UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2006

Probabilistic Programming for Birth-Death Models of Evolution

JAN KUDLICKA

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021

ISSN 1651-6214
ISBN 978-91-513-1123-4
urn:nbn:se:uu:diva-432409

Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Lägerhyddsvägen 2, Uppsala, Thursday, 25 March 2021 at 14:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Bret Larget (University of Wisconsin).

Abstract

Kudlicka, J. 2021. Probabilistic Programming for Birth-Death Models of Evolution. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2006. 84 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-1123-4.

Phylogenetic birth-death models constitute a family of generative models of evolution. In these models an evolutionary process starts with a single species at a certain time in the past, and the speciations—splitting one species into two descendant species—and extinctions are modeled as events of non-homogenous Poisson processes. Different birth-death models admit different types of changes to the speciation and extinction rates.

The result of an evolutionary process is a binary tree called a phylogenetic tree, or phylogeny, with the root representing the single species at the origin, internal nodes speciation events, and leaves currently living—extant—species (in the present time) and extinction events (in the past). Usually only a part of this tree, corresponding to the evolution of the extant species and their ancestors, is known via reconstruction from e.g. genomic sequences of these extant species.

The task of our interest is to estimate the parameters of birth-death models given this reconstructed tree as the observation. While encoding the generative birth-death models as computer programs is easy and straightforward, developing and implementing bespoke inference algorithms are not. This complicates prototyping, development, and deployment of new birth-death models.

Probabilistic programming is a new approach in which the generative models are encoded as computer programs in languages that include support for random variables, conditioning on the observed data, as well as automatic inference. This thesis is based on a collection of papers in which we demonstrate how to use probabilistic programming to solve the above-mentioned task of parameter inference in birth-death models. We show how these models can be implemented as simple programs in probabilistic programming languages. Our contribution also includes general improvements of the automatic inference methods.

Keywords: probabilistic programming, birth-death models, statistical phylogenetics, particle filters, sequential Monte Carlo (SMC)

Jan Kudlicka, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-1123-4
urn:nbn:se:uu:diva-432409 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-432409)

"Mathematics reveals its secrets only to those who approach it with pure love, for its own beauty."

— Archimedes

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I F. Ronquist[†], J. Kudlicka[†], V. Senderov[†], J. Borgström, N. Lartillot, D. Lundén, L. Murray, T. B. Schön, and D. Broman. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. Preprint at `https://www.biorxiv.org/content/10.1101/2020.06.16.154443v4`, 2020.

[†] Equal contribution.

II J. Kudlicka, L. M. Murray, F. Ronquist, and T. B. Schön. Probabilistic programming for birth-death models of evolution using an alive particle filter with delayed sampling. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Tel Aviv, Israel, 2019.

III L. M. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. B. Schön. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1037–1046, Playa Blanca, Lanzarote, Spain, 2018.

IV J. Kudlicka, L. M. Murray, T. B. Schön, and F. Lindsten. Particle filter with rejection control and unbiased estimator of the marginal likelihood. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5860–5864, 2020.

Contributions

Paper [I]

The ideas, concepts and algorithms were developed by all authors in close collaboration. The algorithms and models were implemented by me, V. Senderov and F. Ronquist. Verification experiments and empirical analyses were run by me and V. Senderov. We together also generated most of the illustrations. All authors contributed to writing and revising the manuscript and the supplementary material.

Paper [II]

The ideas were developed in close collaboration among all authors. I implemented the models and support for delayed sampling for relevant distributions in Birch, and contributed to implementing, testing and debugging the extended alive particle filter. I ran all experiments and analyses, and wrote the manuscript, including the proof of unbiasedness of the marginal likelihood estimator for the extended alive particle filter.

Paper [III]

The methods and algorithms used in delayed sampling were developed in close collaboration among all authors. In addition I contributed to developing the examples, and prepared Figure 1 in the manuscript. I also gave detailed feedback on manuscript drafts.

Paper [IV]

The ideas are based on unpublished work that was done in collaboration with L. M. Murray and T. B. Schön, and on [II]. The idea for the first use case was developed in discussion with F. Lindsten, and the second use case is based on a model previously developed by L. M. Murray. The models were implemented in Birch by me and L. M. Murray. I ran all experiments and analyses, and wrote the manuscript, including the proofs.

Contents

Introduction

Welcome to the exciting world of probabilistic programming and phylogenetics!

Phylogenetic birth-death models are a family of rather simple models of evolution of species (or other taxonomic groups). Starting with a single species in the past, they essentially model *speciation* events, when a parent species splits into two descendant species, and *extinction* events, when the whole population of a species dies out. Biologists use these models to estimate their parameters (such as the speciation and extinction rates) based on a phylogenetic tree representing the evolution of the currently living, *extant*, species. This tree can be inferred from our knowledge about some of the extant species, e.g. their morphological traits and genomic data. Estimation of the parameters is challenging due to the fact that this tree is usually only a part of a complete tree that also includes the evolution of (unknown) extinct species.

Birth-death models of evolution play an important role in statistical phylogenetics, yet the path from a new model idea to a software implementation ready to be used by biologists is time-consuming and error-prone. Describing a new model in the language of mathematics, deriving a bespoke inference algorithm, and then implementing it, either in an existing software package, or creating a new one, takes a lot of time and requires experts from several different areas.

Recent developments in probabilistic programming open up an exciting alternative aiming to shorten and simplify the whole process significantly: new models can be expressed as short programs in probabilistic programming languages with just a basic understanding of programming, and without deep knowledge of the complex phylogenetic software packages or libraries. There is no need to derive any bespoke inference algorithm for a new model (nor to implement it): one of the main features of probabilistic programming languages is automatic inference. From the user's point of view, the inference should be as simple as running the program.

During the last five years my colleagues from Uppsala University, KTH Royal Institute of Technology and Swedish Museum of Natural History (as well as other collaborators from abroad) and I have been working towards this goal, not

only to demonstrate that this is indeed possible, but also to make the inference for phylogenetic models fast and efficient.

This thesis is based on a collection of papers written during this exciting time. It also includes a few introductory chapters: Chapter 1 (p. 11) and Chapter 2 (p. 33) introduce probabilistic programming and birth-death models of evolution. Chapter 3 (p. 49) links both topics together and demonstrates how probabilistic programming and probabilistic programming languages can be used in phylogenetics. It also provides a common thread connecting the papers. The conclusion, including some ideas for the future work, is the contents of Chapter 4 (p. 67).

1

Probabilistic programming

1.1 Introduction

Let us start with a simple experiment: Take a die, roll it twice and add the faces. The outcome of both throws, which we will denote by X_1 and X_2, are random variables, and so is their sum $S = X_1 + X_2$. It makes perfect sense to ask questions like these: What is the probability that this sum is 2? Or 3? Or any other possible value? Or said differently, what is the probability distribution of the sum of the faces?

If you are reading this thesis, you probably know quite a lot about probability and statistics, and might already have an idea about how to answer the questions above. In either case, let us go through the solution together. The table below shows all possible outcomes of X_1 and X_2 and their corresponding sum S. We will assume that the die is fair, i.e., the probability of each face is the same. To answer our questions, we just need to count the fractions of each possible value of S in the table.

		Outcome of X_1					
		⚀	⚁	⚂	⚃	⚄	⚅
Outcome of X_2	⚀	2	3	4	5	6	7
	⚁	3	4	5	6	7	**8**
	⚂	4	5	6	7	**8**	9
	⚃	5	6	7	**8**	9	10
	⚄	6	7	**8**	9	10	11
	⚅	7	**8**	9	10	11	12

For example, the probability of the sum being 8 (shown in bold) is 5/36 as there are 5 combinations where $S = 8$ (namely (6, 2), (5, 3), (4, 4), (3, 5) and (2, 6))

out of the 36 possible combinations. If we do this for all values of the sum we get the distribution we are interested in:

s	2	3	4	5	6	7	8	9	10	11	12
$P(S = s)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

This was quite a simple experiment, but how would we go around if we wanted a computer to (automatically) determine this distribution? It is rather easy to write a function (procedure, program) that can simulate the experiment in any programming language that supports random numbers. For example, in Python, such a function could look like this:

```
import random

def model():
   die1 = random.randint(1, 6)
   die2 = random.randint(1, 6)
   return die1 + die2
```

Can we somehow use this function to calculate (an estimate of) the probability distribution? We can call the function we just created "a huge number" of times, and count how many times the function returns 2, 3, . . . , 12:

```
def main():
   N = 10**5
   dist = {}
   for i in range(N):
      s = model()
      if s not in dist:
         dist[s] = 0.0
      dist[s] += 1/N
   return dist
```

In the `main` function we called the `model` function 10^5 times and counted the relative frequencies of all values returned from the function. Figure 1.1 shows the resulting distribution as a bar plot, along with the true probabilities for all values of the sum.

The approach of repeating an experiment to obtain an estimate of the probability distribution of interest is known as the *Monte Carlo* method and it forms the foundation for almost all methods we will use and develop in this thesis.

Now, let us make the experiment a bit more complicated. The die is rolled by a friend and we can't see the outcomes. After the experiment we are told that the sum is less than 5. What can we say about the distribution of X_1 conditioned on this fact?

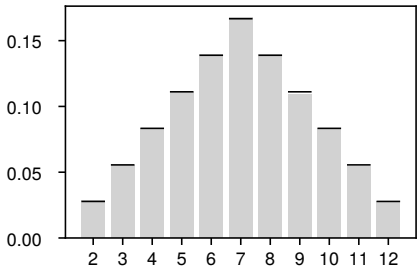


Figure 1.1: Result of the dice experiment. The gray bars represent the estimated probabilities, the black lines the corresponding true probabilities.

Again, we can look at all possible combinations and eliminate those where the sum S is greater than or equal to 5 (shown in gray):

Outcome of X_2 \ Outcome of X_1	⚀	⚁	⚂	⚃	⚄	⚅
⚀	2	3	4	5	6	7
⚁	3	4	5	6	7	8
⚂	4	5	6	7	8	9
⚃	5	6	7	8	9	10
⚄	6	7	8	9	10	11
⚅	7	8	9	10	11	12

There are only 6 combinations that meet the observation, three of them with $X_1 = 1$, two with $X_1 = 2$, and one with $X_1 = 3$, leading to the following conclusion:

x_1	1	2	3	4	5	6
$P(X_1 = x_1 \mid S < 5)$	$\frac{3}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	0	0	0

Before we return to the computer simulations, let us think a little bit about what is happening while we are observing our friend do the experiment. Before we get any information, we believe that the probability of each possible outcome of X_1 is $1/6$—that is our prior belief. As soon we get to know that the sum is less than 5, our belief about (the probability distribution of) X_1 changes. Well, our brains might not immediately calculate the exact probabilities, but we can immediately understand that 4, 5 and 6 are impossible outcomes, i.e., $P(X_1 = 4|S < 5) = P(X_1 = 5|S < 5) = P(X_1 = 6|S < 5) = 0$.

The process of updating the *prior* belief (that is, the belief before the observation) to the *posterior* belief (the belief after the observation) is called *inference*. We looked at a very simple example with two dice, but in general, the inference

for realistic problems might be rather difficult. Developing inference algorithms is a time-consuming and error-prone process. It would be convenient if we could just implement a *generative model*, i.e. a model that just simulates the experiment, and then add statements about the observations, and run the program to get the posterior distribution of interest (or some of its statistics).

Returning to our experiment with the dice, we would like to extend the `model` function we wrote by just adding a single line stating that "the sum is less than 5" (and returning the outcome of the throw instead of the sum):

```
def model():
    die1 = random.randint(1, 6)
    die2 = random.randint(1, 6)
    observe(die1 + die2 < 5)
    return die1
```

Probabilistic programming languages (PPLs) allows us to do exactly this. They extend general purpose (deterministic) programming languages and provide

- support for random variables,
- conditioning on the observed data, and
- automatic inference.

In the existing PPLs the support for random variables is usually more *ergonomic* than what we have seen in our code. For example, instead of

```
die1 = random.randint(1, 6)
```

we can very often see something like

```
die1 ~ DiscreteUniform(1, 6)
```

From the user's point of view, a recipe for probabilistic modeling with a PPL is very simple:

1. Write a generative model as a computer program.
2. Add observe statements.
3. Run the program to automatically do the inference.

But how does this automatic inference work? We will introduce some of the algorithms later, but to offer some intuition already now, consider the following trivial (and often rather inefficient) algorithm, known as *rejection sampling* [62]. Again, we are going to call the `model` function many times, but in the case when the boolean condition in the `observe` statement is not true, we

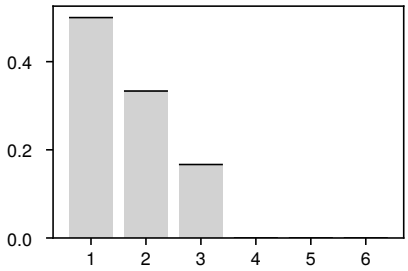


Figure 1.2: Result of the dice experiment with conditioning on the sum being less than 5. The gray bars represent the estimated posterior probabilities, the black lines the corresponding true probabilities.

will return immediately and repeat the function call (and keep repeating until the condition is eventually true). Python is not a probabilistic programming language, but we can use (or rather misuse) exceptions to implement rejection sampling:

```
class RejectionException(Exception):
   pass

def observe(cond):
   if not cond:
      raise RejectionException

def main():
   N = 10**5
   dist = {}
   for i in range(N):
      while True:
         try:
            s = model()
            break
         except RejectionException:
            pass
      if s not in dist:
         dist[s] = 0.0
      dist[s] += 1/N
   return dist
```

Running this program for our toy model results in a distribution shown in Figure 1.2.

Such an approach obviously works only if the rejection ratio is fairly small. We also need to be able to work with continuous random variables, since, after all, most of the random variables in realistic problems are indeed continuous.

In the next section we will define probabilistic programming and PPLs more

formally, go through a few more sample models, and show how they can be implemented as simple programs in PPLs. Execution of probabilistic programs can be modeled using a *programmatic model*, which is introduced in Section 1.3 (p. 22). In Section 1.4 (p. 24) we will turn our attention to automatic inference and the algorithms supporting it, and go deeper into the sequential Monte Carlo (SMC) based inference.

1.2 Basic concepts

As we have seen in the previous section, probabilistic programming is a new programming paradigm that allows encoding a probabilistic model as a computer program, and running the inference automatically, without the need to implement a bespoke inference algorithm.

Probabilistic programming languages (PPLs) can be seen as extensions of general-purpose deterministic programming languages with probabilistic constructs (such as defining random variables and conditioning on the observed data) and automatic inference.

1.2.1 Defining and using random variables

By defining a random variable we inform the compiler or the interpreter about the variable's identifier (i.e., the name of the random variable) and the probability distribution of the random variable. We will use the following notation in pseudocode:

$$x \sim \mathrm{D}(\theta_1, \theta_2, \dots)$$

Here, x denotes a random variable specified by a probability distribution D and its parameters $\theta_1, \theta_2, \dots$. Note that the parameters might, and often will, be expressions using other random variables defined in the program earlier. It is important to realize that the distribution and the parameters specify the conditional probability distribution of the random variable x given the previously defined random variables.

How do PPLs store and work with random variables? The simplest way is to represent them by random variates (i.e., particular outcomes of the random variables). In our illustrative examples in the previous section, we drew outcomes of both throws from `DiscreteUniform(1, 6)` and stored these (integer) values in the (Python) variables `die1` and `die2`. When returning the sum `die1 + die2`, the program just used these stored variate values to calculate and return the sum. Some PPLs even require the user to draw, or *sample*, a value from

the distribution and to store it in a variable (representing the random variate) explicitly. For example, consider the following snippet in WebPPL [23], a simple JavaScript-based PPL that can even execute probabilistic programs in a web browser[1]:

```
var dist = Gaussian({mu: 0, sigma: 1})
var x = sample(dist)
```

Other PPLs deal with random variables in a more advanced way. For example, Birch [46], an object-oriented PPL transpiling to C++, represents random variables as objects with attributes for the specified distribution and its parameters. Birch avoids sampling a particular value as long as it is possible by employing a technique called *delayed sampling* [III]. We will return to it later, for now, consider the following program demonstrating an interesting case where the value of x is never even sampled:

```
x:Random<Real>;
y:Random<Real>;

x ~ Gaussian(0, 1);
y ~ Gaussian(x, 1);
stdout.print("y = " + y.value() + "\n");
```

When executing the program, Birch automatically exploits the conjugacy relationship between $p(x) = \mathcal{N}(0, 1)$ and $p(y|x) = \mathcal{N}(x, 1)$. When the value of y is needed in order to print it to standard output, it is sampled from the marginal distribution $p(y) = \mathcal{N}(0, 2)$. The distribution related to x is then updated based on the sampled value of y, but since we do not need the value of x, it will never get sampled.

