Philipp Gabler, BSc

Automatic Graph Tracking in Dynamic Probabilistic Programs via Source Transformations

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Ing. Dr. mont. Franz Pernkopf

Co-supervisor
Dipl.-Ing. Dr. Martin Trapp, BSc

Institute of Signal Processing and Speech Communication

Faculty of Electrical and Information Engineering

Graz, XXXX 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

[*] pgabler@student.tugraz.at

Abstract

This thesis presents a novel approach for the implementation of a tracking system to facilitate program analysis, based on program transformations. The approach is then applied to a specific problem in the field of probabilistic programming.

The main contribution is a general system for the extraction of rich computation graphs in the Julia programming language, based on a transformation of the intermediate representation (IR) used by the compiler. These graphs contain a slice of the whole recursive structure of any Julia program in terms of executed IR instructions. The system is flexible enough to be used for multiple purposes that require dynamic program analysis or abstract interpretation, such as automatic differentiation or dependency analysis.

The second part of the thesis describes the application of this graph tracking system to probabilistic programs written for `Turing.jl`, a probabilistic programming system implemented as an embedded language within Julia. Through this, an executed Turing model can be analyzed, and the dependency structure of involved random variables be extracted from it. Given this structure, analytical Gibbs conditionals can be calculated for a large set of models and passed to Turing's inference mechanism, where they are used in Markov-Chain Monte Carlo samplers approximating the modelled distribution.

Contents

Notation

$\mathbb{P}[\Theta \in A \mid X = x]$	Random variables and their realizations will usually be denoted by upper and lower case letters, respectively (with occasional exceptions for Greek variable names). Sets are also named by uppercase letters.
$\mathbb{E}[X], \mathbb{V}_X[f(X, Y)]$	Expectation and variance; if necessary, the variable with respect to which the moment is taken is indicated as a subscript.
$\phi(x), f_Z(x)$	Density functions are named using letters commonly used for functions, with an optional subscript indicating the random variable they belong to. Densities always come with implied base measures depending on the type of the random variable.
$p(x, y \mid z)$	The usual abuse of notation with the letter "p" standing for any density indicated by the names of the variables given to it is used when no confusion arises (in this case, $f_{X,Y\mid Z}$ is implied). A q may be used as well, mostly for proposal distributions or unnormalized densities.
$X_i \sim \mathrm{Normal}(\mu, \sigma)$	The tilde notation for describing random variables is used throughout, often without explicitely specifying dependence or independence, where understood from context. Named distributions that are not themselves random variables are spelled out in upright script.
$Y \sim q(\cdot, X_{i-1})$	The same notation is used when a random variable is specified to be sampled from a given, possibly unnormalized, density. In this context and elsewhere, the midpoint is employed to denote anonymous functions of one variable given by partial application.
$y \mapsto p(x \mid y, z)$	Anonymous functions are distinguised from function evaluation; this is crucial to differentiate between probability densities and likelihoods, for example.
$\int p(x)\,\mathrm{d}x = 1$	Integrals over the whole domain of a density or measure are written as indefinite integrals, where the usage is clear.

$[x, y, z] = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$	For consistency with Julia code, vectors (arrays of rank 1) are written in brackets, with elements separated by commas. Thereby, the form written in a row denotes a column vector; actual row vectors are written as transposed column vectors.
$\Theta^{(k)} = [\Theta_1^{(k)}, \dots, \Theta_N^{(k)}]$	Superscript indices in parentheses are used for series or sequences of variables, and subscript indices for components of multivariate variables.
$z_{-i} = [z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_N]$	Negative indices denote all components of a variable without the negated one.
$f.(x, 1) = [f(x_1, 1), \dots, f(x_N, 1)]$	Function application with a period indicates vectorized application, as in Julia code[*]: the function is applied over all elements of the input arrays individually, whereby arrays of lower rank or scalars are "broadcasted" along dimensions as necessary.
`f(x) = rand(x)`	Julia code (including identifiers mention in the text) is always typeset in typewriter font.

