Compiled Inference with Probabilistic Programming for Large-Scale Scientific Simulations

Mario Lezcano Casado

Kellogg College

University of Oxford

A thesis submitted for the degree of

MSc in Computer Science

Trinity 2017

Machine Learning is like teenage sex:
Everyone talks about it,
nobody really knows how to do it,
everyone thinks everyone else is doing it,
so everyone claims they are doing it.

Dan Ariely

Abstract

Bayesian inference has always been a prominent topic in the statistics and machine learning field, but the interest on it has been steadily increasing in the last twenty years with the development of Probabilistic Programming. In particular, in the last three years with the blooming of Turing-complete probabilistic programming languages. However, the current approach to probabilistic programming has limited influence, since it cannot be applied to pre-existing models.

We introduce CPProb, a high-performance, extensible probabilistic programming library written in C++14. CPProb aims be the first probabilistic programming library to perform inference in large-scale C++ projects. This library aspires to be the bridge between theory and practice in the probabilistic programming field.

We also include theoretical review pointing out novel theoretical issues that were omitted in the papers presenting these ideas, and that should be resolved in order for the field of particle filters on Turing-complete probabilistic programming languages to be theoretically grounded. In this respect, we link Compiled Sequential Importance Sampling [25] to recent results in the field of importance sampling, providing the first theoretical justification for the choice of the objective function used in this algorithm.

Acknowledgements

First of all, I would like to express my thanks to both of my advisors, Dr. Frank Wood and Dr. Sam Staton, for their constant guidance throughout this project and their constant valuable feedback, in particular on general Bayesian statistics and foundations of probabilistic programming. I would also like to thank Dr. Atılım Güneş Baydin and Tuan Anh Le for the development of PyProb, without which CPProb would never be in the development state that it currently is. Similarly, I would like to thank David Martínez Rubio for his invaluable feedback throughout the whole project, specially when dealing with optimisation methods. Furthermore, I would like to thank David Tolpin, William Harvey, Lukas Heinrich, Yuan Zhou and Rob Cornish for their contributions that helped improve the quality of both the presentation and the content of this dissertation. I would also like to thank the whole team of Systematics for making this amazing and exciting project possible, giving CPProb a chance to show its true potential in a real world large-scale application. Finally, I would also like to thank the whole Wood group for making the workday routine in the lab much more enjoyable.

My most sincere thanks to my parents and my brother for their constant support in the good and the bad times. Thank you for giving me this amazing opportunity.

When it comes to less-academic issues, I would like to thank Juan and Danny. Without their "Cheeky pint at St.Johns after work, mate?", and their "Quick night out in London next weekend?" I would have probably finished this dissertation way too early. Cheers for keeping the deadline pressure on point. Thanks also to the Wellington Square group and, in general, all those amazing friends in Kellogg that made this year so special.

Contents

Chapter 1

Introduction

1.1 Motivation

In general, Bayesian inference in high dimensional Bayesian models is hard [8]. Even though, in words of Papadimitriou, "there are not that many problems that are not hard and are interesting". Given these two premises, we should not be discouraged from attacking an interesting problem just because its worst case complexity can potentially be very high.

Probabilistic programming languages provide the primitives to define multi-dimensional distributions in a Turing-complete language [34]. The distributions defined in a probabilistic programming language are Bayesian, in the sense that they factorise into a prior and a likelihood. With this approach, it is possible to perform inference in a completely transparent way, easing the whole model-iteration process by several orders of magnitude.

The current approach to probabilistic programming is to define a fully fledged new programming language. Then, this probabilistic programming language offers the user the tools to define a model, but it also restricts what the user can and cannot do. Furthermore, in many cases, the restrictions that the probabilistic programming language impose are, by no means, subtle. In some cases they go as far as not allowing variable-dependant loops or recursion, making the language non-Turing complete [38, 28]. When this is not the case, we are usually presented with intermediate languages that compile the code into other language [45, 26]. This is fine when we deal with small enough distributions, programs that consist

of no more than a few hundreds of lines —sometimes not even that long—, but this definitely does not scale to large models used in many fields such as Particle Physics, Fluid Dynamics, Engineering or Economics.

When we deal with large models we have two problems. The first one is, of course, efficiency. A team is not going to develop a physics simulator in Clojure, Haskell, Python or, in general, anything different to C, C++ or Fortran. Given this premise, most of the models are written in one of these three languages. Now, if a team was to perform inference in their model, which is in most of the cases the product of several years of iterations and improvement, given the current tools to perform inference, they would have to translate their model into one of these probabilistic programming languages. This is just not realistic.

Because of this reason, probabilistic programming currently has a limited applications in large-scale models.

1.2 Contribution

We will follow a theoretical and a practical approach; hence, the contributions of this dissertation are twofold.

First, we present CPProb, a C++ library that allows the user to perform inference on probabilistic programs defined as arbitrary C++ programs. The four main design objectives are:

- Generality: It should make no assumptions on the C++ features that the model uses or how it is written. It could be legacy code, use exceptions, make use of the keyword static or extern or almost any kind of standard complaint code.

- Zero-cost abstraction: The library should provide high-level abstractions without sacrificing runtime performance.

- Extensibility: The user should be able to use their own types. Furthermore, the user should be able to extend the library with new distributions without having to modify the core library.

- Ease of use: The user should be able to perform inference on her models changing the code as least as possible. In particular, using the provided functions `sample`, `observe` and `predict` should be enough.

Our approach builds on top of PyProb, an neural network developed by Tuan Anh Le and Atılım Güneş Baydin that allows for the approximation of good proposal distributions in the framework of Compiled Sequential Importance Sampling [25].

The second contribution is a formal review of algorithms for forward simulation sampling, quality of the estimators associated to these algorithms and how they can be used to perform inference in Turing-complete probabilistic programming language. We found that a formal review of the techniques used in the field was lacking, given that this field is fairly new, with the first Turing-complete programming language dating back to 2012 [15]. Throughout the exposition, we will make explicit several hypotheses that are missing in most of the papers in the field. Without these hypotheses, derivations present in some papers are lacking the justification or are outright incorrect. We will either give results or conditions under which these derivations are correct. Finally, we give the first formal justification for the objective used in the framework of Compiled Sequential Importance Sampling.

1.3 Dissertation Structure

In the second chapter we present the definitions and theorems from measure theory and statistics that we will use during the rest of the dissertation. Most of them are covered in any first course in probability theory, measure theory and statistics. Given these results, we aim make the rest of the dissertation completely self-contained.

In the third chapter, we develop in a completely self-contained manner, the theoretical basis of several forward sampling methods. We develop these in the *state-space model*, as is it the most common approach. It will not be until the introduction of *Compiled Sequential Importance Sampling* that we will introduce the framework for Turing-complete probabilistic programs. We will briefly discuss how the state-space model can be adapted to approximately model this space and what are the problems that arise from this more general approach. Finally,

we improve on a bound for the non-asymptotic bias of the sequential importance sampling estimator, and we give the first formal justification for the objective function used in Compiled Sequential Importance Sampling.

In the fourth chapter, we introduce CPProb. CPProb is a C++14 library developed to be able to perform inference in C++ in the most general settings, imposing almost no restrictions on the model. In this chapter we discuss general architectural decisions, certain non-trivial implementation details, and the novel proposals introduced in this library to solve domain-specific problems; as the addressing system or the rejection sampling optimisation. In this section some knowledge of the C++14 language will be assumed, although we will give footnote references whenever we mention more advanced C++ programming techniques as type-erasure or different template meta-programming tricks.

In the fifth chapter, we describe the experiments run with the High Energy Physics software for simulation of particle interaction SHERPA. In this experiment we aim to infer the decay channel and initial momentum of a tau lepton. This is the first time that probabilistic programming has been applied to large-scale models.

In the sixth chapter, we present experiments that evaluate the inference performance of CPProb in various standard models with a computable ground-truth. We evaluate the rate at which the estimators given by CPProb converge to the analytical solutions of the problem. We find that the compiled inference method is able to outperform SMC in Markovian models. This is specially remarkable given that SMC is an improvement over regular sequential importance sampling that benefits from the Markovian properties of the models.

In the seventh chapter, we recapitulate the main ideas that were presented during the exposition and we will expose the main extensions of CPProb that will guide its future development.

Chapter 2

Bayesian Inference

This section will give a short introduction of the main concepts that will be used during the dissertation.

2.1 Probability Essentials

In the dissertation we will almost always work with finite measures which are absolutely continuous with respect to some reference measure for simplicity, but in certain points we will mention some measure-theoretical issues. For this reason, we start with a very short introduction of the main concepts in measure theory.

Definition 2.1.1 (Measurable space). Let X be a set. A set of subsets $\Sigma \subseteq \mathcal{P}(X)$ is called a *σ-algebra over X* if $\emptyset \in \Sigma$ and Σ is closed under compliment and countable unions and countable intersections.

The pair (X, Σ) is a *measurable space*.

Definition 2.1.2 (Measure). Let X be a set and Σ be a σ-algebra over X. A function $\mu\colon \Sigma \to [0, \infty]$ is a measure if $\mu(\emptyset) = 0$ and it is countably additive this is, for every countable family of disjoint subsets $\{F_n\} \subseteq \Sigma$,

$$\mu\Big(\bigcup_{n\in\mathbb{N}} F_n\Big) = \sum_{n\in\mathbb{N}} \mu(F_n).$$

A measurable space equipped with a measure is called a *measure space*.

A measurable is called *finite* if $\mu(X) < \infty$.

A measure is called a *probability measure* if $\mu(X) = 1$. In some cases we will denote our measure by Pr if the measure is a probability measure.

A measurable space equipped with a probability measure is called a *probability space.*

Definition 2.1.3 (Measurable function). Let (X, Σ_X), (Y, Σ_Y) be two measurable spaces. A function $f\colon X \to Y$ is measurable if for every $B \in \Sigma_Y$, $f^{-1}(B) \in \Sigma_X$.

If X is a probability space $(X, \Sigma_X, \Pr)$, we say that the function is a random variable.

Proposition 2.1.4. *Let (X, Σ_X, μ) be a measure space, (Y, Σ_Y) a measurable space and let f be a measurable function between them, then the push-forward measure defined as*

$$\mu_*(B) = \mu(f^{-1}(B)) \quad \forall B \in \Sigma_Y$$

is a measure over Y.

Informally, this proposition describes how measurable functions help us defining measures in other sets.

Definition 2.1.5 (Equivalence of Measurable Functions). Let $\pi, \pi'\colon X \to Y$ be two measurable functions. π *is distributed as* π' if they attain the same values outside of subset of zero measure. We write $\pi \sim \pi'$.

Definition 2.1.6 (Conditional Probability). Let X be a probability space with probability measure $\Pr$ and let $A, B \subseteq X$ be two events where $\Pr(B) \neq 0$, we define

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}.$$

Remark. The previous definition will most of the time be seen as the special case where X is a product space. In this case we can write $X = X_1 \times X_2$ where $A \subseteq X_1$ and $B \subseteq X_2$, and $\Pr(A \cap B) = \Pr(A, B)$, where $\Pr$ denotes the probability measure over the product space.

Theorem 2.1.7 (Bayes' theorem). *Given two events $A, B \in X$ in a probability space X with probability measure* $\Pr$*, where* $\Pr(B) \neq 0$*, we have that*

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)}.$$

The proof is a direct consequence of the definition of conditional probability.

All the work in this dissertation will be related to this simple formula in one way or another.

To keep this introduction short, we will skip the integration theory formalism. We refer the reader to [42] for a comprehensive introduction on this topic.

2.2 Statistics Essentials

This section will give a short exposition of the main definitions and results that will be used later in the dissertation. The proofs for all the results stated in this section can be found in almost every book that covers a first course in statistics.

Furthermore, when we write probability or measure space, we will refer to either a discrete measure space or $\mathbb{R}^n$. In these sets we have as dominating measures the Lebesgue measure in the continuous case, and the counting measure in the discrete case.

For simplicity, during most of the exposition we will also use integrals when dealing with discrete random variables, since these can be seen as embedded into $\mathbb{R}$ using a Dirac measure.

For an $x \in \mathbb{R}$ we will use the notation δ_x to refer to the *delta function* with support x.

In general, we will assume that the random variables considered are integrable with respect to the considered measure, unless stated otherwise.

Definition 2.2.1 (Cumulative Distribution Function (cdf)). [1] Given a measure space with measure $(\Omega, \mathcal{F}, \mu)$ and a random variable X, we define its *cdf* as $F_X(x) = \mu(\{\omega \in \Omega \mid X(\omega) < x\})$.

Definition 2.2.2 (Probability Density Function (pdf)). Given a random variable X, we say that $f_X \colon \mathbb{R} \to [0, \infty]$ is its pdf if

$$F_X(a) = \int_{-\infty}^{a} f_X(x)\,\mathrm{d}x.$$

In what follows, we will always assume that all the random variables admit a pdf distribution.

Definition 2.2.3 (Kernel of a Random Variable). We say that a function g is a kernel of a random variable X if g is proportional to f_X.

[1] The reader familiar with the measure theoretical formalism might have observed that the definition given here is far from being formal. Even though, we consider that there is no point on complicating this introductory exposition with even more technical details.

Remark. It is easy to see that, given a function $g \in \mathcal{L}^1(\mathcal{X})$ for some space $\mathcal{X}$, we can define in a natural way the normalised version of g as $h(x) = \frac{|g(x)|}{\int_{\mathcal{X}} |g(x)| \, \mathrm{d}x}$. In simple words, every $\mathcal{L}^1$ function has an associated random variable. Furthermore, if the kernels of two random variables coincide, then the random variable are identically distributed.

Definition 2.2.4 (Expectation of a Random Variable). Given a random variable g and a distribution π over a space $\mathcal{X}$, we define the *expectation of g with respect to π* as the value of

$$\mathrm{E}_\pi[g] = \int_{\mathcal{X}} g \, \mathrm{d}\pi = \int_{\mathcal{X}} g(x) f_\pi(x) \, \mathrm{d}x$$

if the integral (or the sum, in the discrete case) exists.

When the expectation is taken over the reference measure we say that it is the *mean of the random variable*. In this case we just write $\mathrm{E}[g]$.

Definition 2.2.5 (Moments of a Random Variable). Let X be a random variable defined over a measure space with measure π.

The *n-th raw moment of X* is defined as $\mathrm{E}_\pi[X^n]$.

The *n-th central moment of X* is defined as $\mathrm{E}_\pi[(X - \mathrm{E}[X])^n]$.

The second central moment is called *variance*, and we denote it as $\mathrm{Var}_\pi[X]$.

We define the standard deviation $\sigma \geq 0$ of a random variable as $\sigma = \sqrt{\mathrm{Var}[X]}$.

Definition 2.2.6 (Covariance). The covariance of two random variables X, Y defined over a common probability space with measure π is defined as

$$\mathrm{Cov}_\pi(X, Y) = \mathrm{E}_\pi[(X - \mathrm{E}_\pi[X])(Y - \mathrm{E}_\pi[Y])].$$

The following basic properties of the operators expectation, variance and covariance will be used throughout the rest of the dissertation.

Lemma 2.2.7. *For every distribution π, random variables X, Y and real numbers a, b we have*

- *Linearity of Expectation:* $\mathrm{E}_\pi[aX + bY] = a \, \mathrm{E}_\pi[X] + b \, \mathrm{E}_\pi[Y]$.
- *Variance acting on linear operators:*

  $\mathrm{Var}_\pi[aX + bY] = a^2 \, \mathrm{Var}_\pi[X] + b^2 \, \mathrm{Var}_\pi[Y] + 2ab \, \mathrm{Cov}_\pi(X, Y)$

- Var *is definite positive:* $\mathrm{Var}_\pi[X] \geq 0$.
- $\mathrm{Var}_\pi[X] = 0$ *if and only if* X *is constant.*
- *Decomposition of the variance:* $\mathrm{Var}_\pi[X] = \mathrm{E}_\pi[X^2] - \mathrm{E}_\pi[X]^2$.
- *Decomposition of the covariance:* $\mathrm{Cov}_\pi(X, Y) = \mathrm{E}_\pi[XY] - \mathrm{E}_\pi[X]\,\mathrm{E}_\pi[Y]$

Proposition 2.2.8 (Jensen's inequality for measures). *Let* $\phi\colon \mathbb{R} \to \mathbb{R}$ *be a convex function and* X *a random variable over a measure space. For every distribution* π*, then*

$$\phi(\mathrm{E}_\pi[X]) \leq \mathrm{E}_\pi[\phi(X)]$$

with equality only when ϕ *is not strictly convex (e.g., linear) or when* X *is constant.*

A frequently used tool in statistics applied to machine learning are the divergences. The most popular one is the KL-divergence

Definition 2.2.9 (KL-divergence). The Kullback-Leibler divergence between two real-valued probability measures P and Q with densities p and q, is defined as

$$D_{\mathrm{KL}}(P \,||\, Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) \mathrm{d}x.$$

The following property makes the KL-divergence a reasonable operator to consider when comparing measures.

Proposition 2.2.10 (KL-divergence properties). *For every pair of probability measures* P *and* Q*, the KL-divergence is positive semidefinite and it is zero if and only if* $P = Q$ *almost everywhere.*

Observe that the KL-divergence is not a distance, since it is not symmetric. This makes the choice of this operator to compare different distributions slightly arbitrary. We will just mention that there are deeper information-theoretical properties that support the use of this operator over other semidefinite positive operators.

Finally, we will define some notions on sequences of random variables.

Definition 2.2.11 (Independent and Identically Distributed). A sequence of random variables $\{X_n\}$ is independent and identically distributed if each random variable has the same probability distribution and all of them are mutually independent. We usually write i.i.d. for short.

We sometimes say that these random variables form a *sample* of the underlying random variable.

The two convergence notions that we will use throughout this dissertation are the following.

Definition 2.2.12 (Convergence in distribution). We say that a sequence of random variables $\{X_n\}$ *converges in distribution* to a random variable X if the associated sequence of cdf functions converges pointwise to the cdf function of X.

Definition 2.2.13 (Almost Sure Convergence). A sequence of random variables $\{X_n\}$ converges almost surely to a value x if

$$\Pr\left(\lim_{n\to\infty} X_n = x\right) = 1$$

we write $X_n \xrightarrow{a.s.} x$.

These two forms of convergence are related via the following proposition

Proposition 2.2.14. *Let X_n be a sequence of random variables and X another random variable. If $X_n \xrightarrow{a.s.} X$ then $X_n \xrightarrow{d} X$. The converse is not true in general.*

In short, almost sure convergence is strictly stronger than convergence in distribution.

Theorem 2.2.15 (Strong Law of Large Numbers (LNN)). *Let $\{X_n\}$ be a sequence of i.i.d. random variables with mean μ, then we have that the empirical mean*

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

converges almost surely to μ.

There are many central limit theorems. Here we state one of the most classic versions.

Theorem 2.2.16 (Central Limit Theorem (CLT)). *Let $\{X_i\}$ be a sequence of i.i.d. random variables with mean μ and $\sigma^2 < \infty$. Then, the sequence of random variables $\sqrt{n}(\overline{X}_n - \mu)$ converges in distribution to $\mathcal{N}(0, \sigma^2)$.*

In the exposition we will prove other variations of the central limit theorem for some estimators for Sequential Monte Carlo algorithms.

The following intuitive theorem will come in handy several times during the exposition

Theorem 2.2.17 (Slutsky's theorem). *Let X_n and Y_n be two sequences of random variables such that $X_n \xrightarrow{d} X$ and $Y_n \xrightarrow{d} c$, then we have the following three convergence results*

$$X_n + Y_n \xrightarrow{d} X + c \qquad X_n Y_n \xrightarrow{d} cX \qquad X_n / Y_n \xrightarrow{d} X/c$$

Finally we define the concept of estimator, the main object that we will use in this dissertation.

Definition 2.2.18 (Estimator). Let $\theta \in \Theta$ be a fixed value in some space that we want to estimate. We say that a random variable $X \colon \Omega \to \Theta$ is an estimator of theta, where Ω is some probability space.

This definition is very broad, and as such, not very useful. We usually want the estimator to have certain properties

Definition 2.2.19 (Properties of an estimator). Let X be an estimator and let $\{X_n\}$ be a sequence of estimators for certain value θ.

X is *unbiased* if $\mathrm{E}[X] = \theta$. If not we say that X is *biased*.

The sequence X_n is *consistent* if $X_n \xrightarrow{d} \theta$.

The sequence X_n is *strongly consistent* if $X_n \xrightarrow{a.s.} \theta$.

2.3 Introduction to Bayesian Inference

2.3.1 General Idea

The random variables that we will be interested in studying, are random variables on some product space $\Theta \times \mathcal{Y}$ where Θ is a set of parameters and $\mathcal{Y}$ is a set of data. This random variable is usually factorised as

$$p(\theta, y) = p(\theta) p(y \mid \theta)$$

where θ represents some parameters in our model and $p(y \mid \theta)$ is the *likelihood* of generating a point y given the parameters θ of the model. The elements of $\mathcal{Y}$ are called observations.

We say that $p(\theta)$ is the *prior* on the parameters and $p(y \mid \theta)$ is the *likelihood*.

To perform inference in a model $p(\theta)p(y \mid \theta)$ is to compute or approximate the posterior distribution, this is $p(\theta \mid y)$.

By Bayes' formula, we have that

$$p(\theta \mid y) = \frac{p(\theta)p(y \mid \theta)}{p(y)}.$$

where

$$p(y) = \int_{\Theta} p(y \mid \theta)p(\theta)\,\mathrm{d}\theta.$$

is called the *marginal distribution* with respect to y. The process of passing from a joint distribution $p(\theta, y)$ to a distribution over y is called *marginalisation* or *marginalising with respect to* y.

In general, the integral in $p(y)$ is not tractable, and that is one of the main reasons why doing inference is hard.

Observe that for a fixed y, $p(y)$ is a constant, so

$$p(\theta \mid y) \propto p(y, \theta).$$

2.3.2 Simple Models

In this section we present some basic Bayesian models, for which an explicit posterior can be computed.

Example 2.3.1 (Coin bias model). We have a coin and we would like to create a model of how likely is to get heads or tails.

Let θ be the bias of the coin towards heads, this is, $\theta = 1$ would be a coin that always yields heads. For convenience, we will denote getting heads with a 1 and getting tails with a 0.

Since at the beginning we do not have any information on the bias of the coin, we can assume that the distribution on the parameter space, $[0, 1]$ in this case, is uniform. This is, the prior $p(\theta) = 1$. Furthermore, for just one observation the likelihood is given by a $\mathrm{Ber}(\theta)$ and hence $p(y \mid \theta) = \theta^y(1-\theta)^{1-y}$. Then, the posterior distribution is given by

$$p(\theta \mid y) = \frac{\theta^y(1-\theta)^{1-y}}{\int_0^1 \theta^y(1-\theta)^{1-y}\,\mathrm{d}\theta}.$$

In this case, the posterior follows a beta distribution with parameters $\mathrm{Beta}(y - 1, -y)$.

During the rest of the text we will use the notation $y_{1:n} = y_1, \ldots, y_n$.

If we have n observations, from which n_h are heads and n_t are tails, then we have that the likelihood is

$$p(y_{1:n} \mid \theta) = \prod_{i=1}^{n} \theta^{y_i}(1-\theta)^{1-y_i} = \theta^{n_h}(1-\theta)^{n_t}$$

and the posterior distribution is distributed as a Beta$(n_h - 1, n_t - 1)$.

Example 2.3.2 (Gaussian Conjugate with Unknown Mean). Suppose that in this case we model some data y_i as being generated by a normal distribution $\mathcal{N}(\mu, \sigma^2)$ where σ^2 is known but μ is not. We can assume that $\mu \sim \mathcal{N}(\mu_0, \sigma_0^2)$ for some fixed μ_0, σ_0.

