The School of Mathematics

THE UNIVERSITY
of EDINBURGH

Robust Optimisation Monte Carlo for Likelihood-Free Inference

by

Vasileios Gkolemis

Dissertation Presented for the Degree of
MSc in Operational Research with Data Science

August 2020

Supervised by
Dr. Michael Gutmann

n

Abstract

Acknowledgments

Own Work Declaration

Here comes your own work declaration

Contents

List of Tables

List of Figures

1 Introduction

This dissertation is mainly focused on the implementation of the Robust Optimisation Monte Carlo (ROMC) method as it was proposed by (Ikonomov and Gutmann 2019), at the Python package ELFI - "Engine For Likelihood-Free Inference" (Lintusaari et al. 2018). ROMC is a novel likelihood-free inference approach for simulator-based models.

1.1 Motivation

Explanation of simulation-based models

A simulator-based model is a parameterised stochastic data generating mechanism (Gutmann and Corander 2016). The key characteristic of these models is that although we can sample (simulate) data points, we cannot evaluate the likelihood of a specific set of observations $\mathbf{y_0}$. Formally, a simulator-based model is described as a parameterised family of probability density functions $\{p_{\mathbf{y}|\boldsymbol{\theta}}(\mathbf{y})\}_{\boldsymbol{\theta}}$, whose closed-form is either unknown or intractable to evaluate. Whereas evaluating $p_{\mathbf{y}|\boldsymbol{\theta}}(\mathbf{y})$ is intractable, sampling is feasible. Practically, a simulator can be understood as a black-box machine M_r[1] that given a set of parameters $\boldsymbol{\theta}$, produces samples $\mathbf{y}$ in a stochastic manner i.e. $M_r(\boldsymbol{\theta}) \rightarrow \mathbf{y}$.

Simulator-based models are particularly captivating due to the modelling freedom they provide; any physical process that can be conceptualised as a computer program of finite (deterministic or stochastic) steps can be modelled as a simulator-based model without any compromise. The modelling freedom includes any amount of hidden (unobserved) internal variables or logic-based decisions. As always, this degree of freedom comes at a cost; performing the inference is particularly demanding from both computational and mathematical perspective. Unfortunately, the algorithms deployed so far permit the inference only at low-dimensionality parametric spaces, i.e. $\boldsymbol{\theta} \in \mathbb{R}^D$ where D is small.

Example

For underlying the importance of simulator-based models, let us use the tuberculosis disease spread example as described in (Tanaka et al. 2006). An overview of the disease spread model is presented at figure 1. At each stage one of the following *unobserved* events may happen; (a) the transmission of a specific haplotype to a new host, (b) the mutation of an existent haplotype or (c) the exclusion of an infectious host (recovers/dies) from the population. The random process, which stops when m infectious hosts are reached[2], can be parameterised by the transmission rate α, the mutation rate τ and the exclusion rate δ, creating a $3D$-parametric space $\boldsymbol{\theta} = (\alpha, \tau, \delta)$. The outcome of the process is a variable-size tuple $\mathbf{y}_{\boldsymbol{\theta}}$, containing the population contaminated by each different haplotype, as described in figure 1. Let's say that the disease has been spread in a real population and when m hosts were contaminated simultaneously, the vector with the infectious populations has been measured to be $\mathbf{y_0}$. We would like to discover the parameters $\boldsymbol{\theta} = (\alpha, \tau, \delta)$ that generated the spreading process and led to the specific outcome $\mathbf{y_0}$. Computing $p(\mathbf{y} = \mathbf{y_0}|\boldsymbol{\theta})$ requires tracking all tree-paths that could generate the specific tuple; such exhaustive enumeration becomes intractable when m grows larger, as in real-case scenarios. In figure 1 we can observe that a transmission followed by a recovery/death creates a loop, reinstating the process to the previous step, which also complicates the exhaustive enumeration. Hence, representing the process with a simulator-based model[3] and performing likelihood-free inference is the recommended solution.

Goal of Simulation-Based Models

As in most Machine Learning (ML) concepts, the fundamental goal is the derivation of one(many) parameter configuration(s) $\boldsymbol{\theta}^*$ that *describe* the data best i.e. generate samples $M_r(\boldsymbol{\theta}^*)$ that are as close as possible to the observed data $\mathbf{y_0}$. In our case, following the approach of Bayesian ML, we

[1] The subscript r in M_r indicates the *random* simulator. In the next chapters we will introduce M_d witch stands for the *deterministic* simulator.

[2] We suppose that the unaffected population is infinite, so a new host can always be added until we reach m simultaneous hosts.

[3] which is simple and efficient

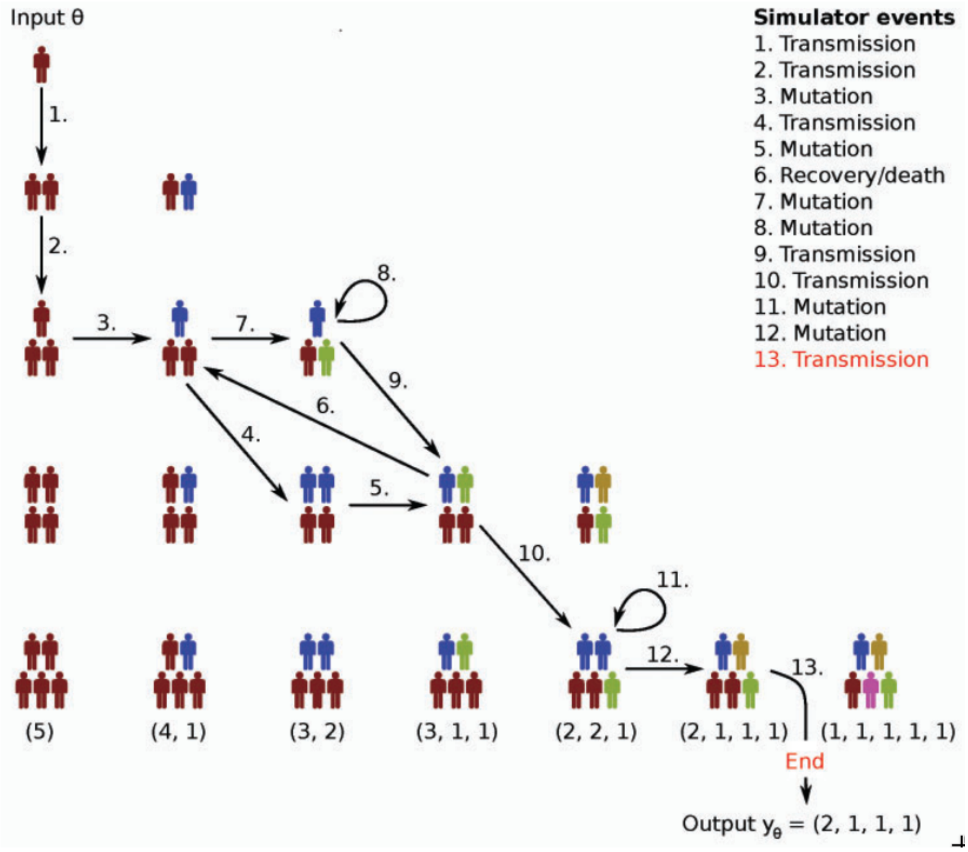


Figure 1: Depiction of a random example from the tuberculosis spreading process. The image has been taken from (Lintusaari et al. 2017).

treat the parameters of interest $\boldsymbol{\theta}$ as random variables and we try to *infer* a posterior distribution $p(\boldsymbol{\theta}|\mathbf{y_0})$ on them.

Robust Optimisation Monte Carlo (ROMC) method

The ROMC method (Ikonomov and Gutmann 2019) is very a recent likelihood-free approach. Its fundamental idea is the transformation of the stochastic data generation process $M_r(\boldsymbol{\theta})$ to a deterministic mapping $g_i(\boldsymbol{\theta})$, by sampling the variables that produce the randomness $\mathbf{v}_i \sim p(\mathbf{V})$. Formally, in every stochastic process the randomness is influenced by a vector of random variables $\mathbf{V}$, whose state is unknown prior to the execution of the simulation; sampling the state makes the procedure deterministic, namely $g_i(\boldsymbol{\theta}) = M_d(\boldsymbol{\theta}, \mathbf{V} = \mathbf{v}_i)$. This approach initially introduced at (Meeds and Welling 2015) with the title *Optimisation Monte Carlo (OMC)*. The ROMC extended this approach by resolving a fundamental failure-mode of OMC. The ROMC describes a methodology for approximating the posterior through a series of steps, without explicitly enforcing which algorithms must be utilised for each step[4]; in this sense, it can be perceived as a meta-algorithm.

Implementation

The most important contribution of this work is the implementation of the ROMC method in the Python package Engine for Likelihood-Free Inference (ELFI) (Lintusaari et al. 2018). Since the method has been published quite recently, it has not been implemented by now in any ML software. This work attempts to provide the research community with a robust and extensible implementation for further experimentation.

[4] The implementation chooses a specific algorithm for each task, but this choice has just a demonstrative value; any appropriate algorithm can be used instead.

1.2 Outline of Thesis

The remainder of the dissertation is organised as follows; in Chapter 2, we establish the mathematical formulation. Namely, we initially describe the simulator-based models and provide some background information on the fundamental algorithms proposed so far. Afterwards, we provide the mathematical description of the ROMC approach (Ikonomov and Gutmann 2019). Finally, we transform the mathematical description to algoriths. In Chapter 3, we illustrate the implementation part; we initially provide some information regarding the Python package Engine for Likelihood-Free Inference (ELFI) (Lintusaari et al. 2018) and subsequently, we present the implementation details of ROMC in this package. In general, the conceptual scheme followed by the dissertation is

$$\text{Mathematical modelling} \rightarrow \text{Algorithm} \rightarrow \text{Software}$$

In Chapter 4, we demonstrate the functionalities of the ROMC implementation at some real-world examples; this chapter demonstrates the accuracy of the ROMC method and our implementation's at likelihood-free tasks. Finally, in Chapter 5, we conclude with some thoughts on the work we have done and some future research ideas.

1.3 Notation

In this section, we provide an overview of the symbols utilised in the rest of the document. At this level, the quantities are introduced quite informally; most of them will be defined formally in the next chapters. We try to keep the notation as consistent as possible throughout the document. The symbol $\mathbb{R}^N$, when used, describes that a variable belongs to the N-dimensional Euclidean space; N does not represent a specific number. The bold formation ($\mathbf{x}$) indicates a vector, while (x) a scalar. Random variables are represented with capital letters ($\mathbf{\Theta}$) while the samples with lowercase letters ($\boldsymbol{\theta}$), i.e. $\boldsymbol{\theta} \sim \mathbf{\Theta}$.

Random Generator

- $M_r(\boldsymbol{\theta}) : \mathbb{R}^D \rightarrow \mathbb{R}$: The black-box data simulator

Parameters/Random Variables/Symbols

- $D \in \mathbb{N}$, the dimensionality of the parameter-space
- $\mathbf{\Theta} \in \mathbb{R}^D$, random variable representing the parameters of interest
- $\mathbf{y_0} \in \mathbb{R}^N$, the vector with the observations
- $\epsilon \in R$, the threshold setting the limit on the region around $\mathbf{y_0}$. When further notation is introduced regarding ϵ[5]
  - ϵ_{filter}, threshold for discarding solutions
  - ϵ_{region}, threshold for building bounding box regions
  - ϵ_{cutoff}, threshold for the indicator function
- $\mathbf{V} \in \mathbb{R}^N$, random variable representing the randomness of the generator. It is also called nuisance variable, because we are not interested in inferring a posterior distribution on it.
- $\mathbf{v}_i \sim \mathbf{V}$, a specific sample drawn from $\mathbf{V}$
- $\mathbf{Y}_{\boldsymbol{\theta}}$, random variable describing the simulator $M_r(\boldsymbol{\theta})$.
- $\mathbf{y}_i \sim \mathbf{Y}_\theta$, a sample drawn from $\mathbf{Y}_\theta$. It can be obtained by executing the simulator $\mathbf{y}_i \sim M_r(\boldsymbol{\theta})$

[5] If ϵ_*, $* : \{filter, region, cutoff\}$ values are not specified explicitly, they all share the common value of ϵ

Sets

- $B_{d,\epsilon}(\mathbf{y_0})$, the set of $\mathbf{y}$ points close to the observations, i.e. $\mathbf{y} := \{\mathbf{y} : d(\mathbf{y}, \mathbf{y_0}) \leq \epsilon\}$
- $B^i_{d,\epsilon}$, the set of points defined around $\mathbf{y}_i$ i.e. $B^i_{d,\epsilon} = B_{d,\epsilon}(\mathbf{y}_i)$
- $\mathcal{S}_i$, the set of parameters $\boldsymbol{\theta}$ that generate data close to the observations using the i-th deterministic generator, i.e. $\{\boldsymbol{\theta} : M_d(\boldsymbol{\theta}, \mathbf{v}_i) \in B_{d,\epsilon}(\mathbf{y_0})\}$

Generic Functions

- $p(\cdot)$, any valid pdf
- $p(\cdot|\cdot)$, any valid conditional distribution
- $p(\boldsymbol{\theta})$, the prior distribution on the parameters
- $p(\mathbf{v})$, the prior distribution on the nuisance variables
- $p(\boldsymbol{\theta}|\mathbf{y_0})$, the posterior distribution
- $p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})$, the approximate posterior distribution
- $d(\mathbf{x}, \mathbf{y}) : \mathbb{R}^{2N} \rightarrow \mathbb{R}$: any valid distance, the L_2 norm: $||\mathbf{x} - \mathbf{y}||_2$

Functions (Mappings)

- $M_d(\boldsymbol{\theta}, \mathbf{v}) : \mathbb{R}^D \rightarrow \mathbb{R}$, the deterministic generator; all stochastic variables that are part of the data generation process are represented by the parameter $\mathbf{v}$
- $f_i(\boldsymbol{\theta}) = M_d(\boldsymbol{\theta}, \mathbf{v}_i)$, deterministic generator associated with sample $\mathbf{v}_i \sim p(\mathbf{v})$
- $g_i(\boldsymbol{\theta}) = d(f_i(\boldsymbol{\theta}), \mathbf{y_0})$, distance of the generated data $f_i(\boldsymbol{\theta})$ from the observations
- $T(\mathbf{x}) : \mathbb{R}^{D_1} \rightarrow \mathbb{R}^{D_2}$ where $D_1 > D_2$, the mapping that computes the summary statistic
- $\mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(\mathbf{y})$, the indicator function; returns 1 if $d(\mathbf{y}, \mathbf{y_0}) \leq \epsilon$, else 0
- $L(\boldsymbol{\theta})$, the likelihood
- $L_{d,\epsilon}(\boldsymbol{\theta})$, the approximate likelihood

1.4 Online notebooks

Since the dissertation is mainly focused on the implementation of the ROMC, we have created some jupyter notebooks for supporting the main document. The reader is advised to exploit the notebooks in order to (a) view the code used for performing the inference (b) understand the functionalities of our implementation in a practical sense (c) check the validity of our claims and (c) interactively execute experiments.

The notebooks are provided in two different formats:[6]

- The first format is located In this Github repository. In order for the reader to experiment with the examples, they should *clone* the repository locally, execute the installation instructions and run the notebooks using the Jupyter package.
- The second format is the google colab notebook. Following the links below, the reader can view the notebook and make an online clone of it. Hence, without any installation overhead, they may interactivelly explore all the provided functionalities.

[6] The notebooks in the two repositories are identical.

Throughout the document, we may refer to the notebooks whenever we want to let the reader observe a specific functionality. The notebooks provide a practical overview of the implemented functionalities and it are easy to use; in particular, the google colab version is entirely plug-and-play. Therefore, we encourage the reader of the dissertation to use them as supporting material. The following list contains the implemented examples along with their links:

- Simple 1D example
  - Google colab
  - Github repository
- Simple 2D example
  - Google colab
  - Github repository
- Moving Average example
  - Google colab
  - Github repository
- Extensibility example
  - Google colab
  - Github repository

2 Background

2.1 Simulator-based models

As already stated at Chapter 1, in simulator-based models we cannot evaluate the posterior $p(\boldsymbol{\theta}|\mathbf{y_0}) \propto L(\boldsymbol{\theta})p(\boldsymbol{\theta})$, due to the intractability of the likelihood $L(\boldsymbol{\theta}) = p(\mathbf{y_0}|\boldsymbol{\theta})$. The following equation allows incorporating the simulator in the place of the likelihood and forms the basis of all likelihood-free inference approaches,

$$L(\boldsymbol{\theta}) = \lim_{\epsilon \to 0} c_\epsilon \int_{\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0})} p(\mathbf{y}|\boldsymbol{\theta}) d\mathbf{y} = \lim_{\epsilon \to 0} c_\epsilon \Pr(M_r(\boldsymbol{\theta}) \in B_{d,\epsilon}(\mathbf{y_0})) \tag{2.1}$$

where c_ϵ is a proportionality factor dependent on ϵ, needed when $\Pr(M_r(\boldsymbol{\theta}) \in B_{d,\epsilon}(\mathbf{y_0})) \to 0$, as $\epsilon \to 0$. Equation 2.1 describes that the likelihood of a specific parameter configuration $\boldsymbol{\theta}$ is proportional to the probability that the simulator will produce outputs equal to the observations, using this configuration.

