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 intractable likelihood is challenging; most likelihood-free inference (LFI) methods encounter accuracy and efficiency limitations. In this paper, we present the implementation of Robust Optimisation Monte Carlo (ROMC) in the Python package **ELFI**. ROMC is a novel and efficient (highly-parallelisable) 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 harmonised 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, all steps run in a fully-parallelisable 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 modelling freedom they provide. In essence, a simulator-based model can describe any data generating mechanism that can be written as a finite set of algorithmic steps. Such freedom comes at a cost; performing the inference, i.e., sampling or evaluating the posterior, is challenging.

Optimisation Monte Carlo (OMC) proposed by Meeds and Welling (2015) is a novel LFI approach for approximating the posterior distribution. The central idea is turning the stochastic data-generating mechanism into a set of deterministic optimisation processes. Afterwards, Forneron and Ng (2016) provided a similar method under the name 'reverse sampler'. In their

work, Ikonomov and Gutmann (2020), located some critical limitations of OMC. They proposed Robust OMC (ROMC), an alternative version of OMC with appropriate modifications.

In this paper, we present an implementation of ROMC at the Python package **Engine for Likelihood-Free inference (ELFI)**. We follow appropriate design principles for ensuring extensibility. As we describe analytically in the next chapter, ROMC is a general framework for obtaining weighted samples from the posterior; it defines a sequence of algorithmic steps without enforcing a specific algorithm for solving each step[1]. A researcher may adopt ROMC and develop novel methods to solve each task. We have designed our software for facilitating such alterations. Finally, we have tested the accuracy and the efficiency of our implementation on some LFI examples supported by the **ELFI** package.

To the best of our knowledge, this is the first attempt to implement the ROMC inference method to a generic LFI framework; there are no other packages for making direct comparisons. Therefore, we test it against (i) an artificial example with tractable likelihood and (ii) the second-order moving average (MA2) example from the **ELFI** package. In the latter, we generate data using known parameters and compare our results with the typical Rejection ABC Beaumont, Zhang, and Balding (2002).

2. Background

We first give a short introduction to simulator-based models and LFI. We then focus on the methods forming the basis of this paper; OMC and its robust improvement, ROMC. Finally, we introduce ELFI, the software package used for our implementation.

2.1. Simulator-based models and likelihood-free inference

An implicit or simulator-based model is a parameterised stochastic data generating mechanism. The key characteristic of these models is that we can sample data points, but we cannot evaluate the likelihood. Formally, a simulator-based model is a parameterised family of probability density functions $\{p(\mathbf{y}|\boldsymbol{\theta})\}_{\boldsymbol{\theta}}$ whose closed-form is either unknown or computationally intractable. Practically, in these scenarios, we can only a access a simulator $M_r(\boldsymbol{\theta})$, i.e. a random black-box machine (computer code) that generates samples $\mathbf{y}$ in a stochastic manner for any given a set of parameters $\boldsymbol{\theta}$; $M_r(\boldsymbol{\theta}) \to \mathbf{y}$. We can isolate the randomness by introducing the stochastic nuisance variables $\mathbf{u} \sim p(\mathbf{u})$ so that the simulator becomes a deterministic mapping g that maps $(\boldsymbol{\theta}, \mathbf{u})$ to the data $\mathbf{y}$, i.e. $\mathbf{y} = g(\boldsymbol{\theta}, \mathbf{u})$. Within the computer code, the distribution $p(\mathbf{u})$ is defined as the random number generating process.

Simulator-based models provide considerable modelling freedom; any physical process that can be conceptualised as a computer program of finite steps can be modelled as a simulator-based model without any compromise. The modelling freedom allows for any amount of hidden (unobserved) internal variables or rule-based decisions. Hence, implicit models are often used to model physical phenomena in the natural sciences such as e.g. genetics, epidemiology, or neuroscience. Further background on simulator-based models and example applications can be found in the articles by Gutmann and Corander (2016); Lintusaari, Gutmann, Dutta, Kaski, and Corander (2017); Sisson, Fan, and Beaumont (2018); Cranmer, Brehmer, and Louppe (2020).

---

[1]For being a ready-to-use algorithm, Ikonomov and Gutmann (2020) proposed a default method for each step, but this choice is by no means restrictive.

The modelling freedom of simulator-based models, however, comes at the price of difficulties in inferring their parameters. Denoting the observed data as $\mathbf{y_0}$, the main difficulty is that the likelihood function $L(\boldsymbol{\theta}) = p(\mathbf{y_0}|\boldsymbol{\theta})$ is generally intractable. To better see the sources of the intractability, and to address them, we go back to the basic characterisation of the likelihood as the (rescaled) probability of a parameter of the model to generate data $\mathbf{y}$ that is similar to the observed data $\mathbf{y_0}$. More formally, the likelihood $L(\boldsymbol{\theta})$ equals

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

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 via a distance function d, i.e. $B_{d,\epsilon}(\mathbf{y_0}) := \{\mathbf{y} : d(\mathbf{y}, \mathbf{y_0}) \leq \epsilon\}$.

The basic characterisation of the likelihood in (1) highlights two sources of intractability: The first is the computation of the probability $\Pr(g(\boldsymbol{\theta}, \mathbf{u}) \in B_{d,\epsilon}(\mathbf{y_0}))$, the second is the limit of $\epsilon \to 0$. Approximating the probability with samples becomes computationally infeasible if ϵ is too small. Hence, a large class of inference methods work with $\epsilon > 0$, which leads to the approximate likelihood function $L_{d,\epsilon}(\boldsymbol{\theta})$

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

and, in turn, to the approximate posterior

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

The approximation in (2) is by no means the only strategy to deal with the intractabilities of the likelihood function in (1). Other strategies include modelling 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 Francois (2010); Wood (2010); Papamakarios and Murray (2016); Papamakarios, Sterratt, and Murray (2019); Chen and Gutmann (2019); Thomas, Dutta, Corander, Kaski, and Gutmann (2020); Hermans, Begy, and Louppe (2020). However, both OMC and robust OMC, which we introduce next, are based on the approximation in (2).