The model can be summarised as

$$\mu \sim \mathcal{N}(\mu_0, \sigma_0^2) \qquad y_i \sim \mathcal{N}(\mu, \sigma^2).$$

In this settings

$$p(\mu, \sigma, \mu_0, \sigma_0, y_{1:n}) = \exp\Big(-\frac{1}{2\sigma_0^2}(\mu - \mu_0)^2\Big) \prod_{i=1}^{n} \exp\Big(-\frac{1}{2\sigma^2}(y_i - \mu)^2\Big)$$

and since the probability $p(\mu)$ is just a constant (the normalisation constant)

$$p(\mu \mid \sigma^2, \mu_0, \sigma_0^2, y_{1:n}) \propto p(\mu, \sigma^2, \mu_0, \sigma_0^2, y_{1:n}).$$

Now we just have to manipulate the expression, considering that the only non-constant variable is μ to get

$$\begin{aligned}
p(\mu \mid \sigma^2, \mu_0, \sigma_0^2, y_{1:n}) &\propto \exp\Big(-\frac{1}{2\sigma^2\sigma_0^2}\Big(\sigma^2(\mu - \mu_0)^2 + \sigma_0^2 \sum_{i=1}^{n}(y_i - \mu)^2\Big)\Big) \\
&\propto \exp\Big(-\frac{1}{2\sigma^2\sigma_0^2}\Big(\sigma^2(\mu - \mu_0)^2 + \sigma_0^2 \sum_{i=1}^{n}(y_i^2 + \mu^2 - 2\mu y_i)\Big)\Big) \\
&\propto \exp\Big(-\frac{1}{2\sigma^2\sigma_0^2}\Big(\sigma^2\mu^2 - 2\sigma^2\mu\mu_0 + n\sigma_0^2\mu^2 - 2n\sigma_0^2\mu\overline{y_{1:n}}\Big)\Big) \\
&\propto \exp\Big(-\frac{\sigma^2 + \sigma_0^2}{2\sigma^2\sigma_0^2}\Big(\mu - \frac{\sigma^2\mu_0 + n\sigma_0^2\overline{y_{1:n}}}{\sigma_0^2 + \sigma^2}\Big)^2\Big)
\end{aligned}$$

where in the last expression we completed the square with respect to μ.

This last expression is just the kernel of the pdf function of $\mathcal{N}(\hat{\mu}, \hat{\sigma})$ for

$$\hat{\sigma} = \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}\right)^{-1} \qquad \hat{\mu} = \hat{\sigma}\left(\frac{\mu_0}{\sigma_0^2} + \frac{n\overline{y_{1:n}}}{\sigma^2}\right)$$

and thus, the posterior distribution over the parameter μ is normally distributed.

In these examples we could compute the posterior distribution analytically. This was not only because the models were simple, but also because the prior and likelihood distributions were carefully chosen.

Definition 2.3.3 (Conjugate Prior). We say that the family of distributions $\mathcal{F}$ is a conjugate prior of the family $\mathcal{G}$ if, for every choice of a likelihood in $\mathcal{G}$ and for every prior chosen from the family $\mathcal{F}$, the posterior distribution is an element of $\mathcal{F}$.

Example 2.3.4. For the family $\mathcal{G} = \{\mathcal{N}(\mu, \sigma) \mid \mu \in \mathbb{R}\}$, the family of normal distributions is a conjugated prior.

Example 2.3.5. The coin toss model problem can be generalised without much work to show that the family of beta distributions is a conjugate prior for $\mathcal{G} = \{\text{Bin}(n, p) \mid p \in [0, 1]\}$.

2.3.3 Graphical Models

Graphical models provide a very compact way of representing two of the main ideas in Bayesian models: Independence between random variables and the definition of observable and latent variables. In this section we introduce the main ideas of directed graphical models. For a more comprehensive approach to this subject we refer the reader to [22].

Definition 2.3.6 (Observable and Latent Variables). The observable variables in a model are the variables that can be observed and directly measured. They are usually represented with the letter Y.

Latent (or hidden) variables are the variables that cannot be measured. They are usually represented either with Greek letters when they correspond to parameters of a distribution or with the letter X when they represent the input data of some process.

Example 2.3.7. In the coin toss problem the bias of the coin θ is a latent variable of the model and the result after a coin toss is an observable variable of the model.

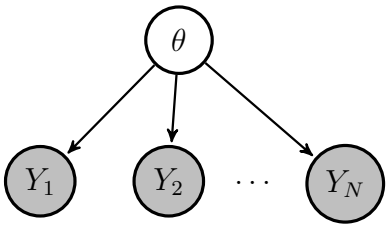

Figure 2.1: Graphical model for the coin toss model with N observations.

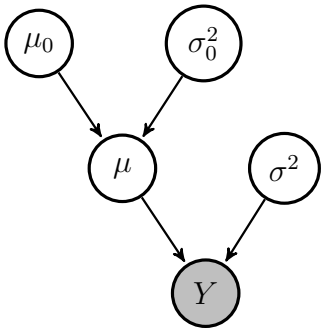

Figure 2.2: Graphical model for a Gaussian model with unknown mean and one observation.

Definition 2.3.8 (Directed Graphical Model). A directed acyclic graph is a *directed graphical model* representing the multidimensional random variable $X = X_{1:n}$ if every node v_i represents a random variables X_i and for every node v_i, the random X_i and the random variables represented by the non-direct predecessors of v_i, $\text{pred}(v_i) \setminus \text{pa}(v_i)$ are conditionally independent given the random variables represented by direct predecessors of v_i $\text{pa}(v_i)$. Formally

$$\Pr(X_i, \text{pred}(v_i) \setminus \text{pa}(v_i) \mid \text{pa}(v_i)) = \Pr(X_i \mid \text{pa}(v_i)) \Pr(\text{pred}(v_i) \setminus \text{pa}(v_i) \mid \text{pa}(v_i)).$$

We will refer to them simply as *graphical models*.

When we draw the graph, by convention the nodes representing observable random variables are shaded in grey and the ones representing hidden random variables are white.

Observe that for every multidimensional random variable there are always many directed graphical models that represent it, since the trivial graph with one node and no edges is always a valid graphical model representing the random variable.

2.4 Probabilistic Programming

2.4.1 Introduction

Probabilistic Programming aims to provide an unified framework to describe probabilistic models and perform inference in those models, easing the development process of modelling data.

A probabilistic programming system provides syntax for the definition and conditioning of random variables.

To define a distribution in the sense of Bayesian statistics, probabilistic programs have two extra functions to define priors and likelihoods. To define priors, we have the statement `sample` and to define likelihoods we have the statement `observe`.

Example 2.4.1. Juan works in a company from Monday to Friday. Juan starts very motivated every week, but this motivation decays as the weekend approaches. As a result, the number of times that he checks Facebook increases linearly during the week. If on Monday he just checks Facebook, on average, a couple of times, while this number increases as $f(x) = 2x + 2$ throughout the week for $x = 0, \ldots, 4$.

Juan has an doctor's appointment on Wednesday this week, so he would like to know what day is today. The only information he has is that today he checked Facebook 7 times.

Leaving aside considerations on the effectiveness of his method for determining whether he has to go to the doctor today, he can express this problem as a probabilistic program as showed in listing 2.1.

```
1 void juan() {
2     int fb_checks = 7;
3     int day = sample(uniform(5));
4     observe(poisson(2 * day + 2), fb_checks);
5     predict(day == 2);
6 }
```

Listing 2.1: Juan's inference problem as a probabilistic program in C++.

In this model we see how we set an uniform prior on the days of the week on line 2, then we condition the model on the number of times that Juan has checked that day and finally we tell the model to report us what is the probability with which the—latent—variable `day` is equal to 2, this is, that it is Wednesday.

In this case we can compute this probability explicitly. Recall that the density of a Poisson distribution is given by the formula

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

for $k \in \mathbb{N}$ and $\lambda \in \mathbb{R}^+$. Hence, the probability that Juan has checked Facebook in a Wednesday is given by the formula

$$\Pr(\texttt{day} = 2, \texttt{fb\_checks} = 7) = \frac{1}{5}\frac{6^7 e^{-6}}{7!} \approx 0.0275$$

and he probability that he checked Facebook 7 is

$$\Pr(\texttt{fb\_checks} = 7) = \frac{1}{5}\sum_{k=0}^{4}\frac{(2k+2)^7 e^{-(2k+2)}}{7!} \approx 0.0860.$$

Finally, applying Bayes' rule, we have that the probability that today is Wednesday given that he checked Facebook 7 times is

$$\Pr(\texttt{day} = 2 \mid \texttt{fb\_checks} = 7) = \frac{\Pr(\texttt{day} = 2, \texttt{fb\_checks} = 7)}{\Pr(\texttt{fb\_checks} = 7)} \approx 0.319.$$

Let us analyse why we could compute this probability. We computed the probability of the observation, given the target day of the week, this is, the joint probability. If we observe that formula closely, this formula is equivalent to setting $\texttt{day} = 2$, executing the program forward and accumulating the likelihoods. Even though this way we can approximate the joint distribution, we are still missing the normalisation constant, $\Pr(\texttt{fb\_checks} = 7)$. Computing this constant is not so straightforward. We will devote the entirety of next chapter to explain in detail how to approximate this constant when it cannot be analytically computed.

2.4.2 From Simulators to Probabilistic Programs

In this section we will look at simulators through the perspective of probabilistic programming, and we will argue how probabilistic programming can be used to develop tools as auto-tuners or performing general inference.

A simulator is a program such that replicates some real-world process. These simulators can go from a program that, given some shape and the wind momentum, generates the pressure suffered by the wing on some points, to simulators of the standard model in particle physics that, given some particles and their momenta, returns the output particles generated by the interactions between them. Let us formalise these ideas.

Definition 2.4.2 (Simulator). A simulator-based model is a collection of random variables $g(\cdot, \theta)$ parametrised by θ over some sample space.

This is idea is formalised with the concept of a probability kernel. We say that $\kappa(\cdot, \cdot)$ is a probability kernel if it is a probability measure in the first variable and it is a measurable function in the second.

Random algorithms can be seen as simulators since, for a fixed input, the algorithm samples from different random variables during its execution and then generates an output.

In this context, the samples in the random algorithm from random variables can be seen in the Bayesian framework as prior distributions over the variables that are assigned to. Furthermore, if we have an execution trace that drew n random values $x_{1:n}$ from distributions with densities p_i we can define

$$\Pr(x_{1:n}) = \prod_{i=1}^{n} p_i(x_i).$$

Hence, we can assign a probability to an execution trace. In Juan's example, this probability was uniform over the different days of the week.

An important part of the development of models is the choice of the noise in the observations. This follows from the fact that observations might be inherently random, or that they are never perfectly correct, and neither is our model. This idea can be seen in the Bayesian framework as the likelihood function

$$\Pr(y \mid x_{1:n}).$$

In example 2.4.1 the likelihood function was a Poisson with mean $f(x) = 2x + 2$. This effectively models how is distributed the number of times that Juan checks Facebook given certain mean.

With these two ideas, we can see a simulator-based model as a family of random variables that define joint distributions over the space of sampled random variables and the space of outputs.

Finally, if we put a prior on the parameter θ, we have that a simulator-based model can be seen as a distribution over the set of input parameters, random draws and outputs

$$p(\theta, x, y).$$

In this framework, if we are able to perform inference, given an output that we are interested in studying, we can obtain a distribution over $p(\theta, x \mid y)$. If then, for example, we compute the maximum a posteriori estimator over the coordinate θ, we may get an approximation of the parameters that were most likely to generate y.

This approach is followed in [25]. In this case the simulator is a captcha generator with inputs a string of numbers and letters, a font style and other parameters. In this case, with a uniform distribution over the inputs, computing the maximum a posteriori estimator is equivalent to breaking the captcha.

Others have applied this approach to hand-picked parameters deterministic programs to obtain the so-called auto-tuners. For example, in [9], they were able to optimise `libstdc++`'s implementation of `std::sort`, arguably the most efficient implementation of a comparison-based sorting algorithm.

Chapter 3

Sampling Algorithms

In this chapter we introduce particle filters and smoothers. The first particle method family that we present is IS. This will be the particle method family on which we will base the *compiled inference* algorithm for our inference framework. The second one, SMC, is one of the most classical particle methods used currently. We use SMC as a baseline for the experiments in chapter 6.

3.1 Introduction

In section section 2.3.2 we introduced a few very simple models showed how to compute explicitly the posterior distributions. Here we will introduce state-space models and we will show a few motivating examples.

3.1.1 State-Space Models

Definition 3.1.1 (Stochastic Process). Let $(\mathcal{X}, \mathcal{F}, P)$ be probability space, let (Ω, Σ) be a measurable space and let T be a set of indices. An $\mathcal{X}$*-valued stochastic process* is a sequence of random variables from the sample space $\mathcal{X}$ to the space Ω indexed by the elements of T.

In our case, state-space models, T will always be the natural numbers or a subset of them.

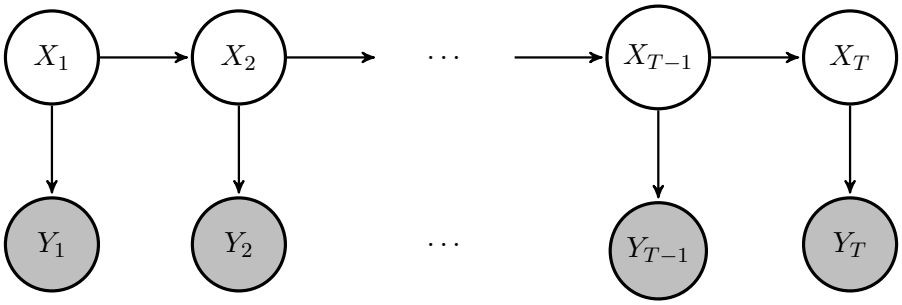

Figure 3.1: Graphical model representing a Hidden Markov Model (HMM).

Definition 3.1.2 (Markov Process). A *Markov process* is a class of $\mathcal{X}$-valued stochastic process such that the Markov property holds, this is

$$\Pr(X_n = x_n \mid X_{n-1} = x_{n-1}, \ldots, X_1 = x_1) = \Pr(X_n = x_n \mid X_{n-1} = x_{n-1}).$$

We denote the initial density and the marginals as

$$\Pr(X_1 = x_1) = p(x_1) \qquad \Pr(X_n = x_n \mid X_{n-1} = x_{n-1}) = f(x_n \mid x_{n-1}).$$

Definition 3.1.3 (Hidden Markov Model (HMM)). It is defined by a Markov Process, where we just have access to a set of $\mathcal{Y}$-valued observations $\{Y_n\}$. $\{Y_n\}$ are independent given $\{X_n\}$ and their marginal densities are given by

$$\Pr(Y_n = y_n \mid X_n = x_n) = g(y_n \mid x_n).$$

The model can be represented by the graphical model shown in figure 3.1 where the white nodes are the hidden variables and they grey nodes are the observable variables.

In the HMM model, we have that the full joint distribution is defined as

$$p(x_{1:T}, y_{1:T}) = p(x_1) \prod_{t=2}^{T} f(x_t \mid x_{t-1}) \prod_{t=1}^{T} g(y_t \mid x_t).$$

We present HMM's as the simplest example of state-space models. Hidden Markov Models are very popular in the literature, since in many cases they support efficient inference. Examples of these are finite state-space HMM's via the Forward-Backward recursion and Linear Gaussian Models via Kalman filters. On the other hand, if we consider more complex models we eventually hit a wall, since most of

the optimisation problems in non-linear, non-Gaussian state-space models do not have analytic solutions. In these cases, the integrals involved are too difficult to approximate by standard methods as Monte Carlo integration. To attack these problems, particle methods were introduced in 1993 by Gordon et.al. [16]. From there, particle methods have become the go-to tool in many fields to approximate intractable posteriors and, effectively, perform inference in probabilistic models.

3.1.2 Problem Setting

SMC methods are a special case of Monte Carlo methods that sample sequentially from a sequence of target probability densities $\{\pi_n(x_{1:n})\}$ of increasing dimension, where each distribution $\pi_n(x_{1:n})$ is defined on the product space $\mathcal{X}^n$ [13].

In general we just have access to π_n pointwise up to a normalising constant

$$\pi_n(x_{1:n}) = \frac{\gamma_n(x_{1:n})}{Z_n} \tag{3.1}$$

where $\gamma_n > 0$ and Z_n is the normalising constant

$$Z_n = \int_{\mathcal{X}^n} \gamma_n(x_{1:n}) \, \mathrm{d}x_{1:n}.$$

Remark. When it is clear from the context, we will omit the domain on which we are integrating. This is in general a common practice, but sometimes it may hide subtle problems, as we will see in section 3.4.

In many cases, we will be interested in finding estimators for the expectation of a test function $f \colon \mathcal{X}^n \to \mathbb{R}$ with respect to the distributions π_n. We will denote by $\mu = \mathrm{E}_{\pi_n}[f]$ when π_n is clear from the context. We will also assume that $\mu < \infty$. We will not use any subindex in the variable f to relax a little bit the already very index-heavy notation of the dissertation.

Now we introduce the two main areas of interest in inference problems in state-space models, filtering and smoothing.

Definition 3.1.4 (Filtering). The filtering distribution in a state-space model of length T is defined as the posterior at time n given the observations until that point. Formally, we define the filtering distributions as $\{p(x_{1:n} \mid y_{1:n})\}_{n=1}^{T}$ where p denotes the conditional probability defined by the model.

In these problems we may also be interested in characterising the *marginal likelihoods* $\{p(y_{1:n})\}_{n=1}^{T}$.

Definition 3.1.5 (Smoothing). The target distribution in a smoothing problem over a state-space of length T is defined as $p(x_{1:T} \mid y_{1:T})$. In this setting we are also interested in approximating the associated marginals $\{p(x_n \mid y_{1:T})\}_{n=1}^{T}$.

Remark. Filtering approximations can be used to solve smoothing problems since the last filtering distribution is exactly the target distribution in a smoothing problem. On the other hand, in general, SMC based filtering algorithms perform poorly for large T as we will point out in section 3.3.

In the previous set-up, if we fix $y_{1:n}$ and we let $\gamma_n(x_{1:n}) = p(x_{1:n}, y_{1:n})$ so that $Z_n = p(y_{1:n})$, we recover the filtering problem, since $\pi_n(x_{1:n}) = p(x_{1:n} \mid y_{1:n})$. On the other hand, if we set $\gamma_n(x_{1:n}) = p(x_{1:n}, y_{1:T})$ we recover the smoothing problem.

Remark. To keep the notation across the exposition uniform, we will always use the state-space notation. During the presentation of the first estimators this notation will be unnecessarily verbose, but its utility will become apparent with the introduction of sequential sampling methods.

3.1.3 The Monte Carlo Estimator

The first estimator that we will present is the Monte Carlo Estimator. This estimator assumes that we that we can sample from the target distributions π_n. In this case, let $X_{1:n}^i \sim \pi_n$ be N independent random variables. The Monte Carlo estimator is defined as

$$\pi_n^{\mathrm{MC}}(x_{1:n}) = \frac{1}{N} \sum_{i=1}^{N} \delta_{X_{1:n}^i}(x_{1:n}).$$

Using this approximation, we can approximate the expectation μ via the estimator

$$\mu^{\mathrm{MC}} = \int f(x_{1:n}) \pi_n^{\mathrm{MC}}(x_{1:n}) \, \mathrm{d}x_{1:n} = \frac{1}{N} \sum_{i=1}^{n} f(X_{1:n}^i).$$

μ^{MC} is unbiased since

$$\mathrm{E}_{\pi_n}[\mu^{\mathrm{MC}}] = \mathrm{E}_{\pi_n}\left[\frac{1}{N} \sum_{i=1}^{n} f(X_{1:n}^i)\right] = \frac{1}{N} \sum_{i=1}^{n} \mathrm{E}_{\pi_n}[f(X_{1:n}^i)] = \mu,$$

has variance

$$\mathrm{Var}_{\pi_n}[\mu^{\mathrm{MC}}] = \frac{1}{N}\mathrm{Var}_{\pi_n}[f] = \frac{1}{N}(\mathrm{E}_{\pi_n}[f^2] - \mu^2)$$

and is strongly consistent (LLN). Furthermore, it satisfies the CLT

$$\sqrt{n}(\mu^{\mathrm{MC}} - \mu) \xrightarrow{d} \mathcal{N}(0, \mathrm{Var}_{\pi_n}[f]).$$

The advantage of the Monte Carlo estimator is that its variance decays at a rate $\mathcal{O}(1/N)$. This means that the variance of the estimator is independent of the dimension of the integral being considered, in contrast to integration methods based on Riemann sums. On the other hand, the assumption of being able to sample from the distribution π_n is, in general, too strong. We will introduce now several methods to overcome this problem.

3.2 Importance Sampling

We will first introduce Importance Sampling (IS) and some variations of this method. This will provide a solid starting point for particle methods. After that, we will introduce Sequential Monte Carlo (SMC) methods and we will show how this family of algorithms generalises the ideas present in IS and how it solves certain problems that IS presents. Finally, we will end up introducing Compiled Sequential Importance Sampling (CSIS), the main inference algorithm used in the library that we present in this dissertation.

3.2.1 Basic Importance Sampling

In this section we will assume that, although we cannot sample from π_n, we do have access to evaluate its pdf pointwise.

In *importance sampling* methods, we introduce a family of distributions q_n defined over $\mathcal{X}^n$. We will choose q_n such that we can sample from it and $q_n > 0$ if $\pi_n > 0$. In these settings, we can express the target value μ as

$$\begin{aligned} \mu = \mathrm{E}_{\pi_n}[f] &= \int f(x_{1:n})\pi_n(x_{1:n})\,\mathrm{d}x_{1:n} \\ &= \int f(x_{1:n})\frac{\pi_n(x_{1:n})}{q_n(x_{1:n})}q_n(x_{1:n})\,\mathrm{d}x_{1:n} = \mathrm{E}_{q_n}\left[f\frac{\pi_n}{q_n}\right]. \end{aligned} \tag{3.2}$$

We say that q_n is the *proposal distribution*. Since we have chosen q_n specifically so we can sample from them, now we may draw N samples $X^i_{1:n} \sim q_n$ and the importance sampling estimator just follows naturally

$$\mu^{\mathrm{IS}} = \frac{1}{N}\sum_{i=1}^{n} \frac{f(X^i_{1:n})\pi_n(X^i_{1:n})}{q_n(X^i_{1:n})}.$$

It is usually very helpful to think of importance sampling from a measure-theoretical perspective. In this framework, importance sampling is nothing but a change of measure in the system that we want to estimate. The term π_n/q_n accounts for this change. To keep the ideas simple, as we said at the beginning of the exposition, we will always assume that all the random variables that we are considering admit a pdf function, so we will keep a more *reader-friendly* notation. For an approximation to the subject of Sequential Monte Carlo algorithms from a formal measure theoretical perspective, we refer the reader to the part one of [5], especially chapters 7, 8 and 9.

Proposition 3.2.1. *μ^{IS} is unbiased and has variance*

$$\begin{aligned} \mathrm{Var}_{q_n}[\mu^{\mathrm{IS}}] := \sigma^{2\ \mathrm{IS}} &= \frac{1}{N}\Big(\int \frac{(f(x_{1:n})\pi_n(x_{1:n}))^2}{q_n(x_{1:n})}\,\mathrm{d}x_{1:n} - \mu^2\Big) \\ &= \frac{1}{N}\int \frac{(f(x_{1:n})\pi_n(x_{1:n}) - \mu q_n(x_{1:n}))^2}{q_n(x_{1:n})}\,\mathrm{d}x_{1:n} \end{aligned} \tag{3.3}$$

Proof. The fact that μ^{IS} is unbiased follows directly from the definition of μ^{IS}. The two expressions for $\sigma^{2\ \mathrm{IS}}$ follow from the formula $\mathrm{Var}_\pi[X] = \mathrm{E}_\pi[X^2] - \mathrm{E}_\pi[X]^2$. □

Remark. The variance of the estimator μ^{IS} depends heavily on the function f. This should come as no surprise, since different target functions will distribute the mass over different areas in their domain. As an extreme example, consider the functions $f_1(x) = \mathbb{1}(x > 1000)$ and $f_2(x) = \mathbb{1}(x < -1000)$.

