An Extendable **Python** Implementation of Robust Optimisation Monte Carlo

Vasilis Gkolemis
ATHENA RC

Michael Gutmann
University of Edinburgh

Henri Pesonen
University of Oslo

Abstract

Performing inference in statistical models with an intractable likelihood is challenging, therefore, most likelihood-free inference (LFI) methods encounter accuracy and efficiency limitations. In this paper, we present the implementation of the LFI method Robust Optimisation Monte Carlo (ROMC) in the Python package **ELFI**. ROMC is a novel and efficient (highly-parallelizable) LFI framework that provides accurate weighted samples from the posterior. Our implementation can be used in two ways. First, a scientist may use it as an out-of-the-box LFI algorithm; we provide an easy-to-use API harmonized with the principles of **ELFI**, enabling effortless comparisons with the rest of the methods included in the package. Additionally, we have carefully split ROMC into isolated components for supporting extensibility. A researcher may experiment with novel method(s) for solving part(s) of ROMC without reimplementing everything from scratch. In both scenarios, the ROMC parts can run in a fully-parallelized manner, exploiting all CPU cores. We also provide helpful functionalities for (i) inspecting the inference process and (ii) evaluating the obtained samples. Finally, we test the robustness of our implementation on some typical LFI examples.

Keywords: Bayesian inference, implicit models, likelihood-free, Python, **ELFI**.

1. Introduction

Simulator-based models are particularly captivating due to the modeling freedom they provide. In essence, any data generating mechanism that can be written as a finite set of algorithmic steps can be programmed as a simulator-based model. Hence, these models are often used to model physical phenomena in the natural sciences such as, e.g., genetics, epidemiology or neuroscience Gutmann and Corander (2016); Lintusaari *et al.* (2017); Sisson *et al.* (2018); Cranmer *et al.* (2020). In simulator-based models, it is feasible to generate samples using the simulator but is infeasible to evaluate the likelihood function. The intractability of the

likelihood makes the so-called likelihood-free inference (LFI), i.e., the approximation of the posterior distribution without using the likelihood function, particularly challenging.

Optimization Monte Carlo (OMC), proposed by Meeds and Welling (2015), is a novel LFI approach. The central idea is to convert the stochastic data-generating mechanism into a set of deterministic optimization processes. Afterwards, Forneron and Ng (2016) described a similar method under the name 'reverse sampler'. In their work, Ikonomov and Gutmann (2019) identified some critical limitations of OMC, so they proposed Robust OMC (ROMC) an improved version of OMC with appropriate modifications.

In this paper, we present the implementation of ROMC at the Python package **ELFI** (**Engine for likelihood-free inference**) Lintusaari *et al.* (2018). The implementation has been designed to facilitate extensibility. ROMC is an LFI framework; it defines a sequence of algorithmic steps for approximating the posterior without enforcing a specific algorithm for each step. Therefore, a researcher may use ROMC as the backbone method and apply novel algorithms to each separate step. For being a ready-to-use LFI method, Ikonomov and Gutmann (2019) propose a particular (default) algorithm for each step, but this choice is by no means restrictive. We have designed our software for facilitating such experimentation.

To the best of our knowledge, this is the first implementation of the ROMC inference method to a generic LFI framework. We organize the illustration and the evaluation of our implementation in three steps. First, for securing that our implementation is accurate, we test it against an artificial example with a tractable likelihood. The artificial example also serves as a step-by-step guide for showcasing the various functionalities of our implementation. Second, we use the second-order moving average (MA2) example Marin *et al.* (2012) from the **ELFI** package, using as ground truth the samples obtained with Rejection ABC Lintusaari *et al.* (2017) using a very high number of samples. Finally, we present the execution times of ROMC, measuring the speed-up achieved by the parallel version of the implementation.

The code of the implementation is available at the official **ELFI** repository. Apart from the examples presented in the paper, there are five **Google Colab** Bisong (2019) notebooks available online, with end-to-end examples illustrating how to: (i) use ROMC on a synthetic $1D$ example, (ii) use ROMC on a synthetic $2D$ example, (iii) use ROMC on the Moving Average example, (iv) extend ROMC with a Neural Network as a surrogate model, (v) extend ROMC with a custom proposal region module.

2. Background

We first give a short introduction to simulator-based models, we then focus on OMC and its robust version, ROMC, and we, finally, introduce **ELFI**, the Python package used for the implementation.

2.1. Simulator-based models and likelihood-free inference

An implicit or simulator-based model is a parameterized stochastic data generating mechanism, where we can sample data points but we cannot evaluate the likelihood. Formally, a simulator-based model is a parameterized family of probability density functions $\{p(\mathbf{y}|\boldsymbol{\theta})\}_{\boldsymbol{\theta}}$ whose closed-form is either unknown or computationally intractable. In these cases, we can only access the simulator $m_r(\boldsymbol{\theta})$, i.e., the black-box mechanism (computer code) that generates samples $\mathbf{y}$ in a stochastic manner from a set of parameters $\boldsymbol{\theta} \in \mathbb{R}^D$. We denote the

process of obtaining samples from the simulator with $m_r(\boldsymbol{\theta}) \to \mathbf{y}$. As shown by Meeds and Welling (2015), it is feasible to isolate the randomness of the simulator by introducing a set of nuisance random variables denoted by $\mathbf{u} \sim p(\mathbf{u})$. Therefore, for a specific tuple $(\boldsymbol{\theta}, \mathbf{u})$ the simulator becomes a deterministic mapping g, such that $\mathbf{y} = g(\boldsymbol{\theta}, \mathbf{u})$. In terms of computer code, the randomness of a random process is governed by the global seed. There are some differences on how each scientific package handles the randomness; for example, at **Numpy** Harris *et al.* (2020) the pseudo-random number generation is based on a global state, whereas, at **JAX** Bradbury *et al.* (2018) random functions consume a key that is passed as a parameter. However, in all cases, setting the initial seed to a specific integer converts the simulation to a deterministic piece of code.

The modeling freedom of simulator-based models comes at the price of difficulties in inferring the parameters of interest. Denoting the observed data as $\mathbf{y_0}$, the main difficulty lies at the intractability of the likelihood function $L(\boldsymbol{\theta}) = p(\mathbf{y_0}|\boldsymbol{\theta})$. To better see the sources of the intractability, and to address them, we go back to the basic characterization of the likelihood as the (rescaled) probability of generating data $\mathbf{y}$ that is similar to the observed data $\mathbf{y_0}$, using parameters $\boldsymbol{\theta}$. Formally, the likelihood $L(\boldsymbol{\theta})$ is:

$$L(\boldsymbol{\theta}) = \lim_{\epsilon \to 0} c_\epsilon \int_{\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0})} p(\mathbf{y}|\boldsymbol{\theta})d\mathbf{y} = \lim_{\epsilon \to 0} c_\epsilon \Pr(g(\boldsymbol{\theta}, \mathbf{u}) \in B_{d,\epsilon}(\mathbf{y_0}) \mid \boldsymbol{\theta}) \tag{1}$$

where c_ϵ is a proportionality factor that depends on ϵ and $B_{d,\epsilon}(\mathbf{y_0})$ is an ϵ region around $\mathbf{y_0}$ that is defined through a distance function d, i.e., $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(\mathbf{y}, \mathbf{y_0}) \leq \epsilon\}$. In cases where the output $\mathbf{y}$ belongs to a high dimensional space, it is common to extract summary statistics Φ before applying the distance d. In these cases, the ϵ-region is defined as $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(\Phi(\mathbf{y}), \Phi(\mathbf{y_0})) \leq \epsilon\}$. In our notation, the summary statistics are sometimes omitted for simplicity. Equation 1 highlights the main source of intractability; computing $\Pr(g(\boldsymbol{\theta}, \mathbf{u}) \in B_{d,\epsilon}(\mathbf{y_0})|\boldsymbol{\theta})$ as the fraction of samples that lie inside the ϵ region around $\mathbf{y_0}$ is computationally infeasible in the limit where $\epsilon \to 0$. Hence, the constraint is relaxed to $\epsilon > 0$, which leads to the approximate likelihood:

$$L_{d,\epsilon}(\boldsymbol{\theta}) = \Pr(\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0}) \mid \boldsymbol{\theta}), \quad \text{where } \epsilon > 0. \tag{2}$$

and, in turn, to the approximate posterior:

$$p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0}) \propto L_{d,\epsilon}(\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3}$$

Equations 2 and 3 is by no means the only strategy to deal with the intractability of the likelihood function in Equation 1. Other strategies include modeling the (stochastic) relationship between $\boldsymbol{\theta}$ and $\mathbf{y}$, and its reverse, or framing likelihood-free inference as a ratio estimation problem, see for example Blum and François (2010); Wood (2006); Papamakarios and Murray (2016); Papamakarios *et al.* (2019); Chen and Gutmann (2019); Thomas *et al.* (2022); Hermans *et al.* (2020). However, both OMC and robust OMC, which we introduce next, are based on the approximation in Equation 2.