2.2. Optimisation Monte Carlo (OMC)

Our description of OMC Meeds and Welling (2015) follows Ikonomov and Gutmann (2020) who base their explanation of OMC on the approximate likelihood function in (2). With the indicator function

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

we can write the approximate likelihood function $L_{d,\epsilon}(\boldsymbol{\theta})$ in (2) as

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

which can be approximated as a sample's average

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

We thus approximate the likelihood of a specific $\boldsymbol{\theta}$ by counting the fraction of samples that lie inside a volume around the observations. Isolating the randomness of the simulator via the $\mathbf{u}_i$ has two crucial consequences: First, we can sample the nuisance variables $\mathbf{u}_i \sim p(\mathbf{u})$ only once and reuse them to approximate $L_{d,\epsilon}(\boldsymbol{\theta})$ for different $\boldsymbol{\theta}$. Second, since $\mathbf{y}_i = g(\boldsymbol{\theta}, \mathbf{u}_i)$ is a function of $\boldsymbol{\theta}$, checking whether $\mathbf{y}_i$ is contained in $B_{d,\epsilon}(\mathbf{y_0})$ is the same as checking whether $\boldsymbol{\theta}$ is in the acceptance region $C_\epsilon^i = \{\boldsymbol{\theta} : g(\boldsymbol{\theta}, \mathbf{u}_i) \in B_{d,\epsilon}(\mathbf{y_0})\}$. This leads to the likelihood approximation

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

and the corresponding approximate posterior

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

As argued by Ikonomov and Gutmann (2020), 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 characterising 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 realisation $\mathbf{u}_i \sim p(\mathbf{u})$, i.e.

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

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 2020).

2.3. Robust optimisation Monte Carlo (ROMC)

Ikonomov and Gutmann (2020) 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, the curvature alone may wrongly indicate that C_ϵ^i is much larger than it actually is. In our software package we implement the robust generalisation of OMC by Ikonomov and Gutmann (2020) that resolves this issue.

ROMC, firstly, approximates the acceptance regions C_ϵ^i, and defines proposal distributions $q_i(\boldsymbol{\theta})$ on them. The proposal distributions are used for generating posterior samples $\boldsymbol{\theta}_{ij} \sim q_i$. The samples are assigned (importance) weights w_{ij} that compensate for using the proposal distributions $q_i(\boldsymbol{\theta})$ and not the prior $p(\boldsymbol{\theta})$,

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

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

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

Ikonomov and Gutmann (2020) considered uniform distributions as proposal distributions so that the main task is to approximate the acceptance regions C_ϵ^i and to represent them so that uniform sampling is easy. The approximation of the acceptance regions contains two compulsory and one optional step: (1) solving the optimisation problems as in OMC, (2) constructing bounding boxes around C_ϵ^i and optionally, (3) refining the approximation via a surrogate model of the distance.

Solving the deterministic optimisation 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 general, d can be any valid distance, but for the rest of the paper we consider d as the squared Euclidean distance. For notational convenience, we denote $d(g(\boldsymbol{\theta}, \mathbf{u}_i), \mathbf{y_0})$ as $d_i(\boldsymbol{\theta})$. Obtaining θ_i^* involves solving the following optimisation problem:

$$\min_{\boldsymbol{\theta}} \quad d_i(\boldsymbol{\theta}) \tag{12a}$$

$$\text{subject to} \quad d_i(\boldsymbol{\theta}) \leq \epsilon \tag{12b}$$

The optimisation problem can be treated as unconstrained, accepting the optimal point $\boldsymbol{\theta}_i^* = \text{argmin}_{\boldsymbol{\theta}} d_i(\boldsymbol{\theta})$ only if $d_i(\boldsymbol{\theta}_i^*) \leq \epsilon$. If $d_i(\boldsymbol{\theta})$ is differentiable any gradient-based optimizer can be used for 12a. The gradients $\nabla_{\boldsymbol{\theta}} d_i(\boldsymbol{\theta})$ can be either provided in closed form or approximated by finite differences. In case d_i is not differentiable, Bayesian Optimisation (Shahriari, Swersky, Wang, Adams, and de Freitas 2016) provides an alternative approach. In this scenario, apart from obtaining an optimal $\boldsymbol{\theta}_i^*$, a surrogate model $\hat{d}_i(\boldsymbol{\theta})$ of the distance function $d_i(\boldsymbol{\theta})$ is also automatically obtained; $\hat{d}_i$ can then substitute the actual distance function in downstream steps of the algorithms, with possible computational gains especially if evaluating the actual distance $d_i(\boldsymbol{\theta})$ is expensive.

Estimating the acceptance regions

The acceptance region C_ϵ^i is approximated by a bounding box $\hat{C}_\epsilon^i$. Ideally, we want the bounding box to be as tight as possible to C_ϵ^i to ensure high acceptance rate in the importance sampling, but big enough for not discarding valid parts. The bounding boxes are built in two steps. First, we define their axes $\mathbf{v}_m, m = \{1, \ldots, D\}$ based on the (estimated) curvature of the distance at $\boldsymbol{\theta}_i^*$. Second, we determine the size of the box via a one-dimensional line-search method along each axis, see 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.

Refining the estimate via a local surrogate model (optional)

When computing the weight w_{ij} in (10), we need to check whether the samples $\boldsymbol{\theta}_{ij} \sim q_i$ lie 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 evaluting the distance function $d_i(\boldsymbol{\theta}_{ij})$, which can be expensive if the model is complex. Ikonomov and

Gutmann (2020) thus proposed to fit a surrogate model $\tilde{d}_i(\boldsymbol{\theta})$ of the distance function $d_i(\boldsymbol{\theta})$, on data points that lie inside $\hat{C}_\epsilon^i$. They used a simple quadratic model whilst other regression models are, in principle, possible too. The advantage of using a quadratic model is that it has ellipsoidal isocontours, which thus naturally allowed Ikonomov and Gutmann (2020) to replace the bounding box approximation of C_ϵ^i with a tigher-fitting ellipsoidal approximation.[2]

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 Optimisation 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)[3] Lintusaari, Vuollekoski, Kangasrääsiö, Skytén, Järvenpää, Marttinen, Gutmann, Vehtari, Corander, and Kaski (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, making it straightforward to perform comparisons. **ELFI** handles the probabilistic model as a Directed Acyclic Graph (DAG). This functionality is based on the package **NetworkX** Hagberg, Swart, and S Chult (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 include in the `elfi.Prior` class.