2.1.1 Approximate Bayesian Computation (ABC) Rejection Sampling

ABC rejection sampling is a modified version of the traditional rejection sampling method, for cases when the evaluation of the likelihood is intractable. In the typical rejection sampling, a sample obtained from the prior $\boldsymbol{\theta} \sim p(\boldsymbol{\theta})$ gets accepted with probability $L(\boldsymbol{\theta})/\text{max}_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$. Though we cannot use this approach out-of-the-box (evaluating $L(\boldsymbol{\theta})$ is impossible in our case), we can modify the method incorporating the simulator.

In the discrete case scenario where $\mathbf{Y}_{\boldsymbol{\theta}}$ can take a finite set of values, the likelihood becomes $L(\boldsymbol{\theta}) = \Pr(\mathbf{Y}_{\boldsymbol{\theta}} = \mathbf{y_0})$ and the posterior $p(\boldsymbol{\theta}|\mathbf{y_0}) \propto \Pr(\mathbf{Y}_{\boldsymbol{\theta}} = \mathbf{y_0})p(\boldsymbol{\theta})$; hence, we can sample from the prior $\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta})$, run the simulator $\mathbf{y}_i = M_r(\boldsymbol{\theta}_i)$ and accept $\boldsymbol{\theta}_i$ only if $\mathbf{y}_i = \mathbf{y_0}$.

The method above becomes less useful as the finite set of $\mathbf{Y}_{\boldsymbol{\theta}}$ values grows larger, since the probability of accepting a sample becomes smaller. In the limit where the set becomes infinite (i.e. continuous case) the probability becomes zero. In order for the method to work in this set-up, a relaxation is introduced; we relax the acceptance criterion by letting $\mathbf{y}_i$ lie in a larger set of points i.e. $\mathbf{y}_i \in B_{d,\epsilon}(\mathbf{y_0}), \epsilon > 0$. The region can be defined as $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(\mathbf{y}, \mathbf{y_0}) \leq \epsilon\}$ where $d(\cdot,\cdot)$ can represent any valid distance. With this modification, the maintained samples follow the approximate posterior,

$$p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0}) \propto Pr(\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0}))p(\boldsymbol{\theta}) \tag{2.2}$$

This method is called Rejection ABC.

2.1.2 Summary Statistics

When $\mathbf{y} \in \mathbb{R}^D$ lies in a high-dimensional space, generating samples inside $B_{d,\epsilon}(\mathbf{y_0})$ becomes rare even when ϵ is relatively large; this is the curse of dimensionality. As a representative example lets make the following hypothesis;

- d is set to be the Euclidean distance, hence $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : ||\mathbf{y} - \mathbf{y_0}||_2^2 < \epsilon^2\}$ is a hyper-sphere with radius ϵ and volume $V_{hypersphere} = \frac{\pi^{D/2}}{\Gamma(D/2+1)}\epsilon^D$
- the prior $p(\boldsymbol{\theta})$ is a uniform distribution in a hyper-cube with side of length 2ϵ and volume $V_{hypercube} = (2\epsilon)^D$
- the generative model is the identity function $\mathbf{y} = f(\boldsymbol{\theta}) = \boldsymbol{\theta}$

The probability of drawing a sample inside the hypersphere equals the fraction of the volume of the hypersphere inscribed in the hypercube:

$$Pr(\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0})) = Pr(\boldsymbol{\theta} \in B_{d,\epsilon}(\mathbf{y_0})) = \frac{V_{hypersphere}}{V_{hypercube}} = \frac{\pi^{D/2}}{2^D \Gamma(D/2+1)} \to 0, \quad \text{as} \quad D \to \infty \tag{2.3}$$

We observe that the probability tends to 0, independently of ϵ; enlarging ϵ will not increase the acceptance rate. Intuitively, we can think that in high-dimensional spaces the volume of the hypercube concentrates at its corners. This generates the need for a mapping $T : \mathbb{R}^{D_1} \rightarrow \mathbb{R}^{D_2}$ where $D_1 > D_2$, for squeezing the dimensionality of the output. This dimensionality-reduction step that redefines the area as $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(T(\mathbf{y}), T(\mathbf{y_0})) \leq \epsilon\}$ is called *summary statistic* extraction, since the distance is not measured on the actual outputs, but on a summarisation (i.e. lower-dimension representation) of them.

2.1.3 Approximations introduced so far

So far, we have introduced some approximations for inferring the posterior as $p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0}) \propto Pr(\mathbf{Y}_{\boldsymbol{\theta}} \in B_{d,\epsilon}(\mathbf{y_0}))p(\boldsymbol{\theta})$ where $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(T(\mathbf{y}), T(\mathbf{y_0})) < \epsilon\}$. These approximations introduce two different types of errors:

- ϵ is chosen to be *big enough*, so that enough samples are accepted. This modification leads to the approximate posterior introduced in (2.2)
- T introduces some loss of information, making possible a $\mathbf{y}$ far away from $\mathbf{y_0}$ i.e. $\mathbf{y} : d(\mathbf{y}, \mathbf{y_0}) > \epsilon$, to enter the acceptance region after the dimensionality reduction $d(T(\mathbf{y}), T(\mathbf{y_0})) \leq \epsilon$

In the following sections, we will not use the summary statistics in our expressions for the notation not to clutter. One could understand it as absorbing the mapping $T(\cdot)$ inside the simulator. In any case, all the propositions that will be expressed in the following sections are valid with the use of summary statistics.

2.1.4 Optimisation Monte Carlo (OMC)

Before we define the likelihood approximation as introduced in the OMC, approach lets define the indicator function based on $B_{d,\epsilon}(\mathbf{y})$. The indicator function $\mathbb{1}_{B_{d,\epsilon}(\mathbf{y})}(\mathbf{x})$ returns 1 if $\mathbf{x} \in B_{d,\epsilon}(\mathbf{y})$ and 0 otherwise. If $d(\cdot, \cdot)$ is a formal distance, due to symmetry $\mathbb{1}_{B_{d,\epsilon}(\mathbf{y})}(\mathbf{x}) = \mathbb{1}_{B_{d,\epsilon}(\mathbf{x})}(\mathbf{y})$, so the expressions can be used interchangeably.

$$\mathbb{1}_{B_{d,\epsilon}(\mathbf{y})}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in B_{d,\epsilon}(\mathbf{y}) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Based on equation (2.2) and the indicator function as defined above (2.4), we can approximate the likelihood as:

$$L_{d,\epsilon}(\boldsymbol{\theta}) = \int_{\mathbf{y} \in B_\epsilon(\mathbf{y_0})} p(\mathbf{y}|\boldsymbol{\theta})d\mathbf{y} = \int_{\mathbf{y} \in \mathbb{R}^D} \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(\mathbf{y})p(\mathbf{y}|\boldsymbol{\theta})d\mathbf{y} \tag{2.5}$$

$$\approx \frac{1}{N}\sum_i^N \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(\mathbf{y}_i), \text{ where } \mathbf{y}_i \sim M_r(\boldsymbol{\theta}) \tag{2.6}$$

$$\approx \frac{1}{N}\sum_i^N \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(\mathbf{y}_i) \text{ where } \mathbf{y}_i = M_d(\boldsymbol{\theta}, \mathbf{v}_i), \mathbf{v}_i \sim p(\mathbf{v}) \tag{2.7}$$

This approach is quite intuitive; approximating the likelihood of a specific $\boldsymbol{\theta}$ requires sampling from the data generator and count the fraction of samples that lie inside the area around the observations. Nevertheless, by using the approximation of equation (2.6) we need to draw N new samples for each distinct evaluation of $L_{d,\epsilon}(\boldsymbol{\theta})$; this makes this approach quite inconvenient from a computational point-of-view. For this reason, we choose to approximate the integral as in equation (2.7); the nuisance variables are sampled once $\mathbf{v}_i \sim p(\mathbf{v})$ and we count the fraction of samples that lie inside the area using the deterministic simulators $M_d(\boldsymbol{\theta}, \mathbf{v}_i) \forall i$. Hence, the evaluation $L_{d,\epsilon}(\boldsymbol{\theta})$ for each different $\boldsymbol{\theta}$ does not imply drawing new samples all over again. Based on this approach, the unnormalised approximate posterior can be defined as:

$$p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0}) \propto p(\boldsymbol{\theta}) \sum_i^N \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(\mathbf{y}_i) \tag{2.8}$$

Further approximations for sampling and computing expectations

The posterior approximation in (2.2) does not provide any obvious way for drawing samples. In fact, the set $\mathcal{S}_i = \{\boldsymbol{\theta} : M_d(\boldsymbol{\theta}, \mathbf{v}_i) \in B_{d,\epsilon}(\mathbf{y_0})\}$ can represent any arbitrary shape in the D-dimensional Euclidean space; it can be non-convex, can contain disjoint sets of $\boldsymbol{\theta}$ etc. We need some further simplification of the posterior for being able to draw samples from it.

As a side-note, weighted sampling could be performed in a straightforward fashion with importance sampling. Using the prior as the proposal distribution $\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta})$ and we can compute the weight as $w_i = \frac{L_{d,\epsilon}(\boldsymbol{\theta}_i)}{p(\boldsymbol{\theta}_i)}$, where $L_{d,\epsilon}(\boldsymbol{\theta}_i)$ is computed with the expression (**??**). This approach has the same drawbacks as ABC rejection sampling; when the prior is wide or the dimensionality D is high, drawing a sample with non-zero weight is rare, leading to either poor Effective Sample Size (ESS) or huge execution time.

The OMC proposes a quite drastic simplification of the posterior; it squeezes all regions $\mathcal{S}_i$ into a single point $\boldsymbol{\theta}_i^* \in \mathcal{S}_i$ attaching a weight w_i proportional to the volume of $\mathcal{S}_i$. For obtaining a $\boldsymbol{\theta}_i^* \in \mathcal{S}_i$, a gradient based optimiser is used for minimising $g_i(\boldsymbol{\theta}) = d(\mathbf{y_0}, f_i(\boldsymbol{\theta}))$ and the estimation of the volume of $\mathcal{S}_i$ is done using the Hessian approximation $\mathbf{H}_i \approx \mathbf{J}_i^{*T}\mathbf{J}_i^*$, where $\mathbf{J}_i^*$ is the Jacobian matrix of $g_i(\boldsymbol{\theta})$ at $\boldsymbol{\theta}_i^*$. Hence,

$$p(\boldsymbol{\theta}|\mathbf{y_0}) \propto p(\boldsymbol{\theta}) \sum_i^N w_i \delta(\boldsymbol{\theta} - \boldsymbol{\theta}_i^*) \tag{2.9}$$

$$\boldsymbol{\theta}_i^* = \text{argmin}_{\boldsymbol{\theta}}\, g_i(\boldsymbol{\theta}) \tag{2.10}$$

$$w_i \propto \frac{1}{\sqrt{det(\mathbf{J}_i^{*T}\mathbf{J}_i^*)}} \tag{2.11}$$

The distribution (2.9) provides weighted samples automatically and an expectation can be computed easily with the following equation,

$$E_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})] = \frac{\sum_i^N w_i p(\boldsymbol{\theta}_i^*) h(\boldsymbol{\theta}_i^*)}{\sum_i^N w_i p(\boldsymbol{\theta}_i^*)} \tag{2.12}$$

2.2 Robust Optimisation Monte Carlo (ROMC) approach

The simplifications introduced by OMC, although quite useful from a computational point-of-view, they suffer from some significant failure modes:

- The whole acceptable region $\mathcal{S}_i$, for each nuisance variable, shrinks to a single point $\boldsymbol{\theta}_i^*$; this simplification may add significant error when then the area $\mathcal{S}_i$ is relatively big.

- The weight w_i is computed based only at the curvature at the point $\boldsymbol{\theta}_i^*$. This approach is error prone at many cases e.g. when g_i is almost flat at $\boldsymbol{\theta}_i^*$, leading to a $\det(\mathbf{J}_i^{*T}\mathbf{J}_i^*) \rightarrow 0 \Rightarrow w_i \rightarrow \infty$, thus dominating the posterior.

- There is no way to solve the optimisation problem $\boldsymbol{\theta}_i^* = \text{argmin}_{\boldsymbol{\theta}}\, g_i(\boldsymbol{\theta})$ when g_i is not differentiable.

2.2.1 Sampling and computing expectation in ROMC

The ROMC approach resolves the aforementioned issues. Instead of collapsing the acceptance regions $\mathcal{S}_i$ into single points, it tries to approximate them with a bounding box.[7]. A uniform distribution is then defined on the bounding box area, used as the proposal distribution for importance sampling. If we define as q_i, the uniform distribution defined on the $i-th$ bounding box, weighted sampling is performed as:

$$\boldsymbol{\theta}_{ij} \sim q_i \tag{2.13}$$

$$w_{ij} = \frac{\mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(M_d(\boldsymbol{\theta}_{ij}, \mathbf{v}_i))p(\boldsymbol{\theta}_{ij})}{q(\boldsymbol{\theta}_{ij})} \tag{2.14}$$

Having defined the procedure for obtaining weighted samples, any expectation $E_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$, can be approximated as,

$$E_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})] \approx \frac{\sum_{ij} w_{ij} h(\boldsymbol{\theta}_{ij})}{\sum_{ij} w_{ij}} \tag{2.15}$$

2.2.2 Construction of the proposal region

In this section we will describe mathematically the steps needed for computing the proposal distributions q_i. There will be also presented a Bayesian optimisation alternative when gradients are not available.

Define and solve deterministic optimisation problems

For each set of nuisance variables $\mathbf{v}_i, i = \{1, 2, \dots, n_1\}$ a deterministic function is defined as $f_i(\boldsymbol{\theta}) = M_d(\boldsymbol{\theta}, \mathbf{v}_i)$. For constructing the proposal region, we search for a point $\boldsymbol{\theta}_* : d(f_i(\boldsymbol{\theta}_*), \mathbf{y_0}) < \epsilon$; this point can be obtained by solving the the following optimisation problem:

$$\min_{\boldsymbol{\theta}} \quad g_i(\boldsymbol{\theta}) = d(\mathbf{y_0}, f_i(\boldsymbol{\theta})) \tag{2.16a}$$

$$\text{subject to} \quad g_i(\boldsymbol{\theta}) \leq \epsilon \tag{2.16b}$$

We maintain a list of the solutions $\boldsymbol{\theta}_i^*$ of the optimisation problems. If for a specific set of nuisance variables $\mathbf{v}_i$, there is no feasible solution we add nothing to the list. The optimisation problem can be treated as unconstrained, accepting the optimal point $\boldsymbol{\theta}_i^* = \text{argmin}_{\boldsymbol{\theta}} g_i(\boldsymbol{\theta})$ only if $g_i(\boldsymbol{\theta}_i^*) < \epsilon$.

Gradient-based approach

The nature of the generative model $M_r(\boldsymbol{\theta})$, specifies the properties of the objective function g_i. If g_i is continuous with smooth gradients $\nabla_{\boldsymbol{\theta}} g_i$ any gradient-based iterative algorithm can be used for solving 2.16a. The gradients $\nabla_{\boldsymbol{\theta}} g_i$ can be either provided in closed form or approximated by finite differences.

Bayesian optimisation approach

In cases where the gradients are not available, the Bayesian optimisation scheme provides an alternative choice (Shahriari et al. 2016). With this approach, apart from obtaining an optimal $\boldsymbol{\theta}_i^*$, a surrogate model $\hat{d}_i$ of the distance g_i is fitted; this approximate model can be used in the following steps for making the method more efficient. Specifically, in the construction of the proposal region and in equations (2.2), (2.13), (2.15) it could replace g_i in the evaluation of the indicator function, providing a major speed-up.

[7] The description on how to estimate the bounding box is provided in the following chapters.

Construction of the proposal area q_i

After obtaining a $\boldsymbol{\theta}_i^*$ such that $g_i(\boldsymbol{\theta}_i^*) < \epsilon$, we need to construct a bounding box around it. The bounding box $\hat{\mathcal{S}}_i$ must contain the acceptance region around $\boldsymbol{\theta}_i^*$, i.e. $\{\boldsymbol{\theta} : g_i(\boldsymbol{\theta}) < \epsilon,\ d(\boldsymbol{\theta}, \boldsymbol{\theta}_i^*) < M\}$. The second condition $d(\boldsymbol{\theta}, \boldsymbol{\theta}_i^*) < M$ is meant to describe that if $\mathcal{S}_i := \{\boldsymbol{\theta} : g_i(\boldsymbol{\theta}) < \epsilon\}$ contains a number of disjoint sets of $\boldsymbol{\theta}$ that respect $g_i(\boldsymbol{\theta}) < \epsilon$, we want our bounding box to fit only the one that contains $\boldsymbol{\theta}_i^*$. We seek for a bounding box that is as tight as possible to the local acceptance region (enlarging the bounding box without a reason decreases the acceptance rate) but large enough for not discarding accepted areas.

