Approximate Bayesian Computation

MATH-414, Stochastic Simulation Project

Mahammad Ismayilzada

January 15, 2024

1 Introduction

Many problems in data science involve estimating a set of parameters $\theta \in \Theta$ for a model $\mathcal{M}$ that describes the processes underlying the problem of interest. In the Bayesian inference paradigm, after having observed some data $\mathcal{D}$, the posterior distribution of θ is determined by the Bayes rule as follows:

$$f(\theta|\mathcal{D}) = \mathbb{P}(\mathcal{D}|\theta)\pi(\theta)/\mathbb{P}(\mathcal{D}) \tag{1}$$

where $\mathbb{P}(\mathcal{D}) = \int_\Theta \mathbb{P}(\mathcal{D}|\theta)\pi(\theta)d\theta$ is called the *evidence* and represents the normalizing constant.

Stochastic simulation approaches for generating observations from the posterior distribution $f(\theta|\mathcal{D})$ often depend on knowing explicitly the likelihood function $\mathbb{P}(\mathcal{D}|\theta)$, possibly up to a multiplicative constant (i.e. being able to evaluate it for any θ and $\mathcal{D}$). However, for many complex probabilistic models, such likelihoods are either inaccessible or computationally prohibitive to evaluate, so one has to resort to the so-called *likelihood-free* methods, of which, most notably the *Approximate Bayesian Computation (ABC)*.

In this report, we consider two variations of the ABC method, namely, *ABC Rejection* and *ABC Markov Chain Monte Carlo (ABC MCMC)* [Marjoram et al., 2003] algorithms (see project description for details), apply them in the context of two problems, one synthetic problem and one real-world problem and discuss their performances.

2 Theory

In this section, we provide some theoretical ground on the convergence of these two methods to desired posterior distribution.

2.1 Convergence of ABC Rejection

First, we would like to show that the ABC Rejection algorithm generates samples distributed as $f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)$.

Proof: Since ABC Rejection algorithm generates samples θ^* along with some data $\mathcal{D}^*$, let's consider the joint distribution of the accepted $(\theta^*,\ \mathcal{D}^*)$ and denote it as $f(\theta^*, \mathcal{D}^*)$. Given that the algorithm only accepts those data $\mathcal{D}^*$ that satisfies the condition $\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon$ (i.e. sufficiently close to the observed data), we have the following:

$$f(\theta^*, \mathcal{D}^*) = \mathbb{P}(\mathcal{D}^*|\theta^*)\pi(\theta^*)\mathbb{I}_\varepsilon(\rho(\mathcal{D}^*, \mathcal{D})) \tag{2}$$

where $\mathbb{I}_\varepsilon(\rho(\mathcal{D}^*, \mathcal{D})) = 1$ iff $\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon$ and zero otherwise. Now if we marginalize over $\mathcal{D}^*$, we get the following marginal distribution for θ^*:

$$f(\theta^*) = \int_{\mathbb{D}} \mathbb{P}(\mathcal{D}^*|\theta^*)\pi(\theta^*)\mathbb{I}_\varepsilon(\rho(\mathcal{D}^*, \mathcal{D}))d\mathcal{D}^* = \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta^*)\pi(\theta^*) \propto f(\theta^*|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) \tag{3}$$

which we achieve by integrating over only the region of $\mathbb{D}$ that satisfies the condition $\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon$ as the rest is zero. Hence, we obtain a marginal distribution that is proportional to the $f(\theta^*|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)$ indicating that the algorithm samples from this distribution which is we what we wanted to show.

2.2 Stationarity of ABC MCMC

Second, we want to prove that $f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)$ is the stationary distribution of the chain generated by the ABC MCMC algorithm.

Proof: We essentially need to show that $f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)$ satisfies the detailed balance condition as below:

$$f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) r(\theta \to \theta^*) = f(\theta^*|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) r(\theta^* \to \theta) \tag{4}$$

where $r(\theta \to \theta^*)$ denotes the transition kernel of the chain. Without loss of generality, let's assume that $\theta^* \neq \theta$ and the acceptance probability α is less than 1:

$$\alpha(\theta, \theta^*) = \frac{\pi(\theta^*) q(\theta^*, \theta)}{\pi(\theta) q(\theta, \theta^*)} \leq 1 \tag{5}$$

We also know from the ABC MCMC algorithm that the transition probability $r(\theta \to \theta^*)$ can be defined as the following:

$$r(\theta \to \theta^*) = q(\theta, \theta^*) \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta^*) \alpha(\theta, \theta^*) \tag{6}$$

since we transition from θ to θ^* with the probability that the chosen proposal distribution generates it and the data generated from θ^* is sufficiently close to observed data and it is accepted with the probability $\alpha(\theta, \theta^*)$. Then we can write $f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) r(\theta \to \theta^*)$ as following:

$$\begin{aligned} f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) r(\theta \to \theta^*) = f(\theta|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) q(\theta, \theta^*) \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta^*) \alpha(\theta, \theta^*) = \\ \frac{\mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta) \pi(\theta)}{\mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)} \times q(\theta, \theta^*) \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta^*) \times \frac{\pi(\theta^*) q(\theta^*, \theta)}{\pi(\theta) q(\theta, \theta^*)} = \\ \frac{\mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta^*) \pi(\theta^*)}{\mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon)} \times q(\theta^*, \theta) \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta) = \\ f(\theta^*|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) q(\theta^*, \theta) \mathbb{P}(\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon|\theta) \alpha(\theta^*, \theta) = f(\theta^*|\rho(\mathcal{D}^*, \mathcal{D}) < \varepsilon) r(\theta^* \to \theta) \end{aligned} \tag{7}$$

Hence we achieve the detailed balance equation which makes the distribution stationary.

3 Experiments

In this section, we validate the theoretical findings with experimental results on two problems.

3.1 Synthetic problem

In this section, we consider a synthetic (academic) problem where the likelihood function and true posterior distribution are known so that we can compare the results of the simulation results with the true posterior.

3.1.1 Setup

In this problem, the observed data $\mathcal{D} = \{x_i\}_{i=1}^M \subset \mathbb{R}$ is an iid sample drawn with probability 1/2 from $\mathcal{N}(\theta, \sigma_1^2)$ and with probability 1/2 from $\mathcal{N}(\theta + a, \sigma_1^2)$. As a prior we take $\pi = \mathcal{N}(0, \sigma^2)$. Then, the posterior distribution is a Gaussian mixture given by

$$f(\theta \mid \mathcal{D}) = \alpha \mathcal{N}\left(\frac{\sigma^2}{\sigma^2 + \sigma_1^2/M} \bar{x}, \frac{\sigma_1^2}{M + \sigma_1^2/\sigma^2}\right) + (1 - \alpha) \mathcal{N}\left(\frac{\sigma^2}{\sigma^2 + \sigma_1^2/M} (\bar{x} - a), \frac{\sigma_1^2}{M + \sigma_1^2/\sigma^2}\right) \tag{8}$$

with

$$\alpha = \frac{1}{1 + \exp\left\{a\left(\bar{x} - \frac{a}{2}\right) \frac{M}{M\sigma^2 + \sigma_1^2}\right\}} \tag{9}$$

We consider the following parameter values: $M = 100, \sigma_1^2 = 0.1, \sigma^2 = 3, a = 1, \bar{x} = 0$. Discrepancy metric is defined as the following:

$$\rho(S(\mathcal{D}^*), S(\mathcal{D})) = |\bar{x}^* - \bar{x}| \tag{10}$$

where $S(\cdot)$ is the summary statistics of the data and in this example, it is the mean function.