All inference methods in **ELFI** share two rules;

- they follow the signature `elfi.<Class name>(<output node>, *arg)`; the initial argument is the output node of the model and the rest of the arguments are hyper-parameters of the method.

- they obtain posterior samples under the API `<method_name>.sample()`

3. Implementation

This section is split in two parts. We first express ROMC as an algorithm and then we present the general implementation principles we follow.

---

[2]The difference to the infinitesimal ellipsoidal model in OMC is the estimation procedure: OMC uses information at $\boldsymbol{\theta}_i^*$ whilst, here, information in $\hat{C}_\epsilon^i$ is used, which results in a more stable fit.

[3]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)
```
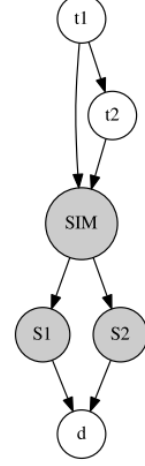
Figure 1: Baseline example for creating an **ELFI** model. Image taken from Lintusaari *et al.* (2018)

3.1. Algotirhmic view of ROMC

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

Training part

At the training (fitting) part, the goal is the estimation of the proposal regions $\hat{C}_\epsilon^i$. The tasks are; (a) sampling the nuisance variables $\mathbf{u}_i \sim p(\mathbf{u})$ for obtaining the deterministic functions $d_i(\boldsymbol{\theta})$, (b) solving the optimisation problems $\min_{\boldsymbol{\theta}} d_i(\boldsymbol{\theta})$ for obtaining $\boldsymbol{\theta}_i^*, d_i^*$, and (c) estimate the proposal distribution q_i.

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. This approximation requires two evaluations of d_i for *each* parameter $\theta_m, m \in \{1, \ldots, D\}$[4], which works in low-dimensional problems. If $d_i(\boldsymbol{\theta})$ is not differentiable, Bayesian Optimisation can be used as an alternative solution. In this scenario, the training part becomes slower due to fitting of the surrogate model and the blind optimisation steps.

After obtaining the optimal points $\boldsymbol{\theta}^*$, we estimate the proposal regions. Algorithm 2 describes the line search approach for finding the region's boundaries. An important step for the proposal region estimation is deciding the axes of the bounding box; the directions $\mathbf{v}_m, m = \{1, \ldots, D\}$ we follow for reaching the boundaries. We approximate them as the direction of the highest curvature of d_i at $\boldsymbol{\theta}_i^*$. This estimation is given by the eigenvalues of the Hessian

---

[4]D is the dimensionality of $\boldsymbol{\theta}$, i.e. $\boldsymbol{\theta} \in \mathbb{R}^D$

Algorithm 1 ROMC. Requires the prior $p(\boldsymbol{\theta})$, the simulator $M_r(\boldsymbol{\theta})$, number of optimisation 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 optimisation 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 unnormalised posterior
19: **end procedure**

---

matrix $\mathbf{H}_i$ of d_i[5] at $\boldsymbol{\theta}_i$. The Hessian matrix is approximated numerically. In case where the distance function is the Euclidean, 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 $\Phi(g(\boldsymbol{\theta}, \mathbf{u} = \mathbf{u}_i))$ at $\boldsymbol{\theta}_i^*$. The approximation through the Jacobian matrix has the computational advantage of using only first-order derivatives.

Inference Part

Performing the inference includes one or more of the following three tasks; (a) sampling from the posterior $\boldsymbol{\theta}_i \sim p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y}_0)$, (b) computing an expectation $\mathsf{E}_{\boldsymbol{\theta}|\mathbf{y}_0}[h(\boldsymbol{\theta})]$ and (c) evaluating the unnormalised posterior $p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y}_0)$. Sampling is performed by getting n_2 samples from each proposal distribution q_i. For each sample $\boldsymbol{\theta}_{ij}$, the distance function[6] is evaluated for checking if it lies inside the acceptance region. Algorithm 3 defines the steps for computing a weighted sample. Computing the expectation is easy after weighted samples are obtained using the equation 11, so we do not discuss it separately. Evaluating the unnormalised posterior requires the deterministic functions g_i and the prior distribution $p(\boldsymbol{\theta})$[7]. The evaluation requires iterating over all g_i and evaluating the distance from the observed data.

---

[5] Either the real distance d_i or the Gaussian Process approximation $\hat{d}_i$
[6] As before, if a surrogate model $\hat{d}$ is available, it can be utilised as the distance function.
[7] There is no need for solving the optimisation problems and building the proposal regions.

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 $\mathbf{H}_i$ $(d = 1, \ldots, D)$
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:          $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} + \dfrac{\eta\_\text{start}}{2^K}\mathbf{u}_m$ ▷ Then, make the minimum step
18:      **end if**
19:      Set $\tilde{\boldsymbol{\theta}}$ as the positive end point along $\mathbf{v}_m$
20:      Runc 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

---

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

---
1: $\boldsymbol{\theta}_{ij} \sim q_i$
2: **if** $d_i(\boldsymbol{\theta}_{ij}) > \epsilon$ **then**
3:      Go to 2   ▷ Reject sample
4: **else**
5:      $w_{ij} = \frac{p(\boldsymbol{\theta}_{ij})}{q(\boldsymbol{\theta}_{ij})}$ ▷ Compute weight
6:      Store $(w_{ij}, \boldsymbol{\theta}_{ij})$ ▷ Store weighted sample
7: **end if**

3.2. General implementation principles

The overview of our implementation is illustrated in Figure 2. Following Python's naming principles, the methods starting with an underscore (green rectangles) represent internal (private) functions, whereas the rest (blue rectangles) are the methods exposed as the public API. In Figure 2, it can be easily 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 optimisation problems, solving them, constructing the
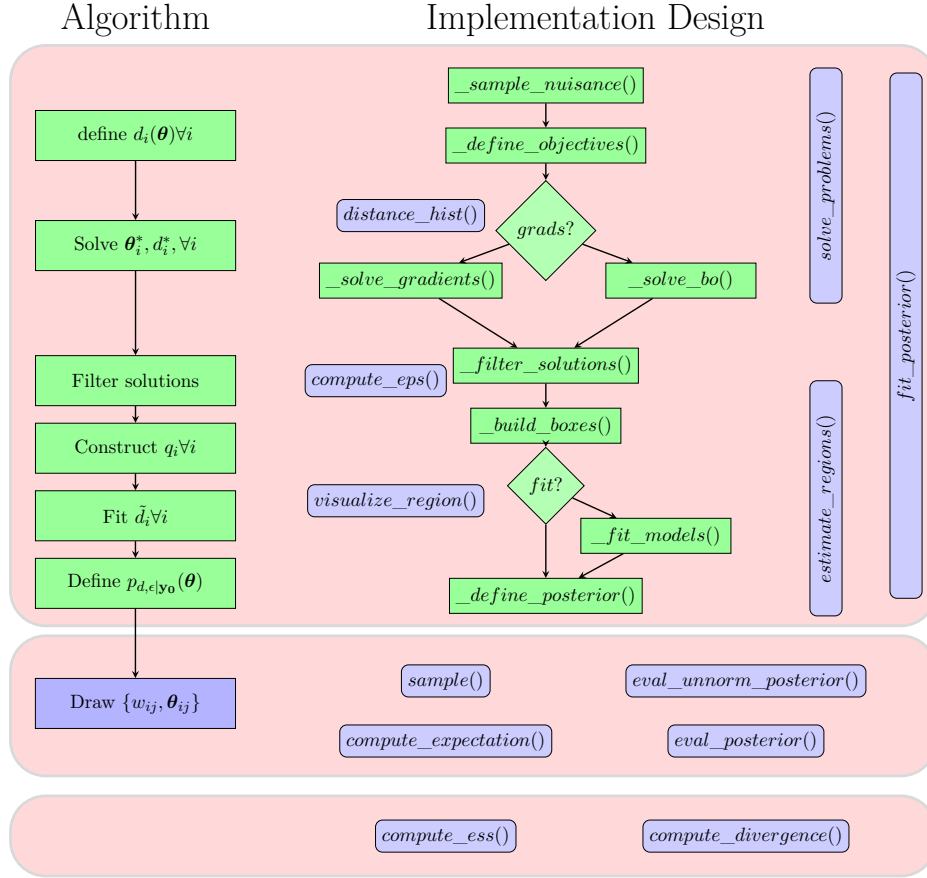
Figure 2: Overview of the ROMC implementation. On the left side, we depict ROMC as a sequence of algotirhmic steps. On the right side, we present the functions that form our implementation; the green rectangles (starting with underscore) are the internal functionalities and the blue rectangles the publicly exposed API. This side-by-side illustration highlights that our implementation follows strictly the algorithmic view of ROMC.

regions and fitting local surrogate models. The inference part comprises of evaluating the unnormalised posterior (and the normalised 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 visualising 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.[8]

Parallelising ROMC method

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

---

[8]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 (i.e. ABC Rejection Sampling), and we would like to measure the discrepancy between the two approximations.

optimisation problems, (b) constructing bounding box regions. We parallelise these processes using the built-in Python package **multiprocessing**. The specific package enables concurrency, using subprocesses instead of threads, for side-stepping the Global Interpreter (GIL). For activating the parallel version of the algorithm, the user simpy has to pass the argument `parallelize=True` when instantiating ROMC.