[*]See `https://docs.julialang.org/en/v1/manual/functions/#man-vectorized-1`

1 Introduction

This chapter gives an overview over the scope of the thesis and existing approaches in the literature. A preliminary version of this work has already been presented in Gabler et al. (2019), which forms the basis of the introduction.

Many methods in the field of machine learning work on computation graphs of programs that represent mathematical expressions. One example are forms of automatic differentiation (AD) which derive an "adjoint" expression from a expression that usually represents a loss function, to calculate its gradient (Gebremedhin; Walther 2020; Griewank; Walther 2008). Another one are message passing algorithms (Minka 2005), which use the graph as the basic data structure for the operation they perform: passing values between nodes, representing random variables that depend on each other (in fact, message passing generalizes various other methods, including AD). But also in more or less unrelated fields, such as program analysis or program transformation (cf. XXX), the same requirements might occur through the need to derive abstract graphs of program flow from a given program.

examples

abstract interpretation

There are several options how to provide the computation graph in question to an application, many of which are already established in the AD community (see Baydin et al. (2018) for a survey on AD methods). For one, graphs can be required to be written out explicitely by the user, by defining a custom input format or a library to build graphs "by hand" through an API (e.g., PyTorch (Paszke et al. 2017) or TensorFlow (Abadi et al. 2015)). Such APIs are called *operator overloading* in AD language, because they extend existing operations to additionally track the computation graph at runtime on so-called tapes or Wengert lists (Bartholomew-Biggs et al. 2000). This kind of tracking is dynamic, in the sense that a new tape is recorded for every execution. However, being implemented on a library level, it usually requires the programmer to use non-native constructs instead of language primitives, leading to cognitive overhead. Furthermore, there are additional runtime costs due to the separate interpretation of derivatives stored on the tape.

example

Alternatively, an implementation can allow the user to write out computations as a "normal" program in an existing programming language (or possibly a restricted subset of it), and use metaprogramming techniques to extract graphs from the input program. Such metaprograms, known under the name *source transformations*, can in turn operate on plain source code (cf. Tapenade (Tapenade developers 2019)), or on another, more abstracted notion used by the programming language infrastructure, like the abstract syntax tree (AST), or an intermediate representation (IR). They operate on the syntactic structure of the whole program, during or before compilation. Unlike in operator overloading, it is hence possible to inspect and exploit control structures directly. This can lead to more efficient results, compared to operator overloading, since the transformation is done only once per program and eligible for compiler optimisations. Additionally, the user is not restricted to the domain specific language provided by a libray, and can use regular language constructs, data structures, and custom functions rather freely. But in this approach, no records of the actual execution paths are constructed explicitly – purely static information is used only at compile time, and cannot be accessed for further analysis or transformation during execution.

In a variety of domains, though, the execution path of programs can drastically change at each run. Examples of this from machine learning are models with non-uniform data, such as parse trees (Socher et al. 2011) or molecular graphs (Bianucci et al. 2000), Bayesian nonparametric models (Hjort et al. 2010), or simply the occurrence of stochastic control flow in any probabilistic model. Such programs we call dynamic models. The lack of an explicit, unique graph structure makes it impossible, or at least diffult, to apply source transformation approaches on them. Operator overloading is the more direct way for supporting dynamic models, since it automatically records a new tape for each input. In fact, many state-of-the-art machine learning libraries are based on dynamic graphs using operator overloading in some form.

However, relying on operator overloading makes it impossible to take advantage of the benefits of source transformations, such as utilizing information about the control flow, integrating with optimizations at compile time, or exploiting the source model structure. The source transformation approach based on intermediate representations has recently gained popularity in machine learning.

1.1 Related Work

2 Background

This chapter provides the background for the concepts used later in chapters 3 and 4. Initially, it gives a quick overview of Baysian inference and probabilistic programming in general, necessary to understand the requirements and usual approaches of probabilistic programming systems.

Consequently, the machinery and language used to develop the graph tracking system forming the main part of the work are described. This consists firstly of a short introduction to graph tracking and source-to-source automatic differentiation, which contain many ideas and terminology that will be used later, and often provided inspiration. Secondly, the basic notions and techniques of the Julia compilation process as well as the language's metaprogramming capabilities are described, which form the basis of the implementation.