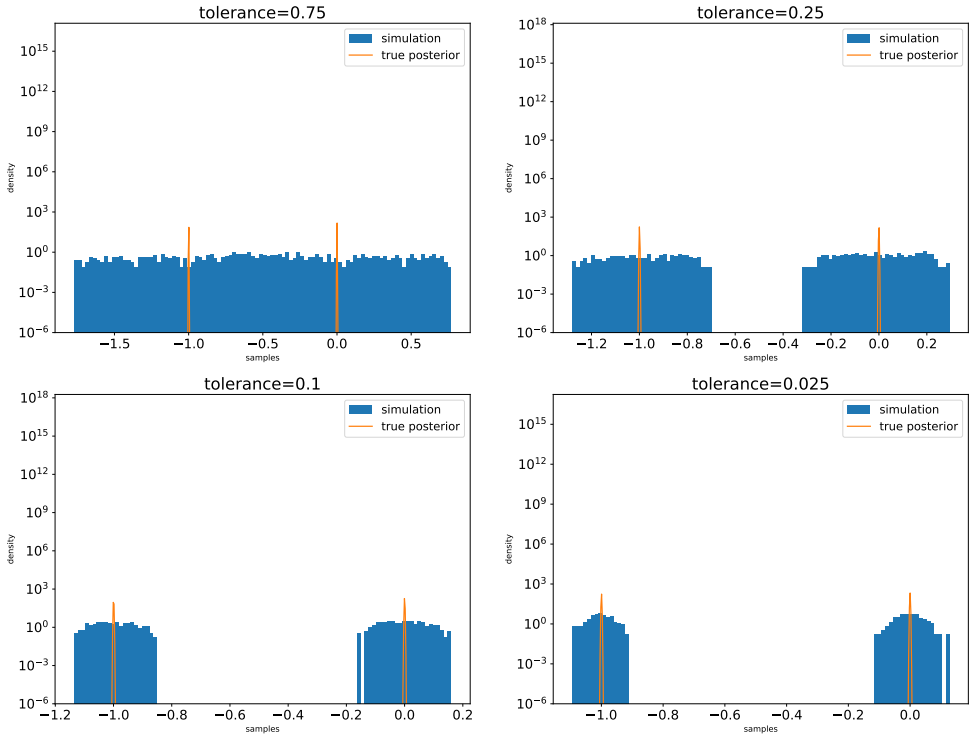

Figure 1: Histogram of ABC Rejection simulation results (synthetic problem) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.

3.1.2 Experiments with ABC Rejection

First, we apply the basic ABC Rejection algorithm (Listing 2) to our synthetic problem and run the algorithm for different tolerances $\varepsilon = \{0.75, 0.25, 0.1, 0.025\}$, until $N = 500$ samples are accepted (Listing 4). This algorithm achieves the following acceptance rates accordingly (given as a percentage) $r = \{30.8\%, 10.4\%, 4.1\%, 1.0\%\}$. Figure 1 shows the histogram of samples compared to the given true mixture distribution for all tolerance levels. We notice that the choice of ε reflects a tension between the computability and accuracy of the method as we decrease the tolerance level, the acceptability rate decreases, but the accuracy increases. The true posterior distribution in the given example is a mixture of two normal distributions centered around -1.0 and 0 respectively and what we observe is that when $\varepsilon = 0.75$, more samples are accepted and the resulting distribution is uniform-like. However, as we decrease the value of ε, our simulation samples start to form the expected islands of normal distributions with a decreasing variance.

3.1.3 Experiments with ABC MCMC

In this question, we apply the ABC MCMC method (Listing 3) with random walk proposal $q(\theta, \cdot) = \mathcal{N}(\theta, \nu^2)$ to the given academic example for $\nu^2 = \{0.5, 1, 2, 4, 8\}$ and $\varepsilon = \{0.75, 0.25, 0.1, 0.025\}$, until we achieve an effective sample size $N_{eff} \approx 500$ (Listing 5). To achieve this effective size, we use the concept of *burn-in period*, that is the number of points that have to be discarded before the chain is deemed to be in stationary mode [Dirk P. Kroese, 2011]. In particular, we set the burn-in size to 10% of the desired size (i.e. N_{eff}), hence, run the algorithm for $N = N_{eff} + N_{eff} * 0.1$ iterations and discard the first $N_{eff} * 0.1$ points to obtain N_{eff} number of stationary points. Table 1 shows the acceptance rates (in percentage) for different combinations of ν^2 and ε values. What we observe is

ν^2	$\varepsilon = 0.75$	$\varepsilon = 0.25$	$\varepsilon = 0.1$	$\varepsilon = 0.025$
0.5	51.1%	13.6%	8.5%	2.2%
1	38.4%	15.1%	7.6%	0.7%
2	32.5%	9.2%	4.7%	1.1%
4	28.7%	9.5%	3.6%	0.5%
8	19.8%	7.2%	3.6%	0.5%

Table 1: ABC MCMC simulation acceptance rates (synthetic problem) for different values of proposal variance (ν^2) and tolerance level (ε)

that the acceptance rate largely goes down as the variance of the proposal distribution increases and this aligns with our expectation that the larger variance implies more risky explorations (i.e. values that deviate more from the current accepted parameters). Moreover, compared to ABC Rejection algorithm, acceptance rates are also much higher for most of the tolerance levels pointing to the higher efficiency of the ABC MCMC algorithm. Figure 2 on the other hand shows that ABC MCMC is not only more efficient but also more accurate in simulating the posterior distribution (for $\nu^2 = 1$). Histograms of the distributions for other values of ν^2 can be found in Figures 5, 6, 7, 8 in Appendix. We observe a similar phenomenon in these settings as well.

3.2 Real-world problem

3.2.1 Setup

In this section, we consider a real-world problem where we want to estimate the posterior distribution of parameters for a dynamical model of the pharmacokinetics of Theophylline [Picchini, 2014], a drug used in the treatment of asthma and chronic obstructive pulmonary disease. In pharmacokinetics one aims to study a drug of interest by describing its absorption, distribution, metabolism, and excretion mechanisms from the body. A fundamental concept in pharmacokinetics is drug clearance, that is, elimination of drugs from the body. Let X_t be the level of Theophylline concentration in blood at time t, then the evolution of X_t over time can be modeled by means of the following stochastic differential equation (SDE):

$$dX_t = \left(\frac{DK_aK_e}{Cl} e^{-K_a t} - K_e X_t \right) dt + \sigma dW_t \tag{11}$$

where D is the known drug oral dose received by a subject, K_e is the elimination rate constant, K_a the absorption rate constant, Cl the clearance of the drug, and σ the intensity of intrinsic stochastic noise driven by the Brownian motion W_t. We consider nine blood samples taken at $\{t_1, ..., t_9\} = \{0.25, 0.5, 1, 2, 3.5, 5, 7, 9, 12\}$. The drug oral dose is chosen to be $D = 4mg$ and is administered at $t_0 = 0$. Initial drug concentration in blood is $X_0 = 0$. We are interested in estimating the posterior distribution of $\theta = (K_a, K_e, Cl, \sigma)$ for which the following prior distributions are set: $logK_a \sim \mathcal{N}(0.14, 0.4^2)$, $logK_e \sim \mathcal{N}(-2.7, 0.6^2)$, $logCl \sim \mathcal{N}(-3, 0.8^2)$, $log\sigma \sim \mathcal{N}(-1.1, 0.3^2)$.