2.2. Optimization Monte Carlo (OMC)

Our description of OMC Meeds and Welling (2015) follows Ikonomov and Gutmann (2019).

We define the indicator function (boxcar kernel) that equals one only if $\mathbf{x}$ lies in $B_{d,\epsilon}(\mathbf{y})$:

$$\mathbb{1}_{B_{d,\epsilon}(\mathbf{y})}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in B_{d,\epsilon}(\mathbf{y}) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

We, then, rewrite the approximate likelihood function $L_{d,\epsilon}(\boldsymbol{\theta})$ of Equation 2 as:

$$L_{d,\epsilon}(\boldsymbol{\theta}) = \Pr(\mathbf{y} \in B_{d,\epsilon}(\mathbf{y_0})|\boldsymbol{\theta}) = \int_{\mathbf{u}} \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(g(\boldsymbol{\theta},\mathbf{u}))d\mathbf{u} \tag{5}$$

which can be approximated using samples from the simulator:

$$L_{d,\epsilon}(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{B_{d,\epsilon}(\mathbf{y_0})}(g(\boldsymbol{\theta},\mathbf{u}_i)) \quad \text{where } \mathbf{u}_i \sim p(\mathbf{u}). \tag{6}$$

In Equation 6, for each $\mathbf{u}_i$, there is a region C_ϵ^i in the parameter space $\boldsymbol{\theta}$ where the indicator function returns one, i.e., $C_\epsilon^i = \{\boldsymbol{\theta} : g(\boldsymbol{\theta},\mathbf{u}_i) \in B_{d,\epsilon}(\mathbf{y_0})\}$. Therefore, we can rewrite the approximate likelihood and posterior as:

$$L_{d,\epsilon}(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{C_\epsilon^i}(\boldsymbol{\theta}) \tag{7}$$

$$p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0}) \propto p(\boldsymbol{\theta})\sum_{i}^{N} \mathbb{1}_{C_\epsilon^i}(\boldsymbol{\theta}). \tag{8}$$

As argued by Ikonomov and Gutmann (2019), these derivations provide a unique perspective for likelihood-free inference by shifting the focus onto the geometry of the acceptance regions C_ϵ^i. Indeed, the task of approximating the likelihood and the posterior becomes a task of characterizing the sets C_ϵ^i. OMC by Meeds and Welling (2015) assumes that the distance d is the Euclidean distance $||\cdot||_2$ between summary statistics Φ of the observed and generated data, and that the C_ϵ^i can be well approximated by infinitesimally small ellipses. These assumptions lead to an approximation of the posterior in terms of weighted samples $\boldsymbol{\theta}_i^*$ that achieve the smallest distance between observed and simulated data for each realization $\mathbf{u}_i \sim p(\mathbf{u})$, i.e.,

$$\boldsymbol{\theta}_i^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} ||\Phi(\mathbf{y_0}) - \Phi(g(\boldsymbol{\theta},\mathbf{u}_i))||_2, \quad \mathbf{u}_i \sim p(\mathbf{u}). \tag{9}$$

The weighting for each $\boldsymbol{\theta}_i^*$ is proportional to the prior density at $\boldsymbol{\theta}_i^*$ and inversely proportional to the determinant of the Jacobian matrix of the summary statistics at $\boldsymbol{\theta}_i^*$. For further details on OMC we refer the reader to Meeds and Welling (2015); Ikonomov and Gutmann (2019).

2.3. Robust optimization Monte Carlo (ROMC)

Ikonomov and Gutmann (2019) showed that considering infinitesimally small ellipses can lead to highly overconfident posteriors. We refer the reader to their paper for the technical details and conditions for this issue to occur. Intuitively, it happens because the weights in OMC are only computed from information at $\boldsymbol{\theta}_i^*$, and using only local information can be misleading.

For example, if the curvature of $||\Phi(\mathbf{y_0}) - \Phi(g(\boldsymbol{\theta}, \mathbf{u}_i))||_2$ at $\boldsymbol{\theta}_i^*$ is nearly flat, it may wrongly indicate that C_ϵ^i is much larger than it actually is. In our software package we implement the robust generalization of OMC by Ikonomov and Gutmann (2019) that resolves this issue.

ROMC approximates the acceptance regions C_ϵ^i with D-dimensional bounding boxes $\hat{C}_\epsilon^i$. A uniform distribution, $q_i(\boldsymbol{\theta})$, is defined on top of each bounding box and serves as a proposal distribution for generating posterior samples $\boldsymbol{\theta}_{ij} \sim q_i$. The samples get an (importance) weight w_{ij} that compensate for using the proposal distributions $q_i(\boldsymbol{\theta})$ instead of the prior $p(\boldsymbol{\theta})$:

$$w_{ij} = \mathbb{1}_{C_\epsilon^i}(\boldsymbol{\theta}_{ij})\frac{p(\boldsymbol{\theta}_{ij})}{q(\boldsymbol{\theta}_{ij})}. \tag{10}$$

Given the weighted samples, any expectation $\mathsf{E}_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$ of some function $h(\boldsymbol{\theta})$, can be approximated as

$$\mathsf{E}_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})] \approx \frac{\sum_{ij} w_{ij} h(\boldsymbol{\theta}_{ij})}{\sum_{ij} w_{ij}} \tag{11}$$

The approximation of the acceptance regions contains two compulsory and one optional step: (i) solving the optimization problems as in OMC, (ii) constructing bounding boxes around C_ϵ^i and, optionally, (iii) refining the approximation via a surrogate model of the distance.

(i) Solving the deterministic optimization problems

For each set of nuisance variables $\mathbf{u}_i, i = \{1, 2, \ldots, n_1\}$, we search for a point $\boldsymbol{\theta}_i^*$ such that $d(g(\boldsymbol{\theta}_i^*, \mathbf{u}_i), \mathbf{y_0}) \leq \epsilon$. In principle, $d(\cdot)$ can refer to any valid distance function. For the rest of the paper we consider $d(\cdot)$ as the squared Euclidean distance, as in Ikonomov and Gutmann (2019). For simplicity, we use $d_i(\boldsymbol{\theta})$ to refer to $d(g(\boldsymbol{\theta}, \mathbf{u}_i), \mathbf{y_0})$. We search for θ_i^* solving:

$$\boldsymbol{\theta}_i^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, d_i(\boldsymbol{\theta}) \tag{12}$$

and we accept the solution only if it satisfies the constraint $d_i(\boldsymbol{\theta}_i^*) \leq \epsilon$. If $d_i(\boldsymbol{\theta})$ is differentiable, Equation 12 can be solved using any gradient-based optimizer. The gradients $\nabla_{\boldsymbol{\theta}} d_i(\boldsymbol{\theta})$ can be either provided in closed form or approximated by finite differences. If d_i is not differentiable, Bayesian Optimization (Shahriari *et al.* 2015) can be used instead. In this scenario, apart from obtaining an optimal $\boldsymbol{\theta}_i^*$, we can also automatically build a surrogate model $\hat{d}_i(\boldsymbol{\theta})$ of the distance function $d_i(\boldsymbol{\theta})$. The surrogate model $\hat{d}_i$ can then substitute the actual distance function in downstream steps of the algorithms, with possible computational gains especially in cases where evaluating the actual distance $d_i(\boldsymbol{\theta})$ is expensive.

(ii) Estimating the acceptance regions

Each acceptance region C_ϵ^i is approximated by a bounding box $\hat{C}_\epsilon^i$. The acceptance regions C_ϵ^i can contain any number of disjoint subsets in the D-dimensional space and any of these subsets can take any arbitrary shape. We should make three important remarks. First, since the bounding boxes are built around $\boldsymbol{\theta}_i^*$, we focus only on the connected subset of C_ϵ^i that contains $\boldsymbol{\theta}_i^*$, which we denote as $C_{\epsilon,\theta_i^*}^i$. Second, we want to ensure that the bounding box $\hat{C}_\epsilon^i$ is big enough to contain on its interior all the volume of $C_{\epsilon,\theta_i^*}^i$. Third, we want $\hat{C}_\epsilon^i$ to be as tight as possible to $C_{\epsilon,\theta_i^*}^i$ to ensure high acceptance rate on the importance sampling step that

follows. Therefore, the bounding boxes are built in two steps. First, we compute their axes $\mathbf{v}_m$, for $m = \{1, \ldots, D\}$ based on the (estimated) curvature of the distance at $\boldsymbol{\theta}_i^*$, and, second, we apply a line-search method along each axis to determine the size of the bounding box. We refer the reader to Algorithm 2 for the details. After the bounding boxes construction, a uniform distribution q_i is defined on each bounding box, and is used as the proposal region for importance sampling.