2.1 Bayesian Inference and MCMC methods

Generative modelling is an approach for modelling phenomena based on the assumption that observables can be fully described through some stochastic process. When we assume this process to belong to a specified family of processes, the estimation of the "best" process is a form of learning: if we have a good description of how observations are generated, we can make summary statements about the whole population (descriptive statistics) or predictions about new observations. When observations come in pairs of independent and dependent variables, learning the conditional model of one given the other solves a regression or classification problem.

Within a Baysian statistical framework, we assume that the family of processes used is specified by random variables related through conditional distributions with densities, which describe how the observables would be generated: some *unobserved variables* are generated from *prior distributions*, and the *observed data* are generated conditionally on the unobserved variables. The goal is to learn the *posterior distribution* of the parameters given the observations, which is a sort of "inverse" of how the problem is specified.

As an example, consider image classification: if we assume that certain percentages of an image data set picture cats and dogs, respectively, the distribution of these labels forms the prior. Given the information which kind of animal is depicted on it, an image can then be generated as a matrix of pixels based on a distribution of images conditioned on labels. The posterior distribution is then conditional distribution of the label given an image. When we have this information, we can, for example, build a Baysian classifier, by returning for a newly observed image that label which has the highest probability under the posterior.

This kind of learning is called Bayesian inference since, in the form of densities, the form of the model can be expressed using Bayes' theorem as the conditional distribution with density[1]

$$\overbrace{p(\theta \mid x)}^{\text{posterior}} = \frac{\overbrace{p(x \mid \theta)}^{\text{likelihood}} \; \overbrace{p(\theta)}^{\text{prior}}}{p(x)}, \tag{2.1}$$

where x are the observed data, and θ are the unobserved parameters. The posterior represents the distribution of the unobserved variables as a combination of the prior belief updated by what has been observed (Congdon 2006). (In practice, not all of the unobserved variables have to be model parameters we are actually interested in; these can be integrated out).

Going beyond simple applications like the classifier mentioned above, handling the posterior gets difficult, though. Simply evaluating the posterior density $\theta \mapsto p(\theta \mid x)$ at single points is not enough in a Baysian setting for usages such as prediction, parameter estimation, or evaluation of probabilities of continuous variables. The problem is that almost all of the relevant quantities depend on some sort of expectation over the posterior density, an integral of the form

$$\mathbb{E}[f(\Theta) \mid X = x] = \int f(\theta) p(\theta \mid x) \, \mathrm{d}\mu(\theta), \tag{2.2}$$

for some measurable function f (with the base measure μ depending on the type of Θ). This in turn involves calculating the normalizing marginal

$$p(x) = \int p(x, \tilde{\theta}) \, \mathrm{d}\mu(\tilde{\theta}). \tag{2.3}$$

in equation 2.1, often called the "evidence".

When the distributions involved form a sufficiently "nice" combination, e.g., a conjugate pair (see Marin; Robert 2007, chapter 2.2.2; Murphy 2012, chapter 9.2.5), the integration can be performed analytically, since the posterior density has a closed form for a certain known distribution, or at least is a known integral. In general, however, this is not tractable, not even by standard numerical integration methods, and approximations have to be made. Even for discrete variables, the applicability of simple summation is limited by combinatorial explosion.

Different techniques for posterior approximation are available: among them are distribution-based approaches for general graphical models, such as variational inference (Murphy 2012, chapter 21 and 22) and other methods generalized under the

[1] Note the abuse of notation regarding $p(\cdot)$; see page xi on notation.

framework of message passing (Minka 2005). The methods described in this thesis, however, fall into the category of Monte Carlo methods, and are based on sampling (Murphy 2012, chapter 23; Vihola 2020). Their fundamental idea is to derive, for a specified density of $\Theta \sim \pi$, a sampling procedure with a consistent estimator for expectations:

$$I^{(k)}(f) \to \mathbb{E}[f(\Theta)], \quad \text{as} \quad k \to \infty \tag{2.4}$$

in some appropriate stochastic convergence (usually convergence in probability is enough). We leave out the conditional dependency on X here for simplicity in notation, and since the data are usually fixed in inference problems.