```python
--------------------------------- python ----------------------------------
>>> 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 (2020),

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

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

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

The prior is a uniform distribution in the range $[-2.5, 2.5]$ and the likelihood is defined at 14. The constant c is $0.5 - 0.5^4$ ensures the continuity of the pdf. 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.

In the following code snippet, we code the model at **ELFI**.

```python
--------------------------- python snippet ----------------------------
  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

  # observation
  y = 0
```

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

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

4. Implemented functionalities

In this section, we present the implementation, dividing it into three parts; at 4.1 we present
the fitting part, at 4.2 the inference part and at 4.3 the evaluation part. Finally, at 4.4 we
describe how a user may extend ROMC with its custom modules.

4.1. Training part

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

This method (a) draws the nuisance variables, (b) defines the optimisation problems and
(c) solves them using either a gradient-based optimiser or the Bayesian optimisation (B0)
scheme. The three tasks are completed sequentially, as shown in Figure 2. The definition
of the optimisation problems is performed by drawing `n1` 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. The user may select the Bayesian Optimisation scheme by setting the argument
`use_bo=True` and pass custom arguments to the optimizer through `optimize_args`.

-------------------------------- python --------------------------------
```
>>> 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 accepts all keyword arguments and forwards them to the underlying `pyplot.hist()`
function of the package **matplotlib**. In this way the user may customise some properties of
the histogram, such as the number of bins or the range of values.

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

This method estimates the proposal region around the optimal points, following Algorithm 2. By default, the distance model that will be used follows the decision from the previous step; if a gradient-based optimizer has been used, then the real distance function d will be chosen. If BO, then then the surrogate model $\hat{d}$. In case, the user wants to enforce using the original distance function d after BO, they may set `use_surrogate=False`. Finally, the option `fit_models` defines whether to fit local surrogate models $\tilde{d}$ after estimating the proposal regions.