In contrast with the OMC approach, we construct the bounding box by obtaining search directions and querying the indicator function as we move on them. The search directions $\mathbf{v}_d$ are computed as the eigenvectors of the curvature at $\boldsymbol{\theta}_i^*$ and a line-search method is used to obtain the limit point where $g_i(\boldsymbol{\theta}_i^* + \kappa \mathbf{v}_d) \geq \epsilon$[8]. The Algorithm 3 describes the method in-depth. After the limits are obtained along all search directions, we define bounding box and the uniform distribution q_i. This is the proposal distribution used for the importance sampling as explained in (2.13).

Fitting a local surrogate model $\hat{g}_i$

After the construction of the bounding box $\hat{\mathcal{S}}_i$, we are no longer interested in the surface outside the box. In the future steps (e.g. sampling, evaluating the posterior) we will only evaluate g_i inside the corresponding bounding box. Hence, we could fit a local surrogate model $\hat{g}_i$ for representing the local area around $\theta_i^* F$. Doing so, in the future steps we can exploit $\hat{g}_i$ for evaluating the indicator function instead of running the whole deterministic simulator.

Any ML regression model may be chosen as local surrogates. The choice should consider the properties of the local region (i.e. size, smoothness). The ROMC proposes fitting a simple quadratic model. The training set $X : \mathbb{R}^{N \times D}$ is created by sampling N points from the bounding box and the labels $Y : \mathbb{R}^N$ are the computed by evaluating g_i. The quadratic model is fitted on the data points, for minimising the square error.

This additional step places an additional step in the training part, increasing the computational demands, but promises a major speed at the inference phase (sampling, posterior evaluation). It is frequent in ML problems, to be quite generous with the execution time at the training phase, but quite eager at the inference phase. Fitting a local surrogate model aligns with this requirement.

2.3 Algorithmic description of ROMC

In this section, we will provide the algorithmic description of the ROMC method; how to solve the optimisation problems using either the gradient-based approach or the Bayesian optimisation alternative and the construction of the bounding box. Afterwards, we will discuss the advantages and disadvantages of each choice in terms of accuracy and efficiency.

At a high-level, the ROMC method can be split into the training and the inference part.

Training part

At the training (fitting) part, the goal is the estimation of the proposal regions q_i. The tasks are (a) sampling the nuisance variables $\mathbf{v}_i \sim p(\mathbf{v})$ (b) defining the optimisation problems $\min_{\boldsymbol{\theta}} g_i(\boldsymbol{\theta})$ (c) obtaining $\boldsymbol{\theta}_i^*$ (d) checking whether $d_i^* \leq \epsilon$ and (e) building the bounding box for obtaining the proposal region q_i. If gradients are available, using a gradient-based method is advised for obtaining $\boldsymbol{\theta}_i^*$ much faster. Providing $\nabla_{\boldsymbol{\theta}} g_i$ in closed-form provides an upgrade in both accuracy and efficiency; If closed-form description is not available, approximate gradients with finite-differences requires two evaluations of g_i for **every** parameter $\boldsymbol{\theta}_d$, which works adequately well for low-dimensional problems. When gradients are not available or g_i is not differentiable, the Bayesian optimisation paradigm exists as an alternative solution. In this scenario, the training part becomes slower due to fitting of the surrogate model and the blind optimisation steps. Nevertheless, the subsequent task of computing

[8] $-\kappa$ is used as well for the opposite direction along the search line

the proposal region q_i becomes faster since $\hat{d}_i$ can be used instead of g_i; hence we avoid to run the simulator $M_d(\boldsymbol{\theta}, \mathbf{v}_i)$ for each query. The algorithms 1 and 2 present the above procedure.

Inference Part

Performing the inference includes one or more of the following three tasks; (a) evaluating the unnormalised posterior $p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})$ (b) sampling from the posterior $\boldsymbol{\theta}_i \sim p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})$ (c) computing an expectation $E_{\boldsymbol{\theta}|\mathbf{y_0}}[h(\boldsymbol{\theta})]$. Computing an expectation can be done easily after weighted samples are obtained using the equation 2.15, so we will not discuss it separately.

Evaluating the unnormalised posterior requires solely the deterministic functions g_i and the prior distribution $p(\boldsymbol{\theta})$; there is no need for solving the optimisation problems and building the proposal regions. The evaluation requires iterating over all g_i and evaluating the distance from the observed data. In contrast, using the GP approach, the optimisation part should be performed first for fitting the surrogate models $\hat{d}_i(\boldsymbol{\theta})$ and evaluate the indicator function on them. This provides an important speed-up, especially when running the simulator is computationally expensive.

Sampling is performed by getting n_2 samples from each proposal distribution q_i. For each sample $\boldsymbol{\theta}_{ij}$, the indicator function is evaluated $\mathbb{1}_{B^i_{d,\epsilon}(\mathbf{y_0})}(\boldsymbol{\theta}_{ij})$ for checking if it lies inside the acceptance region. If so the corresponding weight is computed as in (2.13). As before, if a surrogate model $\hat{d}$ is available, it can be utilised for evaluating the indicator function. At the sampling task, the computational benefit of using the surrogate model is more valuable compared to the evaluation of the posterior, because the indicator function must be evaluated for a total of $n_1 \times n_2$ points. The sampling algorithms are presented step-by-step in algorithms 4 and **??**.

In summary, we can state that the choice of using a Bayesian optimisation approach provides a significant speed-up in the inference part with the cost of making the training part slower and a possible approximation error. It is typical in many Machine-Learning use cases, being able to provide enough time and computational resources for the training phase, but asking for efficiency in the inference part.

Algorithm 1 Training Part - Gradient-based. Requires $g_i(\boldsymbol{\theta}), p(\boldsymbol{\theta})$

```
1: for $i \leftarrow 1$ to $n$ do
2:     Obtain $\boldsymbol{\theta}_i^*$ using a Gradient Optimiser
3:     if $g_i(\boldsymbol{\theta}_i^*) > \epsilon$ then
4:         go to 1
5:     else
6:         Approximate $\mathbf{J}_i^* = \nabla g_i(\theta)$ and $H_i \approx \mathbf{J}_i^T \mathbf{J}_i$
7:         Use Algorithm 3 to obtain $q_i$
   return $q_i, p(\theta), g_i(\theta)$
```

Algorithm 2 Training Part - Bayesian optimisation. Requires $g_i(\boldsymbol{\theta}), p(\boldsymbol{\theta})$

```
1: for $i \leftarrow 1$ to $n$ do
2:     Obtain $\boldsymbol{\theta}_i^*, \hat{d}_i(\boldsymbol{\theta})$ using a GP approach
3:     if $g_i(\boldsymbol{\theta}_i^*) > \epsilon$ then
4:         go to 1
5:     else
6:         Approximate $H_i \approx \mathbf{J}_i^T \mathbf{J}_i$
7:         Use Algorithm 3 to obtain $q_i$
   return $q_i, p(\theta), \hat{d}_i(\theta)$
```

Algorithm 4 Sampling. Requires a function of distance $(g_i(\theta)$ or $\hat{d}_i$ or $\hat{g}_i), p(\theta), q_i$

```
1: for $i \leftarrow 1$ to $n_1$ do
2:     for $j \leftarrow 1$ to $n_2$ do
3:         $\boldsymbol{\theta}_{ij} \sim q_i$
4:         if $g_i(\boldsymbol{\theta}_{ij}) > \epsilon$ then
5:             Reject $\theta_{ij}$
6:         else
7:             $w_{ij} = \frac{p(\boldsymbol{\theta}_{ij})}{q(\boldsymbol{\theta}_{ij})}$
8:             Accept $\boldsymbol{\theta}_{ij}$, with weight $w_{ij}$
```

Algorithm 3 Computation of the proposal distribution q_i; Needs, a model of distance d, optimal point $\boldsymbol{\theta}_i^*$, number of refinements K, step size η and curvature matrix $\mathbf{H}_i$ ($\mathbf{J}_i^T \mathbf{J}_i$ or GP Hessian)

```
 1: Compute eigenvectors $\mathbf{v}_d$ of $\mathbf{H}_i$ $(d = 1, \dots, ||\boldsymbol{\theta}||)$
 2: for $d \leftarrow 1$ to $||\boldsymbol{\theta}||$ do
 3:     $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}_i^*$
 4:     $k \leftarrow 0$
 5:     repeat
 6:         repeat
 7:             $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} + \eta \ \mathbf{v}_d$        ▷ Large step size $\eta$.
 8:         until $d(f_i(\tilde{\boldsymbol{\theta}}), \mathbf{y_0}) > \epsilon$
 9:         $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} - \eta \ \mathbf{v}_d$
10:         $\eta \leftarrow \eta/2$        ▷ More accurate region boundary
11:         $k \leftarrow k + 1$
12:     until $k = K$
13:     Set final $\tilde{\boldsymbol{\theta}}$ as region end point.
14:     Repeat steps 3 - 13 for $\mathbf{v}_d = -\mathbf{v}_d$
15: Fit a rectangular box around the region end points and define $q_i$ as uniform distribution
```

2.4 Engine for Likelihood-Free Inference (ELFI) package

The Engine for Likelihood-Free Inference (ELFI) (Lintusaari et al. 2018) is a Python package dedicated to Likelihood-Free Inference (LFI). ELFI models in a convenient manner all the fundamental components of a probabilistic model such as priors, simulators, summaries and distances. Furthermore, ELFI already supports some recently proposed likelihood-free inference methods.

2.4.1 Modelling

ELFI models the probabilistic model as a Directed Acyclic Graph (DAG); it implements this functionality based on the package `NetworkX`, which is designed for creating general purpose graphs. Although not restricted to that, in most cases the structure of a likelihood-free model follows the pattern presented in figure 2; there are edges that connect the prior distributions to the simulator, the simulator is connected to the summary statistics that consequently are connected to the distance. The distance is the output node. Samples can be obtained from all nodes through sequential sampling. The nodes that are defined as `elfi.Prior`[9] are automatically considered as the parameters of interest and are the only nodes that, apart from sampling, should also provide PDF evaluation. The function passed as argument in the `elfi.Summary` node can be any valid Python function. Finally, the observations should be passed in the appropriate node through the argument `observed`.

2.4.2 Inference Methods

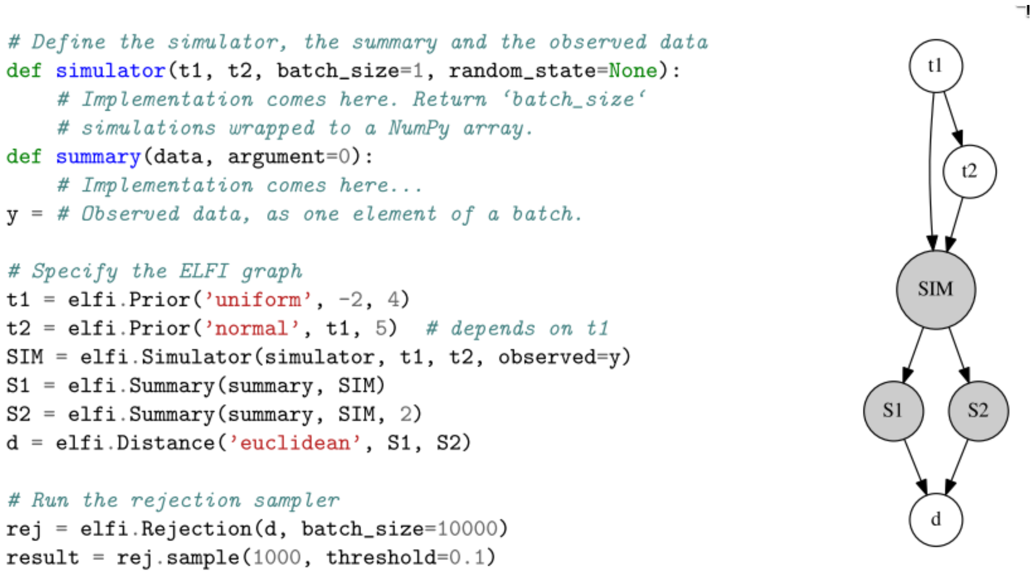
The inference Methods implemented at the ELFI follow some common guidelines;

- the initial argument should is the output node of the model. It is followed by the rest hyper-parameters of the method.

- each inference method provides a central sampling functionality. In most cases it is named `<method_name>.sample()`.

The collection of likelihood-free inference methods implemented so far contain the *ABC Rejection Sampler*, the *Sequential Monte Carlo ABC Sampler* and the *Bayesian Optimisation for Likelihood-Free Inference (BOLFI)*. The latter has methodological similarities to the ROMC method that we implement in the current work.

[9] The `elfi.Prior` functionality is a wrapper around the `scipy.stats` package.



```
# Define the simulator, the summary and the observed data
def simulator(t1, t2, batch_size=1, random_state=None):
    # Implementation comes here. Return ‘batch_size‘
    # simulations wrapped to a NumPy array.
def summary(data, argument=0):
    # Implementation comes here...
y = # Observed data, as one element of a batch.

# Specify the ELFI graph
t1 = elfi.Prior(’uniform’, -2, 4)
t2 = elfi.Prior(’normal’, t1, 5)  # depends on t1
SIM = elfi.Simulator(simulator, t1, t2, observed=y)
S1 = elfi.Summary(summary, SIM)
S2 = elfi.Summary(summary, SIM, 2)
d = elfi.Distance(’euclidean’, S1, S2)

# Run the rejection sampler
rej = elfi.Rejection(d, batch_size=10000)
result = rej.sample(1000, threshold=0.1)
```

Figure 2: Image taken from Lintusaari et al. 2018

3 Implementation

In this chapter, we will exhibit the implementation of the ROMC inference method at the ELFI package. The presentation is divided in two logical blocks; In Sections 3.1, 3.2, 3.3, 3.4 we present the functionalities provided from the **user's point-of-view**. These sections demonstrate how a practitioner could use our ROMC implementation for performing the inference in a simulator-based model. For providing a practical overview of the implementation, we set-up a simple running example and illustrate the functionalities on top of it. In contrast, in the final Section 3.5 we delve into the internals of the code, presenting all the tiny details of the implementation. This Section mainly refers to a **researcher or a developer** who would like to use ROMC as a meta-algorithm and experiment with novel approaches. We have designed our implementation preserving extensibility and customisation; hence, a researcher may intervene in parts of the method without too much effort. A driver example which demonstrates the way of extending the method with custom utilities is also contained in this Section.

3.1 General Design

In figure **??** we present an overview of our implementation; one may interpret figure **??** as a depiction of the main class of our implementation, called (`ROMC`), while the entities inside the green and blue ellipses are the main functions of the class. Following the common naming principles, the methods starting with an underscore (green ellipses) represent internal (private) functions and are not meant to be used by a user, whereas the rest of the methods (blue ellipses) are the functionalities the user interacts with. As mentioned before, the implementation favours extensibility; the building blocks that compose the method have been designed in an isolated fashion so that a practitioner may replace them without the method to collapse.

Figure **??** groups the ROMC implementation into the training, the inference and the evaluation part, following the arrangement used in the algorithmic presentation (section 2.3). The training part includes all the steps until the computation of the proposal regions; sampling the nuisance variables, defining the optimisation problems, solving them, constructing the regions and fitting local surrogate models. The inference part comprises of evaluating the unnormalised posterior (and the normalised one, in low-dimensional cases), sampling and computing an expectation. Moreover, the ROMC implementation provides some utilities for inspecting the training process, such as plotting the histogram of the distances $d_i^* = g_i(\theta_i^*), \forall i \in \{1, \dots, n_1\}$ after solving the optimisation problems and visualising the constructed bounding box[10]. Finally, two functionalities for evaluating the inference are implemented; (a) computing the Effective Sample Size (ESS) of the weighted samples and (b) measuring the divergence between the approximate posterior the ground-truth, if the latter is available.[11]

Simple 1D example

For illustrating the functionalities we choose as running example the following model, introduced by (Ikonomov and Gutmann 2019),

$$p(\theta) = \mathcal{U}(\theta; -2.5, 2.5) \tag{3.1}$$

$$p(y|\theta) = \begin{cases} \theta^4 + u & \text{if } \theta \in [-0.5, 0.5] \\ |\theta| - c + u & \text{otherwise} \end{cases} \tag{3.2}$$

$$u \sim \mathcal{N}(0, 1) \tag{3.3}$$

In the model (3.1), the prior is the uniform distribution in the range $[-2.5, 2.5]$ and the likelihood a Gaussian distribution. There is only one observation $y_0 = 0$. The inference in this particular

[10] if the parametric space is up to $2D$

[11] Normally, the ground-truth posterior is not available; that is the meaning of performing the inference! Though this functionality is useful in cases where the posterior can be computed numerically or with an alternative method (i.e. ABC Rejection Sampling) and we would like to measure the discrepancy between the two approximations.

example can be performed quite easily, without icorporating a likelihood-free inference approach. Ee can exploit this fact for validating the accuracy of our implementation. The ground-truth posterior, approximated computationally, is shown in figure 3.