Probabilistic programs can use the random variables as any other (deterministic) variables, including to control the flow of the execution in conditionals, loops and recursion. Let us have a look at two examples; the first example demonstrates using a random variable in the predicate of a conditional:

$x \sim \mathcal{N}(0, 1)$
if $x > 0$ **then**
    $y \sim \text{Exponential}(x)$
else
    $y \sim \text{Uniform}(0, 1)$
end if

The second example demonstrates how to use unbounded recursion to define the geometric distribution:

[1] `http://webppl.org`

```
function GEOMETRIC(p)
    x ~ Bernoulli(p)
    if x then
        return 1
    else
        return 1 + GEOMETRIC(p)
    end if
end function
```

Stochastic branching and unbounded recursion make computationally universal PPLs more expressive than probabilistic graphical models (PGM) [32]: all models that can be expressed in the PGM framework can also be expressed using computationally universal, or Turing-complete, PPLs, while the opposite is not true.

1.2.2 Conditioning on the observed data

Probabilistic programming is closely related to Bayesian probabilistic modeling: some random variables are observed and we want to reason about the (posterior) probability distribution of the remaining—unobserved or latent—random variables given the values of the observed variables.

This is where the *observe* construct comes into play. Both the observed value and the probability distribution of the observed random variable must be specified. In pseudocode we will use the following notation:

observe $y \sim \mathrm{D}(\theta_1, \theta_2, \dots)$

As an example, consider a simple linear Gaussian state-space model encoded as a probabilistic program:

$x_0 \sim \mathcal{N}(0, 0.1)$
for $t \in \{1, 2, \dots, 100\}$ **do**
    $x_t \sim \mathcal{N}(0.5 x_{t-1}, 0.1)$
    observe $y_t \sim \mathcal{N}(0.7 x_t, 0.1)$
end for

Here, x_t denotes the unknown (unobserved, latent) state at time t, and y_t the corresponding measurement.

Before we move on to the automatic inference, recall the example from the first section, where we used `observe(die1 + die2 < 5)`. We did not specify any distribution here; we just used a simple predicate. This is however equivalent to

observe true $\sim$ Bernoulli([die1 + die2 < 5])

The parameter is specified using the Iverson bracket $[P]$:

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

1.2.3 Automatic inference

Let X denote all latent random variables in our model, Y all observed random variables, and y the observed values.

For most problems, to implement a generative program (encoding the joint distribution $p(X, Y)$) is simple. As we have seen above, we do not even need PPLs to do so; any programming language with support for drawing random numbers will suffice. We can run such a program multiple times to simulate the model and draw samples from the joint distribution. We can then use these samples to estimate the distribution or the expected value of any test function with respect to this distribution.

However, changing the program to be able to sample from the posterior distribution $P(X|Y = y)$ in standard programming languages might be rather difficult and involves finding (and possibly tailoring) a suitable inference algorithm or, if we are unlucky, developing and implementing a completely new bespoke inference algorithm.

In PPLs such a change is trivial. For example, consider a potentially biased coin with unknown probability of heads π, and an experiment in which we throw the coin 50 times and count the number of heads, encoded as the following generative program:

$\pi \sim$ Uniform(0, 1)
$c \sim$ Binomial(50, π)

Now, how can we get the posterior distribution of π given that we have seen 28 heads? In PPLs, we only need to change the definition of the random variable c to an observe:

$\pi \sim$ Uniform(0, 1)
observe 28 $\sim$ Binomial(50, π)

Running this program produces a sample from the posterior distribution. The inference is automatic; we have not implemented any inference algorithm anywhere in the program. This decoupling of the model and the inference is

one of the main advantages of PPLs. Although it might sound like magic, automatic inference is based on composing suitable general-purpose inference algorithms, some of which we will cover later in this chapter.

Extending and developing new general inference algorithms and heuristics to make automatic inference better (in terms of speed and variance of the estimated quantities) are the subject of ongoing research.

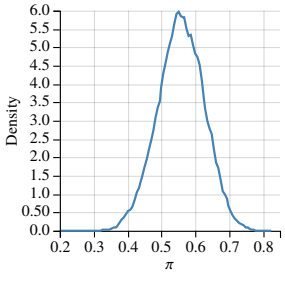
1.2.4 Illustrative examples

Let us now look at a couple of models and their implementation in WebPPL [23]. First, let us return to the example with a coin from the previous paragraph:

```
var coinExperiment = function() {
  var π = sample(Uniform({a: 0, b: 1}))
  observe(Binomial({n: 50, p: π}), 28)
  return π
}

Infer({model: coinExperiment, method: 'MCMC', samples: 100000})
```

The model itself is implemented as a function (`coinExperiment`) which is passed together with the inference parameters (we use Markov chain Monte Carlo (MCMC) as the inference method to collect 100000 samples) to the `Infer` function. This function is responsible for running the inference and produces the estimate of the posterior distribution for π (note the return statement at the end of the `coinExperiment` function). If we run the program in a web browser, WebPPL will also visualize the distribution:



In the second and more advanced example we will demonstrate how easy it is to implement Bayesian linear regression in WebPPL. Consider the following model:

$$\begin{aligned}\beta_0 &\sim \mathcal{N}(0,1),\\ \beta_1 &\sim \mathcal{N}(0,1),\\ y_n &\sim \mathcal{N}(\beta_0 + \beta_1 x_n, 0.01)\end{aligned}$$

and the following measurements:

x_n	1	2	3	4	5
y_n	0.1993	0.1401	0.4304	0.6206	0.7807

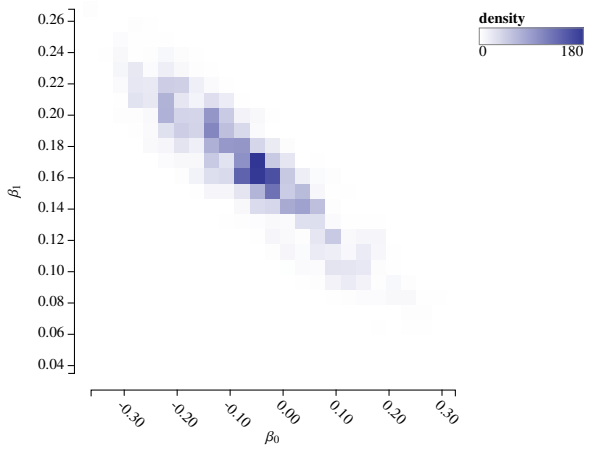
The implementation of this model is straightforward, perhaps with the exception of using the `map` function to process the measurements (WebPPL does not support for-loops):

```
var observation = [
  [1, 0.1993],
  [2, 0.1401],
  [3, 0.4304],
  [4, 0.6206],
  [5, 0.7807]
]

var regression = function() {
  var β0 = sample(Gaussian({mu: 0, sigma: 1}))
  var β1 = sample(Gaussian({mu: 0, sigma: 1}))
  map(function(xy) {
    var x = xy[0], y = xy[1]
    observe(Gaussian({mu: β0 + β1*x, sigma: 0.1}), y)
  }, observation)
  return [β0, β1]
}

viz.heatMap(Infer({
  model: regression, method: 'SMC', particles: 20000
}))
```

We use SMC-based inference (with 20000 particles) and visualize the distribution of the parameters β_0, β_1 in the form of a heat map:



1.2.5 Existing probabilistic programming languages

We have so far seen a few examples of probabilistic programs in Birch and WebPPL. As with deterministic programming languages, there exist many PPLs to choose from, including the following languages (in alphabetical order): Anglican [59], BayesDB [37], Biips [58], Birch [46], BLOG [40], BUGS [21], Church [24], Edward [60], Figaro [52], Gen [10], Hakaru [49], JAGS [53], LibBi [44], Probabilistic-C [51], Pyro [5], STAN [8], Turing.jl [19], Venture [38], WebPPL [23].

These probabilistic programming languages support various programming paradigms, implement various automatic inference algorithms, and target various scientific communities. Many of them are based on existing deterministic languages and/or use existing libraries for machine learning.

For example, Birch, which most of the work included in this thesis is related to, is an object oriented PPL transpiling to C++, and implementing several sequential Monte Carlo (SMC) based inference algorithms.

1.3 Programmatic model

Before we look closer at some of the inference algorithms, let us describe an execution model of probabilistic programs, based on the programmatic model by Murray and Schön [46].

Let $\{V_i\}$ denote a set of all (both latent and observed) random variables in a probabilistic program. Note that this set is either finite or countably infinite. In general we cannot make a one-to-one map from the definitions of random variables and observes in the program to the variables in $\{V_i\}$. Consider, for example, random variables in a loop or in a function that is called multiple times. A definition of a random variable at a given location in the program might correspond to multiple random variables in $\{V_i\}$.

When a program is executed, it encounters the random variables in a specific order. This order might be different for different executions due to stochastic branching (with the exception of the very first encountered random variable). Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_{|\sigma|})$ denote a sequence of indices into the set $\{V_i\}$ in which the random variables are encountered during the execution. Further, let v_i denote a realization of the encountered random variable V_i. If V_i has not been encountered during the execution, the corresponding variate v_i has no value, which we will denote by $v_i = \perp$.

We can define the execution state as a set $x = \{(i, v_i)\}_{i \in \sigma}$ of tuples consisting of indices of the variables encountered so far and their corresponding variates.

At any time during the execution, the index κ of a random variable to be encountered next depends on the current state x deterministically, i.e.

$$\kappa = \mathrm{Ne}(x),$$

where Ne (for *next*) denotes this deterministic function. In case there are no more random variables to be encountered, the Ne function will return $\perp$.

Also the parameters of the distribution p_κ associated with the corresponding random variable V_κ (and given by the probabilistic program) depend on the state deterministically, i.e.

$$V_\kappa \sim p_\kappa(\mathrm{Pa}(x)),$$

where Pa (for *parents*) denotes this deterministic function.

Once V_κ has been realized, κ is appended to the sequence σ, and the current state is updated to $x \cup \{(\kappa, v_\kappa)\}$.

The joint probability distribution of yet unencountered random variables, given the current state can be stated recursively [II]:

$$p(\{v_i\}_{i \notin x} | x) = \begin{cases} p_\kappa(v_\kappa | \mathrm{Pa}(x)) \times p(\{v_i\}_{i \notin x \cup \{(\kappa, v_\kappa)\}} | x \cup \{(\kappa, v_\kappa)\}) & \text{if } \kappa \neq \perp, \\ 1 & \text{if } \kappa = \perp \wedge \forall i \notin x : v_i = \perp, \\ 0 & \text{otherwise,} \end{cases}$$

where $\kappa = \mathrm{Ne}(x)$ and $i \in x$ (resp. $i \notin x$) means that there exists (resp. does not exist) a pair in x whose first element is i. The first case is an application of the chain rule, while the second and third case cover the situation where there are no more random variables to encounter (in that case each member of $\{v_i\}_{i \notin x}$ must be $\perp$). The joint distribution of all random variables encoded by a probabilistic program is given by $p(\{v_i\} | \varnothing)$.

Let $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_T)$ denote a sequence of indices of the observed variables corresponding to the observations $y_1, y_2, \ldots, y_T$. The goal of the inference is to determine (estimate) the posterior distribution

$$p(\{v_i\}_{i \notin \gamma} | \{(\gamma_t, y_t)\}_{t=1}^T).$$

We will assume that each execution encounters all observed variables and in the same order (given by γ), and, for the sake of keeping the algorithms simple, that the last observed variable is also the very last encountered random variable.

1.4 Sequential Monte Carlo based inference

1.4.1 Introduction

Automatic inference in PPLs is based on inference algorithms that were originally developed for other types of models (for example, state-space models). Exact inference algorithms, such as Kalman filtering [30] or enumeration (iterating through all possible outcomes), are methods suitable only for a small class of programs.

On the other hand, approximate and evaluation based inference methods, such as Monte Carlo methods and variational inference (e.g. [61, 63]) as well as combinations of these methods (e.g. [48]), are easily applicable to a much larger class of programs. Monte Carlo methods cover a huge family of algorithms, including Metropolis–Hastings algorithm [39, 26], pseudo-marginal Metropolis–Hastings algorithm [1], Gibbs sampling [20], Hamiltonian Monte Carlo (HMC) [50], and sequential Monte Carlo (SMC) [14, 13].

1.4.2 Monte Carlo integration

Monte Carlo integration (e.g. [7]) is a method of estimating the expected value of a test function $h(x)$ of a random variable[2] $x \sim p(x)$, i.e.

$$I = \mathbb{E}_p[h(x)] = \int h(x)p(x)\mathrm{d}x.$$

Under the assumption that we can sample from $p(x)$ we can draw N samples $\{x^n\}_{n=1}^N$, and estimate the expected value using the following estimator:

$$\widehat{I}_N = \frac{1}{N}\sum_{n=1}^{N} h(x^n).$$

The law of large numbers ensures that

$$\lim_{N\to\infty} \widehat{I}_N = I \text{ with probability 1,}$$

and the central limit theorem ensures that

$$\sqrt{N}(\widehat{I}_N - I) \to \mathcal{N}(0, \sigma^2) \text{ in distribution,}$$

where $\sigma^2 = \mathrm{var}_p\, h(x)$.

[2] Analogously to the notation used in probabilistic programs, we will usually use small letters to denote random variables from now on. We will also assume that continuous random variables admit probability density functions.

1.4.3 Rejection sampling

What can we do if we cannot sample from $p(x)$? Rejection sampling [62], which we have used in the first section, is one of the options we have. Assume that

- $p(x)$ can be evaluated point-wise for all x, and that
- there exists another distribution $q(x)$, which we will call the *proposal* distribution, that we can both sample from and evaluate point-wise, and such that

  $$p(x) \leq Mq(x)$$

  for some $M \in \mathbb{R}$ and all x.

We can get samples from $p(x)$ using the following algorithm (rejection sampling):

1. Sample x from the proposal distribution $q(x)$.
2. Sample u from the uniform distribution on $(0, 1)$.
3. If

   $$u < \frac{p(x)}{Mq(x)},$$

   accept x as a sample from $p(x)$, otherwise go back to step 1.

The obvious disadvantage is that it might be difficult to find a proposal $q(x)$ that ensures low rejection rate. The problem gets more prominent with increasing dimension: the rejection rate grows exponentially with the dimension of the distribution (this is sometimes referred to as the *curse of dimensionality*).

1.4.4 Importance sampling

Unlike rejection sampling, importance sampling [25] does not reject any of the proposed samples. Assume that

- we can evaluate $p(x)$ point-wise up to the proportionality, i.e. we can evaluate

  $$\tilde{p}(x) = Zp(x)$$

  point-wise for all x, where Z is a (possibly unknown) normalization constant, and

- there exists another distribution $q(x)$ that we can both sample from and evaluate point-wise, and such that

$$q(x) = 0 \Rightarrow p(x) = 0,$$

or in other words, that the support of $p(x)$ is a subset of the support of $q(x)$.

The expected value of $h(x)$ with respect to $p(x)$ can be rewritten as follows:

$$\mathbb{E}_p[h(x)] = \int h(x)p(x)\mathrm{d}x = \frac{1}{Z}\int h(x)\underbrace{\frac{\tilde{p}(x)}{q(x)}}_{w(x)}q(x)\mathrm{d}x = \frac{1}{Z}\mathbb{E}_q[h(x)w(x)].$$

The function $w(x) = \tilde{p}(x)/q(x)$ is called the *importance weight* function. The normalizing constant Z is given by

$$Z = \int \tilde{p}(x)\mathrm{d}x = \int \frac{\tilde{p}(x)}{q(x)}q(x)\mathrm{d}x = \int w(x)q(x)\mathrm{d}x.$$

Both the expected value and the normalizing constant can be estimated using Monte Carlo integration. The following algorithm summarizes the estimation of the expected value of a test function using importance sampling:

1. Draw N samples $\{x^n\}_{n=1}^N$ from the proposal distribution $q(x)$.
2. Calculate the importance weights $w^n = \tilde{p}(x^n)/q(x^n)$ for $n = 1, \ldots, N$.
3. Estimate the expected value as follows:

$$\mathbb{E}_p[h(x)] \approx \frac{\sum_{n=1}^N w^n h(x^n)}{\sum_{n=1}^N w^n}.$$

1.4.5 Importance sampling for state-space models

Consider a state-space model of a discrete-time stochastic process, where the state x_t at time t depends only on the state x_{t-1} at the previous time step $t-1$, and x_t can only be observed via a random variable y_t that depends only on x_t (see the graphical representation depicted in Figure 1.3).