3.2.2 Data Generation

We first consider the data generation process for this model. While in the synthetic problem, it was easier to generate the data given the model parameters since we were provided the explicit likelihood function, in this problem data generation process is indirectly modeled as a stochastic differential equation. Although the given process X_t is Gaussian and can be simulated exactly by solving the Equation 11 for X_t and estimating its mean and variance using Monte Carlo simulation, we instead opt for generating our data using Euler-Maruyama discretization[1] with sufficiently small time step for practical purposes. Implementation of this method is given in Listing 6. We set the discretization step to a sufficiently small value of $dt = 0.005$. Given the model parameters $\theta = (1.5, 0.08, 0.04, 0.2)$, we use the Euler-Maruyama method to simulate model 11 and generate the observed data at sampling times $\{t_1, ..., t_9\}$ (Listing 7). This process results in the following data (rounded to two decimal places): $X_t = \{2.37, 4.01, 5.85, 6.91, 6.28, 5.97, 5.40, 5.15, 3.88\}$. The resulting data seems reasonable given our

[1] https://en.wikipedia.org/wiki/Euler-Maruyama_method

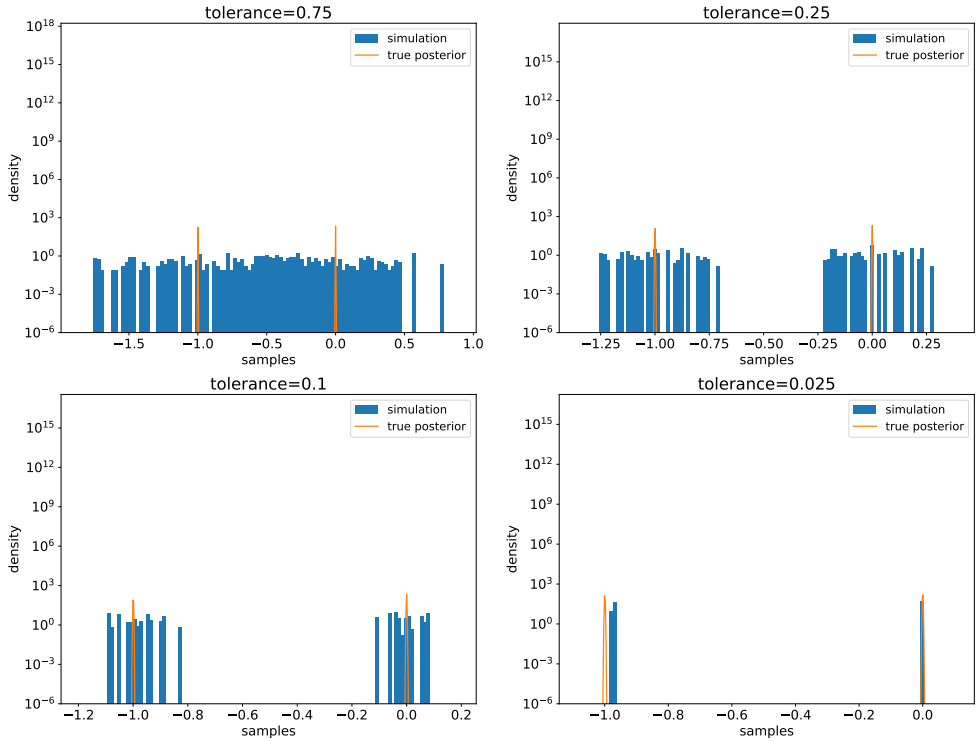

Figure 2: Histogram of ABC MCMC simulation results ($\nu^2 = 1$) (synthetic problem) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.

model which characterizes first an exponential increase in the concentration level followed by a gradual decrease.

3.2.3 Parameter Estimation

Next, we need a summary statistics function $S(\cdot)$ used in computing the discrepancy metric. In this problem, we consider the following setup to construct this function. We first fit the following multivariate linear regression model to estimate our model parameters θ from the data $\mathcal{D}$:

$$\theta = \beta_0 + \beta_1 x_1 + ... + \beta_9 x_9 + \xi \tag{12}$$

where $\theta = (K_a, K_e, Cl, \sigma)$ is a vector of parameters, $\beta_i \in \mathbb{R}^4, i = 0, ..., 9$ are unknown regression coefficients, $\{x_i\}_{i=1}^9$ are the generated data $\mathcal{D}$ and $\xi = (\xi_1, ..., \xi_4)$ is a random vector with zero mean, independent components and constant variance. In order to train this model, we prepare the training data of size P by sampling P parameters $\theta^{(1)}, ..., \theta^{(P)}$ from the respective prior distributions and generating corresponding data $\mathcal{D}^{(1)}, ..., \mathcal{D}^{(P)}$. Given the estimated regression coefficients $\{\hat{\beta}_i\}_{i=0}^9$, we define the summary statistics of data $\mathcal{D}$ as the following:

$$S(\mathcal{D}) = \mathbb{E}(\theta|\mathcal{D}) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_9 x_9 \in \mathbb{R}^4 \tag{13}$$

For our experiments, we generate a training dataset of size $P = 1000$ and train four models of 12 respectively for our four parameters using `statsmodels` Python package[2] (Listing 8). We report the respective values of R^2 that show the proportion of variance in the dependent variable explained by the

[2]https://www.statsmodels.org/stable/index.html

Parameter	β_0	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9
K_a	1.244	0.211	0.132	0.029	−0.143	−0.120	0.058	0.072	−0.035	0.013
K_e	0.069	0.005	−0.002	−0.003	0.005	0.004	0.005	−0.003	−0.004	−0.009
Cl	0.111	−0.009	0.001	0.006	−0.001	−0.004	−0.000	−0.001	0.002	−0.011
σ	0.355	0.058	−0.042	−0.006	0.015	−0.019	0.012	0.012	−0.003	−0.012

Table 2: Estimated regression coefficients for parameters of the Pharmacokinetic model

independent variables $R^2 = \{0.44, 0.43, 0.31, 0.02\}$. These metrics indicate that we achieve a moderate goodness of fit for the parameters K_a, K_e, Cl and a very low goodness of fit for the parameter σ which would be expected as the σ parameter is the intrinsic stochastic noise which is not supposed to be modeled. Table 2 shows the coefficient values for each parameter.

3.2.4 Experiments with ABC MCMC

With our summary statistics function defined as above, we are left with two components that need to be defined to be ready to run ABC MCMC algorithm on this problem. First is the discrepancy metric which is defined as below:

$$\rho(S(\mathcal{D}^*), S(\mathcal{D})) = ||S(\mathcal{D}^*) - S(\mathcal{D})|| \tag{14}$$

where $||\theta||^2 = \sum \frac{\theta_i^2}{(\theta_0^2)}$ is weighted euclidean norm in $\mathbb{R}^4$.

Finally, we need to choose a suitable proposal for the MCMC algorithm. In this problem, we consider two types of proposal distributions:

- First is the random walk proposal which is the simplest and most popular choice for this algorithm. In particular, we define the following four proposal distributions respectively for our four parameters of interest:

$$\begin{aligned} q(K_a, \cdot) &\sim LogNormal(log(K_a), 0.4^2) \\ q(K_e, \cdot) &\sim LogNormal(log(K_e), 0.6^2) \\ q(Cl, \cdot) &\sim LogNormal(log(Cl), 0.8^2) \\ q(\sigma, \cdot) &\sim LogNormal(log(\sigma), 0.3^2) \end{aligned} \tag{15}$$

  since the parameters follow a lognormal distribution and we use the same variance values as their prior distributions. Implementation of this proposal distribution can be found in Listing 9. Assuming these parameters are independent, we define their joint density function as the product of their respective individual density functions.