Examples of such methods are rejection sampling, importance sampling, and particle filters. Many Monte Carlo methods are defined in a form that directly samples a sequence of individual random variables $(Y^{(k)})_{k\geq 1}$, called a *chain*, for which the estimator is given by the arithmetic mean, such that a law of large numbers (LLN) holds:

$$I^{(k)}(f) = \frac{1}{n}\sum_{i=1}^{n} f(Y^{(k)}) \to \mathbb{E}[f(\Theta)] \tag{2.5}$$

When we can sample $Y^{(k)} \sim \pi$ exactly, they are i.i.d. and the LLN holds trivially; such samplers exist, but might also be difficult to derive or not possess good enough convergence properties (especially in high dimensions). Another large class of samplers is formed by *Markov Chain Monte Carlo* (MCMC) methods, which, instead of sampling exactly from the density, define $Y^{(k)}$ via a Markov chain:

$$\begin{aligned}
&\mathbb{P}[Y^{(k+1)} \in \mathrm{d}\zeta \mid Y^{(k)} = y^{(k)}, \dots, Y^{(1)} = y^{(1)}] \\
&\quad = \mathbb{P}[Y^{(k+1)} \in \mathrm{d}\zeta \mid Y^{(k)} = y^{(k)}] \\
&\quad = K(\zeta \mid y^{(k)})\, \mathrm{d}\mu(\zeta)
\end{aligned} \tag{2.6}$$

for all $k \geq 1$. By constructing the *transition kernel*, K, in the right way, the resulting chain is ergodic with the target density π as the unique stationary distribution, i.e.,

$$\int \pi(\zeta) K(\theta \mid \zeta)\, \mathrm{d}\mu(\zeta) = \pi(\theta) \tag{2.7}$$

(which for discrete spaces is usually written in matrix form as a left eigenvalue equation on a stochastic matrix: $\pi K = \pi$). The advantage of MCMC methods is that they apply equally well to many structurally complex models, and treat densities in a uniform way, without requiring special knowledge about the specific distribution in question. I refer to Vihola (2020, chapter 6), Robert; Casella (1999), and Murphy (2012, chapters 24 and following) for an introductions to MCMC theory and practice.

Frequently, MCMC methods use variations of the *Metropolis-Hastings algorithm* (MH), which replace the general definition of the transitions kernel by two helper fuctions: a proposal distribution, given by a conditional density q that needs to be easy to sample from, and an acceptance rate α. Subsequent samples are then produced by proposing values from q given the previous element of the cahin, and incorporating them into the chain with a probability given through α (see

Algorithm 1 General scheme for the Metropolis-Hastings algorithm.

```
1. Start from an arbitrary $Y^{(1)} = y^{(1)}$ with $\pi(y^{(k)}) > 0$.
2. For each $k \geq 1$:
    1. Sample a proposal $\hat{Y}^{(k)} \sim q(Y^{(k-1)}, \cdot)$.
    2. With probability $\alpha(\hat{Y}^{(k)}, Y^{(k-1)})$, set $Y^{(k)} = \hat{Y}^{(k)}$; else, keep $Y^{(k)} = Y^{(k-1)}$.
```

algorithm 1). There exist many MH-based schemes with different properties and requirements: from the classical random-walk Metropolis algorithm with Gaussian proposals, over Reversible Jump MCMC for varying dimensions (Green 1995), to gradient-informed methods like Metropolis Adjusted Langevin and Hamiltonian Monte Carlo (HMC) (Betancourt 2018; Girolami; Calderhead 2011).

nicer algorithm formatting

Still, when we have a multi-component structure $\Theta = [\Theta_1, \dots, \Theta_N]$, a good transition kernel can be hard to find (especially manually). One way to break down the problem is to use a family of componentwise updates, given by conditional kernels q_i operating on only one component of Θ, with the others fixed:

$$\begin{aligned} \hat{Y}_{-i}^{(k)} &= Y_{-i}^{(k-1)} \\ \hat{Y}_{i}^{(k)} &\sim q_i(Y_i^{(k-1)}, \cdot \mid Y_{-i}^{(k-1)}) \end{aligned} \tag{2.8}$$

The components can be scalar or multivariate blocks, and the kernel may itself be any valid transition kernel (Vihola 2020, chapter 6.6). This allows one to freely mix different MCMC methods suitable for each variable in a problem.