```python
-------------------------------- python ----------------------------------
>>> romc.fit_posterior(args*)  # training in a single call
>>> romc.visualize_region(i)   # acceptance region
>>> romc.compute_eps(quantile) # estimates eps
--------------------------------------------------------------------------
```

The function `fit_posterior` is a syntactic sugar for applying `solve_problems` and `estimate_regions` into a single step. The function `visualize_region` can be used for plotting the bounding box around the optimal point, when the parameter space is up to 2D. The argument `i` is the index of the corresponding optimisation problem. Finally, `compute_eps` returns the appropriate distance value $d^*_{i=\kappa}$ where $\kappa = \lfloor quantile \cdot n \rfloor$ from the collection $\{d^*_i\} \forall i = \{1, \ldots, n\}$ 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

In the following snippet, we put together the routines described above to perform the training part at our running example.

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

  # 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=1) # for inspecting visually the bounding box

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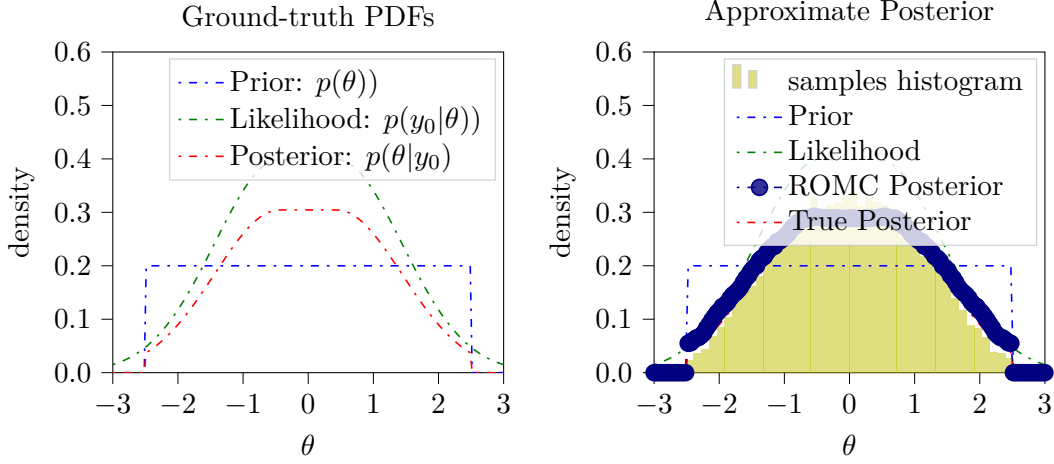
4.2. Inference part

Figure 3: Histogram of distances and visualisation of a specific region.

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

This is the basic inference utility of the ROMC implementation; we draw n_2 samples for each bounding box region. This gives a total of $k \times n_2$, where $k < n_1$ is the number of the optimal points remained after filtering[9]. 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.

```python
-------------------------------- python --------------------------------
>>> romc.compute_expectation(h) # E[h(x)|theta]
>>> romc.eval_unorm_posterior(theta, eps_cutoff=False) # unnorm p(theta)
>>> romc.eval_posterior(theta, eps_cutoff=False) # normalised p(theta)
------------------------------------------------------------------------
```

The function `compute_expectation` computes the expectation $\mathsf{E}_{p(\boldsymbol{\theta}|\mathbf{y_0})}[h(\boldsymbol{\theta})]$ using expression (11). The argument `h` can be any python `Callable`. The method `eval_unorm_posterior` computes the unnormalised posterior approximation using the expression (3). The method `eval_posterior` evaluates the normalised posterior estimating the partition function $Z = \int p_{d,\epsilon}(\boldsymbol{\theta}|\mathbf{y_0})d\boldsymbol{\theta}$; with Riemann's integral approximation. The approximation is computationally feasible only in a low-dimensional parametric space.

Example - Sampling and compute expectation

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 the utility `compute_expectation`, we demonstrate how to use it in order to

---

[9]From the n_1 optimisation problems, only the ones with $d_i(\boldsymbol{\theta}^*) < \epsilon$ are maintained for building a bounding box

compute the samples mean and variance. Finally, we evaluate `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.

```
---------------------------- python snippet -----------------------------
  # Inference part
  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: 19300
  # Sample means: theta: -0.0116

  # 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
  romc.eval_unnorm_posterior(theta=0)

  # check eval posterior
  romc.eval_posterior(theta=0)
-------------------------------------------------------------------------
```

4.3. Evaluation part

```
------------------------------- python ----------------------------------
>>> romc.compute_divergence(gt_posterior, bounds, step, distance)
>>> romc.compute_ess()
-------------------------------------------------------------------------
```

The utility `compute_divergence` estimate the divergence between the ROMC approximation and the ground truth posterior. Since the estimation is performed using the Riemann's

approximation, the method can only work in low dimensional spaces. As mentioned at the beginning of this chapter, in a real-case scenario it is not expected the ground-truth posterior to be available. In cases the ground truth posterior is not availabe (as in real-scenarios), the user may select the approximation 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"`.

The method `compute_ess` computes the Effective Sample Size (ESS) using the following expression Sudman (1967),

$$ESS = \frac{(\sum_i w_i)^2}{\sum_i w_i^2} \tag{16}$$

The ESS is a valuable measure of the **actual** sample size in cases of weighted samples. For example, there are cases where in a big population, a single sample with huge weight dominates. The ESS provides a nice metric in these cases.

```
---------------------------- python snippet ----------------------------
  # 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. Extensibility

ROMC describes a sequence of steps for approximating the posterior distribution, without explicitly enforcing the methods that solve these steps. Even though for each step a specific algorithm is proposed by Ikonomov and Gutmann (2020), in general ROMC is functional if a practitioner thinks of an alternative way of approaching a specific step. Considering this particularity, we designed the implementation to support extensibility.

We have specified four specific points where a user may intervene with their custom modules; (a) the gradient-based optimisation, (b) the Bayesian Optimisation, (c) the proposal region construction and (d) the surrogate model fitting. These are the four critical parts of the ROMC procedure, whereas the rest of the code is the backbone of the algorithm.

The four replaceable parts described above, are solved using the four methods of the `OptimisatioProblem` class; (a) `solve_gradients(**kwargs)`, (b) `solve_bo(**kwargs)`, (c) `build_region(**kwargs)`, (d) `fit_local_surrogate(**kwargs)`. The user can create a custom class that inherits the basic `OptimisatioProblem` and, then, overwrite any of the four functions with custom ones. Suppose a user wants to fit Neural Networks as local surrogate models $\tilde{d}_i$. The user should overwrite the `fit_local_surrogate(**kwargs)` function with one that internally trains a neural network. In the following snippet we illustrate that, using the `neural_network.MLPRegressor` class of the **scikit-learn** package.

```python
-------------------------------- 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

        # 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 the same way, the user may replace each of the other three functionalities. The only restriction that must be respected, concerns the side-effects each method has at the `OptimisatioProblem` class level. In the following four snippets we present the signature of each method. We name the class-level variables that must be set under the comment `# side-effects`.

```python
-------------------------------- python ---------------------------------
def solve_gradients(self, **kwargs):
  # custom solution procedure
  x = ...
  y = ...
  jac = ...
  hess_inv = ...
```

```
  # side-effects
  self.state["attempted"] = True
  if success:
    self.result = RomcOpimisationResult(x, y, jac, hess_inv)
    self.state["solved"] = True
    return True
  else:
    return False
--------------------------------------------------------------------------------
def solve_bo(self, **kwargs):
  # custom procedure
  x = ...
  y = ...
  custom_surrogate = ...