- We also test a slightly more advanced proposal distribution which is data-driven [Murphy, 2022]. This family of proposal distributions depends on not only the previous state in the chain but also the data generated by the previous state. In particular, we consider our parameter estimation method 13 which gives us an expected value of the parameters $\mathbb{E}(\theta|\mathcal{D})$ based on the generated data $\mathcal{D}$. Since in the ABC MCMC algorithm we generate data $\mathcal{D}$ at each step from the model with parameters θ, we can construct a proposal distribution $q(\theta, \cdot)$ that uses the regression coefficients estimated before along with the generated data from the previous step to sample new parameters θ^* as following: $q(\theta, \cdot) \sim LogNormal(log(\mathbb{E}(\theta^*|\mathcal{D}, \theta)), \nu_\theta^2)$. Implementation of this proposal distribution is given in Listing 10. The joint distribution of these parameters is defined similar to the simple random walk proposal case.

With everything set up, we run $N = 10,000$ iterations of the ABC MCMC algorithm with initial state $\theta = (1.15, 0.07, 0.05, 0.33)$ for tolerances $\varepsilon = \{0.25, 0.7, 1\}$ on our problem (Listing 11). We report the acceptance rates for each proposal distribution and tolerance level combination in Table 3. We first note that as expected the acceptance rates go down as the tolerance level decreases. Secondly, we achieve much higher acceptance rates with the data-driven proposal distribution compared to the random walk proposal model and this indicates that the data-driven approaches are more efficient. In Figures 3, 4, we plot the kernel density estimation (KDE) of the marginal distributions of model parameters for both proposals respectively. What we observe is that as the tolerance level increases, the variance of the samples also increases as we would expect since more difference between the observed and

Proposal distribution	$\varepsilon = 1.0$	$\varepsilon = 0.7$	$\varepsilon = 0.25$
Random Walk	31.2%	22.4%	3.7%
Data-driven	82.5%	56.2%	1.3%

Table 3: ABC MCMC simulation acceptance rates (Pharmacokinetics problem) for different proposal distributions and tolerance levels (ε)

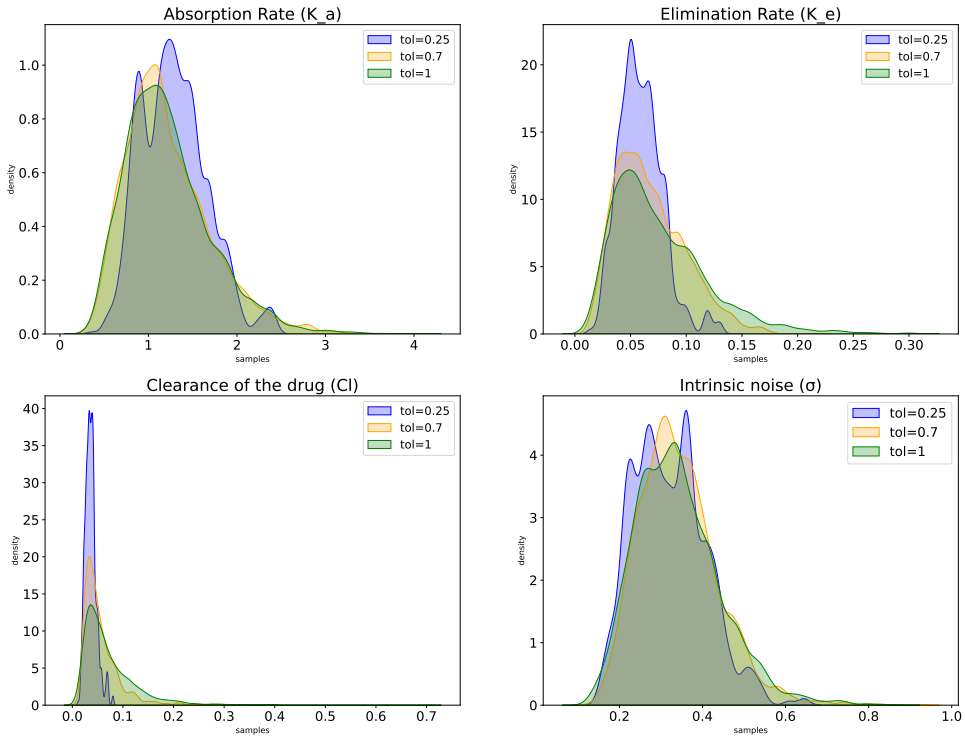

Figure 3: Maginal distributions of Pharmacokinetics problem parameters sampled with ABC MCMC algorithm using simple Random Walk proposal distribution and different tolerance levels.

generated data is tolerated. In addition, we note that in the Random Walk proposal case, parameter distributions mostly center on the same values with different variances while in the Data-driven case, there is slightly more difference between the distributions for the K_e and Cl parameters, however, in both cases values look reasonable compared to their prior distributions.

3.2.5 Variance Reduction

Finally, in this section we consider estimating $\mathbb{E}[X_9]$, the expected concentration of Theophylline after 12 hours using Monte Carlo estimator. First, we run a crude Monte Carlo estimation without any variance reduction where we use the posterior mean θ^{PM} of the four parameters for $\varepsilon = 0.25$ generated using the Random Walk proposal distribution and then sample $N = 1000$ values of X_9 from our Pharmacokinetic model 11 using Euler-Maruyama method. Then we average this values to estimate the expectation $\mathbb{E}[X_9] = \hat{\mu}_N = \frac{1}{N}\sum_{i=1}^{N} X_9^{(i)}$. It results in the mean value of $\hat{\mu}_N = 3.331$ with a variance $var = 0.622$. Implementation of this method is given in Listing 12.

Next, we apply a variance reduction technique to reduce the variance of our estimator. In particular, we consider the variance reduction using antithetic variables for its simplicity. Given our random variable X_9, we would like to find an antithetic variable X_9^a such that $X_9^a \sim X_9$ (i.e. they follow the

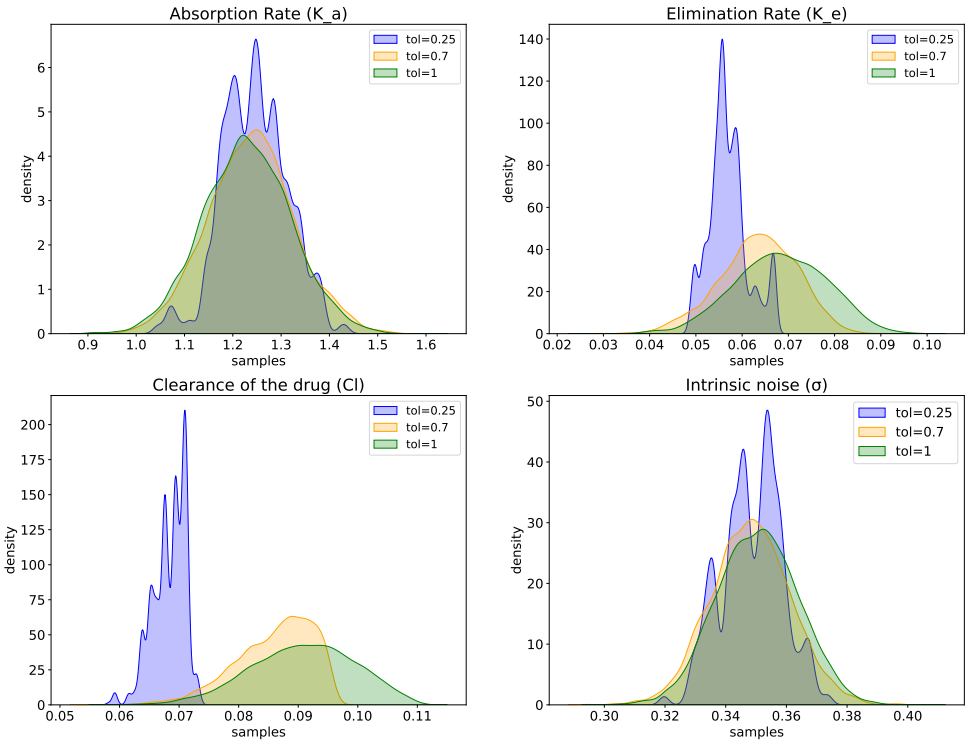

Figure 4: Marginal distributions of Pharmacokinetics problem parameters sampled with ABC MCMC algorithm using Data-driven proposal distribution and different tolerance levels.