The joint distribution of this model is given by

$$p(x_1, x_2, \ldots, x_T, y_1, y_2, \ldots, y_T) = p(x_1)p(y_1|x_1)\prod_{t=2}^{T} p(x_t|x_{t-1})p(y_t|x_t).$$

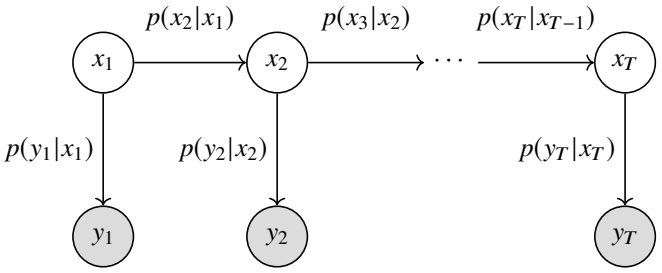


Figure 1.3: Graphical representation of a state-space model.

How can we use importance sampling to estimate the expected value of a test function with respect to the posterior distribution $p(x_1, x_2, \ldots, x_T|y_1, y_2, \ldots, y_T)$?

If we use

$$q(x_1, x_2, \ldots, x_T) = p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1})$$

as the proposal distribution, the importance weight function is given by

$$w(x_1, x_2, \ldots, x_T) = \frac{p(x_1)p(y_1|x_1) \prod_{t=2}^{T} p(x_t|x_{t-1})p(y_t|x_t)}{p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1})} = \prod_{t=1}^{T} p(y_t|x_t).$$

The form of this function allows us to calculate the importance weights in a sequential manner: for each sample $x^n = (x_1^n, x_2^n, \ldots, x_T^n)$ we simulate the Markov chain, starting with sampling

$$x_1^n \sim p(x_1),$$

and then sampling x_t^n based on x_{t-1}^n in a loop:

$$x_t^n \sim p(x_t|x_{t-1}^n).$$

The corresponding importance weight can be calculated iteratively, starting with $w^n \leftarrow 1$ and updating the weight after sampling x_t^n at each time step:

$$w^n \leftarrow w^n p(y_t|x_t^n).$$

Sequential importance sampling, which is the name for importance sampling with the sequential calculation of importance weights, is one of the inference methods used in probabilistic programming languages. Recalling the programmatic model we introduced in Section 1.3 (p. 22), an execution of a

probabilistic program can be modeled as a state-space model, where the latent state x_t corresponds to the execution state at the time of processing the t-th observe, corresponding to the t-th observed random variable y_t.

Sampling from the above-mentioned proposal distribution means nothing else than running the program and sampling values of the unobserved random variables encountered during the execution. The importance weights are calculated by multiplying the likelihoods of the observed values with respect to the sampled values (realizations) of previously encountered random variables. Algorithm 1.1 summarizes sequential importance sampling for probabilistic programming. The PROPAGATE(x) function (Algorithm 1.2) resumes the execution of the program from the state x and continues running the program until it reaches the next observe.

Algorithm 1.1: Sequential importance sampling for probabilistic programming.

for $n = 1, \ldots N$ **do**
    $x^n \leftarrow \varnothing$ ▷ Initialize the n-th execution of the program
    $w^n \leftarrow 1$
end for
for $t = 1, \ldots T$ **do**
    for $n = 1, \ldots N$ **do**
        $x^n \leftarrow \textsc{Propagate}(x^n)$ ▷ Resume the n-th execution until it reaches next observe
        $w^n \leftarrow w^n p_{\gamma_t}(y_t \mid \mathrm{Pa}(x^n))$
        $x^n \leftarrow x^n \cup \{(\gamma_t, y_t)\}$
    end for
end for
for $n = 1, \ldots N$ **do**
    $h^n \leftarrow h(x^n)$ ▷ Calculate the value of the test function of interest
end for
$\mathbb{E}[h] \approx \sum_n w^n h^n / \sum_n w^n$

Algorithm 1.2: The PROPAGATE function.

function PROPAGATE(x)
    $\kappa \leftarrow \mathrm{Ne}(x)$
    while $k \notin \gamma$ **do** ▷ While the next random variable to encounter is not observed
        $v \sim p_\kappa(\mathrm{Pa}(x))$ ▷ Sample the random variable
        $x \leftarrow x \cup \{(\kappa, v)\}$ ▷ Add the random variable to the state
        $\kappa \leftarrow \mathrm{Ne}(x)$
    end while
    return x
end function

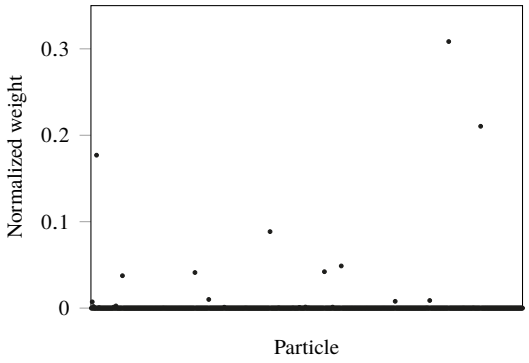


Figure 1.4: Demonstration of the weight degeneracy problem. See the description in the text.

Unfortunately, sequential importance sampling suffers from a problem known as *weight degeneracy* that limits its application: as t increases, the number of samples with extremely small normalized weights (the weights divided by their sum) increases. Eventually, a single sample will have the normalized weight approaching 1 and all remaining samples will have the normalized weights approaching 0. Figure 1.4 demonstrates the problem by showing the weights of 1000 samples for the linear Gaussian state-space model we used as an example on page 18 at time $t = 100$.

Note that this problem does not mean that sequential importance sampling is not useful, nor that it should not be used at all. For example, *inference compilation* [33], combining sequential importance sampling and neural networks, has been successfully used in probabilistic programming inference.

1.4.6 Sequential Monte Carlo

Sequential Monte Carlo (SMC) methods, also known as *particle filters*, addresses the weight degeneracy by resampling the samples, often denoted as *particles*, at certain times. The bootstrap particle filter (BPF), the simplest of the SMC methods, resamples the particles at each time step.

Extending the notation from the previous paragraph, let $\mathcal{S}_t = \{(x_t^n, w_t^n)\}_{n=1}^N$ denote the particle set at time t. The initial set $\mathcal{S}_1$ is constructed by sampling the state

$$x_1^n \sim p(x_1),$$

for each of the N particles, and calculating the corresponding weights

$$w_1^n = p(y_1|x_1^n).$$

To propagate the particles from time $t-1$ to time t, we construct the set $\mathcal{S}_t = \{(x_t^n, w_t^n)\}_{n=1}^N$ by following these steps for each particle:

1. **Resampling**—selecting one of the particles from $\mathcal{S}_{t-1}$ with probabilities proportional to their weights (at time $t-1$), identified by the *ancestor index* a_t^n:

   $$a_t^n \sim \text{Categorical}(\{w_{t-1}^m\}_{m=1}^N).$$

   For the sake of brevity in the notation, we allow the categorical distribution to be parameterized with unnormalized probabilities.

2. **Propagating** the selected particle to the next time step:

   $$x_t^n \sim p(x_t | x_{t-1}^{a_t^n})$$

3. **Weighting** the propagated particle:

   $$w_t^n = p(y_t | x_t^n)$$

Note that this procedure (probabilistically) discards particles with small weights, and that particles with high weights are likely to be selected and propagated to the next time step multiple times.

Once the final set $\mathcal{S}_T$ has been constructed, we can use its particles and their weights to estimate the expected value of a test function of interest. To estimate the normalizing constant Z we need to use the weights from all time steps:

$$\widehat{Z} = \prod_{t=1}^{T} \frac{1}{N} \sum_{n=1}^{N} w_t^n.$$

This estimator is unbiased (e.g. [12]), which allows us to use the BPF (and other SMC methods) in exact approximation methods (such as particle Markov Chain Monte Carlo).

Algorithm 1.3 shows how the bootstrap particle filter (BPF) is used for automatic inference in PPLs. The PROPAGATE function (Algorithm 1.2, p. 28) is the same as before. The biggest challenge when implementing this algorithm in probabilistic programming languages is related to handling of multiple concurrent executions: as we have mentioned above, a single particle might be chosen several times during resampling, which means that the inference engine needs to be able to copy, or clone, running executions. There are several strategies how to handle this efficiently, including using continuation-passing style (CPS) (e.g. [4]) or copy-on-write mechanisms (e.g. [45]).

Algorithm 1.3: Bootstrap particle filter (BPF) for probabilistic programming.

for $n = 1, \ldots N$ **do**
  $w_0^n \leftarrow 1$
  $x_0^n \leftarrow \varnothing$ ▷ Initialization
end for
for $t = 1, \ldots T$ **do**
  for $n = 1, \ldots N$ **do**
    $a_t^n \sim \text{Categorical}(\{w_{t-1}^m\}_{m=1}^M)$ ▷ Resampling
    $x_t^n \leftarrow \textsc{Propagate}(x_{t-1}^{a_t^n})$ ▷ Propagation
    $w_t^n \leftarrow p_{\gamma_t}(y_t | \text{Pa}(x_t^n))$ ▷ Weighting
    $x_t^n \leftarrow x_t^n \cup \{(\gamma_t, y_t)\}$
  end for
end for
for $n = 1, \ldots N$ **do**
  $h^n \leftarrow h(x_T^n)$
end for
$\mathbb{E}[h] \approx \sum_n w_T^n h^n / \sum_n w^n$

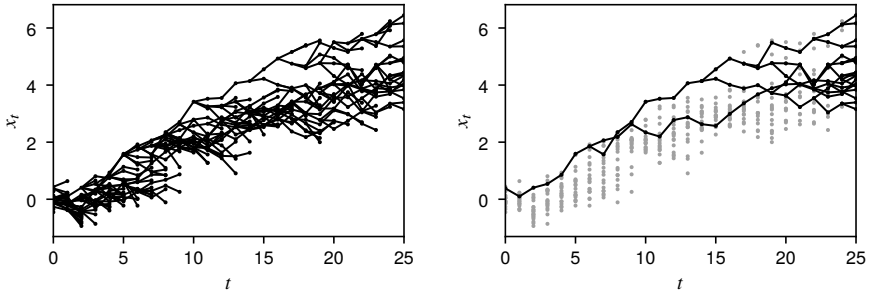


Figure 1.5: Path degeneracy problem. See the description in the text.

Although particle filters do not suffer from the weight degeneracy problem, they do suffer from a problem known as *path degeneracy* (e.g. [2]). Figure 1.5 illustrates the problem for a state-space model with one dimensional state. The plot on the left shows the states of all particles at all time steps. The lines visualize the ancestry history connecting the states of all particles $\{x_t^n\}$ at time t to the states of their ancestors $\{x_{t-1}^{a_t^n}\}$ at time $t-1$. The plot on the right shows the complete ancestry paths of the particles at the very last time step. Note that for $t \leq 6$ all paths coalesce. In the context of probabilistic programming, this corresponds to a situation where all particles will have the same values of the random variables sampled early in the program.

To mitigate the path degeneracy problem, more advanced SMC methods do not

resample at each time step. Common practice is to use the effective sample size (ESS), given by

$$\text{ESS} = \frac{(\sum_n w_{t-1}^n)^2}{\sum_n (w_{t-1}^n)^2},$$

and resample only if the ESS drops below a chosen threshold (e.g. $N/2$). If the ESS is above the threshold, no resampling takes place and the ancestry indices are set deterministically: $a_t^n = n$. The problem might be further mitigated by using different methods of sampling the ancestry indices, including stratified and systematic resampling (e.g. [47]).

Despite the path degeneracy problem, the bootstrap particle filter and other SMC methods are well established as automatic inference engines used in PPLs. The propagation and weighting steps are embarrassingly parallelizable, and can run on separate CPUs, GPUs or cluster nodes.

2

Birth-death models of evolution

2.1 Introduction

In 1859 Charles Darwin in his influential work *On the Origin of Species* [11] introduced the idea that all species have descended from one "primordial form", and that species evolve through natural selection and can split into separate lineages—a process known today as *speciation* (the term itself was coined first in 1906 by Orator F. Cook [9]). Together with *extinction*, when a species dies out, these ideas constitute the foundation of the theory of evolution, including phylogenetics, which focuses on the inference of the evolutionary history of species (and other taxonomic groups) and the relationships between them.

In this chapter we will introduce phylogenetic birth-death models, a family of continuous-time stochastic models of evolution in which the state, represented by all species living at a given time, can change in two possible ways:

- *speciation event* (representing birth), when one species splits into exactly two descendant species, and
- *extinction event* (representing death), when a species dies out.

The evolutionary process starts with a single species at some time $t_{\text{orig}} > 0$ in the past. A species undergoes a speciation after an exponentially distributed waiting time with the speciation rate λ, and goes extinct after an exponentially distributed waiting time with the extinction rate μ. These rates are inherited from the parent species at the speciation events (we will therefore also refer to these rates as *per-lineage* rates). In other words, starting at the origin, the speciation and extinction events follow a Poisson process with per-lineage rates λ and μ, and this process is "duplicated" at each speciation. Both descendants continue to follow their own independent processes. The speciation and extinc-

tion rates do not necessarily remain constant over time. In general, we expect that these rates may be subject to continuous or sudden changes that might affect a single species, but also a group of species or all species (e.g. in a mass extinction).

A couple of things require an explanation. First, we have used positive time for events in the past. Phylogeneticists often use *before present* (BP) time to denote how long ago something happened, for example, 8 *Ma ago* (*Mya* or *mya* are used as well) means 8 million years ago. There is a subtle difference between units of absolute time and relative time (i.e. duration), for example, 8 *Ma* (without "ago"), 8 *Myr* or 8 *myr* refer to a duration of 8 million years. Second, speciations are processes that take long time, rather than sudden events. However, considering the time scale of evolution, regarding speciations and extinctions as instantaneous events is a justified simplification.

2.2 Complete and surviving trees

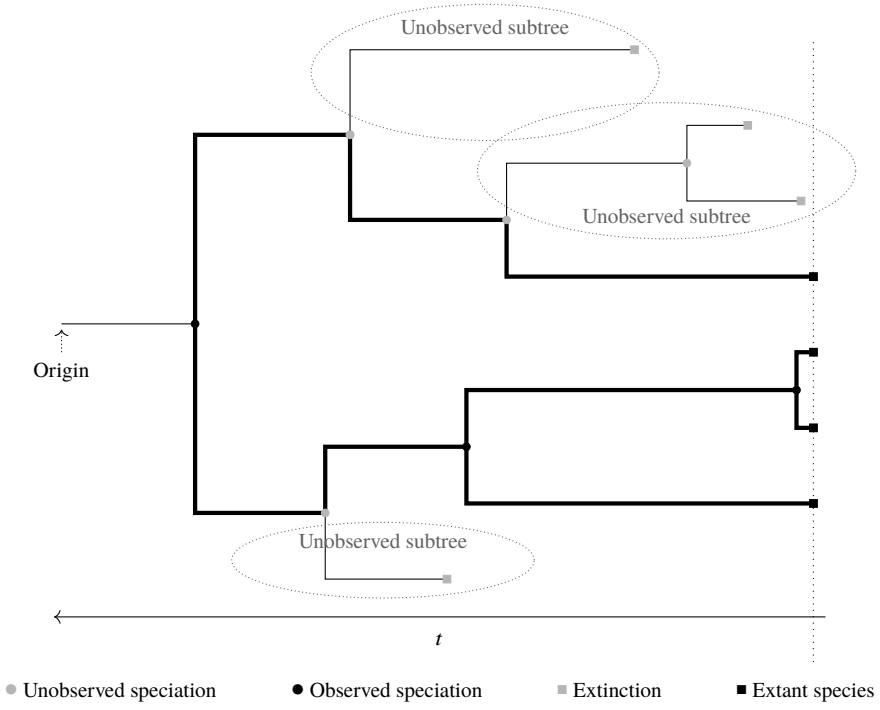
The result of a birth-death process (also called a diversification process) can be represented as a binary tree called a *phylogenetic tree*, or simply a *phylogeny*.

Its root represents the initial species at the time of origin t_{orig}; internal nodes represent speciation events; and leaves represent extinction events (if in the past, $t > 0$) and currently living—*extant*—species (if in the present time, $t = 0$). The edge length is defined as the difference between the times of the connected nodes.