Remark. The second expression for $\sigma^{2\ \mathrm{IS}}$ indicates how good is a proposal. The variance of the estimator is small when $f(x_{1:n})\pi_n(x_{1:n}) - \mu q_n(x_{1:n})$ is small, in other words, when

$$q_n(x_{1:n}) \propto f(x_{1:n})\pi_n(x_{1:n}).$$

From the denominator we observe that when q_n is not proportional to $f\pi_n$, small values of q_n highly increase the variance of the estimator. This leads to the idea that

light-tailed proposal distributions might lead to disastrous results. This intuition is confirmed with the following standard example:

Example 3.2.2 (Sample from Gaussian with Gaussian proposal). Let $f(x) = x$, $\pi_n \sim \mathcal{N}(0,1)$ and $q_n \sim \mathcal{N}(0,\sigma^2)$ for $\sigma > 0$. Then we have that

$$\begin{aligned}
\sigma^{2\ \mathrm{IS}} &= \int_{-\infty}^{\infty} x^2 \frac{(\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}})^2}{\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}} \,\mathrm{d}x \\
&= \sigma \int_{-\infty}^{\infty} \frac{x^2}{\sqrt{2\pi}} e^{-\frac{x^2(2-\sigma^{-2})}{2}} \,\mathrm{d}x \\
&= \sigma(2-\sigma^{-2})^{-1/2} \operatorname{Var}\Big[\mathcal{N}(0,(2-\sigma^{-2})^{-1})\Big] \\
&= \begin{cases} \frac{\sigma}{(2-\sigma^{-2})^{3/2}}, & \sigma^2 > \frac{1}{2} \\ \infty, & \text{else.} \end{cases}
\end{aligned}$$

This example shows two important ideas. The first one is that the variance of μ^{IS} can be infinity even when the variance of π_n and q_n are not. We will characterise this idea later, but for now the take-home message is that if our proposal distribution is *not similar enough* to π_n and has light tails, then we might have some severe variance problems. The second one is that, contrary to what one might expect at a first sight, the optimal σ for this problem is not 1 but $\sqrt{2}$. This makes again explicit the dependency of the variance of μ^{IS} with respect to the function f.

Remark (Zero variance estimator). If $f(x_{1:n}) \geq 0$ and $\mu > 0$, the proposal distribution that makes the numerator of $\sigma^{2\ \mathrm{IS}}$ be exactly zero is

$$q_n^*(x_{1:n}) = \frac{f(x_{1:n})\pi_n(x_{1:n})}{\mu}.$$

This is a random variable since it integrates to 1 and it attains $\sigma^{2\ \mathrm{IS}}_{q_n^*} = 0$, making it optimal. On the other hand, this proposal is not very practical in realistic scenarios since its definition involves exactly the quantity μ which is the quantity that we are trying to approximate. If we had access to such a random variable we could compute μ directly from q_n^*, f and π_n as

$$\mu = \frac{f(X_{1:n})\pi_n(X_{1:n})}{q_n^*(X_{1:n})}$$

with just one sample $X_{1:n} \sim q_n^*$.

There exists an optimal proposal distribution for functions f that are not necessarily positive—or negative—definite.

Proposition 3.2.3. *Let $f\colon \mathcal{X}^n \to \mathbb{R}$ be a measurable function with respect to a measure π_n, then there exists a proposal distribution q_n^* such that it minimises $\sigma_n^{2\,\mathrm{IS}}$ and it is given by*

$$q_n^*(x_{1:n}) = \frac{|f(x_{1:n})|\pi_n(x_{1:n})}{\mathrm{E}_{\pi_n}[|f|]}.$$

Proof. From the first expression for $\sigma^{2\,\mathrm{IS}}$ we have that for every distribution q_n

$$\begin{aligned}
\sigma_{q_n^*}^{2\,\mathrm{IS}} + \mu^2 &= \int \frac{(f(x_{1:n})\pi_n(x_{1:n}))^2}{q_n^*(x_{1:n})}\,\mathrm{d}x_{1:n} \\
&= \int \frac{(f(x_{1:n})\pi_n(x_{1:n}))^2}{|f(x_{1:n})|\pi_n(x_{1:n})/\,\mathrm{E}_{\pi_n}[|f|]}\,\mathrm{d}x_{1:n} \\
&= \mathrm{E}_{\pi_n}[|f|]^2 = \mathrm{E}_{q_n}\left[|f|\frac{\pi_n}{q_n}\right]^2 \\
&\leq \mathrm{E}_{q_n}\left[\left(|f|\frac{\pi_n}{q_n}\right)^2\right] = \sigma_{q_n}^{2\,\mathrm{IS}} + \mu^2
\end{aligned}$$

where the inequality follows from Jensen's inequality. □

This proof is slightly artificial, since we had to specify the optimal proposal distribution beforehand. To deduce the form of q_n^* one may resort to second variation methods in calculus of variations, although these methods are far beyond the scope of this exposition.

This general optimal proposal does not have zero variance, but there are zero variance estimators that just need two samples [30].

For this estimator, we also have the problem that q_n^* will not be in general realisable, since if the problem that we want to solve is to approximate μ, we will rarely have access to a good approximation for $\mathrm{E}_{\pi_n}[|f|]$. On the other hand, it confirms the idea that the first optimal for positive definite functions gave us: The best possible proposal is one that is proportional to $|f|\pi_n$.

3.2.2 Normalised Importance Sampling

Basic Importance Sampling assumes that we can compute the pdf of π_n. In most of the scenarios that we will encounter we will not be able to compute the

pdf of π_n, but we will have access to an unnormalised version of it $\gamma_n \colon \mathcal{X}^n \to \mathbb{R}^+$ as defined in equation (3.1).

In this case we may define the unnormalised weight function as

$$w_n(x_{1:n}) = \frac{\gamma_n(x_{1:n})}{q_n(x_{1:n})}.$$

With this notation we have

$$\begin{aligned}
\gamma_n(x_{1:n}) &= w_n(x_{1:n}) q_n(x_{1:n}) \\
Z_n &= \int w_n(x_{1:n}) q_n(x_{1:n}) \, \mathrm{d}x_{1:n} \\
\pi_n(x_{1:n}) &= \frac{w_n(x_{1:n}) q_n(x_{1:n})}{Z_n}.
\end{aligned}$$

These expressions induce the corresponding estimators with $X^i_{1:n} \sim q_n$

$$\begin{aligned}
\gamma_n^{\mathrm{NIS}}(x_{1:n}) &= \frac{1}{N} \sum_{i=1}^{N} w_n(X^i_{1:n}) \delta_{X^i_{1:n}}(x_{1:n}) \\
Z_n^{\mathrm{NIS}} &= \frac{1}{N} \sum_{i=1}^{N} w_n(X^i_{1:n}) \\
\pi_n^{\mathrm{NIS}}(x_{1:n}) &= \sum_{i=1}^{N} W_n^i \delta_{X^i_{1:n}}(x_{1:n})
\end{aligned}$$

where W_n^i are the normalised weights defined as

$$W_n^i = \frac{w_n(X^i_{1:n})}{\sum_{j=1}^{N} w_n(X^j_{1:n})}.$$

The estimator for π_n and the associated estimator for μ

$$\mu^{\mathrm{NIS}} = \sum_{i=1}^{N} W_n^i f(X^i_{1:n})$$

are the ratio of two estimators. Z_n^{NIS} is unbiased and it is strongly consistent (LLN). On the other hand, the estimator μ^{NIS}, as most estimators that come from a ratio of estimators, is biased. We can still get an approximation of its mean and variance using the delta method.

Proposition 3.2.4 (Mean and Variance for μ^{NIS}). *The estimator μ^{NIS} is biased for any finite number of samples and the following first order approximations hold*

$$\mathrm{E}_{q_n}[\mu^{\text{NIS}} - \mu] \approx \frac{1}{NZ_n^2}(\mu \operatorname{Var}_{q_n}[w_n] - \operatorname{Cov}_{q_n}(fw_n, w_n))$$
$$\operatorname{Var}_{q_n}[\mu^{\text{NIS}}] \approx \frac{1}{NZ_n^2}(\operatorname{Var}_{q_n}[fw_n] + \mu^2 \operatorname{Var}_{q_n}[w_n] - 2\mu \operatorname{Cov}_{q_n}(fw_n, w_n)).$$

Proof. In this proof all the expectations are taken over q_n. Let

$$x = \frac{1}{N}\sum_{i=1}^{N} f(X_{1:n}^i) w_n(X_{1:n}^i) \qquad y = \frac{1}{N}\sum_{i=1}^{N} w_n(X_{1:n}^i).$$

The first order Taylor expansion of $f(x, y) = \frac{x}{y}$ around $(\mathrm{E}[x], \mathrm{E}[y])$ is

$$\frac{x}{y} \approx \frac{\mathrm{E}[x]}{\mathrm{E}[y]} + (x - \mathrm{E}[x])\frac{1}{\mathrm{E}[y]} - (y - \mathrm{E}[y])\frac{\mathrm{E}[x]}{\mathrm{E}[y]^2}.$$

Hence, we can approximate the variance as

$$\operatorname{Var}\left[\frac{x}{y}\right] \approx \frac{\operatorname{Var}[x]}{\mathrm{E}[y]^2} + \operatorname{Var}[y]\frac{\mathrm{E}[x]^2}{\mathrm{E}[y]^4} - 2\operatorname{Cov}(x, y)\frac{\mathrm{E}[x]}{\mathrm{E}[y]^3}.$$

Finally, since $\mathrm{E}[x] = \mu Z_n$, $\mathrm{E}[y] = Z_n$ we get

$$\operatorname{Var}[\mu^{\text{NIS}}] \approx \frac{1}{NZ_n^2}(\operatorname{Var}[fw_n] + \mu^2 \operatorname{Var}[w_n] - 2\mu \operatorname{Cov}(fw_n, w_n)).$$

To approximate of the expectation, we just have to apply the operator E to a second order Taylor expansion to get

$$\mathrm{E}\left[\frac{x}{y}\right] \approx \frac{\mathrm{E}[x]}{\mathrm{E}[y]} + \operatorname{Var}[y]\frac{\mathrm{E}[x]}{\mathrm{E}[y]^3} - \operatorname{Cov}(x, y)\frac{1}{\mathrm{E}[y]^2}$$

and

$$\mathrm{E}[\mu^{\text{NIS}} - \mu] \approx \frac{1}{NZ_n^2}(\mu \operatorname{Var}[w_n] - \operatorname{Cov}(fw_n, w_n)).$$

□

These last proposition allows us to give several asymptotic results for μ^{NIS}. The first one is a direct result of the application of the LLN and Slutsky's theorem to the ratio estimator. So, even thought μ^{NIS} is biased for every finite N, it still converges almost surely to the true value μ. We will prove a finer result regarding this

convergence. For that, we just need to slightly modify the proof in proposition 3.2.4 to show that the order of the approximation is indeed $\mathcal{O}(1/N)$, this is, the discarded terms in the Taylor expansion are in $\mathcal{O}(1/N^2)$. This and the LLN yield the following asymptotic limit for the bias of the mean

Proposition 3.2.5. *μ^{NIS} converges almost surely to μ as*

$$N \,\mathrm{E}_{q_n}[\mu^{\text{NIS}} - \mu] \xrightarrow{a.s.} \frac{1}{Z_n^2}(\mu \,\mathrm{E}_{q_n}[w_n^2] - \mathrm{E}_{q_n}[f w_n^2]) = -\int \frac{\pi_n^2(x_{1:n})}{q_n(x_{1:n})}(f(x_{1:n}) - \mu)\,\mathrm{d}x_{1:n}.$$

Finally, the normalised importance sampling estimator follows the following Central Limit Theorem

Proposition 3.2.6. *μ^{NIS} is normally distributed in the limit as*

$$\sqrt{N}(\mu^{\text{NIS}} - \mu) \xrightarrow{d} \mathcal{N}(0, \sigma^{2\ \text{NIS}})$$

where

$$\sigma^{2\ \text{NIS}} = \int \frac{\pi_n^2(x_{1:n})}{q_n(x_{1:n})}(f(x_{1:n}) - \mu)^2\,\mathrm{d}x_{1:n}.$$

Proof. We can write

$$\sqrt{N}(\mu^{\text{NIS}} - \mu) = \frac{\frac{1}{\sqrt{N}}\sum_{i=1}^N w_n(X_{1:n}^i)(f(X_{1:n}^i) - \mu)}{\frac{1}{N}\sum_{i=1}^N w_n(X_{1:n}^i)}.$$

Since the numerator is unbiased, by the CLT it tends in distribution to $\mathcal{N}(0, \sigma^2)$ with

$$\sigma^2 = \int \frac{\gamma_n^2(x_{1:n})}{q_n(x_{1:n})}(f(x_{1:n}) - \mu)^2\,\mathrm{d}x_{1:n}.$$

Furthermore, the denominator tends almost surely to Z_n by the LLN and by Slutsky's theorem the result follows. □

In this case, we have the analogue result on optimal proposal distribution to the one for Importance Sampling.

Proposition 3.2.7. *There exists an optimal proposal for normalised importance sampling that yields a minimal asymptotic variance, and its given by*

$$q_n^*(x_{1:n}) = \frac{|f(x_{1:n}) - \mu|\pi_n(x_{1:n})}{\int |f(x_{1:n}) - \mu|\pi_n(x_{1:n})\,\mathrm{d}x_{1:n}}.$$

Proof. The proof is identical to the one for importance sampling. □

Finally, we present a property of Z_n^{NIS} that will come in handy later to compare the effectiveness of different methods.

Proposition 3.2.8 (Properties of Z_n^{NIS}). *The estimator Z_n^{NIS} is unbiased and has a relative variance of*

$$\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{NIS}}]}{Z_n^2} = \frac{1}{N}\Big(\int \frac{\pi_n^2(x_{1:n})}{q_n(x_{1:n})}\,\mathrm{d}x_{1:n} - 1\Big). \tag{3.4}$$

Proof. Analogous to the proof of proposition 3.2.1. □

3.2.3 Sequential Importance Sampling

The convergence and variance results presented in the last section might be of limited use in practical scenarios where we might want to approximate several test functions f, but they provide very useful theoretical insight of the properties of both importance sampling and normalised importance sampling. It is always useful to take them in account when designing proposals for an importance sampler.

Even when we are just interested in estimating one function, this approach can be problematic. Imagine that we have access to the optimal proposal at any time $q_n^*(x_{1:n})$. This distribution will almost certainly not be the marginal distribution of $q_{n+1}^*(x_{1:n+1})$. In this case, to compute $q_{n+1}^*(x_{1:n+1})$ we may have to evaluate again the n points, which incurs on a linear cost in each iteration. To solve this problem we introduce a more restrictive version of normalised importance sampling.

Sequential Importance Sampling is nothing but a sub-family of normalised importance sampling algorithms for which the computational complexity at each time-step is fixed.

In sequential importance sampling we consider proposals with the following structure

$$\begin{aligned} q_n(x_{1:n}) &= q_{n-1}(x_{1:n-1})q_n(x_n|x_{1:n-1}) \\ &= q_1(x_1)\prod_{t=2}^{n} q_t(x_t|x_{1:t-1}). \end{aligned}$$

This structure also allows us, as we promised, to compute the weights w_n in

amortised constant time. To do so we just have to observe that

$$\begin{aligned} w_n(x_{1:n}) &= \frac{\gamma_n(x_{1:n})}{q_n(x_{1:n})} \\ &= \alpha_n(x_{1:n}) \frac{\gamma_{n-1}(x_{1:n-1})}{q_{n-1}(x_{1:n-1})} \end{aligned}$$

where α is the *incremental importance weight function* given by

$$\alpha_n(x_{1:n}) = \frac{\gamma_n(x_{1:n})}{\gamma_{n-1}(x_{1:n-1}) q_n(x_n \mid x_{1:n-1})}.$$

This approach gives us the following strongly consistent incremental estimator for $X^i_{1:n} \sim q_n$

$$\frac{Z_n}{Z_{n-1}}^{\text{SIS}} = \sum_{i=1}^{N} W^i_{n-1} \alpha_n(X^i_{1:n})$$

just by nothing that

$$\begin{aligned} &\int \alpha_n(x_{1:n}) \pi_{n-1}(x_{1:n-1}) q_n(x_n \mid x_{1:n-1}) \, \mathrm{d}x_{1:n} \\ &\qquad = \int \frac{\gamma_n(x_{1:n}) \pi_{n-1}(x_{1:n-1}) q_n(x_n \mid x_{1:n-1})}{\gamma_{n-1}(x_{1:n-1}) q_n(x_n \mid x_{1:n-1})} \, \mathrm{d}x_{1:n} \\ &\qquad = \frac{Z_n}{Z_{n-1}} \end{aligned}$$

Given these restrictions for q_n, a sensible choice of q_n would be the one that minimises $\mathrm{Var}_{q_n}[w_n]$ since this expression is involved in both the bias and the variance of the estimator μ^{NIS} as the leading coefficient if we see the approximation as a polynomial in μ. Marginalising out $x_{1:n-1}$ in equation (3.4) we get that the optimal distribution in this case is

$$q^*_n(x_{1:n}) = \pi_n(x_n \mid x_{1:n-1}).$$

In general, it will be not possible to sample exactly from $\pi_n(x_n | x_{1:n-1})$ but, as usual, it can be used as an objective, and we can try to design our proposal distributions q_n so that they approximate it.

3.2.4 Problems of Importance Sampling

All the importance sampling algorithms presented suffer from a severe drawback when we deal with high-dimensional distributions. To see this, we shall remark the importance of the quantity $\mathrm{E}_{q_n}[w_n^2]$. For Normalised Importance Sampling, the quantity $\mathrm{E}_{q_n}[w_n^2]$ plays a very important role, since it is involved in the bias and variance of the estimator μ^{NIS}. For example, by proposition 3.2.5, the convergence of the bias of μ^{NIS} can be rewritten as

$$N\,\mathrm{E}_{q_n}[\mu^{\mathrm{NIS}} - \mu] \xrightarrow{a.s.} -\frac{1}{Z_n^2}\,\mathrm{E}_{q_n}[w_n^2(f-\mu)].$$

This expression depends indirectly on $\mathrm{E}_{q_n}[w_n^2]$. We have that $\mathrm{Var}_{q_n}[\mu^{\mathrm{NIS}}]$ depends on this value in a similar way.

These estimators suggest that it might be beneficial to minimise the value $\mathrm{E}_{q_n}[w^2]$ to reduce the bias and the variance of the normalised estimators. A much more important reason of why it might be important to minimise this estimator is shown in proposition 3.4.1.

Although this goal looks as feasible as the others, it will turn out to be extremely challenging, as we will show now.

Suppose that we want to sample form distribution that fully factorises over its coordinates, $\pi_n(x_{1:n}) = \prod_{k=1}^n \pi_n(x_k)$. In this case, a natural choice for the distribution is a distribution that also factorises as $q_n(x_{1:n}) = \prod_{k=1}^n q_n(x_k)$. Then we can define the weights $w_n(x_k) = \gamma_n(x_k)/q_n(x_k)$ and we have that

$$\mathrm{E}_{q_n}[w_n^2(x_{1:n})] = \prod_{k=1}^n (1 + \mathrm{Var}_{q_n(x_k)}[w_n(x_k)]).$$

This expression tells us that we need to choose proposal distributions such that the variance of the weights decreases exponentially fast in n, and this is not an easy task. Consider the following concrete example of this problem in a very simple model where the proposal distributions are not chosen carefully enough.

Example 3.2.9. Let $\mathcal{X} = \mathbb{R}$ and

$$\begin{aligned}\pi_n(x_{1:n}) &= \prod_{k=1}^{n} \pi_n(x_k) = \prod_{k=1}^{n} \mathcal{N}(x_k; 0, 1),\\ \gamma_n(x_{1:n}) &= \prod_{k=1}^{n} e^{-\frac{x_k^2}{2}},\\ Z_n &= (2\pi)^{n/2}.\end{aligned}$$

and consider the natural proposal distribution

$$q_n(x_{1:n}) = \prod_{k=1}^{n} q_n(x_k) = \prod_{k=1}^{n} \mathcal{N}(x_k; 0, \sigma^2).$$

In this problem we have $\mathrm{Var}_{q_n}[Z_n^{\mathrm{NIS}}] < \infty$ if and only if $\sigma^2 > \frac{1}{2}$, and by proposition 3.2.8

$$\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{NIS}}]}{Z_n^2} = \frac{1}{N}\Big[\Big(\frac{\sigma^4}{2\sigma^2 - 1}\Big)^{n/2} - 1\Big]. \tag{3.5}$$

The coefficient $\sigma^4/(2\sigma^2 - 1)$ is greater than zero for $\sigma > 1/2$ if $\sigma \neq 1$. In this case the variance will grow exponentially. This exponential growth will happen even when the chosen σ is very close to the real sigma 1. This means that, even with proposals q_n that approximate π_n very well, the variance of our estimator will explode exponentially fast.

3.3 Sequential Monte Carlo

Since their introduction in [16], Sequential Monte Carlo methods have been in constant development. There have been constant modifications, variations and improvements on the basic idea [35, 12, 7, 41]. This chapter will introduce the general idea and the main convergence results without proofs, which will be enough for our needs. For a more comprehensive introduction on these methods we refer the reader to [13] or [10].

Sequential Monte Carlo (SMC) methods were introduced to try to mitigate the variance problem present in importance sampling based methods. SMC algorithms are a class of greedy algorithms that try to exploit promising local minima via the process of *resampling*.

The basic idea behind resampling is to duplicate particles that have higher weights w_n and do not keep propagating particles with lower weights. The rationale behind this idea is the same as those in greedy algorithms: Particles with higher weight are potentially more likely to ending up having a higher probability. With this particles we would be exploring parts of the distribution with higher density. Let us formalise this idea.

Consider the empirical distribution given by normalised importance sampling

$$\pi_n^{\mathrm{NIS}}(x_{1:n}) = \sum_{i=1}^{N} W_n^i \delta_{X_{1:n}^i}(x_{1:n}).$$

This distribution is an approximation of π_n via the particles $X_{1:n}^i$ for $i = 1, \ldots, N$. Each of these particles has an associated weight W_n^i. The approximation π_n^{NIS} can be seen as a discrete distribution over the samples $X_{1:n}^i$ and as such, we can sample from it in a natural way. This would be equivalent to approximately sampling from π_n, since π_n^{NIS} is an approximation of π_n. After this, if we denote by N_n^i the number of times that each sample $X_{1:n}^i$ was sampled in this resampling step we can define the distribution defined by the resampled particles as

$$\pi_n^{\mathrm{R}}(x_{1:n}) = \sum_{i=1}^{N} \frac{N_n^i}{N} \delta_{X_{1:n}^i}(x_{1:n}).$$

We have that $\mathrm{E}[N_n^i \,|\, W_n^{1:N}] = NW_n^i$. With this definition, is direct to see that π_n^{R} is an unbiased approximation of π_n^{NIS}.

Finally, we just have to carefully put all these ideas together, we have what is usually regarded as the generic Sequential Monte Carlo algorithm. To do this, we must analise the first two steps of the algorithm. To initialise the algorithm we sample $X_1^i \sim q_1$ and we compute the associated weights W_1^i. Then we do the resampling step to get $\widehat{X}_1^i$. As we observed before, $\widehat{X}_1^i$ are samples of π_1^{NIS}, and as such, since π_1^{NIS} is an approximation of π_1, we can see them as approximately distributed as $\widehat{X}_1^i \sim \pi_1$. This is formalised by the formula

$$\mathrm{E}[N_1^i \,|\, W_1^{1:N}] = NW_1^i.$$

As such, these new particles $\widehat{X}_1^i$ have equal weights $1/N$. After this observation, we can continue normal sequential importance sampling with the particles $\widehat{X}_1^i$. Given

that these particles have equal weights, we have that after the new component of the particles X_2^i is sampled from $q_2(x_2 \mid \widehat{X}_1^i)$, the weights that we have to assign to these new particles $X_{1:2}^i = (\widehat{X}_1^i, X_2^i)$ are exactly the incremental weights $\alpha_2(X_{1:2}^i)$. This gives us the generic SMC algorithm presented in algorithm 1.