Parameter	$\varepsilon = 1.0$	$\varepsilon = 0.7$	$\varepsilon = 0.25$
K_a	1.222	1.219	1.288
K_e	0.076	0.068	0.059
Cl	0.072	0.056	0.037
σ	0.341	0.342	0.321

Table 4: Estimate for the posterior mean θ^{PM} (Pharmacokinetics problem) for different tolerance levels (ε) and using Random Walk proposal model.

Parameter	$\varepsilon = 1.0$	$\varepsilon = 0.7$	$\varepsilon = 0.25$
K_a	1.229	1.238	1.245
K_e	0.068	0.063	0.057
Cl	0.091	0.086	0.068
σ	0.351	0.348	0.349

Table 5: Estimate for the posterior mean θ^{PM} (Pharmacokinetics problem) for different tolerance levels (ε) and using Data-driven proposal model.

same distribution) and $cov(X_9, X_9^a) < 0$ (i.e. they are negatively correlated). In order to construct such a random variable, we look at the data generation process from model 11. We note that given all the parameter values the only stochasticity in this model comes from the Brownian motion W_t. Hence, we can express X_9 as a function of the series of W_t random variables: $X_9 = \phi(W_1, W_2, ..., W_m)$ where $W_t \sim \mathcal{N}(0, dt)$. Since all W_t random variables are independent and ϕ is monotonic in its arguments, we can construct a new random variable $W_t^a = 2\mathbb{E}[W_t] - W_t = -W_t \sim W_t$ and correspondingly, $X_9^a = \phi(W_1^a, W_2^a, ..., W_m^a) = \phi(-W_1, -W_2, ..., -W_m)$. Then the theorem about variance reduction using antithetic variable tells us that $\mathbb{E}[X_9^a] = \mathbb{E}[X_9]$ and $cov(X_9^a, X_9) < 0$ and we can estimate the mean as following:

$$\hat{\mu}_{AV} = \frac{1}{N/2} \sum_{i=1}^{N} \frac{X_9^{(i)} + X_9^{a(i)}}{2} \tag{16}$$

which should give us a reduction in variance. We run the Monte Carlo simulation with the same setup as before except that we use the antithetic variable $\hat{\mu}_{AV}$ (Listing 12) and achieve the following mean and variance: $\hat{\mu}_{AV} = 3.360$ and $var = 0.349$. Results show that we obtain a similar expectation value with an almost two times reduction in variance.

4 Conclusion

References

[Dirk P. Kroese, 2011] Dirk P. Kroese, Thomas Taimre, Z. I. B. (2011). *Handbook of Monte Carlo Methods*. John Wiley Sons, Inc.

[Marjoram et al., 2003] Marjoram, P., Molitor, J., Plagnol, V., and Tavaré, S. (2003). Markov chain monte carlo without likelihoods. *Proceedings of the National Academy of Sciences*, 100(26):15324–15328.

[Murphy, 2022] Murphy, K. P. (2022). *Probabilistic Machine Learning: An introduction*. MIT Press.

[Picchini, 2014] Picchini, U. (2014). Inference for sde models via approximate bayesian computation. *Journal of Computational and Graphical Statistics*, 23(4):1080–1100.

A Code

```python
from abc import ABC
import scipy.stats as stats
import numpy as np

class Model(ABC):
    """Abstract class for statistical models."""

    def sample(self, size=None, *args, **kwargs):
        """Sample from the model.

        Args:
            size (int, optional): Number of samples to generate. Defaults to None.

        Returns:
            float: The sample.
        """
        raise NotImplementedError

    def pdf(self, x):
        """Compute the probability density function of the model.

        Args:
            x (float): Value to compute the pdf at.

        Returns:
            float: The pdf value.
        """
        raise NotImplementedError

class Normal(Model):
    """Normal distribution."""

    def __init__(self, mu=0, sigma=1):
        """Initialize the model.

        Args:
            mu (float, optional): Mean of the distribution. Defaults to 0.
            sigma (float, optional): Standard deviation of the distribution. Defaults to 1.
        """
        self.mu = mu
        self.sigma = sigma
        self.dist = stats.norm(self.mu, self.sigma)

    def sample(self, size=None):
        return self.dist.rvs(size=size)

    def pdf(self, x):
        return self.dist.pdf(x)

class LogNormal(Model):
    """Log-normal distribution."""

    def __init__(self, mu=0, sigma=1):
        """Initialize the model.

        Args:
            mu (float, optional): Mean of the underlying normal distribution. Defaults to 0.
            sigma (float, optional): Standard deviation of the underlying normal distribution. Defaults to 1.
        """
        self.mu = mu
        self.sigma = sigma
        self.dist = stats.lognorm(s=self.sigma, scale=np.exp(self.mu))

    def sample(self, size=None):
        return self.dist.rvs(size=size)

```

```
    def pdf(self, x):
        return self.dist.pdf(x)

def weighted_euclidean_norm(x: np.array, weights: np.array = 1) -> float:
    """Compute the weighted Euclidean norm.

    Args:
        x (np.array): Array of values.
        weights (np.array, optional): Array of weights. Defaults to 1.

    Returns:
        float: The weighted Euclidean norm.
    """
    return np.sqrt(np.sum(np.square(x)/np.square(weights)))
```

Listing 1: Some utility functions and classes used throughout the project

```
import numpy as np

def run_abc_rejection(N: int, observed_data: Union[List, np.array], prior_model: Model
    , generate_data: Callable, compute_discrepancy: Callable, tolerance: float = 0.1)
    -> Tuple[List, float]:
    """Run the ABC rejection algorithm.

    Args:
        N (int): Number of samples to generate.
        observed_data (Union[List, np.array]): Observed data.
        prior_model (Model): Prior model of the samples.
        generate_data (Callable): Function to generate data from given parameters.
        compute_discrepancy (Callable): Function to compute the discrepancy between
    two sets of data.
        tolerance (float, optional): Tolerance for the discrepancy. Defaults to 0.1.

    Returns:
        Tuple[List, float]: The generated samples and acceptance rate.
    """
    sample = []
    num_tries = 0

    while len(sample) < N:
        num_tries += 1
        theta = prior_model.sample()
        generated_data = generate_data(theta, len(observed_data))
        if compute_discrepancy(observed_data, generated_data) < tolerance:
            sample.append(theta)

    acceptance_rate = N / num_tries

    return sample, acceptance_rate
```

Listing 2: Implementation of ABC Rejection algorithm

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

def run_abc_mcmc(N: int, observed_data: Union[List, np.array], make_proposal_model:
    Callable, prior_model: Model, generate_data: Callable, compute_discrepancy:
    Callable, tolerance: float = 0.1, theta_0: Any = 0, burn_in: float = 0.1, data_0:
    Any = 0) -> Tuple[List, float]:
    """Run the ABC MCMC algorithm.