  # side-effects
  self.state["attempted"] = True
  if success:
    self.result = RomcOpimisationResult(x, y)
    self.surrogate = custom_surrogate
    self.state["solved"] = True
    self.state["has_fit_surrogate"] = True
    return True
  else:
    return False
--------------------------------------------------------------------------------
def build_region(self, **kwargs):
  # custom build_region method
  bounding_box: List[NDimBoudningBox] = ...
  success = True/False # whether region built correctly

  # side-effects
  self.eps_region = eps_region
  if success:
    # construct region
    self.regions = bounding_box
    self.state["region"] = True
    return True
  else:
    return False
--------------------------------------------------------------------------------
def fit_local_surrogate(self, **kwargs):
  # custom local surrogates
  custom_surrogates = ...
  success = True/False # whether local surrogates fit correctly
```

```
  # side-effects
  if success:
    self.local_surrogate = local_surrogates
    self.state["local_surrogates"] = True
    return True
  else:
    return False
-----------------------------------------------------------------------------
```

The two classes that may be needed for creating the custom routines are
(a) `RomcOpimisationResult` and (b) `NDimBoundingBox`. We present their signatures below.

```python
-------------------------------- python --------------------------------
class RomcOpimisationResult:
    def __init__(self, x_min, f_min, hess_appr):
        Parameters
        ----------
        x_min: np.ndarray (D,) or float, the minimum point
        f_min: float, distance at the minimum point
        hess_appr: np.ndarray (DxD), Hessian approximation at x_min
        """
-----------------------------------------------------------------------------
class NDimBoundingBox:
    def __init__(self, rotation, center, limits, eps_region):
        Parameters
        ----------
        rotation: np.array (D,D),  rotation matrix for the bounding box
        center: np.array (D,) center of the bounding box
        limits: np.ndarray, shape: (D,2), the limits of the bounding box
        eps_region: float, distance threshold
-----------------------------------------------------------------------------
```

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) using a gradient-based optimiser, (ii) using the Bayesian Optimisation scheme 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 achieved by the parallelised version of ROMC.

Model Definition

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

$$y_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2}, \quad t = 1, \ldots, T \tag{17}$$

$$\theta_1, \theta_2 \in \mathbb{R}, \quad w_k \sim \mathcal{N}(0,1), k \in \mathbb{Z} \tag{18}$$

The r.v. $w_k \sim \mathcal{N}(0,1)$ is white noise and the two parameters of interest, θ_1, θ_2, model the dependence from the previous observations. The number of sequential observations T is a constant and 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, Pudlo, Robert, and Ryder (2012),

$$p(\boldsymbol{\theta}) = p(\theta_1)p(\theta_2|\theta_1) = \mathcal{U}(\theta_1; -2, 2)\mathcal{U}(\theta_2; \theta_1 - 1, \theta_1 + 1) \tag{19}$$

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 ouptput vector represents a time-series signal, we select the autocovariances with lag = 1 and lag = 2, as shown in equations (20) and (21). The final distance node is the squared Euclidean distance (23).

$$s_1(\mathbf{y}) = \frac{1}{T-1} \sum_{t=2}^{T} y_t y_{t-1} \tag{20}$$

$$s_2(\mathbf{y}) = \frac{1}{T-2} \sum_{t=3}^{T} y_t y_{t-2} \tag{21}$$

$$s(\mathbf{y}) = (s_1(\mathbf{y}), s_2(\mathbf{y})) \tag{22}$$

$$d = ||s(\mathbf{y}) - s(\mathbf{y}_0)||_2^2 \tag{23}$$

Inference