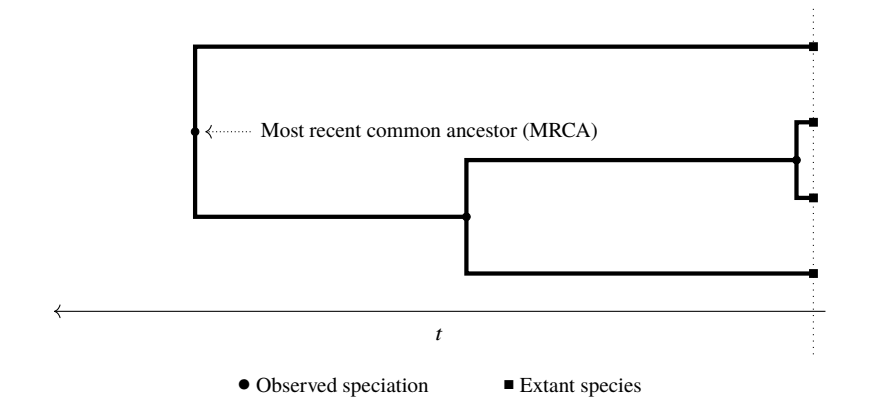
Figure 2.1a shows an example of a *complete* tree, representing the evolution of both the extant and extinct species. Note that the x-axis represents the time. Figure 2.1b depicts the corresponding *extant* tree, representing only the evolution of the extant species and their ancestors, with their *most recent common ancestor* (MRCA) at the root.

An example of a real surviving tree representing the evolution of cetaceans (whales, dolphins and porpoises) [57] is shown in Figure 2.2 (p. 36). Note that this tree is *labeled*—each extant species has a name. Surviving trees for taxonomic groups of interest are reconstructed from available data on the extant species, such as the genome sequences. Our goal in this thesis is to perform Bayesian inference of the model parameters given these reconstructed trees.

Let us introduce some necessary terminology: we will say that a speciation is *observed* if both descendants are ancestors of extant species, and *unobserved* or *hidden* otherwise. In Figure 2.1 we have used black circles to denote the observed speciations and gray circles to denote the unobserved ones.



(a) Complete tree.



(b) Surviving tree.

Figure 2.1: Example of a complete tree and its corresponding surviving tree.

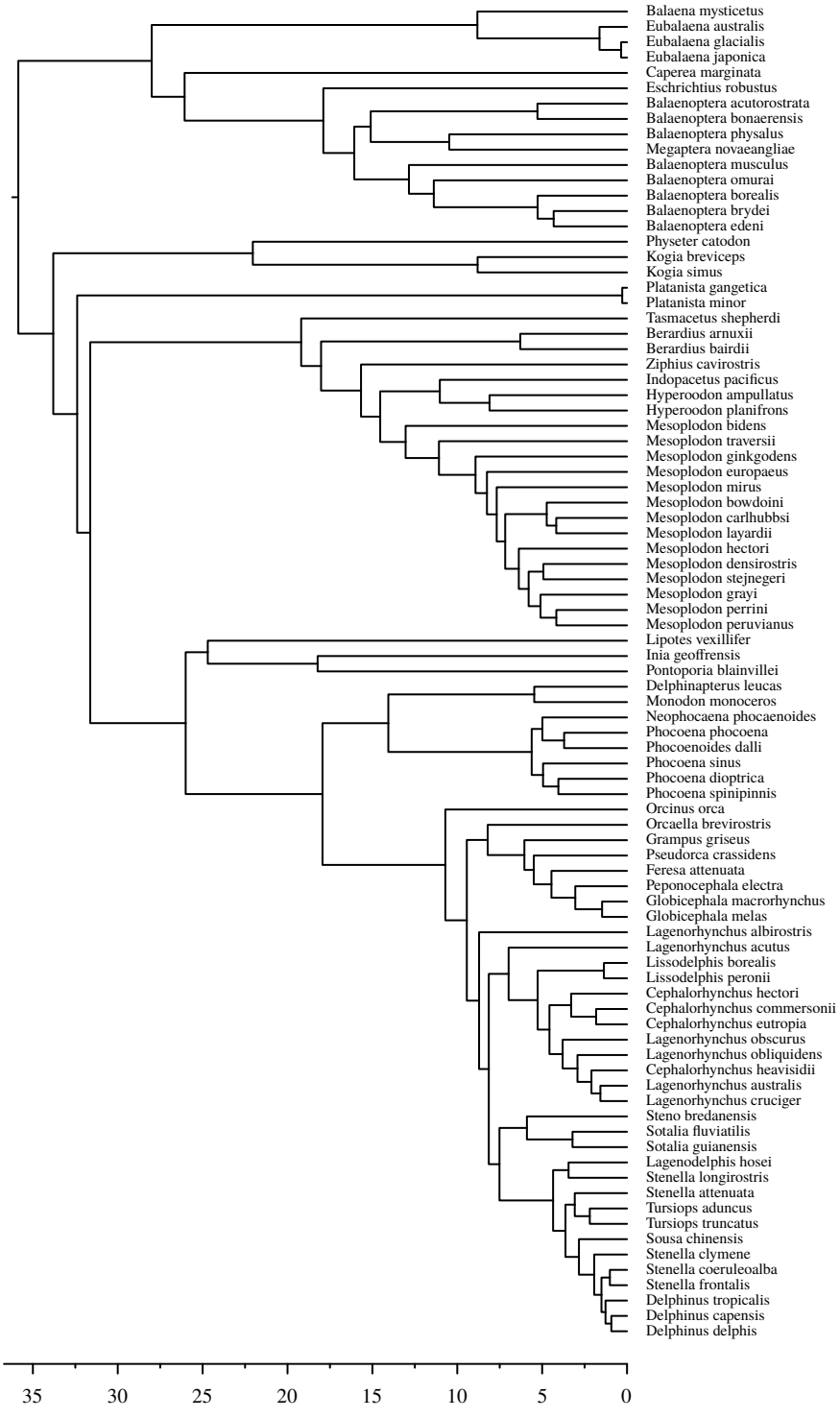


Figure 2.2: Extant phylogeny of cetaceans. Time axis in Mya.

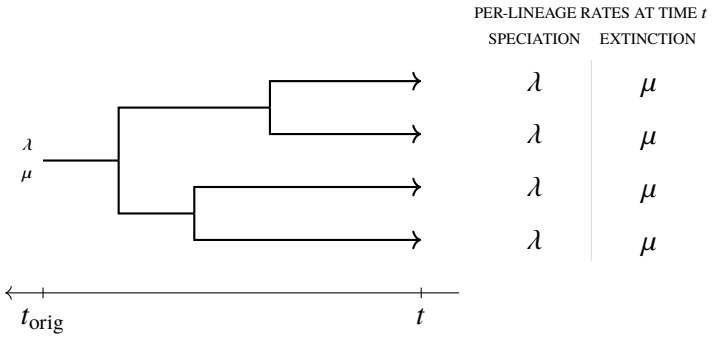


Figure 2.3: Per-lineage rates in the CRBD model.

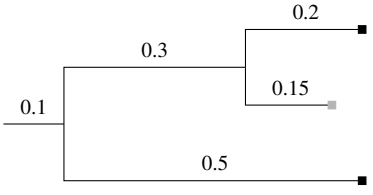
Similarly, we will say that a subtree of a complete tree is *unobserved* or *hidden* if it does not include any extant species.

Given a complete tree, the corresponding surviving tree can be obtained by *pruning*: the nodes of the surviving tree are simply the observed speciations and the extant species; an edge (also called a branch) in the surviving tree means that there exists a path between the corresponding nodes in the complete tree that can only go through unobserved speciations. In Figure 2.1a (p. 35) we have marked these paths by thick lines.

2.3 Constant-rate birth-death model

The constant-rate birth-death (CRBD) model [16] is the simplest diversification model. As the name reveals, both the speciation rate λ and extinction rate μ are constant over time (Figure 2.3).

The generative probabilistic program for the CRBD model is shown in Algorithm 2.1 (p. 38). The CRBD function accepts the time of origin t_{orig} as well as the rates λ and μ, and returns a complete tree as a recursive data structure of the NonEmptyTree type. This type, defined at the beginning of the program (lines 1–2), uses the constructors Extant, Extinction and Speciation




```
Speciation(0.1,
    Speciation(0.3,
        Extant(0.2),
        Extinct(0.15)),
    Extant(0.5))
```

Figure 2.4: Example of a simple tree and its data representation.

Algorithm 2.1: Generative constant-rate birth-death model.

```
1: type NonEmptyTree = Extant(Real) | Extinction(Real) |
2:                     Speciation(Real, NonEmptyTree, NonEmptyTree)
3: function CRBD(t, λ, μ)
4:     Δ ~ Exponential(λ + μ)
5:     if Δ > t then
6:         return Extant(t)
7:     end if
8:     s ~ Bernoulli(λ/(λ + μ))
9:     if s then
10:        return Speciation(Δ, CRBD(t − Δ, λ, μ), CRBD(t − Δ, λ, μ))
11:    else
12:        return Extinction(Δ)
13:    end if
14: end function
```

representing different types of nodes. An example of a simple tree and its data representation is depicted in Figure 2.4 (p. 37). The first parameter of each constructor represent the length of the edge between the node and its parent; the second and third parameters of Speciation represent the subtrees. Also the complete tree in Figure 2.1a (p. 35) was generated using this algorithm with $t = 2$, $\lambda = 1$, and $\mu = 0.5$.

Let us go through the CRBD function in more detail: We draw a waiting time Δ until the next event (be it a speciation or an extinction) from an exponential distribution with rate $\lambda + \mu$ (line 4). If the waiting time is longer than the starting time t, the species has survived to the present time, and we return Extant(t) (line 6). Otherwise we decide if the event is a speciation (rather than an extinction) by a draw from the Bernoulli distribution with probability $\lambda/(\lambda + \mu)$ (line 8). If so, we call the CRBD function twice—once for each descendant—to generate the subtrees starting at time $t - \Delta$, and return a Speciation node using Δ and both subtrees as parameters (line 10). If not, we return Extinct(Δ) (line 12).

Recall the first chapter; it is the possibility of unbounded recursion in computationally universal probabilistic programming languages that allowed us to encode the generative CRBD model as a very simple probabilistic program.

We should mention here that there also exist pure birth models, that is, models with no extinction events. For example, the *constant-rate birth* (CRB) model, also known as the *Yule* model [64], is the simplest pure birth model with constant speciation rate. We will however consider such models as special cases of birth-death models with the extinction rate μ set to 0.

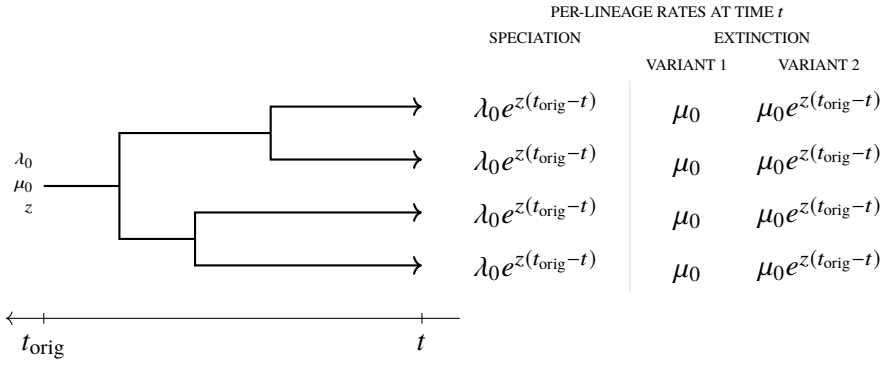


Figure 2.5: Per-lineage rates in the presented time-dependent birth-death model.

2.4 Selected non-constant-rate birth-death models

In this section we relax the assumption of the speciation and extinction rates being constant, and present a few models that we have been using in our experiments, and that allow continuous and/or sudden changes in the diversification rates.

2.4.1 Time-dependent birth-death models

Kendall [31] extended the CRBD model by allowing the speciation and extinction rates to be specified as functions of time. For example, consider the following function for the speciation rate:

$$\lambda(t) = \lambda_0 e^{z(t_{\text{orig}}-t)},$$

where λ_0 is the speciation rate at t_{orig}, and z is the trend parameter controlling how quickly the speciation rate might increase (when $z > 0$) or decrease (when $z < 0$). Choosing this particular form for the speciation rate is motivated by the empirical finding that diversification rates often slow down over time (e.g. [41]).

For the extinction rate we can consider two variants:

1. Keeping the extinction rate μ constant over time:

$$\mu(t) = \mu_0.$$

2. Keeping the turnover $\mu(t)/\lambda(t)$ constant over time:

$$\mu(t) = \mu_0 e^{z(t_{\text{orig}}-t)},$$

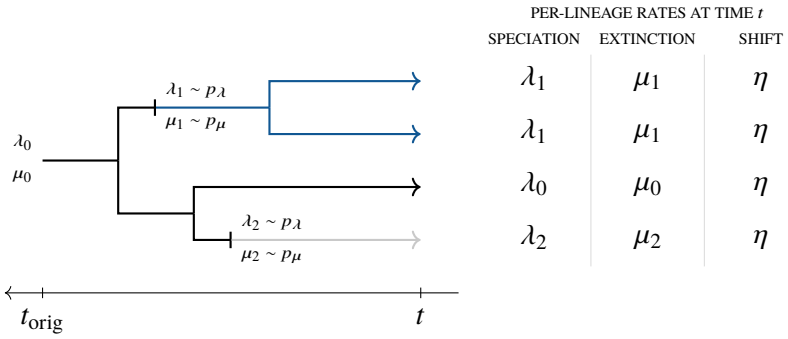


Figure 2.6: Per-lineage rates in the LSBDS model.

where μ_0 is the extinction rate at t_{orig}.

Figure 2.5 (p. 39) summarizes the rates for both variants.

2.4.2 Lineage-specific birth-death-shift model

The lineage-specific birth-death-shift (LSBDS) model by Höhna et al. [27] extends the CRBD model by introducing *diversification-rate shift* events. Like speciation and extinction events, also the shift events are related to a specific lineage (see Figure 2.6) and model sudden changes of the speciation and extinction rates.

Specifically, shift events follow a Poisson process with per-lineage rate η. Let λ_0 and μ_0 denote the initial speciation and extinction rate at the origin. At the i-th shift event the target species switches to a new process with the speciation and extinction rates drawn from the a priori chosen base distributions p_λ and p_μ:

$$\lambda_i \sim p_\lambda,$$
$$\mu_i \sim p_\mu.$$

At speciation events both descendants continue to follow the parent's process.

2.4.3 Bayesian analysis of macro-evolutionary mixtures

Bayesian analysis of macro-evolutionary mixtures (BAMM) by Rabosky [54] admits both exponentially increasing or decreasing speciation rates and rate shifts. The original model has been criticized in phylogenetic literature for its shortcomings by Moore et al. [42] and we follow their reinterpretation of the model.

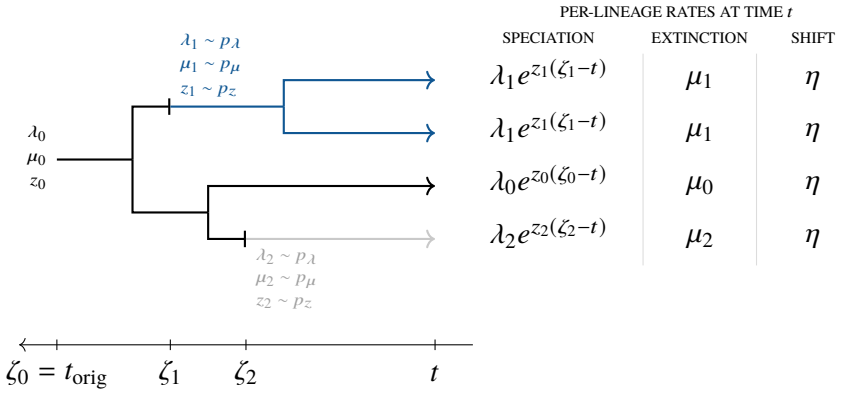


Figure 2.7: Per-lineage rates in the BAMM model.

Let λ_0, μ_0 and z_0 denote the initial values of the speciation rate, the extinction rate and the trend parameter at the origin $\zeta_0 = t_{\text{orig}}$. The shift events follow a Poisson process with per-lineage rate η. At the i-th shift event, occurring at time ζ_i, the target species changes to a new process (identified by i) with the parameters λ_i, μ_i and z_i drawn from the a priori chosen base distributions p_λ, p_μ and p_z:

$$\begin{aligned} \lambda_i &\sim p_\lambda, \\ \mu_i &\sim p_\mu, \\ z_i &\sim p_z. \end{aligned}$$

The speciation and extinction rates for a species following the process i are given by

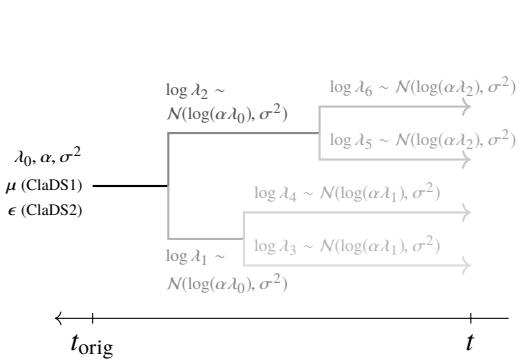
$$\begin{aligned} \lambda_i(t) &= \lambda_i e^{z_i(\zeta_i - t)}, \\ \mu_i(t) &= \mu_i. \end{aligned}$$

Figure 2.7 illustrates rate changes allowed by this model.