(iii) Refining the estimate via a local surrogate model (optional)

For computing the weight w_{ij} at Equation 10, we need to check whether the samples $\boldsymbol{\theta}_{ij}$, drawn from the bounding boxes, are inside the acceptance region C_ϵ^i. This can be considered to be a safety-mechanism that corrects for any inaccuracies in the construction of $\hat{C}_\epsilon^i$ above. However, this check involves evaluating the distance function $d_i(\boldsymbol{\theta}_{ij})$, which can be expensive if the model is complex. Ikonomov and Gutmann (2019) proposed fitting a surrogate model $\tilde{d}_i(\boldsymbol{\theta})$ of the distance function $d_i(\boldsymbol{\theta})$, on data points that lie inside $\hat{C}_\epsilon^i$. In principle, any regression model can be used as surrogate model. Ikonomov and Gutmann (2019) used a simple quadratic model because it has ellipsoidal isocontours, which facilitates replacing the bounding box approximation of C_ϵ^i with a tighter-fitting ellipsoidal approximation.

The training data for the quadratic model is obtained by sampling $\boldsymbol{\theta}_{ij} \sim q_i$ and accessing the distances $d_i(\boldsymbol{\theta}_{ij})$. The generation of the training data adds an extra computational cost, but leads to a significant speed-up when evaluating the weights w_{ij}. Moreover, the extra cost is largely eliminated if Bayesian optimization with a Gaussian process (GP) surrogate model $\hat{d}_i(\boldsymbol{\theta})$ was used to obtain $\boldsymbol{\theta}_i^*$ in the first step. In this case, we can use $\hat{d}_i(\boldsymbol{\theta})$ instead of $d_i(\boldsymbol{\theta})$ to generate the training data. This essentially replaces the global GP model with a simpler local quadratic model which is typically more robust.

2.4. Engine for likelihood-free inference (ELFI)

Engine for Likelihood-Free Inference (ELFI)[1] Lintusaari *et al.* (2018) is a Python package for LFI. We selected to implemented ROMC in **ELFI** since it provides convenient modules for all the fundamental components of a probabilistic model (e.g. prior, simulator, summaries etc.). Furthermore, **ELFI** already supports some recently proposed likelihood-free inference methods. **ELFI** handles the probabilistic model as a Directed Acyclic Graph (DAG). This functionality is based on the package **NetworkX** Hagberg *et al.* (2008), which supports general-purpose graphs. In most cases, the structure of a likelihood-free model follows the pattern of Figure 1; some edges connect the prior distributions to the simulator, the simulator is connected to the summary statistics that, in turn, lead to the output node. Samples can be obtained from all nodes through sequential (ancestral) sampling. **ELFI** automatically considers as parameters of interest, i.e., those we try to infer a posterior distribution, the ones included in the `elfi.Prior` class.

All inference methods of **ELFI** are implemented following two conventions. First, their constructor follows the signature `elfi.<Class name>(<output node>, *arg)`, where `<output node>` is the output node of the simulator-based model and `*arg` are the parameters of the method. Second, they provide a method `elfi.<Class name>.sample(*args)` for drawing samples from the approximate posterior.

---

[1]Extended documentation can be found https://elfi.readthedocs.io

```
# 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)
```
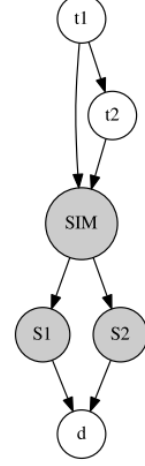
Figure 1: Baseline example for creating an **ELFI** model. Image taken from Lintusaari *et al.* (2018)

3. Overview of the implementation

In this section, we express ROMC as an algorithm and then we present the general implementation principles.

3.1. Algorithmic view of ROMC

For designing an extendable implementation, we firstly define ROMC as a sequence of algorithmic steps. At a high level, ROMC can be split into the training and the inference part; the training part covers the steps for estimating the proposal regions and the inference part calculates the weighted samples. In Algorithm 1, that defines ROMC as an algorithm, steps 2-11 (before the horizontal line) refer to the training part and steps 13-18 to the inference part.

Training part

At the training (fitting) part, the goal is the estimation of the proposal regions $\hat{C}_\epsilon^i$, which expands to three mandatory tasks; (a) sample the nuisance variables $\mathbf{u}_i \sim p(\mathbf{u})$ for defining the deterministic distance functions $d_i(\boldsymbol{\theta})$ (steps 3-5), (b) solve the optimization problems for obtaining $\boldsymbol{\theta}_i^*, d_i^*$ and keep the solutions inside the threshold ϵ (steps 6-9), and (c) estimate the bounding boxes $\hat{C}_\epsilon^i$ to define uniform distributions q_i on them (step 10). Optionally, a surrogate model $\tilde{d}_i$ can be fitted for a faster inference phase (step 11).

If $d_i(\boldsymbol{\theta})$ is differentiable, using a gradient-based method is advised for obtaining $\boldsymbol{\theta}_i^*$ faster. In this case, the gradients $\nabla_{\boldsymbol{\theta}} d_i$ gradients are approximated automatically with finite-differences, if they are not provided in closed-form by the user. Finite-differences approximation requires two evaluations of d_i for each parameter $\theta_m, m \in \{1, \ldots, D\}$, which scales well only in low-dimensional problems. If $d_i(\boldsymbol{\theta})$ is not differentiable, Bayesian Optimization can be used as an alternative. In this scenario, the training part becomes slower due to fitting of the surrogate

---

Algorithm 1 ROMC. Requires the prior $p(\boldsymbol{\theta})$, the simulator $M_r(\boldsymbol{\theta})$, number of optimization problems n_1, number of samples per region n_2, acceptance limit ϵ

 1: **procedure** ROMC
 2:     **for** $i \leftarrow 1$ to n_1 **do**
 3:         $\mathbf{u}_i \sim p(\mathbf{u})$ ▷ Draw nuisance variables
 4:         Convert $M_r(\boldsymbol{\theta})$ to $g(\boldsymbol{\theta}, \mathbf{u} = \mathbf{u}_i)$ ▷ Define deterministic simulator
 5:         $d_i(\boldsymbol{\theta}) = d(g(\boldsymbol{\theta}, \mathbf{u} = \mathbf{u}_i), \mathbf{y}_0)$ ▷ Define distance function
 6:         $\boldsymbol{\theta}_i^* = \operatorname{argmin}_{\boldsymbol{\theta}} d_i, d_i^* = d_i(\boldsymbol{\theta}_i^*)$ ▷ Solve optimization problem
 7:         **if** $d_i^* > \epsilon$ **then**
 8:             Go to 2   ▷ Filter solution
 9:         **end if**
10:         Estimate $\hat{C}_\epsilon^i$ and define q_i ▷ Estimate proposal area
11:         (Optional) Fit $\tilde{d}_i$ on $\hat{C}_\epsilon^i$ ▷ Fit surrogate model
12:

13:         **for** $j \leftarrow 1$ to n_2 **do**
14:             $\boldsymbol{\theta}_{ij} \sim q_i$, compute w_{ij} as in Algorithm 3   ▷ Sample
15:         **end for**
16:     **end for**
17:     $\mathsf{E}_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$ as in eq. (11)   ▷ Estimate an expectation
18:     $p_{d,\epsilon}(\boldsymbol{\theta})$ as in eq. (8)   ▷ Evaluate the unnormalized posterior
19: **end procedure**

---

model and the blind optimization steps.

After obtaining the optimal points $\boldsymbol{\theta}_i^*$, we estimate the proposal regions. Algorithm 2 describes the line search approach for finding the region boundaries. The axes of each bounding box $\mathbf{v}_m, m = \{1, \ldots, D\}$ are defined as the directions with the highest curvature at $\boldsymbol{\theta}_i^*$ computed by the eigenvalues of the Hessian matrix $\mathbf{H}_i$ of d_i at $\boldsymbol{\theta}_i$ (step 1). Depending on the algorithm used in the optimization step, we either use the real distance d_i or the Gaussian Process approximation $\hat{d}_i$. When the distance function is the Euclidean distance (default choice), the Hessian matrix can be also computed as $\mathbf{H}_i = \mathbf{J}_i^T \mathbf{J}_i$, where $\mathbf{J}_i$ is the Jacobian matrix of the summary statistics $\Phi(g(\boldsymbol{\theta}, \mathbf{u}_i))$ at $\boldsymbol{\theta}_i^*$. This approximation has the computational advantage of using only first-order derivatives. After defining the axes, we search for the bounding box limits with a line step algorithm (steps 2-21). The key idea is to take long steps η_start until crossing the boundary and then take small steps backwards to find the exact boundary position.