    Args:
        N (int): Number of samples to generate.
        observed_data (Union[List, np.array]): Observed data.
        make_proposal_model (Callable): Function to make the proposal model given the
    current parameters.
        prior_model (Model): Prior model of the samples.
        generate_data (Callable): Function to generate data from given parameters.
        compute_discrepancy (Callable): Function to compute the discrepancy between
    two sets of data.
        tolerance (float, optional): Tolerance for the discrepancy. Defaults to 0.1.
```

```
        theta_0 (Any, optional): Initial parameter value. Defaults to 0.
        burn_in (float, optional): Burn-in ratio. Defaults to 0.1.
        data_0 (Any, optional): Initial data value. Defaults to 0.

    Returns:
        Tuple[List, float]: The generated samples and acceptance rate.
    """
    sample = [theta_0]
    sample_data = [data_0]
    num_accepted = 0
    burn_in_size = int(N * burn_in)

    for i in range(burn_in_size+N):
        current_theta = sample[-1]
        current_data = sample_data[-1]
        # Define q(theta, .)
        current_proposal_model = make_proposal_model(theta=current_theta, data=
    current_data)

        # Sample theta* from q(theta, .)
        new_theta = current_proposal_model.sample()

        # Generate data from theta*
        generated_data = generate_data(new_theta, len(observed_data))

        # Define q(theta*, .)
        new_proposal_model = make_proposal_model(theta=new_theta, data=generated_data)

        if compute_discrepancy(observed_data, generated_data) < tolerance:
            alpha = min(1, (prior_model.pdf(new_theta) * new_proposal_model.pdf(
    current_theta)) / (prior_model.pdf(current_theta) * current_proposal_model.pdf(
    new_theta)))
            prob = stats.uniform.rvs()
            if prob < alpha:
                sample.append(new_theta)
                sample_data.append(generated_data)
                num_accepted += 1
            else:
                sample.append(current_theta)
        else:
            sample.append(current_theta)

    acceptance_rate = num_accepted / (N+burn_in_size)

    return sample[burn_in_size+1:], acceptance_rate
```

Listing 3: Implementation of ABC MCMC algorithm

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

example_a = 1
example_p = 1/2
example_M = 100
example_sigma_1 = np.sqrt(0.1)
example_sigma = np.sqrt(3)
example_mean = 0
example_N = 500
example_tolerances = [0.75, 0.25, 0.1, 0.025]

class ExampleLikelihoodModel(Model):
    """Likelihood model for the synthetic problem."""

    def __init__(self, theta, a, sigma, p):
        self.theta = theta
        self.a = a
        self.sigma = sigma
        self.p = p
        self.dist1 = stats.norm(loc=self.theta, scale=self.sigma)
        self.dist2 = stats.norm(loc=self.theta+self.a, scale=self.sigma)

```

```
    def sample(self, size=None):
        prob = stats.uniform.rvs()

        if prob < self.p:
            return self.dist1.rvs(size=size)

        return self.dist2.rvs(size=size)

def generate_example_data(theta, size):
    return ExampleLikelihoodModel(theta, example_a, example_sigma_1, example_p).sample(size=size)

def compute_example_discrepancy(observed_data, generated_data):
    # The mean of the observed data is given to us (example_mean), so we ignore the data itself
    return np.abs(np.mean(generated_data) - example_mean)

prior_model = Normal(0, example_sigma)

example_theta = prior_model.sample()
observed_data = generate_example_data(example_theta, example_M)

for tolerance in example_tolerances:
    sample, acceptance_rate = run_abc_rejection(example_N, observed_data, prior_model, generate_example_data, compute_example_discrepancy, tolerance)
```

Listing 4: Experiments with ABC Rejection algorithm on the synthetic problem

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

class ExampleProposalModel(Model):
    """Proposal model for the synthetic problem."""

    def __init__(self, theta, sigma):
        self.theta = theta
        self.sigma = sigma
        self.dist = stats.norm(loc=self.theta, scale=self.sigma)

    def sample(self, size=None):
        return self.dist.rvs(size=size)

    def pdf(self, x):
        return self.dist.pdf(x)

def make_example_proposal_model(theta, sigma=np.sqrt(0.1), **kwargs):
    return ExampleProposalModel(theta, sigma)

prior_model = Normal(0, example_sigma)

example_theta = prior_model.sample()
observed_data = generate_example_data(example_theta, example_M)

# Define proposal variances
proposal_vars = [0.5, 1, 2, 4, 8]

for proposal_var in proposal_vars:
    for tolerance in example_tolerances:
        sample, acceptance_rate = run_abc_mcmc(example_N, observed_data, partial(
    make_example_proposal_model, sigma=np.sqrt(proposal_var)),
                                               prior_model, generate_example_data,
    compute_example_discrepancy, tolerance)
```

Listing 5: Experiments with ABC MCMC algorithm on the synthetic problem

```
import numpy as np

def run_euler_maruyama(sampling_times: List[int], model: Model, x_0: Any = 0, dt: float = 0.01) -> List:
    """Run the Euler-Maruyama algorithm.

```

```
    Args:
        sampling_times (List[int]): List of sampling times.
        model (Model): Model to sample from.
        x_0 (Any, optional): Initial value. Defaults to 0.
        dt (float, optional): Time step. Defaults to 0.01.

    Returns:
        List: The generated samples.
    """
    x = [x_0]
    sample = []
    t = 0

    while len(sample) < len(sampling_times):
        x_t = x[-1]
        x_t_plus_1 = x_t + model.sample(x_t=x_t, t=t)
        x.append(x_t_plus_1)

        # Check if the current timestep is a sampling time that we want to record at
        if any([np.isclose(t, sampling_time) for sampling_time in sampling_times]):
            sample.append(x_t_plus_1)

        t = t + dt

    return sample
```

Listing 6: Implementation of Euler-Maruyama method

```
import numpy as np

sampling_dt = 0.005
sampling_times = [0.25, 0.5, 1, 2, 3.5, 5, 7, 9, 12]
drug_dose = 4

class PHKModel(Model):
    """Pharmacokinetics model given as a stochastic differential equation (SDE)."""

    def __init__(self, D, K_a, K_e, Cl, sigma=1, dt=0.01, brownian=None):
        self.D = D
        self.K_a = K_a
        self.K_e = K_e
        self.Cl = Cl
        self.sigma = sigma
        self.dt = dt
        self.brownian = Normal(0, np.sqrt(dt)) if not brownian else brownian

    def sample(self, size=None, x_t=0, t=0.01):
        return ((self.D * self.K_a * self.K_e * np.exp(-self.K_a * t))/self.Cl - self.
    K_e * x_t) * self.dt + self.sigma * self.brownian.sample(size=size)

phk_model = PHKModel(D=drug_dose, K_a=1.5, K_e=0.08, Cl=0.04, sigma=0.2, dt=
    sampling_dt)
observed_data = run_euler_maruyama(sampling_times, phk_model, dt=sampling_dt)
```

Listing 7: Observed data generation for Pharmacokinetics model

```
import numpy as np
import pandas as pd
import statsmodels.api as sm

train_size = 1000

class PHKPriorModel(Model):
    """Prior model for the pharmacokinetics problem."""

    def __init__(self):
        self.prior_K_a = LogNormal(0.14, 0.4)
        self.prior_K_e = LogNormal(-2.7, 0.6)
        self.prior_Cl = LogNormal(-3, 0.8)
        self.prior_sigma = LogNormal(-1.1, 0.3)

```