In order to show all the capabilities of the implementation, we perform the inference (i) using the gradient-based optimizer, (ii) using the Bayesian Optimisation scheme and (iii) fitting a Neural Network (NN) as a surrogate model. We use the Rejection ABC algorithm for evaluating all approaches. Replacing the typical quadratic surrogate model with a NN serves as an illustrator of the extensibility of our implementation. The replacement is done by coding a custom optimisation function with a surrogate model of our own preference, as shown in Chapter 4.4. For the definition of the NN, we use the **MLPRegressor** class of the **scikit-learn** package. Therefore, the NN substitutes the real distance function d_i inside the proposal region q_i $\forall i$ at the inference phase i.e. sampling, computing an expectation and evaluating the posterior. In our example we use a neural network of two hidden layers of 10 neurons each and we train it sampling 500 examples from each proposal region.

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

In Figure 5, we demonstrate the histograms of the marginal posteriors, for each approach; (a) Rejection ABC (first column), (b) ROMC with gradient-based optimisation (second column) (c) ROMC with Bayesian optimisation (third column) and (d) ROMC with the NN

	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)	0.501	0.142	0.033	0.169
ROMC (Bayesian optimisation)	0.494	0.16	0.086	0.167
ROMC (Neural Network)	0.491	0.138	0.04	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.

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 unnormalised posterior for the three different variations of the ROMC method. The results show that all ROMC variations provide consistent results between them and in comparison with the standard Rejection ABC algorithm.
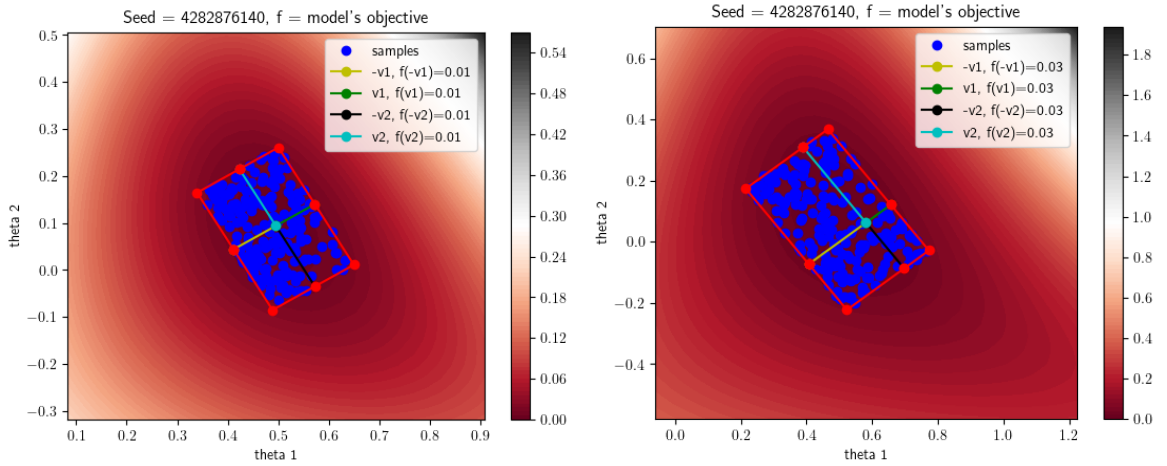
Figure 4: The acceptance region in a specific optimisation problem. In the left figure the region obtained with gradient-based optimiser and in the right one with Bayesian Optimisation.

Parallelisation

As stated above, ROMC is a ridiculously parallelisable method. Therefore, it is straight-forward to parallelise the fitting part, i.e. (i) solving the optimisation problems and (ii) estimating the proposal regions. Our implementation supports exploiting all the available CPU cores through the built-in Python package **multiprocess**[10]. 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.

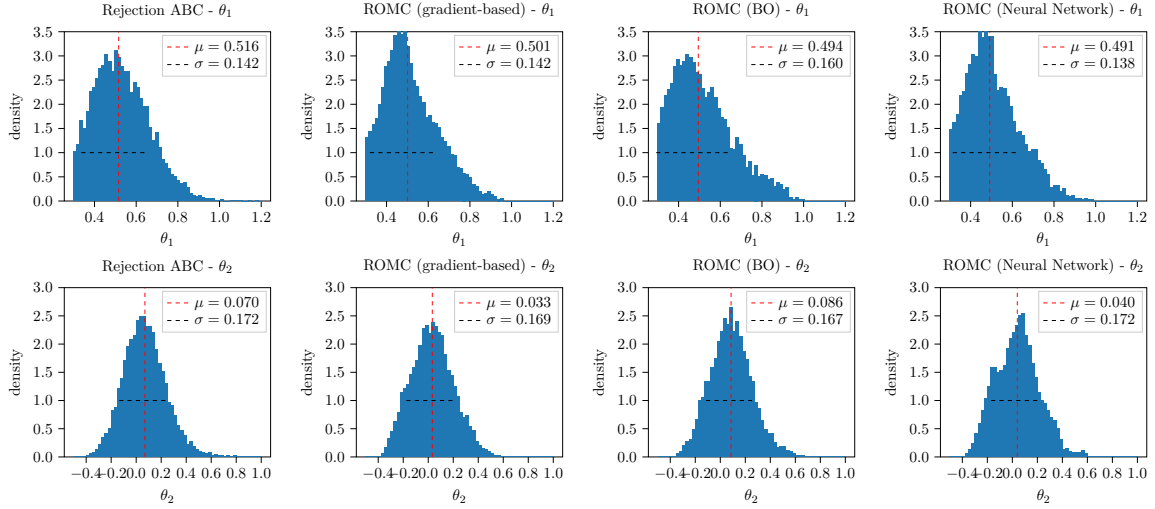[10]https://docs.python.org/3/library/multiprocessing.html

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 optimiser and (c) in the third row, using ROMC with Bayesian optimisation approach. The vertical (red) line represents the samples mean μ and the horizontal (black) the standard deviation σ.

6. Summary and discussion

In this paper, we presented the implementation details we followed for developing the LFI method ROMC at the **ELFI** package. We paid thorough attention to two specific use-case scenarios. Firstly, we illustrate how a user may take advantage of our ready-to-use API for solving its LFI problem. Secondly, we focus on the scenario where a researcher wants to intervene and alter parts of the method. Our implementation is designed to support this as well.

There are still open challenges for the left for future research. Two directions may enable ROMC to solve high-dimensional problems efficiently. The first one is enabling ROMC's execution into 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 one refers to implementing the method in a framework that supports automatic differentiation. Automatic differentiation is necessary for efficiently solving optimisation problems, especially in high-dimensional parametric models.

Computational details

The results in this paper were obtained using Python 3.7.9 with the **ELFI** 0.8.3 package. The experiments have been executed in a single machine with an Intel® Core™ i7-8750H Processor an with Ubuntu 20.04 lts operating system.
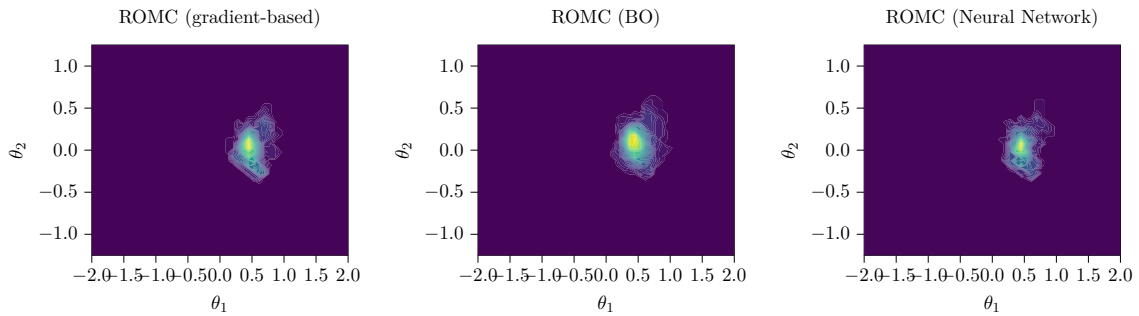
Figure 6: The unnormalised posterior distribution using the ROMC method with (a) a gradient-based optimisation (b) Bayesian Optimisation (c) gradient-based with a Neural Network as a surrogate model.
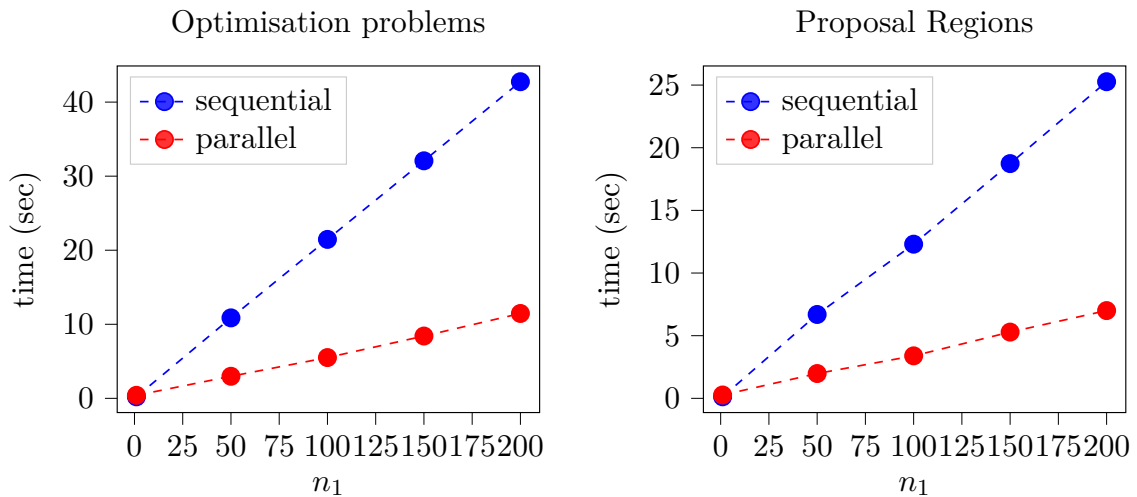

Figure 7: Comparison between parallel and sequential execution of ROMC. We observe that the parallel version runs almost 5 times faster.

Acknowledgments

References