Inference Part

The inference part includes one or more of the following three tasks; (a) sample from the posterior distribution $\boldsymbol{\theta}_i \sim p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})$ (Equation 10), (b) compute an expectation $\mathsf{E}_{\boldsymbol{\theta}|\mathbf{y_0}}[h(\boldsymbol{\theta})]$ (Equation 11) and/or (c) evaluate the unnormalized posterior $p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})$ (Equation 8). Sampling is performed by getting n_2 samples from each proposal distribution q_i. For each sample $\boldsymbol{\theta}_{ij}$, the distance function[2] is evaluated for checking if it lies inside the acceptance region. Algorithm 3 defines the steps for computing a weighted sample. After we obtain weighted

---

[2]As before, a surrogate model $\hat{d}$ can be utilized as the distance function if it is available.

Algorithm 2 Approximation C_ϵ^i with a bounding box $\hat{C}_\epsilon^i$; Requires: a model of distance $d_i(\boldsymbol{\theta})$, an optimal point $\boldsymbol{\theta}_i^*$, a number of refinements K, a step size η_start, maximum iterations M and a curvature matrix $\mathbf{H}_i$ ($\mathbf{J}_i^T \mathbf{J}_i$ or GP Hessian)

1: Compute eigenvectors $\mathbf{v}_m$ of $\mathbf{H}_i$ ($m = 1, \ldots, D$)
2: **for** $m \leftarrow 1$ to D **do**
3:     $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}_i^*$
4:     $k \leftarrow 0$
5:     $\eta \leftarrow \eta\_\text{start}$ ▷ Initialize η
6:     **repeat**
7:         $j \leftarrow 0$
8:         **repeat**
9:             $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} + \eta \, \mathbf{v}_m$ ▷ Large step size η.
10:             $j \leftarrow j + 1$
11:         **until** $d(g(\tilde{\boldsymbol{\theta}}, \mathbf{u} = \mathbf{u}_i), \mathbf{y_0}) > \epsilon$ or $j \geq M$ ▷ Check distance or maximum iterations
12:         $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} - \eta \, \mathbf{v}_m$
13:         $\eta \leftarrow \eta/2$ ▷ More accurate region boundary
14:         $k \leftarrow k + 1$
15:     **until** $k = K$
16:     **if** $\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}_i^*$ **then**   ▷ Check if no step has been done
17:         $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} + \dfrac{\eta\_\text{start}}{2^K} \mathbf{v}_m$ ▷ Then, make the minimum step
18:     **end if**
19:     Set $\tilde{\boldsymbol{\theta}}$ as the positive end point along $\mathbf{v}_m$
20:     Run steps 3 - 18 for $\mathbf{v}_m = -\mathbf{v}_m$ and set $\tilde{\boldsymbol{\theta}}$ as the negative end point along $\mathbf{v}_m$
21: **end for**
22: Fit a rectangular box around the region end points and define q_i as uniform distribution

---

samples, computing the expectation is straightforward using Equation 11. Finally, evaluating the unnormalized posterior at a specific point $\boldsymbol{\theta}$ requires access to the distance functions d_i and the prior distribution $p(\boldsymbol{\theta})$. Following Equation 8, we simply count for how many deterministic distance functions it holds that $d_i(\boldsymbol{\theta}) < \epsilon$. It is worth noticing that for evaluating the unnormalized posterior, there is no need for solving the optimization problems and building the proposal regions.

Algorithm 3 Sampling. Requires a function of distance d_i, the prior distribution $p(\boldsymbol{\theta})$, the proposal distribution q_i

---

1: $\boldsymbol{\theta}_{ij} \sim q_i \forall i$ ▷ Sample parameters
2: **for** $i \leftarrow 1$ to n_1 **do**
3:     **for** $j \leftarrow 1$ to n_2 **do**
4:         **if** $d_i(\boldsymbol{\theta}_{ij}) \leq \epsilon$ **then**   ▷ Accept sample
5:             $w_{ij} = \dfrac{p(\boldsymbol{\theta}_{ij})}{q(\boldsymbol{\theta}_{ij})}$ ▷ Compute weight
6:             Store $(w_{ij}, \boldsymbol{\theta}_{ij})$ ▷ Store weighted sample
7:         **end if**
8:     **end for**
9: **end for**

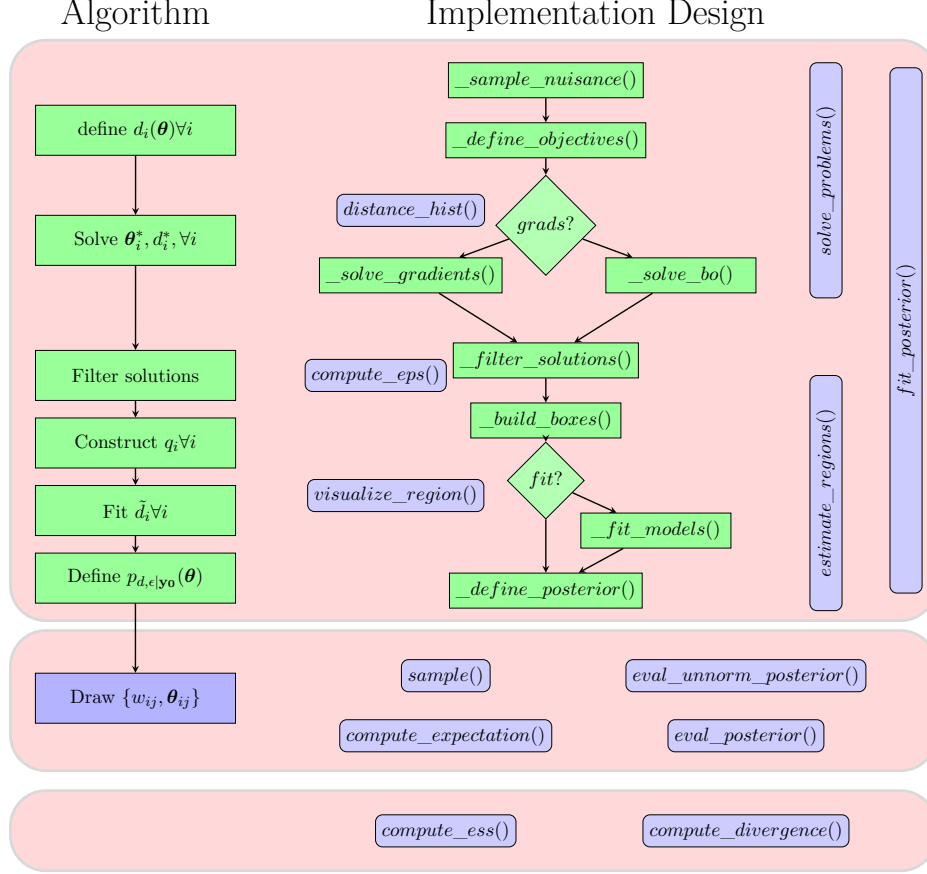## 3.2. General implementation principles

Figure 2: Overview of the ROMC implementation. On the left side, we depict ROMC as a sequence of algotirhmic steps. On the right side, we present the functions that form our implementation; the green rectangles (starting with underscore) are the internal functionalities and the blue rectangles the publicly exposed API. This side-by-side illustration highlights that our implementation follows strictly the algorithmic view of ROMC.

The overview of our implementation is illustrated in Figure 2. Following Python naming principles, the methods starting with an underscore (green rectangles) represent internal (private) functions, whereas the rest (blue rectangles) are the methods exposed at the API. In Figure 2, it can be observed that the implementation follows Algorithm 1. The training part includes all the steps until the computation of the proposal regions, i.e., sampling the nuisance variables, defining the optimization problems, solving them, constructing the regions and fitting local surrogate models. The inference part comprises of evaluating the unnormalized posterior (and the normalized when is possible), sampling and computing an expectation. We also provide some utilities for inspecting the training process, such as plotting the histogram of the final distances or visualizing the constructed bounding boxes. Finally, in the evaluation part, we provide two methods for evaluating the inference; (a) computing the Effective Sample Size (ESS) of the samples and (b) measuring the divergence between the approximate posterior

the ground-truth, if the latter is available.[3]

Parallel version of ROMC

As discussed, ROMC has the significant advantage of being fully parallelisable. We exploit this fact by implementing a parallel version of the major fitting components; (a) solving the optimization problems, (b) constructing bounding box regions. We parallelize these processes using the built-in Python package **multiprocessing**. The specific package enables concurrency, using sub-processes instead of threads, for side-stepping the Global Interpreter (GIL). For activating the parallel version of the algorithm, the user simply has to set `elfi.ROMC(<output_node>, parallelize = True)`.

