Journal of Statistical Software

November 2021, Volume 100, Issue 7. doi: 10.18637/jss.v100.i07

ABCpy: A High-Performance Computing Perspective to Approximate Bayesian Computation

Ritabrata Dutta
University of Warwick

Marcel Schoengens
ETH Zurich

Lorenzo Pacchiardi
University of Oxford

Avinash Ummadisingu
Università della Svizzera italiana

Nicole Widmer
ETH Zurich

Pierre Künzli
University of Geneva

Jukka-Pekka Onnela
Harvard University

Antonietta Mira
Università della Svizzera italiana
Università dell'Insubria

Abstract

ABCpy is a highly modular scientific library for approximate Bayesian computation (ABC) written in Python. The main contribution of this paper is to document a software engineering effort that enables domain scientists to easily apply ABC to their research without being ABC experts; using **ABCpy** they can easily run large parallel simulations without much knowledge about parallelization. Further, **ABCpy** enables ABC experts to easily develop new inference schemes and evaluate them in a standardized environment and to extend the library with new algorithms. These benefits come mainly from the modularity of **ABCpy**. We give an overview of the design of **ABCpy** and provide a performance evaluation concentrating on parallelization. This points us towards the inherent imbalance in some of the ABC algorithms. We develop a dynamic scheduling **MPI** implementation to mitigate this issue and evaluate the various ABC algorithms according to their adaptability towards high-performance computing.

Keywords: ABC, HPC, **Spark**, **MPI**, parallel, imbalance, Python library.

1. Introduction

Today, computers are used to simulate different aspects of nature. Natural scientists traditionally hypothesize models underlying natural phenomena. As a running example throughout

the paper, we will consider a popular weather prediction benchmark model known as the Lorenz95 model (Lorenz 1995), which represents an idealized weather system with two sets of variables, the former evolving slowly in time and the latter evolving much faster. The evolution follows a set of differential equations, and each of the slow variables is coupled to three neighbor ones and to a subset of the fast variables (that outnumber the slow ones), and similarly for the evolution of the fast variables. We will focus on a stochastic modification of the original model due to Wilks (2005), in which the fast variables are unobserved and their effect on the slow variables is replaced by a stochastic forcing term; see Appendix A for more details. The implementation of the model is a discrete time integration of the set of stochastic differential equations, each integration of the model corresponding to a possible trajectory with a finite timestep. The equations depend on a set of parameters collectively called θ, on which we want to perform inference given an observation. Therefore, denoting the model by $\boldsymbol{\mathcal{M}}$ and the observed slow variables at timestep t by $y^{(t)}$, an integration of the model yields:

$$\boldsymbol{\mathcal{M}}(y^{(0)}, \theta) \rightarrow \{y^{(t)},\ t = 1, \ldots, T\},$$

where the initial configuration $y^{(0)}$ is assumed to be known. Simulator-based models as the above one[1] are used in a wide range of scientific disciplines to simulate different aspects of nature, ranging from dynamics of sub-atomic particles (Martinez *et al.* 2016) to evolution of human societies (Turchin, Currie, Turner, and Gavrilets 2013) and formation of universes (Schaye *et al.* 2015).

However, often the true parameter θ^0 of simulator-based models is not known. If the true parameter value could be learned rigorously in a data-driven manner, we could substantially improve the accuracy of these models. Consider the problem of estimating the true value and quantifying uncertainty in θ based on an observed dataset $\boldsymbol{y}_0$, e.g., in the Lorenz95 model $\boldsymbol{y}_0 \equiv \{y_0^{(t)},\ t = 1, \ldots, T\}$. A further extension of this inferential problem is the selection of a model, given an observed dataset, from a set of possible models. Traditional methods in statistics can infer, from the observed data, model and corresponding parameters and quantify the associated uncertainty only when the data generating mechanism has a known likelihood function. In many cases, however, we may not have access to an explicit formula for the latter or, if we have, its evaluation can be too computationally expensive; for instance, if the data generating model consists of the integration of a set of stochastic differential equations (as in the Lorenz95 model above), there is no easy way to evaluate the likelihood of each integration for a set of parameter values. Alternatively, it can be that the likelihood depends on the inversion of a high-dimensional covariance matrix, which can be very costly.

In the above scenarios, approximate Bayesian computation (ABC; Tavaré, Balding, Griffiths, and Donnelly 1997; Pritchard, Seielstad, Perez-Lezaun, and Feldman 1999; Beaumont, Zhang, and Balding 2002) can still offer a way to perform sound statistical inference, e.g., point estimation, hypothesis testing, and model selection. ABC methods infer parameters by first simulating a dataset using a proposed parameter value and accepting or rejecting that parameter value either by comparing the closeness of the simulated dataset to the observed dataset, usually through the use of summary statistics, or by approximating the likelihood function using simulated datasets (Wood 2010; Thomas, Dutta, Corander, Kaski, and Gutmann 2021).

[1] In this manuscript, we will use the term simulator-based model to refer to a model that enables direct simulation of model outcomes using a set of stochastic rules. This term is well established within the ABC literature, but we point out that these types of models are sometimes called mechanistic models or agent based models in different fields of science.

We direct interested readers to the review paper by Lintusaari, Gutmann, Dutta, Kaski, and Corander (2016).

The necessity to simulate datasets from simulator-based models makes ABC algorithms extremely expensive when this forward simulation itself is costly. Applications of ABC algorithms to complex problems show the necessity of adapting them to high-performance computing (HPC) facilities and developing an ecosystem where new ABC algorithms can be investigated while respecting the architecture of existing computing facilities. ABC and HPC were first brought together in the **ABC-sysbio** package of Liepe *et al.* (2010) for the systems biology community, where the sequential Monte Carlo ABC (SMCABC) algorithm (Toni, Welch, Strelkowa, Ipsen, and Stumpf 2009) was efficiently parallelized using graphics processing units (GPUs).

Our goal is to overcome the need for users to have knowledge of parallel programming, as is required for using **ABC-sysbio**, and also to make a software package available for scientists across domains. These objectives were partly addressed by parallelization of SMCABC using **MPI**/**OpenMPI** (Stram, Marjoram, and Chen 2015), and by making SMCABC available for the astronomical community (Jennings and Madigan 2017). Regardless of these advances, a recent ABC review article (Lintusaari *et al.* 2016) highlights the depth and breadth of available ABC algorithms, which can be made efficient via parallelization using an HPC environment (Kulakova, Angelikopoulos, Hadjidoukas, Papadimitriou, and Koumoutsakos 2016; Chiachio, Beck, Chiachio, and Rus 2014). These developments emphasize the need of a generalized HPC supported platform for efficient ABC algorithms, which can be parallelized on multi-processor computers or computing clusters and is accessible to a broad range of scientists.

We address the need for a user-friendly scientific library for ABC algorithms by introducing **ABCpy**, which is written in Python (Van Rossum *et al.* 2021) and designed in a highly modular fashion. Most existing ABC software suites are mainly domain-specific and optimized for a narrower class of problems. Our main goal was to make **ABCpy** modular, which makes it intuitive to use and easy to extend. Further, it enables users to run ABC sampling schemes in parallel without too much re-factoring of existing code. **ABCpy** includes likelihood free inference schemes, both based on discrepancy measures and approximate likelihood, providing a complete environment to develop new ABC algorithms. The source code can be downloaded from `https://github.com/eth-cscs/abcpy`.

For parallelization of ABC algorithms, we use the map-reduce paradigm. This choice was motivated by our experience that ABC algorithms are usually parallelizable in a loosely coupled fashion. Additionally, opting for map-reduce we were able to implement parallelization backends in two different frameworks (namely, Apache **Spark**, Zaharia *et al.* 2016, and **MPI**, Message Passing Interface Forum 2012), that target the needs of two different but important communities (correspondingly, industry users and researchers). Thus, the choice of map-reduce increases the user's flexibility given widely available commercial cloud computing facilities. In Section 4.1 we discuss in detail the reasons for these choices.

Of particular interest to practitioners might be the MPI backend since in contrast to **Spark**, **MPI** is a low level communication framework without sophisticated task scheduling facilities. A straightforward **MPI** implementation can therefore result in load imbalance between the different workers for the ABC algorithms. To handle this, we use a greedy approach to dynamically allocate map tasks to workers in our **MPI** backend. More details on this can be found in Section 5.2.

We give a brief description of ABC (Section 2) and of the structure of the software suite **ABCpy** (Section 3) with a specific focus on modularity (Section 4) and parallelism. Section 5 deals with the different map-reduce implementations available through **ABCpy** and a detailed comparison of the speed-up and efficiency for ABC algorithm using the Lorenz95 model; specifically, the scalability of different ABC algorithms is compared in Section 5.3. Finally, we compare our package with similar ones in Section 6, where we also give a detailed overviews of the most important features that our package implements that are not available in any other up to now, namely the possibility of automatically learn summary statistics, the handling of co-occurring datasets, the use of nested parallelization and the diagnostic checks. We conclude in Section 7 with some final remarks.

2. ABC

We can quantify the uncertainty of the unknown parameter θ by a posterior distribution $p(\theta \mid \boldsymbol{y})$ given the observed dataset $\boldsymbol{y} = \boldsymbol{y}_0$. A posterior distribution can be written, by Bayes' Theorem, as:

$$p(\theta \mid \boldsymbol{y}) = \frac{\pi(\theta)p(\boldsymbol{y} \mid \theta)}{m(\boldsymbol{y})}, \tag{1}$$

where $\pi(\theta)$, $p(\boldsymbol{y} \mid \theta)$ and $m(\boldsymbol{y}) = \int \pi(\theta)p(\boldsymbol{y} \mid \theta)d\theta$ are, correspondingly, the prior distribution on the parameter θ, the likelihood function, and the marginal likelihood. The prior distribution $\pi(\theta)$ ensures a way to leverage the learning of parameters with prior knowledge. If the likelihood function can be evaluated, at least up to a normalizing constant, then the posterior distribution can be approximated by drawing a sample of parameter values using (Markov chain) Monte Carlo sampling schemes (Robert and Casella 2005). In many real-world problems, however, the analytic form of the posterior distribution is unknown because the likelihood is not analytically available. This is typical for simulator-based models for which the likelihood function is often intractable or difficult to compute (as for instance the Lorenz model above or other integrations of stochastic differential equation models), and therefore the inference schemes are adapted following two alternative approaches: (i) by measuring the discrepancy between simulated and observed dataset, and (ii) by approximating the likelihood function.

2.1. Measuring discrepancy

In the simplest ABC implementation we forward simulate from the model, $p(\boldsymbol{y} \mid \theta)$, producing a synthetic dataset $\boldsymbol{y}_{\text{sim}}$ for a given parameter value θ, and measure the closeness between $\boldsymbol{y}_{\text{sim}}$ and $\boldsymbol{y}_0$ using a pre-defined discrepancy function $\rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0)$. Based on this discrepancy measure, ABC accepts the parameter value θ when $\rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0)$ is less than a pre-specified threshold value ϵ. This simple algorithm will be referred to as rejection ABC (RejectionABC). A review of different methods based on discrepancy can be found in Marin, Pudlo, Robert, and Ryder (2012) and Lintusaari *et al.* (2016); in Section 2.3, we briefly describe those which we implement in **ABCpy**.

To implement any ABC sampling scheme, we need to define how to measure the discrepancy between $\boldsymbol{y}_{\text{sim}}$ and $\boldsymbol{y}_0$. As the dataset can be of varied type and complexity (e.g., high-dimensional time-series or network data), in practice discrepancies are measured using informative summary statistics extracted from the dataset. We therefore need to define two

functions: one for computing the summary statistics from the dataset, and one for measuring the discrepancy between them. From now on, we will denote these two functions as *statistics* and *distance*, which need to be defined by the user and are problem specific.

For illustration and comparison, in this paper we will consider the Lorenz95 model for numerical weather prediction (Lorenz 1995; Wilks 2005) with a stochastic modification, as discussed above. For this model, a possible choice of *statistics* are the summary statistics suggested in Hakkarainen *et al.* (2012) called `HakkarainenLorenz` (details in Appendix A), while we can use as *distance* the `Euclidean` distance. Besides the latter, **ABCpy** also implements distances based on logistic regression (`LogReg`) and penalized logistic regression (`PenLogReg`) classifiers (Gutmann, Dutta, Kaski, and Corander 2018); both work by fitting the classifier to distinguish between observed datasets and datasets generated from the model with a fixed parameter value and by using the resulting classification accuracy as discrepancy measure. Finally, we also provide a `Wasserstein` distance (Peyré and Cuturi 2019). Specifically, if several independent and identically distributed (i.i.d.) observations are available, the latter can be used as in Bernton, Jacob, Gerber, and Robert (2019), by generating i.i.d. simulations from the model for each parameter value and afterwards computing the Wasserstein distance between the empirical distributions defined by the observed and synthetic dataset.

2.2. Approximate likelihood

The second approach is based on directly approximating the likelihood function at θ, up to a constant, using the data, $\boldsymbol{y}_{\text{sim}}$, simulated for that given parameter value θ. Following the pseudo-marginal likelihood idea of Andrieu and Roberts (2009), an unbiased approximation of the likelihood function can then be used in a traditional Monte Carlo sampling scheme to sample from the posterior distribution.

Similarly to the scheme described in Section 2.1, to perform any approximate likelihood based sampling scheme we need to define two functions. We require the *statistics* function and, additionally, we need a function to compute the approximate likelihood based on the extracted summary statistics from $\boldsymbol{y}_{\text{sim}}$. We denote this function by `approx_lhd` and the user needs to choose from one of the three currently available implementations of `approx_lhd` in **ABCpy**:

- Synthetic likelihood (`SynLikelihood`, Wood 2010), which works by assuming the statistics to have a multivariate normal likelihood and by estimating the mean and covariance parameters from $\boldsymbol{y}_{\text{sim}}$.

- Semiparametric synthetic likelihood (`SemiParametricBSL`, An, Nott, and Drovandi 2020), which is an extension of the above in which the likelihood of the summary statistics is represented as the product of a Gaussian copula (whose parameters are estimated from $\boldsymbol{y}_{\text{sim}}$) and univariate marginals obtained with kernel density estimates.