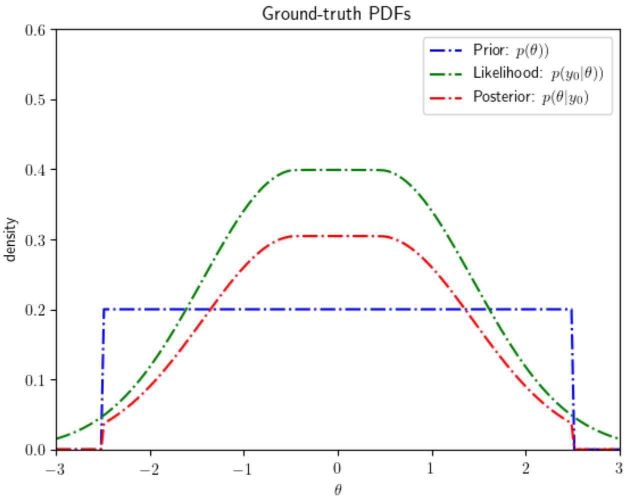


Figure 3: Ground-truth posterior distribution for our simple 1D example

ELFI code for modelling the example

In the following code snippet, we code the model at *ELFI* and we initialise the ROMC inference method. We observe that the initialisation of the ROMC inference method is quite intuitive; we just pass the final (distance) node of the simulator as argument, as in all *ELFI* inference methods. The argument bounds, although optional, is important for many functionalities (e.g. approximating the partition function, setting the bounds of the Bayesian optimisation etc.) so it is recommended to be passed.

```
import elfi
import scipy.stats as ss
import numpy as np

def simulator(t1, batch_size=1,random_state=None):
    if t1 < -0.5:
        y = ss.norm(loc=-t1-c, scale=1).rvs(random_state=random_state)
    elif t1 <= 0.5:
        y = ss.norm(loc=t1**4, scale=1).rvs(random_state=random_state)
    else:
        y = ss.norm(loc=t1-c, scale=1).rvs(random_state=random_state)
    return y

# observation
y = 0

# Elfi graph
t1 = elfi.Prior('uniform', -2.5, 5)
sim = elfi.Simulator(simulator, t1, observed=y)
d = elfi.Distance('euclidean', sim)
```

```
# Define ROMC inference method
bounds = [(-2.5, 2.5)]
romc = elfi.ROMC(d, bounds=bounds)
```

3.2 Training

The training part contains the 6 following functionalities:

- `romc.solve_problems(n1, use_bo=False, optimizer_args=None, seed=None)`

-
  ```
  romc.estimate_regions(eps_filter,
                        use_surrogate=None, region_args=None,
                        fit_models=False, fit_models_args=None,
                        eps_region=None, eps_cutoff=None)
  ```

-
  ```
  romc.fit_posterior(n1, eps_filter, use_bo=False, optimizer_args=None,
                     seed=None, use_surrogate=None, region_args=None,
                     fit_models=False, fit_models_args=None,
                     eps_region=None, eps_cutoff=None)
  ```
- `romc.distance_hist(savefig=False, **kwargs)`
- `romc.visualize_region(i, savefig=False)`
- `romc.compute_eps(quantile)`

Function (i): Define and solve the optimisation problems

```
romc.solve_problems(n1, use_bo=False, optimizer_args=None, seed=None)
```

This routine is responsible for (a) drawing the nuisance variables, (b) define the optimisation problems and (c) solve them using either a gradient-based optimiser or Bayesian optimisation. The aforementioned tasks are done in a sequential fashion, as show in figure **??**. The defininition of the optimisation problems is performed by drawing n_1 integer numbers from a discrete uniform distribution $u_i \sim \mathcal{U}\{1, 2^{32} - 1\}$. Each integer u_i is the seed used in ELFI's random simulator. Hence from an algorithmic point-of-view drawing the state of all random variables $\mathbf{v}_i$ as described in the previous chapter, traces back to just setting the seed that initialises the state of the pseudo-random generator, before asking a sample from the simulator.

Finally, passing an integer number as the argument `seed` absorbs all the randomness of the optimistation part (e.g. drawing initial points for the optimisation procedure), making the whole process reproducible.

Setting the argument `use_bo=True`, chooses the Bayesian Optimisation scheme for obtaining θ_i^*. In this case, apart from obtaining the optimal points θ_i^*, we also fit a Gaussian Process (GP) as surrogate model $\hat{d}_i$. In the following steps, $\hat{d}_i(\boldsymbol{\theta})$ will replace $g_i(\boldsymbol{\theta})$ when calling the indicator function.

Function (ii): Construct bounding boxes and fit local surrogate models

```
romc.estimate_regions(eps_filter,
                      use_surrogate=None, region_args=None,
                      fit_models=False, fit_models_args=None,
                      eps_region=None, eps_cutoff=None)
```

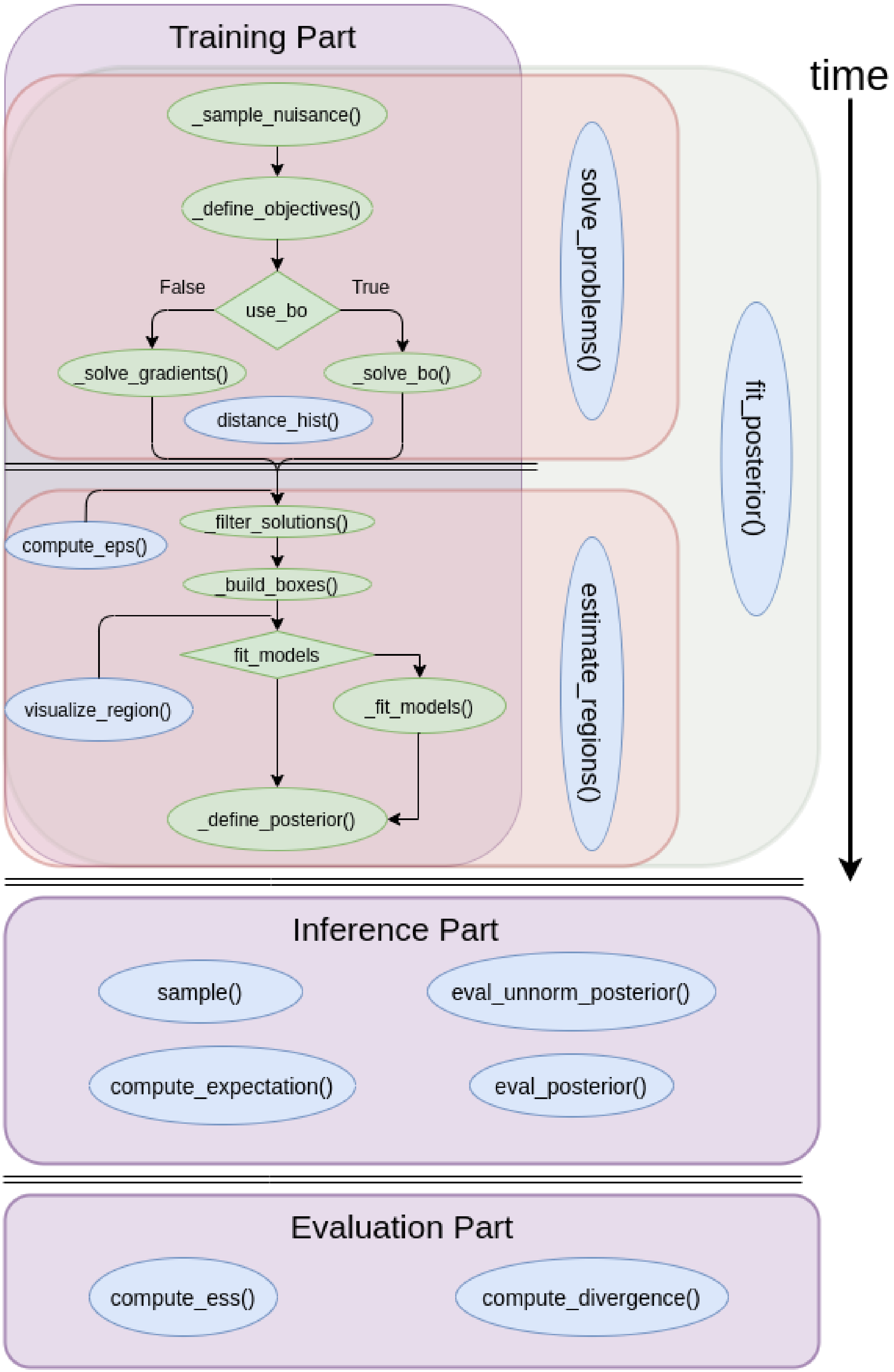


Figure 4: Overview of the ROMC implementation. The training part follows a sequential pattern; the functions in the green ellipses must be called in a sequential fashion for completing the training part and define the posterior distribution. The functions in blue ellipses are the functionalities provided to the user.

This routine constructs the bounding boxes around the optimal points $\boldsymbol{\theta}_i^* : i = 1, 2, \dots, n_1$ following the Algorithm 3. The Hessian matrix is approximated based on the Jacobian $\mathbf{H}_i = \mathbf{J}_i^T \mathbf{J}_i$. The eigenvectores are computed using the function `numpy.linalg.eig()` that calls, under the hood, the `_geev LAPACK`. A check is performed so that the matrix $\mathbf{H}_i$ is not singular; if this is the case, the eigenvectors are set to be the vectors of the orthonormal basis. Afterwards, the limits are obtained by repeteadly querying the distance function ($g_i(\boldsymbol{\theta})$ or $\hat{d}(\boldsymbol{\theta})$) along the search directions. In section 3.5, we provide some details regarding the way the bounding box is defined as a class and sampling is performed on it.

Function (iii): Perform all training steps in a single call

```
romc.fit_posterior(n1, eps_filter, use_bo=False, optimizer_args=None,
                   seed=None, use_surrogate=None, region_args=None,
                   fit_models=False, fit_models_args=None,
                   eps_region=None, eps_cutoff=None)
```

This function merges all steps for constructing the bounding box into a single command. If the user doesn't want to manually inspect the histogram of the distances before deciding where to set the threshold ϵ, he may call `romc.fit_posterior()` and the whole training part will be done end-to-end. There are two alternatives for setting the threshold ϵ; the first is to set to a specific value blindly and the second is to set at as a specific quantile of the histogram of distances. In the second scenario the `quantile` argument must be set to a floating number in the range $[0, 1]$ and `eps='auto'`.

Function (iv): Plot the histogramm of the optimal points

```
romc.distance_hist(**kwargs)
```

This function can serve as an intermediate step of manual inspection, for helping the user choose which threshold ϵ to use. It plots a histogram of the distances at the optimal point $g_i(\boldsymbol{\theta}_i^*) : \{i = 1, 2, \dots, n_1\}$ or d_i^* in case `use_bo=True`. The function accepts all keyword arguments and forwards them to the underlying `matplotlib.hist()` function; in this way the user may customize some properties of the histogram, such as the number of bins or the range of values.

Function (v): Plot the acceptance region of the objective functions

```
romc.visualize_region(i)
```

It can be used as an inspection utility for cases where the parametric space is up to two dimensional. The argument `i` is the index of the corresponding optimization problem i.e. $i < n_1$.

Function (vi): Compute ϵ automatically based on the distribution of d^*

```
romc.compute_eps(quantile)
```

This function return the appropriate distance value $d^*_{i=\kappa}$ where $\kappa = \lfloor \frac{quantile}{n} \rfloor$ from the collection $\{d_i^*\} \forall i = \{1, \dots, n\}$ where n is the number of accepted solutions. It can be used to automate the selection of the threshold ϵ, e.g. `eps=romc.compute_eps(quantile=0.9)`.

Example

Here we will illustrate the aforementioned functionalities using the simple 1D example we set up in the previous chapter. The following code snippet performs the training part at ELFI.

```
n1 = 500
seed = 21
eps = .75
use_bo = False # set to True for switching to Bayesian optimisation
```

```
# Training set-by-step
romc.solve_problems(n1=n1, seed=seed, use_bo=use_bo)
romc.theta_hist(bins=100)
romc.estimate_regions(eps=eps)
romc.visualize_region(i=1)

# Equivalent one-line command
# romc.fit_posterior(n1=n1, eps=eps, use_bo=use_bo, seed=seed)
```

As stated before, switching to the Bayesian optimisation scheme needs nothing more the setting the argument `use_bo=True`; all the following command remain unchanged. In figure 8 we illustrate the distribution of the distances obtained and the acceptance area of the first optimisation problem. We observe that most optimal points produce almost zero distance.

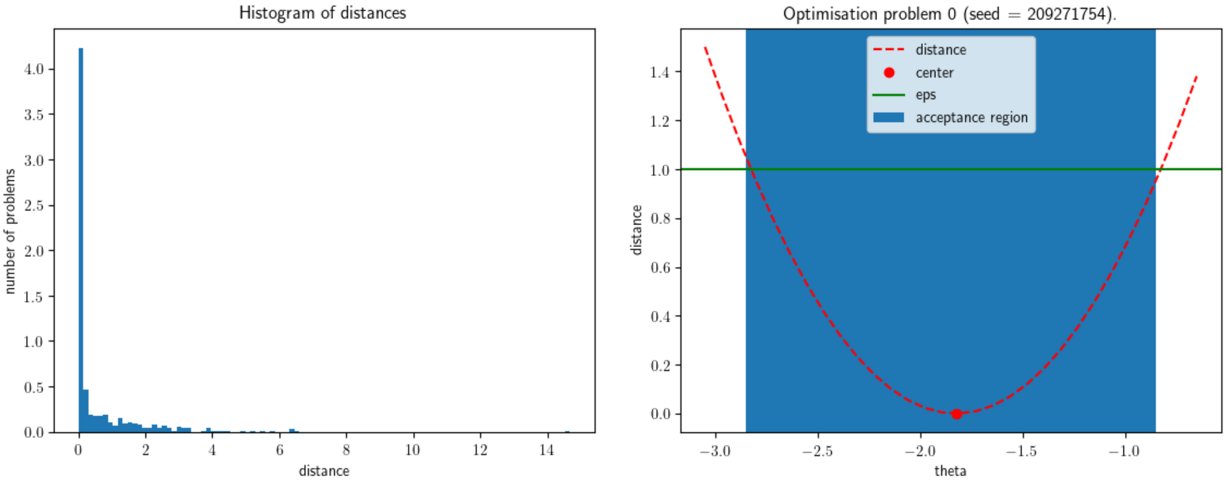


Figure 5: Histogram of distances and visualisation of a specific region.

3.3 Performing the Inference

The inference part contains the 4 following functionalities:

- `romc.sample(n2, seed=None)`
- `romc.compute_expectation(h)`
- `romc.eval_unnorm_posterior(theta)`
- `romc.eval_posterior(theta)`

Function (i): Perform weithted sampling

`romc.sample(n2)`

This is the basic inference utility of the ROMC implementation. The samples are drawn from a uniform distribution q_i defined over the corresponding bounding box and the weight w_i is computed as in equation (2.13). The function stores an `elfi.Result` object as `romc.result` attribute. The `elfi.Result` provides by some usefull functionalities for inspecting the obtained samples such as `romc.result.summary()` that prints the number. A comlete overview of these functionalities is provided in ELFI's official documentation.

Function (ii): Compute an expectation

```
romc.compute_expectation(h)
```

This function computes the expectation $E_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$ using the expression (2.15). The argument `h` can be any python `Callable` that accepts a one-dimensional `np.ndarray` as input and returns a one-dimensional `np.ndarray` as output.

Function (iii): Evaluate the unnormalised posterior

```
romc.eval_unorm_posterior(theta, eps_cutoff=False)
```

This function computes the unnormalised posterior approximation using the expression (2.2).

Function (iv):Evaluate the normalised posterior

```
romc.eval_posterior(theta, eps_cutoff=False)
```

This function evaluates the normalised posterior. For doing so it needs to approximate the partition function $Z = \int_{\boldsymbol{\theta}:p(\boldsymbol{\theta})>0} p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})d\boldsymbol{\theta}$; this is done using the Riemann integral aproximation. Unfortunatelly, the Riemann approximation does not scale well in high-dimensional spaces, hence the approximation is tractable only at low-dimensional parametric spaces. Given that this functionality is particulary useful for ploting the posterior, we could say that it is meaningful to be used for up to $3D$ parametric spaces, even though it is not restricted to that. Finally, for this functionality to work, the left and right limit determinining a bounding box that includes the area where the prior distribution has mass must have been passed as the `left_lim, right_lim` in the initialisation of the `elfi.ROMC` object.