Beaumont MA, Zhang W, Balding DJ (2002). "Approximate Bayesian Computation in Population Genetics." *Genetics*, **162**(4), 2025–2035. ISSN 1943-2631. doi:10.1093/genetics/162.4.2025. https://academic.oup.com/genetics/article-pdf/162/4/2025/42049447/genetics2025.pdf, URL https://doi.org/10.1093/genetics/162.4.2025.

Blum M, Francois O (2010). "Non-linear regression models for Approximate Bayesian Computation." *Statistics and Computing*, **20**(1), 63–73. ISSN 0960-3174. URL http://dx.doi.org/10.1007/s11222-009-9116-0.

Chen Y, Gutmann MU (2019). "Adaptive Gaussian Copula ABC." In K Chaudhuri, M Sugiyama (eds.), *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pp. 1584–1592. PMLR. URL https://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*.

Forneron JJ, Ng S (2016). *A Likelihood-Free Reverse Sampler of the Posterior Distribution*, volume 36, pp. 389–415. Emerald Publishing Ltd. ISBN 978-1-78560-787-5. doi:10.1108/S0731-905320160000036020. URL https://EconPapers.repec.org/RePEc:eme:aecozz:s0731-905320160000036020.

Gutmann MU, Corander J (2016). "Bayesian Optimization for Likelihood-Free Inference of Simulator-Based Statistical Models." *Journal of Machine Learning Research*, **17**(125), 1–47. Http://arxiv.org/abs/1501.03291, URL http://jmlr.org/papers/v17/15-017.html.

Hagberg A, Swart P, S Chult D (2008). "Exploring network structure, dynamics, and function using NetworkX."

Hermans J, Begy V, Louppe G (2020). "Likelihood-free MCMC with Amortized Approximate Ratio Estimators." In *Proceedings of the thirty-seventh International Conference on Machine Learning (ICML)*.

Ikonomov B, Gutmann MU (2020). "Robust Optimisation Monte Carlo." In S Chiappa, R Calandra (eds.), *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pp. 2819–2829. PMLR. URL https://proceedings.mlr.press/v108/ikonomov20a.html.

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. ISSN 1063-5157. doi:10.1093/sysbio/syw077. URL http://dx.doi.org/10.1093/sysbio/syw077.

Lintusaari J, Vuollekoski H, Kangasrääsiö A, Skytén K, Järvenpää M, Marttinen P, Gutmann M, Vehtari A, Corander J, Kaski S (2018). "ELFI: Engine for Likelihood Free Inference." arXiv:1708.00707.

Marin JM, Pudlo P, Robert C, Ryder R (2012). "Approximate Bayesian computational methods." *Statistics and Computing.* doi:10.1007/s11222-011-9288-2.

Meeds T, Welling M (2015). "Optimization Monte Carlo: Efficient and Embarrassingly Parallel Likelihood-Free Inference." In C Cortes, N Lawrence, D Lee, M Sugiyama, R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2015/file/a284df1155ec3e67286080500df36a9a-Paper.pdf.

Papamakarios G, Murray I (2016). "Fast epsilon-free Inference of Simulation Models with Bayesian Conditional Density Estimation." In DD Lee, M Sugiyama, UV Luxburg, I Guyon, R Garnett (eds.), *Advances in Neural Information Processing Systems 29*, pp. 1028–1036. Curran Associates, Inc. URL http://papers.nips.cc/paper/6084-fast-free-inference-of-simulation-models-with-bayesian-conditional-density-estimat

Papamakarios G, Sterratt D, Murray I (2019). "Sequential Neural Likelihood: Fast Likelihood-free Inference with Autoregressive Flows." In K Chaudhuri, M Sugiyama (eds.), *Proceedings of Machine Learning Research*, volume 89, pp. 837–848. PMLR, Proceedings of Machine Learning Research.

Shahriari B, Swersky K, Wang Z, Adams RP, de Freitas N (2016). "Taking the Human Out of the Loop: A Review of Bayesian Optimization." *Proceedings of the IEEE*, **104**(1), 148–175. doi:10.1109/JPROC.2015.2494218.

Sisson S, Fan Y, Beaumont M (eds.) (2018). *Handbook of Approximate Bayesian Computation.* Chapman and Hall/CRC Press.

Sudman S (1967). " Survey Sampling. Leslie Kish ." *American Journal of Sociology.* ISSN 0002-9602. doi:10.1086/224359.

Thomas O, Dutta R, Corander J, Kaski S, Gutmann MU (2020). "Likelihood-Free Inference by Ratio Estimation." *Bayesian Analysis*, (advance publication). doi:https://doi.org/10.1214/20-BA1238. URL https://projecteuclid.org/euclid.ba/1599876022.

Wood SN (2010). "Statistical inference for noisy nonlinear ecological dynamic systems." *Nature*, **466**(7310), 1102–1104. ISSN 0028-836. URL http://dx.doi.org/10.1038/nature09319.

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