- Penalized logistic regression (`PenLogReg`, Thomas *et al.* 2021), which instead builds a likelihood approximation by fitting a probabilistic classifier between data generated from the model for a fixed parameter value and data generated from the marginal $p(x)$; this in fact approximates the ratio $\frac{p(x|\theta)}{p(x)}$, which is proportional to the likelihood with respect to θ.

Algorithm 1 Population Monte Carlo ABC (PMCABC) algorithm for generating N samples from the approximate posterior distribution. Here $K_t(\cdot \mid \theta, \Sigma_{t-1})$ is the perturbation kernel, and `weighted-Covariance` (not shown here) updates the covariance matrix of the perturbation kernel according to the drawn samples and weights.

Require: Specify $q_\epsilon \in [0, 100]$ and a decreasing sequence of thresholds $\epsilon_1 \geq \epsilon_2 \geq \cdots \geq \epsilon_T$ for T iterations.

for $i = 1$ **to** N **do**
  repeat
    Generate θ from the prior $\pi(\cdot)$
    Generate $\boldsymbol{y}_{\text{sim}}$ from $\boldsymbol{\mathcal{M}}$ using θ
  until $\rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0) \leq \epsilon_1$
  $d^{(i)} = \rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0)$
  $\theta_1^{(i)} \leftarrow \theta$
  $\omega_1^{(i)} \leftarrow 1/N$
end for
$\Sigma_1 \leftarrow 2 * \texttt{weighted-Covariance}(\theta_1, \omega_1)$
for $t = 2$ **to** T **do**
  $\epsilon_t = \max(q_\epsilon\text{-th percentile of } d,\ \epsilon_t)$
  for $i = 1$ **to** N **do**
    repeat
      Draw θ^* from among θ_{t-1} with probabilities ω_{t-1}
      Generate θ from $K_t(\theta^*, \Sigma_{t-1})$
      Generate $\boldsymbol{y}_{\text{sim}}$ from $\boldsymbol{\mathcal{M}}$ using θ
    until $\rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0) \leq \epsilon_t$
    $d^{(i)} = \rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0)$
    $\theta_t^{(i)} \leftarrow \theta$
    $\omega_t^{(i)} \leftarrow \pi(\theta)/(\sum_{k=1}^{N} \omega_{t-1}^{(k)} K_t(\theta \mid \theta_{t-1}^{(k)}, \Sigma_{t-1}))$
  end for
  Normalize $\omega_t^{(i)}$ over $i = 1, \ldots, N$
  $\Sigma_t \leftarrow 2 * \texttt{weighted-Covariance}(\theta_t, \omega_t)$
end for

2.3. Implemented algorithms

In **ABCpy**, besides the standard RejectionABC algorithm, we implement widely used and advanced variants, namely: population Monte Carlo ABC (PMCABC, Beaumont 2010; Toni *et al.* 2009), sequential Monte Carlo ABC (SMCABC, Del Moral, Doucet, and Jasra 2012), replenishment sequential Monte Carlo ABC (RSMCABC, Drovandi and Pettitt 2011), adaptive population Monte Carlo ABC (APMCABC, Lenormand, Jabot, and Deffuant 2013), ABC with subset simulation (ABCsubsim, Chiachio *et al.* 2014), and simulated annealing ABC (SABC, Albert, Künsch, and Scheidegger 2015). **ABCpy** also includes a parallelized version of a random forest ensemble model selection algorithm (Pudlo, Marin, Estoup, Cornuet, Gautier, and Robert 2015).

In Algorithm 1, we provide a description of the PMCABC algorithm, which we will use in the following to illustrate the idea of ABC algorithms and their parallelization. In a nutshell, PM-

Algorithm 2 PMC algorithm using an approximate likelihood function and producing N samples from the approximate posterior distribution. Here $K_t(\cdot \mid \theta, \Sigma_{t-1})$ is the perturbation kernel, and `weighted-Covariance` (not shown here) updates the covariance matrix of the perturbation kernel according to the drawn samples and weights.

Require: Specify $\hat{L}_{\boldsymbol{y}_{\text{sim}}}(\cdot \mid \theta)$ function to evaluate approximate likelihood function at θ using simulated data $\boldsymbol{y}_{\text{sim}}$.

1: **for** $i = 1$ **to** N **do**
2: Generate θ from the prior $\pi(\cdot)$
3: Generate $\boldsymbol{y}_{\text{sim}}$ from $\boldsymbol{\mathcal{M}}$ using θ
4: $\theta_1^{(i)} \leftarrow \theta$
5: $\omega_1^{(i)} \leftarrow \pi(\theta)\hat{L}_{\boldsymbol{y}_{\text{sim}}}(\boldsymbol{y}_0 \mid \theta)$
6: **end for**
7: $\Sigma_1 \leftarrow 2 * \texttt{weighted-Covariance}(\theta_1, \omega_1)$
8: **for** $t = 2$ **to** T **do**
9: **for** $i = 1$ **to** N **do**
10: Draw θ^* from among θ_{t-1} with probabilities ω_{t-1}
11: Generate θ from $K_t(\theta^*, \Sigma_{t-1})$
12: Generate $\boldsymbol{y}_{\text{sim}}$ from $\boldsymbol{\mathcal{M}}$ using θ
13: $\theta_t^{(i)} \leftarrow \theta$
14: $\omega_t^{(i)} \leftarrow \pi(\theta)\hat{L}_{\boldsymbol{y}_{\text{sim}}}(\boldsymbol{y}_0 \mid \theta)/(\sum_{k=1}^{N} \omega_{t-1}^{(k)} K_t(\theta \mid \theta_{t-1}^{(k)}, \Sigma_{t-1}))$
15: **end for**
16: Normalize $\omega_t^{(i)}$ over $i = 1, \ldots, N$
17: $\Sigma_t \leftarrow 2 * \texttt{weighted-Covariance}(\theta_t, \omega_t)$
18: **end for**

CABC considers a set of sample points on the parameter space (particles) which evolve across iterations. At iteration t, the position of each particle is perturbed with a *perturbation kernel*, and then simulations from the model are run until the simulation matches the observation at some level of tolerance ϵ_t, according to the considered distance and statistics. The sequence of thresholds $\epsilon_1, \epsilon_2, \ldots$ need to be decreasing to ensure convergence of the algorithm; they can be either fixed a priori by the user or defined as some quantile of the distances at previous iteration. Similarly, the other algorithms which **ABCpy** implements (except for RejectionABC) follow the idea of considering a set of particles and evolving them across iterations, but differ in how the particles are perturbed from one iterations to the next (which is linked to drawing simulations from the model) and how the importance weights are computed. Some of the most recent algorithms (such as SABC and APMCABC) usually provide faster convergence to the posterior distribution and a smaller number of simulations required.

We can however classify the above algorithm into two groups, based on how simulations from the model are run. In one group, algorithms have an explicit acceptance step similar to Lines 2–5 of PMCABC (see Algorithm 1), where we keep simulating $\boldsymbol{y}_{\text{sim}}$ until the condition $\rho(\boldsymbol{y}_{\text{sim}}, \boldsymbol{y}_0) < \epsilon$ (for an adaptively chosen threshold ϵ), is met and the perturbed parameter is accepted. By enforcing this explicit acceptance for each perturbed parameter, we have a theoretical warranty that the accepted parameters are drawn from an approximate posterior distribution indexed by the chosen threshold ϵ. For the second group of algorithms, we do not impose explicit acceptance but we rather use a probabilistic acceptance, in which we accept

the perturbed parameter with a probability that depends on ϵ; if it is not accepted, we keep the present value of the parameter. The algorithms belonging to the explicit acceptance group are RejectionABC and PMCABC, whereas the algorithms in the probabilistic acceptance group are SMCABC, RSMCABC, APMCABC, SABC and ABCsubsim. Note that algorithms with an explicit acceptance step are usually much less efficient computationally, although they come with more theoretical guarantees. In fact, for each iteration you may need to perform the forward simulations many times, so that there is no way to know in advance how much time the algorithm will take overall.

In the same way as PMCABC, all sequential sampling schemes exploit a perturbation kernel to explore the parameter space. In **ABCpy**, we usually refer to this simply as *kernel*; **ABCpy** implements a multivariate Normal or multivariate Student's-T for continuous variables, and a random walk kernel for discrete ones. It is also possible to specify different kernel functions for different subsets of the parameters, as described in Section 6.4.

We also implement the population Monte Carlo (PMC, Cappé, Guillin, Marin, and Robert 2004) and the standard Metropolis-Hastings Markov Chain Monte Carlo (Hastings 1970) sampling schemes to be used with the different likelihood approximations discussed in Section 2.2. A detailed description of PMC algorithm is provided in Algorithm 2. Note that, similarly to sequential ABC algorithms, PMC sampling scheme also uses a perturbation kernel to explore the parameter space (Line 11 in Algorithm 2).

3. ABCpy

First we give a brief overview of how the **ABCpy** package works and how it is used. Note that **ABCpy** is under active development and thus the presented API is prone to changes. All coded examples work against major version 0.5.x and 0.6.x of **ABCpy**. As described in Section 2, the fundamental components required by ABC methods are:

- Observed data $\boldsymbol{y}_0$.
- Simulator-based model $\boldsymbol{\mathcal{M}}$.
- Prior distribution $\pi(\theta)$.
- Summary statistics.
- Discrepancy measure (*distance*) or approximate likelihood function (*approx_lhd*).

Though not standard for Python, we implemented abstract classes to define a clear application programming interface (API) on how to use and extend the library (see Figure 1). The abstract classes reflect, among others, the components above:

- `ProbabilisticModel` defines how to provide methods to simulate data given parameters θ.
- `Statistics` defines how to provide methods to extract statistics.
- `Distance` defines how to provide distance calculations.
- `ApproxLikelihood` defines how to provide a likelihood approximation.

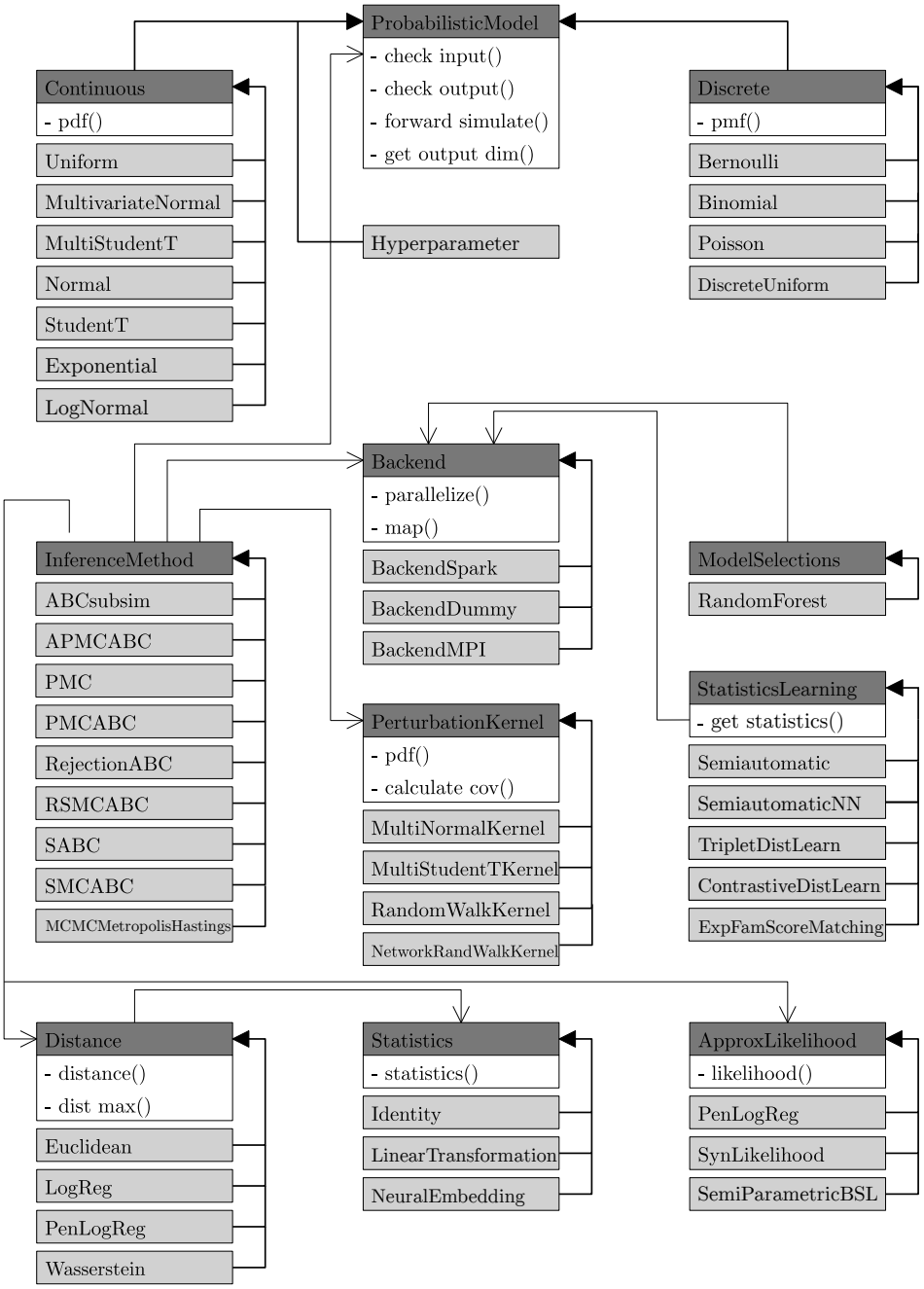


Figure 1: This diagram shows selected classes with their most important methods. Abstract classes, which cannot be instantiated, are highlighted in dark gray and derived classes are highlighted in light gray. Inheritance is shown by filled arrows. Arrows with no filling highlight associations, e.g., `Distance` is associated with `Statistics` because it calls a method of the instantiated class to translate the input data to summary statistics. `ContrastiveDistLearn`, `TripletDistLearn`, `MultiNormalKernel` and `MultiStudentTKernel` are used as an abbreviation for `ContrastiveDistanceLearning`, `TripletDistanceLearning`, `MultivariateNormalKernel` and `MultivariateStudentTKernel` respectively.