Example - Sampling and compute expectation

With the following code snippet, we perform weighted sampling from the ROMC approximate posterior. Afterwards, we used some ELFI's built-in tools to get a summary of the obtained samples. In figure 6, we observe the histogram of the weighted samples and the acceptance region of the first deterministic function (as before) alongside with the obtained samples obtained from it. Finally, in the code snippet we demonstrate how to use the `compute_expectation` function; in the current example we define `h` in order to compute firstly the empirical mean and afterwards the empirical variance. In both cases, the empirical result is close to the ground truth $\mu = 0$ and $\sigma^2 = 1$.

```
seed = 21
n2 = 50
romc.sample(n2=n2, seed=seed)

# visualize region, adding the samples now
romc.visualize_region(i=1)

# Visualise marginal (built-in ELFI tool)
romc.result.plot_marginals(weights=romc.result.weights, bins=100, range=(-3,3))

# Summarize the samples (built-in ELFI tool)
romc.result.summary()
# Number of samples: 1720
# Sample means: theta: -0.0792

# compute expectation
```

```
print("Expected value   : %.3f" % romc.compute_expectation(h=lambda x: np.squeeze(x)))
# Expected value   : -0.079

print("Expected variance: %.3f" % romc.compute_expectation(h=lambda x: np.squeeze(x)**2))
# Expected variance: 1.061
```

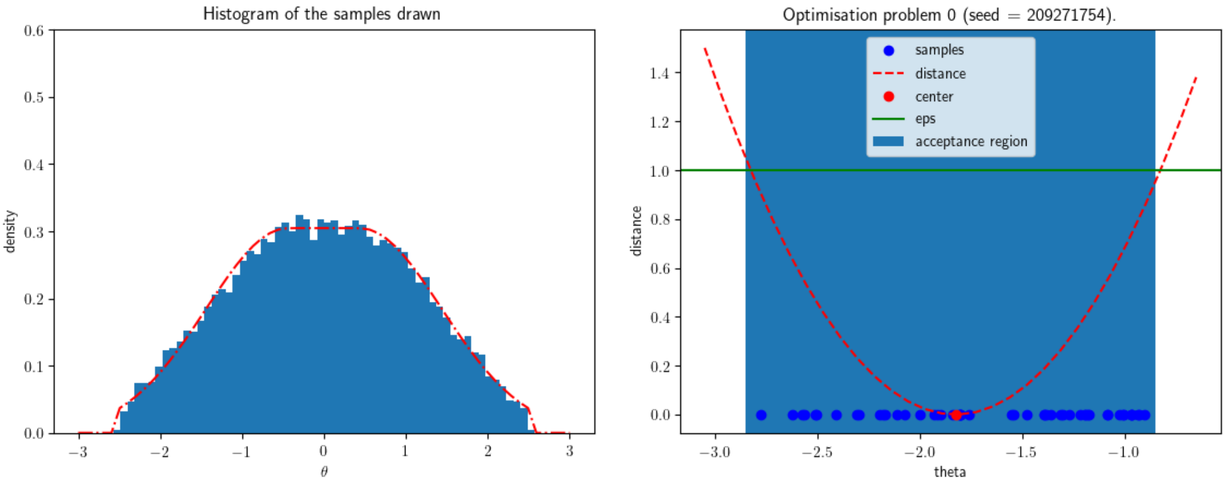


Figure 6: (a) Left: Histogram of the obtained samples. (b) Right: Acceptance region around θ_1^* with the obtained samples ploted inside.

Example - Evaluate Posterior

The `romc.eval_unnorm_posterior(theta)` evaluates the posterior at point θ using the expression (2.8). The `romc.eval_posterior(theta)` approximates the partition function $Z = \int_{\boldsymbol{\theta}} p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})d\boldsymbol{\theta}$ using the Riemann approximation in the points where the prior has mass; hence it doesn't scale well to high-dimensional spaces. In our simple example, this utility can provide a nice plot of the approximate posterior as illustrated in figure 7. We observe that the approximation is quite close to the ground-truth posterior.

3.4 Evaluation

The ROMC implementation provides two functions for evaluating the inference results,

- `romc.compute_divergence(gt_posterior, bounds=None, step=0.1,`
  `distance="Jensen-Shannon")`
- `romc.compute_ess()`

Function (i): Compute the divergence between ROMC approximation and a ground-truth posterior

```
romc.compute_divergence(gt_posterior, bounds=None, step=0.1,
    distance="Jensen-Shannon")
```

This function computes the divergence between the ROMC approximation and the ground truth posterior. Since the computation is performed using the Riemann's approximation, this method can only work in low dimensional parametric spaces; it is suggested to be used for up to a $3D$ parametric space. As mentioned in the beginning of this chapter, in a real-case scenario it is not expected the

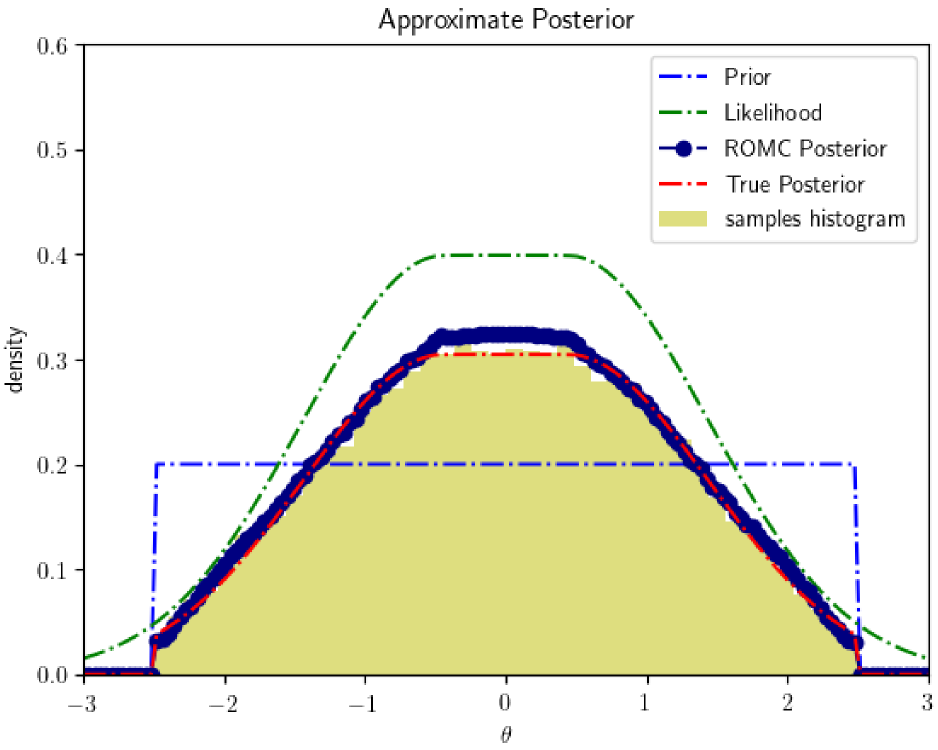


Figure 7: Approximate posterior evaluation.

ground-truth posterior to be available; this is actually the whole meaning of performing the inference. However, in cases where the posterior can be approximated decently well with a computational approach (as in the current example) or with any another inference approach, this function can provide a numerical measure of the aggreement between the two approximations. The argument `step` defines the step used in the Riemann's approximation and the argument `distance` can take either the `Jensen-Shannon` or the `KL-divergence` value, for computing the appropriate distance.

Function (ii): Compute the effective sample size of the weighted samples

```
romc.compute_ess()
```

This function compute the Effective Sample Size (ESS) using the following expression,

$$ESS = \frac{(\sum_i w_i)^2}{\sum_i w_i^2} \tag{3.4}$$

The ESS is a usefull measure of the **actual** sample size, when the samples are weighted. For example if in a population of 100 samples one has a very large weight (e.g. ≈ 100) whereas the rest have small (i.e. ≈ 1), the real sample size is close to 1; one sample dominates over the rest. Hence, the ESS provides a measure of the equivalent uniformly weighted sample population.

```
res = romc.compute_divergence(wrapper, distance="Jensen-Shannon")
print("Jensen-Shannon divergence: %.3f" % res)
# Jensen-Shannon divergence: 0.025

print("Nof Samples: %d, ESS: %.3f" % (len(romc.result.weights), romc.compute_ess()))
# Nof Samples: 19950, ESS: 16694.816
```

3.5 Implementation details for developers

In the previous sections we described ROMC from the user's point-of-view; the demonstration focused on performing the inference with the ready-to-use tools. Apart from this use-case, a practitioner can also use ROMC as a meta-algorithm, adding custom algorithms as part of the method. Our implementation allows such intervention. In the rest of the chapter, we will initially present the main internal classes and then then we will exhibit how to add custom methods.

3.5.1 Entities presentation

In figure 8 we provide an overview of the classes used in the algorithm. The class `ROMC` can be thought as the interface between the user and the method. The rest of the classes are responsible for the back-end functionality. It is important for the reader to remember the sequence of the events for performing the inference, as demonstrated in figure 4.

The class `ROMC` is the main class of the method; it is initialised when the user calls the method for the first time and its state is updated throughout the inference.The initialisation of the `ROMC` object sets the attributes `model`, `model_prior`, `bounds` and `inference_state` to the appropriate values; the rest of the attributes are set to `None`.[12].

The `_sample_nuisance()` routine samples n_1 integers from the discrete integer distribution. The `_define_objectives()` routine initialises n_1 `OptimisationProblem` objects, appends them to a `List` and stores this `List` as the `romc.optim_problems` attribute. The `_define_objectives()` function, initialises only the attributes `objective` and `bounds` of the `OptimisationProblem`; the rest are set to `None`. The attribute `objective`, which is a `Callable`, is created by setting the seed of the `model.generate()` to a specific integer, turning the random generator to a deterministic.

Afterwards, depending on the boolean argument `use_bo=True/False`, the function `_solve_gradients` or `_solve_bo` is called. Both of them follow the same sequence of steps; for each optimisation problem they (a) solve it, (b) create a `RomcOptimisationResult` object with the solution and (c) store it as the `OptimisationProblem.result` attribute. Although they both apply the same sequence of steps, each method uses a different optimisation scheme; `_solve_gradients` uses the `minimize` class of the `scipy.optimize` library (Virtanen et al. 2020), whereas `_solve_bo` uses the `BoDeterministic` class. `BoDeterministic` is a class we implemented for performing Bayesian Optimisation and fitting a Gaussian Process surrogate model to the objective function. It relies on the Gpy framework (GPy since 2012) for fitting the Gaussian Process. The only difference between `_solve_gradients` and `_solve_bo`, as effect in the state of the `OptimisationProblem` class, is the initialisation of the attribute `OptimisationProblem.surrogate`, which is done only by `_solve_bo`. The attribute is set to a `Callable` that wraps the `GPyRegression.predict_mean` function. If `_solve_gradients` is called, the attribute remains `None`.

At this level, all optimisation problems have been solved. The method `_filter_solutions` is used for discarding the optimisation results that are over threshold. Afterwards, `_build_boxes` estimates the bounding boxes around the accepted objective functions. For each accepted objective function, a `RegionConstructor` object is initialised in order to construct the region. In the current implementation, for each objective function we constract a single bounding box, but this may change in a future approach; for being able to support multiple bounding boxes per objective function, we decided to return a `List` of `NDimBoundingBox` objects[13]. The `List` is stored as the `OptimisationProblem.regions` attribute.

By now, we have estimated the bounding boxes around the optimal points. The last step, before defining the posterior, is the fitting local surrogate models. This step is optional and is performend only if the argument `fit_models` is set to `True`. The routine `_fit_models`, fits a quadratic model on the area around the optimal point for each objective function. For achieving so, it asks for samples from the `NDimBoundingBox` object and evaluates them using the `OptimisationProblem.objective` function. Afterwards, based on these points, it fits a quadratic model using the `linear_model.LinearRegression`

[12] in all cases, we use the value None for indicating that an attribute is not yet initialised."

[13] Each `NDimBoundingBox` object represents a region.

and preprocessing.PolynomialFeatures functions of the scikit-learn package (Pedregosa et al. 2011). The trained model is stored as the attribute OptimisationProblem.local_surrogate.

Finally, the _define_posterior method is used for creating the RomcPosterior object and storing it as the ROMC.posterior attribute. The method collects (a) all the bounding boxes created so far (accessing the OptimisationProblem.regions attributes and (b) all the objective functions. As the objective function it collects either OptimisationProblem.local_surrogate or OptimisationProblem.surrogate or OptimisationProblem.objective, depending on which one is available based on the previous steps.

The description above summarises the sequence of steps needed for the training part of the method. The conclusion is that a ROMC object is intialised when the user calls the method. Throughout the inference process, the ROMC object is always as a specific state, which gets updated whenever an algorithmic step is executed. The rest of the classes provide objects which are stored as attributes of ROMC.

3.5.2 Extensibility of the ROMC method

In this section we will explain how a practitioner may replace some parts of the ROMC method with their custom alogorithms. We can locate at least four such points; (a) the gradient-based solver (b) the Bayesian Optimisation solver (c) the bounding box region construction and (d) the fitting of the local surrogate models. Each of the aforementioned tasks may be approached with completely different algorithms than the ones we propose, without the rest of the method to change.

The four replacable parts described above, are solved using the four methods of the OptimisatioProblem class; solve_gradients(**kwargs), solve_bo(**kwargs), build_region(**kwargs), fit_local_surrogate(** Therefore, the practitioner should not alter at all the basic ROMC class. Instead, they should deploy a custom optimisation problem class which inherits the basic OptimisatioProblem class and overwrites the above four functiones with custom ones. The only rule that should be followed carefully, is that the cutom methods must have the same effect on the attributes of the ROMC class, i.e. update them with the appropriate objects as presented in table 3.5.2. For example, a function that replaces solve_gradients() must have the effect of storing a RomcOptimisationResult object to the OptimisationObject attribute.

Table 1: Table explaining the appropriate effect of each replacable routine, i.e. which object they should attach to the appropriate attribute. The functions of the first column (ROMC class) call the correpsonding functions of the second column (OptimisationProblem class). The functions of the second column should execute their main functionality and update the appropriate attribute with a certain object, as described in the third column.

OptimisationProblem	Effect
solve_gradients()	result <- RomcOptimisationResult

solve_bo()	result <- RomcOptimisationResult
	surrogate <- Callable

build_region()	regions <- List[NDimBoundingBox]
fit_local_surrogate()	local_surrogate <- Callable

Example: use a Neural Network as a local surrogate model

Let's say we have observed that the local area around θ_i^* is too complex to be represented by a simple quadratic model[14]. Hence, the user selects a neural network as a good alternative. In the following snippet, we demonstrate how they could implement this enhancement, without much effort; (a) they have to develop the neural network using the package of their choice (b) they must create a custom optimisation class which inherites the basic OptimisationClass and (c) they have to overwrite the fit_local_surrogate routine, with one that sets the neural network's prediction function as the local_surrograte attribute. The argument **kwargs may be used for passing all the important arguments e.g. training epochs, gradient step etc. If, for example, they would like

[14] as in the current implementation

to set the size of the training set dynamically, we may replace `x = self.regions[0].sample(30)` with `x = self.regions[0].sample(kwargs["nof_examples"])`. Finally, they must pass the custom optimisation class, when calling the `ROMC` method.

```
class NeuralNetwork:
    def __init__(self, **kwargs):
        # set the input arguments

    def train(x, y):
        # training code

    def predict(x):
        # prediction code

# Inherit the base optimisation class
class customOptim(elfi.methods.parameter_inference.OptimisationProblem):
    def __init__(self, **kwargs):
        super(customOptim, self).__init__(**kwargs)

    # overwrite the function you want to replace
    def fit_local_surrogate(**kwargs):
        # init and train the NN
        x = self.regions[0].sample(30) # 30 training points
        y = [np.array([self.objective(ii) for ii in x])]
        nn = NeuralNet()
        nn.train(x,y)

        # set the appropriate attribute
        self.local_surrogate = nn.predict

        # update the state
        self.state["local_surrogate"] = True

# pass the custom inference method as argument
romc = elfi.ROMC(dist, bounds, custom_optim_class=customOptim)
```

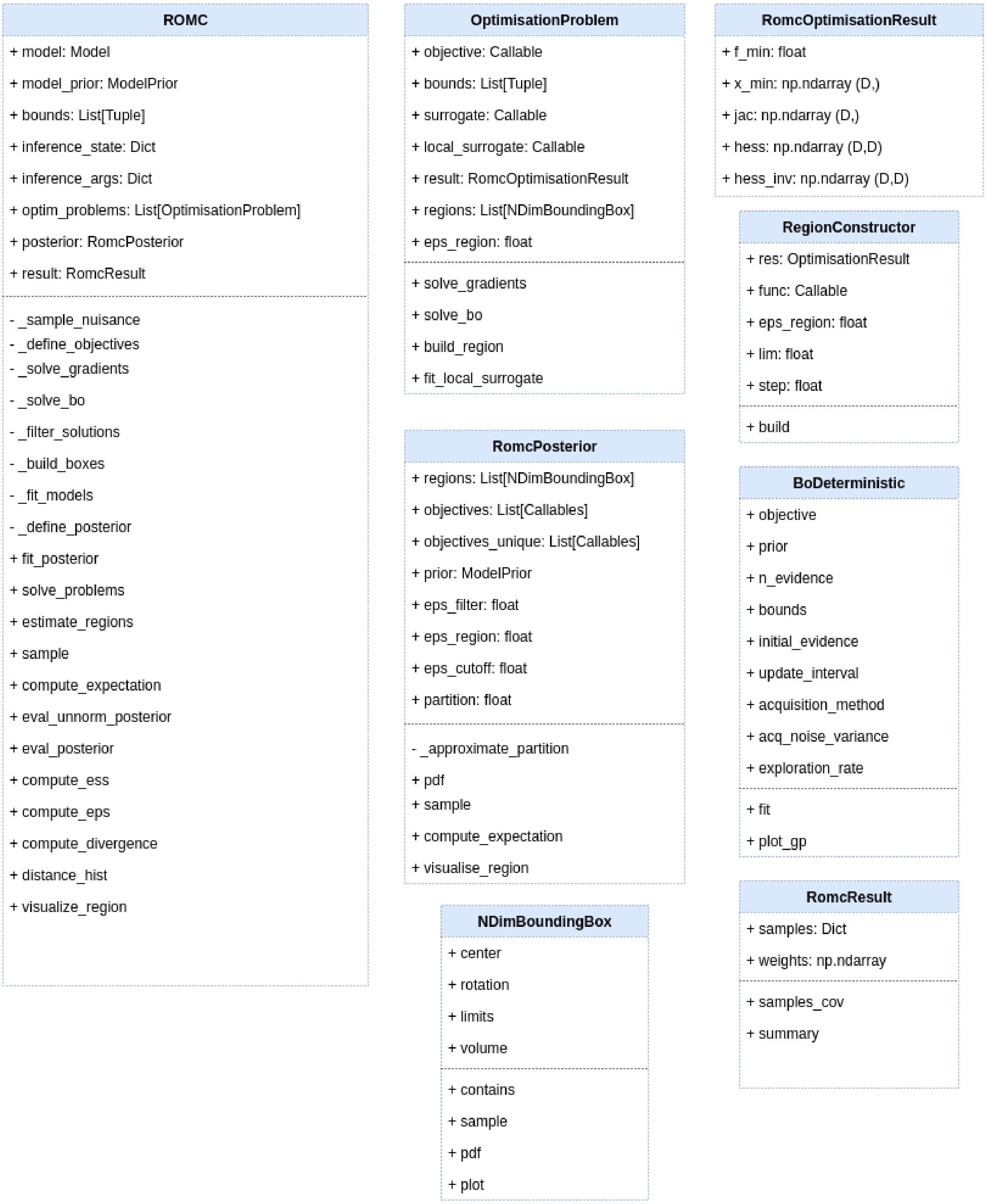


Figure 8: Histogram of distances and visualisation of a specific region.