Simple one-dimensional example

For illustrating the functionalities we will use the following running example introduced by Ikonomov and Gutmann (2019),

$$p(\theta) = \mathcal{U}(\theta; -2.5, 2.5) \tag{13}$$

$$p(y|\theta) = \begin{cases} \theta^4 + u & \text{if } \theta \in [-0.5, 0.5] \\ |\theta| - c + u & \text{otherwise} \end{cases} \tag{14}$$

$$u \sim \mathcal{N}(0, 1) \tag{15}$$

The prior is a uniform distribution in the range $[-2.5, 2.5]$ and the likelihood is defined at Equation 14. The constant c is $0.5 - 0.5^4$ ensures that the PDF is continuous. There is only one observation $y_0 = 0$. The inference in this particular example can be performed quite easily without using a likelihood-free inference approach. We can exploit this fact for validating the accuracy of our implementation. At the following code snippet, we code the model at **ELFI**:

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

def simulator(t1, batch_size = 1, random_state = None):
    c = 0.5 - 0.5**4
    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
```

---

[3]Normally, the ground-truth posterior is not available; However, this functionality is useful in cases where the posterior can be computed numerically or with an alternative method, e.g., Rejection Sampling, and we want to measure the discrepancy between the two approximations.

```
# observation
y = 0

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

# Initialize the ROMC inference method
bounds = [(-2.5, 2.5)] # bounds of the prior
parallelize = False # True activates parallel execution
romc = elfi.ROMC(d, bounds = bounds, parallelize = parallelize)
```

4. Implemented functionalities

At this section, we analyze the functionalities of the training, the inference and the evaluation part. Extended documentation for each method can be found in **ELFI**'s official documentation. Finally, we describe how a user may extend ROMC with its custom modules.

4.1. Training part

In this section, we describe the six functions of the training part:

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

This method (a) draws integers for setting the seed, (b) defines the optimization problems and (c) solves them using either a gradient-based optimizer (default choice) or Bayesian optimization (BO), if `use_bo = True`. The tasks are completed sequentially, as shown in Figure 2. The optimization problems are defined after drawing `n1` integer numbers from a discrete uniform distribution $u_i \sim \mathcal{U}\{1, 2^{32} - 1\}$, where each integer u_i is the seed passed to **ELFI**'s random simulator. The user can pass a `Dict` with custom parameters to the optimizer through `optimizer_args`. For example, in the gradient-based case, the user may pass `optimizer_args = {"method":  "L-BFGS-B", "jac":  jac}`, to select the `"L-BFGS-B"` optimizer and use the callable `jac` to compute the gradients in closed-form.

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

This function helps the user decide which threshold ϵ to use by plotting a histogram of the distances at the optimal point $d_i(\boldsymbol{\theta}_i^*) : \{i = 1, 2, \ldots, n_1\}$ or $\hat{d}_i^*$ in case `use_bo = True`. The function forwards the keyword arguments to the underlying `pyplot.hist()` of the **matplotlib** package. In this way the user may customize some properties of the histogram, e.g., the number of bins.

```
>>> romc.estimate_regions(eps_filter, use_surrogate = None, fit_models = False)
```

This method estimates the proposal regions around the optimal points, following Algorithm 2. The choice about the distance function follows the previous optimization step; if a gradient-based optimizer has been used, then estimating the proposal region is based on the real distance d_i. If BO is used, then the surrogate distance $\hat{d}$ is chosen. Setting `use_surrogate=False` enforces the use of the real distance d even after BO. Finally, the parameter `fit_models` selects whether to fit local surrogate models $\tilde{d}$ after estimating the proposal regions.

The training part includes three more functions. The function `romc.fit_posterior(args*)` which is a syntactic sugar for applying `.solve_problems()` and `.estimate_regions()` in a single step. The function `romc.visualize_region(i)` plots the bounding box around the optimal point of the i-th optimization problem, when the parameter space is up to $2D$. Finally, `romc.compute_eps(quantile)` returns the appropriate distance value $d_{i=\kappa}^*$ where $\kappa = \lfloor quantile \cdot n \rceil$ from the collection $\{d_i^*\}, i = \{1, \ldots, n\}$ 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 - Training part

In the following snippet, we put together the routines to code the training part of the running example.

```
# Training (fitting) part
n1 = 500 # number of optimization problems
seed = 21 # seed for solving the optimization problems
use_bo = False # set to True for switching to Bayesian optimization

# Training step-by-step
romc.solve_problems(n1 = n1, seed = seed, use_bo = use_bo)
romc.theta_hist(bins = 100) # plot hist to decide which eps to use

eps = .75 # threshold for the bounding box
romc.estimate_regions(eps = eps) # build the bounding boxes

romc.visualize_region(i = 2) # for inspecting visually the bounding box

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

4.2. Inference part

In this section, we describe the four functions of the inference part:

```
>>> romc.sample(n2)
```

This is the basic functionality of the inference, drawing n_2 samples for each bounding box region, giving a total of $k \cdot n_2$ samples, where $k < n_1$ is the number of the optimal points remained after filtering. 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 Algorithm 3.
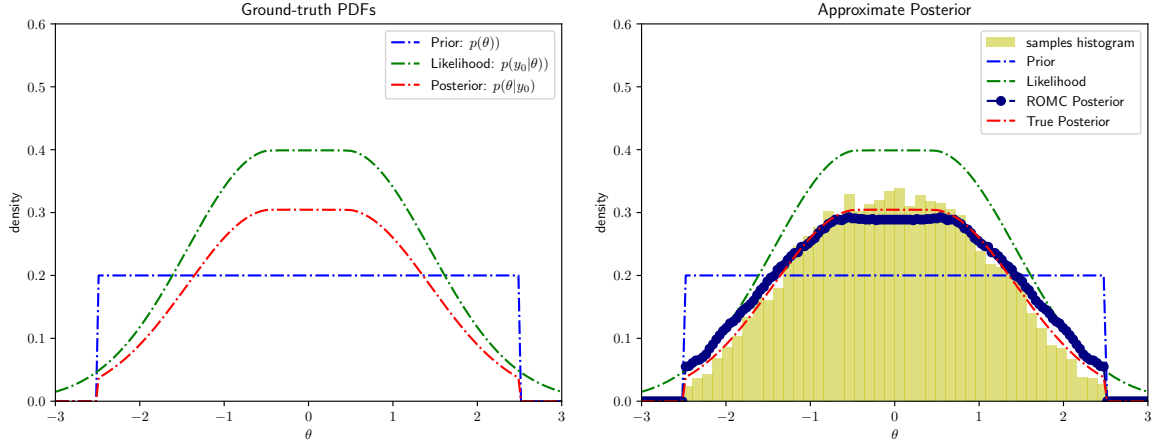
Figure 3: Histogram of distances and visualization of a specific region.

The inference part includes three more function. The function `romc.compute_expectation(h)` computes the expectation $\mathsf{E}_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$ as in Equation 11. The argument `h` can be any python `Callable`. The method `romc.eval_unnorm_posterior(theta, eps_cutoff = False)` computes the unnormalized posterior approximation as in Equation 3. The method `romc.eval_posterior(theta, eps_cutoff = False)` evaluates the normalized posterior estimating the partition function $Z = \int p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})d\boldsymbol{\theta}$ using Riemann's integral approximation. The approximation is computationally feasible only in a low-dimensional parametric space.

Example - Inference part

In the following code snippet, we use the inference utilities to (a) get weighted samples from the approximate posterior, (b) compute an expectation and (c) evaluate the approximate posterior. We also use some of **ELFI**'s built-in tools to get a summary of the obtained samples. For `romc.compute_expectation()`, we demonstrate its use to compute the samples mean and the samples variance. Finally, we evaluate `romc.eval_posterior()` at multiple points to plot the approximate posterior of Figure 3. We observe that the approximation is quite close to the ground-truth.