All provided components derive from these abstract classes and implement the required methods; moreover, the user can easily extend the library by sub-classing the above abstract classes.

In **ABCpy**, the '`abcpy.probabilisticmodels.ProbabilisticModel`' class represents the probabilistic relationship between random variables or between random variables and observed data. Each of the `ProbabilisticModel` objects has a number of input parameters: they are either random variables (output of another `ProbabilisticModel` object) or constant values known to the user (of type `Hyperparameter`).

To define the parameter of a model as a random variable, the user has to assign a *prior distribution* on it. To this aim they can exploit *prior* knowledge about the parameter value and its distribution. In the absence of prior knowledge, we still need to provide prior information and a flat distribution on the parameter space can be used. The prior distribution on the random variables are assigned by a probabilistic model which can take, as inputs, either other random variables or hyper-parameters.

We consider now the Lorenz95 model as discussed in Section 1. Assuming we observe a realization of the model, we are interested in inferring two one-dimensional parameters (θ_1, θ_2) that enter in the definition of the equations; more information on the structure of the model is given in Appendix A. We define the graphical structure of the model as follows[2]:

```
>>> from abcpy.continuousmodels import Uniform
>>> theta1 = Uniform([[0.5], [3.5]], name = "theta1")
>>> theta2 = Uniform([[0], [0.3]], name = "theta2")
>>> sigma_e = 1; phi = 0.4; T = 1024
>>> lorenz = StochLorenz95([theta1, theta2, sigma_e, phi, T],
...   name = "lorenz")
```

We have thus defined the parameter θ_1 and θ_2 of the Lorenz95 model as random variables and have specified Uniform prior distributions for them. The parameters of the prior distribution and the parameters σ_e and ϕ of the model are assumed to be known to the user, hence they are called hyper-parameters. Also, internally, the hyper-parameters are converted to `Hyperparameter` objects. Finally, T defines the number of integration timestep used for the model.

Note that you can pass a name string (e.g, `"theta_1"`) while defining a random variable. In the final output, you will see these names, together with the relevant outputs corresponding to them.

As the output of each integration of the model is a 40 dimensional timeseries with T steps, it is computationally inefficient to apply ABC inference on the output directly. Therefore, we extract a six-dimensional set of summary statistics suggested in Hakkarainen *et al.* (2012) before computing the discrepancy measure as the Euclidean distance between statistics of different realizations. The definition of these summary statistics looks as follows (the class definition is reported in the full example):

```
>>> statistics_calculator = HakkarainenLorenzStatistics(degree = 1,
...   cross = False)
```

[2]The code needed to run this and the following examples are provided in the supplementary material (this also includes the definition of the `StochLorenz95` model and the `HakkarainenLorenzStatistics` statistics used later in the text, not shown here for brevity).

The discrepancy measure is defined in the next piece of code and takes as argument the corresponding statistics; when the inference algorithm is run, it will automatically extract the statistics from the datasets and subsequently compute the distance between the two statistics.

```
>>> from abcpy.distances import Euclidean
>>> distance_calculator = Euclidean(statistics_calculator)
```

As discussed in Section 2.3, most algorithms in **ABCpy** (except for `RejectionABC`) require a perturbation kernel to explore the parameter space. For this example we use the default kernel, which in the case of continuous parameters uses a multivariate Gaussian distribution; it can be defined in the following way:

```
>>> from abcpy.perturbationkernel import DefaultKernel
>>> kernel = DefaultKernel([theta1, theta2])
```

Finally, we need to specify a backend that determines the parallelization framework to use. The example code here uses the **MPI** backend `BackendMPI` which parallelizes the computation of the inference schemes using **MPI**. As mentioned earlier, a parallelization backend supporting **Spark** (`BackendSpark`) is available, as well as a dummy one (`BackendDummy`) which does not parallelize the computations, but is handy for prototyping and testing. A detailed description of how the parallelization schemes work is in the Section 5.

```
>>> from abcpy.backends import BackendMPI as Backend
>>> backend = Backend()
```

For the sake of illustration we choose the PMCABC algorithm as the inference scheme to draw posterior samples of the parameters. Therefore, we instantiate a `PMCABC` object by passing the model, the distance function, backend object, perturbation kernel and a seed for the random number generator.

```
>>> from abcpy.inferences import PMCABC
>>> sampler = PMCABC([lorenz], [distance_calculator], backend, kernel,
...   seed = 1)
```

Finally, we can parametrize the sampler by specifying the number of steps `steps`, the number of posterior samples `n_samples` and the number of simulations for each parameter value `n_samples_per_param`:

```
>>> steps, n_samples, n_samples_per_param, full_output = 3, 10000, 1, 1
>>> eps_arr = np.array([500]); eps_percentile = 10
```

Note that the **ABCpy** implementation of the PMCABC algorithm (Algorithm 1) is parametrized with an array of threshold values $(\epsilon_t)_t$ (`eps_arr` here) and a percentile value (`eps_percentile`), and that at iteration t of the algorithm the actual threshold will be the maximum between ϵ_t and the percentile of the distances from the previous iteration (see Algorithm 1). **ABCpy** allows however to specify only the first threshold values, in which case the iterations starting from the second one will use the percentile of the previous iteration distances.

For illustration purposes, we generate an artificial dataset `observation` from the Lorenz95 model (the replication materials provide the code for function `lorenz.forward_simulate()`):

```
>>> parameter_true_values = [2, .1]
>>> observation = lorenz.forward_simulate([2, .1, sigma_e, phi, T], 1,
...   rng = np.random.RandomState(42))
```

We can now sample from the posterior distribution of the parameters given the observed dataset `observation`:

```
>>> journal = sampler.sample([observation], steps, eps_arr, n_samples,
...   n_samples_per_param, eps_percentile, full_output = full_output)
```

The above inference scheme gives us samples from the posterior distribution of the parameters `theta_1` and `theta_2`, implicitly quantifying the uncertainty of the inferred parameter, which are stored in the `journal` object. In particular the posterior mean and covariance matrix of (θ_1, θ_2) are obtained as:

```
>>> print(journal.posterior_mean())

{'theta1': 2.2076253010296836, 'theta2': 0.09532895309059163}

>>> print(journal.posterior_cov())

(array([[ 0.23369259, -0.02576822],
       [-0.02576822,  0.00470818]]), dict_keys(['theta1', 'theta2']))
```

A plot for the bivariate and univariate marginals posterior distributions can be obtained and saved to the disk with:

```
>>> journal.plot_posterior_distr(path_to_save =
...   "../Figures/lorenz_hakkarainen_pmcabc.pdf")
```

The resulting graphic can be seen in the top-left panel of Figure 7. Note that the model and the observations are given as a list. This is due to the fact that in **ABCpy**, it is possible to have hierarchical models and to build relationships between co-occurring groups of datasets, as detailed in Section. 6.3.

4. Modular API

As one can notice from the structure of the code, the design of **ABCpy** is highly modular, so that adapting to different use cases and scenarios can be done with as little overhead as possible. In this section, we show how **ABCpy**'s modularity addresses the needs of various use cases in a user-friendly, intuitive way. The contributions to each use case are detailed as follows:

1. Non-ABC experts do not have to worry about the details of the sampling scheme; no knowledge of the interaction between sampling schemes, models, kernels etc. is needed.
2. Non-HPC experts can easily run the ABC schemes on hundreds of cores even without explicitly parallelizing their code.
3. ABC experts can easily extend the library with new ABC algorithms (rapid prototyping) and compare their performance in a standardized environment.

Scientists who want to use ABC to calibrate their models only need an abstract understanding of the ABC methodology and only need to provide information in the domain of their expertise. The model and the means to forward simulate data for given model parameters are the most fundamental information they need to provide. Further, scientists usually have a way to discriminate two simulation outcomes and can make an informed decision on which better fits the observed data. This knowledge domain expertise can drive the choice of the ABC summary statistics. Apart from this, the user only has to provide prior information and parametrizations of the sampling scheme. These include a perturbation kernel, simulation length and simulation stopping criteria. All ABC details are completely handled by the corresponding modules.

ABC experts can extend the library by providing new sampling schemes, distances or approximate likelihood methods. To do so, the user can sub-class the `InferenceMethod`, `Distance` or `ApproxLikelihood` abstract classes and implement the relevant methods. Those can be subsequently used with any `ProbabilisticModel` and `Backend`, providing simple and fixed environment for benchmarking and for testing reproducibility. Moreover, we provide implementations of several fundamental ABC algorithm (PMCABC, SMCABC, RejectionABC), which can be used as a starting point to rapidly prototype similar ones. For instance, a new SMCABC-type algorithm can be added by adapting the relevant lines of code in our SMCABC implementation.

HPC-experts can adapt the library to their specific system. For example, in case Apache **Spark** or **MPI** is not available or suitable, a system engineer might extend the library to available parallel architecture by sub-classing the `Backend` class.

4.1. API design decisions

In this section, we provide some background on what led to current design decisions, in particular why we chose Python, **MPI**, **Spark**, and the map-reduce paradigm.

Let us first explain why Python was selected over other languages. For high-level scripting languages, Python is one of the most used languages in data science. It comes with a large range of well-tested scientific libraries, such as **NumPy** (Harris *et al.* 2020) and **SciPy** (Virtanen *et al.* 2020). Further, if one considers the standard use case of data scientists, usually rapid prototyping is required rather than finding a solution and then tweaking it to work optimally to solve the same problem over and over again. Thus we chose it against low-level languages such as C++ or Fortran. Further, in ABC most computation time is spent simulating from the model. In case this might be too inefficient in Python, it can be implemented in a lower level language as Fortran or C++, and connected to Python using e.g., **SWIG** (Beazley *et al.* 1996), for which we provide examples in the documentation at `https://abcpy.readthedocs.io/en/latest/`.

The parallelization backend follows the map-reduce programming model. An important argument for map-reduce is its simplicity: there is no need to explicitly handle communication or worry about thread-safety, deadlocks, or race-conditions. The price to pay is that not every problem is easily expressible in a map-reduce fashion. However, this is not a constraint for us since the individual tasks of the ABC sampling schemes are more or less independent and no sophisticated communication patter is required. We consider the map-reduce paradigm to be sufficient for the implemented methods. This belief is also supported by thes performance measurements presented in Section 5.

We have implemented two different parallelization backends for the library, one based on Apache **Spark** (Zaharia *et al.* 2016) and the other based on **MPI** (Message Passing Interface Forum 2012) with the idea that they account for most of the computing infrastructure nowadays available to researchers and data scientists. Apache **Spark** is widely used in industry for large scale data analytics and many computer infrastructure services at universities also offer **Spark** clusters to their researchers. Even if this is not an option, there are many commercial **Spark** providers (for instance Amazon Web Services), some of which even offer free access to researchers. On the other hand, many high performance clusters found at supercomputing centers use **MPI** as a communication framework, which is often optimized to the respective infrastructure. To enable users of such facilities to easily adopt and experiment with **ABCpy**, we also implemented an **MPI** backend.

5. Parallelism

As discussed in Section 2.3, the different sampling schemes implemented in **ABCpy** follow a similar flow of instructions. Thus, to explain how the parallelism works, we first refer to Algorithm 1. The flow of the main loop is as follows:

(i) (Re-)sample a set of parameters θ either from the prior or from an already existing set of parameters (Lines 3, 16, code block).

(ii) For each parameter, perturb it using the perturbation kernel, simulate the model and generate pseudo-data, compute the distance between generated and observed data, and either accept the parameter value if the distance is "small", or repeat the whole second step (Lines 4–7, 17–21, code block).

(iii) For each parameter value calculate its corresponding weight (Lines 8, 22, code block).

(iv) Normalize the weights, calculate a covariance matrix and a quantile (Lines 10, 24–26, code block).

These four steps are repeated until the weighted set of parameters, interpreted as an approximation of the posterior distribution, converges. There are several ways to define "convergence"; however, we will not go into the details here. See Section 6.6 for **ABCpy** tools to assess convergence.

Parallelization of the algorithms is done in the following way: resampling the parameters in step (i) and the small computations in step (iv) are usually quite fast, even for large numbers of parameters, and thus we refrain from parallelizing them. On the other hand, step (ii) and (iii) are the computationally expensive parts. The generation of simulated data from the model, for a given parameter value, usually requires substantial computational resources. This step therefore has the highest potential for parallelization. As already mentioned, we parallelize in a map-reduce fashion (Dean and Ghemawat 2008). Therefore, we created a mapping function that maps each parameter value to a perturbed parameter value and next to a pseudo-observation $\boldsymbol{y}_{\text{sim}}$ generated from the model with the corresponding perturbed parameter value. With this, we can create one task for each parameter such that step (ii) can be fully parallelized. The results of the mapping phase, i.e., the accepted parameters, are then collected by (sent back to) the master. The weight computation in step (iii) has

a quadratic time complexity in the number of parameters. Thus, we again parallelize it by mapping the parameters to their weights.

Usually the parallelized steps (model simulation and weight computation) take sufficient time so the communication overhead plays only a minor role in the overall execution time. Further, in both steps, all tasks can be run independently of each other since they do not require any communication. One would thus expect nearly linear scalability, at least as long as the inherently sequential parts of the program have a run time much shorter than the parallel parts.

Map-reduce assumes an underlying master/worker architecture, where the master orchestrates the work, performs light-weight operations, and distributes independent tasks to a large set of worker nodes; each worker can usually run tasks in parallel using *executors* (for instance, different processors). In a map phase, the master sends a task in form of a function to the workers, whose executors apply it independently to each element of data local to the worker node. In a reduce phase, the master makes the workers reshuffle the data and apply a reduce function to the data. As a matter of fact, we only need a very simple implementation of reduce, i.e., a *collect*, that sends the data back to the master without applying any function. As mentioned, this paradigm is simple to implement but has the disadvantage of being limited in its expression complexity. Fortunately the presented algorithms can be parallelized quite easily, as the parallel parts of the algorithms can mostly run independently from each other, so that worker-to-worker interaction is not needed.

Apache **Spark** is a sophisticated implementation of map-reduce. Creating a parallelization backend using Apache **Spark** is rather simple since we can entirely rely on the built-in functions. The **Spark** backend can be seen as a wrapper that connects the **ABCpy** internal map-reduce functions to the Apache **Spark** ones.