```
    def sample(self, size=None):
        return [self.prior_K_a.sample(), self.prior_K_e.sample(), self.prior_Cl.sample(), self.prior_sigma.sample()]

    def pdf(self, x):
        return self.prior_K_a.pdf(x[0]) * self.prior_K_e.pdf(x[1]) * self.prior_Cl.pdf(x[2]) * self.prior_sigma.pdf(x[3])

def generate_train_data(train_size=100):
    train_data = []
    prior_model = PHKPriorModel()

    for i in tqdm(range(train_size), total=train_size, desc="Generating train data"):
        prior_sample = prior_model.sample()
        priors = {
            "K_a": prior_sample[0],
            "K_e": prior_sample[1],
            "Cl": prior_sample[2],
            "sigma": prior_sample[3]
        }
        phk_model = PHKModel(D=drug_dose, dt=sampling_dt, **priors)
        sample = run_euler_maruyama(sampling_times, phk_model, dt=sampling_dt)
        train_data.append(sample + [priors["K_a"], priors["K_e"], priors["Cl"], priors["sigma"]])

    train_df = pd.DataFrame(train_data, columns=[f"x_[t={t}]" for t in sampling_times] + ["K_a", "K_e", "Cl", "sigma"])

    return train_df

def train_model(train_df, target):
    train_x = train_df[[col for col in train_df.columns if col.startswith("x_")]]
    train_y = train_df[target]
    train_x = sm.add_constant(train_x)
    model = sm.OLS(train_y, train_x).fit()
    return model

# Generate train data
train_df = generate_train_data(train_size=train_size)

# Train models
k_a_model = train_model(train_df, target="K_a")
k_e_model = train_model(train_df, target="K_e")
cl_model = train_model(train_df, target="Cl")
sigma_model = train_model(train_df, target="sigma")
```

Listing 8: Parameter estimation for Pharmacokinetics model using linear regression

```
import numpy as np

class PHKRWLogNormProposalModel(Model):
    """Random walk log-normal proposal model for the pharmacokinetics problem."""

    def __init__(self, theta):
        self.k_a_model = LogNormal(np.log(theta[0]), 0.4)
        self.k_e_model = LogNormal(np.log(theta[1]), 0.6)
        self.cl_model = LogNormal(np.log(theta[2]), 0.8)
        self.sigma_model = LogNormal(np.log(theta[3]), 0.3)

    def sample(self, size=None):
        return [self.k_a_model.sample(), self.k_e_model.sample(), self.cl_model.sample(), self.sigma_model.sample()]

    def pdf(self, x):
        return self.k_a_model.pdf(x[0]) * self.k_e_model.pdf(x[1]) * self.cl_model.pdf(x[2]) * self.sigma_model.pdf(x[3])
```

Listing 9: Simple random walk proposal distribution for Pharmacokinetic model

```
import numpy as np

```

```python
def compute_expected_theta(coefficients, data):
    """Compute the expected theta from the coefficients and the data (E[theta | D])

    Args:
        coefficients (np.array): Coefficients of the linear regression model (beta_i).
        data (np.array): Data (D).

    Returns:
        np.array: The expected theta.
    """
    return np.dot(coefficients, np.hstack([[1], data]).reshape(-1, 1)).squeeze()

class PHKRWDataDrivenProposalModel(Model):
    """Random walk data-driven proposal model for the pharmacokinetics problem."""

    def __init__(self, coefficients, data):
        self.coefficients = coefficients  # estimated coefficients (beta_i) from the linear regression model
        self.data = data
        self.theta = compute_expected_theta(coefficients, data)
        self.k_a_model = LogNormal(np.log(self.theta[0]), 0.4)
        self.k_e_model = LogNormal(np.log(self.theta[1]), 0.6)
        self.cl_model = LogNormal(np.log(self.theta[2]), 0.8)
        self.sigma_model = LogNormal(np.log(self.theta[3]), 0.3)

    def sample(self, size=None):
        return self.theta

    def pdf(self, x):
        return self.k_a_model.pdf(x[0]) * self.k_e_model.pdf(x[1]) * self.cl_model.pdf(x[2]) * self.sigma_model.pdf(x[3])
```

Listing 10: Data-driven proposal distribution for Pharmacokinetic model

```python
import numpy as np
import pickle

sampling_dt = 0.005
sampling_times = [0.25, 0.5, 1, 2, 3.5, 5, 7, 9, 12]
phk_tolerances = [0.25, 0.7, 1]
phk_N = 10000
theta_0 = [1.15, 0.07, 0.05, 0.33]
burn_in = 0.1

def generate_phk_data(theta, size):
    phk_model = PHKModel(D=drug_dose, K_a=theta[0], K_e=theta[1], Cl=theta[2], sigma=theta[3], dt=sampling_dt)
    return run_euler_maruyama(sampling_times, phk_model, dt=sampling_dt)

def compute_phk_discrepancy(coefficients, theta_0, observed_data, generated_data):
    s_observed_data = compute_expected_theta(coefficients, observed_data)
    s_generated_data = compute_expected_theta(coefficients, generated_data)
    return weighted_euclidean_norm(s_generated_data - s_observed_data, weights=np.array(theta_0))

def make_phk_rw_lognorm_proposal_model(theta, **kwargs):
    return PHKRWLogNormProposalModel(theta)

def make_phk_rw_data_driven_proposal_model(coefficients, theta, data, **kwargs):
    return PHKRWDataDrivenProposalModel(coefficients, data)

# Load observed data
observed_data = np.load(...)

# Load model coefficients
k_a_model = pickle.load(...)
k_e_model = pickle.load(...)
cl_model = pickle.load(...)
sigma_model = pickle.load(...)
coefficients = np.vstack([k_a_model.params, k_e_model.params, cl_model.params, sigma_model.params])
```

```

# Initialize prior model
phk_prior_model = PHKPriorModel()

# Initialize proposal models
proposal_map = {
    "rw_lognorm": make_phk_rw_lognorm_proposal_model,
    "rw_dd": partial(make_phk_rw_data_driven_proposal_model, coefficients=coefficients)
}

proposal = "rw_lognorm"

data_0 = None

if proposal == "rw_dd":
    # If we are using the data-driven proposal, we need to generate data for the initial theta
    phk_model = PHKModel(D=drug_dose, K_a=theta_0[0], K_e=theta_0[1], Cl=theta_0[2], sigma=theta_0[3], dt=sampling_dt)
    data_0 = run_euler_maruyama(sampling_times, phk_model, dt=sampling_dt)

for tolerance in phk_tolerances:
    sample, acceptance_rate = run_abc_mcmc(phk_N,
                                           observed_data,
                                           proposal_map[proposal],
                                           phk_prior_model,
                                           generate_phk_data,
                                           partial(compute_phk_discrepancy, coefficients, theta_0),
                                           tolerance,
                                           theta_0=theta_0,
                                           burn_in=burn_in,
                                           data_0=data_0)
```

Listing 11: Experiments with ABC MCMC algorithm on the Pharmacokinetics problem

```

import numpy as np

class NegativeNormal(Normal):
    """Negative normal distribution."""
    def sample(self, size=None):
        return -super().sample(size=size)

# Load posterior samples and compute posterior theta mean
posterior_theta_samples = np.load(...)
posterior_theta = np.mean(posterior_theta_samples, axis=0)

# Initialize model with posterior theta
phk_model = PHKModel(D=drug_dose, K_a=posterior_theta[0], K_e=posterior_theta[1], Cl=posterior_theta[2], sigma=posterior_theta[3], dt=sampling_dt)

# Run Crude Monte Carlo Estimation
cmc_sample = []

size = 1000

for i in range(size):
    data = run_euler_maruyama([12], phk_model, dt=sampling_dt)
    cmc_sample.append(data[0])

print("Crude Monte Carlo Estimation of E[X_9]:")
print(f"mean: {np.mean(cmc_sample)}, variance: {np.var(cmc_sample)}")

# Run Monte Carlo Estimation using anthithetic variates
av_sample = []

# This is identical to the phk_model, except that the brownian motion is negative normal
negative_phk_model = PHKModel(D=drug_dose, K_a=posterior_theta[0], K_e=posterior_theta[1], Cl=posterior_theta[2], sigma=posterior_theta[3],
```