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

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

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

# Summarize the samples (built-in ELFI tool)
```

```
romc.result.summary()
# Method: ROMC
# Number of samples: 19300
# Parameter                Mean                 2.5%                97.5%
# theta:                  -0.012               -1.985               1.987


# compute expectation
exp_val = romc.compute_expectation(h = lambda x: np.squeeze(x))
print("Expected value  : %.3f" % exp_val)
# Expected value: -0.012


exp_var = romc.compute_expectation(h = lambda x: np.squeeze(x)**2)
print("Expected variance: %.3f" % exp_var)
# Expected variance: 1.120


# eval unnorm posterior
print("%.3f" % romc.eval_unnorm_posterior(theta = np.array([[0]])))
# 58.800


# check eval posterior
print("%.3f" % romc.eval_posterior(theta = np.array([[0]])))
# 0.289
```

4.3. Evaluation part

The method `romc.compute_ess()` computes the Effective Sample Size (ESS) as $\frac{(\sum_i w_i)^2}{\sum_i w_i^2}$, which is a useful quantity to measure how many samples actually contribute to an expectation. For example, in an extreme case of a big population of samples where only one has big weight, the ESS is much smaller than the samples population.

The method `romc.compute_divergence(gt_posterior, bounds, step, distance)` estimate the divergence between the ROMC approximation and the ground truth posterior. Since the estimation is performed using Riemann's approximation, the method can only work in low dimensional spaces. The method can be used for evaluation in synthetic examples where the ground truth is accessible. In a real-case scenarios, where it is not expected to have access to the ground-truth posterior, the user may set the approximate posterior obtained with any other inference approach for comparing the two methods. The argument `step` defines the step used in the Riemann's approximation and the argument `distance` can be either `"Jensen-Shannon"` or `"KL-divergence"`.

```
# Evaluation part
res = romc.compute_divergence(wrapper, distance = "Jensen-Shannon")
print("Jensen-Shannon divergence: %.3f" % res)
# Jensen-Shannon divergence: 0.035


nof_samples = len(romc.result.weights)
ess = romc.compute_ess()
```

```
print("Nof Samples: %d, ESS: %.3f" % (nof_samples, ess))
# Nof Samples: 19300, ESS: 16196.214
```

4.4. Extend the implementation with custom modules

ROMC is a generic LFI framework as it describes a sequence of steps for approximating the posterior distribution without explicitly enforcing a specific algorithm for each step. For completeness, Ikonomov and Gutmann (2019) propose a method for each step but, in general, a user can experiment with alternative methods. Considering that, we designed the implementation to support flexibility.

We have specified four critical parts where a user may intervene using custom methods; (a) gradient-based optimization, (b) Bayesian optimization, (c) proposal region construction and (d) surrogate model fitting. Each of these parts corresponds to an internal function inside the `romc.OptimisationProblem` class; (a) `solve_gradients()`, (b) `solve_bo()`, (c) `build_region()` and (d) `fit_local_surrogate()`, respectively. To replace any of these parts, the user must create a custom class that inherits `OptimisatioProblem` and overwrite the appropriate function(s).

To illustrate this in practice, suppose a user wants to fit Deep Neural Networks instead of, the default, quadratic models as local surrogates $\tilde{d}_i$. Therefore, the user must create a new class that inherits `OptimisationProblem` and overwrite the `fit_local_surrogate(**kwargs)` function with one that fits neural networks as local surrogates. We illustrate that in the following snippet using the `neural_network.MLPRegressor` class of the **scikit-learn** package. The reader can find the end-to-end example here as an online Colab notebook.

```python
class CustomOptim(OptimisationProblem):
    def __init__(self, **kwargs):
        super(CustomOptim, self).__init__(**kwargs)

    def fit_local_surrogate(self, **kwargs):
        nof_samples = 500
        objective = self.objective # the distance function

        # helper function
        def local_surrogate(theta, model_scikit):
            assert theta.ndim == 1
            theta = np.expand_dims(theta, 0)
            return float(model_scikit.predict(theta))

        # create local surrogate model as a function of theta
        def create_local_surrogate(model):
            return partial(local_surrogate, model_scikit = model)

        local_surrogates = []
        for i in range(len(self.regions)):
            # prepare dataset
            x = self.regions[i].sample(nof_samples)
```

```
        y = np.array([objective(ii) for ii in x])

        # train Neural Network
        mlp = MLPRegressor(hidden_layer_sizes = (10,10), solver = 'adam')
        model = Pipeline([('linear', mlp)])
        model = model.fit(x, y)
        local_surrogates.append(create_local_surrogate(model))

    self.local_surrogates = local_surrogates
    self.state["local_surrogates"] = True
```

In a similar way, the user can replace any of the other three functions. In each case, the custom function must update some class-level variables that hold the state of the training phase. In the following sections, we present which are these variables in each function. Furthermore, when implementing custom functions, the user may use two helping classes; (a) `RomcOpimisationResult`, that stores the result of the optimization and (b) `NDimBoundingBox`, that stores the bounding box. We present their definitions in the following snippets and we illustrate how to use them in the next sections. Both classes can be imported from the module `elfi.methods.inference.romc`.

```
class RomcOpimisationResult:
    def __init__(self, x_min, f_min, hess_appr):
        Parameters
        ----------
        x_min: np.ndarray (D,), minimum
        f_min: float, distance at x_min
        hess_appr: np.ndarray (D,D), Hessian approximation at x_min

class NDimBoundingBox:
    def __init__(self, rotation, center, limits):
        Parameters
        ----------
        rotation: np.ndarray (D,D), rotation matrix for the bounding box
        center: np.ndarray (D,) center of the bounding box
        limits: np.ndarray (D,2), the limits of the bounding box
```

(a) Extending gradient-based optimization

For replacing the default gradient-based optimization method, the user must overwrite the function `solve_gradients()`. Using the objective function d_i (`self.objective`), the custom method must store the result of the optimization as a `RomcOptimisationResult` instance. In the following snippet, after the comment `# state variables`, we present the class-level variables that must be set by the method.

```
def solve_gradients(self, **kwargs):
    # useful variables
```

```
    # self.objective: Callable, the distance function

    # code custom solution here
    result = RomcOptimisationResult(x = .., y = .., jac = .., hess_inv = ..)
    success: bool = ... # whether optimization was successful

    # state variables
    self.state["attempted"] = True
    if success:
        self.result = result
        self.state["solved"] = True
        return True
    else:
        return False
```

(b) Extending Bayesian Optimization

For replacing the default Bayesian optimization function, the procedure is similar to the gradient-based case. As presented in the following snippet, the additional class-level variables that must be set are; (a) `self.surrogate = custom_surrogate`, where `custom_surrogate` is a `Callable` and (b) `self.state["has_fit_surrogate"] = True` if the optimization is successful.

```
def solve_bo(self, **kwargs):
    # useful variables
    # self.objective: Callable, the distance function

    # code custom solution here
    result = RomcOptimisationResult(x = .., y = .., jac = .., hess_inv = ..)
    custom_surrogate = ... # store a Callable here
    success: bool = ... # whether optimization was successful

    # state variables
    self.state["attempted"] = True
    if success:
        self.result = result
        self.surrogate = custom_surrogate
        self.state["solved"] = True
        self.state["has_fit_surrogate"] = True
        return True
    else:
        return False
```

(c) Extending the proposal region construction

For replacing the construction of the proposal region the user must overwrite the `build_region` method. Using the objective function d_i (`self.objective`) the method must estimate a list

of bounding boxes as a `List` with `NDimBoundingBox` instances and set the state variables presented below. An end-to-end example for using a custom region construction module can be found here.

```python
def build_region(self, **kwargs):
    # useful variables
    # self.objective: Callable, the distance function

    # custom build_region method
    eps: float = ... # epsilon used in region estimation
    bounding_box: List[NDimBoundingBox] = ...
    success: bool = ... # whether region built successfully

    # state variables
    self.eps_region = eps
    if success:
        # construct region
        self.regions = bounding_box
        self.state["region"] = True
        return True
    else:
        return False
```

(d) Extending the surrogate model fitting

For replacing the surrogate model fitting the user must overwrite the `fit_local_surrogate` method. Using the objective function d_i (`self.objective`) and the estimated bounding boxes (`self.regions`), the method must create a list of local surrogates, one for each region, as a `List` with `Callables` and set the state variables as presented in the following snippet.