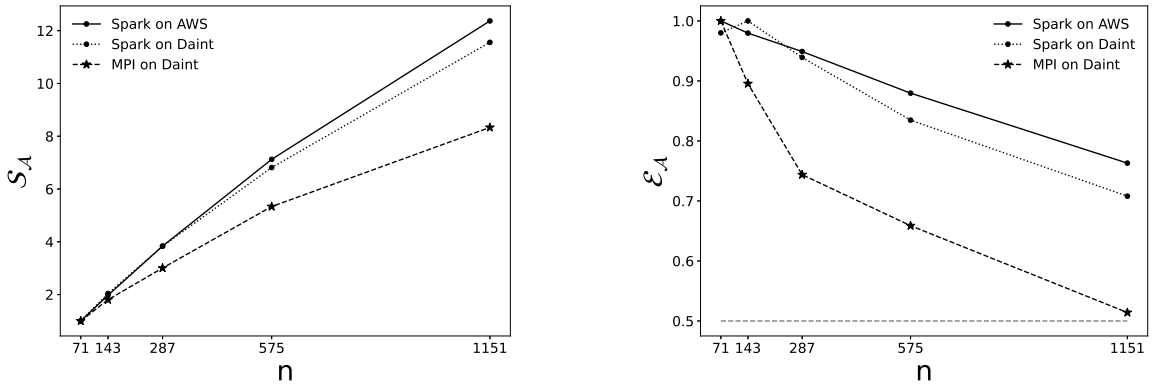
Creating an **MPI** backend for **ABCpy** is a completely different story, since **MPI** only comes with a set of low-level functions that enable nodes to exchange information in a one-to-one, one-to-many, and many-to-many fashion with additional control mechanisms. The map and reduce functions thus have to be implemented with these low-level primitives. **MPI** does not naturally provide a master/worker architecture. Instead, we select one node to act as the master and rest are treated as worker nodes. **MPI** does not directly deal with nodes as entities but instead provides a rank which can be seen as a process that has been bound to a certain number of cores. We thus implement our executors to run on a rank. In our implementation of the map phase, the master splits the work into tasks and assigns them to executors such that every executor performs roughly the same number of tasks (or ideally the some amount of work). The collect phase is more easy to implement since we only require the data to be sent back to the master without any shuffling.

5.1. Performance evaluation

Here we present a performance evaluation of the parallelized architecture of the PMCABC algorithm (Algorithm 1) by analyzing the scalability with the Apache **Spark** and **MPI** backends using the Lorenz95 model and the PMCABC algorithm, both of which were described in Section 3.

Full details about the model and the algorithmic parameters for the experiments in this and the following sections are reported in Appendix A.

To test scalability, we ran the same experiment using the **Spark** and **MPI** backends on the



(a) Performance: speedup

(b) Performance: efficiency

Figure 2: Speedup $\mathcal{S}_\mathcal{A}(n)$ and efficiency $\mathcal{E}_\mathcal{A}(n)$ of PMCABC algorithm for Lorenz95 model using **Spark** and **MPI** backend with different number of cores n.

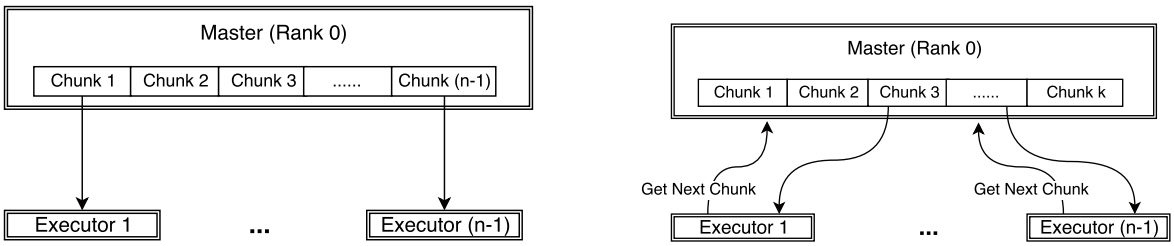
CSCS (Centro Svizzero di Calcolo Scientifico) super computer Piz Daint, where we used multi-core nodes each having two Intel Broadwell processors with 36 cores in total and 64GB RAM each. We kept the size of the problem fixed and we scaled up the number of worker nodes from 2 to 32 in powers of 2, leading to experiments being run on 72, 144, 288, 576 and 1152 cores respectively. We also ran a similar experiment using **Spark** on AWS in order to investigate the performance of the library on a commercial cloud computing platform. We used "c4.8xlarge" instances which provide an equivalent 36 vCPUs and 60GB RAM each. Due to the multi-core architecture of Daint and AWS, the cores here are equivalent to the executors discussed above. Further, for the **MPI** backend to be comparable to **Spark**, we did not perform any computation on the cores belonging to the first node and dedicated it to be a Master node.

To study scalability, we considered two quantities: speedup and efficiency. The *speedup* $\mathcal{S}_\mathcal{A}(n)$ of a parallel algorithm $\mathcal{A}$ on n cores with respect to a baseline (number of cores) $m, m \leq n$, is the ratio of the algorithm's running time $t(m)$ on m cores and the running time $t(n)$ on n cores, $\mathcal{S}_\mathcal{A}(n) = t(n)/t(m)$. The *efficiency* $\mathcal{E}_\mathcal{A}(n)$ of an algorithm $\mathcal{A}$ on n cores is defined as the speedup normalized by the ratio of n to the baseline m, i.e., $\mathcal{E}_\mathcal{A}(n) = \mathcal{S}_\mathcal{A}(n)m/n$.

Figure 2 shows that with the **Spark** backend on both Piz Daint and AWS perform similarly. The performance increases close to linearly for smaller number of cores but fails to do so for larger ones. We attribute this to the fact that the entire process is not perfectly parallelizable but has serial and parallel regions interlaced. As the parallel execution gets faster, the time spent in serial execution begins to affect overall performance. Confirming Amdahl's law (Amdahl 1967), with increasing parallelism the efficiency depicted in Figure 2b drops as the number of cores increases. One can observe that the **MPI** backend is roughly on par with the Apache **Spark** backend in terms of performance, at least up to 576 cores i.e., when Amdahl's law starts kicking in.

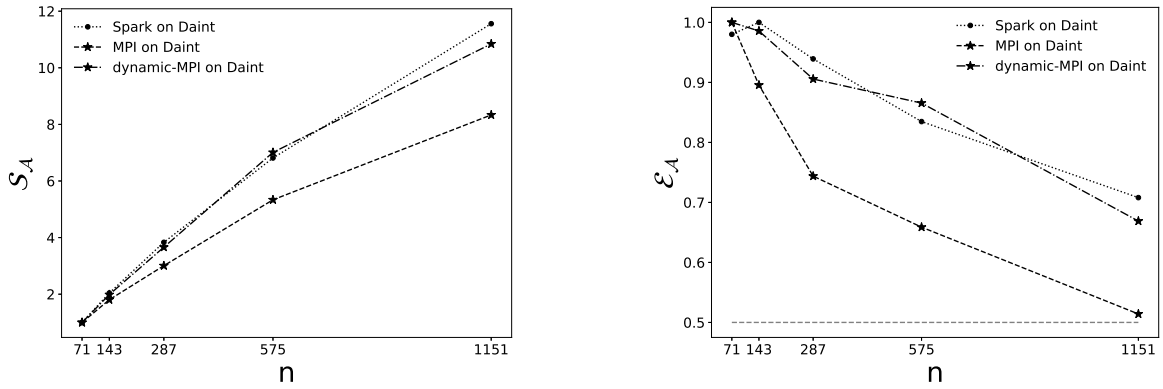
5.2. Dynamic allocation for MPI

In this section, we discuss the inherent imbalances of some ABC algorithms and consequently the importance to study the respective effects. As a solution to the imbalance issues, we also



(a) MPI backend (b) dynamic-MPI backend

Figure 3: Comparison of workflow between **MPI** and dynamic-**MPI** backend.



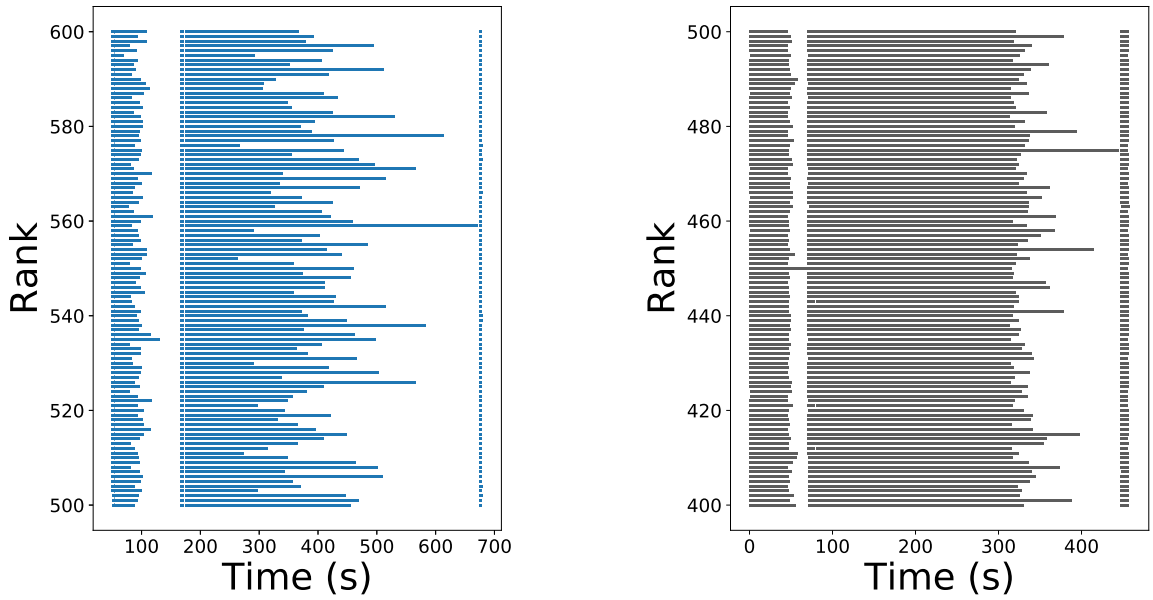
(a) Performance: speedup (b) Performance: efficiency

Figure 4: Speedup $\mathcal{S}_{\mathcal{A}}(n)$ and efficiency $\mathcal{E}_{\mathcal{A}}(n)$ of PMCABC algorithm for the Lorenz95 model (with $T = 1024$) using **Spark**, **MPI**(straight-forward) and **MPI**(dynamic-allocation) backends on different number of cores, n.

discuss the importance of a dynamic work allocation strategy for map-reduce. We provide an empirical comparison of a straightforward allocation approach versus an online greedy approach.

In the straightforward approach, the allocation scheme initially distributes m tasks to n executors splitting them identically, and then sends the map function to each executor, which in turn applies the map function one after the other for its m/n map tasks. This approach is visualized in Figure 3a, where a chunk represents the set of m/n map tasks. For example, if we want to draw $20,000$ samples from the posterior distribution and we have $n = 100$ cores available, at each step of PMCABC we create chunks of 200 parameters and each chunk is assigned to one individual executor.

On the other hand, the dynamic allocation scheme initially distributes $k < m$ tasks to the k executors, sends the map function to each executors, which in turn applies it to the single task available. In contrast to the straightforward allocation, the executor requests a new map task as soon as the old one is finished. This has the benefit that the work is better balanced, as we show in Figure 4. The dynamic allocation strategy is an implementation of a greedy algorithm for job-shop scheduling, which can be shown to have an overall processing time (makespan) up to twice the best makespan (Graham 1966). This approach is depicted in Figure 3b.



(a) MPI(straight-forward)

(b) MPI(dynamic-allocation)

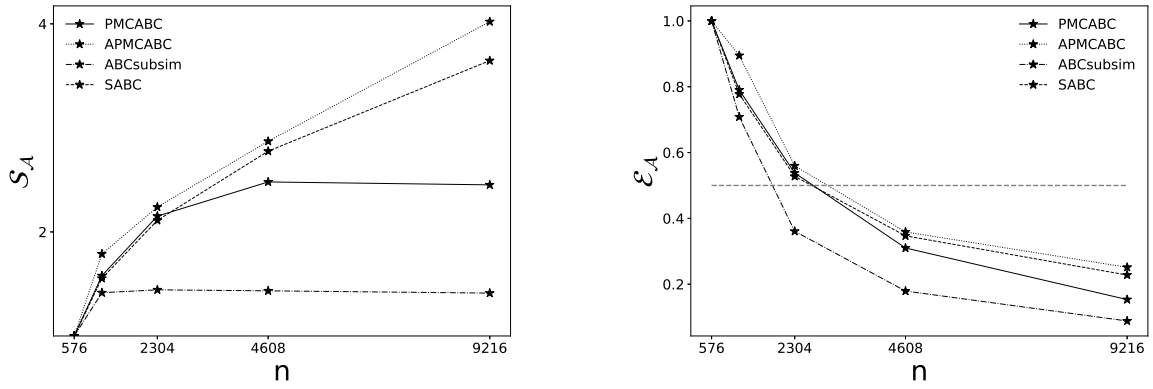
Figure 5: Imbalance of the PMCABC algorithm using **MPI**(straight-forward) and **MPI**(dynamic-allocation) backend for the Lorenz95 model ($T = 1024$). Note the large difference in the time-scale (in seconds) on the horizontal axis.

The unbalanced behavior can be made apparent by visualizing the run time of the individual map tasks on each executor. In Figure 5, the individual map task's processing time is shown for PMCABC. Each row corresponds to an executor and each bar corresponds to the total time spent on all tasks assigned to the respective executor for one map call. For the straightforward allocation strategy, Figure 3a, one can easily see that a majority of executors finish their map tasks in half the time of the slowest one. However, to continue with the next step of the map reduce execution, all workers and its executors have to be finished. This clearly leads to large inefficiencies. Conversely, using the dynamic allocation strategy, Figure 3b, the work is more evenly distributed across the executors. The cause of the different execution times lie in the stochasticity of the forward simulation and to a major extent is particular to the PMCABC algorithm as we discuss later in Section 5.3.