```
                                dt=sampling_dt, brownian=NegativeNormal(0, np.sqrt(
    sampling_dt)))

for i in range(int(size/2)):
    positive_data = run_euler_maruyama([12], phk_model, dt=sampling_dt)
    negative_data = run_euler_maruyama([12], negative_phk_model, dt=sampling_dt)
    av_sample.append((positive_data[0] + negative_data[0])/2)

print("AV Monte Carlo Estimation of E[X_9]:")
print(f"mean: {np.mean(av_sample)}, variance: {np.var(av_sample)}")
```

Listing 12: Estimation of expected concentration of Theophylline after 12 hour using Crude Monte Carlo Estimation and Antithetic Variable Monte Carlo Estimation

B Additional Results

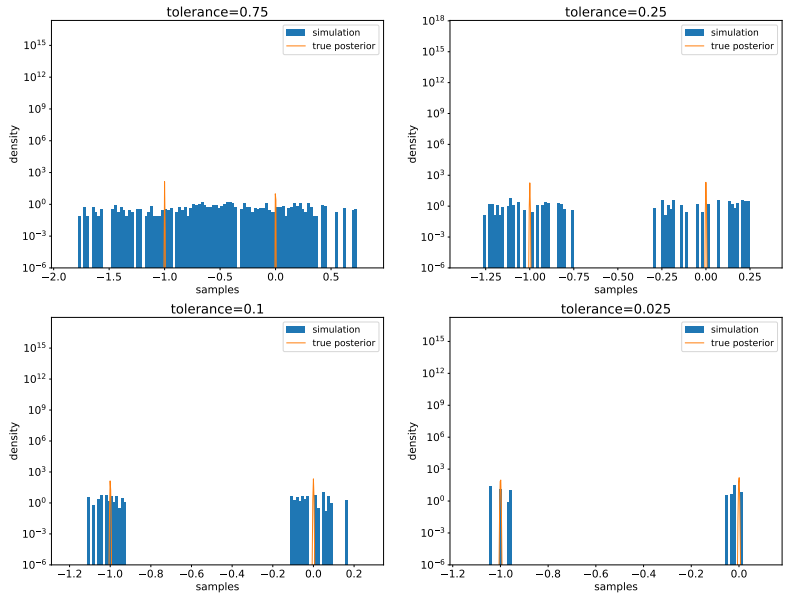

Figure 5: Histogram of ABC MCMC simulation results (synthetic problem) ($\nu^2 = 0.5$) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.

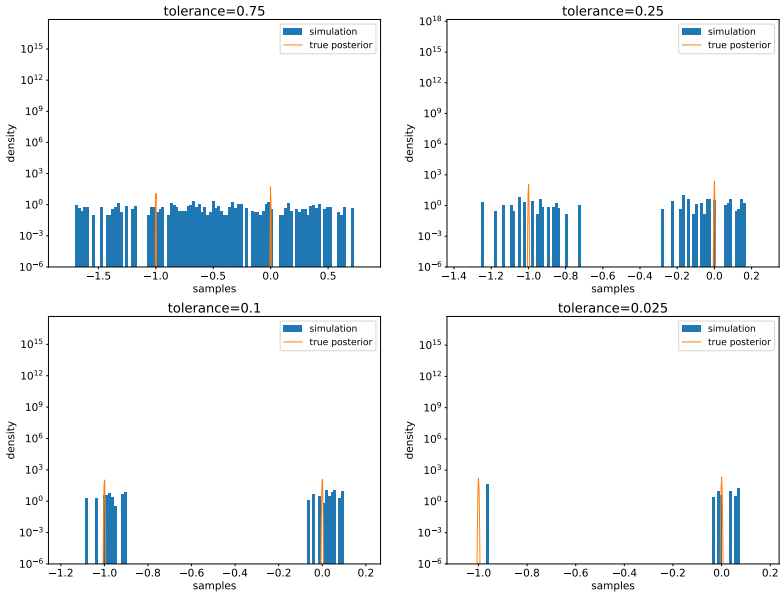

Figure 6: Histogram of ABC MCMC simulation results (synthetic problem) ($\nu^2 = 2$) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.

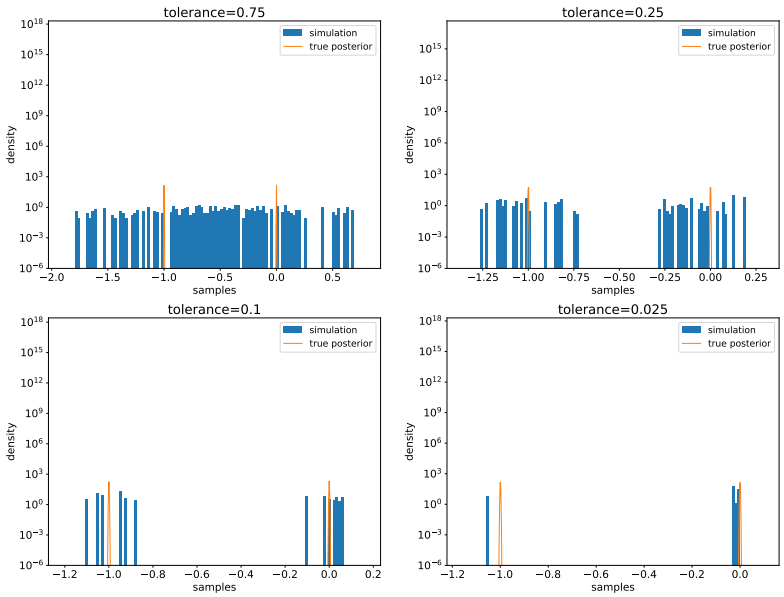

Figure 7: Histogram of ABC MCMC simulation results (synthetic problem) ($\nu^2 = 4$) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.

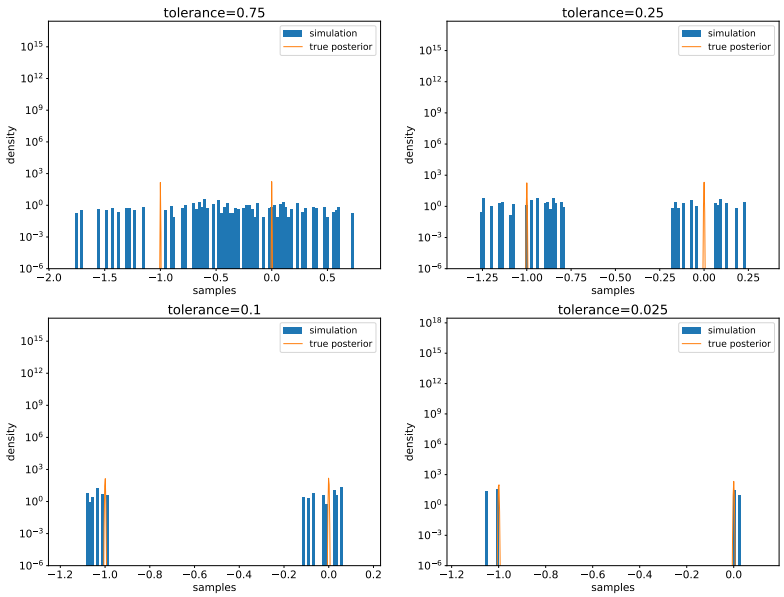

Figure 8: Histogram of ABC MCMC simulation results (synthetic problem) ($\nu^2 = 8$) compared against the true distribution for different tolerance levels. For better visual readability, the y-axis is on a log scale and starts at 10^{-6}.