Algorithm 1 Generic SMC

procedure SMC($\gamma_1, \ldots, \gamma_T, q_1, \ldots, q_T$)
  for $i = 1, \ldots, N$ **do**
    $X_1^i \sim q_1(x_1)$ ▷ Sample initial particles
    $w_1(X_1^i) \leftarrow \frac{\gamma_1(X_1^i)}{q_1(X_1^i)}$ ▷ Compute unnormalised weights
  for $i = 1, \ldots, N$ **do**
    $W_1^i \leftarrow \frac{w_1(X_1^i)}{\sum_{i=1}^N w_1(X_1^i)}$ ▷ Compute normalised weights
  for $i = 1, \ldots, N$ **do**
    $\widehat{X}_1^i \sim \text{Multinom}(W_1^1, \ldots, W_1^N)$ ▷ Resample
  for $n = 2, \ldots, T$ **do**
    for $i = 1, \ldots, N$ **do**
      $X_n^i \sim q_n(x_n \mid \widehat{X}_{1:n-1}^i)$ ▷ Sample next components
      $X_{1:n}^i \leftarrow (\widehat{X}_{1:n-1}^i, X_n^i)$
      $\alpha_n(X_{1:n}^i) \leftarrow \frac{\gamma_n(X_{1:n}^i)}{\gamma_{n-1}(X_{1:n-1}^i) q_n(X_n^i \mid \widehat{X}_{1:n-1}^i)}$ ▷ Update weights
    for $i = 1, \ldots, N$ **do**
      $W_n^i \leftarrow \frac{\alpha_n(X_{1:n}^i)}{\sum_{i=1}^N \alpha_n(X_{1:n}^i)}$ ▷ Normalise weights
    for $i = 1, \ldots, N$ **do**
      $\widehat{X}_{1:n}^i \sim \text{Multinom}(W_n^1, \ldots, W_n^N)$ ▷ Resample

Remark. There are other resampling systems with lower variance than the one we presented. Examples of these are, for example, *Systematic Resampling* [21] or *Residual Resampling*. For a theoretical analysis of the performance of these and others we refer the reader to [11]

Remark. The resampling step can be beneficial to reduce the variance when most of the particles have low weights and a few have higher weights, since we do not want to spend computational power evaluating particles that seem to not have a high mass in the end. On the other hand, when almost all the particles have comparable weights, the resampling step can be a bad idea, since we are modifying

π_n^{NIS} for π_n^{R} which has a higher variance [6], and we are not getting anything in exchange. For this reason, in practice, the resampling step is only performed when the particles are *dissimilar enough*. This is approximated via the *Effective Sampling Size* estimator (ESS) defined as

$$\mathrm{ESS} = \left(\sum_{i=1}^{N} (W_n^i)^2\right)^{-1}.$$

The idea behind the ESS estimator is that if we have a random variable

$$X = \frac{\sum_{i=1}^{N} w_i Y_i}{\sum_{i=1} w_i}$$

for $w_i \in \mathbb{R}^+$ and where Y_i have the same mean and variance $\sigma^2 > 0$, then if we denote by $\overline{Y} = \frac{1}{N}\sum Y_i$ the empirical mean, we have that $\mathrm{Var}[\overline{Y}] = \sigma^2/\,\mathrm{ESS}$, and solving for ESS we get

$$\mathrm{ESS} = \frac{(\sum_{i=1}^{n} w_i)^2}{\sum_{i=1}^{n} w_i^2} = \left(\sum_{i=1}^{N} (W_n^i)^2\right)^{-1}.$$

In plain words, the effective sample size of a set of samples is the size of an equivalent sample from from the true target distribution.

In practice, the resampling step is just performed when ESS is below certain threshold. A common choice for this threshold is $N/2$.

For the SMC algorithm, when using multinomial resampling, under very mild assumptions, we have the following asymptotic result:

$$N\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{SMC}}]}{Z_n^2} \xrightarrow{d} \int \frac{\pi_1^2(x_1)}{q_1(x_1)}\,\mathrm{d}x_1 - 1 + \sum_{k=2}^{n} \int \frac{\pi_k^2(x_{1:k})}{\pi_{k-1}(x_{1:k-1}) q_k(x_k \mid x_{1:k-1})}\,\mathrm{d}x_{k-1:k} - 1.$$

This result might look intimidating at first, but we invite the reader to observe the similarities of this result with the one in proposition 3.2.8. The following iterative expressions, equivalent to the left-hand side of the previous formula, offer a better insight into this asymptotic variance:

$$V_1 = \frac{\mathrm{Var}_{q_1}[Z_1^{\mathrm{SMC}}]}{Z_1^2} = \frac{\mathrm{Var}_{q_1}[Z_1^{\mathrm{NIS}}]}{Z_1^2}$$
$$V_n = V_{n-1} + \frac{\mathrm{Var}_{q_n}[\alpha_n]}{Z_n^2}.$$

If we observe algorithm 1, this formula show explicitly the idea that SMC *resets* the algorithm after each resampling step. As such, the variance of this estimator depends linearly on the variance of the number of steps. A proof of this result can be found in [29].

Armed with this estimator, we can show how SMC performs in the problematic example shown in section 3.2.4.

Example 3.3.1. If we use SMC to sample from the random variable proposed in the problematic example we get that

$$\begin{aligned}\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{SMC}}]}{Z_n^2} &\approx \frac{1}{N}\Big(\int \frac{\pi_1^2(x_1)}{q_1(x_1)}\,\mathrm{d}x_1 - 1 + \sum_{k=2}^{n}\int \frac{\pi_k^2(x_{1:k})}{q_k(x_k)}\,\mathrm{d}x_{k-1:k} - 1\Big)\\ &= \frac{n}{N}\Big[\Big(\frac{\sigma^4}{2\sigma^2-1}\Big)^{1/2} - 1\Big]\end{aligned}$$

In this case we have that the asymptotic variance is finite just for $\sigma^2 > \frac{1}{2}$, but we can observe that the variance grows linearly instead of exponentially, as it is the case for the variance in equation (3.5).

This example is unusually favourable for SMC, since π_n fully factorises in its coordinates. We can informally see SMC as exploiting the Markovian properties of the model. SMC will perform better when the model does not present strong long-distance dependencies.

In words of N.Chopin, "It may be recommended to use the sequential importance sampling algorithm for studying short series of observations, provided that the dimension of the parameter space is low. But, in general, one should rather implement a more elaborate particle filter which includes mutation steps in order to counter the particle depletion." [6].

3.4 Compiled Inference

All the algorithms presented in this chapter had some hyperparameters, namely the proposal distributions. The convergence results and the variance of the estimators of these algorithms depend heavily on the choice of these proposal distributions. Although we have presented estimators and conditions that these proposals should

have, we have not mentioned techniques to systematically choose them. Compiled Inference aims to attack this problem from a variational perspective.

Variational inference is a field in which some unknown distribution is approximated via variational methods. The name comes from the calculus of variations, where the variable that we want to approximate lives in an infinite-dimensional space, instead of just being a finite-dimensional vector. In most of the cases, these parameters are functions that live in some Banach space.

The application of variational inference for probabilistic programs was first introduced in [44]. This method is just a variation of the derivation of the ELBO estimator, widely used in machine learning literature [19].

In the variational inference approach the hypothesis is that some given observations $y_{1:T}$ are fixed, and we try to approximate the posterior. On the other hand, in the compiled inference framework [25], the optimisation is performed on the expectation of all possible observations. As such, we try to find a family of proposals that works well for every possible observation on average. The name *compiled inference* comes exactly from this idea. We expend some time compiling an artefact—a neural network in this case—so that we can perform fast inference afterwards. In some way, that artefact can be seen as a tailored inference algorithm for the probabilistic model for which it was compiled.

3.4.1 Informal Introduction to Universal Probabilistic Programming

Before introducing the main ideas in compiled inference, we will introduce an abstraction of universal probabilistic programs.

Universal probabilistic programs are a way to define priors and likelihoods using a Turing-complete programming language. Therefore, a probabilistic program can then be seen as a random variable on the space of execution traces and observations.

We say that this is an informal introduction to probabilistic programming since we will not formalise what we mean by Turing-complete language and how this definition is affected by the randomness and the conditioning in the language. A formal introduction to semantics of programs with randomness in them was first introduced in [23]. These semantics only consider a programming language

with a source of randomness, but they do not include conditioning on random variables to be able to perform inference. In recent years, after the blooming of the first Turing-complete programming languages, a substantial effort has been put in formalising the concepts of conditioning in a sensitive manner [40, 39].

Probabilistic programming languages have two extra keywords that allow the user to define prior distributions and likelihoods. We say that the statements that define a prior are `sample` statements and those that define the likelihood are `observe` statements. For now it should be enough to think of `sample` statements as a function that takes a distribution and samples from it, and to think of `observe` statements as a function that, given a distribution and a value in the execution trace, multiplies the probability of the current execution trace by the pdf of the variable distribution. In other words, the `observe` functions serve the purpose of scoring the current trace, this is, of defining how likely is that certain latent states generated some given observed values.

Let us denote the prior density functions as f and the likelihood density functions as g. We will assume that we have a fixed number of observations T. In this settings, given a trace where M `sample` statements were hit, we may assign a probability to that trace as

$$p(x_{1:M}, y_{1:T}) = \prod_{t=1}^{M} f_{a_t}(x_t \mid x_{1:t-1}) \prod_{t=1}^{T} g_t(y_t \mid x_{1:\tau(t)})$$

where a_t are the *addresses* in the code of the different `sample` statements respectively and $\tau(t)$ is the number of `sample` statements that were hit before the t-th `observe` statement was hit. The reason to restrict the number of observations to a fixed number will become apparent when we define the algorithm.

Remark. The labelling of the `sample` statements with addresses is necessary, since it could be the case that some of these statements were just hit conditionally, depending on the result of some prior sampled values. We discuss different addressing schemes and how they modify the semantics of a probabilistic program in section 4.2.1.

Compare this formula with the one for HMM models

$$p(x_{1:T}, y_{1:T}) = p(x_1) \prod_{t=2}^{T} f(x_t \mid x_{t-1}) \prod_{t=1}^{T} g(y_t \mid x_t).$$

Probabilistic programs are a generalisation of this model, where we allow the number of observable to be different to the number of latent variables. We also allow the number of these variables to be a random variable themselves and the latent and observable variables to be dependant not only on the last latent variable generated but on all of the variables generated before them.

In this case, our objective has not changed: We are looking to perform inference. We will centre our efforts on computing the smoothing distribution $p(\mathbf{x} \mid y_{1:T})$ where $\mathbf{x}$ lives in the space of possible execution traces. We will denote the length of a trace as $|\mathbf{x}|$. In what follows we will always assume that programs stop with probability one. Note that this space, for Turing-complete languages, is an infinite dimensional space, although it is locally finite.

3.4.2 Compiled Sequential Importance Sampling

The idea behind every compiled inference algorithm is, as previously mentioned in the introduction, to find a way to approximate proposals that are *good enough*. In order to achieve this, we maximise some divergence between our proposal distributions and the measures that we want to approximate. Then, we use this *metric* as a loss function in a neural network, so that the neural network learns how to generate good proposal distributions.

Consider for now the simplified case where we want to perform inference on certain fixed set of observations $y_{1:T}$. As we said, our target distribution in this case is the smoothing distribution

$$\begin{aligned} \gamma(\mathbf{x}) &= p(\mathbf{x}, y_{1:T}) = \prod_{i=1}^{|\mathbf{x}|} f_{a_i}(x_i \mid x_{1:i-1}) \prod_{t=1}^{T} g_t(y_t \mid x_{1:\tau(t)}) \\ \pi(\mathbf{x}) &= p(\mathbf{x} \mid y_{1:T}) = \frac{\gamma(\mathbf{x})}{Z} \end{aligned} \tag{3.6}$$

where γ denotes the target distribution defined by the trace $\mathbf{x}$ and

$$Z = \int \gamma(\mathbf{x}) \, \mathrm{d}\mathbf{x}.$$

Issue 1. Since we are now working in an infinite-dimensional space, it is not clear if the normalisation constant Z even exists, or whether it could be infinite.

In fact, in general, it does not exist. Not every probabilistic program defines a probability measure. In general, a probabilistic programs defines a—possibly infinite—measure as shown in [39].

A formal treatment of the subject of probabilistic programming should therefore be done in a measure theoretical framework, instead of a probabilistic framework [1].

The second problem that arises is that, in this case, we do not have a sequence of distributions, but a partial order given by the execution traces. Not only that but, since this space is far from being a lattice, we do not have such a thing as a *last* distribution.

One approach to formalise this problem is to define two traces to be from the same measure space if and only if the sequence of `sample` statements that they hit was the same. Then, we can say that the target distribution is a distribution over the infinite direct sum of all the spaces defined by the possible traces. This definition as a tree of traces is very fine—in the topological sense—. A more coarse space could be defined after an identification of traces that were not equal, but that at some point would start following the same path as in [44]. This turns the space into an infinite DAG. The main drawback of this second approach is that there is no easy way to formalise when two traces are exactly the same after differing in some point. If not enough care is taken the identification can be disastrous for the inference, even leading the algorithms to converge to incorrect distributions [20].

Now that we have defined over which measure space are we working, we can start to approximate it.

To approximate equation (3.6), we can proceed as usual. We can simulate from f_{a_i} just by executing the program and we can do so N times to get particles $\mathbf{x}^1, \ldots, \mathbf{x}^N$. With these particles we may approximate γ using normalised importance sampling

$$\gamma(\mathbf{x}) \approx \frac{1}{N} \sum_{i=1}^{N} w(\mathbf{x}^i) \delta_{\mathbf{x}^i}(\mathbf{x})$$

$$Z \approx \frac{1}{N} \sum_{i=1}^{N} w(\mathbf{x}^i)$$

[1] As we mentioned in the introduction, we did not follow a measure-theoretic presentation to make the text more approachable to the reader.

where

$$w(\mathbf{x}^i) = \prod_{t=1}^{T} g_t(y_t \mid x^i_{1:\tau(t)}).$$

Issue 2. This initial derivation is where the first problem arises. As we mentioned at the very beginning, to get this approximation we are relying implicitly on the Monte Carlo integration method. To see this more clearly, suppose that we want to integrate a function $Q(\mathbf{x})$ against $\gamma(\mathbf{x})$. We get the approximation

$$\mu = \int Q(\mathbf{x})\gamma(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^{N} w(\mathbf{x}^i)Q(\mathbf{x}^i).$$

This is nothing but the Monte Carlo estimator for $\mathrm{E}_\gamma[f]$. Now, the Monte Carlo estimator is defined over finite measurable sets (or integrals, for the case) in $\mathbb{R}^n$ or $\mathbb{Z}$. In this case we do not have a finite dimensional space, so it is by no means clear whether the Monte Carlo estimator can be applied here.

Luckily, the space is the disjoint union of copies of $\mathbb{R}^{d_i}$ for different dimensions d_i. Given this premise, we can draw on the treatment of the subject given in the Metropolis-Hastings-Green algorithm, introduced in [17].

Provided that we can apply regular importance sampling in this problem, we might as well try to change the measures f_{a_i} to approximate faster μ, as we have been doing throughout this chapter. We will follow the ideas presented in [25]. As a side-note, we shall mention that the issues raised in this exposition do not only apply to [25] approximation, but to every published approximation of our knowledge where they use this variants of this method in probabilistic programs (*e.g.*, [44, 36]).

As it is customary in machine learning when we want to approximate a distribution with other distribution, we can do so by minimising the KL-divergence between them. In this case, our target distribution is $\pi(\mathbf{x}) = p(\mathbf{x} \mid y_{1:T})$ and suppose that we would like to approximate it with a parametric family of distributions q_ϕ. We will assume that the family q_ϕ is differentiable with respect to the parameter ϕ. If we had just one observation $y_{1:T}$ our objective function would be

$$D_{\mathrm{KL}}(p(\cdot \mid y_{1:T}) \,||\, p_\phi(\cdot \mid y_{1:T}))$$

but we would like our family of distributions to be as generic as possible. As such, a sensible choice would be to try that our algorithm performs well on average for

every observation. In this case, we minimise the expectation its average over all the possible observations

$$\mathcal{L}(\phi) = \mathrm{E}_{p(y_{1:T})}[D_{\mathrm{KL}}(p(\cdot \mid y_{1:T}) \,||\, q_\phi(\cdot \mid y_{1:T}))].$$

In what follows we will suppose that the integrals in $\mathcal{L}(\phi)$ are well defined. Given this premise, we can rewrite $\mathcal{L}(\phi)$ as

$$\begin{aligned}\mathcal{L}(\phi) &= \mathrm{E}_{p(y_{1:T})}[D_{\mathrm{KL}}(p(\cdot \mid y_{1:T}) \,||\, p_\phi(\cdot \mid y_{1:T}))] \\ &= \int_{y_{1:T}} p(y_{1:T}) \int_{\mathbf{x}} p(\mathbf{x} \mid y_{1:T}) \log \frac{p(\mathbf{x} \mid y_{1:T})}{q_\phi(\mathbf{x} \mid y_{1:T})} \,\mathrm{d}\mathbf{x} \,\mathrm{d}y_{1:T} \\ &= \mathrm{E}_{p(\mathbf{x}, y_{1:T})}[-\log q_\phi(\mathbf{x} \mid y_{1:T})] + \mathrm{E}_{p(\mathbf{x}, y_{1:T})}[\log p(\mathbf{x} \mid y_{1:T})].\end{aligned}$$

where we have used Fubini's theorem in the last equality. Note that the second term of this expression does not depend on ϕ and hence it is a constant. If we want to minimise $\mathcal{L}(\phi)$ it is enough to minimise the first term of this expression.

Here is where compiled inference differs from regular variational inference. Variational inference fixes an observation $y_{1:T}$ and given this observation, computes the posterior. Compiled inference is much more ambitious. Compiled inference finds a parametric family q_ϕ such that it performs well for all the observations $y_{1:T}$ on average.

To minimise $\mathcal{L}(\phi)$ we can approximate via sampling from $p(\mathbf{x}, y_{1:T})$ and the usual Monte Carlo estimator

$$\frac{\partial}{\partial \phi}\mathcal{L}(\phi) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial \phi}\Big(-\log q_\phi(\mathbf{x}^i \mid y^i_{1:T})\Big).$$

where $(\mathbf{x}^i, y^i_{1:T}) \sim p(\mathbf{x}, y_{1:T})$. This approach presents one more problem, besides the one remarked in issue 2.

Issue 3. It is not true in general that

$$\frac{\partial}{\partial \phi}\int f(x, \phi) \,\mathrm{d}\mu(x) = \int \frac{\partial}{\partial \phi} f(x, \phi) \,\mathrm{d}\mu(x).$$

where μ a measure. In general the derivative on the left hand side may not not even exist!

Furthermore, it could be the case where the parameter ϕ is infinite-dimensional [2]. This is another issue to address, but one that is easily solved, since although the space is infinite-dimensional, we can see it as a direct sum of finite-dimensional spaces. In this space, a function is nothing but a collection of functions from finite-dimensional spaces, and we can define the derivative formally as the derivative component-wise.

There are sufficient conditions under which this derivative exists for $\phi \in \mathbb{R}^n$. The most celebrated is Leibniz' rule [3].

Sadly, it is very common to interchange the integral and derivative in literature without the proper checks, or at least noting that this step does not hold in general. This happens over and over in almost every paper that derives variations of the ELBO estimator, for variational auto encoders or reinforcement learning. Not even the original paper presenting Variational Auto Encoders mentions this [19].

It must be noted that although the trend is to overlook this problem, there are a few papers that at least mention this issue and others that even attack it [4].

Let us continue with the exposition of the compiled inference framework. Now that we have an approximation of the gradient of $\mathcal{L}$, we would like to perform gradient descent on ϕ to optimise $\mathcal{L}$. This raises our last issue.

Issue 4. It is not clear how to perform gradient descent in general infinite-dimensional Banach spaces. The methods for doing this, as the mirror-descent algorithm, require a more careful treatment of the problem, namely the definition of functionals in the dual space.

Luckily, since the trace-space is locally finite (every element in the space is finite-dimensional), we can define the analogous smoothness and strong convexity inequalities necessary for the standard proofs of convergence for stochastic gradient descent methods. With these tools, the proof can be generalised for the space in which we are working. A proof of these methods for the finite dimensional case that can be generalised as we mentioned can be found in [18].

[2] This will be the case for general probabilistic models written in CPPROB. In CPPROB, the addressing scheme used allows for an unbounded amount of different `sample` addresses, and since we will have one unique for each address, this means that the ϕ lives in an infinite-dimensional space.

[3] See Theorem 24.5 in [3].

Let us leave all these theoretical problems aside for a moment and consider how we would implement this estimator. We will try to separate the implementation details regarding the neural network from the general idea, but it is a good idea to have in mind that, in practice, this optimisation and the proposal for the parameters are implemented via a neural network.

The only detail that changes from the normal simulation with $\gamma(\mathbf{x}) = p(\mathbf{x}, y_{1:T}$ in the state-space model is that, in this case, the observations are not fixed, since we want to perform compiled inference instead of variational inference. As such, when we generate samples from $p(\mathbf{x}, y_{1:T})$ to compute the estimator of the gradient, we have to be able to sample from the distribution $p(\mathbf{x}, \cdot)$ as well.

Recall that a `observe` statement is a function that takes a distribution and a value and scores the trace according to the pdf of that distribution. If we can also sample from that distribution, during training we can consider the `observe(distr, x)` as a sample statement `sample(distr)`, effectively sampling from $p(\mathbf{x}, \cdot)$.

At first, the choice of the KL-divergence as the divergence in the objective function seems fairly arbitrary. Why the KL-divergence instead of other divergence, as it could be the χ^2-divergence or the JS-divergence?

To give a formal answer to these questions, we will first have to analyse the role that the following quantity plays in importance sampling

$$\rho_n = \frac{\mathrm{E}_{q_n}[w_n^2]}{\mathrm{E}_{q_n}[w_n]^2}$$

where w_n are the unnormalised weights in Normalised Importance Sampling.

This estimator is very closely related to an estimator that we already used to compare the performance of different sampling algorithms

$$\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{NIS}}]}{Z_n^2} = \frac{\rho_n - 1}{N}.$$

The following proposition states the importance of ρ_n

Proposition 3.4.1 (Properties of ρ_n). *We have the following inequalities*

$$\sup_{\|f\|_\infty \leq 1} \left| \mathrm{E}_{q_n}[\mu^{\mathrm{NIS}}[f] - \mu[f]] \right| \leq \frac{12}{N} \rho_n$$
$$\sup_{\|f\|_\infty \leq 1} \mathrm{E}_{q_n}[(\mu^{\mathrm{NIS}}[f] - \mu[f])^2] \leq \frac{4}{N} \rho_n.$$

where we have made explicit the dependency of the estimator and μ with respect to the function f.

Proof. We follow the original proof presented in [1].

We will drop the subindex for the proposal distribution q_n and the unnormalised weights w_n for sort.

To make the notation slightly less cumbersome let us first define the Monte Carlo approximation for q

$$q^{MC} = \frac{1}{N}\sum_{i=1}^{N} \delta_{X_{1:n}^i}$$

where $X_{1:n}^i \sim q$. This operator does nothing but evaluating a given function on the random variables $X_{1:n}^i$ and when applied to a test function ϕ is an unbiased estimator of $\mathrm{E}_q[\phi]$.

Let f be a function such that $\|f\|_\infty \leq 1$.