From this observation it follows that the unbalancedness cannot be fixed by adding resources, and has a severe impact on scalability, as Figure 4 shows. Speed-up and efficiency drop drastically compared to the **Spark** implementation and the dynamic allocation strategy with increasing number of executors. This can be understood as follows: in the strong scaling setting, the total number of map tasks m is fixed, so if we increase the number of executors k, the number of tasks per executor m/k gets smaller. A small number of map tasks per executor has a higher variance in the total execution time.

5.3. Parallelism and ABC algorithms

In Section 5.2, we pointed out the presence of an inherent imbalance of the PMCABC algorithm as the execution time of step (ii) for different parameters varied significantly. In this



(a) Performance: speedup

(b) Performance: efficiency

Figure 6: Comparison of speedup and efficiency for PMCABC, SABC, APMCABC and ABCsubsim using the Lorenz95 model ($T = 1024$).

section, we explain the fundamental reason behind this imbalance and then compare different algorithms in **ABCpy** from a parallelization perspective.

The acceptance in step (ii) (at page 14) can be easily split into independent jobs and parallelized for all the algorithms in each group. Recall now the distinction between ABC algorithms with explicit and implicit acceptance (Section 2.3); for the latter, one single simulation for each perturbed parameter value is generated and the parameter is accepted probabilistically. In the former case, instead, simulations are run until the simulation matches the observation at some tolerance level ϵ_t. For an explicit acceptance to occur, therefore, it may take different amounts of time for different perturbed parameters (more repeated steps are needed if the proposed parameter value is distant from the true parameter value). Hence the first group of algorithms are inherently imbalanced as illustrated for the PMCABC algorithm in Figure 5. Instead, the algorithms with probabilistic acceptance do not have a similar issue of imbalance as a probabilistic acceptance step takes approximately the same amount of time for each parameter.

Next we compare the achieved performance gain by exploiting parallelism for four ABC algorithms: PMCABC, APMCABC, SABC and ABCsubsim. The choice of these four algorithms were motivated by three aspects: (a) PMCABC is the most classical ABC algorithm; (b) APMCABC and SABC are, to the best of our knowledge, the ABC algorithms with faster convergence to posterior distribution and the minimal number of model simulations needed (Lenormand *et al.* 2013; Albert *et al.* 2015); (c) ABCsubsim is instead a popular algorithm for engineering applications (Kulakova *et al.* 2016). Further we comment that we exclude SMCABC, RSMCABC and RejectionABC from our analysis, due to the almost similar performance of SMCABC in comparison to SMCABC and the inability of RSMCABC and RejectionABC to scale up while using correspondingly more than 1 or 100 parallel cores, which is a much smaller number with respect to the one we considered here.

We run now the above algorithms on the Lorenz95 model as discussed in Section 3; as the code for PCMABC was provided there, we show here how to run the inference with the other three algorithms; all parameters, except for the specified ones, are left to their default value. The inference with SABC is run by instantiating an '`SABC`' object, defining sampling parameters,

and finally sampling from the posterior distribution:

```
>>> from abcpy.inferences import SABC, APMCABC, ABCsubsim
>>> sampler = SABC([lorenz], [distance_calculator], backend, kernel,
...   seed = 1)
>>> steps, n_samples, n_samples_per_param, full_output = 20, 10000, 1, 1
>>> epsilon = 500
>>> journal = sampler.sample([observation], steps, epsilon, n_samples,
...   n_samples_per_param, full_output = full_output)
```

Similarly for ABCsubsim:

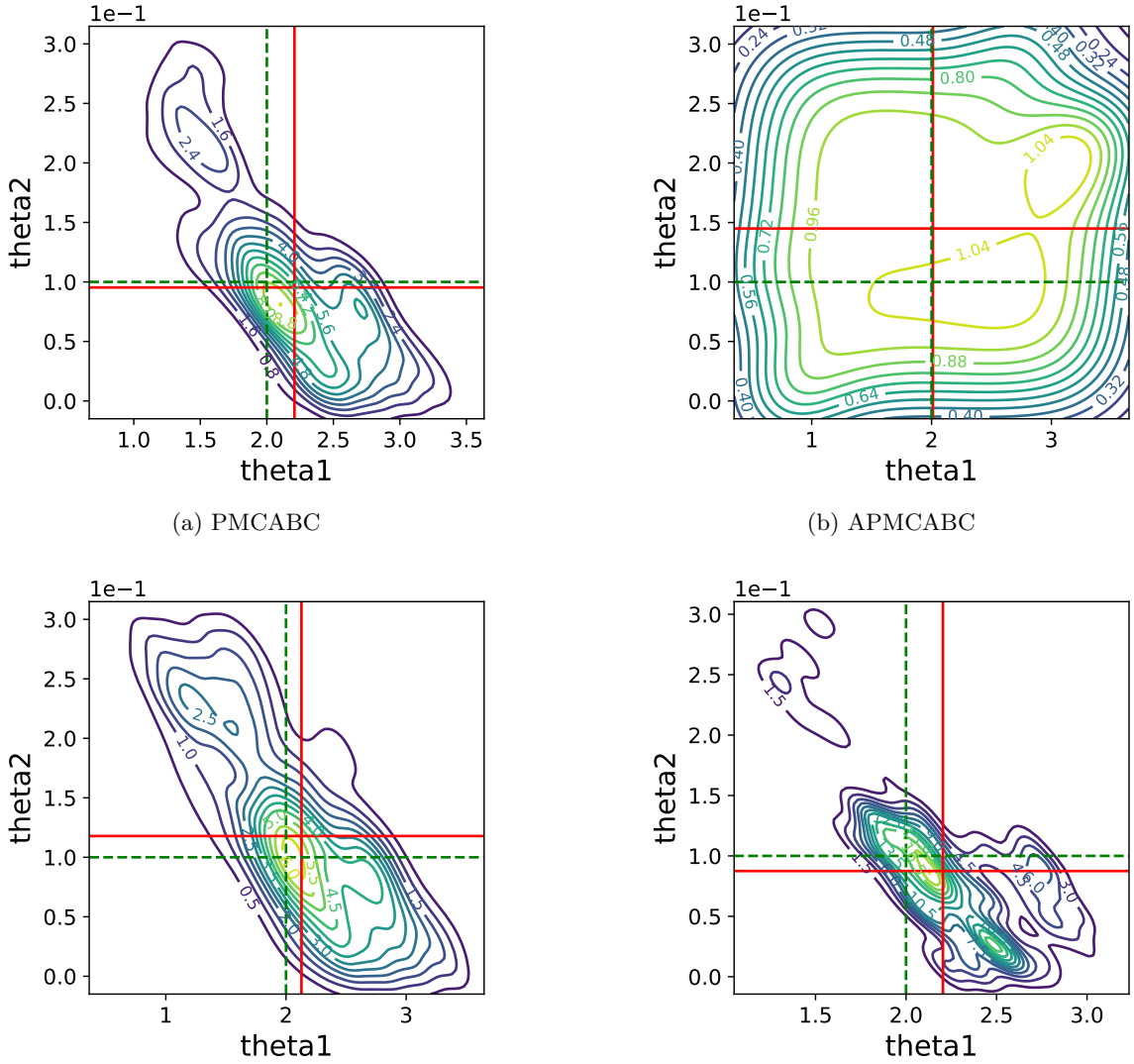
```
>>> sampler = ABCsubsim([lorenz], [distance_calculator], backend, kernel,
...   seed = 1)
>>> steps, n_samples, n_samples_per_param, full_output = 20, 10000, 1, 1
>>> journal = sampler.sample([observation], steps, n_samples,
...   n_samples_per_param, full_output = full_output)
```

and APMCABC:

```
>>> sampler = APMCABC([lorenz], [distance_calculator], backend, kernel,
...   seed = 1)
>>> steps, n_samples, n_samples_per_param, full_output = 20, 10000, 1, 1
>>> acceptance_cutoff = 0.003
>>> journal = sampler.sample([observation], steps, n_samples,
...   n_samples_per_param, full_output = full_output)
```

In Figure 6, we compare the speed-up and efficiency of the considered algorithms. More details on the settings of the different algorithms can be found in Appendix A. We notice that ABC algorithms with "probabilistic acceptance" do not have an inherent imbalance, but they may not be easily parallelizable due to the sequential nature of the algorithm, which is illustrated by the poor performance of ABCsubsim algorithm compared to the others. We also conclude that the performance of APMCABC and SABC is significantly better compared to PMCABC due to the absence of imbalance in them and are therefore better suited for a parallelization with the map-reduce paradigm.

Moreover, with regards to the total computational complexity of the different algorithms, note that running one of the algorithms implemented in the above code chunk (not involving explicit acceptance step) for 20 iterations with 10000 posterior sample points took roughly as long as running the PMCABC for 3 iterations, with the same number of samples (see code in Section 3). In fact, once PMCABC reached the 4th iteration, for each accepted simulation of the model, around 1000 simulations were needed; this is extremely expensive, so that we were not able to run the algorithm for more than three iterations due to limitations in computing capability. This also explains the worse approximation to the posterior density obtained with this algorithm with respect to the other ones (Figure 7).



(a) PMCABC

(b) APMCABC

(c) SABC

(d) ABCsubsim

Figure 7: Contour plots for the approximate posteriors inferred with the different ABC algorithms. The red lines denote position of the posterior mean, while the green lines denote the position of the true parameter values. The posteriors were obtained with kernel density estimate starting from the posterior samples.

6. Innovations of ABCpy compared with similar packages

We now compare **ABCpy** with other general-purpose ABC packages for high-level languages, namely **ELFI** (Lintusaari *et al.* 2018) and **pyABC** (Klinger, Rickert, and Hasenauer 2018) for Python and **abc** (Csilléry, François, and Blum 2012) and **EasyABC** (Jabot, Faure, Dumoulin, and Albert 2015) for R (R Core Team 2021). In the following Sections, we highlight the important innovations that are included in our package and are not available in any of the competing ones.

Feature	ABCpy	ELFI	pyABC	abc	EasyABC
Language	Python	Python	Python	R	R
Latest release	2021	2021	2021	2015	2015
Version	0.6.3	0.8.0	0.11.2	2.1	1.5
Parallelization	Multicore, distributed	Multicore, distributed	Multicore, distributed	No	Multicore
Graph based	Yes	Yes	No	No	No
Inference schemes	Wide choice of techniques	BOLFI, SMCABC, Rejection-ABC	SMCABC only	RejectionABC only	Wide choice of techniques
Co-occurring dataset	Yes	No	No	No	No
Nested parallelization	Yes	No	No	No	No
Composite kernel	Yes	No	No	No	No
Statistics learning	Yes	No	Preliminary	No	No
Convergence diagnostics	Yes	Yes	No	No	No

Table 1: Review of the main features of different ABC packages.

In terms of inference techniques, **ABCpy** is arguably the most complete one. In fact, it implements a selection of Sequential and MCMC based methods, as well as Simulated Annealing ABC. **EasyABC** provides a similar selection, but the latest release was in 2015, therefore missing out the latest algorithmic developments. **ELFI** implements SMCABC, RejectionABC and BOLFI (Gutmann and Corander 2016), that uses Gaussian process Bayesian optimization to speed up computation. **pyABC** only consider sequential techniques, while **abc** only provides the RejectionABC scheme, complemented with two post-processing techniques (Beaumont *et al.* 2002; Blum and François 2010). Moreover, **ELFI** and **EasyABC** are not able to perform model selection.

As discussed above, **ABCpy** can use **Spark** and **MPI** to parallelize the computation on multicore and distributed systems; the same is possible with **ELFI** and **pyABC**, the former using **ipyparallel** (**IPython** Development Team 2021) for distributed systems and Python built in library for multiple cores, while the latter is able to work with several backends, among which **Dask** (**Dask** Development Team 2016), the **IPython** (Pérez and Granger 2007) parallel cluster and **Redis** (Carlson 2013). We remark moreover that **ABCpy** is the only package to offer the nested parallelization feature, which is detailed in Section 6.5. **ELFI** is moreover able to vectorize simple operations in the simulator, by performing batches of simulations at once. **abc** does not provide any parallelization, as it assumes the model simulations had been run beforehand and the output formatted and passed to the package; instead, **EasyABC** is able to parallelize only on multicore machines, but if the simulator code is a binary executable, parallelization requires modifying it.

The description of the dependencies between the different components of the probabilistic model, as done in Section 2, creates an underlying computational graph in **ABCpy**. This allows great flexibility in specifying an overall model, as different components may be composed in several ways with no need to changing their structure. This approach is also present in **ELFI**, while it is missing in the other packages considered here.

With regards to code modularity, we believe **ABCpy** to be the best package, alongside with **ELFI**. With the exception of **abc**, that requires the observation and simulations to be provided to the inference scheme as matrices and does not allow the implementation of other methods, the other packages all have a modular structure, but in different ways. **EasyABC** allows

models to be specified in functions or external binary files, but does not separate the model and the statistics component. **pyABC** allows the models to be either functions or classes, but they need to work with Python dictionaries as input and output. Moreover, it is not possible to easily extend **pyABC** to other inference schemes, but only to modify the parameters or the scheduling of the SMCABC algorithm. **ELFI** and **ABCpy** are instead similar in terms of their modularity.

Additionally, **ABCpy** implements semiautomatic summary selection routines, as well as the possibility of using neural networks to learn and implement statistics in ABC inference; this is described in detail in Section 6.1. The only other package allowing to automatically learn summary statistics is **pyABC** (which at the current release only implements a preliminary version of the semiautomatic method, with no neural network support).

Also, to the best of our knowledge, **ABCpy** is the only package offering the possibility of performing inference with co-occurring measurements of different quantities that belong to the same graphical model (see Section 6.3). Finally, **ABCpy** and **ELFI** are the only packages to provide tools for assessing the convergence of the posterior approximation for ABC algorithms (Section 6.6).

In Table 1 we display a quick summary of the features of the different packages discussed here.

6.1. Learning summary statistics

As discussed above, informative summary statistics are a main component of ABC algorithms. Practitioners may choose knowledge domain driven summaries, thus focusing the inferential process on specific data features encoded by those summaries. However, in many cases we would like the approximate posterior to be as close as possible to the one obtained with the whole dataset, but we still need to use summary statistics as the dimension of the raw data is too large, leading to poor computational performance.