This so-called "within-Gibbs" sampler bears its name because it is a generalization of the classical *Gibbs sampling* algorithm: often, the simplest available set of transition kernels is given by the conditional densities $\Theta_i \mapsto p(\Theta_i \mid \Theta_{-i}, x)$. They can directly be used as component proposals for a within-Gibbs sampler, leading to a cancelling acceptance rate of $\alpha \equiv 1$. This approach has the advantage that it is in many models it is rather easy to derive, even manually, from a given joint density; examples are extensively covered in Murphy (2012, chapter 24.2).

$$\begin{aligned} \mu_k &\overset{\text{iid}}{\sim} \text{Normal}(m, s) \quad \text{for } 1 \leq k \leq K, \\ Z_n &\overset{\text{iid}}{\sim} \text{Categorical}(K) \quad \text{for } 1 \leq n \leq N, \\ X_n &\overset{\text{iid}}{\sim} \text{Normal}(\mu_{Z_n}, \sigma) \quad \text{for } 1 \leq n \leq N \end{aligned} \tag{2.9}$$

2.2 Probabilistic Programming

Probabilistic programming is a means of describing generative models through the syntax of a programming language. It makes sense to consider probabilistic programs not only as syntactic sugar for denoting a function that calculates a joint probability density over some set of variables, but as structured objects in their own right: they open up possibilities that "black box" density functions cannot automatically provide. In more concise terms from van de Meent et al. (2018):

> Probabilistic programming is largely about designing languages, interpreters, and compilers that translate inference problems denoted in programming language syntax into formal mathematical objects that allow and accommodate generic probabilistic inference, particularly Bayesian inference and conditioning.

A probabilistic program differs from a regular program with stochastic values through the possibility of being conditioned on: some of the internal variables can be fixed to observed values. As such, the program denotes on the one hand a joint distribution, that can be *forward sampled* from by simply running the program top to bottom and calling a pseudo-random functions. But at the same time, it also represents a conditional distribution, given as an unnormalized conditional density, which together with an inference algorithm can also be sampled from. Consider the model (2.9) from above: its mathematical description might be translated into a program in `Turing.jl` syntax as

```julia
@model function normal_mixture(x, K, m, s, σ)
    N = length(x)

    μ = Vector{Float64}(undef, K)
    for k = 1:K
        μ[k] ~ Normal(m, s)
    end

    z = Vector{Int}(undef, N)
    for n = 1:N
        z[n] ~ Categorical(K)
    end

    for n = 1:N
        x[n] ~ Normal(μ[z[n]], σ)
    end

    return x
end
```

We can then *query* the model in several ways:

```julia
julia> m = normal_mixture(x_observations, K, m, s, σ);
julia> forward = sample(m, Prior(), 10);
julia> chain = sample(m, MH(), 1000);
```

The value of `forward` will be an dataframe-like object containing 10 values for each variable sampled from the forward (i.e., joint) distribution, matching the size of `x_observations`. Similarly, `chain` will contain a length 1000 sample from a Markov chain targetting the posterior, created using the MH algorithm. If we were to write these two functionalities manually, in idiomatic Julia, we would end up with at least the following two functions:

```
function normal_mixture_sampler(N, K, m, s, σ)
    μ = rand(Normal(m, s), K)
    z = rand(Categorical(K), N)
    x = rand.(Normal.(μ[z], s))
    return μ, z, x
end

function normal_mixture_logpdf(μ, z, x, K, m, s, σ)
    N = length(x)
    ℓ = 0.0
    ℓ += sum(logpdf(Normal(m, s), μ[k]) for k = 1:K)
    ℓ += sum(logpdf(Categorical(K), z[n]) for n = 1:N)
    ℓ += sum(logpdf(Normal(μ[z[n]]), x[n]) for n = 1:N)
    return ℓ
end
```

And still, with these, there would be no flexible interface for sampling algorithms to automatically detect all latent and observed variables, put them all into a dataframe with their names, etc.