```python
def fit_local_surrogate(self, **kwargs):
    # useful variables
    # self.objective: Callable, the distance function
    # self.regions: List[NDimBoundingBox], the bounding boxes

    # custom local surrogates
    local_surrogates: List[Callable] = ... # the surrogate models
    success: bool = ... # whether surrogates fit successfully

    # state variables
    if success:
        self.local_surrogates = custom_surrogates
        self.state["local_surrogates"] = True
        return True
    else:
        return False
```

5. Use-case illustration

In this section, we test the implementation using the second-order moving average (MA2) example, which is one of the standard models of **ELFI**. We perform the inference using three different versions of ROMC; (i) with a gradient-based optimizer, (ii) with Bayesian Optimization and (iii) fitting a Neural Network as a surrogate model. The later illustrates how to extend the implementation, replacing part of ROMC with a user-defined component. Finally, we measure the execution speed-up using the parallel version of ROMC.

Model Definition

MA2 is a probabilistic model for time series analysis. The observation at time t is given by,

$$y_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2}, \quad t = 1, \ldots, T \tag{16}$$

$$\theta_1, \theta_2 \in \mathbb{R}, \quad w_k \sim \mathcal{N}(0, 1), k \in \mathbb{Z} \tag{17}$$

The random variable $w_k \sim \mathcal{N}(0, 1)$ is white noise and the two parameters of interest, θ_1, θ_2, model the dependence from the previous observations. The parameter T is the number of sequential observations which is set to $T = 100$. For securing that the inference problem is identifiable, i.e., the likelihood has only one mode, we use the prior proposed by Marin *et al.* (2012),

$$p(\boldsymbol{\theta}) = p(\theta_1)p(\theta_2|\theta_1) = \mathcal{U}(\theta_1; -2, 2)\mathcal{U}(\theta_2; \theta_1 - 1, \theta_1 + 1) \tag{18}$$

The observation vector $\mathbf{y}_0 = (y_1, \ldots, y_{100})$ is generated with $\boldsymbol{\theta}^* = (0.6, 0.2)$. The dimensionality of the output $\mathbf{y}$ is high, therefore we use summary statistics. Considering that the output vector represents a time-series signal, we select the autocovariances with lag $= 1$ and lag $= 2$, as shown in Equations 19 and 20. The distance between the observation and the simulator output is measured with the squared Euclidean distance, as shown in Equation 22.

$$s_1(\mathbf{y}) = \frac{1}{T-1} \sum_{t=2}^{T} y_t y_{t-1} \tag{19}$$

$$s_2(\mathbf{y}) = \frac{1}{T-2} \sum_{t=3}^{T} y_t y_{t-2} \tag{20}$$

$$s(\mathbf{y}) = (s_1(\mathbf{y}), s_2(\mathbf{y})) \tag{21}$$

$$d = ||s(\mathbf{y}) - s(\mathbf{y}_0)||_2^2 \tag{22}$$

Inference

To demonstrate the full capabilities of our ROMC implementation, we perform inference using three different methods: (i) a gradient-based optimizer, (ii) Bayesian optimization, and (iii) fitting a neural network (NN) as a surrogate model. The use of a NN as a surrogate model serves as an example of the extensibility of our implementation, as described in Chapter 4.4. For the NN, we employ the **MLPRegressor** class from the **scikit-learn** package. The NN

	μ_{θ_1}	σ_{θ_1}	μ_{θ_2}	σ_{θ_2}
Rejection ABC	0.516	0.142	0.070	0.172
ROMC (gradient-based)	0.501	0.142	0.033	0.169
ROMC (Bayesian optimization)	0.513	0.169	0.090	0.174
ROMC (Neural Network)	0.491	0.138	0.040	0.172

Table 1: Comparison of the samples obtained from the estimated posterior with (a) Rejection sampling and (b) the different versions of ROMC. We observe that the obtained samples share similar statistics along all methods.

$(\tilde{d}_i)$ substitutes the actual distance function (d_i) inside the proposal regions. Therefore, all inference actions, namely, sampling, expectation computation, and posterior evaluation are based on $\tilde{d}_i$. We use a NN with two hidden layers of 10 neurons each and train it using 500 examples from each proposal region. To compare the results of ROMC inference with a traditional ABC algorithm, we also include Rejection Sampling in our analysis.

In Figure 4, we illustrate the acceptance region of the same deterministic simulator, in the gradient-based and the Bayesian optimization case. The acceptance regions are quite similar even though the different optimization schemes lead to different optimal points.

In Figure 5, we demonstrate the histograms of the marginal posteriors, for each approach; (a) Rejection ABC, (b) ROMC with gradient-based optimization (c) ROMC with Bayesian optimization and (d) ROMC with the NN extension. We observe a significant agreement between the different approaches. At Table 1 we present the empirical mean μ and standard deviation σ for each inference approach and finally, in Figure 6, we illustrate the unnormalized posterior for the three different variations of the ROMC method. The results show that all ROMC variations provide consistent results between them which are in agreement with the Rejection ABC algorithm.
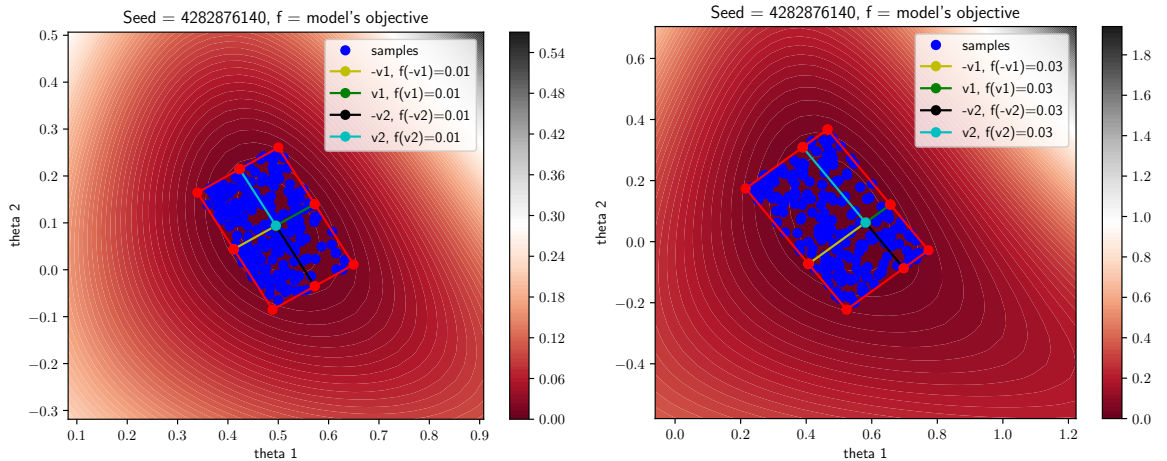

Figure 4: The acceptance region in a specific optimization problem. In the left figure the region obtained with gradient-based optimizer and in the right one with Bayesian Optimization.
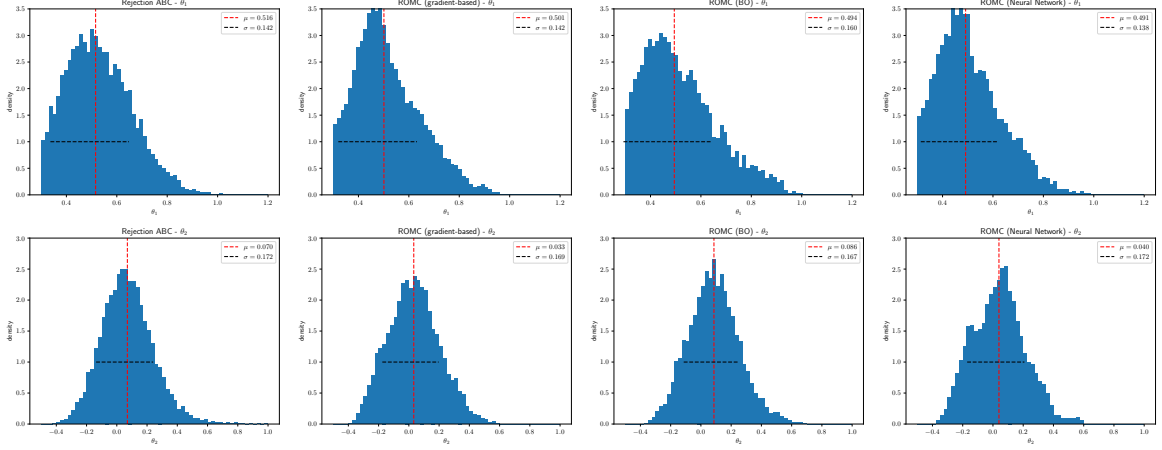
Figure 5: Histogram of the marginal posterior distributions using three different inference approaches; (a) in the first row, the samples are obtained using Rejection ABC sampling (b) in the second row, using ROMC with a gradient-based optimizer and (c) in the third row, using ROMC with Bayesian optimization approach. The vertical (red) line represents the samples mean μ and the horizontal (black) the standard deviation σ.
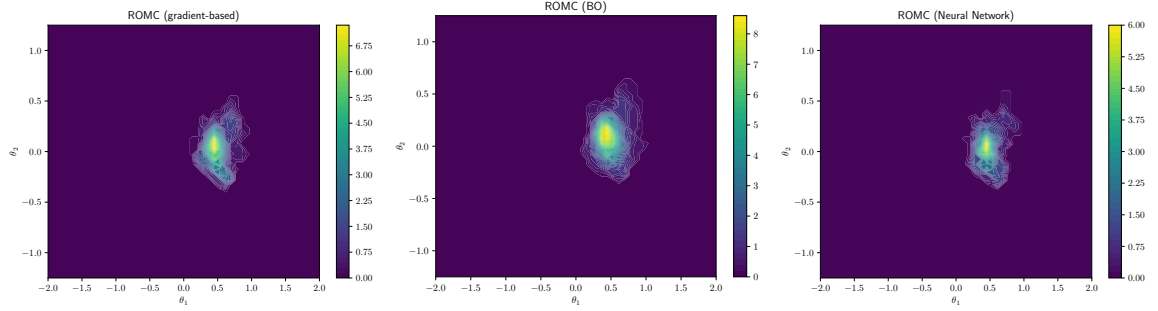

Figure 6: The unnormalized posterior distribution using the ROMC method with (a) a gradient-based optimization (b) Bayesian Optimization (c) gradient-based with a Neural Network as a surrogate model.

Parallelize the implementation