Therefore, ways to automatically learn summary statistics have been developed. **ABCpy** implements some techniques based on mapping the data to lower dimensional subspaces, that are described in the following. For all of them, before the ABC algorithm is run, a set of parameter-simulation pairs $(\theta_i, \boldsymbol{y}_i)_{i=1}^n$ is generated according to the prior and the model; then, a learning algorithm is applied in order to learn a data transformation. During the subsequent inference, the data will be transformed with the latter, providing the summary statistics. We note that before the learning step, the generated data is optionally transformed with a fixed *statistics* function, for instance to obtain a polynomial expansion of the raw data.

A very popular approach is the one introduced in Fearnhead and Prangle (2012), in which the learned transformation is a linear projection to the dimension of the parameter. Specifically, the following linear model is fit:

$$\theta_i = \mathsf{E}(\theta_i \mid \boldsymbol{y}_i) + \xi = \boldsymbol{y}_i^\top \beta + \xi, \tag{2}$$

where ξ is a 0-mean noise vector with independent components and β is the set of parameters that are fitted. During inference, therefore, statistic for a new sample $\boldsymbol{y}_{\text{sim}}$ will be $\boldsymbol{y}_{\text{sim}}^\top \beta$. This is implemented in the '`Semiautomatic`' class and showed in the following piece of code for a generic `model`:

```
from abcpy.statisticslearning import Semiautomatic
from abcpy.statistics import Identity
```

```
statistics_calculator = Identity(degree = 2, cross = True)
new_statistics = Semiautomatic([model], statistics_calculator,
  backend, n_samples = 200).get_statistics()
```

where `statistics_calculator` is the summary statistics applied before learning the transformation and the new summary statistics is learned and stored as `statistics_calculator`. Here, note that the Identity statistics applies a polynomial expansion of order `degree` to the data and optionally computes cross products (argument `cross`), before applying the statistics learning algorithm.

The authors of Jiang, Wu, Zheng, and Wong (2017) extended this approach by using a neural network model instead of a linear transformation, namely replacing $\boldsymbol{y}_i^\top \beta$ by $f_w(\boldsymbol{y}_i)$ in the above expression, where f_w denotes the transformation applied by a specific neural network with weights w, which are determined by iteratively minimizing the corresponding least squared regression loss; this is implemented in the '`SemiautomaticNN`' class. In the same way as before, the statistic will therefore be $f_w(\boldsymbol{y}_{\text{sim}})$. The neural network summary selection allows much more representation power than the linear transformation one, with very small or no additional lines of code required with respect to the linear regression one. We give here an example of this technique for the Lorenz95 model, by using as a neural network the Partially Exchangeable Network introduced in Wiqvist, Mattei, Picchini, and Frellsen (2019), that is an embedding of the 40-dimensional time series whose output is invariant to permutations in the input that are characteristic of the Markovianity of the time series; see Wiqvist *et al.* (2019) for more details on that. After having learned the statistics, we carry out inference using the SABC algorithm. The following piece of code implements both the statistics learning and the inference step[3]. First the statistic that will be applied before learning transformation is defined:

```
>>> from abcpy.statistics import Identity, Statistics
>>> from abcpy.statisticslearning import SemiautomaticNN
>>> preprocessing_statistics = Identity(degree = 1, cross = False)
```

We then define the neural net to be used as a partially exchangeable network:

```
>>> phi_net = PhiNetwork()
>>> rho_net = RhoNetwork(n_parameters=2)
>>> embedding_net = PEN1(phi_net, rho_net, n_timestep = T)
```

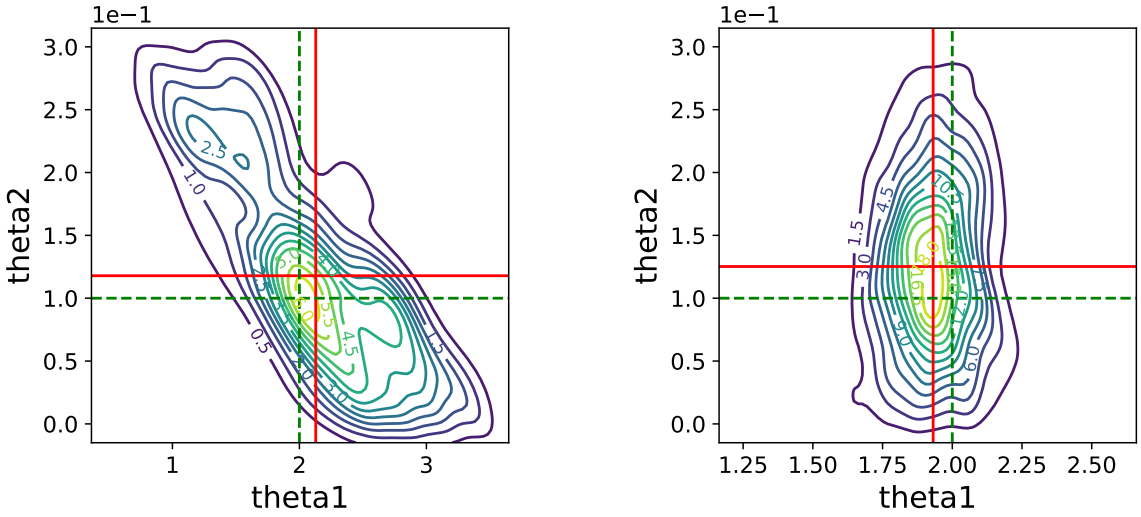
The '`SemiautomaticNN`' algorithm is employed to learn the statistics:

```
>>> summary_selection = SemiautomaticNN([lorenz], preprocessing_statistics,
...   backend, embedding_net, n_samples = 500, seed = 12)
```

The learned statistic is obtained, the distance is re-defined and inference is run using SABC:

```
>>> statistics_calculator = summary_selection.get_statistics()
>>> distance_calculator = Euclidean(statistics_calculator)
>>> sampler = SABC([lorenz], [distance_calculator], backend, kernel,
...   seed = 1)
```

[3]The code containing the definition of the neural network classes '`PhiNetwork`', '`RhoNetwork`' and '`PEN1`' is available in the replication materials.



(a) SABC with hand-defined statistics.

(b) SABC with learned statistics.

Figure 8: Contour plots for the approximate posteriors obtained with the SABC algorithm with the hand-chosen statistics defined in Hakkarainen *et al.* (2012) and the automatically learned ones. The red lines denote position of the posterior mean, while the green lines denote the position of the true parameter values. The posteriors were obtained with kernel density estimate starting from the posterior samples. The algorithm with learned statistics is able to concentrate much more around the true value of θ_1, but not so much around θ_2.

Define sampling parameters:

```
>>> steps, n_samples, n_samples_per_param, full_output = 20, 10000, 1, 1
>>> epsilon = 500
```

Sample from the posterior distribution:

```
>>> journal = sampler.sample([observation], steps, epsilon, n_samples,
...   n_samples_per_param, full_output = full_output)
```

Note that after having learned the statistics, the subsequent sampling inference step is coded in the same way as the one with the hand-chosen statistics. Figure 8 reports the approximate posterior obtained with APMCABC by using both the learned and the Hakkarainen statistics used throughout the text. Additionally, **ABCpy** also implements a newly proposed technique (Pacchiardi, Künzli, Schöngens, Chopard, and Dutta 2020), which finds a neural network transformation $f_w(\cdot)$ that is able to approximately preserve the distance of parameter space; specifically, by denoting as d_E the Euclidean distance, we look for f_w such that $d_E(\theta_i, \theta_j) \approx d_E(f_w(\boldsymbol{y}_i), f_w(\boldsymbol{y}_j))$ for any i, j. The intuition is that if the distance between the statistics is representative of the distance of the corresponding parameters, then ABC inference will perform well. Two different techniques to achieve this are implemented in the classes '`ContrastiveDistanceLearning`' and '`TripletDistanceLearning`', respectively based on comparing pairs and triplets of simulated data when learning the transformation; please refer to Pacchiardi *et al.* (2020) for more details. Finally, **ABCpy** implements the

summary statistics learning approach presented in Pacchiardi and Dutta (2020), in which an exponential family approximation is fit to the likelihood, with two neural networks representing respectively the natural parameters and the sufficient statistics of the exponential family; the latter will therefore represent the sufficient statistics of the best exponential family approximation to the model. This is implemented in the class '`ExponentialFamilyScoreMatching`'.

We note that **ABCpy** uses **Pytorch** (Paszke *et al.* 2017) to handle the neural networks and the corresponding computations. The package allows the user to specify a neural network by either passing `torch.nn` object or by specifying the width and depth of fully connected layers as a list of numbers; alternatively, a default one can be used, whose size is determined from the dimension of the data and of the parameter. As neural networks are not a fundamental part of the ABC pipeline, but only an optional preprocessing tool, **Pytorch** is not a required dependency of **ABCpy**; rather, whenever one of the neural network based routines is called, the code checks if **Pytorch** is available and, if not, asks the user to install it.

6.2. Probabilistic dependency between random variables

Since release 0.5.x of **ABCpy**, probabilistic dependency structures between random variables can be implemented. Behind the scene, **ABCpy** will represent this dependency structure as a directed acyclic graph (DAG) on which inference can be performed, in the spirit of graphical models. New random variables can be defined through operations between existing random variables. To make this concept more approachable, we now exemplify an inference problem on a probabilistic dependency structure.

Let us assume students of a school took an exam and each received a grade. Grades are stored in the variable `grades_obs` (provided in the replication materials). We believe grades depend on several variables: historical grades average, the average size of the classes, as well as the number of teachers at the school.

Here we assume the average size of a class and the number of the teachers at the school are normally distributed with some mean, depending on the budget of the school, and standard deviation equal to 1. We further assume that the budget of the school is uniformly distributed between 1 and 10 millions US dollars.

We can define these random variables and their dependencies in **ABCpy** in the following way:

```
>>> from abcpy.continuousmodels import Uniform, Normal
>>> school_budget = Uniform([[1], [10]], name = "school_budget")
>>> class_size = Normal([[800*school_budget], [1]], name = "class_size")
>>> no_teacher = Normal([[20*school_budget], [1]], name = "no_teacher")
>>> historical_mean_grade = Normal([[4.5], [0.25]],
...   name = "historical_mean_grade")
```

We model the impact of class size and the number of teachers on the final grade each student receives in the following way:

```
>>> final_grade = historical_mean_grade - 0.001 * class_size +
...   0.02 * no_teacher
```

Notice here we created a new random variable `final_grade`, by subtracting the random variables `class_size` and adding `no_teacher`, suitably scaled, from the random variable `historical_mean_grade`. The resulting graphical model is represented in Figure 9.

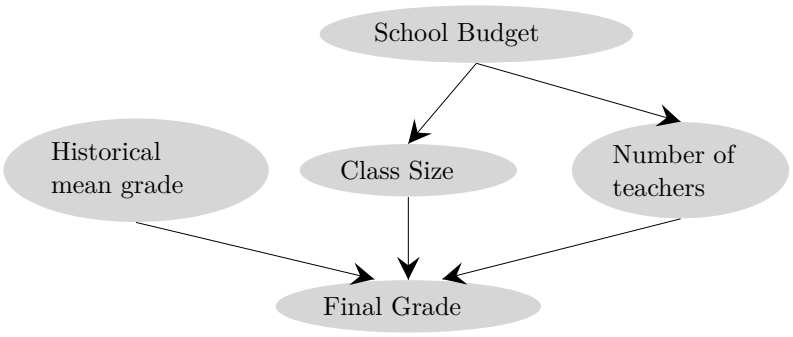


Figure 9: Dependency structure between parameters, when final grades of the students are observed.

In short, this illustrates that you can natively perform standard operations "+", "-", "*", "/" and "**" (the power operator in Python) on any two random variables to get a new random variable. It is possible to perform these operations between random variables on top of the general data types of Python (integer, float, and so on) since they are internally converted to `HyperParameters`. If additional custom operations are needed, users can implement those by sub-classing the '`ModelResultingFromOperation`' class.

6.3. Co-occurring data set

ABCpy supports inference when co-occurring (multiple) datasets are available. To illustrate how this is implemented, we extend the example from Section 6.2 to the case where we also have data on student with scholarships, stored in the variable `scholarship_obs` (provided in the replication materials).

We assume that the final mark of a student awarded a scholarship is similar to the historical mean (restricted now to scholarship students), but there is a correction dependent on the number of teachers in the school; we therefore model it in the following way:

```
>>> historical_mean_scholarship = Normal([[2], [0.5]],
...   name = "historical_mean_scholarship")
>>> final_scholarship = historical_mean_scholarship + 0.03 * no_teacher
```

With this extension, we now have two "root" `ProbabilisticModels` (random variables), namely `final_grade` and `final_scholarship` (see Figure 10), whose output can be directly compared to the observed datasets `grade_obs` and `scholarship_obs`.

Now, we need to choose summary statistics, distance, inference scheme, backend and kernel. However, since we are now considering two observed datasets, we define statistics and distances on them separately. In this example, we use the `Identity` statistics (with different polynomial expansion parameters `degree` and `cross`) and `Euclidean` for both datasets, but in general these can be different. First, we define a summary statistics for final grade and final scholarship:

```
>>> from abcpy.statistics import Identity
>>> statistics_final_grade = Identity(degree = 2, cross = False)
>>> statistics_final_scholarship = Identity(degree = 3, cross = False)
```

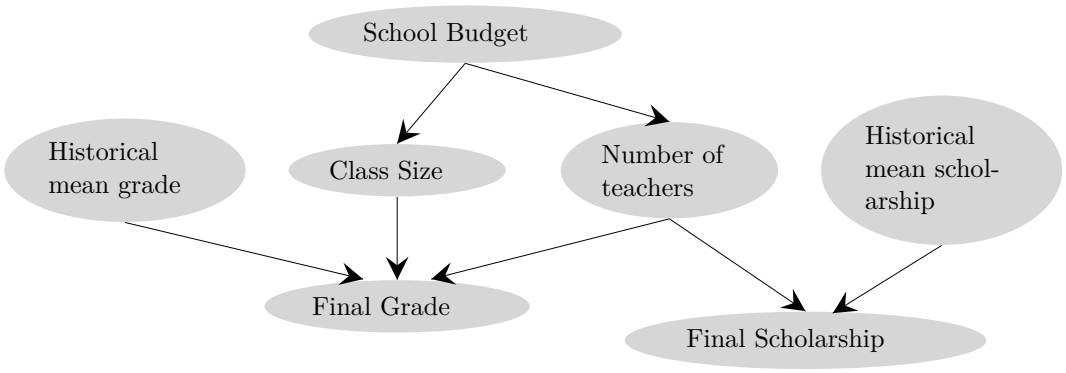


Figure 10: Dependency structure between parameters, when final grades of the students and their scholarship are observed.