While probabilistic programming languages (PPLs) are often implemented as separate, domain-specific languages (DSLs), they can also be embedded into "host" programming languages with sufficient syntactic flexibility. The latter is advantageous if one wants to use regular general-purpose programming constructs or interact with other functionalities of the host language.

There are a variety of further reasons why one would rather describe an inference problem in terms of a program than in more "mathematical" form, like a graph or likelihood function. In a good DSL, models will read as close to textbook model specifications as possible, while allowing to use the host language to express, for example:

- Recursive relationships
- Usage of imperative constructs, such as loops, or mutable intermediate computations for efficiency
- Manual manipulations, e.g. for memoization, scaling (-Inf), or preliminary termination
- Distributions over complex custom data structures, e.g. trees
- Inference involving complex transformations from other domains, for which implementations already exist, e.g. neural networks or differential equation solvers
- Inference that integrates calls to very complex external systems, e.g. simulators or renderers

references for examples

In this sense, a probabilistic programming syntax defines a common format for model description, more general.

See van de Meent et al. (2018) for a general introduction into the implementation of PPLs. Goodman; Stuhlmüller (2014) gives an in-depth overview of the implementation and usage of one specific, continuation-based implementation called WebPPL.

2.3 Compilation and Metaprogramming in Julia

Singer (2018)

2.4 Computation Graphs and Automatic Differentiation

3 Implementation of Dynamic Graph Tracking in Julia

3.1 Automatic Graph Tracking and Extended Wengert Lists

3.2 Evaluation

4 Graph Tracking in Probabilistic Models

4.1 Dependency Analysis in Dynamic Models

4.2 JAGS-Style Automatic Calculation of Gibbs Conditionals

4.3 Evaluation

5 Discussion

The history of this project forms a large arc from a general problem in `Turing.jl`, over a digression into compiler technology, back the the implementation of a proof of concept in the form of a very specific inference method. As we have seen, two separate pieces of software have emerged from it: `IRTracker.jl` and `AutoGibbs.jl`. The underlying issue – that `Turing.jl` lacks a structural representation of models – is not at all resolved by them.

The real difficulty is that dynamic models cannot be satisfactorily handled through static snapshots (cf. Venture).

(`Cassette.jl` is a package very similar to `IRTools.jl`)

> Using Cassette on code you wrote is a bit like shooting youself with a experimental mind control weapon, to force your hands to move like you knew how to fly a helicopter. Even if it works, you still had to learn to fly the helicopter in order to program the mind-control weapon to force yourself to act like you knew how to fly a helicopter.[1]

ask for permission

In summary, I

5.1 Future Work

Bla.[2]

[1]Lyndon White, private communication on `julialang.slack.com`.

[2]These ideas have already been informally described by me online at `https://github.com/phipsgabler/probability-ir`.

Bibliography

Abadi, M. et al. (2015). “TensorFlow: Large-scale machine learning on heterogeneous systems”. Preliminary White Paper. Preliminary White Paper. URL: https://www.tensorflow.org/ (visited on 2020-07-29).

Bartholomew-Biggs, M. et al. (2000). “Automatic differentiation of algorithms”. In: *Journal of Computational and Applied Mathematics*. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations 124.1, pp. 171–190. DOI: 10.1016/S0377-0427(00)00422-2. URL: http://www.sciencedirect.com/science/article/pii/S0377042700004222 (visited on 2019-04-22).

Baydin, A. G. et al. (2018). “Automatic differentiation in machine learning: a survey”. In: *Journal of Machine Learning Research* 18.153, pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

Betancourt, M. (2018). *A Conceptual Introduction to Hamiltonian Monte Carlo*. arXiv: 1701.02434 [stat]. URL: http://arxiv.org/abs/1701.02434 (visited on 2020-10-17).