With this notation, we can rewrite the bias as

$$\mu^{\mathrm{NIS}}[f] - \mu[f] = \frac{q^{\mathrm{MC}}[wf]}{q^{\mathrm{MC}}[w]} - \mu[f] = \frac{q^{\mathrm{MC}}[(f - \mu[f]]w)}{q^{\mathrm{MC}}[w]}$$

Let $\overline{f} = f - \mu[f]$ be the unbiased version of f with respect to π. Since w exactly accounts for the change of measure between π and q_n we have that

$$\mathrm{E}_q[\overline{f}w] = 0$$

and we can write

$$\mu^{\mathrm{NIS}}[f] - \mu[f] = \frac{q^{\mathrm{MC}}[\overline{f}w] - \mathrm{E}_q[\overline{f}w]}{q^{\mathrm{MC}}[w]}.$$

Taking expectations on both sides and since the numerator has expectation zero we have that

$$\begin{aligned}\mathrm{E}_q[\mu^{\mathrm{NIS}}[f] - \mu[f]] &= \mathrm{E}_q\Big[\Big(\frac{1}{q^{\mathrm{MC}}[w]} - \frac{1}{\mathrm{E}_q[w]}\Big)\Big(q^{\mathrm{MC}}[\overline{f}w] - \mathrm{E}_q[\overline{f}w]\Big)\Big] \\ &= \mathrm{E}_q\Big[\frac{1}{q^{\mathrm{MC}}[w]\,\mathrm{E}_q[w]}\Big(\mathrm{E}_q[w] - q^{\mathrm{MC}}[w]\Big)\Big(q^{\mathrm{MC}}[\overline{f}w] - \mathrm{E}_q[\overline{f}w]\Big)\Big].\end{aligned}$$

We are going to bound the bias term in two steps

$$\begin{aligned}\Big|\mathrm{E}_q[\mu^{\mathrm{NIS}}[f] - \mu[f]]\Big| \leq &\Big|\mathrm{E}_q[(\mu^{\mathrm{NIS}}[f] - \mu[f])\mathbb{1}(2q^{\mathrm{MC}}[w] > \mathrm{E}_q[w])]\Big| + \\ &\Big|\mathrm{E}_q[(\mu^{\mathrm{NIS}}[f] - \mu[f])\mathbb{1}(2q^{\mathrm{MC}}[w] \leq \mathrm{E}_q[w])]\Big|\end{aligned}$$

For the first term we have that

$$\begin{aligned}
&\left|\mathrm{E}_q[(\mu^{\mathrm{NIS}}[f]-\mu[f])\mathbb{1}(2q^{\mathrm{MC}}[w]>\mathrm{E}_q[w])]\right| \\
&\qquad\leq \frac{2}{\mathrm{E}_q[w]^2}\,\mathrm{E}_q\Big[|\mathrm{E}_q[w]-q^{\mathrm{MC}}[w]||q^{\mathrm{MC}}[\overline{f}w]-\mathrm{E}_q[\overline{f}w]|\Big] \\
&\qquad\leq \frac{2}{\mathrm{E}_q[w]^2}\,\mathrm{E}_q[(\mathrm{E}_q[w]-q^{\mathrm{MC}}[w])^2]^{1/2}\,\mathrm{E}_q[(q^{\mathrm{MC}}[\overline{f}w]-\mathrm{E}_q[\overline{f}w])^2]^{1/2} \\
&\qquad= \frac{2}{\mathrm{E}_q[w]^2}\,\mathrm{Var}_q[q^{\mathrm{MC}}[w]]^{1/2}\,\mathrm{Var}_q[q^{\mathrm{MC}}[\overline{f}w]]^{1/2} \\
&\qquad\leq \frac{2}{\mathrm{E}_q[w]^2}\,\frac{1}{\sqrt{N}}\,\mathrm{E}_q[w^2]^{1/2}\,\frac{2}{\sqrt{N}}\,\mathrm{E}_q[w^2]^{1/2} \\
&\qquad= \frac{4}{N}\rho_n
\end{aligned}$$

where in the second inequality have used Hölder's inequality and in the last inequality we have used that $\overline{f}\leq 2$ since $\|f\|_\infty\leq 1$.

Now we bound the second term

$$\begin{aligned}
\left|\mathrm{E}_q[(\mu^{\mathrm{NIS}}[f]-\mu[f])\mathbb{1}(2q^{\mathrm{MC}}[w]\leq\mathrm{E}_q[w])]\right| &\leq 2\Pr_q(2q^{\mathrm{MC}}[w]\leq\mathrm{E}_q[w]) \\
&\leq 2\Pr_q(2|q^{\mathrm{MC}}[w]-\mathrm{E}_q[w]|\geq\mathrm{E}_q[w]) \\
&\leq \frac{8}{N}\rho_n
\end{aligned}$$

where in the first inequality we have used again that $\|f\|_\infty\leq 1$ and in the last one we have used Chebyshev's inequality.

This completes the proof of the first bound. The proof of the second bound is very similar. We refer the reader to [1] for the details. □

This proposition shows how the quantity ρ_n governs the behaviour of both the mean and the mean squared error of the estimator in importance sampling methods. This justifies once again why we would like to keep the quotient

$$\frac{\mathrm{Var}_{q_n}[Z_n^{\mathrm{NIS}}]}{Z_n^2}$$

low.

The proposition also explains even more why we would like to maximise ESS, since

$$\mathrm{ESS}=\left(\frac{q^{\mathrm{MC}}[w^2]}{q^{\mathrm{MC}}[w]^2}\right)^{-1}.$$

Hence, ESS is an estimator of the inverse of ρ.

Finally, the following proposition shows why is it desirable to keep the KL-divergence between π_n and q_n low, therefore giving a theoretical ground for the compiled inference objective used for importance sampling.

Proposition 3.4.2. *The following inequality holds*

$$\rho_n \geq e^{D_{KL}(\pi_n \,||\, q_n)}$$

Proof. It follows from Jensen's inequality

$$D_{\mathrm{KL}}(\pi_n \,||\, q_n) = \mathrm{E}_{\pi_n}\left[\log \frac{\pi_n}{q_n}\right] \leq \log \mathrm{E}_{\pi_n}\left[\frac{\pi_n}{q_n}\right] = \log \rho_n.$$

□

Remark. The previous lemma shows that it is desirable to keep the KL-divergence low so that ρ_n is not large and proposition 3.4.1 is useful. We leave as an open problem for future research the analysis of sufficient conditions that link the KL-divergence with the convergence of importance sampling.

Chapter 4

CPProb

4.1 Design Outline

In this section we introduce CPProb, a probabilistic programming library written in C++14 that allows to perform inference in probabilistic models written in C++.

4.1.1 Current Approach to Probabilistic Programming

Current probabilistic programming languages make use of code preprocessing and code transformations to implement inference algorithms such as SMC and variations of it. This approach has two main drawbacks.

The first one is that the implementation of source-to-source transformations is language-specific, and therefore, it is not generalisable to other programming languages. Some probabilistic programming languages are designed on top of languages with certain characteristics that facilitate the implementation of intricate techniques, or provide certain guarantees. For example, the probabilistic programming language Anglican [45] is designed on top of Clojure since every data structure in this language is immutable. Anglican uses this property to implement certain preprocessing in the code to simulate, through the use of CPS guards and other techniques, a coroutine system on the whole model, which allows the inference algorithm to copy and resume execution traces in different points of the program at any time. These ideas do not scale to languages with explicit memory management, like C++. To see this, consider the following piece of code:

```
void add_random_number_to_db() {
    DataBase* db = get_database();

    auto x = sample(normal_distribution(0, 1));

    db->add_entry(x);
    db->close();
    std::free(db);
}
```

In the `sample` statement, we would like to duplicate the execution case during the resampling step in SMC-like algorithms. If we duplicate the execution trace and we continue executing one of the traces until we hit the `std::free` statement, then the pointer `db` will be released. If at some point we resume the second execution trace, when the `std::free` is hit for the second time on the same pointer, we would be freeing a memory address that has been already deallocated which would, in the best of cases, make the program crash. This is an example of a side-effect. Side-effects are the reason why there are no implementations of copyable coroutines in any major non-functional language.

The second problem is that almost every probabilistic programming language defines a framework in which models have to be specified. Some of these languages, *e.g.*, Prism [24] or IBAL [33], just allow the user to use discrete random variables. Others like BUGS [38] or INFER.NET [28] only allow the user to define graphical models, which are much less expressive—no unbounded constructions are allowed—. Most of the new probabilistic languages such as Anglican [45], Venture [26] or Church [15] allow the programmer to define arbitrary probabilistic programs. This suffices when models are *small enough* to be written in high-level programming languages. When models are computationally expensive, as it is the case for many simulators, these languages do not offer the tools to write high-performance, low-level code. Furthermore, in many cases the model consists of hundreds of thousands of lines of code and, is the result of the iteration of a research group over several decades. In these cases, it is not feasible to translate the model to a different programming language to perform inference in it.

4.1.2 Design Goals

We aim to adopt a more flexible approach to probabilistic programming than any other software currently available. Our objective is that the user should be able to select their programming language of choice to write probabilistic models. Performing inference should be as easy as using the `sample` function supplied by the library to generate the randomness in the program, while using the `observe` function to condition on the outputs.

Given that many of the pre-existing simulators are written in C or C++, we chose C++14 as the programming language for the library. However, our approach is flexible enough so that the framework is language-agnostic. More on this in section 7.1.

Through the use of *template meta-programming*, we were able to develop an extensible high-performance inference library focused on zero-overhead abstractions. The library is almost header only and it is type agnostic. This allows the user to choose the precision of the numbers that the library generates, among other things. All the code necessary to do this is generated at compile time.

The library defines a set of implicit interfaces that make it even more flexible. For example, although the library already provides several distributions, the user may provide their own distribution or even their own implementation. There are interfaces for the arithmetic types that the library internally uses as well. All the interfaces are resolved at compile time. This constitutes an example of zero-overhead abstraction.

4.1.3 Syntax

CPProb exports three main functions: `sample`, `observe` and `predict`.

`cpprob::sample` [1] has the following signature.

```
template<template <class ...> class Distr, class ...Params>
typename Distr<Params ...>::result_type
    sample (Distr<Params ...> & distr, bool control = false);
```

Remark (A note on the C++ syntaxis). The ellipsis present in the template parameters are what is called in C++ *variadic templates*. This allows us to accept

[1] All the library is within the namespace `cpprob`.

as a distribution any template class that is parametrised with any number of parameters. The second template parameter is another variadic template parameter, and it corresponds to the template parameters of the first argument. For example, when instantiated with `boost::random::discrete_distribution<int, double>` [2] it accepts two template parameters, the method `sample` will be instantiated as follows

```
int sample<boost::random::discrete_distribution, int, double>(
    boost::random::discrete_distribution<int, double> & distr,
    bool control = false);
```

The `sample` function would be the equivalent of putting a prior on a variable $x \sim p(\theta)$, where $p(\theta) \sim$ `distr`. The parameter θ is implicitly provided in the constructor of `distr`.

It accepts a distribution by reference [3] and a boolean flag.

The distribution is the object that implements the sampling algorithm. We will describe these objects in greater detail in section 4.2.8.

The boolean flag `control` selects whether the parameters for the proposal distribution should be provided by the neural network (controlled), or whether the prior should be used as a proposal (not controlled). Not controlling some random variables is sometimes useful to reduce the training time of the neural network, and in many cases, improve its generalisation. Random variables used to model noise are the most typical example of variables that the designer of the model might choose not to control. Examples of these variables can be the random draws used to generate the noise in a captcha generator model, or the ones that control the diffusion of particles after interacting with the detectors in a particle detector simulator. The variable defaults to `false` so that if a user changes every random draw in their program from a certain random number generator library to a call

[2] The first parameter of this distribution is the integer type used to represent the result and the second parameter is the float type used to represent the probabilities.

[3] The distribution parameter is non-constant since the `operator()` present in the STL and Boost distributions is non-constant. This is useful to efficiently implement sampling algorithms as the Box-Muller transform, where two normal-distributed random numbers can be generated at the same time. This is the implementation present in GLIBCXX 6.3.0. Interestingly enough, Boost 1.64.0 implementation of the normal distribution sampling algorithm does not make use of this transformation, since it implements the *Ziggurat algorithm*, but it still does not make the `operator() const`.

to `cpprob::sample` without any extra parameter, the semantics of the program remain unchanged.

The `cpprob::observe` function has signature

```
template<template <class ...> class Distr, class ...Params>
void observe(Distr<Params ...> & distr,
             const typename Distr<Params ...>::result_type & theta);
```

In this case, the function represents setting the likelihood of the model as $p(x \mid \theta)$, where $p(\cdot \mid \theta) \sim$ `distr`.

Finally, we have `cpprob::predict`, with signature

```
template<class T>
void predict(const T & x, const std::string & addr = "");
```

This function selects the random variables that are to be inferred. In a probabilistic program, every random draw and every variable that depends on a random draw is a random variable. This allows the model designer to choose the random variables they want to infer.

The first argument is the variable to be inferred. The only requirement for this variable is to implement the `operator<<` for `std::ostream`. If the variable is also an integer or a real number, then the library exports helper functions that compute a set of common estimators of the empirical posterior distribution.

The second argument allows the user to specify an identifier for the random variable. If an identifier is not provided, the library will automatically assign a unique identifier to each `predict` statement in the code. We will expand on this in section 4.2.1.

In listing 4.1 we show a basic CPProb model that was already presented in section 2.3.2.

4.2 Design Details

4.2.1 Addressing Scheme

Introduction

The addressing scheme is the internal system that the library uses to distinguish different random variables.

```
void flip(const bool y)
{
    using namespace boost::random;
    uniform_01<double> prior;
    const double theta = cpprob::sample(prior, true);

    bernoulli_distribution<double> likelihood {theta};
    cpprob::observe(likelihood, y);
    cpprob::predict(theta, "Mu");
}
```

Listing 4.1: Coin flip model with one observation.

Consider the code in listing 4.2.

```
auto x = cpprob::sample(normal_distribution<>{0, 1});
auto y = cpprob::sample(normal_distribution<>{1, 2});
```

Listing 4.2: Different types of random variables.

The random draws `x` and `y` represent different random variables.

Given this situation, one may think that representing each random variable by an unique identifier given by its file and the position in the file would be enough, but it is not. Consider the simplified realistic situation presented in listing 4.3. This piece of code describes the basic machinery necessary to load a custom random number generator that implements a certain interface defined by `CustomRNG` and expose it as a singleton object. A variation of this idea is used in SHERPA to allow custom random number generators.

In this case, every call to `RNG::sample_normal` will be assigned the same address, so the inference engine would have no way to know if the call from within `foo` was the one in the first branch or the one in the second.

A more drastic example would be a program where all the randomness is generated from calling one statement with a `std::random_device`, which then transforms the output to different random variables. In this case, if our addressing scheme was the line of the code in which the `cpprob::sample` statement is called, we would have effectively just one address per program execution called many times. This is not enough information to perform inference.

```cpp
class CustomRNG;  // Interface for a Custom Random Generator

class RNG {
public:
    double sample_normal(
        const double mu = 0,
        const double sigma = 1)
    {
        if (rng_) {
            return rng_->sample_normal(mu, sigma);
        }
        else {
            std::normal_distribution<double> normal {mu, sigma};
            return normal(std::random_device{});
        }
    }

    static RNG& get(std::unique_ptr<CustomRNG> && custom_rng =
       nullptr) {
        static RNG rng {std::move(custom_rng)};
        return rng;
    }


private:
    RNG(std::unique_ptr<CustomRNG>&& rng)
        : rng_{std::move(rng)} {}

    const std::unique_ptr<CustomRNG> rng_;
};

void foo(int n) {
    auto rng = RNG::get();
    if (n > 0) {
        return rng.sample_normal();
    }
    else {
        return rng.sample_normal();
    }
}
```

Listing 4.3: Custom RNG singleton

CPProb addressing scheme

The addressing scheme used in CPProb solves these problems although, as we will point out later, there is no addressing scheme that is strictly better than the others.

CPProb addressing scheme is based on the Linux tools `backtrace (3)` and `backtrace_symbols`. When executed, `backtrace` returns a list of currently active function calls. Each element in this list is a `void*` with the return address from the corresponding stack frame. The function `backtrace_symbols` translates the addresses into the mangled name of the function that the pointer refers to and the offset of the function call with respect to the function frame. The names of the functions are then demangled for readability. The final address is given by the concatenation of all the demangled function names and their respective offsets.

This addressing system is not only able to differentiate between different calls to a common random number generator, but is also totally code-preserving. Therefore, no modifications to the source code are needed, and the user does not need to provide explicit addresses, which is highly error-prone in large enough projects.

In listing 4.6, we show an address of the `cpprob::sample` statement in model listing 4.4. In listing 4.5, we show the output of a call to `backtrace_symbols`. We included the demangled version of the functions for comparison.

Given this addressing scheme, we may observe that the addresses are not only platform dependent, but also compiler dependent. They are even dependent on the flags with which the project was compiled, since the offsets of the addresses may vary with different optimisations and even the compiler might remove some functions entirely to inline them. This volatility is the reason why we impose that the model has to be in the namespace `models`, so we can use it as a sentinel to detect when the entry call to the model starts. Again, other methods, like using the name of the model as sentinel, fail, since the compiler might elide the function call.

Other addressing schemes

More fine-grained addressing schemes have been proposed in literature [43], where they assign addresses manually via code transformations. The two differences between the approach presented in [43] and the one presented in this dissertation is

```
namespace models {

template<class RealType>
std::vector<RealType> generate_polynomial(std::size_t degree) {
    boost::random::normal_distribution<RealType> prior{0, 10};
    std::vector<RealType> ret(degree + 1);
    for (auto& coef : ret) {
        coef = cpprob::sample(prior, true);
    }
    return ret;
}

template<class RealType>
RealType eval_poly(const std::vector<RealType> &poly, RealType
    point) {
    // Horner's algorithm
    return std::accumulate(poly.crbegin(), poly.crend(), 0.0,
                           [point](RealType acc, RealType next) {
                               return acc * point + next; });
}

template<std::size_t Degree, std::size_t N, class RealType =
    double>
void poly_adjustment(const td::array<std::array<RealType, 2>, N> &
    points) {
    auto poly = generate_polynomial<RealType>(Degree);

    for (const auto &point : points) {
        boost::random::normal_distribution<RealType> likelihood{
            eval_poly(poly, point[0]), 1};
        cpprob::observe(likelihood, point[1]);
    }
    for (const auto coef : poly) {
        cpprob::predict(coef, "Coefficient");
    }
}

} // end namespace models
```

Listing 4.4: Example Model With Several Functions

```
// MANGLED
./src/cpprob/libcpprob.so(_ZN6cpprob8get_addrB5cxx11Ev+0x3d) [0
    x7fa693aa916c]
// DEMANGLED
cpprob::get_addr[abi:cxx11]()+0x3d
// MANGLED
./src/models_main(
    _ZN6cpprob11sample_implIN5boost6random19normal_distribution
     EJdEEENT_IJDpT0_EE11result_typeERS7_b+0x45) [0x4a3d1f]
// DEMANGLED
boost::random::normal_distribution<double>::result_type cpprob::
    sample_impl<boost::random::normal_distribution, double>(boost::
    random::normal_distribution<double>&, bool)+0x45
// MANGLED
./src/models_main(
    _ZN6cpprob6sampleIN5boost6random19normal_distribution
     EJdEEENT_IJDpT0_EE11result_typeERS7_b+0x6f) [0x49bfc7]
// DEMANGLED
boost::random::normal_distribution<double>::result_type cpprob::
    sample<boost::random::normal_distribution, double>(boost::
    random::normal_distribution<double>&, bool)+0x6f
// MANGLED
./src/models_main(
    _ZN6models19generate_polynomialIdEESt6vectorIT_SaIS2_EEm+0xe0)
    [0x49dc16]
// DEMANGLED
std::vector<double, std::allocator<double> > models::
    generate_polynomial<double>(unsigned long)+0xe0
// MANGLED
./src/models_main(
    _ZN6models15poly_adjustmentILm1ELm6EdEEvRKSt5array
     IS1_IT1_Lm2EEXT0_EE+0x33) [0x49632c]
// DEMANGLED
void models::poly_adjustment<1ul, 6ul, double>(std::array<std::
    array<double, 2ul>, 6ul> const&)+0x33
// MANGLED
./src/models_main(main+0xe0d) [0x48a973]
// MANGLED
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0) [0
    x7fa6924c2830]
// MANGLED
./src/models_main(_start+0x29) [0x489a99]
```

Listing 4.5: Output of `backtrace_symbols` for the sample statement in `generate_polynomial` compiled with -O0. When possible, we also show the demangled version of the function call. Calls to some auxiliary functions were removed for readability.

```
[void models::poly_adjustment<1ul, 6ul, double>(std::array<std::
    array<double, 2ul>, 6ul> const&)+0x33
std::vector<double, std::allocator<double> > models::
    generate_polynomial<double>(unsigned long)+0xe0]
```

Listing 4.6: Final address of the sample statement in `generate_polynomial` compiled with -O0.

that they also assign different addresses to variables within the same loop, and most, they do an identification of the addresses in different branches of an if-then-else statement.

This approach does not fit the requirements of CPProb for two reasons. First, it is extremely difficult to do code-to-code transformations that preserve the semantics in C++, by the inherent complexity of the language, even with modern tools such as the clang suite. Second, this approach is not necessarily better than our approach in many cases. For example, when we encounter some accept-reject sampling, we do not want to assign a new address every time that the accept-reject loop is executed, since that would increase the number of addresses needlessly. We will expand on this in section 4.2.5. Finally, the identification step that they took can lead to incorrect inference results when using some MCMC algorithms, as noted in [20].

4.2.2 Predict Statement

The function `predict` is not standard in the probabilistic programming community, although we are not the first to introduce it [31]. In the standard interpretation of probabilistic programs, the predicted variables are returned by the main function in the model. This allows to have a fixed concrete type for every probabilistic program, which is good for formalising the languages and is convenient when the programs are functional.

This approach is not practical in imperative stateful languages. Consider a situation in which we want to perform inference in a program with a code-base of 30.000 lines of code and we want to predict a variable that is hidden within some recondite function of the program. If we were to return that variable, we would have either to modify the whole execution of the program to return it from the main function, which is not possible in large projects, or use static variables, which

is not recommended for many reasons either. Not only that, but we would have to do this for *every variable that we want to predict*. Adding a new `predict` primitive solves all these problems in a simple and flexible manner.

Now, what is it exactly that `predict` represents?

We mentioned in several occasions the dimensionality problems that cause performing inference in Turing-complete probabilistic programming languages in section 3.4. Predict is an statement that allows the user to map this high-dimensional space to a lower-dimensional space.

The measure space that we presented was the disjoint union of possible execution traces. This space can be seen intuitively as the leaves of a tree with unbounded depth.

This approximation is, in general, too fine—in the topological sense—. To see this, consider the rejection sampling scheme presented in algorithm 2. In rejection sampling, we sample from certain distributions until a sampled value is accepted. In a program that uses as a subroutine rejection sampling to sample from a complex random variable we would say that two traces belong to the same measure space if and only if the number of times that a sample was rejected was the same.

For this reason, although we are going to have to work with this big measure space regardless, we will have to map it to a lower-dimensional space to be able to work with it. We achieve this using the function `predict`. The `predict` function *selects* any variable at any point of the execution of the program. This variable will be added to the list of variables on which we are interested. It could be the case that the number of `predict` statements varies from execution to execution, and that the types of the variables passed to the `predict` statement varies as well. This would mean that the `predict` function is mapping a trace to a sum space again. This is, in general, not desirable, and it is a sign of poor model design, but we do not disallow it.

After this analysis, we find that `predict` is some proxy function to facilitate the definition of the final expectation to be computed. Mathematically

$$\mu = \mathrm{E}_{\pi}[f(\texttt{predict}(\mathbf{x}))].$$

In CPProb the only requirement that the type of a variable has to fulfil to be predicted is to be printable, this is, to implement the `operator<<` for `std::ostream`.