2.4.4 Cladogenetic diversification rate shift models

The cladogenetic diversification rate shift (ClaDS) models by Maliet et al. [36] are three models in which the speciation rate changes after each speciation event and remains constant until the next one. Each descendant draws its new speciation rate from a log-normal distribution:

$$\log \lambda \sim \mathcal{N}(\log(\alpha \lambda_p), \sigma^2),$$



PER-LINEAGE RATES AT TIME t

SPECIATION	EXTINCTION		
	ClaDS0	ClaDS1	ClaDS2
λ_6	0	μ	$\lambda_6\epsilon$
λ_5	0	μ	$\lambda_5\epsilon$
λ_4	0	μ	$\lambda_4\epsilon$
λ_3	0	μ	$\lambda_3\epsilon$

Figure 2.8: Per-lineage rates in the ClaDS models.

where λ_p is the speciation rate of the parent species, α and σ^2 are the model parameters modeling the trend and noise related to the speciation rate (Figure 2.8).

The three models differ in how they model the extinction rate:

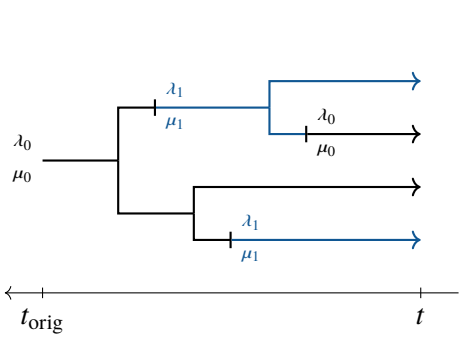
- ClaDS0 does not allow any extinctions ($\mu = 0$),
- in ClaDS1 the speciation rate μ is constant for all species, and
- in ClaDS2 the turnover $\epsilon = \mu/\lambda$ is constant for all species.

2.4.5 Birth-death models with state

An important class of phylogenetic birth-death models are models with state. The main idea behind these is associating species with a state variable (for example to indicate if a species is herbivore or carnivore), and letting the speciation and extinction rates depend on this state.

In the binary state speciation and extinction model (BiSSE) by Maddison et al. [35] the speciation and extinction rates depend on a binary state $s \in \{0, 1\}$, but otherwise remain constant over time: λ_0 and μ_0 for $s = 0$, and λ_1 and μ_1 for $s = 1$. The state of a species is inherited at speciation events. The state changes at rate ς_{01} when switching from state 0 to state 1, and ς_{10} when switching from state 1 to state 0 (Figure 2.9). The states are typically assumed to be observed for some or all of the extant species.

The BiSSE model inspired a lot of similar models, such as MuSSE (multiple state speciation and extinction model) [18], QuaSSE (quantitative state speciation and extinction model) [17] and GeoSSE (geographic state speciation and extinction model) [22].



PER-LINEAGE RATES AT TIME t

SPECIATION	EXTINCTION	SHIFT
λ_1	μ_1	$\varsigma 10$
λ_0	μ_0	$\varsigma 01$
λ_0	μ_0	$\varsigma 01$
λ_1	μ_1	$\varsigma 10$

Figure 2.9: Per-lineage rates in the BiSSE model.

2.5 Bayesian inference of the model parameters

Let us now return to the inference problem. We are interested in inferring the parameters of the model based on the observed reconstructed (surviving) tree. As mentioned in the previous section, for the models with state, the observation might also be annotated with the state for a subset of the extant species.

Let θ denote the parameters of the model, and T the reconstructed tree. We assume that T is known without any error. Further, let $p(\theta)$ denote the prior distribution for the parameters θ. Our goal is to estimate the posterior distribution for the parameter set θ given the observed tree T:

$$p(\theta|T) = \frac{p(T|\theta)p(\theta)}{p(T)},$$

where $p(T|\theta)$ denotes the likelihood of the observed tree T given the parameters θ, and $p(T)$, the marginal likelihood (also called the model evidence), is given by

$$p(T) = \int p(T|\theta)p(\theta)\mathrm{d}\theta.$$

One of the challenges we need to deal with is that, apart from a few of the simplest models, neither the posterior distribution nor the likelihood can be expressed in a closed form. In the next section we will show how to derive the likelihood for the CRBD model, which is one of these simple exceptions. Existing phylogenetic software, such as MrBayes [28], BEAST [15], BEAST 2 [6], RevBayes [29], RPANDA [43] or Diversitree [18], either implement only the models with analytical solutions or approximate the likelihood function e.g. by numerically solving differential equations describing the model, and use Markov chain Monte Carlo (MCMC) methods to sample from the posterior distribution.

In the next chapter, we will demonstrate how probabilistic programming with SMC based inference can be used to draw samples from the posterior distribution as well as to estimate the marginal likelihood $p(T)$ without bias. This is needed, for example, in analyses based on pseudo-marginal algorithms and for the model comparisons with Bayes factors.

2.6 Likelihood function for the CRBD model

As we have mentioned above, the constant-rate birth-death (CRBD) model is one of the few exceptions where we can express the likelihood of the observed tree in a closed form. In this section we derive the likelihood function, show the difference between the likelihood of an unlabelled oriented tree and the likelihood of a labelled unoriented tree, and conclude with a couple of possible approaches to dealing with an unknown time of origin.

Let us start with the extinction probability—the probability that an evolutionary process starting with a single species at time t goes extinct, or said differently, the probability that none of its ancestors survives to the present time.

For the sake of brevity, we will omit conditioning on $\theta = (\lambda, \mu)$ in the notation. Let $p_0(t)$ denote the extinction probability. This probability is a solution of the following differential equation:

$$\frac{\mathrm{d}p_0(t)}{\mathrm{d}t} = \mu + \lambda p_0(t)^2 - (\lambda + \mu)p_0(t),$$

with the boundary condition $p_0(0) = 0$.

To show that, let us consider what events can occur between $t + \Delta t$ and t, where $\Delta t \to 0$:

- an extinction event with probability $\mu\Delta t + O(\Delta t^2)$,
- a speciation event with probability $\lambda\Delta t + O(\Delta t^2)$, leading to two descendants, each going extinct with probability $p_0(t) + O(\Delta t)$,
- no events with probability $1 - (\lambda\Delta t + \mu\Delta t + O(\Delta t^2))$,
- more than one event with probability $O(\Delta t^2)$.

Putting everything together we get

$$\begin{aligned} p_0(t + \Delta t) = {} & \mu\Delta t \times 1 \\ & + \lambda\Delta t \times p_0(t)^2 \\ & + (1 - \lambda\Delta t - \mu\Delta t) \times p_0(t) \\ & + O(\Delta t^2), \end{aligned}$$

which can be rewritten as

$$\frac{p_0(t+\Delta t)-p_0(t)}{\Delta t} = \mu + \lambda p_0(t)^2 - (\lambda+\mu)p_0(t) + O(\Delta t).$$

Taking the limit as $\Delta t \to 0$ leads to the above-mentioned differential equation. The boundary condition is due to zero probability of an extant species going extinct:

$$p_0(0) = 0.$$

The solution of this differential equation is given by

$$p_0(t) = 1 - \frac{\lambda-\mu}{\lambda-\mu e^{-(\lambda-\mu)t}}.$$

Now, let $p(t,t')$ denote the probability that a species living at time t survives until time t', and that no observed speciation events occur during this time. This probability is a solution of the following differential equation:

$$\frac{\partial p(t,t')}{\partial t} = 2\lambda p_0(t)p(t,t') - (\lambda+\mu)p(t,t'),$$

with the boundary condition $p(t',t') = 1$.

To show that we will once again consider what events can occur between time $t+\Delta t$ and time t. The probability of no extinction events occurring in this interval is $1-\mu\Delta t + O(\Delta t^2)$, and

- no speciation event occurs with probability $1-\lambda\Delta t + O(\Delta t^2)$,
- one speciation event occurs with probability $\lambda\Delta t + O(\Delta t^2)$; in this case one of the *two* descendants must go extinct (probability of that being $p_0(t) + O(\Delta t)$) and the other must survive (probability of that being $p(t,t') + O(\Delta t)$),
- more than one speciation event occurs with probability $O(\Delta t^2)$.

Putting everything together we get

$$\begin{aligned} p(t+\Delta t,t') = {} & (1-\mu\Delta t)\times(1-\lambda\Delta t)\times p(t,t') \\ & + 2\times(1-\mu\Delta t)\times\lambda\Delta t\times p_0(t)\times p(t,t') \\ & + O(\Delta t^2), \end{aligned}$$

or, after rearranging the terms, the following equation:

$$\frac{p(t+\Delta t,t')-p(t,t')}{\Delta t} = 2\lambda p_0(t)p(t,t') - (\lambda+\mu)p(t,t').$$

Taking the limit as $\Delta t \to 0$ leads to the above-mentioned differential equation. At $t = t'$ the species has already survived to t', i.e.

$$p(t', t') = 1.$$

The solution of this equation is given by

$$p(t, t') = \frac{e^{-(\lambda-\mu)t}}{e^{-(\lambda-\mu)t'}} \frac{\left(\lambda - \mu e^{-(\lambda-\mu)t'}\right)^2}{\left(\lambda - \mu e^{-(\lambda-\mu)t}\right)^2}.$$

Let us now turn our attention to the surviving tree T. Let N denote the number of the extant nodes in T, and T' denote the tree obtained from T by adding the *stalk*—the edge from the origin to the most recent common ancestor (MRCA). To make the notation simpler, let us enumerate all nodes in T' by time (the order of the extant nodes is not important), using 0 for the origin at $t_0 = t_{\text{orig}}$, and $t_1 = t_{\text{MRCA}}, t_2, \ldots, t_{N-1}$ denoting the times for the internal nodes. The time t_i for all extant nodes $i \in \{N, \ldots, 2N-1\}$ is equal to 0.

For the sake of simplicity we will assume that phylogenetic trees are unlabelled (i.e. the leaves have no labels) and oriented, that is, it is possible to distinguish which child of an internal node is the "left" one and which is the "right" one. This is indeed a simplification: the extant species have been assigned names and phylogenies are unoriented—if we swap the subtrees of any node, the new tree still represents the same result of the evolutionary process. The likelihood of a labelled and unoriented tree can be easily derived from the likelihood of the unlabelled and oriented tree by multiplying it with a constant factor that depends only on the number of extant nodes (e.g. [I]):

$$\tilde{p}(T|\theta) = \frac{2^{N-1}}{N!} p(T|\theta),$$

where $\tilde{p}(T|\theta)$ denotes the likelihood of the labelled and unoriented tree. As this constant factor does not affect the posterior distribution, such an assumption is quite common in the phylogenetic literature, often even without being mentioned explicitly.

The likelihood $p(T'|\theta)$ of the extended tree T' is given by the product of the following factors:

- for each edge in T' from a parent node i to a child node j, the probability of the species surviving along the edge and not containing an observed speciation, given by

$$p(t_i, t_j) = \frac{e^{-(\lambda-\mu)t_i}}{e^{-(\lambda-\mu)t_j}} \frac{\left(\lambda - \mu e^{-(\lambda-\mu)t_j}\right)^2}{\left(\lambda - \mu e^{-(\lambda-\mu)t_i}\right)^2},$$

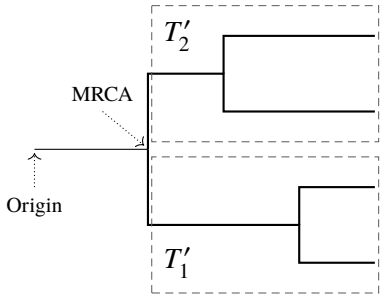


Figure 2.10: Descendants of the MRCA.

- for each internal node $i \in \{1, \ldots, N-1\}$, the probability of a speciation event at the end of the edge from the parent of i to i, given by λ.

Putting everything together we can express the likelihood of the extended surviving tree T' as follows:

$$p(T'|\theta) = \lambda^{N-1}(\lambda-\mu)^{2N} \prod_{i=0}^{N-1} \frac{e^{-(\lambda-\mu)t_i}}{\left(\lambda - \mu e^{-(\lambda-\mu)t_i}\right)^2}.$$

We have in the derivation of $p(T'|\theta)$ used a tree extended with the stalk, the edge from the origin to the MRCA, but we do not actually know t_{orig}. One option is to define the likelihood of T as follows:

$$p(T|\theta) = p(T_1'|\theta)p(T_2'|\theta),$$

where T_1' and T_2' denote the subtrees of T corresponding to the two descendants of the MRCA (the apostrophes are used to emphasize that both these subtrees have a stalk), see Figure 2.10.

A better option is to condition the posterior distribution on t_{MRCA} as well (e.g. [56]):

$$p(\theta|T, t_{\text{MRCA}}) = \frac{p(T|\theta, t_{\text{MRCA}})p(\theta)}{p(T|t_{\text{MRCA}})},$$

where the likelihood is given by

$$p(T|\theta, t_{\text{MRCA}}) = \frac{p(T_1'|\theta)}{S(\theta, t_{\text{MRCA}})} \frac{p(T_2'|\theta)}{S(\theta, t_{\text{MRCA}})}.$$

$S(\theta, t)$ denotes the *survival probability*—the probability that an evolutionary process starting at time t with a single species produces at least one extant species. The survival is complementary to the extinction, i.e. $S(\theta, t) + p_0(t) = 1$.

The model evidence $p(T|t_{\text{MRCA}})$ is in this case given by

$$p(T|t_{\text{MRCA}}) = \int p(T|\theta, t_{\text{MRCA}})p(\theta)\mathrm{d}\theta.$$

3

Probabilistic programming for phylogenetics

3.1 Parameter inference for the CRBD model

Recall the recipe for probabilistic modeling with probabilistic programming languages from the first chapter:

1. Write a generative model as a computer program.
2. Add observe statements.
3. Run the program to automatically do the inference.

For phylogenetic birth-death models, the challenge is related to the second point. Where and how exactly should we specify the observed tree? Can we just implement the generative CRBD model (Algorithm 2.1, p. 38) from the previous chapter, implement a function to prune a complete tree (to return its surviving tree), and condition on the surviving tree being equal to the observed tree (in the sense that it has the same topology and that all edge lengths are the same)? Unfortunately, the answer is no (at least at the time of writing this thesis). To understand why, consider the following example:

$x \sim \mathcal{N}(0, 1)$
... do something with x ...
observe $x = 0.1$

Since the variable x is continuous, the probability of x being equal to 0.1 (or any other value) is 0. This is indeed the reason why we need both the distribution and the observed value to be specified for continuous variables. Fortunately, we can fix this program rather easily:

$x \leftarrow 0.1$
observe $x \sim \mathcal{N}(0, 1)$
... do something with x ...

Can we fix the following program in a similar way?

$x \sim \mathcal{N}(0, 1)$
... do something with x ...
observe $f(x) = 0.1$

It depends on $f(x)$. For example, if it is an injective and differentiable function, we can use the change of variables technique to determine the distribution of $f(x)$. However, for more complex functions, including pruning, we need to choose a different method.

Our approach [II] is based on augmentation, i.e. obtaining a complete tree from the observed one. We traverse the observed tree, and for each edge we use the generative model to simulate all unobserved events: hidden speciations, parameter shifts, state shifts, etc. For each hidden speciation, we use the model also to simulate the evolution of extinct species (starting at the time of the hidden speciation).

As we traverse the observed tree, and augment it with unobserved events and subtrees, we condition on the following:

1. No extinction events occur along the observed edges.

2. Observed speciation events occur at the end of the corresponding edges.

3. No species survive to the present time during simulations of unobserved subtrees. This implies setting the importance weight to 0 (and stopping the execution) if any simulated species survives to the present time (otherwise the species would be observed and had to be a part of the observed tree).

4. We double the probability of the execution for each hidden speciation event. This is related to the fact that either of two descendants can correspond to the hidden subtree.

A probabilistic program for the CRBD model in pseudocode is shown in Algorithm 3.1. The program uses a specific probabilistic construct called **factor** to explicitly multiply the probability of the execution (i.e. the particle weight in the SMC based inference methods) by the given value. The factor construct is closely related to the observe construct, which multiplies the probability by the likelihood of the observed value with respect to the given distribution and

Algorithm 3.1: A probabilistic program for the CRBD model.