Secondly, a distance measure is defined:

```
>>> from abcpy.distances import Euclidean
>>> distance_final_grade = Euclidean(statistics_final_grade)
>>> distance_final_scholarship = Euclidean(statistics_final_scholarship)
```

then a backend, a perturbation kernel, sampling parameters and the sampler are defined:

```
>>> from abcpy.backends import BackendDummy as Backend
>>> backend = Backend()

>>> from abcpy.perturbationkernel import DefaultKernel
>>> kernel = DefaultKernel([school_budget, class_size,
...   historical_mean_grade, no_teacher, historical_mean_scholarship])

>>> T, n_sample, n_samples_per_param = 3, 250, 10
>>> eps_arr = np.array([.75])
>>> eps_percentile = 10

>>> from abcpy.inferences import PMCABC
>>> sampler = PMCABC([final_grade, final_scholarship],
...   [distance_final_grade, distance_final_scholarship], backend, kernel)
```

Finally samples are obtained by:

```
>>> journal = sampler.sample([grades_obs, scholarship_obs], T, eps_arr,
...   n_sample, n_samples_per_param, eps_percentile)
```

Notice that the lists passed to the sampler and the sampling method now contain two entries, each corresponding to the different observed data sets and models respectively. Presently **ABCpy** combines different distances on different datasets by taking an equally weighted convex linear combination of the distances, however customized combination strategies can be implemented by the user.

6.4. Joint perturbation kernels

As pointed out earlier, it is possible to define joint perturbation kernels, perturbing different subsets of random variables using different kernel functions. Considering the example from Section 6.3, now we want to perturb the schools budget, scholarship and grade variables using a multivariate normal kernel, and we want to perturb the remaining parameters with a multivariate Student's T kernel. This can be implemented as follows:

```
>>> from abcpy.perturbationkernel import MultivariateNormalKernel,
...   MultivariateStudentTKernel
>>> kernel_1 = MultivariateNormalKernel([school_budget,
...   historical_mean_grade, historical_mean_scholarship])
>>> kernel_2 = MultivariateStudentTKernel([class_size, no_teacher], df = 3)

>>> # Join the defined kernels
>>> from abcpy.perturbationkernel import JointPerturbationKernel
>>> kernel = JointPerturbationKernel([kernel_1, kernel_2])
```

In the last line, we use the class '`abcpy.perturbationkernel.JointPerturbationKernel`' to join the two different kernels in a single one, by instantiating an object which takes as parameters the kernels to join; this is needed as the sampler object needs to be provided with one single kernel. Note that, in this way, it is possible to combine kernels with dependency structures among disjoint subsets of the parameters.

As a side remark, note also that we cannot use the access operator to perturb one component of a multidimensional random variable differently from another component of the same variable.

6.5. Nested parallelization

As mentioned above, **ABCpy** provides the user with seamless parallelization of ABC algorithms using **MPI** or **Spark**. Modern cluster nodes have usually multiple cores, and by default the **MPI** backend runs one simulation of the model per core. Yet, in case the model supports basic multi-threading at the level of a single machine, the backend can be accordingly configured to achieve this.

There may be however cases in which simulation from the model is extremely computationally demanding, so that each simulation has to be distributed across different nodes at the same time of the parallel execution of different simulations corresponding to different parameter values coming from the use of ABC algorithm. This is possible within **ABCpy** by using the **MPI** backend. Specifically, the model itself has to be implemented with **MPI**, i.e., it has to be independently capable of running over different nodes. In this case, the **MPI** backend in **ABCpy** controls the number of ranks that are assigned to each run of the model. For instance, consider defining the following backend for running simulations on a cluster where each node has one single processor:

```
from abcpy.backends import BackendMPI as Backend
backend = Backend(process_per_model = 2)
```

As we require two ranks per model simulation, the **MPI** backend will automatically split each model run on two different nodes. We remark instead that, if the number of cores in each

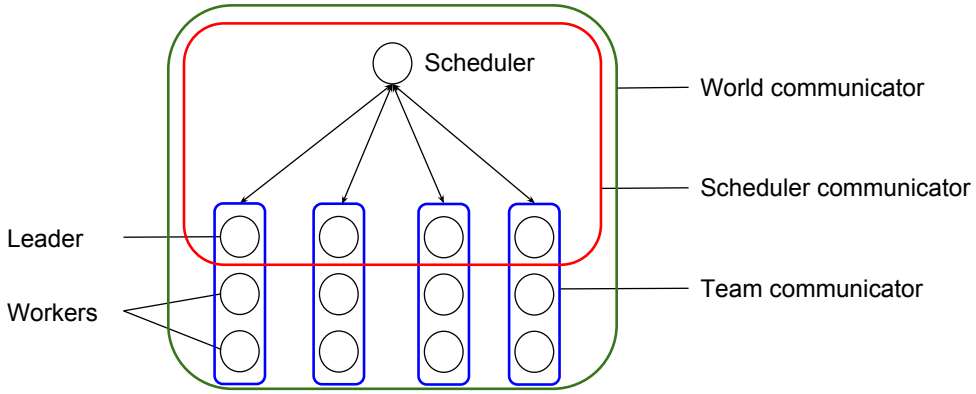


Figure 11: **Nested parallelization:** Description of the communication architecture of the nested **MPI** parallelization for **ABCpy**. Each circle represents a different rank.

node is larger than the requested `process_per_model`, then the **MPI** backend will run each simulation on two cores belonging to the same node.

Technically, **MPI** uses an object called *communicator* in order to control communication between different ranks. Therefore, in order to achieve nested parallelization **ABCpy** creates two kind of communicators: each model simulation uses a *team communicator* to parallelize the computation on the ranks allocated by the `Backend` object; moreover, the *scheduler communicator* is used by the overall master (the scheduler) to control the whole execution. The architecture is visualized in Figure 11. Note that one process of each team communicator is part of the scheduler one as well, in order for communication to be successful.

More details on the nested parallelization scheme and an example of successful application of ABC inference in such a scenario can be found in Pacchiardi *et al.* (2020).

6.6. Convergence diagnostic tools

Most algorithms implemented in **ABCpy** are SMC-type ones (particle filtering); namely, these are sequential algorithms in which a set of weighted particles represent an approximation of the target distribution and are evolved across iterations. As noted in Del Moral, Doucet, and Jasra (2007), contrarily to MCMC-type algorithms, SMC methods do not rely on ergodic properties of the transition kernels. For this reason, there is no need in SMC to perform convergence checks of the kind used in MCMC for assessing whether the chain convergent to the correct distribution.

However, the weights of the particles in these algorithms (e.g., $\omega_t^{(i)}$ for PMCABC in Algorithm 1) can degenerate to a state in which all of the weight is attributed to one single particle. For this reason, it was suggested in Del Moral *et al.* (2007) to monitor the effective sample size (ESS) as the algorithm proceeds. For a set of n particles with normalized weights $\{\omega_t^{(i)} : i = 1, \ldots, n\}$ at t-th iteration, the ESS at t-th iteration can be estimated as:

$$\mathrm{ESS}_t = \left(\sum_{i=1}^{n} (\omega_t^{(i)})^2 \right)^{-1};$$

the above reaches a maximum value of n when all weights are equal, and can be as low as 1, when all the weight is borne by one single particle. In **ABCpy**, when ABC inference is performed with a sequential algorithm, the ESS is computed at each iteration and stored. Subsequently, it is possible to produce a plot displaying its evolution with `journal.plot_ESS()`. Another possibility to assess the convergence of sequential algorithms is computing some measure of distance between the set of samples at subsequent iterations; in fact, the latter can be thought of as defining an empirical distributions approximating the target one. If the approximation is converging, the change as the algorithm proceeds would become smaller. In **ABCpy**, we have implemented a tool to compute the 2-Wasserstein distance (Peyré and Cuturi 2019) between the distributions obtained at subsequent iterations; such metric is chosen as it is a sensible measure of distance between empirical distributions. If the sequential algorithm converges to some approximate distribution, we expect the Wasserstein distance between subsequent iterations to decrease and become smaller. In **ABCpy**, the evolution of the Wasserstein distance can be plot with `journal.Wass_convergence_plot()`.

7. Discussion

There has been significant interest and efforts to develop new algorithms for ABC. A timely need in this area is to create an ecology where all these different algorithms can be integrated in a modular and user-friendly manner. It is also known that ABC algorithms can be very expensive and without HPC integration they cannot be applied to computationally intensive simulator-based models. Although the SMCABC algorithm had been parallelized before (Liepe *et al.* 2010), more efficient algorithms have since then been suggested (for instance, SABC in Albert *et al.* 2015). It is therefore very important to provide a simple way to parallelize ABC algorithms within an unified ecology and compare their parallel performance.

Our main contribution is a framework that (i) brings existing ABC algorithms under one umbrella, (ii) enables easy implementation of new ABC algorithms, and (iii) enables domain scientists to easily apply ABC to their specific problem on a broad scale using parallelization. For point (i), it is important to note that, although there is a strong current interest in ABC, there are only a few software libraries available and, up to our knowledge, none, concurrently, as complete, user-friendly, and extensible as **ABCpy**. To add to point (ii), we stress that having a unified, extensible library is one of the foundations of a principled and reproducible comparison of algorithms. In this paper, we provide a comparison of ABC algorithms from a parallel performance perspective. Hence we have reported on imbalances while parallelizing ABC type algorithms over a large number of cores. We identified inherent properties of ABC algorithms that make efficient parallelization difficult, classified ABC algorithms based on the imbalances, and tried to find the most suitable algorithms capable of utilizing a large parallel architecture through empirical comparisons.

Acknowledgments

The work was supported by Swiss National Science Foundation Grant No. 105218_163196 (Statistical Inference on Large-Scale Mechanistic Network Models). We also thank Swiss National Super Computing Center for providing computing resources. LP is supported by the EPSRC and MRC through the OxWaSP CDT program (EP/L016710/1).

References

Albert C, Künsch HR, Scheidegger A (2015). "A Simulated Annealing Approach to Approximate Bayesian Computations." *Statistics and Computing*, **25**, 1217–1232. doi:10.1007/s11222-014-9507-8.

Amdahl GM (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, pp. 483–485. ACM.

An Z, Nott DJ, Drovandi C (2020). "Robust Bayesian Synthetic Likelihood via a Semi-Parametric Approach." *Statistics and Computing*, **30**(3), 543–557. doi:10.1007/s11222-019-09904-x.

Andrieu C, Roberts GO (2009). "The Pseudo-Marginal Approach for Efficient Monte Carlo Computations." *The Annals of Statistics*, **37**(2), 697–725. doi:10.1214/07-aos574.

Beaumont MA (2010). "Approximate Bayesian Computation in Evolution and Ecology." *Annual Review of Ecology, Evolution, and Systematics*, **41**(1), 379–406. doi:10.1146/annurev-ecolsys-102209-144621.

Beaumont MA, Zhang W, Balding DJ (2002). "Approximate Bayesian Computation in Population Genetics." *Genetics*, **162**(4), 2025–2035. doi:10.1093/genetics/162.4.2025.

Beazley DM, *et al.* (1996). "**SWIG**: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In *Tcl/Tk Workshop*, volume 43, p. 74.

Bernton E, Jacob PE, Gerber M, Robert CP (2019). "Approximate Bayesian Computation with the Wasserstein Distance." *Journal of the Royal Statistical Society B*, **81**(2), 235–269. doi:10.1111/rssb.12312.

Blum MGB, François O (2010). "Non-Linear Regression Models for Approximate Bayesian Computation." *Statistics and Computing*, **20**(1), 63–73. doi:10.1007/s11222-009-9116-0.

Cappé O, Guillin A, Marin JM, Robert CP (2004). "Population Monte Carlo." *Journal of Computational and Graphical Statistics*, **13**(4), 907–929. doi:10.1198/106186004x12803.

Carlson JL (2013). ***Redis** in Action*. Manning Publications, USA.

Chiachio M, Beck JL, Chiachio J, Rus G (2014). "Approximate Bayesian Computation by Subset Simulation." *SIAM Journal on Scientific Computing*, **36**(3), A1339–A1358. doi:10.1137/130932831.

Csilléry K, François O, Blum MGB (2012). "**abc**: An R Package for Approximate Bayesian Computation (ABC)." *Methods in Ecology and Evolution*, **3**(3), 475–479. doi:10.1111/j.2041-210x.2011.00179.x.

Dask Development Team (2016). ***Dask**: Library for Dynamic Task Scheduling*. URL https://dask.org/.

Dean J, Ghemawat S (2008). "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, **51**(1), 107–113. doi:10.1145/1327452.1327492.

Del Moral P, Doucet A, Jasra A (2007). "Sequential Monte Carlo for Bayesian Computation." *Bayesian Statistics*, **8**, 1–34.

Del Moral P, Doucet A, Jasra A (2012). "An Adaptive Sequential Monte Carlo Method for Approximate Bayesian Computation." *Statistics and Computing*, **22**(5), 1009–1020. doi:10.1007/s11222-011-9271-y.

Drovandi CC, Pettitt AN (2011). "Estimation of Parameters for Macroparasite Population Evolution Using Approximate Bayesian Computation." *Biometrics*, **67**(1), 225–233. doi:10.1111/j.1541-0420.2010.01410.x.

Fearnhead P, Prangle D (2012). "Constructing Summary Statistics for Approximate Bayesian Computation: Semi-Automatic Approximate Bayesian Computation." *Journal of the Royal Statistical Society B*, **74**(3), 419–474. doi:10.1111/j.1467-9868.2011.01010.x.