Bianucci, A. M. et al. (2000). “Application of Cascade Correlation Networks for Structures to Chemistry”. In: *Applied Intelligence* 12.1, pp. 117–147. DOI: 10.1023/A:1008368105614.

Congdon, P. (2006). *Bayesian statistical modelling*. 2nd ed. Wiley Series in Probability and Statistics. Chichester, England ; Hoboken, NJ: John Wiley & Sons. 573 pp.

Gabler, P. et al. (2019). “Graph Tracking in Dynamic Probabilistic Programs via Source Transformations”. In: 2nd Symposium on Advances in Approximate Bayesian Inference. Vancouver. URL: https://openreview.net/forum?id=r1eAFknEKr (visited on 2020-07-07).

Gebremedhin, A. H.; A. Walther (2020). “An introduction to algorithmic differentiation”. In: *WIREs Data Mining and Knowledge Discovery* 10.1, e1334. DOI: 10.1002/widm.1334. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1334 (visited on 2020-07-29).

Girolami, M.; B. Calderhead (2011). “Riemann manifold Langevin and Hamiltonian Monte Carlo methods”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 73.2, pp. 123–214. DOI: 10.1111/j.1467-9868.2010.00765.x. URL:

https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9868.2010.00765.x (visited on 2020-10-17).

Goodman, N. D.; A. Stuhlmüller (2014). *The Design and Implementation of Probabilistic Programming Languages.* URL: http://dippl.org (visited on 2019-10-15).

Green, P. J. (1995). "Reversible jump Markov chain Monte Carlo computation and Bayesian model determination". In: *Biometrika* 82.4, pp. 711–732. DOI: 10.1093/biomet/82.4.711. URL: https://academic.oup.com/biomet/article/82/4/711/252058 (visited on 2020-10-17).

Griewank, A.; A. Walther (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation.* 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics. 438 pp.

Hjort, N. L. et al. (2010). *Bayesian nonparametrics.* Cambridge Series in Statistical and Probabilistic Mathematics 28. Cambridge: Cambridge University Press.

Marin, J.-M.; C. P. Robert (2007). *Bayesian core: a practical approach to computational Bayesian statistics.* Springer Texts in Statistics. New York: Springer. 255 pp.

Minka, T. (2005). *Divergence Measures and Message Passing.* Technical Report MSR-TR-2005-173. Microsoft Research. URL: https://www.microsoft.com/en-us/research/publication/divergence-measures-and-message-passing/ (visited on 2019-10-09).

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective.* Adaptive Computation and Machine Learning Series. Cambridge, MA: MIT Press. 1067 pp.

Paszke, A. et al. (2017). "Automatic differentiation in PyTorch". In: NIPS 2017 Workshop Autodiff.

Robert, C. P.; G. Casella (1999). *Monte Carlo Statistical Methods.* New York: Springer.

Singer, J. (2018). "Introduction". In: *Single Static Assignment Book.* URL: http://ssabook.gforge.inria.fr/latest/book-full.pdf (visited on 2020-07-30).

Socher, R. et al. (2011). "Parsing Natural Scenes and Natural Language with Recursive Neural Networks". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning.* ICML'11. USA: Omnipress, pp. 129–136. URL: http://dl.acm.org/citation.cfm?id=3104482.3104499.

Tapenade developers (2019). *The Tapenade A.D. engine.* URL: https://www-sop.inria.fr/tropics/tapenade.html (visited on 2019-10-09).

Van de Meent, J.-W. et al. (2018). *An Introduction to Probabilistic Programming.* arXiv: 1809.10756 [cs, stat]. URL: http://arxiv.org/abs/1809.10756 (visited on 2019-03-08).

Vihola, M. (2020). *Lectures on stochastic simulation.* University of Jyväskylä. URL: http://users.jyu.fi/~mvihola/stochsim/.

List of Algorithms

Colophon

This document was typeset using the pdfLaTeX typesetting system, with the memoir document class. The body text is set in 11 pt Linux Libertine, enhanced by the microtype package. Other fonts include Biolinum and Inconsolata.

The document source has been written in Emacs with AUCTeX mode, using TeXworks as PDF viewer.