$\lambda \sim$ prior distribution for λ
$\mu \sim$ prior distribution for μ

for all $r \in$ nodes of T **do**
    if r is the root **then**
        continue
    end if
    $c_{\text{hs}} \sim \text{Poisson}(\lambda\Delta_r)$
    for $i \leftarrow 1$ **to** c_{hs} **do**
        $t \sim \text{Uniform}(t_r, t_r + \Delta_r)$
        if SURVIVES(t, λ, μ) **then**
            factor 0
        end if
        factor 2
    end for
    observe $0 \sim \text{Poisson}(\mu\Delta_r)$
    if r is a speciation **then**
        observe $0 \sim \text{Exponential}(\lambda)$
    end if
end for
return λ, μ

function SURVIVES(t, λ, μ)
    $\Delta \sim \text{Exponential}(\mu)$
    if $\Delta \geq t$ **then**
        return true
    end if
    $c_b \sim \text{Poisson}(\lambda\Delta)$
    for $i \leftarrow 1$ **to** c_b **do**
        $t' \sim \text{Uniform}(t - \Delta, t)$
        if SURVIVES(t', λ, μ) **then**
            return true
        end if
    end for
    return false
end function

its parameters. Note that we have not implemented conditioning on the time of the MRCA, we will return to this in Section 3.4 (p. 63).

Let us go through the program in more detail. After defining the rates (lines 1–2), we traverse all nodes of the observed tree T in a loop (line 3), for example in the depth-first manner. We skip the root node (since the stalk is not known) (line 4). The current node is denoted by r, the corresponding time by t_r, and the length of the edge to its parent by Δ_r. To sample the times of hidden speciation events along this edge we could sample a waiting time from an exponential distribution with rate λ, reduce the current time (initially $t_r + \Delta_r$) by the sampled time, and repeat this procedure until the current time falls below t_r (recall that we use the time before present).

We use an alternative approach: we sample the number of hidden speciation events c_{hs} along the edge from a Poisson distribution with rate $\lambda\Delta_r$ (line 7). Given the number of events, their times are distributed uniformly (line 9). For each hidden speciation we call the SURVIVES function (line 10). This function essentially implements the generative CRBD model to simulate the evolution of extinct species. It interrupts the simulation and returns true immediately if any species survives to the present time. In that case the execution probability is set to 0 (line 11). Note that the inference engine will terminate the execution at this point. If the simulation only led to extinct species (i.e. the SURVIVES function returned false), we double the execution probability (line 13) and continue to the next hidden speciation.

Finally, we observe that there were no extinction events along the current edge (line 15). More specifically, we use a Poisson distributed random variable with rate $\mu\Delta_r$ to represent the number of extinction events, and observe that this number is 0.

If the current node represents a speciation event (line 17), we use an exponentially distributed random variable with rate λ, representing the waiting time for the next speciation event (starting at time t_r, since the speciation events between $t_r + \Delta_r$ and t_r have been simulated earlier), and observe that this time is 0.

In the SURVIVES function, we first sample a waiting time Δ until the next extinction (line 22). If it is greater than the starting time t, the species has survived to the present time and we return true. Otherwise we sample the number of speciation events along a branch of length Δ (line 26). For each of them we sample the speciation time from a uniform distribution (line 28) and recursively call the SURVIVES function to simulate the evolution starting at that time.

With the exception of the CRBD model we need to iterate over the nodes in

an order which ensures that a node is processed before its descendants, due to the inheritance of the rates (and possibly the state). During our work on [I] we experimented with different orderings: we first generated a few random "permutations" for each observed tree by swapping the left and right subtrees of each node with probability 1/2. For each permutation we ran the program multiple times, using depth-first search, and collected estimates of the marginal likelihood. The variances of the estimates for different permutations of the same observed tree varied significantly, indicating that the traversal order had a significant effect on the quality of the inference. We also discovered that processing the subtree with the smaller total length (i.e. the sum of all edge lengths) first is an easy and effective heuristic to keep the variance low. In the future we plan to look at this question in more detail and try to find other heuristics or methods to determine the optimal traversal order.

In Section 1.4 (p. 24) on SMC based inference we resampled at each observe. Some programming languages (e.g. WebPPL) will do so also at each factor. Due to a random number of hidden speciations, this can create a misalignment of which part of the tree is currently processed by the particles, and lead to poor performance of the particle filter. In our implementation we resample the particles after processing each node of the observed tree. Details and comparison of unaligned and aligned resampling can be found in [I].

To estimate the posterior distribution of the rates λ and μ, we just need to collect the sampled values from all particles (line 20) and their weights. As the program sampled a complete tree during the execution, we can easily change the return statement in order to estimate the expected value of any test function of the rates and/or the complete tree.

The normalization constant Z coincides with the likelihood $p(T|\theta)$. This is indeed one of the very important advantages of our approach, since it enables comparison of different models using Bayes factors, as we have shown in [I]. The estimate of the normalization constant is unbiased, which makes it possible to use it in hierarchical inference, such as particle Markov chain Monte Carlo (PMCMC) [3]. If we also include the factor $2^{N-1}/N!$ (e.g. at the very beginning of the program), the normalization constant will represent the likelihood $\tilde{p}(T|\theta)$ of the labelled and unoriented tree.

Until now we have assumed ideally reconstructed trees that include all extant species. This is not always the case in real life. In birth-death models, *incomplete sampling* of extant species (when some of the extant species and their ancestors are missing in the reconstructed tree) can be easily modeled in several ways, including *uniform sampling* (e.g. [55]): each species at the present time $t = 0$ is included in the reconstructed tree with probability ρ. Algorithm 3.2 (p. 55) shows how to extend the program to support this model of incomplete

sampling. The parts that are different from Algorithm 3.1 are shown in blue.

All ideas and methods we have introduced in this section and demonstrated for the CRBD model are applicable to any birth-death model. Of course, the programs need to handle other types of events as well, implement the formulas calculating the speciation and extinction rates, pass all necessary information from a parent node to its descendants, etc.

One of the non-trivial implementation details is sampling of the event times of a non-homogenous Poisson process. Let $\nu(t)$ denote the rate of the process and Δ its duration. If there exists ν_0 such that $\nu_0 \geq \nu(t)$ for all $t \in (0, \Delta)$, we can use the *thinning* algorithm [34]:

$t \leftarrow 0$
while $t < \Delta$ **do**
    $\delta \sim \text{Exponential}(\nu_0)$
    $t \leftarrow t + \delta$
    if $t < \Delta$ **then**
        $\alpha \sim \text{Bernoulli}(\nu(t)/\nu_0)$
        if α **then**
            yield t
        end if
    end if
end while

For decreasing $\nu(t)$ it is convenient to set ν_0 at the beginning of each loop iteration to $\nu(t)$.

Our implementation of the CRBD model (and other models from Chapter 2) in Birch can be found at https://github.com/phyppl/probabilistic-programming.

3.2 Alive particle filter

While simulating the hidden subtrees, if any of the species survives to the present time, we set the weight to zero and terminate the execution. These executions impoverish the particle set: there are fewer particles (with non-zero weight) to choose from at the next resampling checkpoint. But is this a problem at all? How often does a simulation of a hidden subtree result in setting the weight to zero? For the CRBD model, this probability is given by $1 - p_0(t)$, where $p_0(t)$ is the extinction probability introduced in Section 2.6 (p. 44):

$$1 - p_0(t) = \frac{\lambda - \mu}{\lambda - \mu e^{-(\lambda-\mu)t}}.$$

Algorithm 3.2: A probabilistic program for the CRBD model with uniform incomplete sampling.

$\lambda \sim$ prior distribution for λ
$\mu \sim$ prior distribution for μ
for all $r \in$ nodes of T **do**
  if r is the root **then continue end if**
  $c_{\text{hs}} \sim \text{Poisson}(\lambda \Delta_r)$
  for $i \leftarrow 1$ **to** c_{hs} **do**
    $t \sim \text{Uniform}(t_r, t_r + \Delta_r)$
    if SURVIVES(t, λ, μ) **then**
      factor 0
    end if
    factor 2
  end for
  observe $0 \sim \text{Poisson}(\mu \Delta_r)$
  if r is a speciation **then**
    observe $0 \sim \text{Exponential}(\lambda)$
  end if
  if r is an extant species **then**
    observe true $\sim \text{Bernoulli}(\rho)$
  end if
end for
return λ, μ
function SURVIVES(t, λ, μ)
  $\Delta \sim \text{Exponential}(\mu)$
  if $\Delta \geq t$ **then**
    $o \sim \text{Bernoulli}(\rho)$
    if o **then**
      return true
    end if
    $\Delta \leftarrow t$
  end if
  $c_b \sim \text{Poisson}(\lambda \Delta)$
  for $i \leftarrow 1$ **to** c_b **do**
    $t' \sim \text{Uniform}(t - \Delta, t)$
    if SURVIVES(t', λ, μ) **then return** true **end if**
  end for
  return true
end function

Algorithm 3.3: Alive particle filter (APF) for probabilistic programming.

for $n = 1, \ldots N$ **do**
  $w_0^n \leftarrow 1$
  $x_0^n \leftarrow \varnothing$ ▷ Initialization
end for
for $t = 1, \ldots T$ **do**
  $P_t \leftarrow 0$
  for $n = 1, \ldots N + 1$ **do**
    repeat
      $a_t^n \sim \text{Categorical}(\{w_{t-1}^m\}_{m=1}^M)$ ▷ Resampling
      $x_t^n \leftarrow \textsc{Propagate}(x_{t-1}^{a_t^n})$ ▷ Propagation
      $P_t \leftarrow P_t + 1$
      $w_t^n \leftarrow p_{\gamma[t]}(y_t \mid \text{Pa}(x_t^n))$ ▷ Weighting
      $x_t^n \leftarrow x_t^n \cup \{(\gamma[t], y_t)\}$
    until $w_t^n > 0$
  end for
end for
for $n = 1, \ldots N$ **do**
  $h^n \leftarrow h(x_T^n)$
end for
$\mathbb{E}[h] \approx \sum_n w_T^n h^n / \sum_n w^n$

For most models, this probability does not have a closed form (that is the reason why we use the generative model to simulate the hidden subtrees). Taken into account that there might be several hidden speciations along a single edge, the fraction of particles that have zero weight at a resampling checkpoint might be quite large.

In [II] we proposed a solution based on an *extended alive particle filter* (APF). This inference algorithm replaces every particle that has zero weight by returning to the previous time step and repeating the resampling and propagation steps. The procedure is repeated until all particles have positive (i.e. non-zero) weight. The APF needs to use one extra particle (compared to the bootstrap particle filter), but with a reasonable number of particles this cost is negligible. Algorithm 3.3 summarizes the algorithm (differences between this algorithm and the bootstrap particle filter are shown in blue).

The estimator of the marginal likelihood is given by

$$\widehat{Z} = \prod_{t=1}^{T} \frac{\sum_{n=1}^{N} w_t^n}{P_t - 1},$$

where P_t is the total number of the propagations at time step t, including the propagations for rejected particles and the propagations for the extra particle.

More details, including the proof of the unbiasedness of the estimator of the marginal likelihood can be found in [II]. There we also present experiments comparing the performance of the alive particle algorithm with the bootstrap particle filter for the CRBD and BiSSE models.

In [IV] we present a similar algorithm, *particle filter with rejection control* (PF-RC), in which the weights of the particles are compared to given thresholds (rather than to 0 as in the APF). Particles with weights below the chosen thresholds, are probabilistically discarded (with probabilities given by the weights of the particles divided by the threshold) and the resampling and propagation steps are repeated. We have demonstrated the algorithm with a couple of examples not related to phylogenetics, but believe that this algorithm might be useful for the phylogenetic birth-death models too. The question that needs to be investigated is how to determine the thresholds. An interesting incentive for using this algorithm for phylogenetic models is that it enables the rejection of some particles even before simulating the hidden subtrees. For each node, we can simulate the number of hidden speciation events and update the weight, assuming that no species survive while simulating the hidden subtrees, and reject the particles based on the updated weights. For particles that are (tentatively) accepted, we can simulate the hidden subtrees, and reject those where the assumption is not met (i.e. any species survives to the present time).

3.3 Delayed sampling

Recall the following example in Birch from the first chapter:

```
x:Random<Real>;
y:Random<Real>;

x ~ Gaussian(0, 1);
y ~ Gaussian(x, 1);
stdout.print("y = " + y.value() + "\n");
```

As we mentioned there, the variable x is never realized during the execution of the program. From a mathematical point of view, this makes perfect sense. The program specifies the joint distribution $p(x, y)$, and if sampled immediately, we would first sample x from $\mathcal{N}(0, 1)$, and then y, based on the value of x, from $p(y|x)$, which is $\mathcal{N}(x, 1)$. Due to the conjugacy relationship between x and y there is a closed form solution for both $p(y)$ and $p(x|y)$:

$$p(y) = \int_{-\infty}^{\infty} p(y|x)p(x)dx = \mathcal{N}(y|0, 2)$$

$$p(x|y) = \mathcal{N}(x|y/2, 1/2)$$

We can sample y from $\mathcal{N}(0, 2)$ first, and then sample x from $\mathcal{N}(y/2, 1/2)$. We are indeed getting samples from the same joint distribution encoded by the program, since $p(x, y) = p(x)p(y|x) = p(y)p(x|y)$.

Delayed sampling, introduced in [III], exploits analytically tractable relationships between random variables in probabilistic programs automatically in order to lower the variance of the resulting estimators. Programmers do not need to implement any of the analytical relationships (or even realize that there actually are some).

When executing a program, a directed acyclic graph, or more precisely a forest of trees, is built dynamically, with the nodes representing the encountered random variables and the edges representing the analytical relationships between them. Each node and the corresponding variable can be in one of the three states:

- *initialized* (the random variable has been inserted into the graph but not processed further),
- *marginalized* (the random variable has been marginalized over its parents and potentially conditioned on the value of the dependent variable),
- *realized* (the random variable has been sampled or it is observed).

Nodes in the marginalized state have a proposal distribution associated with them, which is used when realizing the random variables. The method does not allow marginalized nodes to have more than one marginalized child. This means that the marginalized nodes in each tree form a path, which we will refer to as the *M-path*. The M-path of a tree starts at the root node (which is always in the marginalized state) and ends in a node referred to as the *terminal* node.

Each random variable encountered during the execution is inserted into the graph. If there is an analytically tractable relationship to another variable, it is inserted as a child of the node corresponding to that variable and set to be in the initialized state, otherwise it forms a new tree consisting of a single node in the marginalized state (its proposal distribution is the distribution specified by the program).

For a random variable to be realized, the corresponding node must be the terminal node on an M-path. If it already is, the value is sampled from (resp. observed with respect to) its proposal distribution, and the state is changed to realized. The edges to the children are removed, and each child becomes the marginalized root of a new tree (the proposal distribution is given by the

program and the sampled value). The parent of the realized node (if there is one) becomes the new terminal node of the M-path and its proposal distribution is conditioned on the realized value.

If the target node corresponding to the random variable being realized is not a terminal node, the following graph operations must be performed before we can use the above-mentioned procedure:

- If the node lies on the M-path: the M-path is shortened by repeatedly realizing its terminal node until the target node becomes terminal (this operation is called *pruning*).
- If the node does not lie on the M-path: we first use pruning to shorten the M-path until any of the target node's ancestors becomes terminal, and then extend the M-path towards the target node by repeatedly marginalizing the initialized variables over their parents (this operation is called *grafting*).

Specific details of these operations, including the pseudocode, as well as a comparison of the performance of immediate and delayed sampling can be found in [III].

To illustrate these operations, consider the following program:

$a \sim \mathcal{N}(0, 1)$
$b \sim \mathcal{N}(a, 1)$
$c \sim \mathcal{N}(b, 1)$
$d \sim \mathcal{N}(b, 1)$
$e \sim \mathcal{N}(c, 1)$
$f \sim \mathcal{N}(d, 1)$
PRINT(e)
PRINT(d)

Figure 3.1 (p. 60) shows different states of the graph during execution. Figure (a) corresponds to the state before printing the value of e.

When the value of e needs to be sampled, we first marginalize b, then c and finally e, in order to form the M-path from the root a to the node e. The state of the graph at this moment is depicted in Figure (b) together with the proposal distributions.

When e gets sampled from q_e, the sampled value, say 1.2, is used to update the proposal distribution q_c. The state of e is changed to realized, and the edge between c and e is removed. The node c becomes the new terminal node. This state corresponds to Figure (c).