As stated above, ROMC is an approach that can be executed in a fully-parallelized manner, exploiting all CPU cores. In our implementation, we support a parallel version of the the training part, namely, for solving the optimization problems and for estimating the proposal regions. The parallel version of the algorithm is built on top of the built-in Python package **multiprocess** for using all the available CPU cores. In Figure 7 we observe the execution times for performing the inference on the MA2 model; the parallel version performs both tasks almost five times faster than the sequential. The result is reasonable given that the experiments have run in a single machine with the Intel® Core™ i7-8750H Processor, which has six separate cores.
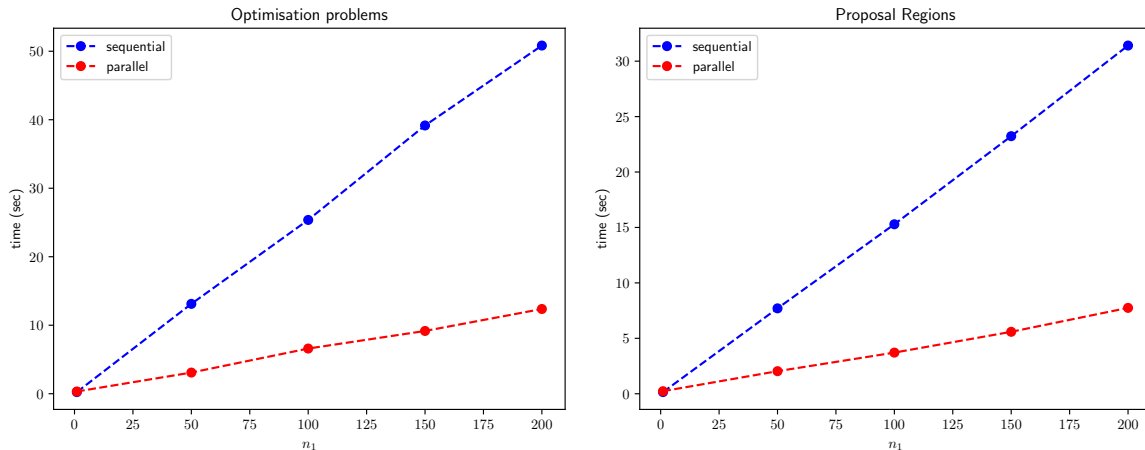
Figure 7: Comparison between parallel and sequential execution of ROMC. We observe that the parallel version runs almost 5 times faster.

6. Summary and discussion

In this paper, we presented the implementation of the LFI method ROMC at the Python package **ELFI**. We highlighted two different use-cases. First, we illustrated how a user may exploit the provided API to solve an LFI problem. Second, we focus on the scenario where a researcher wants to intervene and alter parts of the method to experiment with new approaches. Since (Robust) Optimization Monte Carlo is a novel approach for statistical inference and, to the best of our knowledge, this is the first open-source implementation on a generic package, we believe that the later is the biggest contribution.

There are still open challenges for enabling ROMC to solve high-dimensional LFI problems efficiently. The first is enabling the execution of ROMC execution into a distributed environment, i.e., a cluster of computers. ROMC can be characterized as an *embarrassingly parallel* workload; each optimization problem is an entirely independent task. Therefore, supporting inference into a cluster of computers can radically speed up the inference. The second is the implementation of the method on a framework that supports automatic differentiation. Automatic differentiation is necessary for efficiently solving optimization problems, especially in high-dimensional parametric models.

Computational details

The results in this paper were obtained using Python 3.9, **ELFI 0.8.4** at Ubuntu 20.04 lts operating system, at a single machine with Intel® Core™ i7-8750H processor.

Acknowledgments

References

Bisong E (2019). "**Google Colaboratory**." doi:10.1007/978-1-4842-4470-8_7. URL https://doi.org/10.1007/978-1-4842-4470-8_7.

Blum MG, François O (2010). "Non-Linear Regression Models For Approximate Bayesian Computation." *Statistics and computing*, **20**(1), 63–73.

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, *et al.* (2018). "**JAX**: Composable Transformations of Python + **NumPy** Programs." *Version 0.2*, **5**, 14–24.

Chen Y, Gutmann MU (2019). "Adaptive Gaussian Copula ABC." In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 1584–1592. PMLR. URL http://proceedings.mlr.press/v89/chen19d.htmlhttps://proceedings.mlr.press/v89/chen19d.html.

Cranmer K, Brehmer J, Louppe G (2020). "The Frontier Of Simulation-Based Inference." *Proceedings of the National Academy of Sciences*, **117**(48), 30055–30062.

Forneron JJ, Ng S (2016). "A Likelihood-Free Reverse Sampler Of The Posterior Distribution." In *Essays in Honor of Aman Ullah*. Emerald Group Publishing Limited.

Gutmann MU, Corander J (2016). "Bayesian Optimization For Likelihood-Free Inference Of Simulator-Based Statistical Models." *Journal of Machine Learning Research*.

Hagberg A, Swart P, S Chult D (2008). "Exploring Network Structure, Dynamics, And Function Using **Networkx**." *Technical report*, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Harris CR, Millman KJ, Van Der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, *et al.* (2020). "Array Programming with **NumPy**." *Nature*, **585**(7825), 357–362.

Hermans J, Begy V, Louppe G (2020). "Likelihood-Free MCMC With Amortized Approximate Ratio Estimators." In *International Conference on Machine Learning*, pp. 4239–4248. PMLR.

Ikonomov B, Gutmann MU (2019). "Robust Optimisation Monte Carlo." *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, **108**, 2819–2829. URL http://arxiv.org/abs/1904.00670.

Lintusaari J, Gutmann MU, Dutta R, Kaski S, Corander J (2017). "Fundamentals and Recent Developments in Approximate Bayesian Computation." *Systematic biology*, **66**(1), e66–e82.

Lintusaari J, Vuollekoski H, Kangasraasio A, Skytén K, Jarvenpaa M, Marttinen P, Gutmann MU, Vehtari A, Corander J, Kaski S (2018). "**Elfi: Engine For Likelihood-Free Inference**." *Journal of Machine Learning Research*, **19**(16), 1–7.

Marin JM, Pudlo P, Robert CP, Ryder RJ (2012). "Approximate Bayesian Computational Methods." *Statistics and Computing*, **22**(6), 1167–1180.

Meeds T, Welling M (2015). "Optimization Monte Carlo: Efficient and Embarrassingly Parallel Likelihood-free Inference." *Advances in Neural Information Processing Systems*, **28**. URL https://proceedings.neurips.cc/paper/2015/file/a284df1155ec3e67286080500df36a9a-Paper.pdf.

Papamakarios G, Murray I (2016). "Fast ε-free Inference Of Simulation Models With Bayesian Conditional Density Estimation." *Advances in neural information processing systems*, **29**.

Papamakarios G, Sterratt D, Murray I (2019). "Sequential Neural Likelihood: Fast Likelihood-Free Inference With Autoregressive Flows." In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 837–848. PMLR.

Shahriari B, Swersky K, Wang Z, Adams RP, De Freitas N (2015). "Taking The Human Out Of The Loop: A Review Of Bayesian Optimization." *Proceedings of the IEEE*, **104**(1), 148–175.

Sisson SA, Fan Y, Beaumont M (2018). *Handbook Of Approximate Bayesian Computation.* CRC Press.

Thomas O, Dutta R, Corander J, Kaski S, Gutmann MU (2022). "Likelihood-Free Inference By Ratio Estimation." *Bayesian Analysis*, **17**(1), 1–31.

Wood SN (2006). *Generalized Additive Models: An Introduction With R.* chapman and hall/CRC.

Affiliation:

Vasilis Gkolemis
Information Management Systems Institute (IMSI)
ATHENA Research and Innovation Center
Athens, Greece
E-mail: vgkolemis@athenarc.gr
URL: https://givasile.github.io