4.2.3 Models

Probabilistic models are the fundamental structure in probabilistic programming. A model, defines a joint distribution on the variables that we want to predict and the observations.

Definition 4.2.1 (CPProb models). In CPProb a model is any callable object [4] defined in the namespace `models` that accepts the observations as parameters and returns `void` [5].

Models in CPProb have some restrictions. The first one is that the callable object shall not be a lambda function. This follows from the fact that lambda functions are anonymous functions, and as such, are not defined within a namespace.

CPProb also puts some restrictions on the parameters of the observes. For now, CPProb only accepts base types, iterable types, tuple-like types and combinations of these. This restriction is general enough to include all the types generated by the collections in the standard library.

This restriction is caused by the current dependency of CPProb on PyProb. To be able to perform compiled inference, during inference, we have to serialise the observations and send them to PyProb. In future versions we plan to define a customisation point here, so that the user can implement her own overload of the serialising function for her specific classes.

The last restriction is that the parameters of the model have to be constructible from an rvalue [6]. This rules out non-const reference types. This should not be a real restriction for any sensible model, since observations should be, by default, constant.

Remark. There are more subtle requirements that the types should implement, as having a default constructor. On the other hand, these kind of requirements

[4] The variety of callable objects in C++ is remarkable. The element `f` in the expression `f(x);` can refer to a function (or a function pointer or a function reference), an object overloading `operator()`, an object implicitly convertible to any of the aforementioned elements, an overloaded function name, a lambda, the name of one or more templates, a macro, or even several of the already mentioned elements.

[5] CPProb does not disallow return values other than `void` in its models, but in that case the return value will be discarded whenever the model is executed.

[6] `http://en.cppreference.com/w/c/language/value_category`

are fulfilled by almost every type. Note that even low-requirement implementation of classes also have these requirements. An example of low-requirement implementations are the ones present in the standard library containers.

In section 4.2.6 we present a generalisation of the standard CPProb models, although these do not impose any extra restrictions.

4.2.4 Proposal Distributions

In all the algorithms of the sequential Monte Carlo family the proposal distributions are left to the choice of the practitioner.

As we mentioned the choice of a correct proposal distribution is far from trivial. Furthermore, its choice highly influences the bias and variance of the estimator.

Compiled sequential importance sampling offers a framework in which it is only necessary to specify the family of the proposal distributions, and a neural network will optimise the parameters for the specific model.

As it is customary in variational inference problems, we use the mean-field variational approximation. This approximation corresponds to factorising the family of distributions for

$$p(\mathbf{x}, y_{1:T}) = \prod_{t=1}^{|\mathbf{x}|} f_{a_t}(x_t \mid x_{1:t-1}) \prod_{t=1}^{T} g_t(y_t \mid x_{1:\tau(t)})$$

as

$$q_\phi(\mathbf{x}) = \prod_{t=1}^{|\mathbf{x}|} q_{a_t,\,\phi_{a_t}}(x_t \mid x_{1:t-1}).$$

In other words, we have a family of proposals distributions for each type of prior. Since C++ is statically typed, the type of distribution for each address is fixed at compile time and, therefore we just need to have a different proposal family for each address in the code. Note that since there might be an unbounded number of addresses in the code the space of parameters in CPProb is still infinite-dimensional.

The current families of pairs prior-proposals used in CPProb and PyProb are presented in table 4.1.

Remark. We are well aware that the normal distribution and the multivariate normal are not good proposal distributions given that they have light tails. The

Domain	Prior	Proposal
$\{n, \ldots, m\}$	Bernoulli Uniform Discrete Discrete	Bernoulli Discrete Discrete
$\mathbb{N}$	Poisson	Poisson
$[a, b]$	Beta Uniform Continuous	Beta Beta
$\mathbb{R}^+$	Gamma	Gamma
$\mathbb{R}$	Normal Laplace	Normal Laplace
$\mathbb{R}^n$	Multivariate Normal	Multivariate Normal
$\mathbb{S}^n$	Von Mises-Fisher	Von Mises-Fisher

Table 4.1: Families of proposal distributions

current implementation of the protocol between PyProb and CPProb hardcodes each prior with each proposal, so at the present moment, it is not in our hands to change the proposals for each distribution, until the pull request to change the protocol is accepted by PyProb.

We propose a better approximation to the proposal distribution problem in section 7.2.

4.2.5 Rejection Sampling

Rejection sampling or *accept - reject sampling* is a method to sample from a random variable π from which we just know its pdf function f point-wise.

Suppose that we have access to a proposal function $g(x)$ with

$$\mathcal{D} := \text{supp}(f) = \text{supp}(g)$$

from which we can sample and evaluate its pdf function pointwise. Suppose further, that there exists some constant $M < \infty$ such that $Mg(x) \geq f(x)$ for all $x \in \mathcal{D}$, then we can simulate π using algorithm 2.

Let us show that the algorithm is indeed correct.

Algorithm 2 Rejection Sampling

```
1: procedure rejection_sampling(f, g, M)
2:     while true do
3:         x ← sample(g)
4:         accept ← f(x) / (M g(x))
5:         if sample(U(0, 1)) < accept then
6:             return x                                  ▷ Accept x
```

Proposition 4.2.2. *Rejection sampling algorithm 2 returns a sample distributed as π.*

Proof. Let $A \subseteq \mathcal{D}$ be a measurable set (with respect to the dominating measure dx). We have that

$$\begin{aligned}\Pr(x \in A,\, x \text{ accepted}) &= \int_A \int_0^1 \mathbb{1}\Big(y \leq \frac{f(x)}{Mg(x)}\Big) g(x)\,\mathrm{d}y\,\mathrm{d}x \\ &= \int_A \frac{f(x)}{Mg(x)} g(x)\,\mathrm{d}x \\ &= \frac{\pi(A)}{M}\end{aligned}$$

which implies

$$\Pr(x \text{ accepted}) = \Pr(x \in \mathcal{D},\, x \text{ accepted}) = \frac{\pi(\mathcal{D})}{M} = \frac{1}{M}.$$

Hence

$$\Pr(x \in A \mid x \text{ accepted}) = \frac{\Pr(x \in A,\ x \text{ accepted})}{\Pr(x \text{ accepted})} = \pi(A)$$

□

If we want to control any of the two random draws in a rejection sampling algorithm, the algorithm may produce traces of unbounded length. This presents a problem for the Compiled Inference scheme, where the neural network that generates the parameters for the proposal distribution in the Importance Sampling algorithm has to back-propagate through the full length of the execution traces. Furthermore, all the rejected samples generated within the loop are failed tries of sampling from the random variable. These examples could be seen as pure noise to the neural network.

Given an observation, we would prefer that when the execution trace hits the accept - reject sampling code, the proposed parameters for the proposal distribution make the rejection loop exit as soon as possible. If we give the neural networks the full traces we would be effectively *teaching* the neural network to reproduce this erratic behaviour; *i.e.*, propose parameters so that the program stays in the while loop for several iterations, and just then propose parameters to correctly exit the loop.

In order to solve this problem, we present a two-fold approach. Firstly, we export two functions:

```
void cpprob::start_rejection_sampling();
void cpprob::finish_rejection_sampling();
```

The designer of the model might use these to delimit the parts of the code where rejection sampling is performed.

Suppose that during an execution of a rejection sampling code, k different addresses were hit.

If the program is in training mode, then the trace that is later sent to the neural network will just store the last sample statement of each of the k sample statements. The order in which these k samples are appended to the trace is exactly the order in which the addresses were hit for the last time.

This formalises the rationale given before. We just store in the trace the sampled values that made the program exit the `while` loop. In other words, this is an approximation to the idea of just controlling the `sample` statements that make the loop finish.

Remark. In general, it will not store just the addresses hit during the last execution of the loop. For example, if there are samples in both branches of an `if else` statement within the `while` loop, and if both branches were hit during the execution of the `while` loop, then the trace will have the sampled values of both branches of the `if else` statement, although just one of the addresses was hit during the last execution of the loop. An example of this behaviour can be seen algorithm 3. In the code there are 4 `sample` addresses, but during each execution of the `while` loop, only three `sample` statements will be hit. In this case, if we consider the following

sequence of addresses during an execution

`<start loop>` 1 2 4 1 3 4 1 2 4 1 2 4 `<exit loop>`

then the trace that we will store will be [3 1 2 4] even though the last execution of the loop was [1 2 4].

Algorithm 3 Rejection sampling on a direct sum space

```
1: procedure sum_rejection_sampling(f, g_1, g_2, M)
2:     while true do
3:         if sample(U(0,1)) < 0.5 then                    ▷ Addr 1
4:             x ← sample(g_1)                             ▷ Addr 2
5:         else
6:             x ← sample(g_2)                             ▷ Addr 3
7:         accept ← f(x) / (M g_1⊕g_2(x))
8:         if sample(U(0,1)) < accept then                 ▷ Addr 4
9:             return x
```

During inference mode, we just ask the neural net for parameters for an address a_i the first time that this address is hit. After that, these values are cached and reused as many times as necessary until the next `finish_rejection_sampling()` is hit.

The rationale behind this behaviour is that, during training, the neural network receives only one sample from each address within the loop. This means that the neural network is not trained to process many sample statements that form part of one rejection sampling. If the inference algorithm asked for proposals for every sample statement in the `while` loop until the program exits the loop, this would corrupt the internal state of the LSTM.

The current implementation is based on three assumptions.

1. Every `start_rejection_sampling()` has to be eventually followed by a `finish_rejection_sampling()` matching one to one.

   Furthermore, no `finish_rejection_sampling()` shall be called before invoking `start_rejection_sampling()`.

2. Nested rejection sampling is not allowed.

```
void linear_regression(
        const std::vector<std::pair<double, double>>& observes) {
    normal_distribution<double> prior {0, 10};
    double slope = cpprob::sample(prior);
    double bias = cpprob::sample(prior);
    for (const auto & point : observes) {
        normal_distribution<double>
            likelihood {slope * point.first + bias, 1};
        cpprob::observe(likelihood, point.second);
    }
    cpprob::predict(slope);
    cpprob::predict(bias);
}
```

Listing 4.7: Bayesian Linear regression with Gaussian noise.

3. No `observe` may be called within a rejection sampling.

The implementation could be more general to cover cases 2 and 3 at a slightly higher runtime cost. We did not implement this extension because the use cases for these are too rare to justify the runtime performance penalty.

4.2.6 Free Variables

Some models do not follow exactly the Bayesian models that we presented, where the observed variables are simply observed, and the variables generated from the priors define a distribution over the observes.

Consider for example the Bayesian linear regression with Gaussian noise presented in listing 4.7. The mathematical model that this function represents is

$$a, b \sim \mathcal{N}(0, 10) \qquad y_i \sim \mathcal{N}(ax_i + b, 1) \quad \text{for } i = 1, \ldots, N.$$

As we can see, if the variable x_i is not specified, we do not have a proper distribution on the space of the observed values—$\mathbb{R}^2$ in this case—since the variable x_i does not have any kind of prior or likelihood. It is only used to define the likelihood. This is what we call a *free variable of the model*. Formally, a free variable of a model is any input variable of the model that is used in any place that is not an `observe` statement.

In the compiled inference framework, we must be able to generate meaningful execution traces given the default-constructed observations. In this problem, if

we default construct the observations; *i.e.*, the `std::vector<std::pair<double, double»`, we will get an empty vector. If we feed the empty vector to the program, the `for` loop will not run and no points will be observed. This means that we have a second free variable in the model, namely the size `N`.

The issue with the free variable `N` is easier to solve just by considering it a hyper-parameter of the model, and switching the signature of the function to:

```
template<std::size_t N>
void linear_regression(
        const std::array<std::pair<double, double>, N>& observes);
```

Listing 4.8: Bayesian Linear regression with Gaussian noise.

This solution is not entirely satisfactory, but it serves as a workaround. On the other hand, it does not generalise. Note that generally, placing a prior on the size of an already-build iterable collection type as `std::vector`, `std::map` or `std::deque` is not a trivial task.

It is not so clear how to deal with the problem regarding x_i. One would be tempted to put a prior on the x_i and directly sample them from some distribution, but this would mean that we would not be able to fix them as some observations. If one puts a likelihood on the x_i, then the code would work, but we would be changing the joint distribution and the model. And what if the type of the observations was even more convoluted?

To address all these problems and more, we introduce the *metadistributions*. Through this mechanism, we allow the user to place a prior during training time on the free variables of our choice. We achieve this using metaprogramming so we will encode all the necessary information on the C++ type system. This will provide us with a non-intrusive—the model itself does not have to be changed—efficient and scalable solution.

A metadistribution is a class with a static member representing the distribution from which it generates random numbers according to the distribution that it represents. A more compact implementation can be achieved via the use of the helper meta-type `MetaDistribution` and the CRTP pattern—*i.e.*, static polymorphism [7]—as shown in listing 4.9 [8].

[7] See [27], item 49.

[8] This implementation is slightly simplified. The full implementation would put the instantiation

```
// Full Implementation
template<class ResultType=double, int Mean=0, int Std=1>
struct MetaNormal {
    template<class RNG>
    ResultType operator()(RNG &rng) { return distr(rng); }

    static std::normal_distribution<ResultType> distr{Mean, Std};
};

// Compact Implementation using CRTP
template<class ResultType=double, int Mean=0, int Std=1>
struct MetaNormal : MetaDistribution<MetaNormal<ResultType, Mean,
    Std>> {
    static std::normal_distribution<ResultType> distr{Mean, Std};
};
```

Listing 4.9: Example of implementation of a metadistribution.

When we have our metadistribution of choice defined, we can place priors on the input type of the model just by adding an extra parameter at the end using the helper types `cpprob::Builder` and `cpprob::Prior`.

```
void linear_regression(
        const std::vector<std::pair<double, double>, N>& observes,
        cpprob::Builder<
            cpprob::Prior<
                std::vector<
                    std::pair<
                        cpprob::Prior<RealType,
                                      cpprob::MetaNormal<>>,
                        RealType
                    >
                >,
                cpprob::MetaPoisson<>
            >
        >);
```

Listing 4.10: Linear regression model with metapriors on the size of the vector of observations and x coordinates.

The type-combinator

```
template<class T, class MetaDistr>
class Prior;
```

of the distribution outside of the class definition. A proper implementation would also cast explicitly the `Mean` and `Std` arguments to `ResultType`.

puts a prior with distribution `MetaDistr` on the type `T`. The default behaviour of `MetaDistr` is to forward a random value sampled from `MetaDistr` to the constructor of an element of type `T`. For this reason, the only constraint on the type `T` and the `MetaDistr` is that type `T` has to be constructible with an element of the type returned by `MetaDistr::operator()`. This is very handy for classes as `std::vector` where the first parameter of the constructor is the size of the object. The class `cpprob::Prior` can be specialised for types whose initialisation is not as straightforward as passing the value to the constructor.

The meta-type `cpprob::Builder` has the following signature

```
template<class... Types>
struct Builder;
```

It receives as arguments as many types as the original model has. Each of them could optionally be decorated with a meta-prior. It is not necessary to decorate these types with references or cv-qualifiers.

For example, for a model with `int`, `double`, and `const std::vector<int> &` as parameters, where we want to put a `MetaLaplace` prior on the second argument, we would write the following function:

```
void foo(int, double, const std::vector<int> &,
         cpprob::Builder<int,
                         cpprob::Prior<double,
                                       LaplacePrior<>>,
                         std::vector<int>>);
```

As a corner-case, note that this approach also works for models with variadic arguments. Hence, it does not impose any further restrictions on the model.

4.2.7 Communication Protocol

CPProb communicates with the neural network written in PyTorch over the network using messages and sockets.

The rationale behind this choice is that CPProb targets large scale inference. In large models, the bottleneck of the execution is not going to be the communication protocol but the model. On the other hand, this allows having a language-agnostic neural network. An Anglican front-end is currently being developed and we intend to release a fully-fledged Python front-end after the CPProb and neural network

designs are stable. Another advantage of this approach over, for example Python-C++ bindings, is the possibility to scale the training into a distributed approach where CPProb is prompted for traces concurrently by many different neural networks.

We have developed two different protocols; one for compilation and another one for compiled inference.

During compilation, the neural network acts as a client and CPProb acts as the server. At the beginning the neural network asks for a set of traces that it will use as a validation set. After this, the neural network asks CPProb for minibatches of traces that it will use to train.

Each of the traces consists of a list of samples and a list of observes. These two lists contain all the necessary information to train the neural network.

A schematic representation of this phase is shown in figure 4.1.

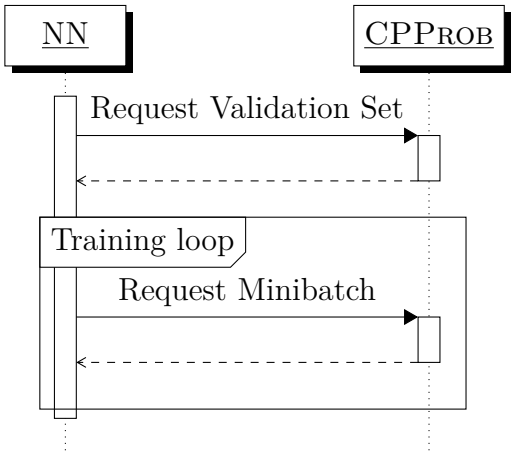

Figure 4.1: Compilation diagram.

During compiled inference, CPProb takes the role of the client. First, CPProb sends the observation. After this, CPProb executes the program, and every time a `sample` statement is hit, CPProb sends a message with information of the current context of the program. Given this information and the state of the LSTM, the neural network proposes parameters to be used in the proposal distribution at that time-step.

A schematic representation of this protocol is shown in figure 4.2

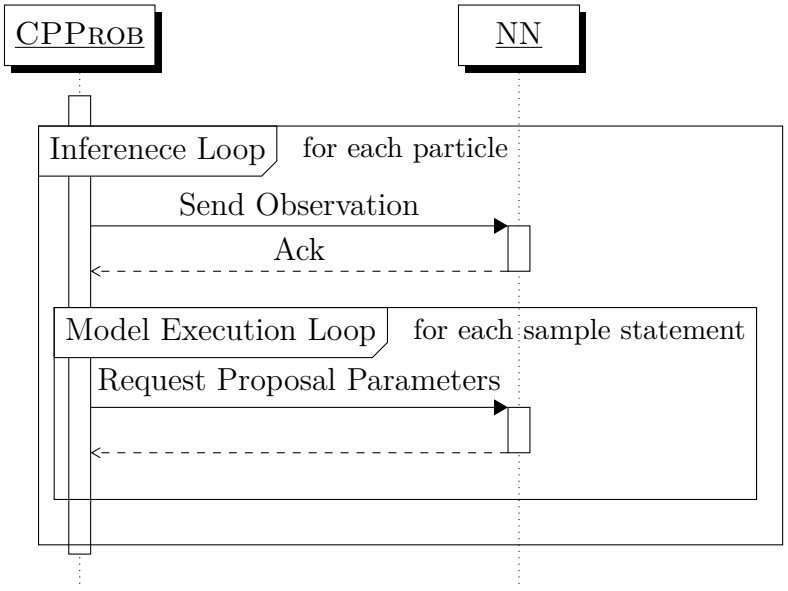

Figure 4.2: Inference diagram.

As a note on the implementation details, Flatbuffers [9] was used for the serialisation of messages and ZeroMQ [10] to handle the sending and receiving of messages.

4.2.8 Customisation Points

One of the main objectives of CPProb is to be extensible by the user without having to modify the library code itself. To achieve this goal, almost all the functions in CPProb are function templates. This allows them to accept general types such that they follow certain protocol. An example of this is the standard library, where almost all the algorithms work on generic iterators.

The first customisation point concerns numbers, both integers and floating point. CPProb supports generating numbers from any number type supported by the distribution used to generate it. As such, the user might want to generate integers larger than 64 bits. CPProb allows the use of these type of integers and any other, provided that they have the same interface as regular integer numbers. The only issue here is that since the neural network just works with

[9] `https://google.github.io/flatbuffers/`
[10] `http://zeromq.org/`

32 bits numbers, the parameters in the proposal distributions will be of, at most, 32 bits, but the generated numbers will be of the type provided by the user. To prove this point we have implemented a class `NDArray` and a distribution `multivariate_normal_distribution` that returns `NDArray`'s.

Distributions also define a customisation point. Both the implementation in the STL and Boost have similarities. We present a minimal template in listing 4.11. The template parameters are used to parametrise the integer, double types and possibly other classes used by the distribution internally. We require every type type in this template to have a default.

```
template<class... Params>
class my_distribution {
public:
  // types
  using input_type = /*unspecified*/;
  using result_type = /*unspecified*/;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    using my_distribution_type = my_distribution;

    // construct/copy/destruct
    param_type() = default;
    explicit param_type(const Params & ...);

    // public member functions

    // getters for the parametrs

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream<CharT, Traits> &
      operator<<(std::basic_ostream<CharT, Traits> &,
                 const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream<CharT, Traits> &
      operator>>(std::basic_istream<CharT, Traits> &,
                 const param_type &);
    friend bool operator==(const param_type &, const param_type&);
    friend bool operator!=(const param_type &, const param_type&);
  };

```

```
36    // construct/copy/destruct
37    explicit my_distribution() = default;
38    explicit my_distribution(const Params & ...);
39    explicit my_distribution(const param_type &);
40
41    // public member functions
42
43    // getters for the parameters
44
45    RealType min() const;
46    RealType max() const;
47    param_type param() const;
48    void param(const param_type &);
49    void reset();
50    template<typename Engine> result_type operator()(Engine &);
51    template<typename URNG>
52    result_type operator()(URNG &, const param_type &);
53
54    // friend functions
55    template<typename CharT, typename Traits>
56      friend std::basic_ostream<CharT, Traits> &
57      operator<<(std::basic_ostream<CharT, Traits> &,
58                 const my_distribution &);
59    template<typename CharT, typename Traits>
60      friend std::basic_istream<CharT, Traits> &
61      operator>>(std::basic_istream<CharT, Traits> &,
62                 const my_distribution &);
63    friend bool operator==(const my_distribution &,
64                           const my_distribution &);
65    friend bool operator!=(const my_distribution &,
66                           const my_distribution &);
67  };
```

Listing 4.11: Minimal interface of a distribution

The most important function of the implementation is the `operator()` used to sample from the distribution given a pseudo-random number generator like `std::mt19937`.

For distributions we also require to implement the free function and trait shown in listing 4.12.

```
1  template<class... Params>
2  RealType logpdf(
3    const my_distribution<Params...>& distr,
4    const typename my_distribution<Params...>::result_type & x);
5
6  // The Params parameter pack contains the parameters of the prior
       distribution
```

```
template<class... Params>
struct proposal<my_distribution, Params...> {
  static constexpr auto type_enum =
            infcomp::protocol::my_distribution::MyDistribution;

  using type = proposal_distribution<Params...>;

  static type
  get_distr(const infcomp::protocol::ProposalReply* msg);
};

template<class... Params>
constexpr infcomp::protocol::my_distribution
    proposal<my_distribution, Params...>::type_enum;
```

Listing 4.12: Free function and trait to be implemented for every distribution.

The free function should return, as the name may suggest, the logarithm of the density function of the distribution evaluated in the argument `x`.

The trait implements the type of the posterior distribution and a static function that, given a message sent from the neural network with the proposal parameters, returns a proposal object. In the current implementation, the distribution must have a proposal distribution. We plan to relax this condition in future versions.

We implement three custom distributions, with names `min_max_discrete`, `min_max_continuous` and `multivariate_normal`.

The first is the discrete distribution on a certain discrete intervals, used as the posterior for the uniform discrete distribution over the same interval. The second is a concave beta distribution scaled to fit in an interval $[a, b]$, used as a posterior for the uniform distribution on $[a, b]$.