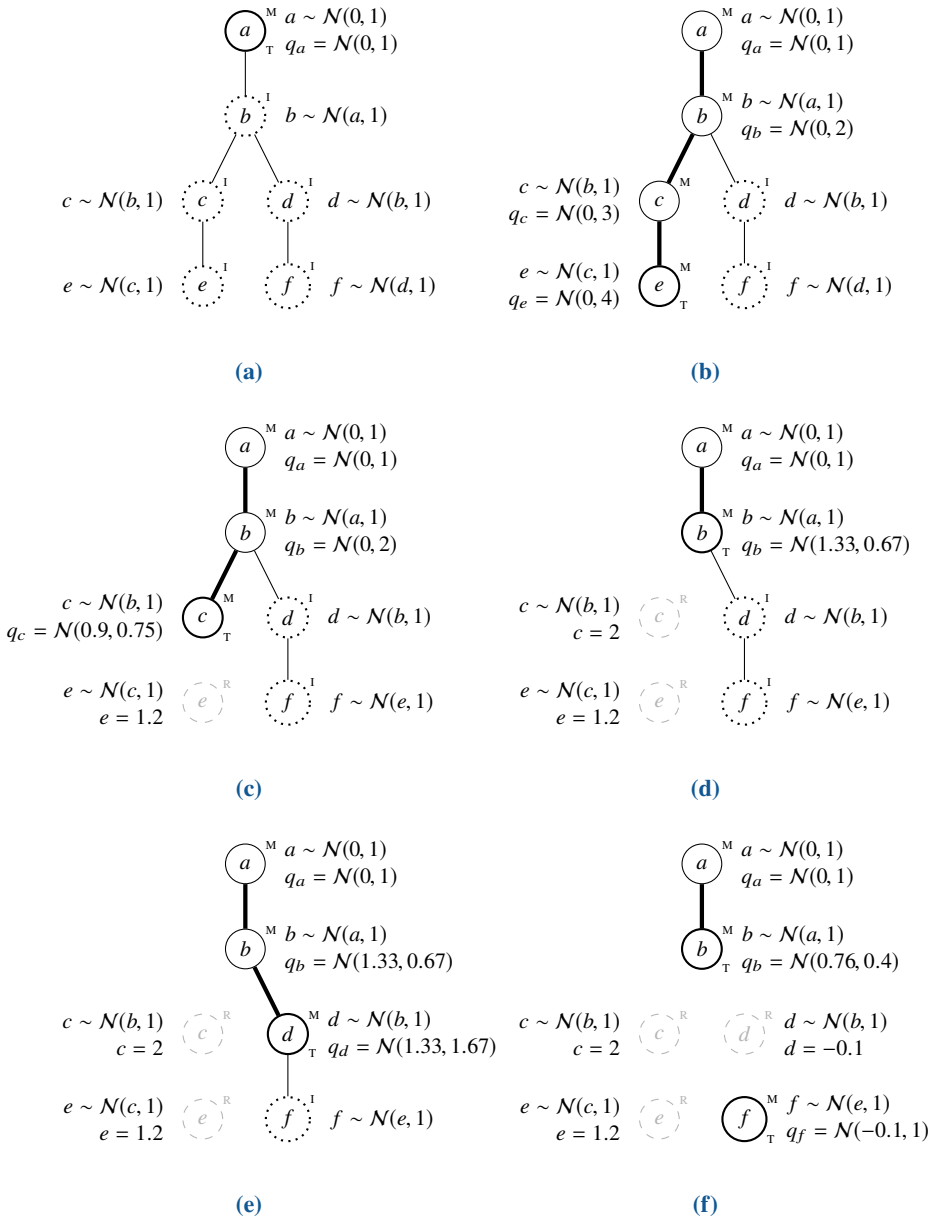


Figure 3.1: Illustration of graph operations in delayed sampling. See the description in the text. Initialized nodes are shown with dotted circles and the letter I, marginalized nodes with solid circles and the letter M, and realized nodes with dashed gray circles and the letter R. The letter T denotes terminal nodes on M-paths, which are shown with thick lines.

Next we wish to print the value of d. Before we can sample it, we need d to become a terminal node. We first shorten the M-path by sampling c (and the sampled value, say 2, is used to update the proposal distribution q_b), see Figure (d), and then extend it to d by marginalization of this variable, see Figure (e). At this moment we can sample the value of d, and use the value, say -0.1, to update the proposal distribution of b. The node for f becomes the root of a new tree, and the proposal distribution is set based on the sampled value.

Let us now return to the phylogenetic birth-death models, and show how delayed sampling can be employed in these models [II]. It is mathematically convenient to use gamma distributions as priors for both λ and μ, since the gamma distribution is a conjugate prior for both the exponential and Poisson likelihood:

$$\lambda \sim \text{Gamma}(k_\lambda, \theta_\lambda),$$
$$\mu \sim \text{Gamma}(k_\mu, \theta_\mu).$$

Note that we use the shape/scale parametrization for gamma distributions.

Let us go through the probabilistic constructs involving the speciation and extinction rates in Algorithm 3.1 (p. 51) and Algorithm 3.2 (p. 55):

1. Sampling the number of speciation events c along an edge of length Δ:

   $$c \sim \text{Poisson}(\lambda\Delta).$$

   In the delayed sampling setting, if λ is marginalized and its proposal distribution is

   $$q_\lambda = \text{Gamma}(k, \theta),$$

   the value of c is sampled from the marginal distribution

   $$q_c = \text{NegativeBinomial}\left(k, \frac{1}{1 + \Delta\theta}\right),$$

   where the sampled value represents the number of failures. (The parameterization used here is the number of successes before the experiment is stopped, and the success probability.)

   The proposal distribution for λ is then updated, based on the sampled value c, to

   $$q_\lambda \leftarrow \text{Gamma}\left(k + c, \frac{\theta}{1 + \Delta\theta}\right),$$

   that is, its shape gets incremented by the sampled value, and the scale is updated to $\theta/(1 + \Delta\theta)$.

2. Observing no extinction events along an edge of length Δ:

   observe $0 \sim \text{Poisson}(\mu\Delta)$

   Similarly, if μ is marginalized with the proposal distribution being

   $$q_\mu = \text{Gamma}(k, \theta),$$

   the importance weight is multiplied by the probability mass of 0 with respect to the proposal distribution q_c of a random variable representing the number of extinction events along the edge with length Δ:

   $$q_c = \text{NegativeBinomial}\left(k, \frac{1}{1 + \Delta\theta}\right).$$

   The proposal distribution for μ is then updated to

   $$q_\lambda \leftarrow \text{Gamma}\left(k, \frac{\theta}{1 + \Delta\theta}\right),$$

   that is, its shape remains the same, and the scale gets updated to $\theta/(1 + \Delta\theta)$.

3. Sampling a waiting time Δ from an exponential distribution with rate μ:

   $$\Delta \sim \text{Exponential}(\mu).$$

   Again, we assume that the proposal distribution for μ is

   $$q_\mu = \text{Gamma}(k, \theta),$$

   and sample the value of Δ from

   $$q_\Delta = \text{Lomax}\left(\frac{1}{\theta}, k\right),$$

   where we have used the scale/shape parametrization of the Lomax distribution (see Appendix A, p. 79).

   The proposal distribution for μ is afterwards updated to

   $$q_\mu \leftarrow \text{Gamma}\left(k + 1, \frac{\theta}{1 + \Delta\theta}\right),$$

   that is, its shape gets incremented by 1, and the scale gets updated to $\theta/(1 + \Delta\theta)$.

4. Finally, observing the speciation event at the end of an observed edge:

   observe $0 \sim \text{Exponential}(\lambda)$.

   Starting with the proposal distribution of λ,

   $$q_\lambda = \text{Gamma}(k, \theta),$$

   the importance weight is multiplied by the probability density of 0 with respect to the proposal distribution q_Δ of a random variable representing the waiting time:

   $$q_\Delta = \text{Lomax}\left(\frac{1}{\theta}, k\right).$$

   The proposal distribution of λ gets updated to

   $$q_\lambda = \text{Gamma}(k + 1, \theta),$$

   that is, its shape gets incremented by 1, and the scale remains the same.

All four situations are summarized in Figure 3.2 (p. 64).

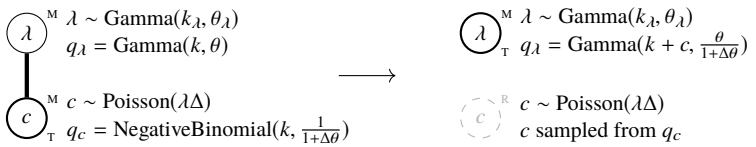
Interestingly, the consequence of employing delayed sampling in Algorithm 3.1 (p. 51) and Algorithm 3.2 (p. 55) is that both λ and μ remain delayed even when the program finishes. This allows us to estimate the posterior distribution of λ resp. μ as a mixture of the particles' proposal distributions at the end of the execution. The mixture weights are just the normalized final weights of the particles.

More details about using delayed sampling with the alive particle filter (APF) for phylogenetic birth-death models, including a comparison to immediate sampling with the bootstrap particle filter (BPF) can be found in [II]. The combination of APF with delayed sampling has also been used to run the inference in Birch in the experiments in [I].

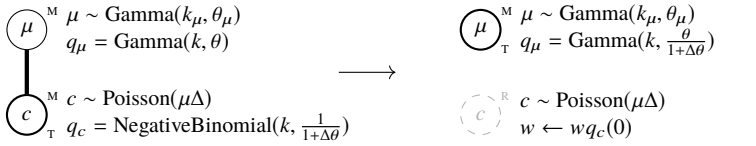
3.4 Conditioning on the time of the most recent common ancestor

At the end of Chapter 2 we considered two options for dealing with the fact that the time of origin is not known, or in other words, that the observed tree does not include the stalk. The first option was to calculate the likelihood of the observed tree as the product of the likelihoods of its two subtrees, i.e.

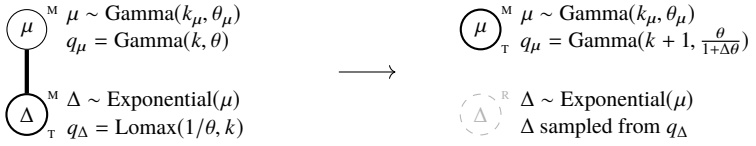
$$p(T|\theta) = p(T_1'|\theta)p(T_2'|\theta).$$



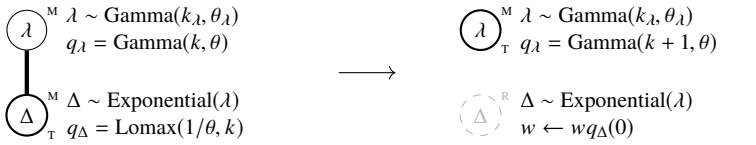
(a) Sampling c from Poisson($\lambda\Delta$).



(b) Observing 0 with respect to Poisson($\mu\Delta$).



(c) Sampling Δ from Exponential(μ).



(d) Observing 0 with respect to Exponential(λ).

Figure 3.2: Delayed sampling in the phylogenetic birth-death models.

This option is exactly what we used so far in this chapter. We will now look closer at the second option: conditioning the posterior distribution on the time of the MRCA. Recall that the likelihood is in this case given by

$$p(T|\theta, t_{\text{MRCA}}) = \frac{p(T_1'|\theta)p(T_2'|\theta)}{S(\theta, t_{\text{MRCA}})^2}.$$

This implies that we need to divide the particle weight (the execution probability) by $S(\theta, t_{\text{MRCA}})^2$. For the CRBD model the survival probability is known analytically, but as mentioned before, this is not the case for most birth-death models.

In [I] we proposed a solution that does not require the survival probability to be known. After traversing the tree, we use the generative model to simulate two independent evolutionary processes starting at t_{MRCA}, and consider the outcome successful if both processes survive to the present time. We repeat this procedure until the first success, and multiply the weight of the particle by the number of trials (up to and including the successful one). This method is based on the fact that $1/S(\theta, t_{\text{MRCA}})^2$ is the expected value of the number of trials of a geometric distribution with the success probability $S(\theta, t_{\text{MRCA}})^2$:

$$\frac{1}{S(\theta, t_{\text{MRCA}})^2} = \sum_{M=1}^{\infty} M(1 - S(\theta, t_{\text{MRCA}})^2)^{M-1} S(\theta, t_{\text{MRCA}})^2.$$

The updated probabilistic program for the CRBD model is shown in Algorithm 3.4 (p. 66).

Algorithm 3.4: A probabilistic program for the CRBD model with the survivorship bias correction.

$\lambda \sim$ prior distribution for λ
$\mu \sim$ prior distribution for μ

for all $r \in$ nodes of T **do**
    if r is the root **then**
        continue
    end if
    $c_{\text{hs}} \sim \text{Poisson}(\lambda \Delta_r)$
    for $i \leftarrow 1$ **to** c_{hs} **do**
        $t \sim \text{Uniform}(t_r, t_r + \Delta_r)$
        if not GoesExtinct(t, λ, μ) **then**
            factor 0
        end if
        factor 2
    end for
    observe $0 \sim \text{Poisson}(\mu \Delta_r)$
    if r is a speciation **then**
        observe $0 \sim \text{Exponential}(\lambda)$
    end if
end for
$M \leftarrow 1$
while GoesExtinct$(t_{\text{MRCA}}, \lambda, \mu)$ or GoesExtinct$(t_{\text{MRCA}}, \lambda, \mu)$ **do**
    $M \leftarrow M + 1$
end while
factor M
return λ, μ

function GoesExtinct(t, λ, μ)
    $\Delta \sim \text{Exponential}(\mu)$
    if $\Delta \geq t$ **then**
        return false
    end if
    $c_b \sim \text{Poisson}(\lambda \Delta)$
    for $i \leftarrow 1$ **to** c_b **do**
        $t' \sim \text{Uniform}(t - \Delta, t)$
        if not GoesExtinct(t', λ, μ) **then**
            return false
        end if
    end for
    return true
end function

4

Conclusion

This thesis is based on a collection of papers that introduce probabilistic programming as a completely new approach to parameter inference for phylogenetic birth-death models, and prove the feasibility of this approach. The birth-death models can be written as simple programs in probabilistic programming languages, and benefit from automatic inference which is an integral part of these languages. These programs are simple enough to allow biologists with just a basic understanding of programming to prototype, test and employ new models quickly, and without the need to derive and implement a bespoke inference algorithm.

With automatic inference based on sequential Monte Carlo methods, our approach, unlike the existing ones, also allows the model evidence to be estimated without bias. This in turn enables the employment of hierarchical inference algorithms as well as comparing different models.

The contribution to the automatic inference methods, namely delayed sampling and the extended alive particle filter, are not only relevant for phylogenetics, but for probabilistic programming in general. We have implemented well-known phylogenetic models and shown how the above-mentioned methods improve the quality of inference for these models.

Future work

Our work is not finished by any means. We have already mentioned some of the possible future directions in the previous chapter, namely the rejection control particle filter (and the question of how to determine its thresholds) and the question of finding the optimal traversal order of the observed tree.

In order to be able to work with big trees (with thousands of extant species) we

need to make the inference even more efficient (in terms of lowering the variance of the estimators and making the inference faster). This might include extending the existing inference algorithms as well as developing new algorithms and heuristics. Solving these problems is relevant not only for phylogenetics, but also for automatic inference in probabilistic programming languages in general.

In the present work we have assumed that the observed tree is known and mentioned briefly that it is reconstructed from such data as morphological traits and genomic data. We would like to join these two processes into one and employ probabilistic programming to infer both the tree and the parameters directly from these data.

Another interesting direction is solving the problem introduced in the beginning of the previous chapter: rather than implementing a generative program and just use a single observe, we needed to implement plenty of "partial" observes. How can we get PPLs to do this automatically?

We hope that our work will encourage other researchers and software developers to join us on our path towards the ultimate goal: to give computational phylogeneticists a set of tools that will enable them to think big about new models and reduce the time needed from an initial idea to running efficient simulation and inference for these models.

5

Acknowledgments

First, I would like to express my gratitude to my fantastic supervisors: Johannes Borgström, Thomas B. Schön and Lawrence M. Murray. A very special thanks to Lawrence: although not initially one of my supervisors, he always found time for me. Our long sessions, brainstorming and discussing ideas, were probably the most exciting time at Uppsala University for me.

A lot of the work included in this thesis has been done in cooperation with researchers from KTH Royal Institute of Technology and Swedish Museum of Natural History. I have always looked forward to our fortnightly meetings in Kista. Big thank you, Daniel Lundén, David Broman, Fredrik Ronquist, Viktor Senderov and Nicolas Lartillot! In addition to my supervisors, Fredrik and Victor also provided valuable feedback on the introductory chapters of this thesis.

Thanks to all my colleagues at the department, especially to the members of the concurrency group: Björn Victor, Lars-Henrik Eriksson, Arve Gengelbach, Tjark Weber, Anke Stüber and Joachim Parrow, as well as the former members Johannes Åman Pohjola, Ramunas Gutkovas and Sophia Knight; and to the members of the the Assemble group.

Thanks to Matteo Magnani for the experience to teach with him, and for the opportunity to take over the Database Design I course after he became the director of the undergraduate studies.

