient Bayesian inference for multimodal problems in cosmology". In: *Monthly Notices of the Royal Astronomical Society* 378.4 (July 2007). arXiv:astro-ph/0701867, pp. 1365–1370. ISSN: 0035-8711, 1365-2966. DOI: 10.1111/j.1365-2966.2007.11871.x. URL: http://arxiv.org/abs/astro-ph/0701867 (visited on 03/11/2023).

[14] D. S. Sivia and J. Skilling. *Data analysis: a Bayesian tutorial*. en. 2nd ed. Oxford science publications. Oxford ; New York: Oxford University Press, 2006. ISBN: 978-0-19-856831-5.

[15] John Skilling. "Nested Sampling". In: *AIP Conference Proceedings* 735.1 (Nov. 2004). Publisher: American Institute of Physics, pp. 395–405. ISSN: 0094-243X. DOI: 10.1063/1.1835238. URL: https://aip.scitation.org/doi/abs/10.1063/1.1835238 (visited on 03/19/2023).

[16] John Skilling. "Nested sampling for general Bayesian computation". In: *Bayesian Analysis* 1.4 (Dec. 2006). Publisher: International Society for Bayesian Analysis, pp. 833–859. ISSN: 1936-0975, 1931-6690. DOI: 10.1214/06-BA127. URL: https://projecteuclid.

org/journals/bayesian-analysis/volume-1/issue-4/Nested-sampling-for-general-Bayesian-computation/10.1214/06-BA127.full (visited on 03/11/2023).

[17] *Stack Overflow Developer Survey 2022*. en. URL: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022 (visited on 04/02/2023).