The last customisation point that CPProb offers is that of metapriors and metafunctions. This custimosation point was described in section 4.2.6.

Remark. The explicit definition of most of these customisation points will be much easier when the Concepts-Lite proposal [11] is finally approved.

[11] See proposal N3351.

Chapter 5

SHERPA

To illustrate the capabilities of CPProb, we consider problems in the realm of high-energy physics (HEP). HEP is a field where models are backed by a solid ground of mathematics, and therefore yield very complex and computationally expensive programs. The inherent uncertainty of quantum mechanics make these models stochastic in nature, making HEP experiments the perfect framework for large-scale probabilistic programming.

In HEP experiments, such as the Large Hadron Collider (LHC), high-energy beams of particles collide tens of millions of times per second. Each collision creates thousands of particles that then are detected after their interaction with around 100 million electronic sensors. In figure 5.1, we show the full process of particle interaction and detection. First, the particles interact at a quantum mechanical level and then the generated particles are detected by the calorimeters after interacting with them at a macroscopic level.

SHERPA [14] (Simulation of High-Energy Reactions of Particles) is a Monte Carlo simulator that models the microscopic quantum mechanical interactions. SHERPA simulates the proton collisions and gives a stochastic output of the stable particles resultant from this interactions. These different phases are represented in the box in figure 5.1. In the current set-up, the detection is modelled as a Gaussian diffusion of the particles and their momenta. Posterior experiments will try to perform inference replacing this Gaussian diffusion with a proper simulator for the particle-matter interaction, as is the current standard GEANT4 [2].

The fundamental complexity of the models in HEP have retrained the community

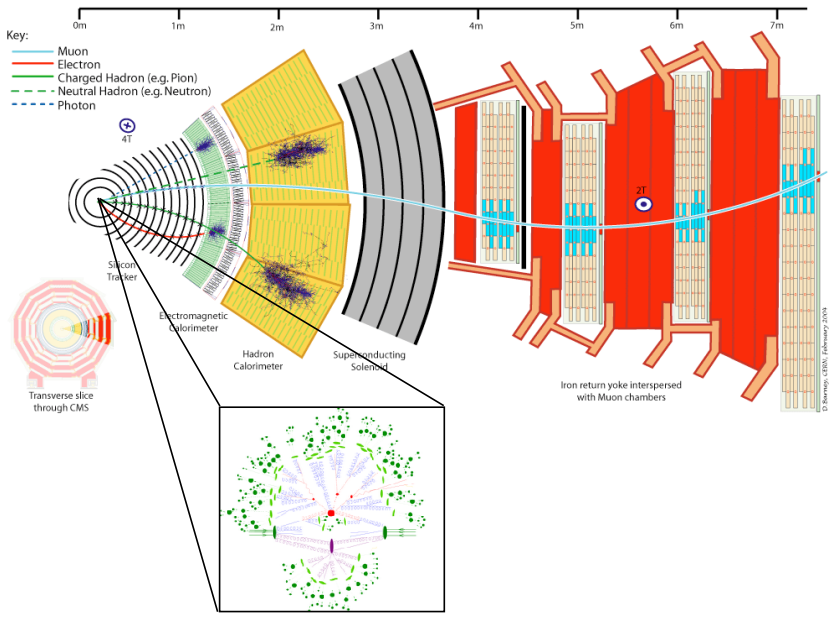

Figure 5.1: A schematic of the generative model for particle physics simulations including the microscopic quantum mechanical interaction modelled by SHERPA and the macroscopic interaction of particles with matter in a particle detector modelled by a Gaussian diffusion.

from applying classical inference methods to these problems, until now. Instead, ad-hoc methods are often used, but all them disregard one of the key features that make these problems unique; the inherent structure present in the microscopic physics models, which is represented by the forward simulator.

5.1 Experimental Setup

HEP problems are usually inference problems. Particles interact and decay into other particles at a quantum mechanic level. The data that is observed from these interactions is the result of the interaction of the generated particles with the detectors. From these observations, we want to infer properties of the particles that generated them.

Most of the particles that are generated from interactions of particles are hadrons, but from analysing hadrons is difficult to separate interesting events from those who are not. Leptons on the other hand, are produced by particles relevant

to HEP studies—*e.g.*, Higgs Boson, W-boson and Z-boson—, while other events rarely produce leptons.

There are three types of leptons, namely electrons, muons, and taus. Electrons and muons are stable particles, in the sense that they do not decay further. For this reasons, they are easier to detect since they just leave one trace in the detectors, so just one trajectory has to be reconstructed. On the other hand, tau particles do decay into other particles, and therefore its detection is much more difficult. Thus, we centre our efforts in studying the decay of the tau lepton.

In this simulations we study the decay of a tau lepton with unknown initial momentum. The initial momentum (p_x, p_y, p_z) of the tau particle follows a distribution

$$p_x \sim \text{Uniform}(-2, 2) \qquad p_y \sim \text{Uniform}(-2, 2) \qquad p_z \sim \text{Uniform}(40, 50).$$

During a simulation, SHERPA stochastically chooses a set of particles—channel—to which the initial tau will decay and samples the momenta of these particles based on microscopic physics. In this settings, given an output of this model as the one presented in figure 5.2, we would like to infer the initial momentum of the tau particle and the decay channel that followed.

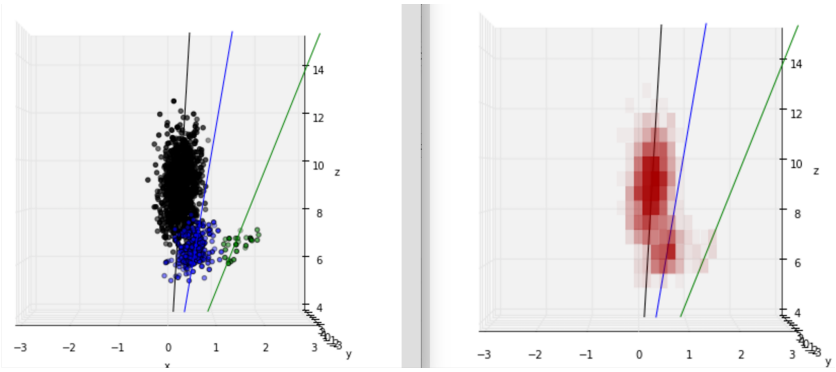

Figure 5.2: Detections generated by SHERPA and posterior rendering of the final observation for the channel $\tau \to \pi^- \gamma\gamma$. The three trajectories are just included for visualisation purposes. The final observation just contains the energy detection present in the image on the right.

5.2 First Approximation

When we first tried to apply the compiled sequential importance sampling method to SHERPAwe found that the execution traces were too long to be able to back-propagate through them in the Neural Network. The average trace length was 249.24, with some traces sampling over 10.000 random numbers. As we can appreciate in the first image in figure 5.3, most of the hits are performed in the cycle $A1, A2, A4, A4, A5$. This sequence of addresses corresponds exactly to the matrix element algorithm, which corresponds to a rejection sampling algorithm to sample from a high-dimensional distribution.

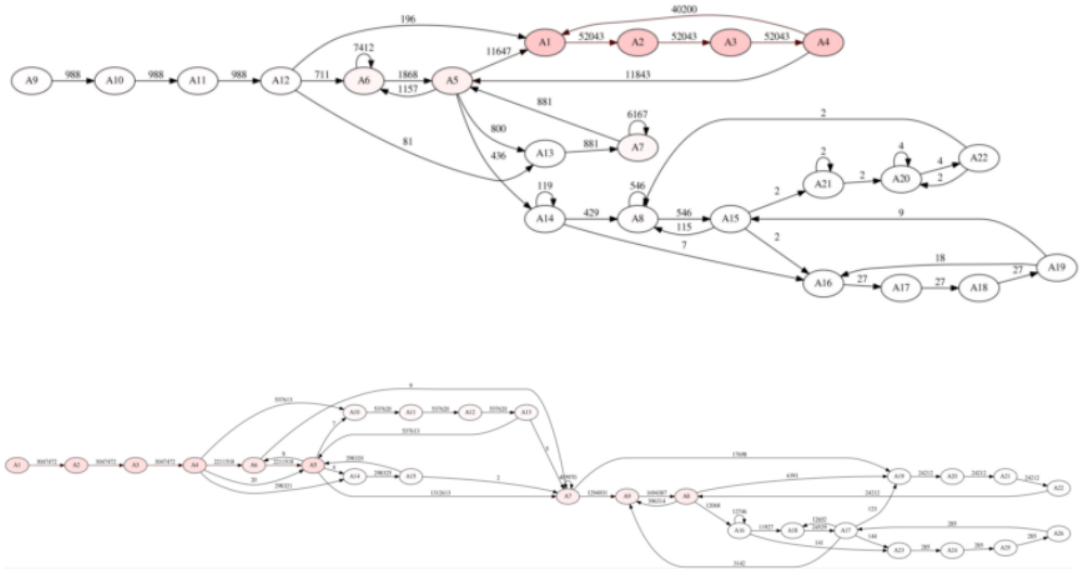

Figure 5.3: Comparison of heatmaps before and after applying the new approach to rejection sampling.

The average length of the traces made training extremely slow, to the point that a computer equipped with a Titan X graphic card could barely process one trace per second.

After the development of the new approach to rejection sampling, the average trace length dropped to an acceptable 8.37, with the longest registered trace being of length 42. This made possible to train the neural network. The second heatmap in figure 5.3 shows how after applying the new approach to rejection sampling, the

addresses that are hit the most are those at the beginning of the program, that are hit in every execution.

Remark. The second heatmap is more complex since it corresponds to more executions of SHERPA, and therefore it is more complete.

5.3 Results

First of all, it is worth mentioning that this is the first time that probabilistic programming has been applied to a problem of this characteristics, so the mere fact of having been able to perform inference using the forward simulator is already remarkable.

Some partial results are shown in figure 5.4 and figure 5.5. The numbers of the particles correspond to the 38 channel in which the τ lepton can decay. The technical names and decay probabilities can be found in [32] under the section τ *branching fractions*.

The channels with lower numbers are the most probable, and therefore are the channels that the neural network has seen most often during training. For this reason, the proposal parameters are better for these observations, and the highest accuracies are found in this area. On the other hand, the decay channels that are less likely to be generated are almost never predicted correctly.

Remark. The posterior distributions are almost always very peaked, since the distribution used in the observation is a Gaussian distribution over an $35 \times 35 \times 20 = 24.500$ dimensional space with low variance. This makes most of the particles very unlikely, and only gives a high weight to the generated particle that is closest to the observation. This will be improved in the future with some ABC-likelihood distribution, to reduce the dimensionality of the problem.

The result of an accuracy of a 52.64% could still be improved, but it is remarkable for a completely new approach to the field.

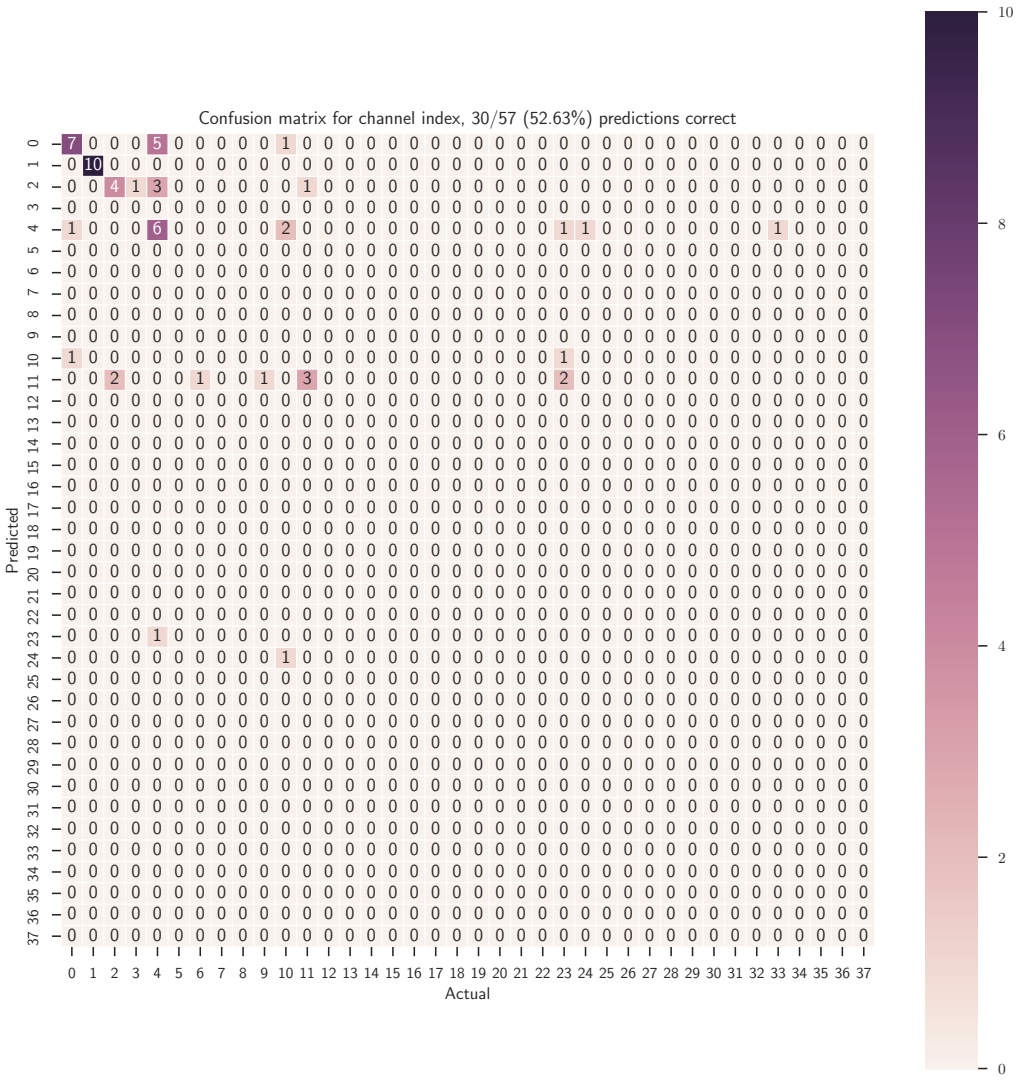

Figure 5.4: Confusion matrix of several channel predictions with 10.000 samples.

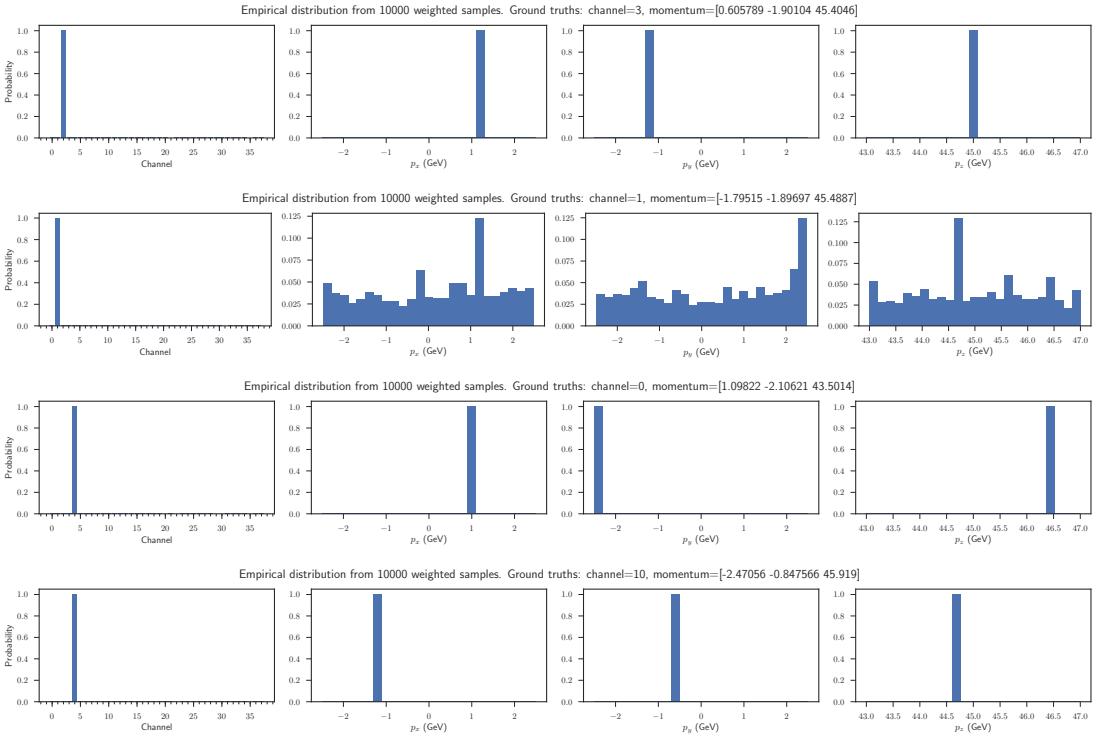

Figure 5.5: Posterior distributions after 10.000 samples. The channel predicted in the first two is correct, and in the last two incorrect.

Chapter 6

Benchmarking CPProb

In this section we benchmark the correctness and efficiency of CPProb on several classic models with analytic posterior. We also present use cases for the rejection sampling and the metapriors and assess their effectiveness.

Remark. In what follows, there will be a slight discordance in the notation between mathematical notation and the one used to define distributions in C++. In mathematics, it is customary to parametrise the normal distribution via its mean and variance. On the other hand, the STL and Boost libraries parametrise it by its standard deviation. Hence, when we write $\mathcal{N}(1, 2)$ we will refer to a normal distribution with variance 2 and when we define a distribution `normal_distribution`, the second argument on its constructor will be its standard deviation.

6.1 Gaussian Conjugate with Unknown Mean

The Gaussian conjugate with unknown mean is one of the most well-known examples of conjugate distributions. The model can be defined mathematically as

$$\mu \sim \mathcal{N}(\mu_0, \sigma_0^2) \qquad y_i \sim \mathcal{N}(\mu, \sigma^2).$$

where the free variables in the model are hyperparameters.

This model was already introduced in section 2.3.2, where we computed its posterior explicitly. We will now use the formula for the posterior computed in that section.

An instance of this model written in CPProb is shown in listing 6.1.

```cpp
template<class RealType = double>
void gaussian_unknown_mean(const RealType y1, const RealType y2)
{
    using boost::random::normal_distribution;
    normal_distribution<RealType> prior {1, std::sqrt(5)};
    const RealType mu = cpprob::sample(prior, true);

    normal_distribution<RealType> likelihood {mu, std::sqrt(2)};
    cpprob::observe(likelihood, y1);
    cpprob::observe(likelihood, y2);
    cpprob::predict(mu, "Mu");
}
```

Listing 6.1: Gaussian conjugate with unknown mean model with hyperparameters $\mu_0 = 1$, $\sigma_0^2 = 5$, $\sigma^2 = 2$ and two observations.

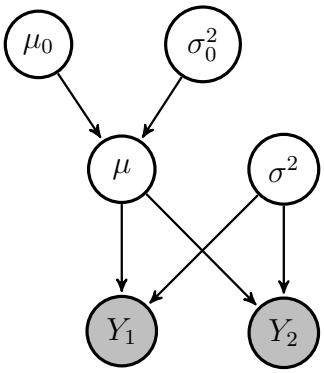

Figure 6.1: Graphical model representing the Gaussian conjugate with unknown mean and two observations problem.

We will test this model with the parameters $y_1 = 8$ and $y_2 = 9$. For these observations, the posterior is given by

$$\mu \mid y_1, y_2 \sim \mathcal{N}\Big(7.25, \frac{5}{6}\Big).$$

In figure 6.1 we show the dependencies of the model as a graphical model.

Note that this is a fairly difficult problem for regular sequential importance sampling, since the true mean of the posterior is approximately three standard deviations away from the prior mean. This means that just around 1 in 1000 particles will have a high weight. Observe that for this problem the SMC algorithm is of no help since there is just one prior. For this problem algorithms of the SMC family with the priors as proposals will behave exactly as their SIS counterparts.

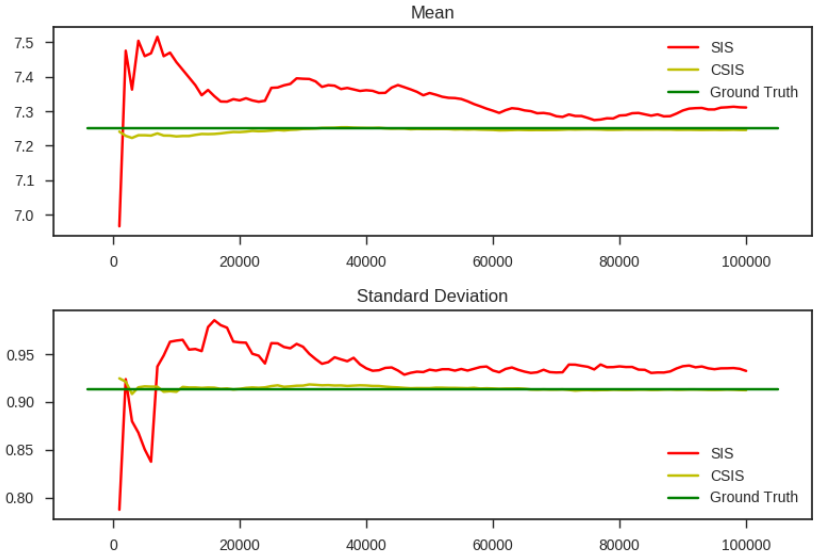

Figure 6.2: Evolution of the empirical estimators for the mean and the standard deviation in the Gaussian prior with unknown mean model.

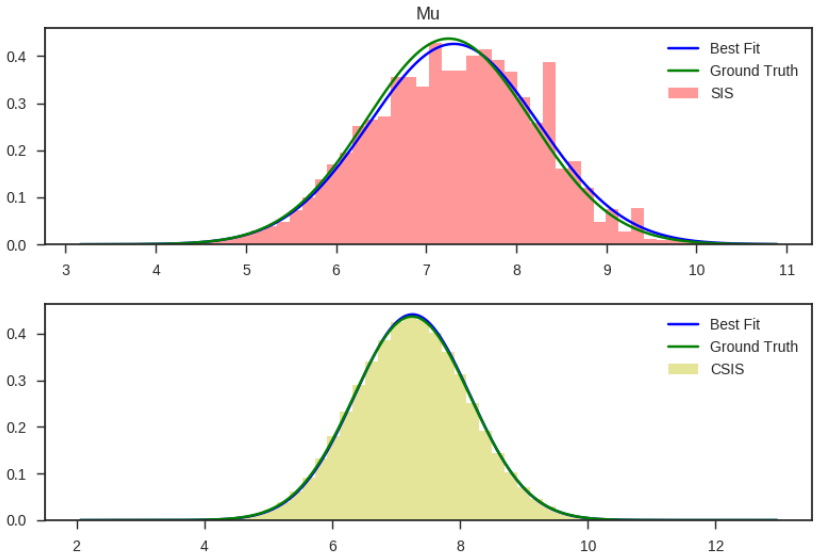

Figure 6.3: Histogram of 10.000 samples of the posterior for SIS and CSIS in the Gaussian prior with unknown mean model.

Figure 6.2 shows the evolution of the mean and standard deviation for the parameter $\mu \mid y_1, y_2$. As we have mentioned, this problem is a very difficult one for standard SIS methods when the prior is used as the proposal. Even with 10.000 particles the estimators are quite suboptimal. On the other hand, CSIS attains almost a perfect convergence with just a few particles. Furthermore, even though the normal distribution that best fits the data is very close to the true posterior in both cases, as seen in figure 6.3, the distribution is much more uneven in the case of SIS. This could be evaluated with the computation of higher order moments of the distribution. As such, we see that CSIS not only converges much faster, but it converges with a higher order precision, in the sense of moment approximation.

6.2 Simulation of a Normal Distribution via Rejection Sampling

In this model, instead of directly sampling from a Gaussian distribution, we approximate it by sampling from a uniform distribution in the range $[\mu_0 - 5\sigma_0, \mu_0 + 5\sigma_0]$ via rejection sampling. The CPProb program that represents this models is shown in listing 6.2.