4 Experiments

This section presents the ROMC implementation on some real-case examples. It is used to present the accuracy of the method both at a conceptual and the implementation level. The two examples that will be presented are multidimensional, in order to confirm that method works fine in higher-dimensions. The first one is chosen to be a simple $2D$ with tractable likelihood, for the ground-truth information to be available for validation purposes. The second one is the second-degree moving average model, which is a basic example used by the ELFI package for checking the inference methods. This model is also chosen to illustrate that our implementation performs well at a general model, not implemented by us.

4.1 Example 1: Simple 2D example

This examples is implemented for validating that the ROMC implementation works accurately in a multidimensional parameter space.

Problem Definition

The equations describing this inference problem are presented below.

$$p(\boldsymbol{\theta}) = p(\theta_1)p(\theta_2) = \mathcal{U}(\theta_1; -2.5, 2.5)\mathcal{U}(\theta_2; -2.5, 2.5) \tag{4.1}$$

$$p(\mathbf{y}|\boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}; \boldsymbol{\theta}, \mathcal{I}) \tag{4.2}$$

$$p(\boldsymbol{\theta}|\mathbf{y}) = \frac{1}{Z} p(\boldsymbol{\theta}) p(\mathbf{y}|\boldsymbol{\theta}) \tag{4.3}$$

$$Z = \int_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) p(\mathbf{y}|\boldsymbol{\theta}) d\boldsymbol{\theta} \tag{4.4}$$

As shown in equations (4.1), in this simple example, it is feasible to evaluate the likelihood and the unnormalised posterior. Additionally, the partition function Z can be estimated using the Riemann's approximation. Hence, computing the ground-truth posterior is computationally feasible. Setting the observation to $\mathbf{y_0} = (-0.5, 0.5)$, the ground-truth posterior is illustrated in figure 12. In table 4.1, we present the ground-truth statistics i.e. μ, σ of the marginal posterior distributions.

Performing the inference

We perform the inference using the following hyperparameters $n_1 = 500, n_2 = 30, \epsilon = 0.4$. This set-up leads to a total of 15000 samples. As observed in the histogram of distances 9, in the gradient-based approach, all optimisation problems reach an almost zero-distance end point; hence all optimal points are accepted. At the Bayesian optimisation scheme, the vast majority of the optimisation procedures has the behaviour; there are only 4 optimal distance above the limit. In figure 10, the acceptance area of a specific optimisation problem is demonstrated. We observe that both optimisation schemes lead to a similar bounding box construction. This specific example is representative of the rest of the optimisation problems; due to the simplicity of the objective function, in most cases the optimal points are similar and the surrogate model represents accurately the local region. Hence, similar proposal regions are obtained by the two optimisation alternatives.

The histograms of the marginal distributions, based on the weighted samples, are presented in figure 11. In the same figure, we can also plot the ground-truth distribution with the red dotted line. We observe that the weighted samples follow quite accurately the ground-truth distribution. This is also confirmed by the statistics provided in table 4.1; the sample mean μ and standard deviation σ are similar to the ground-truth for both parameters θ_1 and θ_2. We also observe that both optimisation schemes produce accurate samples.

Finally, the ground-truth and the approximate posteriors are presented in figure 12. We also confirm that the approximations are close to the ground truth posteriors. As a side-notice, we can observe that the approximate posteriors present a diamond-shape in the mode of the posterior; this happens due to the approximation of a circular gaussian-shape posterior with a sum of square boxes. The divergence between the ground-truth distribution and the approximate ones is 0.077, using the Jensen-Shannon distance, which confirms the matching between the two posteriors.

In this experiment we observed that the implementation fulfilled the theoretical expectations for the ROMC inference method.

	μ_{θ_1}	σ_{θ_1}	μ_{θ_2}	σ_{θ_2}	Divergence
Ground-truth	-0.45	0.935	0.45	0.935	
ROMC (gradient-based)	-0.474	0.994	0.502	0.966	0.068
ROMC (Bayesian optimisation)	-0.485	0.987	0.511	0.939	0.069

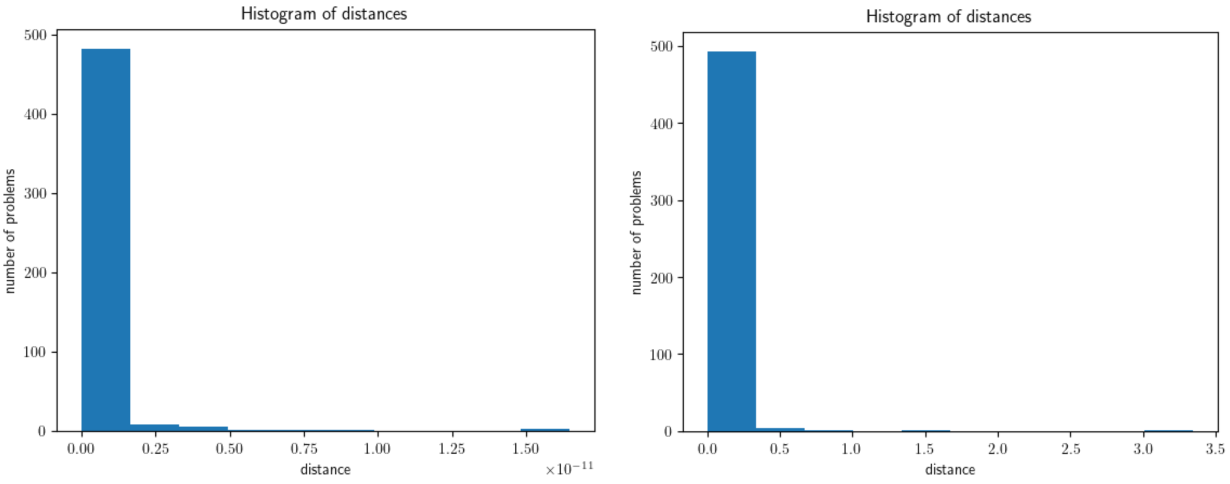


Figure 9: Histogram of distances $d^*_{i,i\in 1,\ldots,n_1}$. The left graph corresponds to the gradient-based approach and the right one to the Bayesian optimisation approach.

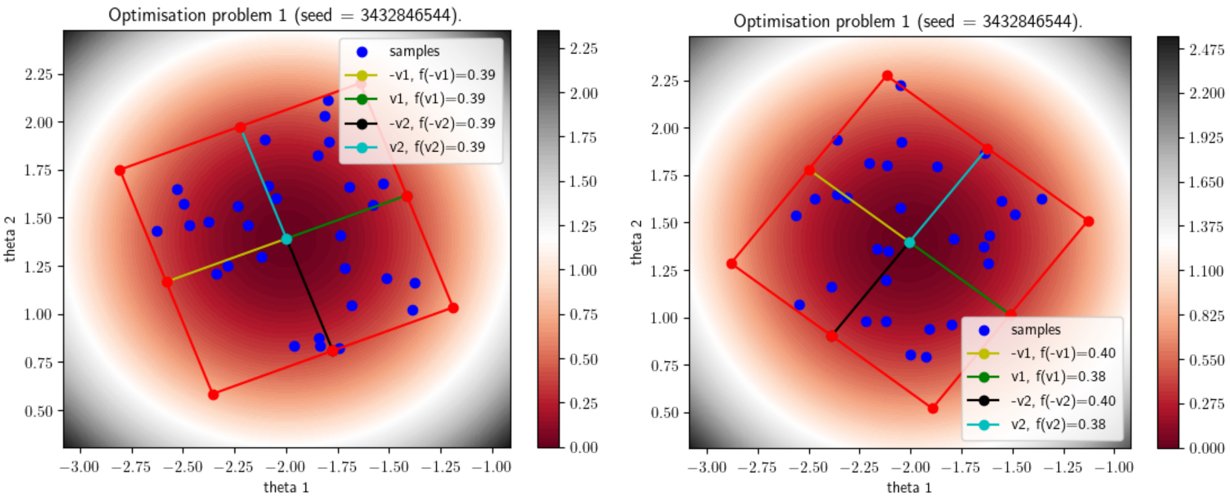


Figure 10: Visualisation of the acceptance region in 3 different optimisation problems. Each row illustrates a different optimisation problem, the left column corresponds to the gradient-based approach and the right column to the Bayesian optimisation approach. The examples have been chosen to illustrate three different cases; in the first case, both optimisation schemes lead to similar optimal point and bounding box, in the second case the bounding box is similar in shape but a little bit shifted to the right relatively to the gradient-based approach and in the third case, both the optimal point and the bounding box is completely different.

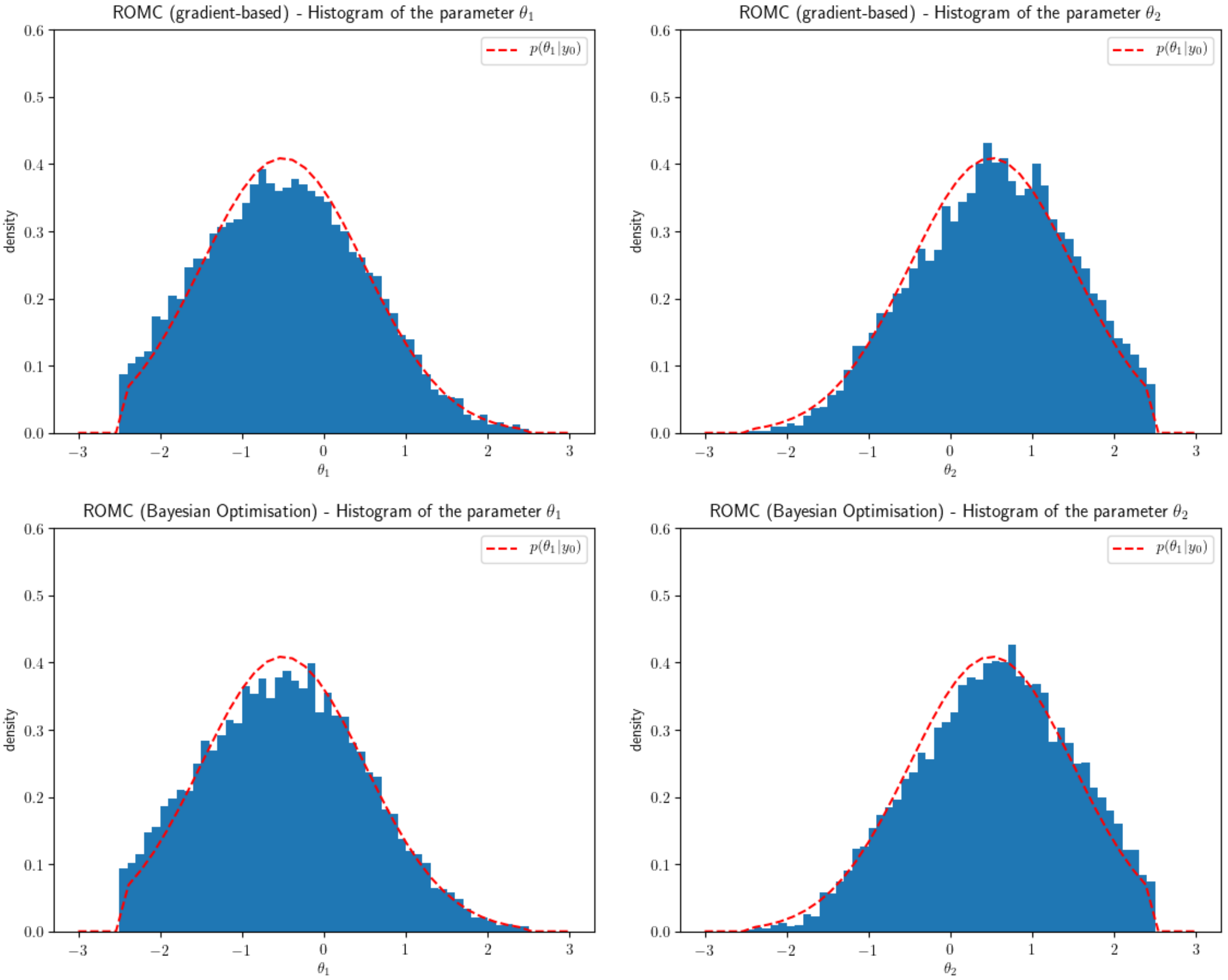


Figure 11: Histogram of the marginal distribution for three different inference approaches; (a) in the first row, the approximate posterior samples are obtained using Rejection ABC sampling (b) in the second row, using ROMC sampling with gradient-based approach and (c) in the third row, using ROMC sampling with Bayesian optimisation approach. The vertical (red) line represents the samples mean μ and the horizontal (black) the standard deviation σ.

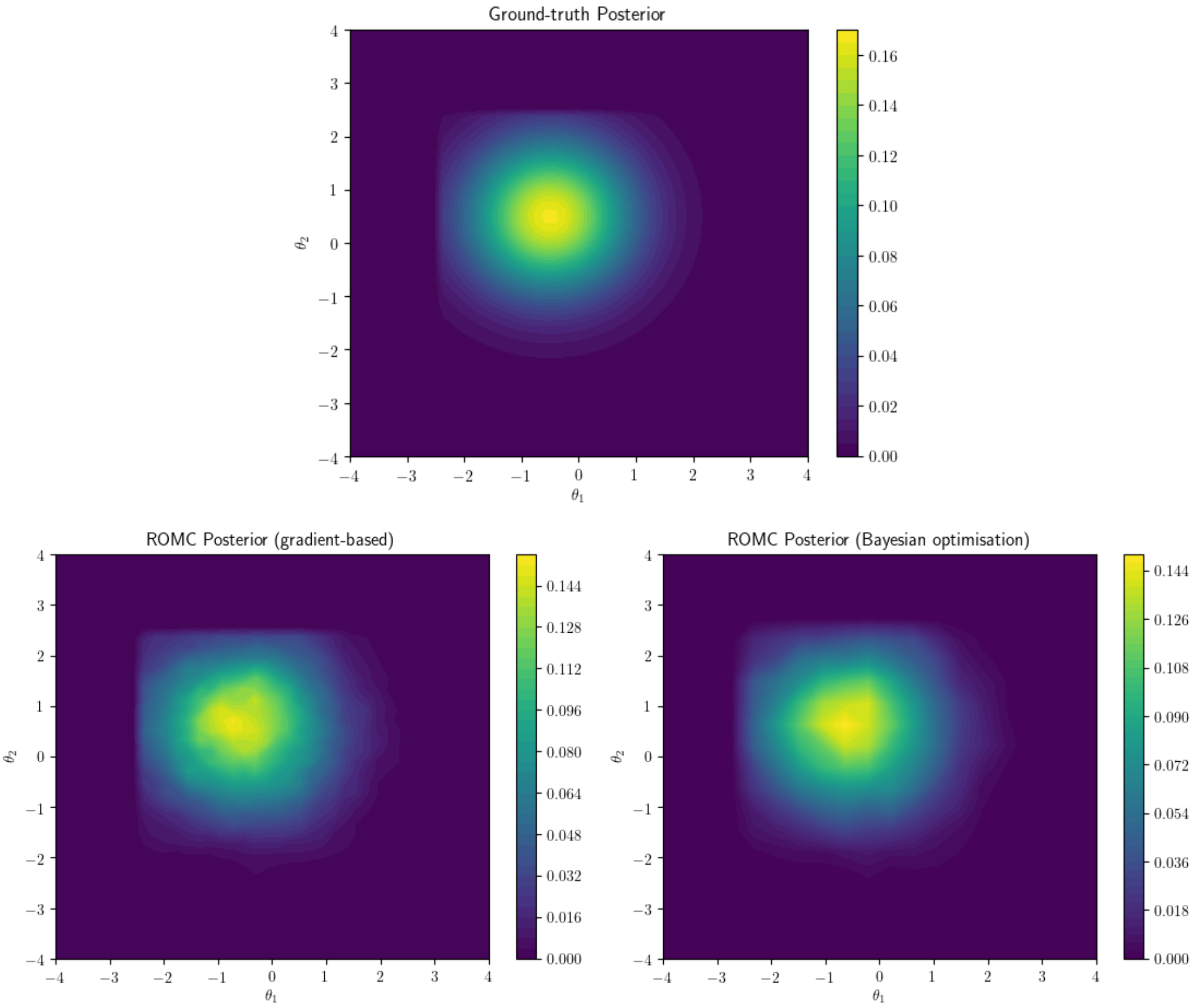
In this simple artificial 2D example, with the ground-truth information available, we confirmed that our implementation produces accurate approximations. In the following section, we will question it in a more involved case.