Thanks to all my unfailing lunch buddies: Daniel Kovacs and his colleagues from the chemistry department, Jan Rusz, Lucia Komendova, Jorge Cayao, Arve and Anke. Thanks to all the friends I have got in Uppsala and who made these five years memorable!

Thanks to everybody who helped me, taught me something new and useful, or made me laugh, and whom I might have forgotten!

My work was supported by the Swedish Research Council via the grant no. 2013-4853 and by the Swedish Foundation for Strategic Research (SSF) via the project ASSEMBLE (contract number RIT15-0012).

Last but definitely not least, I would like to thank Viera, my wife, for all her support.

6

Summary in Swedish

Fylogenetiska födelse-/döds-modeller är en familj av matematiska modeller av evolution där man betraktar *speciering* (när populationen av en art delas upp i två grupper och så småningom bildar två nya arter) och *utrotning* (när hela populationen av en art dör ut) som plötsliga händelser som inträffar vid slumpmässiga tidpunkter. Med tanke på hur lång evolutionen är är ju en sådan förenkling acceptabel. Tiden det tar for en art att genomgå en speciering, respektive tills de utrotas, modelleras som exponentialfördelade slumpmässiga variabler. I de flesta modeller är hastighetsparametrarna inte konstanta: det kan handla om både kontinuerliga och plötsliga förändringar som påverkar en enskild art, grupper av arter eller alla arter. Vilka förändringar som är tillåtna och på vilket sätt dessa modelleras beror på de konkreta födelse-/döds-modellerna. Mer avancerade modeller kan också innehålla olika tillstånd för arter (t.ex. om en art är monogam eller inte) samt modellera deras förändringar.

Att använda fylogenetiska födelse-/döds-modeller för att simulera evolutionära processer på en dator är ganska enkelt: med enbart grundläggande programmeringskunskaper kan man implementera en födelse-/döds-modell som ett datorprogram. Vid körning börjar programmet med en enda art (vid en given tidpunkt i det förflutna) och simulerar steg för steg specierings- och utrotningshändelser. Ett resultat av en sådan simulering är en så kallad fullständig fylogeni (eller ett fullständigt evolutionärt träd). Med fullständig menas här att trädet också inkluderar utdöda arter. Det är lätt att konvertera detta träd till ett ”rekonstruerat” träd som bara visar utvecklingen för de befintliga arterna och deras förfäder – vi behöver bara ta bort de delar av den fullständiga fylogenin som endast är kopplade till de utdöda arterna.

Det omvända problemet är mer intressant och betydelsefullt för fylogenetiker: givet en fylogeni rekonstruerad från tillgängliga data om nu levande arter (t.ex. genomiska sekvenser) och en födelse-/döds-modell, vad kan man säga

om parametrarna för denna modell, t.ex. specierings- och utrotningshastigheter? För att svara på sådana frågor när forskare har föreslagit nya födelse-/dödsmodeller krävs det idag hjälp av externa experter. De hjälper till med att hitta och implementera skräddarsydda inferensalgoritmer, samt att implementera dem i befintliga eller nya programvarupaket som vanligtvis är ganska komplexa.

Vårt arbete löser detta problem på ett helt nytt sätt genom att använda så kallade *probabilistiska programmeringsspråk*. Dessa språk har inbyggt stöd för slumpmässiga variabler, sannolikhetsfördelningar och olika probabilistiska konstruktioner (som t.ex. sampling och observationer). En mycket viktig del är integrerad automatisk inferens: man behöver inte hitta och implementera någon inferensalgoritm (vilket vanligtvis är svårt) utan snarare beskriver man den generativa modellen i form av ett program som kan simulera denna modell (vilket vanligtvis är enkelt), lägger till informationen om vad som observerades, och kör programmet för att genomföra inferens automatiskt.

I denna doktorsavhandling visade vi hur man kan koda olika fylogenetiska födelse-/döds-modeller som enkla program i probabilistiska programmeringsspråk och bevisat att det går att få användbara resultat inom rimlig tid. Vår metod är baserad på augmentation: programmet går igenom den rekonstruerade fylogenin, gren för gren, och lägger till obemärkta händelser och simulerar utvecklingen av utdöda arter. På detta sätt kan man uppskatta (fördelningar av) modellparametrarna.

Med automatisk inferens baserad på sekventiell Monte Carlo tillåter vårt tillvägagångssätt, till skillnad från de befintliga sätten, också att uppskatta marginell sannolikhet (sannolikhet att modellen kunde leda till just den aktuella rekonstruerade fylogenin) utan bias och möjliggör därmed användning av hierarkiska modeller samt jämförelse av olika modeller. Våra bidrag till de automatiska inferensmetoderna (t.ex. fördröjd sampling och det utvidgade *alive particle filter*) är inte relevanta endast för fylogenetik utan även för probabilistisk programmering generellt.

Bibliography

[1] C. Andrieu and G. O. Roberts. The pseudo-marginal approach for efficient Monte Carlo computations. *Annals of Statistics*, 37(2):697–725, 2009.

[2] C. Andrieu, A. Doucet, and V. B. Tadic. On-line parameter estimation in general state-space models. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 332–337, Seville, Spain, 2005.

[3] C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B*, 72(3): 269–342, 2010.

[4] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302, Austin, Texas, USA, 1989.

[5] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019.

[6] R. Bouckaert, J. Heled, D. Kühnert, T. Vaughan, C.-H. Wu, D. Xie, M. A. Suchard, A. Rambaut, and A. J. Drummond. BEAST 2: A software platform for Bayesian evolutionary analysis. *PLOS Computational Biology*, 10(4):e1003537, 2014.

[7] R. E. Caflisch et al. Monte Carlo and quasi-Monte Carlo methods. *Acta Numerica*, 1998:1–49, 1998.

[8] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

[9] O. F. Cook. Factors of species-formation. *Science*, 23(587):506–507, 1906.

[10] M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, Phoenix, Arizona, USA, 2019.

[11] C. Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859.

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

[13] P. Del Moral, A. Doucet, and A. Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[14] A. Doucet, N. De Freitas, and N. Gordon. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo Methods in Practice*, pages 3–14. Springer, 2001.

[15] A. J. Drummond and A. Rambaut. BEAST: Bayesian evolutionary analysis by sampling trees. *BMC Evolutionary Biology*, 7(1):214, 2007.

[16] W. Feller. Die Grundlagen der Volterraschen Theorie des Kampfes ums Dasein in wahrscheinlichkeitstheoretischer Behandlung. *Acta Biotheoretica*, 5(1):11–40, May 1939.

[17] R. G. FitzJohn. Quantitative traits and diversification. *Systematic Biology*, 59(6):619–633, 2010.

[18] R. G. FitzJohn. Diversitree: comparative phylogenetic analyses of diversification in R. *Methods in Ecology and Evolution*, 3(6):1084–1092, 2012.

[19] H. Ge, K. Xu, and Z. Ghahramani. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, Playa Blanca, Lanzarote, Spain, 2018.

[20] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov 1984.

[21] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177, 1994.

[22] E. E. Goldberg, L. T. Lancaster, and R. H. Ree. Phylogenetic inference of reciprocal effects between geographic range evolution and diversification. *Systematic Biology*, 60(4):451–465, 2011.

[23] N. D. Goodman and A. Stuhlmüller. The design and implementation of probabilistic programming languages. `http://dippl.org`, 2014. Accessed: 2021-2-1.

[24] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 220–229, Arlington, Virginia, USA, 2008.

[25] J. M. Hammersley and D. C. Handscomb. Monte Carlo methods. *John Wiley & Sons, New York, USA*, 1964.

[26] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[27] S. Höhna, W. A. Freyman, Z. Nolen, J. P. Huelsenbeck, M. R. May, and B. R. Moore. A Bayesian approach for estimating branch-specific speciation and extinction rates. Preprint at `https://www.biorxiv.org/content/10.1101/555805v1`, 2019.

[28] J. P. Huelsenbeck and F. Ronquist. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics*, 17(8):754–755, 2001.

[29] S. Höhna, M. J. Landis, T. A. Heath, B. Boussau, N. Lartillot, B. R. Moore, J. P. Huelsenbeck, and F. Ronquist. RevBayes: Bayesian phylogenetic inference using graphical models and an interactive model-specification language. *Systematic Biology*, 65(4):726–736, 2016.

[30] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960.

[31] D. G. Kendall. On the generalized "birth-and-death" process. *The Annals of Mathematical Statistics*, 19(1):1–15, 1948.

[32] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[33] T. A. Le, A. G. Baydin, and F. Wood. Inference compilation and universal probabilistic programming. In *International Conference on Artificial Intelligence and Statistics*, pages 1338–1348, Fort Lauderdale, Florida, USA, 2017.

[34] P. W. Lewis and G. S. Shedler. Simulation of nonhomogeneous poisson processes by thinning. *Naval Research Logistics Quarterly*, 26(3):403–413, 1979.

[35] W. P. Maddison, P. E. Midford, and S. P. Otto. Estimating a binary character's effect on speciation and extinction. *Systematic Biology*, 56(5): 701–710, 2007.

[36] O. Maliet, F. Hartig, and H. Morlon. A model with many small shifts for estimating species-specific diversification rates. *Nature Ecology & Evolution*, 3(7):1086–1092, Jul 2019.

[37] V. Mansinghka, R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves. BayesDB: A probabilistic programming system for querying the probable implications of data. Preprint at `https://arxiv.org/abs/1512.05006`, 2015.

[38] V. K. Mansinghka, D. Selsam, and Y. N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. Preprint at `https://arxiv.org/abs/1404.0099`, 2014.

[39] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[40] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1352–1359, San Francisco, CA, USA, 2005.

[41] D. Moen and H. Morlon. Why does diversification slow down? *Trends in Ecology & Evolution*, 29(4):190–197, 2014.

[42] B. R. Moore, S. Höhna, M. R. May, B. Rannala, and J. P. Huelsenbeck. Critically evaluating the theory and performance of Bayesian analysis of macroevolutionary mixtures. *Proceedings of the National Academy of Sciences*, 113(34):9569–9574, 2016.

[43] H. Morlon, E. Lewitus, F. L. Condamine, M. Manceau, J. Clavel, and J. Drury. RPANDA: an R package for macroevolutionary analyses on phylogenetic trees. *Methods in Ecology and Evolution*, 7(5):589–597, 2016.

[44] L. M. Murray. Bayesian state-space modelling on high-performance hardware using LibBi. *Journal of Statistical Software*, 67(10):1–36, 2015.

[45] L. M. Murray. Lazy object copy as a platform for population-based probabilistic programming. Preprint at `https://arxiv.org/abs/2001.05293`, 2020.

[46] L. M. Murray and T. B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[47] L. M. Murray, A. Lee, and P. E. Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

[48] C. Naesseth, S. Linderman, R. Ranganath, and D. Blei. Variational sequential Monte Carlo. In *International Conference on Artificial Intelligence and Statistics*, pages 968–977, Playa Blanca, Lanzarote, Spain, 2018.

[49] P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming*, pages 62–79, Kochi, Japan, 2016. Springer.

[50] R. M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11):2, 2011.

[51] B. Paige and F. Wood. A compilation target for probabilistic programming languages. In *International Conference on Machine Learning*, pages 1935–1943, Bejing, China, 2014.

[52] A. Pfeffer. *Practical Probabilistic Programming*. Manning, 2016.

[53] M. Plummer. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, volume 124, pages 1–10. Vienna, Austria, 2003.

[54] D. L. Rabosky. Automatic detection of key innovations, rate shifts, and diversity-dependence on phylogenetic trees. *PLoS ONE*, 9(2):e89543, 2014.

[55] T. Stadler. On incomplete sampling under birth-death models and connections to the sampling-based coalescent. *Journal of Theoretical Biology*, 261(1):58–66, Nov. 2009.

[56] T. Stadler. Mammalian phylogeny reveals recent diversification rate shifts. *Proceedings of the National Academy of Sciences*, 108(15):6187–6192, 2011.

[57] M. E. Steeman, M. B. Hebsgaard, R. E. Fordyce, S. Y. Ho, D. L. Rabosky, R. Nielsen, C. Rahbek, H. Glenner, M. V. Sørensen, and E. Willerslev. Radiation of extant cetaceans driven by restructuring of the oceans. *Systematic Biology*, 58(6):573–585, 2009.

[58] A. Todeschini, F. Caron, M. Fuentes, P. Legrand, and P. Del Moral. Biips: Software for Bayesian inference with interacting particle systems. Preprint at `https://arxiv.org/abs/1412.3779`, 2014.

[59] D. Tolpin, J.-W. van de Meent, H. Yang, and F. Wood. Design and implementation of probabilistic programming language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages*, pages 6:1–6:12, Leuven, Belgium, 2016.

[60] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. Preprint at `https://arxiv.org/abs/1610.09787`, 2016.

[61] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. Preprint at `https://arxiv.org/abs/1809.10756`, 2018.

[62] J. Von Neumann. Various techniques used in connection with random digits. *Applied Mathematic Series*, 12(36-38):5, 1951.

[63] D. Wingate and T. Weber. Automated variational inference in probabilistic programming. Preprint at `https://arxiv.org/abs/1301.1299`, 2013.

[64] G. U. Yule. A mathematical theory of evolution, based on the conclusions of Dr. JC Willis, FRS. *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, 213: 21–87, 1924.

A

Used distributions and their parameterizations

Bernoulli distribution

Notation: $k \sim \text{Bernoulli}(p)$

Parameters:

- probability $p \in [0, 1]$

Probability mass function:

$$f(k|p) = \begin{cases} p & \text{if } k = 1 \text{ (true)} \\ 1 - p & \text{if } k = 0 \text{ (false)} \end{cases}$$

Binomial distribution

Notation: $k \sim \text{Binomial}(n, p)$

Parameters:

- number of trials $n \in \mathbb{N} \cup \{0\}$
- probability of success $p \in [0, 1]$

Probability mass function:

$$f(k|n, p) = \binom{n}{k} p^k (1 - p)^{n-k} \text{ for } k \in \mathbb{N} \cup \{0\}$$

Exponential distribution

Notation: $x \sim \text{Exponential}(\lambda)$

Parameters:

- rate $\lambda > 0$

Probability density function:

$$f(x|\lambda) = \lambda e^{-\lambda x} \text{ for } x \geq 0$$

Gamma distribution

Notation: $x \sim \text{Gamma}(k, \theta)$

Parameters:

- shape $k > 0$
- scale $\theta > 0$

Probability density function:

$$f(x|k, \theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta} \text{ for } x > 0$$

Lomax distribution

Notation: $x \sim \text{Lomax}(\lambda, \alpha)$

Parameters:

- scale $\lambda > 0$
- shape $\alpha > 0$

Probability density function:

$$f(x|\lambda, \alpha) = \frac{\alpha}{\lambda} \left(1 + \frac{x}{\lambda}\right)^{-(\alpha+1)} \text{ for } x \geq 0$$

Negative binomial distribution

Notation: $r \sim \text{NegativeBinomial}(k, p)$

Parameters:

- number of successes before the experiment is stopped $k \in \mathbb{N}$
- probability of success $p \in [0, 1]$

Probability mass function:

$$f(r|k, p) = \binom{r + k - 1}{k - 1} p^k (1 - p)^r \text{ for } r \in \mathbb{N} \cup \{0\},$$

where r is the number of failures.

Normal (Gaussian) distribution

Notation: $x \sim \mathcal{N}(\mu, \sigma^2)$

Parameters:

- mean μ
- variance $\sigma^2 > 0$

Probability density function:

$$f(x|\mu, \sigma^2) = \mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Poisson distribution

Notation: $k \sim \text{Poisson}(\lambda)$

Parameters:

- rate $\lambda > 0$

Probability mass function:

$$f(k|\lambda) = \frac{\lambda^k}{k!} e^{-\lambda} \text{ for } k \in \mathbb{N} \cup \{0\}$$

Uniform distribution

Notation: $x \sim \text{Uniform}(a, b)$

Parameters:

- lower bound a
- upper bound $b > a$

Probability density function:

$$f(x|a, b) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

B

Used abbreviations

APF Alive particle filter

BAMM Bayesian analysis of macro-evolutionary mixtures (model)

BiSSE Binary state speciation and extinction (model)

BP Before present (time)

BPF Bootstrap particle filter

ClaDS Cladogenetic diversification rate shift (model)

CPS Continuation-passing style

CPU Central processing unit

CRBD Constant-rate birth-death (model)

ESS Effective sample size

GPU Graphics processing unit

LSBDS Lineage-specific birth-death-shift (model)

MCMC Markov chain Monte Carlo

MRCA Most recent common ancestor

PF-RC Particle filter with rejection control

PGM Probabilistic graphical model

PMCMC Particle Markov chain Monte Carlo

PPL Probabilistic programming language

SMC Sequential Monte Carlo

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2006

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-432409

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021