In this case, we change slightly the model, since the prior that we are using is not a full normal distribution, but a Gaussian distribution truncated in the values $1 \pm 5\sqrt{5}$.

Although this model is almost mathematically equivalent to the last model, it is intrinsically different as a probabilistic program. This model cannot be represented with a graphical model, since the depth of the model is not bounded, hence it is a model that it is not representable in a non-Turing-complete probabilistic programming language.

Models where sampling from a distribution defined by an unnormalised probability function that is evaluable pointwise are very common when complex dynamic systems are involved. The SHERPA particle simulator is one of them, as we already mentioned.

With the method to deal with rejection sampling proposed in section 4.2.5 we are able to train the neural network and approximate the posterior accurately. When naïve CSIS is used, we find that the compilation step is extremely slow,

```cpp
template<class RealType=double>
void normal_rejection_sampling(const RealType y1, const RealType
    y2)
{
    using boost::random::normal_distribution;
    using boost::random::uniform_real_distribution;

    const RealType mu_prior = 1;
    const RealType sigma_prior = std::sqrt(5);
    const RealType sigma = std::sqrt(2);

    // Simulate N(mu_prior, sigma_prior) using just its pdf
    auto f = boost::math::normal_distribution<RealType>(mu_prior,
                                                        sigma_prior);

    // Sample from Normal Distr
    const auto maxval = boost::math::pdf(f, mu_prior);
    uniform_real_distribution<RealType> proposal{
                                        mu_prior - 5*sigma_prior,
                                        mu_prior + 5*sigma_prior};
    uniform_real_distribution<RealType> accept {0, maxval};
    RealType mu;

    cpprob::start_rejection_sampling();
    do {
        mu = cpprob::sample(proposal, true);
    } while (cpprob::sample(accept) > boost::math::pdf(f, mu));
    cpprob::finish_rejection_sampling();

    normal_distribution<RealType> likelihood {mu, sigma};
    cpprob::observe(likelihood, y1);
    cpprob::observe(likelihood, y2);
    cpprob::predict(mu, "Mu");
}
```

Listing 6.2: Gaussian conjugate with unknown mean model with hyperparameters $\mu_0 = 1$, $\sigma_0^2 = 5$, $\sigma^2 = 2$, two observations and a truncated normal distribution as prior, simulated using rejection sampling.

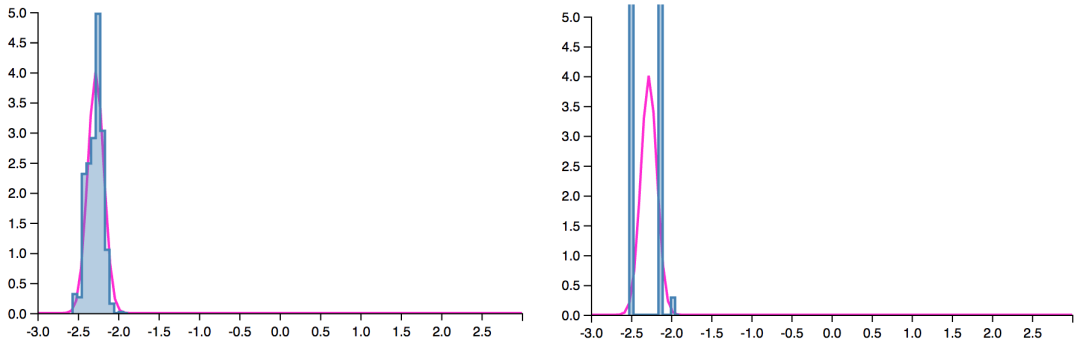

Figure 6.4: Comparison of 100 samples from a model with a prior $\mathcal{N}(0, 1)$ simulated with a $\mathcal{N}(2, 3)$ and then observing a $x = -2.3$.

processing less around 4 traces per second—or a mini-batch per minute—, in comparison to the 300 traces per second processed with our method. This makes the compilation step of naïve CSIS almost impossible. We had to stop the training for this model after 14 minutes without having achieved a clear convergence, while our method converged in 30 seconds.

On top of all this, naïve CSIS algorithm also performs much worse than when trained with the guards, as we show in figure 6.4.

6.3 Finite State-Space Hidden Markov Model

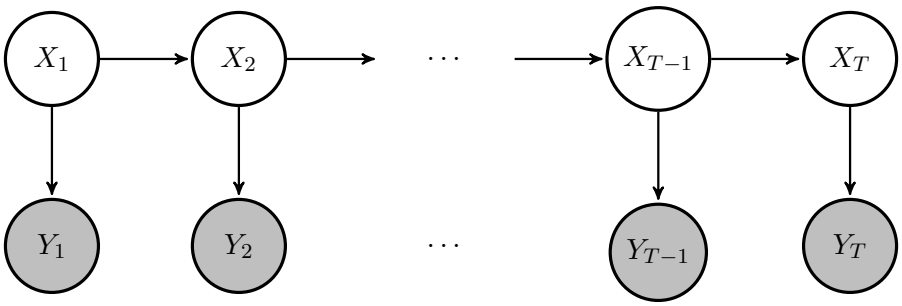

Figure 6.5: Graphical model representing a Hidden Markov Model (HMM).

We have already introduced the hidden Markov model (HMM) family in section 3.1.1 since it is one of the simplest examples of state-space models. We show in figure 6.5 a graphical representation of this family of models.

```
template<std::size_t N>
void hmm(const std::array<double, N> & observed_states)
{
  using boost::random::normal_distribution;
  using boost::random::discrete_distribution;
  using boost::random::uniform_smallint;

  constexpr int k = 3;
  static std::array<double, k> state_mean {{-1, 0, 1}};
  static std::array<std::array<double, k>, k> T {
                                  {{{ 0.1,  0.5,  0.4 }},
                                   {{ 0.2,  0.2,  0.6 }},
                                   {{ 0.15, 0.15, 0.7 }}}};
  uniform_smallint<> prior {0, 2};
  auto state = cpprob::sample(prior, true);
  cpprob::predict(state, "State");
  auto obs_it = observed_states.begin();
  normal_distribution<> likelihood {state_mean[state], 1};
  cpprob::observe(likelihood, *obs_it);
  ++obs_it;

  for (; obs_it != observed_states.end(); ++obs_it) {
    discrete_distribution<> transition_distr {T[state].begin(),
                                              T[state].end()};
    state = cpprob::sample(transition_distr, true);
    cpprob::predict(state, "State");
    likelihood = normal_distribution<>{state_mean[state], 1};
    cpprob::observe(likelihood, *obs_it);
  }
}
```

Listing 6.3: Finite state-space model with gaussian emisions and uniform initial distribution.

The finite state-space model is a hidden Markov model in which the latent variables are discrete variables over some finite space. In most of the cases they are also identically distributed. The observations in our model will be Gaussian distributions and the initial density will be uniform. The full model specified in CPProb can be seen in listing 6.3.

We show in figure 6.6 the behaviour of the sumo of he ℓ_2 distances of the 16 components of the posterior distribution of the finite state-space HMM model. As we can see, through the choice of adaptative proposal distributions we are able to outperform naïve SIS by an order of magnitude.

We get a even more interesting result when we compare the performance between CSIS and SMC. To get these results we used the probabilistic programming language Anglican [45], since CPProb does not implement SMC yet. As we can see in figure 6.7, CSIS is able to outperform SMC even in a Markovian model. This is remarkable, given that, as we showed in section 3.3, SMC is an algorithm that clearly benefits from a Markovian behaviour, since it is a greedy algorithm. On the other hand, as we studied, the variance of the estimators increase exponentially with the dimensionality of the problem if the proposal distributions are not good enough. Here we show that using compiled inference, we are able to find proposals that counter this exponential growth, making this SIS outperform SMC.

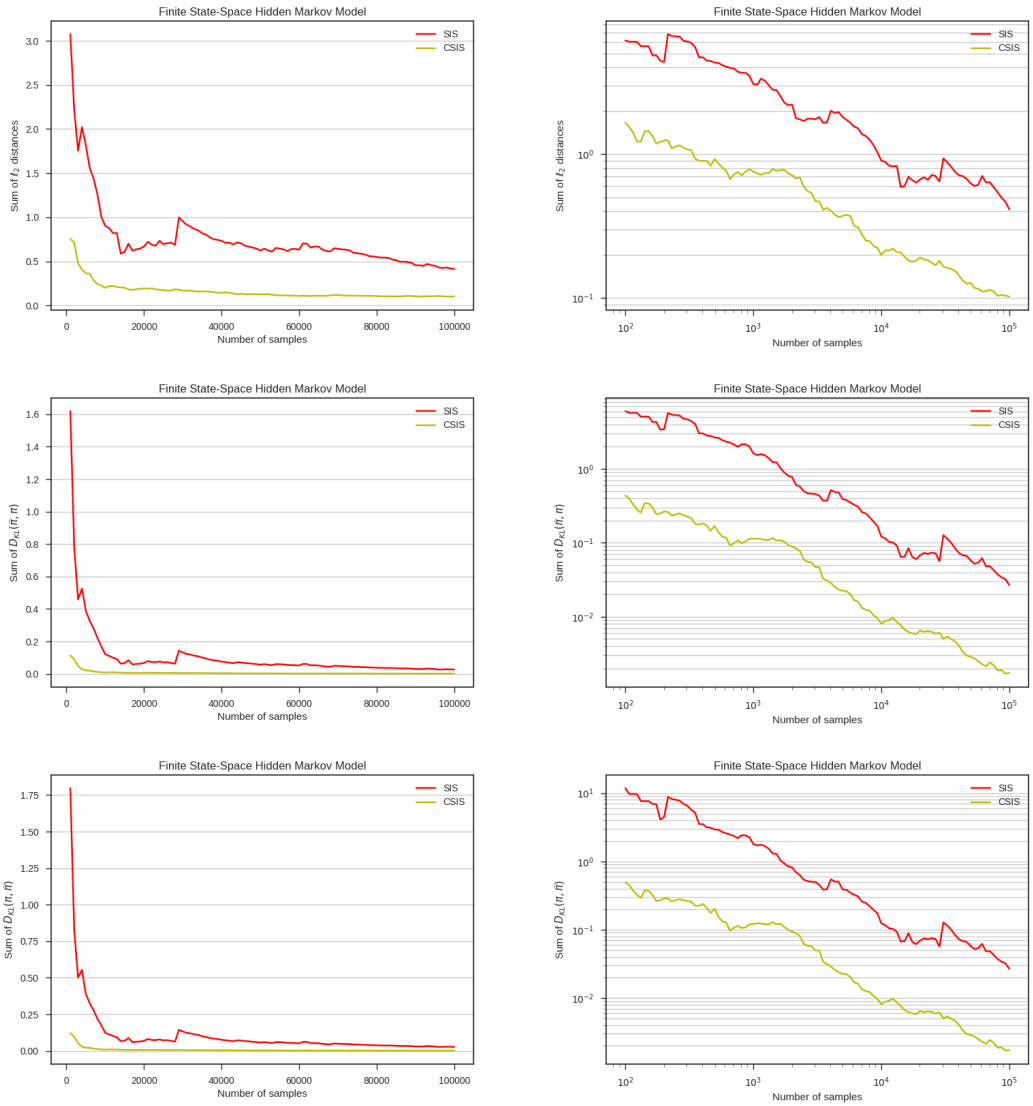

Figure 6.6: Linear and log-log plots of the sum of the KL-divergences and sum of ℓ_2 distances between the true posterior and the approximations given by CSIS and SIS in the finite state-space HMM model of length 16.

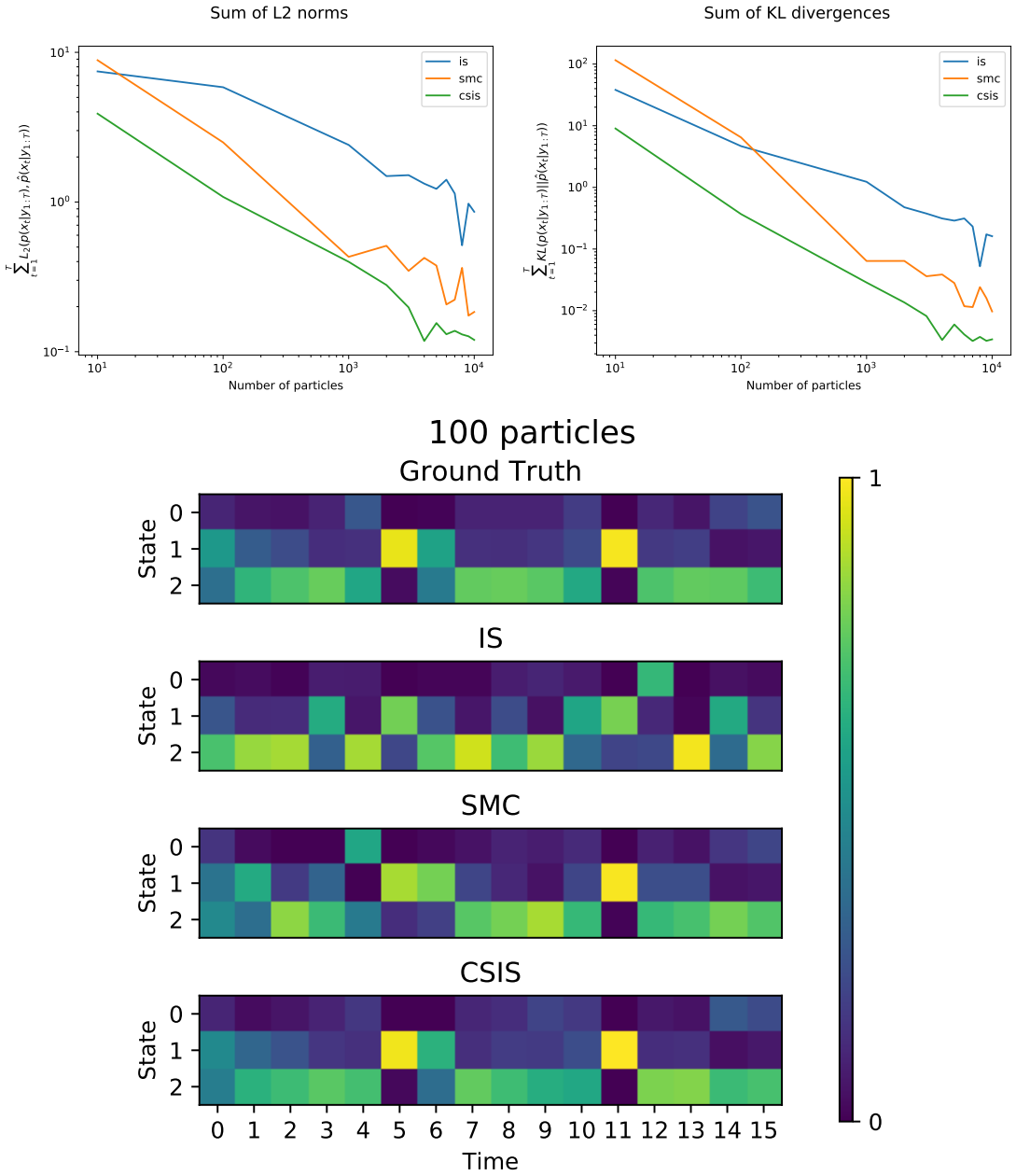

Figure 6.7: Log-log plots of the sum of the KL-divergences and sum of ℓ_2 distances between the true posterior and the approximations given by SMC, CSIS and SIS in the finite state-space HMM model of length 16.

Chapter 7

Conclusions and Future Work

In this project we have investigated the design and implementation of a library to perform inference in probabilistic models written in C++. To our knowledge, this is the first implementation that allows inference to be performed in pre-existing models. We have coupled it with SHERPA, a large-scale high energy particle simulator.

We have compiled an introductory review on the field of particle algorithms, and their counterparts for probabilistic programs, relevant for the understanding of the inference algorithms in CPProb. We mentioned a few points that we understand to be missing in the theoretical treatment of the subject, and we provide justifications for some of them. We have also provided the first justification for the choice of the target distribution in Compiled Sequential Importance Sampling based on theoretical results.

This project is the starting point of what will be a fully-fledged probabilistic programming library with more inference algorithms.

To finish the exposition, we present the two main ideas to be explored in the near future.

7.1 Probabilistic Programming as a Protocol

Probabilistic programming languages only need to implement three functions: `sample`, `observe` and `predict`. Strictly speaking, for a minimal implementation

of a probabilistic programming language, only the first two are needed, as we can define the convention that the predicted values are those returned by the model.

Implementing these functions in a general and high-performance way is far from a trivial task, as the 5.600 lines of template meta-programming code suggest. If on top of that we implement SMC-like algorithms, further measures have to be taken, since concurrency issues start to come into play. Furthermore, different languages have different predefined features and different libraries. This translates into the fact that what in one language can be very easy to implement, in another might be virtually impossible [1].

These reasons and more mean that even though the approach followed here is extensible and can be implemented in many other languages, it is not practical to implement a library like CPProb in every language in which we want to perform inference.

For these reasons, we propose to see probabilistic programming as a protocol. In this protocol there will be a controller, CPProb, and a client language that communicates with it. The client executes the model, and every time a `sample` or `observe` statement is hit, a message is sent to the server. Then, the server answers with the information necessary to continue the execution.

The client does not even need to have access to random number generators. When the client wants to sample from a normal distribution, it sends a `Sample` message to the server indicating the distribution from which it wants to sample, the distribution parameters, and possibly other extra parameters. The server will then return a random sample according to the algorithm of choice and the proposal distribution. Furthermore, the server will return a `Command`. Examples of these commands could be to return the sampled number, or fork the process and resume both of them, or to continue the execution of a previously paused process.

The approximation to sampling algorithms as sampled random numbers, log probabilities, and commands, provides a very powerful and extensible framework. The client language can advertise to CPProb the commands that it has implemented. Given these commands, the CPProb responds with the algorithms

[1] Consider the example of runtime reflection in C++ vs Lisp or Python.

available given these commands. More complex and difficult to implement commands, as fork or copy of coroutines, give access to better algorithms such as SMC or iPMCMC.

As a by-product of abstracting the algorithms through a C++ implementation, we are also able to obtain an extremely lightweight implementation on the client side.

On the performance side, we must say a word on the decision of implementing network protocols over regular C++ bindings. CPProb is designed to be a fast backend, to be used on computationally expensive models. In these models, sending less than 50 messages over the network per execution will not be the bottleneck of the program. This is even less likely in the compiled inference framework, where a neural network is being trained. For the cases where performance is a must, we recall that it will be always possible to call CPProb from the client programming language.

7.2 Normalising Flows

We devoted a full chapter to study in depth the relation between the proposal distributions and the performance of the estimators associated to the algorithms of the sequential Monte Carlo family of algorithms. After that, we did not apply these ideas at their fullest in CPProb, where non-meaningful families of proposals are chosen for the prior distributions. Although we mentioned that this issue is left for future improvement and that we have a better approximation in mind. This approximation involves the use of normalising flows as the family of proposal distribution.

Normalising flows were first introduced in [37]. The main idea is to choose as the parametric family of proposals a family of distributions described by a neural net that, given a random number as a vector, acts as a random variable, transforming this random source into a much more complex, possible multi-modal, random variable.

In compiled inference we just have to be able to perform three operations on a parametric family of distributions to be able to use them as proposal distributions.

- Sample from it.
- Compute the logarithm of its density function.
- Derive the logarithm of its density function with respect to the parameters.

Normalising flows differ from other approaches to approximating random variables, as VAEs, in that it is possible to perform the second and third operations on them. In normalising flows we are able to compute the density function as the Jacobian of the change of variables between the source of randomness and the final variable.

Normalising flows are a step further to closing the space between compiled inference and amortised inference, since we could see the resulting proposal probabilities as *learning* the dependencies in the model. This would mean that we could adapt the model given some observations and continue training them, hence adapting at each step the model and the inference algorithm.

Bibliography

[1] S Agapiou, O Papaspiliopoulos, D Sanz-Alonso, and AM Stuart. Importance sampling: Intrinsic dimension and computational cost. *arXiv preprint arXiv:1511.06196*, 2015.

[2] Sea Agostinelli, John Allison, K al Amako, J Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G Barrand, et al. Geant4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[3] Charalambos D Aliprantis and Owen Burkinshaw. *Principles of real analysis*. Gulf Professional Publishing, 1998.

[4] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[5] Olivier Cappé, Eric Moulines, and Tobias Ryden. *Inference in Hidden Markov Models (Springer Series in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[6] Nicolas Chopin et al. Central limit theorem for sequential monte carlo methods and its application to bayesian inference. *The Annals of Statistics*, 32(6):2385–2411, 2004.

[7] Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. Smc2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 75(3):397–426, 2013.

[8] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2-3):393–405, 1990.

[9] Valentin Dalibard. A framework to build bespoke auto-tuners with structured bayesian optimisation. Technical report, University of Cambridge, Computer Laboratory, 2017.

[10] Nando De Freitas, C Andrieu, Pedro Højen-Sørensen, M Niranjan, and A Gee. Sequential monte carlo methods for neural networks. In *Sequential Monte Carlo methods in practice*, pages 359–379. Springer, 2001.

[11] Randal Douc and Olivier Cappé. Comparison of resampling schemes for particle filtering. In *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, pages 64–69. IEEE, 2005.

[12] Arnaud Doucet, Mark Briers, and Stéphane Sénécal. Efficient block sampling strategies for sequential monte carlo methods. *Journal of Computational and Graphical Statistics*, 15(3):693–711, 2006.

[13] Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3, 2009.

[14] Tanju Gleisberg, Stefan Höche, F Krauss, M Schönherr, S Schumann, F Siegert, and J Winter. Event generation with sherpa 1.1. *Journal of High Energy Physics*, 2009(02):007, 2009.

[15] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[16] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE Proceedings F (Radar and Signal Processing)*, volume 140, pages 107–113. IET, 1993.

[17] Peter J Green. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[18] Hamed Karimi, Julie Nutini, and Mark Schmidt. Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 795–811. Springer, 2016.

[19] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[20] Oleg Kiselyov. Problems of the lightweight implementation of probabilistic programming. In *Proceedings of Workshop on Probabilistic Programming Semantics*, 2016.

[21] Genshiro Kitagawa. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of computational and graphical statistics*, 5(1):1–25, 1996.

[22] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[23] Dexter Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 101–114. IEEE, 1979.

[24] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Computer aided verification*, pages 585–591. Springer, 2011.

[25] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

[26] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[27] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[28] T Minka, J Winn, J Guiver, and D Knowles. Infer .net 2.4, microsoft research cambridge, 2010. *URL: http://research. microsoft. com/infernet*, 2013.

[29] P.D. Moral. *Feynman-Kac Formulae: Genealogical and Interacting Particle Systems with Applications*. Probability and Its Applications. Springer New York, 2004.

[30] Art B. Owen. *Monte Carlo theory, methods and examples*. 2013.

[31] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. *arXiv preprint arXiv:1403.0504*, 2014.

[32] C. Patrignani et al. Review of Particle Physics. *Chin. Phys.*, C40(10):100001, 2016.

[33] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[34] Avi Pfeffer. *Practical probabilistic programming*. Manning Publications Co., 2016.

[35] Michael K Pitt and Neil Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American statistical association*, 94(446):590–599, 1999.

[36] Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial Intelligence and Statistics*, pages 814–822, 2014.

[37] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[38] David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. Bugs 0.5: Bayesian inference using gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pages 1–59, 1996.

[39] Sam Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.

[40] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 525–534. ACM, 2016.

[41] Nick Whiteley, Anthony Lee, Kari Heine, et al. On the role of interaction in sequential monte carlo algorithms. *Bernoulli*, 22(1):494–529, 2016.

[42] David Williams. *Probability with Martingales*. Cambridge University Press, 1991.

[43] David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

[44] David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

[45] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.