4.2 Example 2: Second-order Moving Average MA(2)

The second example is the second-order moving average (MA2) which is used by the ELFI package as a fundamental model for testing all inference implementations. This example is chosen to confirm that our implementation of the ROMC approach produces sensible results in a general model, since the previous ones where artificially created by us.

Problem definition

The second-order moving average (MA2) is a common model used for univariate time series analysis. The observation at time t is given by:


Ground-truth Posterior

ROMC Posterior (gradient-based)

ROMC Posterior (Bayesian optimisation)

Figure 12: (a) First row: Ground-truth posterior approximated computationally. (b) Second row (left): ROMC approximate posteriors using gradient-based approach. The divergence from the ground-truth using the Jensen-Shannon distance is 0.068. (c) Second row (right): ROMC approximate posterior using Bayesian optimisation. The divergence from the ground-truth using the Jensen-Shannon distance is 0.069

$$y_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} \tag{4.5}$$

$$\theta_1, \theta_2 \in \mathbb{R}, \quad w_{k,k\in\mathbb{Z}} \sim \mathcal{N}(0,1) \tag{4.6}$$

The random variables $w_{k,k\in\mathbb{Z}} \sim \mathcal{N}(0,1)$ represent an independent and identically distributed white noise and θ_1, θ_2 the dependence from the previous observations. The number of consecutive observations T is a hyper-parameter of the model; in our case we will set $T = 100$. Generating an MA2 time-series is pretty easy and efficient using a simulator, therefore using a likelihood-free inference method quite convenient. At the specific example, we use the prior proposed by (Marin et al. 2012) for guaranteeing that the inference problem is identifiable, i.e. loosely speaking the likelihood will have just one mode. The multivariate prior, which is given in the equation (4.7), follows a triangular shape as plotted in figure 13.

$$p(\boldsymbol{\theta}) = p(\theta_1)p(\theta_2|\theta_1) = \mathcal{U}(\theta_1; -2, 2)\mathcal{U}(\theta_2; \theta_1 - 1, \theta_1 + 1) \tag{4.7}$$

The vector $\mathbf{y}_0 = (y_1, \dots, y_{100})$ used as the observation, has been generated with $\boldsymbol{\theta} = (0.6, 0.2)$. The

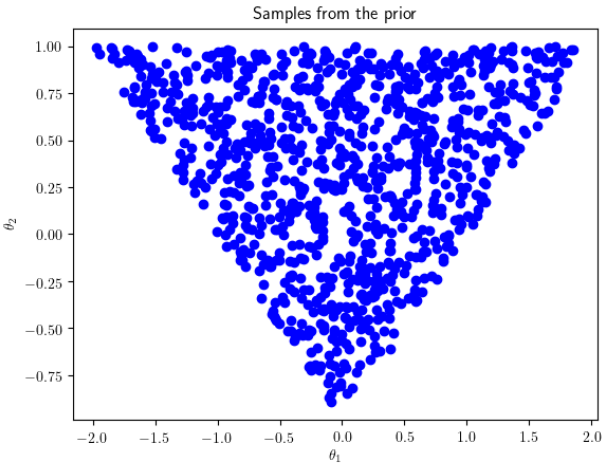


Figure 13: Prior distribution proposed by Marin et al. Marin et al. 2012. The samples follow a triangular shape.

dimensionality of the output $\mathbf{y}$ is quite large, therefore we use summary statistics. Considering that the ouptput vector represents a time-series signal, we prefer the autocovariance as the summary statistic; we incorporate the autocovariances with $lag = 1$ and $lag = 2$, as shown in equation (4.8). Hence, the distance is defined as the squared euclidean distance between the summary statistics.

$$s_1(\mathbf{y}) = \frac{1}{T-1}\sum_{t=2}^{T} y_t y_{t-1} \tag{4.8}$$

$$s_2(\mathbf{y}) = \frac{1}{T-2}\sum_{t=3}^{T} y_t y_{t-2} \tag{4.9}$$

$$s(\mathbf{y}) = (s_1(\mathbf{y}), s_2(\mathbf{y})) \tag{4.10}$$

$$d = ||s(\mathbf{y}) - s(\mathbf{y}_0)||_2^2 \tag{4.11}$$

Perform the inference

As in the previous example, we perform the inference using the two optimisation alternatives, the gradient-based and the Bayesian optimisation. In this way, we compare the results obtained in each step. For comparing the samples drawn with our implementation, we will use the samples obtained with Rejection ABC. In figure 14 we observe that in most cases the optimal distance d_i^* is close to zero in both optimisation approaches.

In figure 15, we have chosen three different optimisation examples for illustrating three different cases. In the first case, both optimisation schemes lead to the creation of almost the same bounding box. In the second case, the bounding box has a similar shape, though different size and it is shifted along the θ_1 axis. Finally, in the third case, the bounding boxes are completely different. We can thus conclude that although fitting a surrogate model has important computational advantages, there is no guarantee that it will reproduce accurately the local region around the optimal point. This approximation error may lead to the construction of a considerably different proposal region, which in turn, explains the differences in the histogram of the marginal distributions presented in figure 16 and in the approximate posteriors in figure 17.

In figure 16, we observe the histograms of the marginal posteriors; undoubtedly, we observe that there is a significant similarity between the three approaches. The Rejection ABC inference has been set to infer 10000 accepted samples, with threshold $\epsilon = 0.1$. The large number of samples and the small distance from the observations let treat them as ground-truth information. In the table 4.2 we present the empirical mean μ and standard deviation σ for each inference approach. We observe that there is a significant agreement between the approaches, which verifies that the ROMC implementation provides sensible results. Finally, in figure 17 we provide the ROMC approximate posteriors using gradients and Bayesian optimisation; as confirmed by the statistics in 4.2, both posteriors have a single mode, located at the same point, with a larger variance observed in the Bayesian Optimisation case. The posteriors mode is quite close to the parameter configuration that generated the data i.e. $\boldsymbol{\theta} = (0.6, 0.2)$.

	μ_{θ_1}	σ_{θ_1}	μ_{θ_2}	σ_{θ_2}
Rejection ABC	0.516	0.142	0.07	0.172
ROMC (gradient-based)	0.495	0.136	0.048	0.178
ROMC (Bayesian optimisation)	0.510	0.156	0.108	0.183

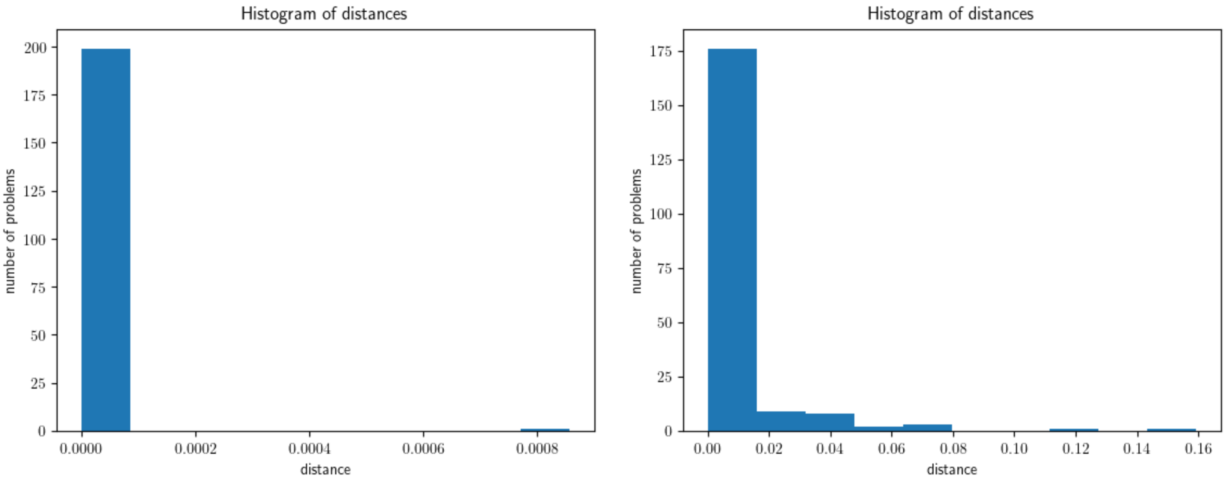


Figure 14: Histogram of distances $d^*_{i,i\in 1,\ldots,n_1}$. The left graph corresponds to the gradient-based approach and the right one to the Bayesian optimisation approach.

4.3 Execution Time Experiments

In this section, we will present the execution time of the basic ROMC functionalities. Apart from performing the inference accurately, one of the notable advantages of ROMC is its efficiency. In terms of performance, ROMC holds two key advantages. Firstly, all its subparts are ridiculously parallelisable; optimising the objective functions, constructing the bounding boxes, fitting the local surrogate models, sampling and evaluating the posterior can be executed in a fully-parallel fashion. Therefore, the speed-up can be extended as much as the computational resources allow. Specifically, at the CPU level, the parallelisation can be incorporate all available cores. A similar design can be utilised in the case of having a cluster of PCs available. Parallelising the process at the GPU level is not that trivial, since the parallel tasks are more complicated than simple floating-point operations. The second significant advantage concerns the execution of the training and the inference phase in distinct timeslots. Therefore, one can consume a lot of training time and resources but ask for accelerated inference. The use of a lightweight surrogate model around the optimal point exploits this essential characteristic; it trades off some additional computational burden at the training phase, for exercising faster inference later. Since our current implementation does not support parallel execution so-far[15], in figures 18, 19, 20, 21 we can only observe the second advantage. The example measured in these figures is the simple 1D example, used in the previous chapter. We observe that fitting local surrogate models slows-down the training phase (estimating the regions) at a factor of $x1.5$, but provides a speed-up of $x15$ at the inference phase (i.e. approximating unnormalised posterior). This outcome would be even more potent in larger models, where running the simulator is even more expensive.

[15] The design of our code permits adding this feature in a future update.

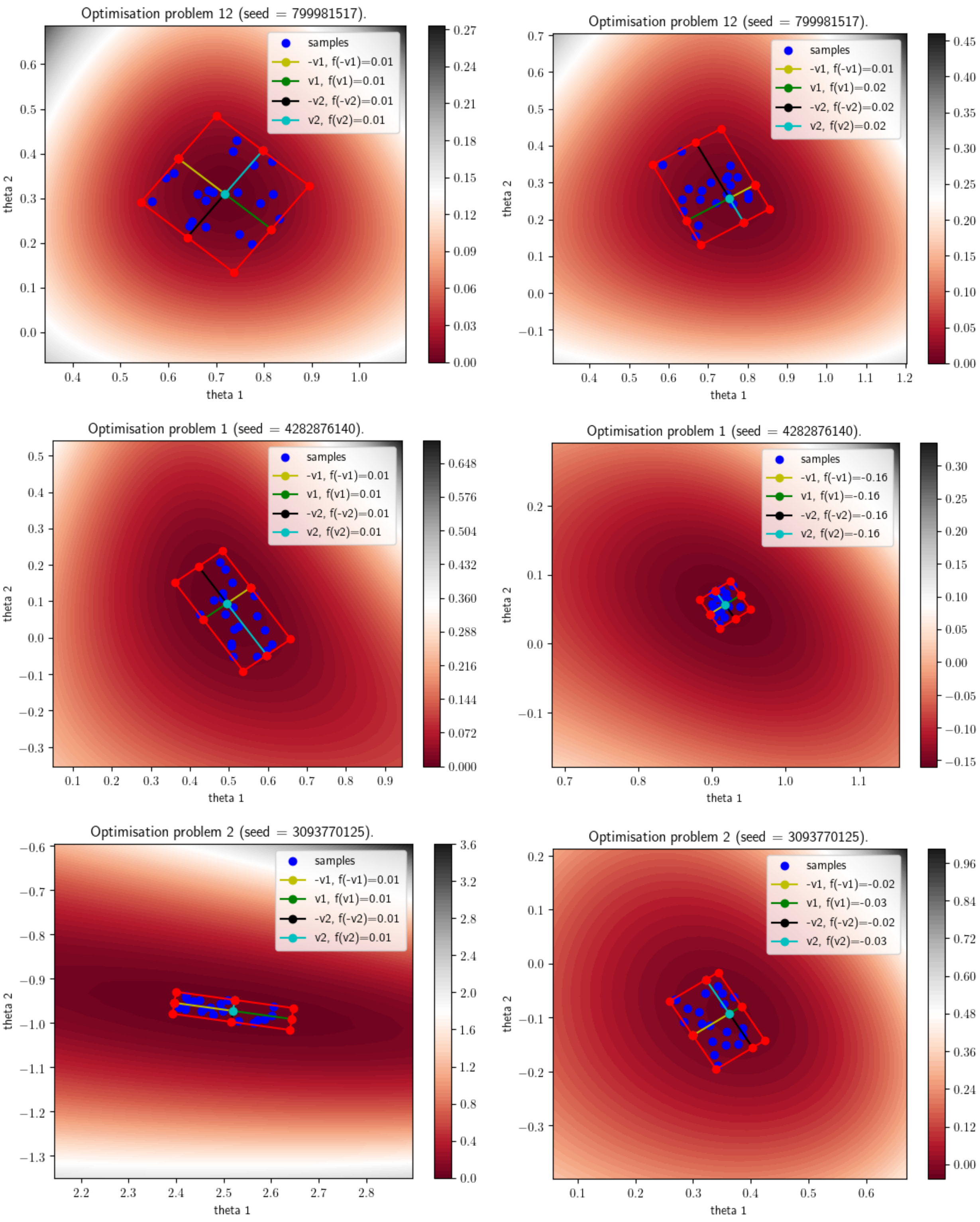


Figure 15: Visualisation of the acceptance region in 3 different optimisation problems. Each row illustrates a different optimisation problem, the left column corresponds to the gradient-based approach and the right column to the Bayesian optimisation approach. The examples have been chosen to illustrate three different cases; in the first case, both optimisation schemes lead to similar optimal point and bounding box, in the second case the bounding box is similar in shape but a little bit shifted to the right relatively to the gradient-based approach and in the third case, both the optimal point and the bounding box is completely different.