Graham RL (1966). "Bounds for Certain Multiprocessing Anomalies." *Bell Labs Technical Journal*, **45**(9), 1563–1581. doi:10.1002/j.1538-7305.1966.tb01709.x.

Gutmann MU, Corander J (2016). "Bayesian Optimization for Likelihood-Free Inference of Simulator-Based Statistical Models." *The Journal of Machine Learning Research*, **17**(1), 4256–4302. URL http://jmlr.org/papers/v17/15-017.html.

Gutmann MU, Dutta R, Kaski S, Corander J (2018). "Likelihood-Free Inference via Classification." *Statistics and Computing*, **28**, 411–425. doi:10.1007/s11222-017-9738-6.

Hakkarainen J, Ilin A, Solonen A, Laine M, Haario H, Tamminen J, Oja E, Järvinen H (2012). "On Closure Parameter Estimation in Chaotic Systems." *Nonlinear Processes in Geophysics*, **19**(1), 127–143. doi:10.5194/npg-19-127-2012.

Harris CR, Millman KJ, Van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, Van Kerkwijk MH, Brett M, Haldane A, Fernández del Río J, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020). "Array Programming with **NumPy**." *Nature*, **585**(7825), 357–362. doi:10.1038/s41586-020-2649-2.

Hastings WK (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika*, **57**(1), 97–109. doi:10.1093/biomet/57.1.97.

IPython Development Team (2021). ***ipyparallel*** *Docs*. URL https://github.com/ipython/ipyparallel.

Jabot F, Faure T, Dumoulin N, Albert C (2015). ***EasyABC****: Efficient Approximate Bayesian Computation Sampling Schemes*. R package version 1.5, URL https://CRAN.R-project.org/package=EasyABC.

Jennings E, Madigan M (2017). "**astroABC**: An Approximate Bayesian Computation Sequential Monte Carlo Sampler for Cosmological Parameter Estimation." *Astronomy and Computing*, **19**, 16–22. doi:10.1016/j.ascom.2017.01.001.

Jiang B, Wu T, Zheng C, Wong WH (2017). "Learning Summary Statistic for Approximate Bayesian Computation via Deep Neural Network." *Statistica Sinica*, pp. 1595–1618. doi:10.5705/ss.202015.0340.

Klinger E, Rickert D, Hasenauer J (2018). "**pyABC**: Distributed, Likelihood-Free Inference." *Bioinformatics*, **34**(20), 3591–3593. doi:10.1093/bioinformatics/bty361.

Kulakova L, Angelikopoulos P, Hadjidoukas PE, Papadimitriou C, Koumoutsakos P (2016). "Approximate Bayesian Computation for Granular and Molecular Dynamics Simulations." In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16, pp. 4:1–4:12. ACM. doi:10.1145/2929908.2929918.

Lenormand M, Jabot F, Deffuant G (2013). "Adaptive Approximate Bayesian Computation for Complex Models." *Computational Statistics*, **28**(6), 2777–2796. doi:10.1007/s00180-013-0428-3.

Liepe J, Barnes C, Cule E, Erguler K, Kirk P, Toni T, Stumpf MPH (2010). "**ABC-SysBio** – Approximate Bayesian Computation in Python with GPU Support." *Bioinformatics*, **26**(14), 1797–1799. doi:10.1093/bioinformatics/btq278.

Lintusaari J, Gutmann MU, Dutta R, Kaski S, Corander J (2016). "Fundamentals and Recent Developments in Approximate Bayesian Computation." *Systematic Biology*, **66**(1), e66–e82. doi:10.1093/sysbio/syw077.

Lintusaari J, Vuollekoski H, Kangasrääsiö A, Skytén K, Järvenpää M, Marttinen P, Gutmann MU, Vehtari A, Corander J, Kaski S (2018). "**ELFI**: Engine for Likelihood-Free Inference." *The Journal of Machine Learning Research*, **19**(1), 643–649.

Lorenz EN (1995). "Predictability: A Problem Partly Solved." In *Proceedings of the Seminar on Predictability, 4-8 September 1995*, volume 1, pp. 1–18. European Center on Medium Range Weather Forecasting, European Center on Medium Range Weather Forecasting, Shinfield Park, Reading.

Marin JM, Pudlo P, Robert C, Ryder R (2012). "Approximate Bayesian Computational Methods." *Statistics and Computing*, **22**(6), 1167–1180. doi:10.1007/s11222-011-9288-2.

Martinez EA, Muschik CA, Schindler P, Nigg D, Erhard A, Heyl M, Hauke P, Dalmonte M, Monz T, Zoller P, Blatt R (2016). "Real-Time Dynamics of Lattice Gauge Theories with a Few-Qubit Quantum Computer." *Nature*, **534**(7608), 516–519. doi:10.1038/nature18318.

Message Passing Interface Forum (2012). ***MPI**: A Message Passing Interface Standard.*

Pacchiardi L, Dutta R (2020). "Score Matched Conditional Exponential Families for Likelihood-Free Inference." arXiv:2012.10903 [stat.ME], URL https://arxiv.org/abs/2012.10903.

Pacchiardi L, Künzli P, Schöngens M, Chopard B, Dutta R (2020). "Distance-Learning for Approximate Bayesian Computation to Model a Volcanic Eruption." *Sankhya B*, pp. 1–30. doi:10.1007/s13571-019-00208-8.

Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017). "Automatic Differentiation in **PyTorch**." In *NIPS Autodiff Workshop*.

Pérez F, Granger BE (2007). "**IPython**: A System for Interactive Scientific Computing." *Computing in Science and Engineering*, **9**(3), 21–29. doi:10.1109/mcse.2007.53.

Peyré G, Cuturi M (2019). "Computational Optimal Transport: With Applications to Data Science." *Foundations and Trends in Machine Learning*, **11**(5-6), 355–607. doi:10.1561/2200000073.

Pritchard JK, Seielstad MT, Perez-Lezaun A, Feldman MW (1999). "Population Growth of Human Y Chromosomes: A Study of Y Chromosome Microsatellites." *Molecular Biology and Evolution*, **16**(12), 1791–1798. doi:10.1093/oxfordjournals.molbev.a026091.

Pudlo P, Marin JM, Estoup A, Cornuet JM, Gautier M, Robert CP (2015). "Reliable ABC Model Choice via Random Forests." *Bioinformatics*, **32**(6), 859–866. doi:10.1093/bioinformatics/btv684.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Robert CP, Casella G (2005). *Monte Carlo Statistical Methods*. Springer-Verlag.

Schaye J, Crain RA, Bower RG, Furlong M, Schaller M, Theuns T, Dalla Vecchia C, Frenk CS, McCarthy IG, Helly JC, Jenkins A, Rosas-Guevara YM, White SDM, Baes M, Booth CM, Camps P, Navarro JF, Qu Y, Rahmati A, Sawala T, Thomas PA, Trayford J (2015). "The EAGLE Project: Simulating the Evolution and Assembly of Galaxies and Their Environments." *Monthly Notices of the Royal Astronomical Society*, **446**(1), 521–554. doi:10.1093/mnras/stu2058.

Stram AH, Marjoram P, Chen GK (2015). "**al3c**: High-Performance Software for Parameter Inference Using Approximate Bayesian Computation." *Bioinformatics*, **31**(21), 3549–3551. doi:10.1093/bioinformatics/btv393.

Tavaré S, Balding DJ, Griffiths RC, Donnelly P (1997). "Inferring Coalescence Times From DNA Sequence Data." *Genetics*, **145**(2), 505–518. doi:10.1093/genetics/145.2.505.

Thomas O, Dutta R, Corander J, Kaski S, Gutmann MU (2021). "Likelihood-Free Inference by Ratio Estimation." *Bayesian Analysis*, pp. 1–31. doi:10.1214/20-ba1238.

Toni T, Welch D, Strelkowa N, Ipsen A, Stumpf MPH (2009). "Approximate Bayesian Computation Scheme for Parameter Inference and Model Selection in Dynamical Systems." *Journal of the Royal Society Interface*, **31**(6), 187–202. doi:10.1098/rsif.2008.0172.

Turchin P, Currie TE, Turner EAL, Gavrilets S (2013). "War, Space, and the Evolution of Old World Complex Societies." *Proceedings of the National Academy of Sciences of the United States of America*, **110**(41), 16384–16389. doi:10.1073/pnas.1308825110.

Van Rossum G, *et al.* (2021). *Python Programming Language*. URL https://www.python.org/.

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, Van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat I, Feng Y, Moore

EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, Van Mulbregt P, **SciPy** 10 Contributors (2020). "**SciPy** 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods*, **17**, 261–272. doi:10.1038/s41592-019-0686-2.

Wilks DS (2005). "Effects of Stochastic Parametrizations in the Lorenz '96 System." *Quarterly Journal of the Royal Meteorological Society*, **131**(606), 389–407. doi:10.1256/qj.04.03.

Wiqvist S, Mattei PA, Picchini U, Frellsen J (2019). "Partially Exchangeable Networks and Architectures for Learning Summary Statistics in Approximate Bayesian Computation." In K Chaudhuri, R Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6798–6807. PMLR. URL http://proceedings.mlr.press/v97/wiqvist19a.html.

Wood SN (2010). "Statistical Inference for Noisy Nonlinear Ecological Dynamic Systems." *Nature*, **466**(7310), 1102–1104. doi:10.1038/nature09319.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016). "**Apache Spark**: A Unified Engine for Big Data Processing." *Communications of the ACM*, **59**(11), 56–65. doi:10.1145/2934664.

A. Details on parameter inference in the Lorenz95 model

We give here more details on the Lorenz95 model considered as a running example throughout the main text. We used a modification of the original weather prediction model of Lorenz (1995) when fast climate variables are unobserved Wilks (2005).

- **Model**: We assume that weather stations measure a high-dimensional time-series of slow climate variables $(y_k^{(t)}, k = 1, \ldots, 40)$, following a coupled stochastic differential equation (SDE), called the forecast model (Wilks 2005):

$$\frac{dy_k^{(t)}}{dt} = -y_{k-1}^{(t)}(y_{k-2}^{(t)} - y_{k+1}^{(t)}) - y_k^{(t)} + F - g(y_k^{(t)}, \theta) + \eta_k^{(t)},$$
$$g(y_k^{(t)}, \theta) = \sum_{i=1}^{2} \theta_i \left(y_k^{(t)}\right)^{i-1},$$

  for $k = 1, \ldots, 40$ and where $F = 10$. Assuming that the initial values $y_k^{(0)}, k = 1, \ldots, 40$ are known, we consider the interval $[0, 4]$ in the time units of the model. The function $g(y_k^{(t)}, \theta)$ represents a deterministic parametrization of the net effect of the unobserved fast weather variables on the observable $y_k^{(t)}$, and $\eta_k^{(t)}$ is a stochastic forcing term representing the uncertainty due to the forcing of the fast variables. The model is cyclic in the variables $y_k^{(t)}$, and the coupled SDEs do not have an analytic solution.

  We discretize the time-interval $[0, 4]$ into T equal steps of length $\Delta t = 4/T$, and solve the SDEs by using a 4th order Runge-Kutta solver at these time-points. Following Wilks (2005) the stochastic forcing term is updated for an interval of Δt as

$$\eta_k^{(t+\Delta t)} = \phi \eta_k^{(t)} + (1 - \phi^2)^{\frac{1}{2}} e^{(t)}, t \in \{0, \Delta t, \ldots, T\Delta t\}$$

  where the $e^{(t)}$ are independent normal random variables with standard deviation σ_e and $\eta^{(0)} = (1 - \phi^2)^{\frac{1}{2}} e^{(0)}$. Here T is chosen to be 1024.

- **Parameters**: We fix $\phi = 0.4,\ \sigma_e = 1$ and infer the parameters $\theta = (\theta_1, \theta_2)$.

- **Prior**: We assume uniform prior distributions with ranges $[0.5, 3.5]$ and $[0, 0.3]$ for the parameters θ_1 and θ_2, respectively; this is motivated by the observations in Hakkarainen *et al.* (2012).

- **Observed dataset ($\boldsymbol{y}_0$)**: A multivariate time series computed by solving the SDEs numerically, as described above, with $\theta^0 = (\theta_1^o, \theta_2^o) = (2.0, 0.1)$ over a period $t \in [0, 4]$ with $T = 1024$.

- **Statistics**: The six summary statistics suggested by Hakkarainen *et al.* (2012): for each k, we compute the mean, variance and auto-co-variance with time lag one of $y_k^{(t)}$, co-variance of $y_k^{(t)}$ with its neighbor $y_{k+1}^{(t)}$ and cross-co-variance of $y_k^{(t)}$ with its two neighbors $y_{k-1}^{(t)}$ and $y_{k+1}^{(t)}$ for time lag one. These values are all averaged over $k = 1, \ldots, 40$ since the model is symmetric with respect to the index k.

- **Distance**: Euclidean distance in both the experiments with hand-chosen and learned statistics.

- **Experimental setting**: All of the algorithms considered in the main text (PMCABC, APMCABC, SABC and ABCsubsim) are sequential population algorithms. We run all of them for 20 steps (expect for PMCABC that is run for only 3 steps, for the reasons described in Section 5.3) and drew 10,000 samples at each step. Therefore, at the end we are provided with 10,000 samples from the approximate posterior distribution of the parameters. A multivariate Normal distribution with 3 degrees of freedom was used as the perturbation kernel and the Euclidean distance as the discrepancy measure. For the PMCABC algorithm, we chose an initial threshold value $\epsilon = 500$ for the first step of the algorithm. For the subsequent steps, the 0.1-quantile of the distances, between observed and simulated pseudo datasets from earlier steps, is considered as the threshold value. For the SABC algorithm, we used $\epsilon = 500$ in analogy with the PMCABC one. All of the other parameters are left at the default value of the package. To choose the above tuning parameters we run multiple pilot runs to detect the parameter values providing the most stable and the best convergence results of the ABC approximate posterior distribution. After this first step, we proceed to the performance evaluation tasks described in the main text.

Affiliation:

Ritabrata Dutta
Department of Statistics
University of Warwick, United Kingdom
E-mail: Ritabrata.Dutta@warwick.ac.uk

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
November 2021, Volume 100, Issue 7 *Submitted:* 2018-09-07
doi:10.18637/jss.v100.i07 *Accepted:* 2021-02-08