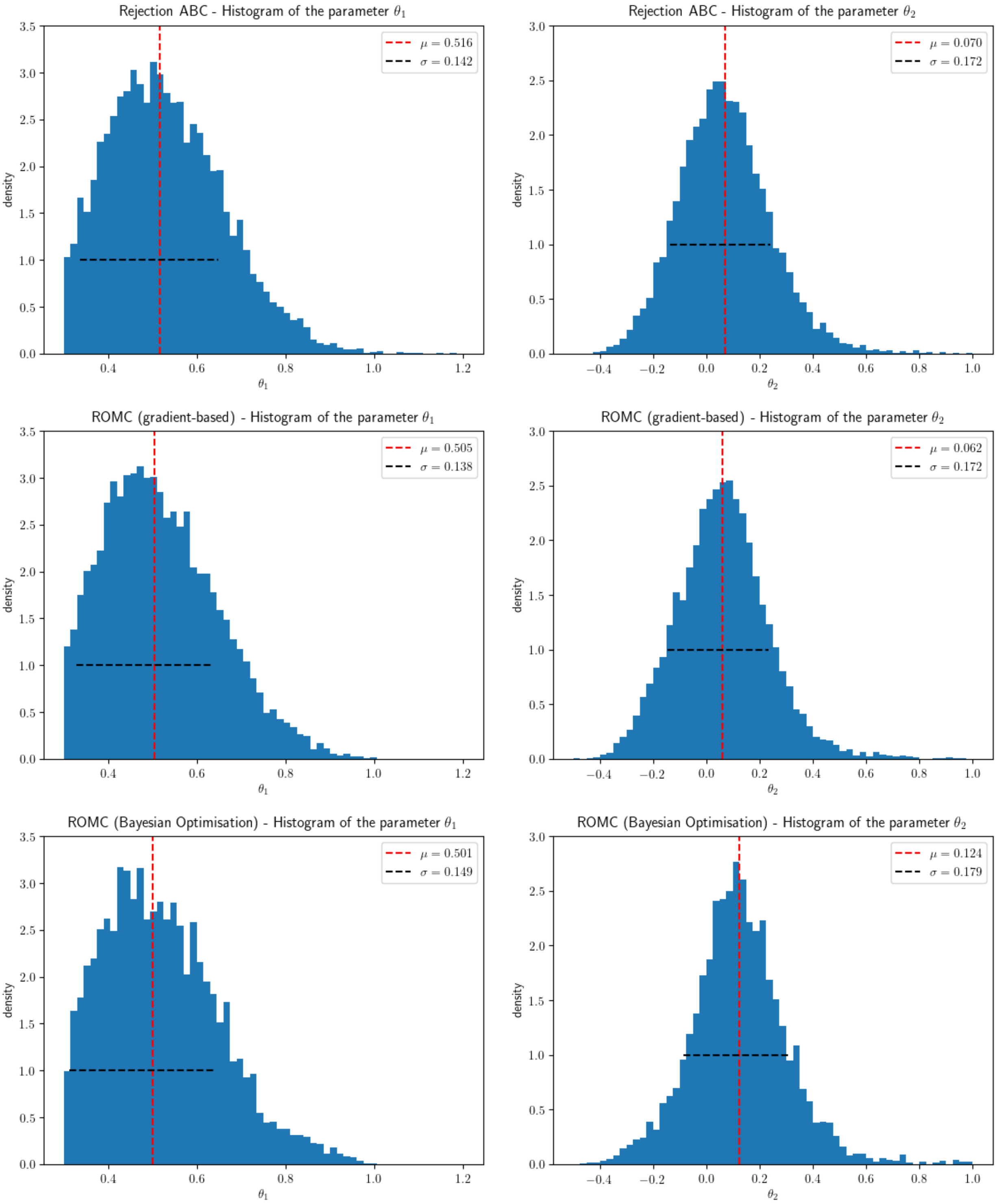


Figure 16: Histogram of the marginal distribution for three different inference approaches; (a) in the first row, the approximate posterior samples are obtained using Rejection ABC sampling (b) in the second row, using ROMC sampling with gradient-based approach and (c) in the third row, using ROMC sampling with Bayesian optimisation approach. The vertical (red) line represents the samples mean μ and the horizontal (black) the standard deviation σ.

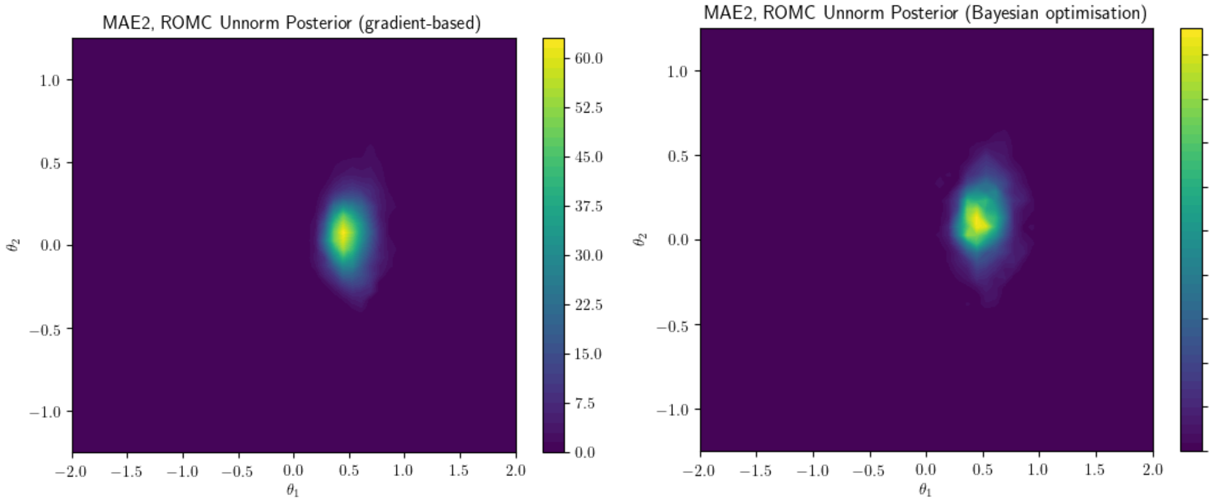


Figure 17: ROMC approximate posteriors using gradient-based approach (left) and Bayesian optimisation approach (right).

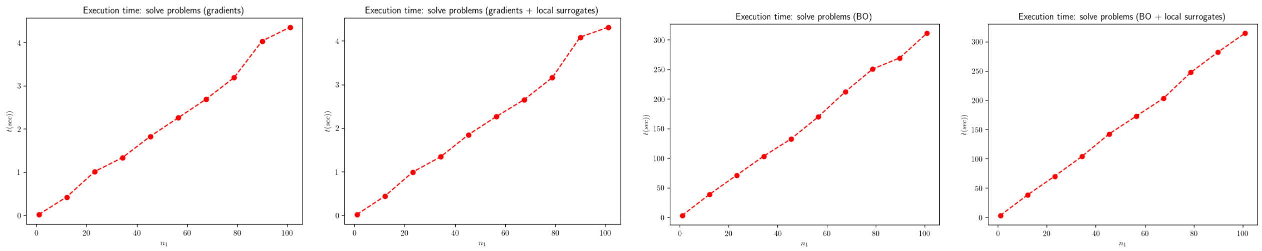


Figure 18: Execution time for defining and solving the optimisation problems. We observe a duration increase of $x75$ when switching to Bayesian optimisation scheme.

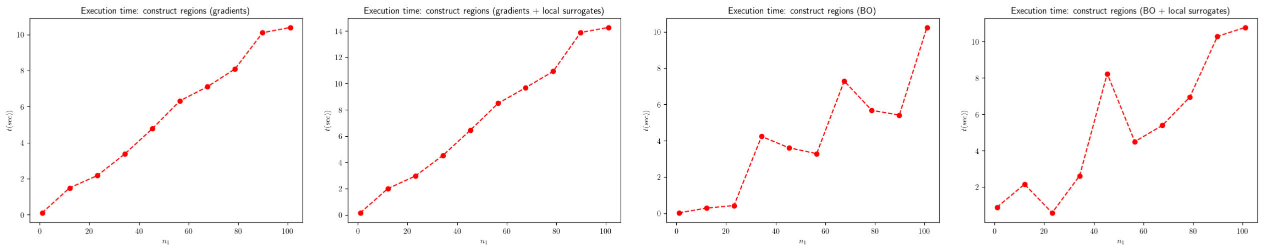


Figure 19: Execution time for constructing the n-dimensional bounding box region and, optionally, fitting the local surrogate models. We observe that fitting the surrogate models incurs a small delay of about $x1.5$.

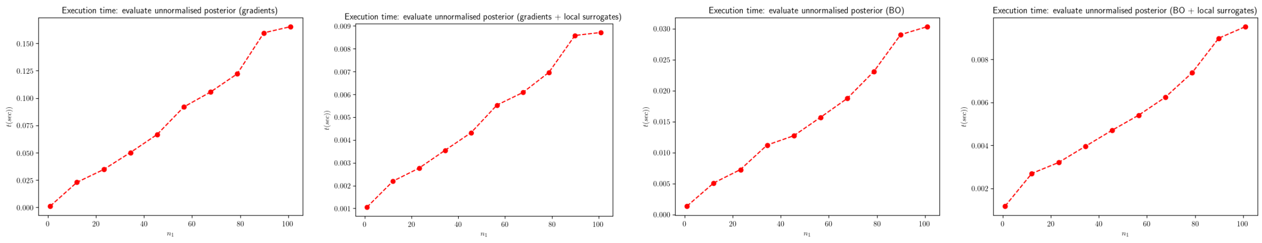


Figure 20: Execution time for evaluating the unnormalised posterior approximation. We observe that there is a major speed-up in the models with fitted local surogate models, of about $x15$.

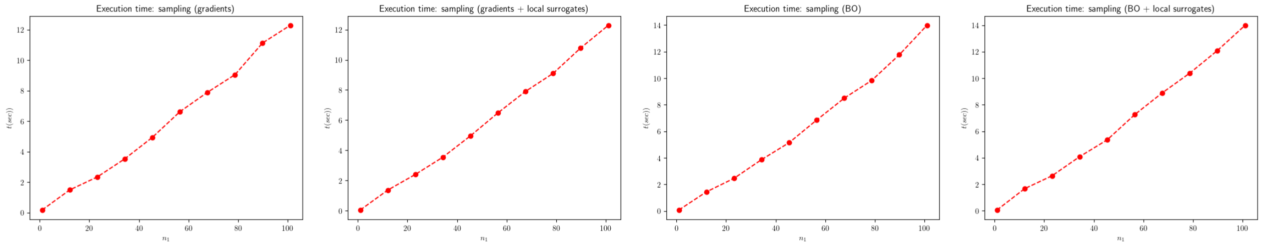


Figure 21: Execution time for sampling from the approximate posterior. We observe a small speed-up when the local surogate models are fitted, of about $x1.5$.

5 Conclusion

This chapter concludes the dissertation. We summarise the main contributions of our work, both in the theoretical and the implementation side. Furthermore, based on our observation throughout the implementation of the ROMC method, we suggest some ideas for future research.

5.1 Outcomes

In this dissertation, we have studied the likelihood-free inference approaches, focusing on the novel ROMC method that we implemented in an open-source software package. The main contribution of the dissertation is the implementation mentioned above which can be used mainly from the research community for further experimentation.

In chapter 2, we presented the simulator-based models explaining the particularities of the inference when the likelihood is not tractable. We then presented an overview of the Optimisation Monte-Carlo methods (OMC and ROMC) examining their strategies on approximating the posterior. We discussed the point-of-view of the ROMC approach, demonstrating the mathematic modelling it introduces. We also presented the aspect of ROMC as a meta-algorithm. Finally, at the end of chapter 2, we transformed the mathematical modelling of ROMC to algorithmic form. Up to this point, the dissertation mainly restates the ideas presented in the original paper ... and, hence, it can be used supplementarily with the paper by a reader who wants to understand the ROMC approach.

The notable contribution of the dissertation is the implementation of the method in the ELFI package. Due to the novelty of the method, it has not been implemented in any other package so far. We tried to implement the method focusing on four principles; simplicity, accuracy, efficiency and extensibility. We tried to provide the user with a simple-to-use method. Therefore, we followed the guidelines of the ELFI package, aligning ROMC with the rest inference methods. We also kept the function calls simple, asking only for the necessary arguments. Secondly, we tested the implementation in a range of examples for ensuring accurate inference results. We designed some artificial examples for evaluating our implementation using ground-truth information. We also tested that it works smoothly under general models, not artificially created by us. Thirdly, we tried to solve the tasks efficiently. We avoided using redundant calls, we exploited vectorisation when possible, and we avoided expensive for-loops. We measured the execution time of the method for providing an overview of the time needed for performing each task. Finally, we preserved extensibility; this was a significant priority in the implementation design. Apart from offering a ready-to-use inference method, we want our implementation to serve as the initial point for researchers who would like to further experiment with the method. This requirement aligns with the nature of ROMC as a meta-algorithm; one can replace the method involved in a specific task without the rest of the algorithm to collapse. Finally, we provide extensive documentation of all implemented methods and a collection of examples for illustrating the main use-cases. We wanted the reader to be able to observe and interact in-practice with the functionalities we offer.

5.2 Future Research Directions

Future research directions involve both the theoretical and implementation aspect of the ROMC approach.

In the theoretical part, an essential drawback of the ROMC approach is the creation of a single bounding box for each objective function. This simplification can be error-prone at situations where we have ignorance about the properties of the objective function. Consider, for example, the scenario where an objective function defines multiple tiny disjoint acceptance areas; the optimiser will (hopefully) reach only one, and a bounding box will be constructed around it, compressing all the mass of the distribution there. Therefore, this small region will erroneously dominate the specific objective function. In general, the existence of multiple disjoint acceptance regions complicates the inference. A naive approach for improving this failure would be solving each optimisation problem more than once, using different starting points. With this approach, we hope that the optimiser will reach a different local minimum in each execution. However, apart from the added computational complexity, this approach does not guarantee an improvement. In general, identifying all the acceptance regions in an unconstrained problem with many local optimums is quite challenging.

Further research on the optimisation part, may also focus on modelling the case where the parametric space is constrained. Consider, for example, the case where a parameter represents a probability (as in the tuberculosis example). In this case, the parameter is constrained in the region $[0, 1]$. Defining a prior with zero mass outside of this area ensures that all non-acceptable values will not contribute to the posterior. However, the optimiser does not consider the prior. Hence, we can observe the phenomenon where the majority of the (or all) acceptance regions is cancelled-out by the prior. In this case, the posterior will be dominated by a few acceptance region parts that overlap with the prior, which is not robust. Dealing with such constrained non-linear optimisation problems, which is particularly challenging, can serve as a second research line.

The previous two research directions overlap with the field of optimisation. Another research avenue, more concentrated on the ROMC method, is the approximation of the acceptance regions with a more complex geometrical shape than the bounding box. In the current implementation,

the bounding box only serves as the proposal region for sampling; running the indicator function afterwards, we manage to discard the samples that are above the distance threshold. This operation is quite expensive from a computational point of view. The workaround proposed from the current implementation is fitting a local surrogate model for avoiding running the expensive simulator. However, fitting a surrogate model for every objective function is still expensive. Approximating the acceptance region accurately with a more complicated geometrical structure could remove the burden of running the indicator function later. For example, a simple idea would be approximating the acceptance region with a set of non-overlapping bounding boxes. We insist on using the square as the basic shape because we can efficiently sample from a uniform distribution defined on it. A future research direction could investigate this capability.

On the implementation side, the improvements are more concrete. A major upgrade would be supporting the parallel execution of the training and inference phase. As mentioned in chapter 4, the ROMC approach can be executed in a completely parallel manner, as each optimisation problem does not interact with the rest. Another critical improvement would be performing the training and the inference phase in a batched fashion. For example, the practitioner may indicate that he would like to approximate the posterior in batches of N optimisation problems; this means that the approximation should be updated after each N new bounding boxes are constructed. Utilising this feature, the user may interactively decide when he would like to terminate the process. Finally, the implementation could also support saving the state of the training or the inference process for continuing later. In the current implementation, although the user can save the inference result, he cannot interrupt the training phase and continue latter.

References

[CG19] Yanzhi Chen and Michael U Gutmann. "Adaptive Gaussian Copula ABC". In: *Proceedings of Machine Learning Research*. Vol. 89. 2019, pp. 1584–1592. URL: http://proceedings.mlr.press/v89/chen19d.html.

[GC16] Michael U. Gutmann and Jukka Corander. *Bayesian optimization for likelihood-free inference of simulator-based statistical models*. 2016. arXiv: 1501.03291.

[GPync] GPy. *GPy: A Gaussian process framework in python*. http://github.com/SheffieldML/GPy. since 2012.

[IG19] Borislav Ikonomov and Michael U. Gutmann. "Robust Optimisation Monte Carlo". In: (2019). arXiv: 1904.00670. URL: http://arxiv.org/abs/1904.00670.

[Lin+17] Jarno Lintusaari et al. "Fundamentals and recent developments in approximate Bayesian computation". In: *Systematic Biology* 66.1 (2017), e66–e82. ISSN: 1076836X. DOI: 10.1093/sysbio/syw077.

[Lin+18] Jarno Lintusaari et al. *ELFI: Engine for Likelihood Free Inference*. 2018. eprint: arXiv:1708.00707.

[Mar+12] Jean Michel Marin et al. "Approximate Bayesian computational methods". In: *Statistics and Computing* (2012). ISSN: 09603174. DOI: 10.1007/s11222-011-9288-2. arXiv: 1101.0955.

[MW15] Edward Meeds and Max Welling. "Optimization Monte Carlo: Efficient and embarrassingly parallel likelihood-free inference". In: *Advances in Neural Information Processing Systems*. 2015. arXiv: 1506.03693.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Sha+16] Bobak Shahriari et al. *Taking the human out of the loop: A review of Bayesian optimization*. 2016. DOI: 10.1109/JPROC.2015.2494218.

[Tan+06] Mark M. Tanaka et al. "Using approximate bayesian computation to estimate tuberculosis transmission parameters from genotype data". In: *Genetics* (2006). ISSN: 00166731. DOI: 10.1534/genetics.106.055574.

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.